Rapid Development of An Assembler Using Python

Miki Tebeka
mtebeka@qualcomm.com

About Me

- Software Process Engineer in Qualcomm Israel
- Started using Python around 1998
- Use Python wherever I can
  - Currently around 90%+ of my code is in Python
- Written from small scripts to a linker and a source level GUI debugger
- Little activity in Python + OSS development
  - Also wxPython, PLY, ...

Background I

- It all started from Conway's Law:

  In every organization there will always be one person who knows what is going on. That person must be fired.

- Luckily for me, I wasn't that person
- However I found out that there is a team writing code for a home grown micro processor in *machine code*
- Promised to deliver them an assembler in two days
  - Only way my boss would let me do it

Background II

- Did manage to pull it through
  - However I cheated :)
- This talk will teach you how to cheat as well

Main Idea

- Lexer?
  - We don't need no stinkin' lexer
- Parser?
  - We don't need no stinkin' parser
- The Python interpreter will do all the parsing for us
  - Users actually write Python code
  - We'll `execfile` to execute the code

User Code Example

```
    MEM1 = 0x200
    add(r0, r2, r3)
    sub(r2, r4, r4)
    load(r2, MEM1)
label('L1')
    move(r2, r7)
    jmp(L1)
```

The Big Picture

- Each command is composed of four instruction code bits and twelve data bits
- Labels are just location in memory
- We will use inheritance for similar commands
- Set execution environment before calling `execfile`
- All commands will be stored in a list called `PROGRAM`

Main Class

```
class ASM:
    '''Base ASM instruction'''
    def __init__(self):
        self.file, self.line = here()
        PROGRAM.append(self)

    def genbits(self):
        '''Generate bits, 'code' and '_genbits'
           will be defined in each derived class
        '''
        return (self.code << INST_SHIFT) |
                self._genbits()
```

ALU Operation

```
class ALU3(ASM):
    '''ALU instruction with 3 operands'''
    def __init__(self, src1, src2, dest):
        ASM.__init__(self)
        self.src1 = src1
        self.src2 = src2
        self.dest = dest

    def _genbits(self):
        return (self.src1 << SLOT1_SHIFT) | \
               (self.src2 << SLOT2_SHIFT)  | \
               (self.dest << SLOT3_SHIFT)
```

Finally A “real” Instruction

```
class add(ALU3):
    '''`add' instruction'''
    code = 0

class sub(ALU3):
    '''`sub' instruction'''
    code = 1
```

Handling Labels

```
def label(name):
    '''Setting a label'''
    ENV[name] = len(PROGRAM)
```

Setting Up the Environment

```
# Add registers
for i in range(8):
    ENV["r%d" % i] = i

# Add operators
for op in (add, sub, move, load, store, label,
           jmp):
    ENV[op.__name__] = op
```

Parsing

```
execfile(infile, ENV, {})
```

Generating Output (binary)

```
a = array("H") # Unsigned short array
for cmd in PROGRAM:
    a.append(cmd.genbits())
open(outfile, "wb").write(a.tostring())
```

Debug Information

- Use Python's Exception mechanism to catch errors
- If we get a `SyntaxError` we can use `e.filename` and `e.lineno`
- For other exception we need to work a bit harder
- During coding we store line information in each instruction using `inspect` module
- Debug file is "`filename:line`" for each address

Summary – The Good

- Can spit out an assembler very fast
- Supported assembler has a very strong macro system
  - All of Python
- Cross platform
  - Check out for that byte order though
- Easy to extend
  - Took few hours to implement new commands in version 0.2

Summary – The Bad

- Users find syntax unusual
- Only Python syntax is supported
- Labels are not “Natural”
  - You define it as string but use it as a variable
- Code can not be divided to modules
  - Can't separate compilation and linkage
- Code is position dependent

Resources

- Article in UnixReview
  - http://tinyurl.com/d62f3
- `inspect` module
  - http://docs.python.org/lib/module-inspect.html
- `execfile`
  - http://www.python.org/doc/2.4.2/lib/built-in-funcs.html

Questions?