

Cargo-culting an SBCL backend

Charles Zhang

December 20, 2019

Outline

- 1 Introduction
- 2 Writing a new backend
- 3 Current status
- 4 Future backend work

Who am I?

Who am I?

- Third year undergraduate student

Who am I?

- Third year undergraduate student
- Linguistics & Math

Who am I?

- Third year undergraduate student
- Linguistics & Math
- Research interest: Historical Linguistics

CLISP GSoC project (2018)

- Wrote an SSA-based optimizing compiler targeting the CLISP VM
 - DCE, Loop invariant hoisting, SCC propagation, inlining etc.

CLISP GSoC project (2018)

- Wrote an SSA-based optimizing compiler targeting the CLISP VM
 - DCE, Loop invariant hoisting, SCC propagation, inlining etc.

SBCL

- Brought up RISC-V backend (Spring 2019)
- Entered the tail-end of PPC64 porting work by adding native threads (once Robert Smith kindly offered access to his PowerPC)

CLISP GSoC project (2018)

- Wrote an SSA-based optimizing compiler targeting the CLISP VM
 - DCE, Loop invariant hoisting, SCC propagation, inlining etc.

SBCL

- Brought up RISC-V backend (Spring 2019)
- Entered the tail-end of PPC64 porting work by adding native threads (once Robert Smith kindly offered access to his PowerPC)

Clasp (Spring 2019)

CLISP GSoC project (2018)

- Wrote an SSA-based optimizing compiler targeting the CLISP VM
 - DCE, Loop invariant hoisting, SCC propagation, inlining etc.

SBCL

- Brought up RISC-V backend (Spring 2019)
- Entered the tail-end of PPC64 porting work by adding native threads (once Robert Smith kindly offered access to his PowerPC)

Clasp (Spring 2019)

LLVM (Summer 2019)

What is RISC-V?

What is RISC-V?

- New CPU architecture spawned by UC Berkeley
 - Original purpose was as a teaching language

What is RISC-V?

- New CPU architecture spawned by UC Berkeley
 - Original purpose was as a teaching language
- A descendant of the original RISC architectures
 - Whence came MIPS as well, the previous teaching language

What is RISC-V?

- New CPU architecture spawned by UC Berkeley
 - Original purpose was as a teaching language
- A descendant of the original RISC architectures
 - Whence came MIPS as well, the previous teaching language
- Fixed some antiquated design decisions
 - No branch delay slots

What is RISC-V?

- New CPU architecture spawned by UC Berkeley
 - Original purpose was as a teaching language
- A descendant of the original RISC architectures
 - Whence came MIPS as well, the previous teaching language
- Fixed some antiquated design decisions
 - No branch delay slots
- Explicitly open-source, libre, patent-free ISA.

What is RISC-V?

- New CPU architecture spawned by UC Berkeley
 - Original purpose was as a teaching language
- A descendant of the original RISC architectures
 - Whence came MIPS as well, the previous teaching language
- Fixed some antiquated design decisions
 - No branch delay slots
- Explicitly open-source, libre, patent-free ISA.
- Extensible
 - Specifications for various ISA extensions modular, base ISA is tiny (only need ~35 instructions to compile C)

What is RISC-V?

- New CPU architecture spawned by UC Berkeley
 - Original purpose was as a teaching language
- A descendant of the original RISC architectures
 - Whence came MIPS as well, the previous teaching language
- Fixed some antiquated design decisions
 - No branch delay slots
- Explicitly open-source, libre, patent-free ISA.
- Extensible
 - Specifications for various ISA extensions modular, base ISA is tiny (only need ~35 instructions to compile C)
- A lot of hype and gaining corporate support
 - Potential uses range from embedded to general purpose (one can hope!)

Why SBCL + RISC-V?

Why SBCL + RISC-V?

- Intro machine structures course last fall semester

Why SBCL + RISC-V?

- Intro machine structures course last fall semester
 - C, RISC-V assembly, CPU architecture (cache, virtual memory etc.)

Why SBCL + RISC-V?

- Intro machine structures course last fall semester
 - C, RISC-V assembly, CPU architecture (cache, virtual memory etc.)
 - Final project was implementing a RISC-V CPU in a visual HDL.

Why SBCL + RISC-V?

- Intro machine structures course last fall semester
 - C, RISC-V assembly, CPU architecture (cache, virtual memory etc.)
 - Final project was implementing a RISC-V CPU in a visual HDL.
 - Upshot: Had to learn every base ISA instruction, its encoding, and how to implement it in hardware

Why SBCL + RISC-V?

- Intro machine structures course last fall semester
 - C, RISC-V assembly, CPU architecture (cache, virtual memory etc.)
 - Final project was implementing a RISC-V CPU in a visual HDL.
 - Upshot: Had to learn every base ISA instruction, its encoding, and how to implement it in hardware
- Christophe Rhodes's blog post about SBCL + RISC-V on trains.

Why SBCL + RISC-V?

- Intro machine structures course last fall semester
 - C, RISC-V assembly, CPU architecture (cache, virtual memory etc.)
 - Final project was implementing a RISC-V CPU in a visual HDL.
 - Upshot: Had to learn every base ISA instruction, its encoding, and how to implement it in hardware
- Christophe Rhodes's blog post about SBCL + RISC-V on trains.
- I like SBCL.

Why SBCL + RISC-V?

- Intro machine structures course last fall semester
 - C, RISC-V assembly, CPU architecture (cache, virtual memory etc.)
 - Final project was implementing a RISC-V CPU in a visual HDL.
 - Upshot: Had to learn every base ISA instruction, its encoding, and how to implement it in hardware
- Christophe Rhodes's blog post about SBCL + RISC-V on trains.
- I like SBCL.
- The time was ripe.

Why SBCL + RISC-V?

- Intro machine structures course last fall semester
 - C, RISC-V assembly, CPU architecture (cache, virtual memory etc.)
 - Final project was implementing a RISC-V CPU in a visual HDL.
 - Upshot: Had to learn every base ISA instruction, its encoding, and how to implement it in hardware
- Christophe Rhodes's blog post about SBCL + RISC-V on trains.
- I like SBCL.
- The time was ripe.
 - Or was it?

Get hardware

- *Really* important

Get hardware

- *Really* important
- I had no hardware, only a (broken) **slow** Fedora qemu VM.
 - cold init initially took about **30** minutes.
 - warm load added another half hour.
 - don't get me started on the test suite.

Get hardware

- *Really* important
- I had no hardware, only a (broken) **slow** Fedora qemu VM.
 - cold init initially took about **30** minutes.
 - warm load added another half hour.
 - don't get me started on the test suite.
- Turns out this porting effort was *very* premature.

Get hardware

- *Really* important
- I had no hardware, only a (broken) **slow** Fedora qemu VM.
 - cold init initially took about **30** minutes.
 - warm load added another half hour.
 - don't get me started on the test suite.
- Turns out this porting effort was very premature.
 - no GDB **:(**

Get hardware

- *Really* important
- I had no hardware, only a (broken) **slow** Fedora qemu VM.
 - cold init initially took about **30** minutes.
 - warm load added another half hour.
 - don't get me started on the test suite.
- Turns out this porting effort was very premature.
 - no GDB :(
 - broken kernel :(

Get hardware

- *Really* important
- I had no hardware, only a (broken) **slow** Fedora qemu VM.
 - cold init initially took about **30** minutes.
 - warm load added another half hour.
 - don't get me started on the test suite.
- Turns out this porting effort was very premature.
 - no GDB **:(**
 - broken kernel **:(**
 - Real Linux-capable SiFive board: \$1000+ **:((**

Get hardware

- *Really* important
- I had no hardware, only a (broken) **slow** Fedora qemu VM.
 - cold init initially took about **30** minutes.
 - warm load added another half hour.
 - don't get me started on the test suite.
- Turns out this porting effort was very premature.
 - no GDB :(
 - broken kernel :(
 - Real Linux-capable SiFive board: \$1000+ :((
 - If I were in the EECS department, maybe more luck.

32-bit vs 64-bit

32-bit vs 64-bit

- RV32 and RV64 not binary compatible.

32-bit vs 64-bit

- RV32 and RV64 not binary compatible.
- Christophe started the port as a 32-bit port.

32-bit vs 64-bit

- RV32 and RV64 not binary compatible.
- Christophe started the port as a 32-bit port.
- So I continued developing it as a 32-bit port.

32-bit vs 64-bit

- RV32 and RV64 not binary compatible.
- Christophe started the port as a 32-bit port.
- So I continued developing it as a 32-bit port.
- Later, found out *only* RV64 is Linux-capable at the moment.

32-bit vs 64-bit

- RV32 and RV64 not binary compatible.
- Christophe started the port as a 32-bit port.
- So I continued developing it as a 32-bit port.
- Later, found out *only* RV64 is Linux-capable at the moment.
- **Oops.**

32-bit vs 64-bit

- RV32 and RV64 not binary compatible.
- Christophe started the port as a 32-bit port.
- So I continued developing it as a 32-bit port.
- Later, found out *only* RV64 is Linux-capable at the moment.
- **Oops.**
- Upshot: Now the only shared {32/64}-bit backend in SBCL.
 - It helps that RV32 and RV64 were designed at the same time.

Write the instruction definitions

Write the instruction definitions

- Christophe defined a good deal of the base ISA instructions.
 - Good macros help!

Write the instruction definitions

- Christophe defined a good deal of the base ISA instructions.
 - Good macros help!
- The regularity of the instruction set is a boon.
 - Adding RV64 support later was easy

Write the instruction definitions

- Christophe defined a good deal of the base ISA instructions.
 - Good macros help!
- The regularity of the instruction set is a boon.
 - Adding RV64 support later was easy
- Needed to implement. . .

Write the instruction definitions

- Christophe defined a good deal of the base ISA instructions.
 - Good macros help!
- The regularity of the instruction set is a boon.
 - Adding RV64 support later was easy
- Needed to implement. . .
 - Labels and addressing modes

Write the instruction definitions

- Christophe defined a good deal of the base ISA instructions.
 - Good macros help!
- The regularity of the instruction set is a boon.
 - Adding RV64 support later was easy
- Needed to implement. . .
 - Labels and addressing modes
 - Relocations

Write the instruction definitions

- Christophe defined a good deal of the base ISA instructions.
 - Good macros help!
- The regularity of the instruction set is a boon.
 - Adding RV64 support later was easy
- Needed to implement. . .
 - Labels and addressing modes
 - Relocations
 - Floating point instructions (F and D extensions)

Write the instruction definitions

- Christophe defined a good deal of the base ISA instructions.
 - Good macros help!
- The regularity of the instruction set is a boon.
 - Adding RV64 support later was easy
- Needed to implement. . .
 - Labels and addressing modes
 - Relocations
 - Floating point instructions (F and D extensions)
 - CSR frobbers

Write the instruction definitions

- Christophe defined a good deal of the base ISA instructions.
 - Good macros help!
- The regularity of the instruction set is a boon.
 - Adding RV64 support later was easy
- Needed to implement. . .
 - Labels and addressing modes
 - Relocations
 - Floating point instructions (F and D extensions)
 - CSR frobbers
- Pseudoinstructions

Write the instruction definitions

- Christophe defined a good deal of the base ISA instructions.
 - Good macros help!
- The regularity of the instruction set is a boon.
 - Adding RV64 support later was easy
- Needed to implement. . .
 - Labels and addressing modes
 - Relocations
 - Floating point instructions (F and D extensions)
 - CSR frobbers
- Pseudoinstructions
 - `define-instruction-macro`

Write the instruction definitions

- Christophe defined a good deal of the base ISA instructions.
 - Good macros help!
- The regularity of the instruction set is a boon.
 - Adding RV64 support later was easy
- Needed to implement. . .
 - Labels and addressing modes
 - Relocations
 - Floating point instructions (F and D extensions)
 - CSR frobbers
- Pseudoinstructions
 - `define-instruction-macro`
 - Load 64-bit Immediate on RV64 - what a doozy!
 - Adding special cases reduced core size in **half**
 - Even the LLVM backend does a worse job

Comment in locall.lisp

```
;;;; Note: Take a look at the compiler-overview.tex  
;;;; section on "Hairy function representation" before  
;;;; you seriously mess with this stuff.  
  
;;;; FIXME: where is that file?
```

Documentation

- Read the CMUCL docs, especially the internals guide by Rob MacLachlan
 - Nothing in the SBCL tree comes close.

Comment in locall.lisp

```
;;;; Note: Take a look at the compiler-overview.tex
;;;; section on "Hairy function representation" before
;;;; you seriously mess with this stuff.

;;;; FIXME: where is that file?
```

Documentation

- Read the CMUCL docs, especially the internals guide by Rob MacLachlan
 - Nothing in the SBCL tree comes close.
- Read Alastair Bridgewater's ARM port logs

Comment in locall.lisp

```
;;;; Note: Take a look at the compiler-overview.tex  
;;;; section on "Hairy function representation" before  
;;;; you seriously mess with this stuff.
```

```
;;;; FIXME: where is that file?
```

Documentation

- Read the CMUCL docs, especially the internals guide by Rob MacLachlan
 - Nothing in the SBCL tree comes close.
- Read Alastair Bridgewater's ARM port logs
- Read the source judiciously.

Comment in locall.lisp

```
;;; Note: Take a look at the compiler-overview.tex  
;;; section on "Hairy function representation" before  
;;; you seriously mess with this stuff.  
  
;;; FIXME: where is that file?
```


Documentation

- Read the CMUCL docs, especially the internals guide by Rob MacLachlan
 - Nothing in the SBCL tree comes close.
- Read Alastair Bridgewater's ARM port logs
- Read the source judicially.
- Get schooled on IRC.

Comment in locall.lisp

```
;;; Note: Take a look at the compiler-overview.tex  
;;; section on "Hairy function representation" before  
;;; you seriously mess with this stuff.
```

```
;;; FIXME: where is that file?
```

Documentation

- Read the CMUCL docs, especially the internals guide by Rob MacLachlan
 - Nothing in the SBCL tree comes close.
- Read Alastair Bridgewater's ARM port logs
- Read the source judicially.
- Get schooled on IRC.
 - I'm karlosz on freenode.

Comment in locall.lisp

```
;;; Note: Take a look at the compiler-overview.tex
;;; section on "Hairy function representation" before
;;; you seriously mess with this stuff.

;;; FIXME: where is that file?
```

Documentation

- Read the CMUCL docs, especially the internals guide by Rob MacLachlan
 - Nothing in the SBCL tree comes close.
- Read Alastair Bridgewater's ARM port logs
- Read the source judicially.
- Get schooled on IRC.
 - I'm karlosz on freenode.
- In the end, none of this really matters that much though.

Comment in locall.lisp

```
;;; Note: Take a look at the compiler-overview.tex
;;; section on "Hairy function representation" before
;;; you seriously mess with this stuff.

;;; FIXME: where is that file?
```

Documentation

- Read the CMUCL docs, especially the internals guide by Rob MacLachlan
 - Nothing in the SBCL tree comes close.
- Read Alastair Bridgewater's ARM port logs
- Read the source judicially.
- Get schooled on IRC.
 - I'm karlosz on freenode.
- In the end, none of this really matters that much though.
 - Just cargo-cult until you understand.

Comment in locall.lisp

```
;;;; Note: Take a look at the compiler-overview.tex
;;;; section on "Hairy function representation" before
;;;; you seriously mess with this stuff.

;;;; FIXME: where is that file?
```

```
(defreg zero 0) (defreg lr 1)
(defreg nsp 2) (defreg global 3)
(defreg thread 4) (defreg lra 5) ; alternate link register
(defreg cfp 6) (defreg ocfp 7)
(defreg nfp 8) (defreg csp 9)
(defreg a0 10) (defreg nl0 11)
(defreg a1 12) (defreg nl1 13)
(defreg a2 14) (defreg nl2 15)
...
(defreg cfunc 26) (defreg lexenv 27)
(defreg null 28) (defreg code 29)
(defreg lip 30) (defreg nargs 31)

(defregset descriptor-regs a0 a1 ... 12 13 ocfp lra lexenv)
(defregset boxed-regs a0 a1 ... 11 12 13 ocfp lra lexenv code)
(define-argument-register-set a0 a1 a2 a3)
```

Write VOPs

Write VOPs

- VOPs are the translators ("templates") that turn backend independent "virtual machine" instructions into native code.

Write VOPs

- VOPs are the translators ("templates") that turn backend independent "virtual machine" instructions into native code.
- Read every backend's version for inspiration on how to do it best. A good way to understand what the IR2 instructions do in the first place.
 - MIPS is architecturally close to RISC-V.
 - ARM64 and x86-64 have the cool new optimizations.
 - PPC somewhere between MIPS and ARM.

Write VOPs

- VOPs are the translators ("templates") that turn backend independent "virtual machine" instructions into native code.
- Read every backend's version for inspiration on how to do it best. A good way to understand what the IR2 instructions do in the first place.
 - MIPS is architecturally close to RISC-V.
 - ARM64 and x86-64 have the cool new optimizations.
 - PPC somewhere between MIPS and ARM.
- Mostly blindly copy and hope it works.

Write VOPs

- VOPs are the translators ("templates") that turn backend independent "virtual machine" instructions into native code.
- Read every backend's version for inspiration on how to do it best. A good way to understand what the IR2 instructions do in the first place.
 - MIPS is architecturally close to RISC-V.
 - ARM64 and x86-64 have the cool new optimizations.
 - PPC somewhere between MIPS and ARM.
- Mostly blindly copy and hope it works.
- The calling convention VOPs are the most design-heavy for a new CPU and least amenable to cargo culting.
 - Good opportunity to use the ISA to its fullest extent.

Write VOPs

- VOPs are the translators ("templates") that turn backend independent "virtual machine" instructions into native code.
- Read every backend's version for inspiration on how to do it best. A good way to understand what the IR2 instructions do in the first place.
 - MIPS is architecturally close to RISC-V.
 - ARM64 and x86-64 have the cool new optimizations.
 - PPC somewhere between MIPS and ARM.
- Mostly blindly copy and hope it works.
- The calling convention VOPs are the most design-heavy for a new CPU and least amenable to cargo culting.
 - Good opportunity to use the ISA to its fullest extent.
- Try and microoptimize every VOP.
 - It's quite satisfying to shave off a few bytes here and there.

Write VOPs

- VOPs are the translators ("templates") that turn backend independent "virtual machine" instructions into native code.
- Read every backend's version for inspiration on how to do it best. A good way to understand what the IR2 instructions do in the first place.
 - MIPS is architecturally close to RISC-V.
 - ARM64 and x86-64 have the cool new optimizations.
 - PPC somewhere between MIPS and ARM.
- Mostly blindly copy and hope it works.
- The calling convention VOPs are the most design-heavy for a new CPU and least amenable to cargo culting.
 - Good opportunity to use the ISA to its fullest extent.
- Try and microoptimize every VOP.
 - It's quite satisfying to shave off a few bytes here and there.
- Dealing with some 64-bit differences in object layout and tagging extremely annoying

Runtime glue

- Implement some assembly routines

Runtime glue

- Implement some assembly routines
- Write operating system support
 - Signal handling, FP handling, and the like

Runtime glue

- Implement some assembly routines
- Write operating system support
 - Signal handling, FP handling, and the like
- GC hookup

- Implement some assembly routines
- Write operating system support
 - Signal handling, FP handling, and the like
- GC hookup
- `call_into_lisp` and `call_into_c`, usually part of a `.S` file.
 - Decided to write it in the Lisp assembler rather than the system assembler, the first backend to do so
 - Have full power of Lisp to generate assembly.

Bootstrap time

Bootstrap time

- OK. Now you've cross-compiled a cold core and have a runtime to run it.

Bootstrap time

- OK. Now you've cross-compiled a cold core and have a runtime to run it.
- Start it in the VM. Lose instantly.

Bootstrap time

- OK. Now you've cross-compiled a cold core and have a runtime to run it.
- Start it in the VM. Lose instantly.
- Debug, fix, repeat.

Bootstrap time

- OK. Now you've cross-compiled a cold core and have a runtime to run it.
- Start it in the VM. Lose instantly.
- Debug, fix, repeat.
- *Anything* bad can happen, especially early in cold-init.

Bootstrap time

- OK. Now you've cross-compiled a cold core and have a runtime to run it.
- Start it in the VM. Lose instantly.
- Debug, fix, repeat.
- *Anything* bad can happen, especially early in cold-init.
 - Illegal instructions, smashed interrupt stacks, off by one errors, bad VOPs, bad instruction encoding, bad runtime glue...

Bootstrap time

- OK. Now you've cross-compiled a cold core and have a runtime to run it.
- Start it in the VM. Lose instantly.
- Debug, fix, repeat.
- *Anything* bad can happen, especially early in cold-init.
 - Illegal instructions, smashed interrupt stacks, off by one errors, bad VOPs, bad instruction encoding, bad runtime glue. . .
 - Good luck finding out why.

Bootstrap time

- OK. Now you've cross-compiled a cold core and have a runtime to run it.
- Start it in the VM. Lose instantly.
- Debug, fix, repeat.
- *Anything* bad can happen, especially early in cold-init.
 - Illegal instructions, smashed interrupt stacks, off by one errors, bad VOPs, bad instruction encoding, bad runtime glue. . .
 - Good luck finding out why.
- GDB will show you the way.

Bootstrap time

- OK. Now you've cross-compiled a cold core and have a runtime to run it.
- Start it in the VM. Lose instantly.
- Debug, fix, repeat.
- *Anything* bad can happen, especially early in cold-init.
 - Illegal instructions, smashed interrupt stacks, off by one errors, bad VOPs, bad instruction encoding, bad runtime glue. . .
 - Good luck finding out why.
- GDB will show you the way.
- I had no GDB.

Bootstrap time

- OK. Now you've cross-compiled a cold core and have a runtime to run it.
- Start it in the VM. Lose instantly.
- Debug, fix, repeat.
- *Anything* bad can happen, especially early in cold-init.
 - Illegal instructions, smashed interrupt stacks, off by one errors, bad VOPs, bad instruction encoding, bad runtime glue. . .
 - Good luck finding out why.
- GDB will show you the way.
- I had no GDB.
 - Write out trace files and manually disassemble cores, step through the instructions manually like it's 1962.

Bootstrap time

- OK. Now you've cross-compiled a cold core and have a runtime to run it.
- Start it in the VM. Lose instantly.
- Debug, fix, repeat.
- *Anything* bad can happen, especially early in cold-init.
 - Illegal instructions, smashed interrupt stacks, off by one errors, bad VOPs, bad instruction encoding, bad runtime glue. . .
 - Good luck finding out why.
- GDB will show you the way.
- I had no GDB.
 - Write out trace files and manually disassemble cores, step through the instructions manually like it's 1962.
 - Hope you didn't smash the stack too hard and corrupt LDB so you can at least read out the registers at crash time.

Bootstrap time

- OK. Now you've cross-compiled a cold core and have a runtime to run it.
- Start it in the VM. Lose instantly.
- Debug, fix, repeat.
- *Anything* bad can happen, especially early in cold-init.
 - Illegal instructions, smashed interrupt stacks, off by one errors, bad VOPs, bad instruction encoding, bad runtime glue. . .
 - Good luck finding out why.
- GDB will show you the way.
- I had no GDB.
 - Write out trace files and manually disassemble cores, step through the instructions manually like it's 1962.
 - Hope you didn't smash the stack too hard and corrupt LDB so you can at least read out the registers at crash time.
- It gets better once you get a Lisp debugger.

The big bad

The big bad

- In late March I hit a bad snag which crippled the porting effort.

The big bad

- In late March I hit a bad snag which crippled the porting effort.
- A signal routine related to sigmask handling in the runtime would always fail.

The big bad

- In late March I hit a bad snag which crippled the porting effort.
- A signal routine related to sigmask handling in the runtime would always fail.
- After much chasing, turned out to be a **kernel** bug.

The big bad

- In late March I hit a bad snag which crippled the porting effort.
- A signal routine related to sigmask handling in the runtime would always fail.
- After much chasing, turned out to be a **kernel** bug.
- By then there were new Fedora images, with GDB and a kernel with a working signal library.

The big bad

- In late March I hit a bad snag which crippled the porting effort.
- A signal routine related to sigmask handling in the runtime would always fail.
- After much chasing, turned out to be a **kernel** bug.
- By then there were new Fedora images, with GDB and a kernel with a working signal library.
- Phillip Matthias Schäfer wanted to try the port and had the same crashes. Indeed he was using the broken VM image.
 - Later he ported sb-rotate-byte support to RISC-V.

- Initially started with cheneygc.
 - Christophe expressed it would probably be easier to start off this way.

- Initially started with cheneygc.
 - Christophe expressed it would probably be easier to start off this way.
- Got a real core with cheneygc eventually.

- Initially started with cheneygc.
 - Christophe expressed it would probably be easier to start off this way.
- Got a real core with cheneygc eventually.
- After pestering on IRC I ported to gengc.

- Initially started with cheneygc.
 - Christophe expressed it would probably be easier to start off this way.
- Got a real core with cheneygc eventually.
- After pestering on IRC I ported to gengc.
 - Much faster

- Initially started with cheneygc.
 - Christophe expressed it would probably be easier to start off this way.
- Got a real core with cheneygc eventually.
- After pestering on IRC I ported to gengc.
 - Much faster
 - cheneygc has some issues on 64-bit platforms.

- Initially started with cheneygc.
 - Christophe expressed it would probably be easier to start off this way.
- Got a real core with cheneygc eventually.
- After pestering on IRC I ported to gengc.
 - Much faster
 - cheneygc has some issues on 64-bit platforms.
- Defaults to gengc now, though technically cheneygc should still work with no problem.

Current status

Current status

- Can run quicklisp.

Current status

- Can run quicklisp.
- Supports `:sb-linkage-table`.

Current status

- Can run quicklisp.
- Supports `:sb-linkage-table`.
- Supports some stack allocation.

Current status

- Can run quicklisp.
- Supports `:sb-linkage-table`.
- Supports some stack allocation.
- Only a few test suite failures remaining

Current status

- Can run quicklisp.
- Supports `:sb-linkage-table`.
- Supports some stack allocation.
- Only a few test suite failures remaining
 - Mostly to do with floating point issues.
 - RISC-V doesn't have hardware floating point traps.

Current status

- Can run quicklisp.
- Supports `:sb-linkage-table`.
- Supports some stack allocation.
- Only a few test suite failures remaining
 - Mostly to do with floating point issues.
 - RISC-V doesn't have hardware floating point traps.
- Much work has been done on optimizing the VOPs and finding a good calling convention.

Current status

- Can run quicklisp.
- Supports `:sb-linkage-table`.
- Supports some stack allocation.
- Only a few test suite failures remaining
 - Mostly to do with floating point issues.
 - RISC-V doesn't have hardware floating point traps.
- Much work has been done on optimizing the VOPs and finding a good calling convention.
- Works for at least one other person on a good qemu image.

Current status

- Can run quicklisp.
- Supports `:sb-linkage-table`.
- Supports some stack allocation.
- Only a few test suite failures remaining
 - Mostly to do with floating point issues.
 - RISC-V doesn't have hardware floating point traps.
- Much work has been done on optimizing the VOPs and finding a good calling convention.
- Works for at least one other person on a good qemu image.
 - Anyone want to test on real hardware?

What's left for RISC-V?

What's left for RISC-V?

- C extension

What's left for RISC-V?

- C extension
 - Potential for big space savings.

What's left for RISC-V?

- C extension
 - Potential for big space savings.
 - Unclear what the right way to codegen this is.

What's left for RISC-V?

- C extension
 - Potential for big space savings.
 - Unclear what the right way to codegen this is.
 - Have instruction emitters choose C-insts AND/OR...

What's left for RISC-V?

- C extension
 - Potential for big space savings.
 - Unclear what the right way to codegen this is.
 - Have instruction emitters choose C-insts AND/OR...
 - Write versions of VOPs using C-insts.

What's left for RISC-V?

- C extension
 - Potential for big space savings.
 - Unclear what the right way to codegen this is.
 - Have instruction emitters choose C-insts AND/OR...
 - Write versions of VOPs using C-insts.
- Alien callbacks

What's left for RISC-V?

- C extension
 - Potential for big space savings.
 - Unclear what the right way to codegen this is.
 - Have instruction emitters choose C-insts AND/OR...
 - Write versions of VOPs using C-insts.
- Alien callbacks
- Better DX handling

What's left for RISC-V?

- C extension
 - Potential for big space savings.
 - Unclear what the right way to codegen this is.
 - Have instruction emitters choose C-insts AND/OR...
 - Write versions of VOPs using C-insts.
- Alien callbacks
- Better DX handling
- Debugger stuffs
 - Single stepping

What's left for RISC-V?

- C extension
 - Potential for big space savings.
 - Unclear what the right way to codegen this is.
 - Have instruction emitters choose C-insts AND/OR...
 - Write versions of VOPs using C-insts.
- Alien callbacks
- Better DX handling
- Debugger stuffs
 - Single stepping
- Native threading
 - Need to understand the RISC-V concurrency model.
 - Need to settle on a synchronization method.
 - AFAICT these specs are still in flux anyway.

What's left for RISC-V?

- C extension
 - Potential for big space savings.
 - Unclear what the right way to codegen this is.
 - Have instruction emitters choose C-insts AND/OR...
 - Write versions of VOPs using C-insts.
- Alien callbacks
- Better DX handling
- Debugger stuffs
 - Single stepping
- Native threading
 - Need to understand the RISC-V concurrency model.
 - Need to settle on a synchronization method.
 - AFAICT these specs are still in flux anyway.
- Jump tables!

What's left for RISC-V?

- C extension
 - Potential for big space savings.
 - Unclear what the right way to codegen this is.
 - Have instruction emitters choose C-insts AND/OR...
 - Write versions of VOPs using C-insts.
- Alien callbacks
- Better DX handling
- Debugger stuffs
 - Single stepping
- Native threading
 - Need to understand the RISC-V concurrency model.
 - Need to settle on a synchronization method.
 - AFAICT these specs are still in flux anyway.
- Jump tables!
 - Leverage Douglas Katzman and Stas Boukarev's efforts on some new IR2 machinery in the backend.

What's left for RISC-V?

- C extension
 - Potential for big space savings.
 - Unclear what the right way to codegen this is.
 - Have instruction emitters choose C-insts AND/OR...
 - Write versions of VOPs using C-insts.
- Alien callbacks
- Better DX handling
- Debugger stuffs
 - Single stepping
- Native threading
 - Need to understand the RISC-V concurrency model.
 - Need to settle on a synchronization method.
 - AFAICT these specs are still in flux anyway.
- Jump tables!
 - Leverage Douglas Katzman and Stas Boukarev's efforts on some new IR2 machinery in the backend.
 - Includes peephole optimizations and case dispatch => jump table work.

What's left for RISC-V?

- C extension
 - Potential for big space savings.
 - Unclear what the right way to codegen this is.
 - Have instruction emitters choose C-insts AND/OR...
 - Write versions of VOPs using C-insts.
- Alien callbacks
- Better DX handling
- Debugger stuffs
 - Single stepping
- Native threading
 - Need to understand the RISC-V concurrency model.
 - Need to settle on a synchronization method.
 - AFAICT these specs are still in flux anyway.
- Jump tables!
 - Leverage Douglas Katzman and Stas Boukarev's efforts on some new IR2 machinery in the backend.
 - Includes peephole optimizations and case dispatch => jump table work.
 - Currently mainly exploited on x86oids and a bit on PPC

Backends for everyone

- Make backend porting easier.

Backends for everyone

- Make backend porting easier.
- Better abstractions

Backends for everyone

- Make backend porting easier.
- Better abstractions
 - Make some helpful macros backend-independent.
 - array getter/setter and floating point VOPs especially.

Backends for everyone

- Make backend porting easier.
- Better abstractions
 - Make some helpful macros backend-independent.
 - array getter/setter and floating point VOPs especially.
- More code sharing, maintainability

Backends for everyone

- Make backend porting easier.
- Better abstractions
 - Make some helpful macros backend-independent.
 - array getter/setter and floating point VOPs especially.
- More code sharing, maintainability
 - Don't want to have to make two sets of identical changes to the 32-bit ports and 64-bit ports living in different directories.

Backends for everyone

- Make backend porting easier.
- Better abstractions
 - Make some helpful macros backend-independent.
 - array getter/setter and floating point VOPs especially.
- More code sharing, maintainability
 - Don't want to have to make two sets of identical changes to the 32-bit ports and 64-bit ports living in different directories.
 - Looking at you, PPC and PPC64.

Backends for everyone

- Make backend porting easier.
- Better abstractions
 - Make some helpful macros backend-independent.
 - array getter/setter and floating point VOPs especially.
- More code sharing, maintainability
 - Don't want to have to make two sets of identical changes to the 32-bit ports and 64-bit ports living in different directories.
 - Looking at you, PPC and PPC64.
- Adding C runtime support could be easier.

Backends for everyone

- Make backend porting easier.
- Better abstractions
 - Make some helpful macros backend-independent.
 - array getter/setter and floating point VOPs especially.
- More code sharing, maintainability
 - Don't want to have to make two sets of identical changes to the 32-bit ports and 64-bit ports living in different directories.
 - Looking at you, PPC and PPC64.
- Adding C runtime support could be easier.
 - Our hookup of GC into backend codegen and runtime is too complicated.

Backends for everyone

- Make backend porting easier.
- Better abstractions
 - Make some helpful macros backend-independent.
 - array getter/setter and floating point VOPs especially.
- More code sharing, maintainability
 - Don't want to have to make two sets of identical changes to the 32-bit ports and 64-bit ports living in different directories.
 - Looking at you, PPC and PPC64.
- Adding C runtime support could be easier.
 - Our hookup of GC into backend codegen and runtime is too complicated.
 - We support two GCs, but in an ad-hoc manner.

Backends for everyone

- Make backend porting easier.
- Better abstractions
 - Make some helpful macros backend-independent.
 - array getter/setter and floating point VOPs especially.
- More code sharing, maintainability
 - Don't want to have to make two sets of identical changes to the 32-bit ports and 64-bit ports living in different directories.
 - Looking at you, PPC and PPC64.
- Adding C runtime support could be easier.
 - Our hookup of GC into backend codegen and runtime is too complicated.
 - We support two GCs, but in an ad-hoc manner.
 - gcgc on older backends seems doable.

Backends for everyone

- Make backend porting easier.
- Better abstractions
 - Make some helpful macros backend-independent.
 - array getter/setter and floating point VOPs especially.
- More code sharing, maintainability
 - Don't want to have to make two sets of identical changes to the 32-bit ports and 64-bit ports living in different directories.
 - Looking at you, PPC and PPC64.
- Adding C runtime support could be easier.
 - Our hookup of GC into backend codegen and runtime is too complicated.
 - We support two GCs, but in an ad-hoc manner.
 - gengc on older backends seems doable.
- Make threads, alien callbacks, gengc etc. less tedious to implement.