# Minutes

1. February 2015

Kevin’s comments in **bold**.

* Stack based virtual machines already do stack caching.   
  Research: keeping the top item on the stack cached reduces stack memory traffic by 50%  
  **Stack virtual machines don’t usually cache more than two items from the top of the stack, because of the difficulty in keeping them up to date. This could be discussed in the treatise.**
* Switch dispatch:  
  Results in very high branch target prediction failure, because of indirect dispatch (ie., table lookup)
  + To mitigate, GNU C’s first class label feature can be used to be more explicit about the switching logic. **However, there are penalties for using first class labels, for instance, absolute jumps might need to be used and so dynamic linking to the interpreter isn’t possible**
* Mentions of close to my project:
  + “It is very difficult to keep virtual register items in real machine registers, because real machine registers cannot be accessed array-like, with an index.” - The case for register virtual machines
    - Not relevant for me because I’m switching on operands too
* Direct threaded dispatch - pre-translation of instructions, translates opcode into a pointer to the handler. We could do arithmetic on the entire instruction and index directly into the table
  + **Then we need to divide the table into equal sections, so longer pieces of code will have to jump out of the table to elsewhere.**
* Architecture discussion (**where architecture is not the same thing as instruction set)**
  + **Some mandatory registers:**
    - PC ⬄ rsi
      * Some tricks can be done with the PC(rsi) and the lods[bwd] opcodes for fast instructions[PC++]
        + It might be useful to park the PC in the middle of the instruction after the opcode but before the operands so that the interpreter code has fast access to possible constant operands
    - IR (currently executed instruction) ⬄rax
    - FP (frame base pointer) ⬄ rbp
    - (optional) module pointer
    - Comment: Did we forget to add the stack top pointer ⬄rsp ?
  + The remaining registers will not be guaranteed to be mapped to real registers, and will have generic numbered names. The first few of these registers will be mapped to real registers, (depending on how many the real machine has left) so code should use lower registers for important variables.
  + We had a bit of a back and forth about how the types of registers should be arranged. **You said we should have separate registers for floats, integers, pointers, etc., like p1, r1, f1, etc. Possibly some of these could map to the same registers with the understanding that some of them are mutually exclusive (eg., don’t use f1 when you’re using p1)** I suggested that registers are only given a type when instructions imply that they have type, so add r1, r2 on general purpose registers r1, r2 will treat them as integers, but concat r1, r2 would treat them like string pointers. **You didn’t seem to like that idea,** but I didn’t really understand why. Something about **wanting operator overloads on types, and having a value-based equality instead of a reference equals.** I hope to understand your feelings about this
* Instructions we need:
  + Arithmetic:
    - add, sub, mul, div, mod
    - and, or, xor, not
    - shl, shr, sar
  + Object Member access:
    - Get, set
  + Global constants:
    - mdl – change context to a module with its own constants list, etc
  + Comparison operators:
    - eq, le, isnan, (if I want floats) lt
  + Conditional branch:
    - jcc, (jump if condition is met) jsw (not sure what this is)
  + Misc:
    - jmp – unconditional jump
    - call
    - return