A Plan 9 C Compiler for RISC-V

Richard Miller

r.miller@acm.org

Plan 9 C compiler

- written by Ken Thompson for Plan 9 OS
- used for Inferno OS kernel and limbo VM
- used to bootstrap first releases of Go
- runs on Plan 9, Unix family, Windows
- language is C89, most of C99, small extensions

Why Plan 9 C?

- light weight
- predictable
- highly portable

Light weight

- source code is kilobytes, not gigabytes
- architecture independent part:
  21 files: 14,000 lines
- architecture specific part (*eg* ARM):
  11 files: 7,500 lines
- compiles itself on a Raspberry Pi in 3.7 sec

Predictable

- example: timing delay loop (for gcc):

```
for (int i = 0; i < 1000000; i++)
  asm volatile ("" ::: "memory");
```

- same thing for Plan 9 compiler:

```
for (int i = 0; i < 1000000; i++);
```

“Plan 9 C implements C by attempting to follow the programmer’s instructions, which is surprisingly useful in systems programming.

The big fat compilers work hard to find grounds to interpret those instructions as ‘undefined behaviour’.”

- Charles Forsyth, *9fans mailing list*

Highly portable

Plan 9 2nd edition had compilers for:

3210 386 68020 960 mips sparc

By the 4th edition, also included were:

29000 68000 alpha amd64 arm power

Community contributions:

arm64 sparc64 power64 nios2 ...   et al

“The compilers are relatively portable, requiring but a couple of weeks' work to produce a compiler for a different computer.”

- Ken Thompson, *Plan 9 Programmer’s Manual*

Re-targeting to RISC-V

complete toolchain consists of:

- C compiler
- linker
- assembler
- libc and other libraries
- object code utilities (ar, nm, size, prof, strip)
- debuggers (db, acid)

1 - disassembler

- part of libmach (object code handling library)

```
das(Map *map, uvlong pc, char *buf, int n)
```

- translation of binary instructions to asm text
- requires thorough study of ISA specification
    - good way to learn machine characteristics
- can be used to debug code generation later

2 - complete libmach

- a few functions to handle machine code
- applies to object files and running processes
  - parse headers
  - insert breakpoints
  - trace back through call stack
  - read and write registers
- encapsulates all machine dependencies
- utilities (nm, ar...) and debuggers are completely portable: one copy handles all architectures

3 - assembler

- Plan 9 asm syntax is similar for all architectures
  - but different from the vendors' assemblers
- output is a binary file of abstract object code
  - slightly higher level than machine code
- linker will translate each object instruction to one or more actual machine instructions
- compiler output is also abstract object code (with optional assembly listing)

simple example:

assembly / object code (same for most ISAs)

```
MOVB R1, N(R2)
```

RISC-V instruction (if N is small)

```
sb   x1, N(x2)
```

RISC-V instructions (if N is large)

```
lui %hi(N), x3
add x3, x3, x2
sb   x1, %lo(N)(x3)
```

- note: if N is an external symbol defined in another source file, only the linker has enough information to select the best instruction sequence
- the “big fat compilers” can’t do this because instruction selection is done at compile time

- assembler syntax is defined by a yacc grammar
- easily adapted from another, similar ISA (mips)
- most of the work is choosing the set of abstract opcodes: balance between needs of
  - C compiler (code generation) and
  - linker (instruction selection)

4 - linker

- separate linker exists for each architecture
- much code is common for all versions (*eg* symbol table handling, removing redundant branches and dead code)
- instruction selection is driven by a table
    - indexed by opcode and types of operands
- must create the table, write routines to translate each object opcode to machine instruction(s)
- check: assemble and link code, disassemble binary output with debugger, should match original source

5 - C compiler

- only need to look at the 11 source files with architecture dependent functions
- generating abstract object code instead of actual machine instructions means less variation between compilers
- start with similar ISA and adapt

6 - libraries

- a small set of assembly routines are needed for functions which can’t be expressed in C
    - *eg* `setjmp/longjmp, tas`
    - 64-bit add/subtract with carry
- some other functions can begin as portable C, rewritten in assembly as needed for efficiency
    - *eg* `memcpy, strcmp`
    - other 64-bit arithmetic and conversion
- most of libc and other library source is machine independent

Status (Sept 2018)

- code generation for RV32IMA is complete
- after ~2 weeks' work, compiler executing on a PC host can cross-compile itself
- result executing on picorv32 (on FPGA) can compile individual functions – insufficient RAM to self-compile entire compiler
- further testing can be done by executing on a RISC-V emulator, with more available RAM