Tangent Project:

Finding Static Information

•••

Implementing a Register Window For a Stack VM Interpreter

Can we take advantage of registers with a stack interpreter?

Lua - Registers for locals
https://the-ravi-programming-language.readthedocs.io/en/latest/lua-parser.html#sliding-register-window-by-mike-pall

We want the top of the stack

- Find information we can leverage
- Specialise the code using templates
- Encode the offset information into the program counter

The Stack Position *Difference* for Each Instruction

Instruction	**Consumes**	**Produces**
PUSH	0	1
ADD, SUB,..	2	1
ROT	0	0
DUP	0	1
DROP	1	0
…		
CALL, RET	*0*	*0*

Templating To Specialise

```
ADD: { auto r = *(sp - 2) + *(sp - 1); sp -= 2; push(r); NEXT(); }

ADDx: { auto r = *(sp - 2) + *(sp - 1); sp -= 2; push(r, x); NEXT(x+(p - c)); }
```

x - register window position

Orange code - Instruction Action

p - produces

c - consumes

Register Window Offset Encoding in Jump Table

How do we track x?

Keep it in a variable - I think you'd have to do this for a switch solution, because control flow goes back to a single block/point for dispatch.

Computed goto Jump Table

```
/*   PUSH,     ADD,     SUB,     ROT,     DUP,     DROP,     CALL1,     IFNEQ,     RET1,     HALT, */
static const void* jmp0[] = {&&PUSH0, &&ADD0, &&SUB0, &&ROT0, &&DUP0, &&DROP0, &&CALL10, &&IFNEQ0, &&RET10, &&HALT0};
static const void* jmp1[] = {&&PUSH1, &&ADD1, &&SUB1, &&ROT1, &&DUP1, &&DROP1, &&CALL11, &&IFNEQ1, &&RET11, &&HALT1};
static const void* jmp2[] = {&&PUSH2, &&ADD2, &&SUB2, &&ROT2, &&DUP2, &&DROP2, &&CALL12, &&IFNEQ2, &&RET12, &&HALT2};
static const void* jmp3[] = {&&PUSH3, &&ADD3, &&SUB3, &&ROT3, &&DUP3, &&DROP3, &&CALL13, &&IFNEQ3, &&RET13, &&HALT3};
static const void* jmp4[] = {&&PUSH4, &&ADD4, &&SUB4, &&ROT4, &&DUP4, &&DROP4, &&CALL14, &&IFNEQ4, &&RET14, &&HALT4};

#define NEXT(N) goto *jmp##N[(unsigned char)*pc++]
```

Bring it all together

```
int r0, r1, r2, r3;

static const void* jmp0[] = {&&PUSH0, &&ADD0, &&SUB0, &&ROT0, &&DUP0, &&DROP0, &&CALL10, &&IFNEQ0, &&RET10, &&HALT0};
static const void* jmp1[] = {&&PUSH1, &&ADD1, &&SUB1, &&ROT1, &&DUP1, &&DROP1, &&CALL11, &&IFNEQ1, &&RET11, &&HALT1};
static const void* jmp2[] = {&&PUSH2, &&ADD2, &&SUB2, &&ROT2, &&DUP2, &&DROP2, &&CALL12, &&IFNEQ2, &&RET12, &&HALT2};
static const void* jmp3[] = {&&PUSH3, &&ADD3, &&SUB3, &&ROT3, &&DUP3, &&DROP3, &&CALL13, &&IFNEQ3, &&RET13, &&HALT3};
static const void* jmp4[] = {&&PUSH4, &&ADD4, &&SUB4, &&ROT4, &&DUP4, &&DROP4, &&CALL14, &&IFNEQ4, &&RET14, &&HALT4};

#define NEXT(N) goto *jmp##N[(unsigned char)*pc++]
NEXT(0);

ADD0: { auto r = *(sp - 2) + *(sp - 1); sp -= 2; r0 = r; NEXT(1); }
ADD1: { auto r = *(sp - 1) + r0;        sp -= 1; r0 = r; NEXT(1); }
ADD2: { r0 = r0 + r1; NEXT(1); }
ADD3: { r1 = r1 + r2; NEXT(2); }
ADD4: { r2 = r2 + r3; NEXT(3); }
```

FLUSH and CALL-RET

When we reach the end of the window, we need to move values to the stack

```
// FLUSH (stack depth)_(keeping in reg)
#define FLUSH4_0 *sp++ = r0; *sp++ = r1; *sp++ = r2; *sp++ = r3;
#define FLUSH4_1 *sp++ = r0; r0 = r3; *sp++ = r1; *sp++ = r2;

PUSH0: { r0 = *pc++; NEXT(1); }
PUSH1: { r1 = *pc++; NEXT(2); }
PUSH2: { r2 = *pc++; NEXT(3); }
PUSH3: { r3 = *pc++; NEXT(4); }
PUSH4: { FLUSH4_0; r0 = *pc++; NEXT(1); }
```

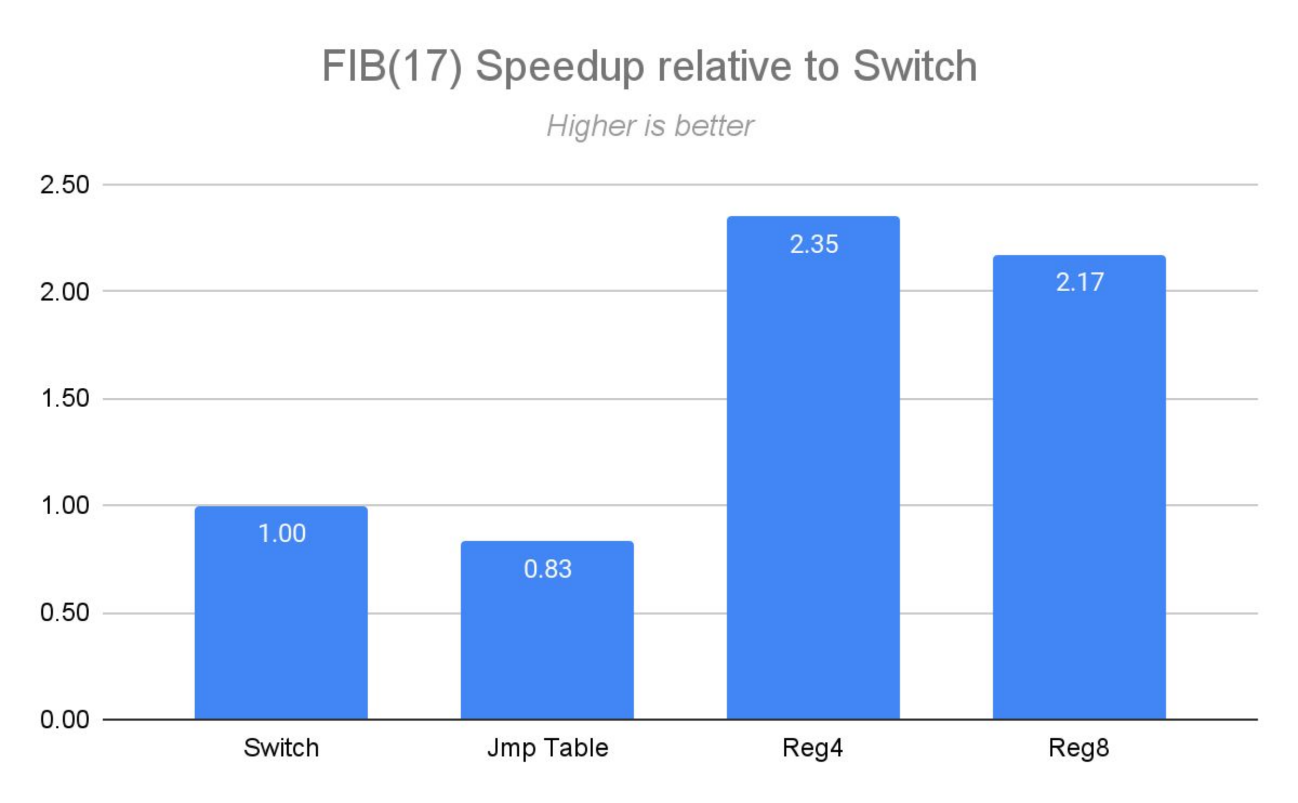
FLUSH and CALL-RET

Stack Superpower: You don’t need to flush for CALLs and RETs.

```
CALL10: { *fp++ = ((pc+1) - code); pc = code + *pc; NEXT(0); }
CALL11: { *fp++ = ((pc+1) - code); pc = code + *pc; NEXT(1); }
CALL12: { *fp++ = ((pc+1) - code); pc = code + *pc; NEXT(2); }
CALL13: { *fp++ = ((pc+1) - code); pc = code + *pc; NEXT(3); }
CALL14: { *fp++ = ((pc+1) - code); pc = code + *pc; NEXT(4); }

RET10:  { auto r = *(fp - 1); fp -= 1; pc = code + r; NEXT(0); }
RET11:  { auto r = *(fp - 1); fp -= 1; pc = code + r; NEXT(1); }
RET12:  { auto r = *(fp - 1); fp -= 1; pc = code + r; NEXT(2); }
RET13:  { auto r = *(fp - 1); fp -= 1; pc = code + r; NEXT(3); }
RET14:  { auto r = *(fp - 1); fp -= 1; pc = code + r; NEXT(4); }
```

Benchmarking



FIB(17)

```
char fib[] = {
      PUSH, 17, CALL1, FIB_FN, HALT,

      // if n == 0, return 0 (actually it’s if n == 0, return n)
      DUP, PUSH, 0, IFNEQ, 1,  RET1,
       // if n == 1, return 1
      DUP, PUSH, 1, IFNEQ, 1, RET1,

      DUP, PUSH, 1, SUB, // (n-1)
      CALL1, FIB_FN, // fib(n-1)

      ROT, PUSH, 2, SUB, // (n-2)
      CALL1, FIB_FN, // fib(n-2)

      ADD,
      RET1
};
```

- Switch vs JmpTable
- Reg4 vs Reg8
  - No ASM - Register allocation done by the compiler
- FIB is mostly call heavy
- FLUSH - We don’t have to drop all values to the stack, we can keep some in registers
  - FLUSH4_1, FLUSH4_2, FLUSH4_3

Future Thoughts

- Stack Instructions - Zero, One, NegOne, => ZERO, DEC, INC
- Dispatch Combining Instructions
  - (Already have PUSH, 1, SUB = DEC)
- Table Size, Block Layout

  -
    ```
    static const int tmp[] = { &&foo - &&foo, &&bar - &&foo, &&hack - &&foo };
    goto *(&&foo + tmp[i]);
    ```

  -
    ```
    ADDSUB: { … }
    SUB: { … NEXT(); }
    ```

Applying this to other VMs

Mill/Belt VM

Similar to the Stack VM, but flush on CALL-RET

Register VM

The register you want to use is in the instruction encoding

```
#define OP1(O, R1) ((O & 0xff) + ((R1 & 0b111) << 8))
    jump_table[OP1(OpMove, 0)] = &&exec_op_move0;
    jump_table[OP1(OpMove, 1)] = &&exec_op_move1;
    jump_table[OP1(OpMove, 2)] = &&exec_op_move2;
exec_op_move0: { regs0 = *(++ip); goto* jump_table[*(++ip)]; }
exec_op_move1: { regs1 = *(++ip); goto* jump_table[*(++ip)]; }
exec_op_move2: { regs2 = *(++ip); goto* jump_table[*(++ip)]; }
```

Mike Brown - mikey.be@gmail.com

Example Source Code:

https://github.com/mikey-b/Register-Window-Stack-VM

Other Tangent Projects:

String Switch - https://github.com/mikey-b/String-Switch

O(m) - where m is the longest case string length.

Linear Pool Allocator - https://github.com/mikey-b/linear_pool_allocator

Mix between a linear and pool allocator - Stack like allocator with support for deallocation anywhere

Fast Dynamic Test/Cast - https://github.com/mikey-b/Parser-Examples/tree/main/C%2B%2B-Dynamic%20Type%20Test

O(1) Dynamic test, supports multiple inheritance