How to Build a Virtual Machine

Terence Parr

University of San Francisco

May 13, 2014

NTLR

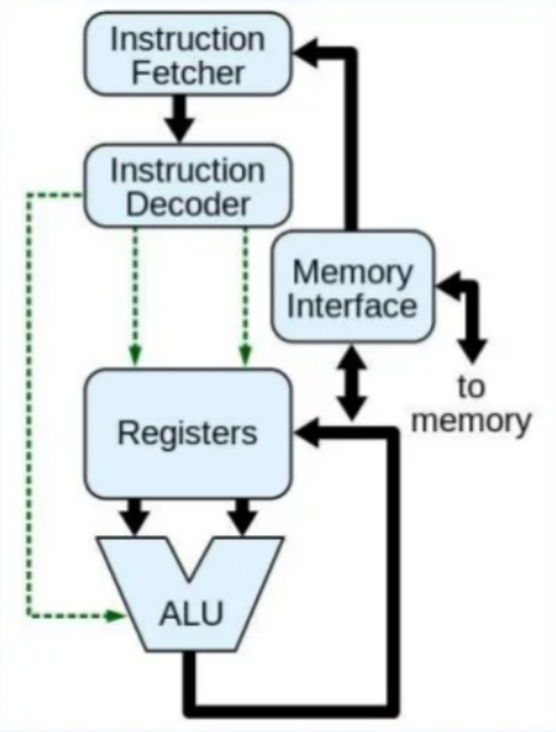
Preliminaries

- What is a VM?
  - A simulated computer that runs a simple instruction set (bytecodes).
- And why do we want one?
  - We can't execute high-level languages like Java directly on the computer
  - Rather than compile to machine code, we generate bytecodes for a VM; much easier
  - We get code portability to anywhere with same VM
  - But bytecodes are slower than raw machine code
  - Ultimately VMs today compile by codes to machine code on the fly

NTLR

Goal: Simulate a simple computer

Here is a real CPU block diagram and code



Machine code Assembly code

```
0600  A4 44   MEMCPY    LDY    CNT+0      ;Set Y = CNT.L
0602  D0 05           BNE    LOOP       ;If CNT.L > 0, then loop
0604  A5 45           LDA    CNT+1      ;If CNT.H > 0,
0606  D0 01           BNE    LOOP       ; then loop
0608  60             RTS               ;Return
0609  B1 40   LOOP      LDA    (SRC),Y    ;Load A from ((SRC)+Y)
060B  91 42           STA    (DST),Y    ;Store A to ((DST)+Y)
060D  88             DEY               ;Decr CNT.L
060E  D0 F9           BNE    LOOP       ;if CNT.L > 0, then loop
0610  E6 41           INC    SRC+1      ;Incr SRC += $0100
0612  E6 43           INC    DST+1      ;Incr DST += $0100
0614  88             DEY               ;Decr CNT.L
0615  C6 45           DEC    CNT+1      ;Decr CNT.H
0617  D0 F0           BNE    LOOP       ;If CNT.H > 0, then loop
0619  60             RTS               ;Return
061A                 END
```

address

NTLR

Programming our VM

- Our bytecodes will be very regular and higher level than machine instructions
- Each bytecode does a tiny bit of work
- Print 1+2:

Stack code

```
ICONST 1
ICONST 2
IADD
PRINT
```

Execution trace

```
0000:   iconst  1    stack=[ 1 ]
0002:   iconst  2    stack=[ 1 2 ]
0004:   iadd         stack=[ 3 ]
0005:   print        stack=[ ]
0006:   halt         stack=[ ]
```

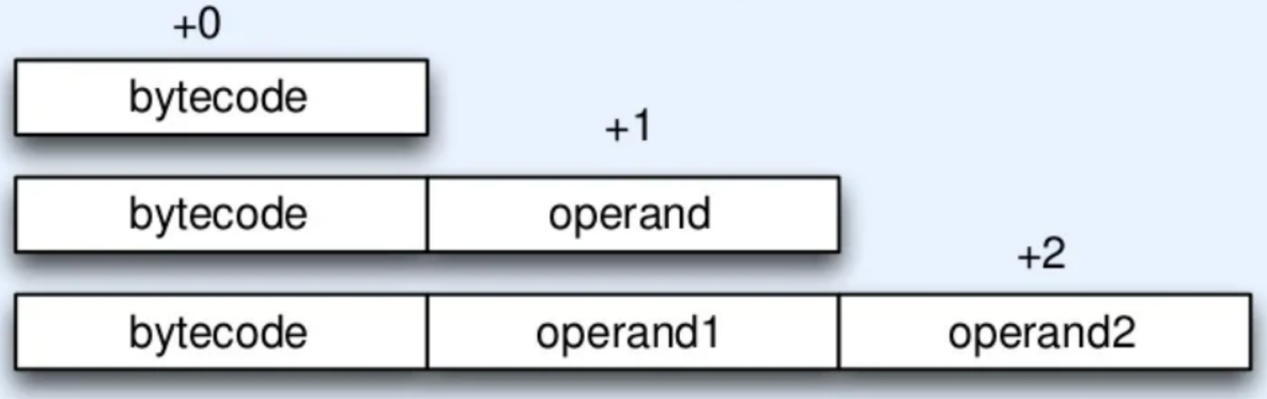
Our instruction set

iadd	integer add (pop 2 operands, add, push result)
isub	
imul	
ilt	integer less than
ieq	
br *addr*	branch to address
brt *addr*	branch if true
brf *addr*	
iconst *value*	push integer constant
load *addr*	load local
gload *addr*	load global variable
store *addr*	
gstore *addr*	
print	
pop	toss out the top of stack
call *addr*, *numArgs*	call *addr*, expect *numArgs*
ret	return from function, expected result on top of stack
halt	

NTLR

Instruction format

- Code memory is 32-bit word addressable
- Bytecodes stored as ints but they are bytes
- Data memory is 32-bit word addressable
- Addresses are just integer numbers
- Operands are 32-bit integers



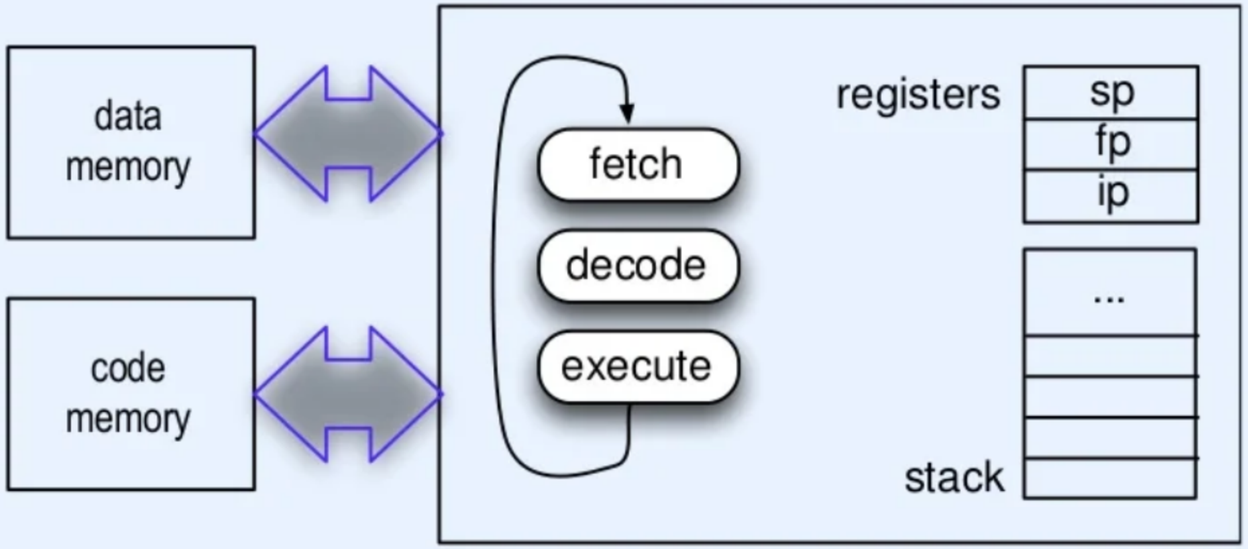
NTLR

Sample bytecodes

Address	*Bytecodes*	*Assembly bytecode*
0000	9 1	ICONST 1
0002	9 2	ICONST 2
0004	1	IADD
0005	14	PRINT

NTLR

Our little stack machine



Fetch: opcode = code[ip]
Decode: switch (opcode) {...}
Execute: stack[++sp] = stack[sp--] + stack[sp--] (iadd instruction)

NTLR

CPU: Fetch, decode, execute cycle

```
void cpu() {
    short bytecode = code[ip];
    while ( «bytecode-not-halt» && ip < code.length ) {
        ip++; //jump to next instruction or first byte of operand
        switch (bytecode) {
            case «bytecode1» : «exec-bytecode1»; break;
            case «bytecode2» : «exec-bytecode2»; break;
            ...
            case «bytecodeN» : «exec-bytecodeN»; break;
        }
        bytecode = code[ip];
    }
}
```

NTLR

Sample implementations

```
case BR :
        ip = code[ip++];
        break;


case IADD:
        b = stack[sp--];          // 2nd opnd at top of stack
        a = stack[sp--];          // 1st opnd 1 below top
        stack[++sp] = a + b;      // push result
        break;
```

NTLR

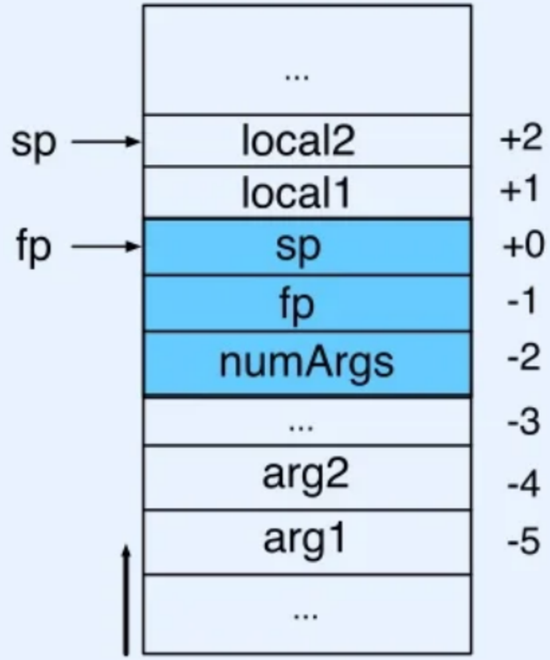
Implementing functions

- f();

```
call f, 0
pop
```

- g(int x,y) {
  int z;
  z = f(x,y);
  ...

```
load x    (fp-4)
load y    (fp-3)
call f, 2
store z   (fp+1)
...
```



- return 9;

```
iconst 9
ret
```

NTLR