A Plan 9 C Compiler for RISC-V RV32GC and RV64GC

Richard Miller

r.miller@acm.org

19 October 2020

Plan 9 C compiler

- written by Ken Thompson for Plan 9 OS
- used for Inferno Operating System
    - kernel and limbo VM built with Plan 9 C
- used to bootstrap the Go language
    - releases up to go1.4 included Plan 9 C compilers, to build Go compiler and runtime
- useful for embedded “bare metal” programming
    - small `lib9` runtime library: strings, formatted printing, qsort, synchronisation, ...

Plan 9 C language

- originally ANSI standard C89 (with some small extensions)
- some of C99 added (`long long`, `//` comments, compound literals, mixed code and declarations)
- no `inline` functions or inline assembly code
- keywords `register`, `const` and `volatile` are ignored (except for `extern register`)
    - all nonlocal data is treated as volatile, which simplifies embedded and OS code

C is not a high level language

- C was created in an era when normal practice was to write operating systems, compilers and runtime libraries in assembly language, for a specific computer architecture.
- (Re)writing UNIX in C had the effect of making it largely portable, while keeping the language close to the hardware model so efficiency of code would be clear by inspection.
- The language should make it easy for programmers to optimise their code, not try to do it for them.

Example

- a timing delay loop, observed in an embedded program (trickery prevents gcc from deleting it all):

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

- with the Plan 9 compiler it’s simpler:

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

A C compiler need not be huge

- source code is kilobytes, not gigabytes
- ARM version compiles itself on a Raspberry Pi 4 in 1.8 seconds

Compiler source lines – machine independent part

```
  303 acid.c           1561 lex.c
   89 bits.c            686 lexbody
  782 cc.h                3 mac.c
 1183 cc.y              856 macbody
 1462 com.c               8 omachcap.c
  619 com64.c           591 pgen.c
   79 compat            268 pickle.c
   47 compat.c          199 pswt.c
 1636 dcl.c             606 scon.c
  494 dpchk.c          2032 sub.c
  400 funct.c         13904 total
```

Compiler source lines – RISC-V specific part

```
1207 cgen.c
 117 enam.c
 336 gc.h
 220 i.out.h
 230 list.c
   9 machcap.c
 608 mul.c
 706 peep.c
1160 reg.c
 240 sgen.c
 612 swt.c
1473 txt.c
6918 total
```

RISC-V linker source lines

```
 966 asm.c
  56 compat.c
 269 compress.c
 360 l.h
 252 list.c
 338 noop.c
1510 obj.c
 194 optab.c
 589 pass.c
 525 span.c
5059 total
```

RISC-V assembler source lines

```
 182 a.h
 505 a.y
 641 lex.c
1328 total
```

Plan 9 began as a networked OS for multiple architectures

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 continue:

arm64 sparc64 power64 nios2 ...   et al

Machines of different types run from the same server root filesystem

- `$cputype` : host machine type (set at login)
- `$objtype` : target machine type for compilers (defaults to be the same as `$cputype`)
- At login, directories with appropriate executable files are bound into the local view of `/bin` :

```
bind /$cputype/bin /bin
bind -a $home/$cputype/bin /bin
```

along with shell scripts (machine independent)

```
bind -a /rc/bin /bin
```

Every Plan 9 compiler is a cross-compiler

- The normal case is to build for multiple targets in the same source directory (perhaps even simultaneously)
- To name every C compiler `cc` and every object code file `something.o` would be confusing
- Plan 9 naming conventions help to keep track of different architectures

Architecture names, tools and suffixes

	compiler	assembler	linker	object	binary
arm	5c	5a	5l	*.5	5.out
386	8c	8a	8l	*.8	8.out
amd64	6c	6a	6l	*.6	6.out
power	qc	qa	ql	*.q	q.out
mips	vc	va	vl	*.v	v.out
riscv	ic	ia	il	*.i	i.out
riscv64	jc	ja	jl	*.j	j.out

... and so on

Using the tools

To compile `prog.c` for ARM and RISCV:

```
5c prog.c && 5l prog.5
ic prog.c && il prog.i
```

To install the resulting binaries:

```
mv 5.out $home/bin/arm/prog
mv i.out $home/bin/riscv/prog
```

Mkfiles abstract the details

- In practice, the compiler and linker are not invoked directly, but by using `mk` (the Plan 9 equivalent of `make`) which uses simple rules in a `mkfile` to select tools for the right architecture:
- To compile, link and install for current $cputype

```
mk prog.install
```

- To compile, link and install for `riscv64`

```
objtype=riscv64 mk prog.install
```

For more information

- *How to Use the Plan 9 C Compiler*, Rob Pike

  https://plan9.io/sys/doc/comp.pdf

- *A Manual for the Plan 9 Assembler*, Rob Pike

  https://9p.io/sys/doc/asm.pdf

- *Maintaining Files on Plan 9 with mk*, Andrew Hume and Bob Flandrena

  https://9p.io/sys/doc/mk.pdf

Re-targeting the tools to RISC-V

- The complete compilation toolchain consists of:
    - C compiler
    - linker
    - assembler
    - libc and other libraries
    - object code utilities (ar, nm, size, prof, strip)
    - debuggers (db, acid)

1 – writing a disassembler function

- part of `libmach` (object code handling library)

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

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

2 – and a few other libmach functions

- low level functions to handle machine code, either within object files, or on the memory of running processes on Plan 9
  - parse headers
  - insert breakpoints
  - trace back through the call stack
  - read and write machine registers
- using libmach to encapsulate machine dependencies makes all utilities and debuggers completely portable: one program handles all architectures

3 – creating an 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 produces the same abstract object code format as the assembler

A simple example:

assembly / object code (same for arm, 386 etc)

```
MOVB R10, label(SB)
```

machine instruction (if `label` is close to the *static base* address SB)

```
sb   x10, N(x3)
```

machine instructions (if `label` is far from SB)

```
lui %hi(N), x4
add x4, x4, x3
sb   x10, %lo(N)(x4)
```

- note: the final address of `label` is only known at link time (it may even be defined in another C source file)
- only the linker has enough information to select the best instruction sequence
- in some other linkers, this is called *relaxation*
- post-compile relaxation requires the linker to decode machine instructions and recognise sequences that can be rewritten
- Plan 9 compiler makes this simpler by passing higher level abstract object code to the linker

- assembler syntax is defined by a yacc grammar
- easily adapted from the assembler for another, similar ISA (in this case, 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 – retargeting the linker

- a 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 opcode/operand pattern into machine code instruction(s)
- check: assemble and link code, disassemble binary output with debugger, should match the original source

5 – retargeting the C compiler

- only need to look at the 12 source files with architecture dependent functions
- generating abstract object code instead of actual machine instructions means less variation between compilers
- start with similar ISA (MIPS) and adapt

6 – runtime 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 for RV32
- some other functions can begin as portable C, rewritten in assembly as needed for efficiency
  - *eg* `memcpy, strcmp`
  - other 64-bit arithmetic and conversions
- most of lb9 / libc and other library source is machine independent

7 - testing

- initial test platform: Claire Wolf’s PicoRV32 on an ICE40 FPGA (myStorm BlackIce board)
- compiled RISC-V binaries are run on bare metal, using Russ Cox’s standalone `lib9pclient` to connect to a 9p server via serial port

1. compile a minimal hello.c program, confirm that it runs on PicoRV32
2. compile ic using /bin/arm/ic, run the result on PicoRV32 to comfirm that it compiles hello.c
3. compile ic source files using /bin/riscv/ic on PicoRV32, confirm the object code matches

- the test hardware didn’t have enough RAM to run the Plan 9 linker, so unable to run a full compiler bootstrap
- for larger scale testing of RISC-V binaries, switched to using Fabrice Bellard’s `tinyemu` RISC-V emulator (ported to run on Plan 9 host)

8 – adding instruction set extensions

- initial toolchain supported only base RV32IM (sufficient to compile the compiler through itself)
- added other extensions one by one, repeating same development steps (disassembler, assembler, linker, compiler, testing)
- floating point (single and double), compressed instructions, 64-bit instructions
- as of October 2020, toolchain supports RV32GC and RV64GC (extensions IMAFDC)