# Virtual Machines

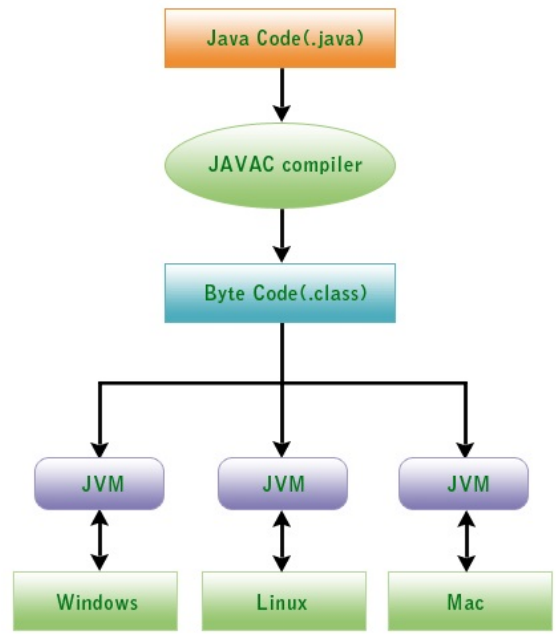
COMP 520: Compiler Design (4 credits)

Alexander Krolik

alexander.krolik@mail.mcgill.ca

MWF 8:30-9:30, TR 1080

http://www.cs.mcgill.ca/~cs520/2019/



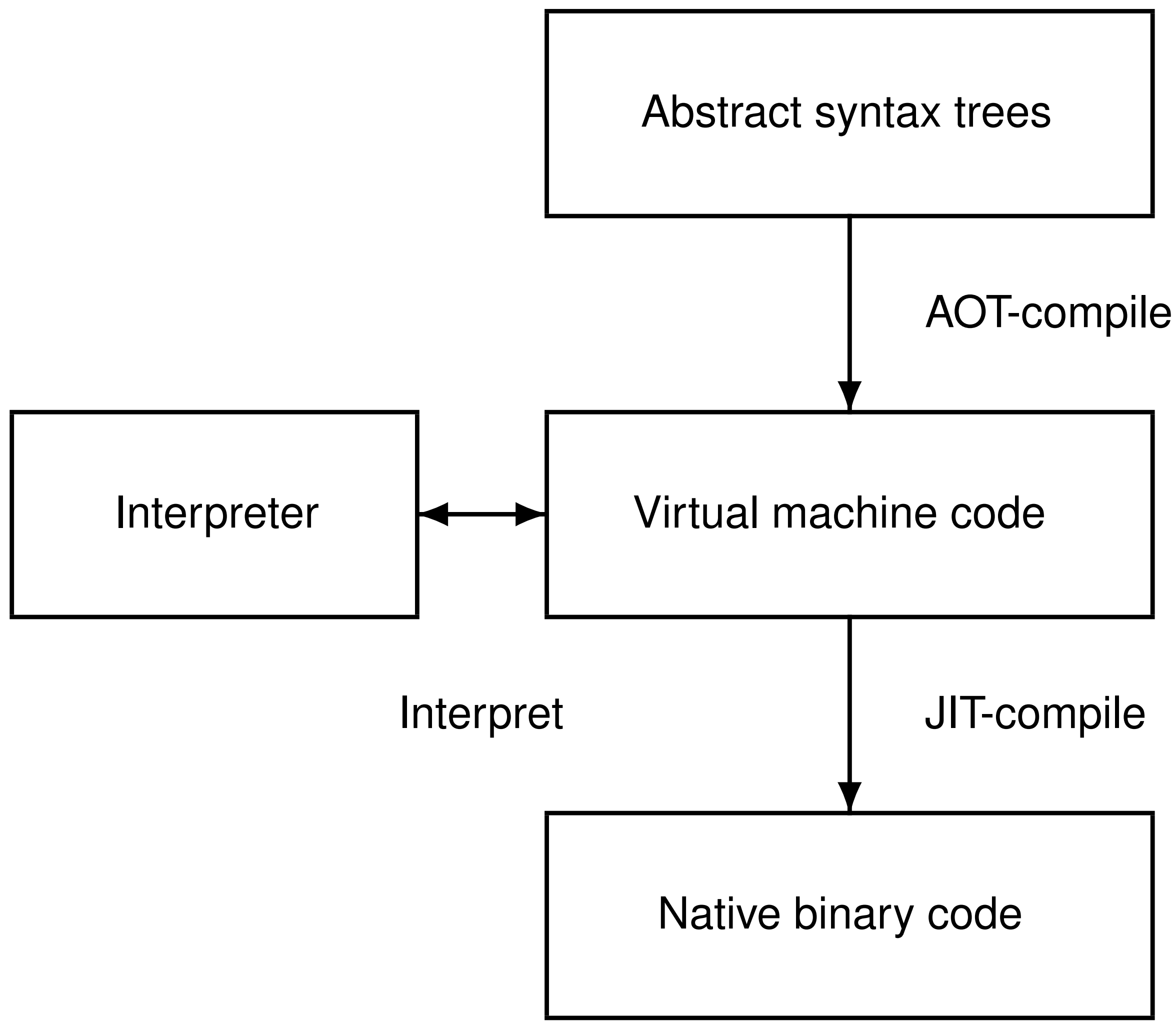

http://www.devmanuals.com/tutorials/java/corejava/JavaVirtualMachine.html

# Announcements (Monday, March 18th)

## Milestones

- Milestone 2 will be graded over the next week, programs posted shortly
- Milestone 3 out today. Due: Wednesday, March 27th 11:59 PM
- Milestone 4 out today. Due: Wednesday, April 10th 11:59 PM
- Final report out today. Due: Wednesday, April 10th 11:59 PM
- Group meeting: Week of April 8th (you may request an extension until the week of April 15th)
- Peephole due: Friday, April 12th 11:59 PM
- Final exam: Thursday, April 18th 2:00 PM

# Compilation and Execution in Virtual Machines



# Virtual Machines

In this class we look at two different virtual machines

**Java Virtual Machine:** stack-based IR

**VirtualRISC:** register-based IR

# VirtualRISC

VirtualRISC is a simple RISC machine (similar to what you've seen in COMP 273)

- Memory;
- Registers;
- Condition codes; and
- Execution unit.

**In this model we ignore**

- Caches;
- Pipelines;
- Branch prediction units; and
- Advanced features.

We focus instead on the basic architecture of register-based machines.

# VirtualRISC Memory

VirtualRISC has several types of memory for storing program information

- A stack
  (used for function call frames);
- A heap
  (used for dynamically allocated memory);
- A global pool
  (used to store global variables); and
- A code segment
  (used to store VirtualRISC instructions).

# VirtualRISC Registers

VirtualRISC has general purpose registers used for computation, and special registers that are managed by the machine

- Unbounded number of general purpose registers `Ri`;
- Stack pointer (`sp`) which points to the top of the stack;
- Frame pointer (`fp`) which points to the current stack frame; and
- Program counter (`pc`) which points to the current instruction.

# VirtualRISC Execution

## Condition codes

- Condition codes are set by instructions which evaluate a predicate (i.e. comparisons); and
- Are used for branching instructions.

## Execution unit

- Reads the VirtualRISC instruction at the current `pc`, decodes the instruction and executes it;
- This may change the state of the machine (memory, registers, condition codes);
- The `pc` is automatically incremented after executing an instruction; but
- Function calls and branches explicitly change the `pc`.

# VirtualRISC Program

A VirtualRISC program consists of a list of instructions and labels

**Instruction types**

- Moves between registers and memory;
- Mathematical operations;
- Comparisons;
- Branches; or
- Other, special instructions.

Operands to instructions can either be memory addresses, registers, or constants.

# Memory Move Instructions

`[..]` indicates the memory location stored in the register

### Store

Store instructions copy the contents from a register to a memory location: `st <src>,<dst>`

```
st Ri,[Rj]          [Rj] := Ri
st Ri,[Rj+C]        [Rj+C] := Ri
```

### Load

Load instructions copy the contents from a memory location to a register: `ld <src>,<dst>`

```
ld [Ri],Rj          Rj := [Ri]
ld [Ri+C],Rj        Rj := [Ri+C]
```

### Move

The last move instruction `mov <src>,<dst>` copies the contents between registers. The source register may also be replaced by a constant (i.e. `mov 5,R1`)

```
mov Ri,Rj           Rj := Ri
```

# Mathematical Operations

Mathematical operations are performed between two source registers and stored in a destination register

```
op <src1>,<src2>,<dst>
```

The source registers may be replaced by constants (i.e. `add R1,5,R2`)

```
add Ri,Rj,Rk        Rk := Ri + Rj
sub Ri,Rj,Rk        Rk := Ri - Rj
mul Ri,Rj,Rk        Rk := Ri * Rj
div Ri,Rj,Rk        Rk := Ri / Rj
```

# Branching Instructions

The `cmp` instruction sets the condition codes depending on the relation between its operands

```
cmp Ri,Rj
```

Just like the mathematical operators, constants may be used as operands.

**Branching instructions**

Depending on the condition codes, the branch operation may/may not be executed

```
b L
bg L
bge L
bl L
ble L
bne L
```

To express `if R1 <= 0 goto L1` we write

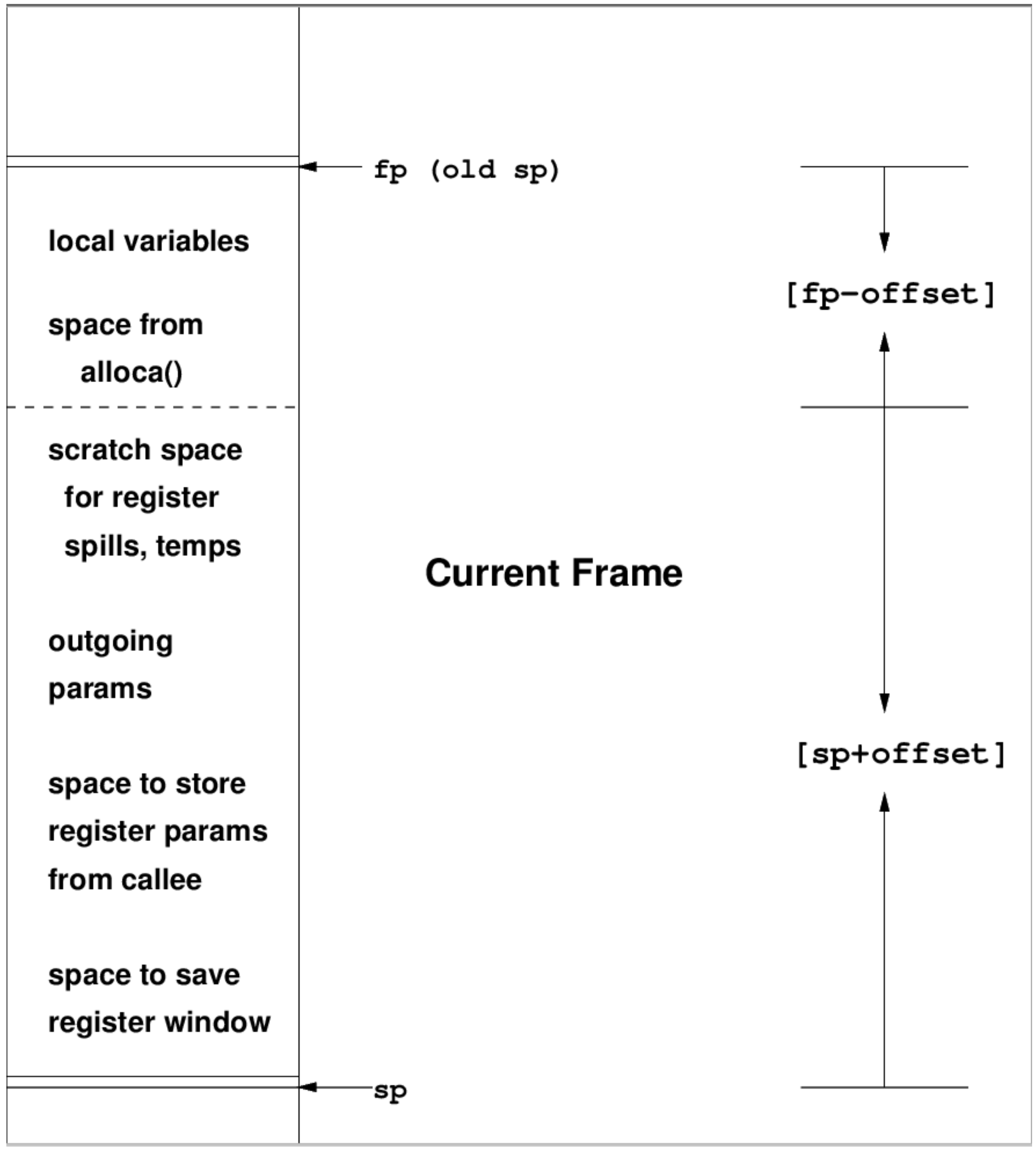
```
cmp R1,0
ble L1
```

# Other Special Instructions

VirtualRISC also has the following special instructions for managing the stack with function calls

```
call L              R15:=pc; pc:=L
save sp,-C,sp       save registers,
                    allocating C bytes
                    on the stack
restore             restore registers
ret                 pc:=R15+8
nop                 do nothing
```

# Stack Frame


fp (old sp)
local variables
space from
alloca()
[fp-offset]
scratch space
for register
spills, temps
Current Frame
outgoing
params
[sp+offset]
space to store
register params
from callee
space to save
register window
sp

# Stack Frames

- Store the function call hierarchy and the respective program memory;
- `sp` and `fp` point to stack frames;
- When a function is called a new stack frame is created:

  `push fp; fp := sp; sp := sp + C;`
- When a function returns, the top stack frame is popped:

  `sp := fp; fp = pop;`
- Local variables are stored relative to `fp`;
- The figure shows additional features of the SPARC architecture.

# Calling semantics

## Calling

- Functions start by allocating the stack frame using `save sp,-C,sp`;
- Functions end by restoring the previous stack frame and register window (`restore`) and returning (`ret`); and
- The return value is stored in register `R0`.

## Parameters

- Passed in registers `R0,R1,`etc; and
- May be stored in memory. By convention we use `fp+68+4k` where `k` is some non-negative integer. Note that this means we are storing parameters in the *callers* frame!

## Local variables

- Use any general purpose register; and
- May be stored in memory. By convention we use `fp-4k` where `k` is some non-zero integer

# Writing VirtualRISC Code

Write the following C code in VirtualRISC. Try using no register allocation scheme - this means that values should be loaded into registers directly before operations and the value stored back to memory immediately.

```
int fact(int n) {
   int i, sum;
   sum = 1;
   i = 2;
   while (i <= n) {
      sum = sum * i;
      i = i + 1;
   }
   return sum;
}
```

# Writing VirtualRISC Code

```
int fact(int n) {
    int i, sum;
    sum = 1;
    i = 2;
    while (i <= n) {
        sum = sum * i;
        i = i + 1;
    }
    return sum;
}
```

```
_fact:
  save sp,-112,sp     // save stack frame
  st R0,[fp+68]       // save arg n in frame of CALLER
  mov 1,R0            // R0 := 1
  st R0,[fp-16]       // [fp-16] is location for sum
  mov 2,R0            // RO := 2
  st RO,[fp-12]       // [fp-12] is location for i
L3:
  ld [fp-12],R0       // load i into R0
  ld [fp+68],R1       // load n into R1
  cmp R0,R1           // compare R0 to R1
  ble L5              // if R0 <= R1 goto L5
  b L4                // goto L4
L5:
  ld [fp-16],R0       // load sum into R0
  ld [fp-12],R1       // load i into R1
  mul R0,R1,R0        // R0 := R0 * R1
  st R0,[fp-16]       // store R0 into sum
  ld [fp-12],R0       // load i into R0
  add R0,1,R1         // R1 := R0 + 1
  st R1,[fp-12]       // store R1 into i
  b L3                // goto L3
L4:
  ld [fp-16],R0       // put return value of sum into R0
  restore             // restore register window
  ret                 // return from function
```

# Fibonacci

More practice! Write the following C program in VirtualRISC

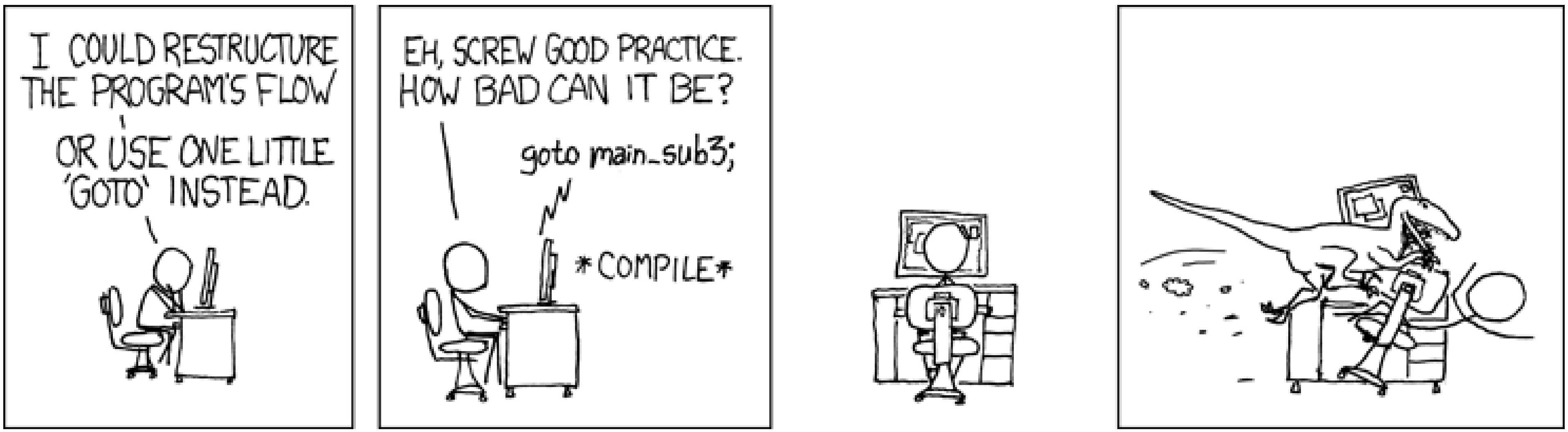
```
int fib(int x) {
   int current, last, sum;

   current = 1;
   last = 1;
   sum = 1;

   x = x - 2;
   while (x > 0) {
      sum = current + last;
      last = current;
      current = sum;
      x = x - 1;
   }

   return sum;
}
```

# Goto!




https://xkcd.com/292/

# What does this go?

Try writing the VirtualRISC code

```
int thing(int a, int b) {
   int temp, iter;

   while (1) {
      temp = a;
      iter = 0;

      while (iter - b) {
         temp = temp + 1;

         if (temp - b) {
            iter = iter + 1;
         } else {
            goto ret;
         }
      }

      a = a - b;
   }
ret:
   return a;
}
```

# This Class

## Java bytecode

- The JOOS compiler produces Java bytecode in Jasmin format; and
- The JOOS peephole optimizer transforms bytecode into more efficient bytecode.

## VirtualRISC

- Java bytecode can be converted into machine code at run-time using a JIT (Just-In-Time) compiler;
- We will study some examples of converting Java bytecode into a language similar to VirtualRISC;
- We will study some simple, standard optimizations on VirtualRISC.

# Let's Practice!

Write VirtualRISC code for the following function

```
int power1(int x, int n) {
   int i;
   int prod = 1;
   for (i = 0; i < n; i++)
      prod = prod * (x + 1);
   return prod;
}
```

## Assumptions

- x is in R0 and n is in R1 on input;
- The result should be returned in R0; and
- The variables are mapped to following spots in the stack frame.

```
Parameters:  x ->  [fp+68]       n ->    [fp+72]
Locals:      i ->  [fp-12]       prod -> [fp-16]
```

Try, `gcc -S power1.c` and `gcc -O -S power1.c`, and compare the difference.

# VirtualRISC Code (Loop Invariant Removal)

```
_power1:
  save sp,-112,sp     // save stack frame
  st R0,[fp+68]       // save input args x, n in frame of CALLER
  st R1,[fp+72]       // R0 holds x,  R1 holds n

  mov 1,R2            // R2 :=1, R2 holds prod
  add R0,1,R4         // R4 := x + 1, loop invariant
  mov 0,R3            // R3 := 0, R3 holds i
begin_loop:
  cmp R3,R1           // if (i < n)
  bge end_loop
begin_body:
  mul R2,R4,R2        // prod = prod * (x+1)
  add R3,1,R3         // i = i + 1
  goto begin_loop
end_loop:
  mov R2, R0          // put return value of prod into R0
  restore             // restore register window
  ret                 // return from function
```