Memory/register instructions:

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

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

Register/register instructions:

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

Constants may be used in place of register values: `mov 5,R1`.

Instructions that set the condition codes:

```
cmp Ri,Rj
```

Instructions to branch:

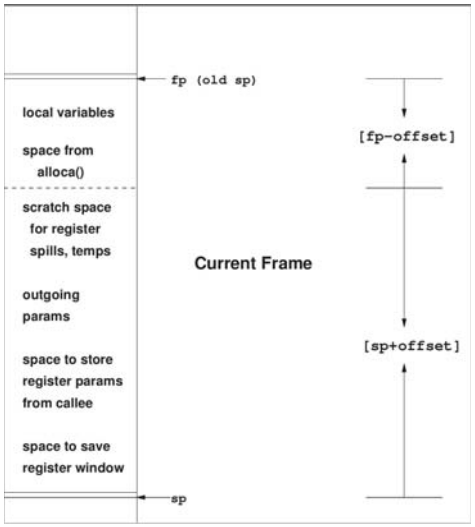
```
b L
bg L
bge L
bl L
ble L
bne L
```

To express: `if R1 <= 9 goto L1`

we code:
```
cmp R1,9
ble L1
```


Other instructions:

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

Stack frames:

- stores function activations;
- `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.

A simple C function:

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


Corresponding VirtualRISC code:

```
_fact:
  save sp,-112,sp   // save stack frame
  st R0,[fp+68]     // save input arg n in frame of CALLER
  mov 1,R0          // R0 := 1
  st R0,[fp-16]     // [fp-16] is location for sum
  mov 2,R0          // R0 := 2
  st R0,[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
```


Unary arithmetic operations:

```
ineg        [...:i] -> [...:-i]
i2c         [...:i] -> [...:i%65536]
```

Binary arithmetic operations:

```
iadd        [...:i1:i2] -> [...:i1+i2]
isub        [...:i1:i2] -> [...:i1-i2]
imul        [...:i1:i2] -> [...:i1*i2]
idiv        [...:i1:i2] -> [...:i1/i2]
irem        [...:i1:t2] -> [...:i1%i2]
```

Direct operations:

```
iinc k a    [...]  -> [...]
            local[k]=local[k]+a
```

Nullary branch operations:

```
goto L          [...]  -> [...]
                branch always
```

Unary branch operations:

```
ifeq L          [...:i] -> [...]
                branch if i == 0
ifne L          [...:i] -> [...]
                branch if i != 0

ifnull L        [...:o] -> [...]
                branch if o == null
ifnonnull L     [...:o] -> [...]
                branch if o != null
```


Binary branch operations:

```
if_icmpeq L     [...:i1:i2] -> [...]
                branch if i1 == i2
if_icmpne L     [...:i1:i2] -> [...]
                branch if i1 != i2
if_icmpgt L     [...:i1:i2] -> [...]
                branch if i1 > i2
if_icmplt L     [...:i1:i2] -> [...]
                branch if i1 < i2
if_icmple L     [...:i1:i2] -> [...]
                branch if i1 <= i2
if_icmpge L     [...:i1:i2] -> [...]
                branch if i1 >= i2

if_acmpeq L     [...:o1:o2] -> [...]
                branch if o1 == o2
if_acmpne L     [...:o1:o2] -> [...]
                branch if o1 != o2
```


Constant loading operations:

```
iconst_0        [...]  -> [...:0]
iconst_1        [...]  -> [...:1]
iconst_2        [...]  -> [...:2]
iconst_3        [...]  -> [...:3]
iconst_4        [...]  -> [...:4]
iconst_5        [...]  -> [...:5]

aconst_null     [...]  -> [...:null]

ldc_int i       [...]  -> [...:i]
ldc_string s    [...]  -> [...:String(s)]
```


Locals operations:

```
iload k           [...]  -> [...:local[k]]
istore k          [...:i] -> [...]
                  local[k]=i

aload k           [...]  -> [...:local[k]]
astore k          [...:o] -> [...]
                  local[k]=o
```

Field operations:

```
getfield f sig    [...:o] -> [...:o.f]
putfield f sig    [...:o:v] -> [...]
                  o.f=v
```

Stack operations:

```
dup          [...:v1] -> [...:v1:v1]
pop          [...:v1] -> [...]
swap         [...:v1:v2] -> [...:v2:v1]
nop          [...]  -> [...]
```


Class operations:

```
new C           [...]  -> [...:o]

(note that new only allocates space, must call <init> to execute bo

instance_of C   [...:o] -> [...:i]
                if (o==null) i=0
                else i=(C<=type(o))

checkcast C     [...:o] -> [...:o]
                if (o!=null && !C<=type(o))
                throw ClassCastException
```


Method operations:

```
invokevirtual m sig
     [...:o:a1:...:an] -> [...]

//overloading already resolved:
// signature of m is known!
entry=lookupHierarchy(m,sig,class(o));
block=block(entry);
push stack frame of size
     block.locals+block.stacksize;
local[0]=o; //local points to
local[1]=a1; //beginning of frame
...
local[n]=an;
pc=block.code;
```


Method operations:

```
invokespecial m sig
     [...:o:a1:...:an] -> [...]

//overloading already resolved:
// signature of m is known!
entry=lookupClassOnly(m,sig,class(o));
block=block(entry);
push stack frame of size
     block.locals+block.stacksize;
local[0]=o; //local points to
local[1]=a1; //beginning of frame
...
local[n]=an;
pc=block.code;
```

For which method calls is `invokespecial` used?
ANSWER: <init>(..), private, super method calls
Also, `invokestatic` and `invokeinterface`.

Method operations:

```
ireturn     [...:<frame>:i] -> [...:i]
            pop stack frame,
            push i onto frame of caller

areturn     [...:<frame>:o] -> [...:o]
            pop stack frame,
            push o onto frame of caller

return      [...:<frame>] -> [...]
            pop stack frame
```

Those operations also release locks in `synchronized` methods.

A Java method:

```
public boolean member(Object item)
{ if (first.equals(item))
    return true;
  else if (rest == null)
    return false;
  else
    return rest.member(item);
}
```


Corresponding bytecode (in Jasmin syntax):

```
.method public member(Ljava/lang/Object;)Z
.limit locals 2            // local[0] = o
                           // local[1] = item
.limit stack 2             // initial stack [ * * ]
aload_0                    // [ o * ]
getfield Cons/first Ljava/lang/Object;
                           // [ o.first *]
aload_1                    // [ o.first item]
invokevirtual java/lang/Object/equals(Ljava/lang/Object;)Z
                           // [ b * ] for some boolean b
ifeq else_1                // [ * * ]
iconst_1                   // [ 1 * ]
ireturn                    // [ * * ]
else_1:
aload_0                    // [ o * ]
getfield Cons/rest LCons;  // [ o.rest * ]
aconst_null                // [ o.rest null]
if_acmpne else_2           // [ * * ]
iconst_0                   // [ 0 * ]
ireturn                    // [ * * ]
else_2:
aload_0                    // [ o * ]
getfield Cons/rest LCons;  // [ o.rest * ]
aload_1                    // [ o.rest item ]
invokevirtual Cons/member(Ljava/lang/Object;)Z
                           // [ b * ] for some boolean b
ireturn                    // [ * * ]
.end method
```