# Example

Andrew Clark

July 3, 2022

## 1 Table of Contents

## Contents

## 2 General Information

- The high level assembly language for `volt32_cpu` is intended to be that of a memory-to-memory machine without the awkwardness of instruction LARs. Data LARs machines approximate memory-to-memory machines, and GCC (and probably LLVM) supports arbitrary addressing modes. The problem with attempting to use GCC for a DLARs machine is that multiple DLARs may update at the same time if they point to the same address, assuming the compiler doesn't know that they point to the same address.
- As a side note, there's an LLVM backend targeting the 6502, which basically uses the first 256 bytes of RAM as extra registers. Since the first 256 bytes of memory can be pointed to, this means that LLVM could probably deal with multiple DLARs that point to the same address updating at the same time.

## 3 Instruction LARs

- "At some performance cost, it's pretty cheap to be excessive and always insert (`fetch`es) after branches and on alignment boundaries, or slightly clever and run a pessimistic heuristic cache policy in the assembler, since spurious loads (fetches) are almost free unless you evict something you needed."
- "I could see it being possible to handle ILARs like this: branches to places the assembler can't see (but perhaps the linker can) mean you don't know what ILARs have been modified

  and as such, when you return from a function not in the current translation unit, you 'fetch' into a dedicated ILAR for following a branch

  well, for following a branch outside the translation unit

  You could handle pc-relative branches the same way, I think..."

## 4 Slightly Higher Level, (Almost) DLARs-Only Assembly Language

- ILARs handled mostly automatically, but ILARs affected by Group 9 instructions (`getaddrs`, `fetchm`, etc.)
- DLARs `dt0` and `dt1` reserved for use by the assembler, other than for setting their initial addresses. These should have their addresses set to something near `dfp` and `dsp`.
- `dcp` is handled automatically, so it can't be the destination DLAR of any instruction. It's probably a good idea to not allow accessing it at all.
  - A family of pseudo instructions with names of the form "`cpy<type>i.s`" would be usable to get constant scalars into DLARs. These instructions might adjust the address of `dcp` by use of `dt0` or `dt1`.

- Allow conditional jumping to labels/DLARs. The assembler will insert `fetch`/`fetchi` before the jump, though not if the assembler knows if the jump target address is already in an ILAR.
  - What about `sel`? Generate it automatically when two branch targets are within a single ILAR. I need to keep in mind that `fetch` could support extra RAM.

## 5 High Level, Pseudo Assembly Language

- Dedicate some number of DLARs (perhaps 32?) to act like regular registers. Call these ”pseudo registers”. These will be loaded, at the start of a function, with locations on the stack, close to the frame pointer. Only registers that get used by the function will be allocated on the stack. This enables us to have temporaries, backed up by memory. They will not be the `dA` argument of native loads and stores.
  - It should be possible to do spill/reload of these pseudo registers by just changing the address (load/store) of the real DLARs (by way of a native load/store).
- Include fake addresing modes that access ”memory” at addresses calculated by `dsp + simm32`, `dfp + simm32`, `pseudo_register + simm32`, and `simm32`. These ”memory” accesses will be preceded by native loads or stores into non-pseudo-register-DLARs that are managed entirely by the assembler. Allow generic three-argument instructions using these types of ”memory” accesses, and/or pseudo registers, and/or immediates.
  - Optimize out spurious loads that I can prove are from addresses already in non-pseudo-reg DLARs. Use a combination of common subexpression elimiation and constant propagation.