

LLJVM Backend

Generated by Doxygen 1.5.8

Tue Dec 29 17:21:35 2009

Contents

1	Class Index	1
1.1	Class List	1
2	Class Documentation	3
2.1	JVMWriter Class Reference	3
2.1.1	Detailed Description	9
2.1.2	Constructor & Destructor Documentation	9
2.1.2.1	JVMWriter	9
2.1.3	Member Function Documentation	10
2.1.3.1	doFinalization	10
2.1.3.2	doInitialization	10
2.1.3.3	getAnalysisUsage	11
2.1.3.4	getBitWidth	11
2.1.3.5	getCallSignature	12
2.1.3.6	getLabelName	12
2.1.3.7	getLocalVarNumber	13
2.1.3.8	getTypeDescriptor	13
2.1.3.9	getTypeID	14
2.1.3.10	getTypeName	14
2.1.3.11	getTypePostfix	15
2.1.3.12	getTypePrefix	16
2.1.3.13	getValueName	16
2.1.3.14	printAllocaInstruction	17
2.1.3.15	printArithmeticInstruction	17
2.1.3.16	printBasicBlock	18
2.1.3.17	printBinaryInstruction	19
2.1.3.18	printBinaryInstruction	19
2.1.3.19	printBitCastInstruction	20

2.1.3.20	printBitIntrinsic	20
2.1.3.21	printBranchInstruction	21
2.1.3.22	printBranchInstruction	21
2.1.3.23	printBranchInstruction	22
2.1.3.24	printCallInstruction	22
2.1.3.25	printCastInstruction	22
2.1.3.26	printCastInstruction	23
2.1.3.27	printCatchJump	24
2.1.3.28	printCmpInstruction	24
2.1.3.29	printConstantExpr	25
2.1.3.30	printConstLoad	26
2.1.3.31	printConstLoad	27
2.1.3.32	printConstLoad	27
2.1.3.33	printConstLoad	28
2.1.3.34	printConstLoad	28
2.1.3.35	printFunction	29
2.1.3.36	printFunctionBody	30
2.1.3.37	printFunctionCall	31
2.1.3.38	printGepInstruction	32
2.1.3.39	printIndirectLoad	33
2.1.3.40	printIndirectLoad	33
2.1.3.41	printIndirectStore	33
2.1.3.42	printIndirectStore	34
2.1.3.43	printInstruction	34
2.1.3.44	printIntrinsicCall	36
2.1.3.45	printInvokeInstruction	37
2.1.3.46	printLabel	37
2.1.3.47	printLabel	37
2.1.3.48	printLocalVariable	38
2.1.3.49	printLoop	38
2.1.3.50	printMallocInstruction	39
2.1.3.51	printMathIntrinsic	39
2.1.3.52	printMemIntrinsic	40
2.1.3.53	printOperandPack	41
2.1.3.54	printPHICopy	41
2.1.3.55	printPtrLoad	42

2.1.3.56	<code>printSelectInstruction</code>	42
2.1.3.57	<code>printSimpleInstruction</code>	43
2.1.3.58	<code>printSimpleInstruction</code>	43
2.1.3.59	<code>printSimpleInstruction</code>	43
2.1.3.60	<code>printSimpleInstruction</code>	44
2.1.3.61	<code>printStaticConstant</code>	44
2.1.3.62	<code>printSwitchInstruction</code>	45
2.1.3.63	<code>printVArgInstruction</code>	46
2.1.3.64	<code>printVAIntrinsic</code>	46
2.1.3.65	<code>printValueLoad</code>	47
2.1.3.66	<code>printValueStore</code>	48
2.1.3.67	<code>printVirtualInstruction</code>	48
2.1.3.68	<code>printVirtualInstruction</code>	49
2.1.3.69	<code>printVirtualInstruction</code>	49
2.1.3.70	<code>printVirtualInstruction</code>	50
2.1.3.71	<code>printVirtualInstruction</code>	50
2.1.3.72	<code>printVirtualInstruction</code>	50
2.1.3.73	<code>runOnFunction</code>	51
2.1.3.74	<code>sanitizeName</code>	51

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

JVMWriter (A FunctionPass for generating Jasmin-style assembly for the JVM)	3
---	---

Chapter 2

Class Documentation

2.1 JVMWriter Class Reference

A FunctionPass for generating Jasmin-style assembly for the JVM.

```
#include <backend.h>
```

Public Member Functions

- [JVMWriter](#) (const TargetData *td, formatted_raw_ostream &o, const std::string &cls, unsigned int dbg)

Constructor.

Private Member Functions

- void [getAnalysisUsage](#) (AnalysisUsage &au) const
Register required analysis information.
- bool [runOnFunction](#) (Function &f)
Process the given function.
- bool [doInitialization](#) (Module &m)
Perform per-module initialization.
- bool [doFinalization](#) (Module &m)
Perform per-module finalization.
- void [printBasicBlock](#) (const BasicBlock *block)
Print the given basic block.
- void [printInstruction](#) (const Instruction *inst)
Print the given instruction.
- void [printPHICopy](#) (const BasicBlock *src, const BasicBlock *dest)

Replace PHI instructions with copy instructions (load-store pairs).

- void [printBranchInstruction](#) (const BasicBlock *curBlock, const BasicBlock *destBlock)
Print an unconditional branch instruction.
- void [printBranchInstruction](#) (const BasicBlock *curBlock, const BasicBlock *trueBlock, const BasicBlock *falseBlock)
Print a conditional branch instruction.
- void [printBranchInstruction](#) (const BranchInst *inst)
Print a branch instruction.
- void [printSelectInstruction](#) (const Value *cond, const Value *trueVal, const Value *falseVal)
Print a select instruction.
- void [printSwitchInstruction](#) (const SwitchInst *inst)
Print a switch instruction.
- void [printLoop](#) (const Loop *l)
Print a loop.
- void [printPtrLoad](#) (uint64_t n)
Load the given pointer.
- void [printConstLoad](#) (const APInt &i)
Load the given integer.
- void [printConstLoad](#) (float f)
Load the given single-precision floating point value.
- void [printConstLoad](#) (double d)
Load the given double-precision floating point value.
- void [printConstLoad](#) (const Constant *c)
Load the given constant.
- void [printConstLoad](#) (const std::string &str, bool cstring)
Load the given string.
- void [printStaticConstant](#) (const Constant *c)
Store the given static constant.
- void [printConstantExpr](#) (const ConstantExpr *ce)
Print the given constant expression.
- std::string [getCallSignature](#) (const FunctionType *ty)
Return the call signature of the given function type.
- void [printOperandPack](#) (const Instruction *inst, unsigned int minOperand, unsigned int maxOperand)

Pack the specified operands of the given instruction into memory.

- void [printFunctionCall](#) (const Value *functionVal, const Instruction *inst)
Print a call/invoke instruction.
- void [printIntrinsicCall](#) (const IntrinsicInst *inst)
Print a call to an intrinsic function.
- void [printCallInstruction](#) (const Instruction *inst)
Print a call instruction.
- void [printInvokeInstruction](#) (const InvokeInst *inst)
Print an invoke instruction.
- void [printLocalVariable](#) (const Function &f, const Instruction *inst)
Allocate a local variable for the given function.
- void [printFunctionBody](#) (const Function &f)
Print the body of the given function.
- unsigned int [getLocalVarNumber](#) (const Value *v)
Return the local variable number of the given value.
- void [printCatchJump](#) (unsigned int numJumps)
Print the block to catch Jump objects (thrown by longjmp).
- void [printFunction](#) (const Function &f)
Print the given function.
- void [printCmpInstruction](#) (unsigned int predicate, const Value *left, const Value *right)
Print an icmp/fcmp instruction.
- void [printArithmeticInstruction](#) (unsigned int op, const Value *left, const Value *right)
Print an arithmetic instruction.
- void [printBitCastInstruction](#) (const Type *ty, const Type *srcTy)
Print a bitcast instruction.
- void [printCastInstruction](#) (const std::string &typePrefix, const std::string &srcTypePrefix)
Print a cast instruction.
- void [printCastInstruction](#) (unsigned int op, const Value *v, const Type *ty, const Type *srcTy)
Print a cast instruction.
- void [printGepInstruction](#) (const Value *v, gep_type_iterator i, gep_type_iterator e)
Print a getelementptr instruction.
- void [printAllocaInstruction](#) (const AllocaInst *inst)
Print an alloca instruction.

- void [printVAArgInstruction](#) (const VArgInst *inst)
Print a va_arg instruction.
- void [printVAIntrinsic](#) (const IntrinsicInst *inst)
Print a vararg intrinsic function.
- void [printMemIntrinsic](#) (const MemIntrinsic *inst)
Print a memory intrinsic function.
- void [printMallocInstruction](#) (const MallocInst *inst)
Print a malloc instruction.
- void [printMathIntrinsic](#) (const IntrinsicInst *inst)
Print a mathematical intrinsic function.
- void [printBitIntrinsic](#) (const IntrinsicInst *inst)
Print a bit manipulation intrinsic function.
- void [printValueLoad](#) (const Value *v)
Load the given value.
- void [printValueStore](#) (const Value *v)
Store the value currently on top of the stack to the given local variable.
- void [printIndirectLoad](#) (const Value *v)
Load a value from the given address.
- void [printIndirectLoad](#) (const Type *ty)
Load a value of the given type from the address currently on top of the stack.
- void [printIndirectStore](#) (const Value *ptr, const Value *val)
Store a value at the given address.
- void [printIndirectStore](#) (const Type *ty)
Indirectly store a value of the given type.
- std::string [sanitizeName](#) (std::string name)
Replace any non-alphanumeric characters with underscores.
- std::string [getValueName](#) (const Value *v)
Return the name of the given value.
- std::string [getLabelName](#) (const BasicBlock *block)
Return the label name of the given block.
- void [printBinaryInstruction](#) (const char *name, const Value *left, const Value *right)
Print the given binary instruction.
- void [printBinaryInstruction](#) (const std::string &name, const Value *left, const Value *right)
Print the given binary instruction.

- void [printSimpleInstruction](#) (const char *inst)
Print the given instruction.
- void [printSimpleInstruction](#) (const char *inst, const char *operand)
Print the given instruction.
- void [printSimpleInstruction](#) (const std::string &inst)
Print the given instruction.
- void [printSimpleInstruction](#) (const std::string &inst, const std::string &operand)
Print the given instruction.
- void [printVirtualInstruction](#) (const char *sig)
Print the virtual instruction with the given signature.
- void [printVirtualInstruction](#) (const char *sig, const Value *operand)
Print the virtual instruction with the given signature.
- void [printVirtualInstruction](#) (const char *sig, const Value *left, const Value *right)
Print the virtual instruction with the given signature.
- void [printVirtualInstruction](#) (const std::string &sig)
Print the virtual instruction with the given signature.
- void [printVirtualInstruction](#) (const std::string &sig, const Value *operand)
Print the virtual instruction with the given signature.
- void [printVirtualInstruction](#) (const std::string &sig, const Value *left, const Value *right)
Print the virtual instruction with the given signature.
- void [printLabel](#) (const char *label)
Print the given label.
- void [printLabel](#) (const std::string &label)
Print the given label.
- void [printHeader](#) ()
Print the header.
- void [printFields](#) ()
Print the field declarations.
- void [printExternalMethods](#) ()
Print the list of external methods.
- void [printConstructor](#) ()
Print the class constructor.
- void [printCInit](#) ()

Print the static class initialization method.

- void [printMainMethod](#) ()
Print the main method.
- unsigned int [getBitWidth](#) (const Type *ty, bool expand=false)
Return the bit width of the given type.
- char [getTypeID](#) (const Type *ty, bool expand=false)
Return the ID of the given type.
- std::string [getTypeName](#) (const Type *ty, bool expand=false)
Return the name of the given type.
- std::string [getTypeDescriptor](#) (const Type *ty, bool expand=false)
Return the type descriptor of the given type.
- std::string [getTypePostfix](#) (const Type *ty, bool expand=false)
Return the type postfix of the given type.
- std::string [getTypePrefix](#) (const Type *ty, bool expand=false)
Return the type prefix of the given type.

Private Attributes

- formatted_raw_ostream & [out](#)
The output stream.
- std::string [sourcename](#)
The name of the source file.
- std::string [classname](#)
The binary name of the generated class.
- unsigned int [debug](#)
The debugging level.
- Module * [module](#)
The current module.
- const TargetData * [targetData](#)
The target data for the platform.
- DenseSet< const Value * > [externRefs](#)
Set of external references.
- DenseMap< const BasicBlock *, unsigned int > [blockIDs](#)
Mapping of blocks to unique IDs.

- DenseMap< const Value *, unsigned int > [localVars](#)
Mapping of values to local variable numbers.
- unsigned int [usedRegisters](#)
Number of registers allocated for the function.
- unsigned int [vaArgNum](#)
Local variable number of the pointer to the packed list of varargs.
- unsigned int [instNum](#)
Current instruction number.

Static Private Attributes

- static char [id](#) = 0
Pass ID.

2.1.1 Detailed Description

A FunctionPass for generating Jasmin-style assembly for the JVM.

Author:

David Roberts

Definition at line 45 of file backend.h.

2.1.2 Constructor & Destructor Documentation

2.1.2.1 JVMWriter::JVMWriter (const TargetData * *td*, formatted_raw_ostream & *o*, const std::string & *cls*, unsigned int *dbg*)

Constructor.

Parameters:

- td* the target data for the platform
- o* the output stream to be written to
- cls* the binary name of the class to generate
- dbg* the debugging level

Definition at line 35 of file backend.cpp.

```
37 : FunctionPass(&id), targetData(td), out(o), classname(cls), debug(dbg) {}
```

2.1.3 Member Function Documentation

2.1.3.1 `bool JVMWriter::doFinalization (Module & m)` [private]

Perform per-module finalization.

Parameters:

m the module

Returns:

whether the module was modified (always false)

Definition at line 105 of file backend.cpp.

```
105                                     {
106     return false;
107 }
```

2.1.3.2 `bool JVMWriter::doInitialization (Module & m)` [private]

Perform per-module initialization.

Parameters:

m the module

Returns:

whether the module was modified (always false)

Definition at line 68 of file backend.cpp.

References `classname`, `instNum`, `module`, `printCIInit()`, `printConstructor()`, `printExternalMethods()`, `printFields()`, `printHeader()`, `printMainMethod()`, and `sourcename`.

```
68                                     {
69     module = &m;
70     instNum = 0;
71
72     std::string modID = module->getModuleIdentifier();
73     size_t slashPos = modID.rfind('/');
74     if(slashPos == std::string::npos)
75         sourcename = modID;
76     else
77         sourcename = modID.substr(slashPos + 1);
78
79     if(!classname.empty()) {
80         for(std::string::iterator i = classname.begin(),
81             e = classname.end(); i != e; i++)
82             if(*i == '.') *i = '/';
83     } else {
84         classname = sourcename.substr(0, sourcename.rfind('.'));
85         for(std::string::iterator i = classname.begin(),
86             e = classname.end(); i != e; i++)
87             if(*i == '.') *i = '_';
88     }
89 }
```

```

90     printHeader ();
91     printFields ();
92     printExternalMethods ();
93     printConstructor ();
94     printClInit ();
95     printMainMethod ();
96     return false;
97 }

```

2.1.3.3 void JVMWriter::getAnalysisUsage (AnalysisUsage & *au*) const [private]

Register required analysis information.

Parameters:

au AnalysisUsage object representing the analysis usage information of this pass.

Definition at line 45 of file backend.cpp.

```

45                                     {
46     au.addRequired<LoopInfo> ();
47     au.setPreservesAll ();
48 }

```

2.1.3.4 unsigned int JVMWriter::getBitWidth (const Type * *ty*, bool *expand* = false) [private]

Return the bit width of the given type.

Parameters:

ty the type

expand specifies whether to expand the type to 32 bits

Returns:

the bit width

Definition at line 32 of file types.cpp.

Referenced by getLocalVarNumber(), getTypeID(), getTypePostfix(), printArithmeticInstruction(), printCastInstruction(), printMathIntrinsic(), and printValueStore().

```

32                                     {
33     if (ty->getTypeID () == Type::ArrayTyID
34         || ty->getTypeID () == Type::VectorTyID
35         || ty->getTypeID () == Type::StructTyID
36         || ty->getTypeID () == Type::PointerTyID)
37         return 32;
38
39     unsigned int n = ty->getPrimitiveSizeInBits ();
40     switch (n) {
41     case 1:
42     case 8:
43     case 16:
44     case 32: if (expand) return 32;

```

```

45     case 64: return n;
46     default:
47         errs() << "Bits = " << n << '\n';
48         llvm_unreachable("Unsupported integer width");
49     }
50 }

```

2.1.3.5 `std::string JVMWriter::getCallSignature (const FunctionType * ty)` [private]

Return the call signature of the given function type.

An empty string is returned if the function type appears to be non-prototyped.

Parameters:

ty the function type

Returns:

the call signature

Definition at line 42 of file function.cpp.

References `getTypeDescriptor()`.

Referenced by `printExternalMethods()`, `printFunctionCall()`, and `printValueLoad()`.

```

42                                     {
43     if(ty->isVarArg() && ty->getNumParams() == 0)
44         // non-prototyped function
45         return "";
46     std::string sig;
47     sig += '(';
48     for(unsigned int i = 0, e = ty->getNumParams(); i < e; i++)
49         sig += getTypeDescriptor(ty->getParamType(i));
50     if(ty->isVarArg()) sig += "I";
51     sig += ')';
52     sig += getTypeDescriptor(ty->getReturnType());
53     return sig;
54 }

```

2.1.3.6 `std::string JVMWriter::getLabelName (const BasicBlock * block)` [private]

Return the label name of the given block.

Parameters:

block the block

Returns:

the label

Definition at line 61 of file name.cpp.

References `blockIDs`, and `sanitizeName()`.

Referenced by `printBasicBlock()`, `printBranchInstruction()`, `printLoop()`, and `printSwitchInstruction()`.

```

61                                     {
62     if(!blockIDs.count(block))
63         blockIDs[block] = blockIDs.size() + 1;
64     return sanitizeName("label" + toString(blockIDs[block]));
65 }

```

2.1.3.7 unsigned int JVMWriter::getLocalVarNumber (const Value * v) [private]

Return the local variable number of the given value.

Register/s are allocated for the variable if necessary.

Parameters:

v the value

Returns:

the local variable number

Definition at line 253 of file function.cpp.

References `getBitWidth()`, `localVars`, and `usedRegisters`.

Referenced by `getValueName()`, `printFunction()`, `printLocalVariable()`, `printValueLoad()`, and `printValueStore()`.

```

253                                     {
254     if(!localVars.count(v)) {
255         localVars[v] = usedRegisters++;
256         if(getBitWidth(v->getType()) == 64)
257             usedRegisters++; // 64 bit types occupy 2 registers
258     }
259     return localVars[v];
260 }

```

2.1.3.8 std::string JVMWriter::getTypeDescriptor (const Type * ty, bool expand = false) [private]

Return the type descriptor of the given type.

Parameters:

ty the type

expand specifies whether to expand the type to 32 bits

Returns:

the type descriptor

Definition at line 112 of file types.cpp.

References `getTypeID()`.

Referenced by `getCallSignature()`, `printArithmeticInstruction()`, `printBitIntrinsic()`, `printCastInstruction()`, `printCmpInstruction()`, `printFields()`, `printFunction()`, `printFunctionCall()`, `printIndirectLoad()`, `printIndirectStore()`, `printLocalVariable()`, `printMainMethod()`, `printMemIntrinsic()`, `printOperandPack()`, and `printStaticConstant()`.

```

112                                     {
113     return std::string() + getTypeID(ty, expand);
114 }

```

2.1.3.9 char JVMWriter::getTypeID (const Type * ty, bool *expand* = false) [private]

Return the ID of the given type.

Parameters:

ty the type
expand specifies whether to expand the type to 32 bits

Returns:

the type ID

Definition at line 59 of file types.cpp.

References `getBitWidth()`.

Referenced by `getTypeDescriptor()`, `getTypeName()`, `getTypePrefix()`, and `printBitCastInstruction()`.

```

59                                     {
60     switch(ty->getTypeID()) {
61     case Type::VoidTyID:
62         return 'V';
63     case Type::IntegerTyID:
64         switch(getBitWidth(ty, expand)) {
65             case 1: return 'Z';
66             case 8: return 'B';
67             case 16: return 'S';
68             case 32: return 'I';
69             case 64: return 'J';
70         }
71     case Type::FloatTyID:
72         return 'F';
73     case Type::DoubleTyID:
74         return 'D';
75     case Type::PointerTyID:
76     case Type::StructTyID:
77     case Type::ArrayTyID:
78     case Type::VectorTyID:
79         return 'I';
80     default:
81         errs() << "Type = " << *ty << '\n';
82         llvm_unreachable("Invalid type");
83     }
84 }

```

2.1.3.10 std::string JVMWriter::getTypeName (const Type * ty, bool *expand* = false) [private]

Return the name of the given type.

Parameters:

ty the type

expand specifies whether to expand the type to 32 bits

Returns:

the type name

Definition at line 92 of file types.cpp.

References `getTypeID()`.

```

92                                     {
93     switch(getTypeID(ty, expand)) {
94     case 'V': return "void";
95     case 'Z': return "boolean";
96     case 'B': return "byte";
97     case 'S': return "short";
98     case 'I': return "int";
99     case 'J': return "long";
100    case 'F': return "float";
101    case 'D': return "double";
102    }
103 }
```

2.1.3.11 std::string JVMWriter::getTypePostfix (const Type * ty, bool expand = false)
 [private]

Return the type postfix of the given type.

Parameters:

ty the type

expand specifies whether to expand the type to 32 bits

Returns:

the type postfix

Definition at line 123 of file types.cpp.

References `getBitWidth()`.

Referenced by `printCastInstruction()`, `printFunctionCall()`, and `printIndirectLoad()`.

```

123                                     {
124     switch(ty->getTypeID()) {
125     case Type::VoidTyID:
126         return "void";
127     case Type::IntegerTyID:
128         return "i" + utostr(getBitWidth(ty, expand));
129     case Type::FloatTyID:
130         return "f32";
131     case Type::DoubleTyID:
132         return "f64";
133     case Type::PointerTyID:
134     case Type::StructTyID:
135     case Type::ArrayTyID:
136     case Type::VectorTyID:
137         return "i32";
138     default:
139         errs() << "TypeID = " << ty->getTypeID() << '\n';
140         llvm_unreachable("Invalid type");
141     }
142 }
```

2.1.3.12 `std::string JVMWriter::getTypePrefix (const Type * ty, bool expand = false)` [private]

Return the type prefix of the given type.

Parameters:

ty the type
expand specifies whether to expand the type to 32 bits

Returns:

the type prefix

Definition at line 151 of file types.cpp.

References `getTypeID()`.

Referenced by `printArithmeticInstruction()`, `printCastInstruction()`, `printInstruction()`, `printLocalVariable()`, `printValueLoad()`, and `printValueStore()`.

```

151                                     {
152     switch (getTypeID (ty, expand)) {
153     case 'Z':
154     case 'B': return "b";
155     case 'S': return "s";
156     case 'I': return "i";
157     case 'J': return "l";
158     case 'F': return "f";
159     case 'D': return "d";
160     case 'V': llvm_unreachable("void has no prefix");
161     }
162 }
```

2.1.3.13 `std::string JVMWriter::getValueName (const Value * v)` [private]

Return the name of the given value.

Parameters:

v the value

Returns:

the name of the value

Definition at line 46 of file name.cpp.

References `getLocalVarNumber()`, `localVars`, and `sanitizeName()`.

Referenced by `printCInit()`, `printExternalMethods()`, `printFields()`, `printFunction()`, `printFunctionCall()`, `printLocalVariable()`, `printValueLoad()`, and `printValueStore()`.

```

46                                     {
47     if (const GlobalValue *gv = dyn_cast<GlobalValue>(v))
48         return sanitizeName (Mangler (*module) .getMangledName (gv));
49     if (v->hasName ())
50         return '_' + sanitizeName (v->getName ());
51     if (localVars.count (v))
52         return '_' + utostr (getLocalVarNumber (v));
53     return "_";
54 }
```

2.1.3.14 void JVMWriter::printAllocaInstruction (const AllocaInst * *inst*) [private]

Print an alloca instruction.

Parameters:

inst the instruction

Definition at line 290 of file instruction.cpp.

References printPtrLoad(), printSimpleInstruction(), printValueLoad(), and targetData.

Referenced by printInstruction().

```

290                                     {
291     uint64_t size = targetData->getTypeAllocSize(inst->getAllocatedType());
292     if(const ConstantInt *c = dyn_cast<ConstantInt>(inst->getOperand(0))) {
293         // constant optimization
294         printPtrLoad(c->getZExtValue() * size);
295     } else {
296         printPtrLoad(size);
297         printValueLoad(inst->getOperand(0));
298         printSimpleInstruction("imul");
299     }
300     printSimpleInstruction("invokestatic",
301                           "lljvm/runtime/Memory/allocateStack(I)I");
302 }
```

2.1.3.15 void JVMWriter::printArithmeticInstruction (unsigned int *op*, const Value * *left*, const Value * *right*) [private]

Print an arithmetic instruction.

Parameters:

op the opcode for the instruction

left the first operand of the instruction

right the second operand of the instruction

Definition at line 89 of file instruction.cpp.

References getBitWidth(), getTypeDescriptor(), getTypePrefix(), printSimpleInstruction(), printValueLoad(), and printVirtualInstruction().

Referenced by printConstantExpr(), and printInstruction().

```

91                                     {
92     printValueLoad(left);
93     printValueLoad(right);
94     std::string typePrefix = getTypePrefix(left->getType(), true);
95     std::string typeDescriptor = getTypeDescriptor(left->getType());
96     switch(op) {
97     case Instruction::Add:
98     case Instruction::FAdd:
99         printSimpleInstruction(typePrefix + "add"); break;
100    case Instruction::Sub:
101    case Instruction::FSub:
102        printSimpleInstruction(typePrefix + "sub"); break;
103    case Instruction::Mul:
```

```

104     case Instruction::FMul:
105         printSimpleInstruction(typePrefix + "mul"); break;
106     case Instruction::SDiv:
107     case Instruction::FDiv:
108         printSimpleInstruction(typePrefix + "div"); break;
109     case Instruction::SRem:
110     case Instruction::FRem:
111         printSimpleInstruction(typePrefix + "rem"); break;
112     case Instruction::And:
113         printSimpleInstruction(typePrefix + "and"); break;
114     case Instruction::Or:
115         printSimpleInstruction(typePrefix + "or"); break;
116     case Instruction::Xor:
117         printSimpleInstruction(typePrefix + "xor"); break;
118     case Instruction::Shl:
119         if(getBitWidth(right->getType()) == 64) printSimpleInstruction("l2i");
120         printSimpleInstruction(typePrefix + "shl"); break;
121     case Instruction::LShr:
122         if(getBitWidth(right->getType()) == 64) printSimpleInstruction("l2i");
123         printSimpleInstruction(typePrefix + "ushr"); break;
124     case Instruction::AShr:
125         if(getBitWidth(right->getType()) == 64) printSimpleInstruction("l2i");
126         printSimpleInstruction(typePrefix + "shr"); break;
127     case Instruction::UDiv:
128         printVirtualInstruction(
129             "udiv(" + typeDescriptor + typeDescriptor + ")" + typeDescriptor);
130         break;
131     case Instruction::URem:
132         printVirtualInstruction(
133             "urem(" + typeDescriptor + typeDescriptor + ")" + typeDescriptor);
134         break;
135     }
136 }

```

2.1.3.16 void JVMWriter::printBasicBlock (const BasicBlock * *block*) [private]

Print the given basic block.

Parameters:

block the basic block

Definition at line 30 of file block.cpp.

References `debug`, `getLabelName()`, `instNum`, `out`, `printInstruction()`, `printLabel()`, `printSimpleInstruction()`, and `printValueStore()`.

Referenced by `printFunctionBody()`, and `printLoop()`.

```

30                                     {
31     printLabel(getLabelName(block));
32     for(BasicBlock::const_iterator i = block->begin(), e = block->end();
33         i != e; i++) {
34         instNum++;
35         if(debug >= 3) {
36             // print current instruction as comment
37             // note that this block of code significantly increases
38             // code generation time
39             std::string str;
40             raw_string_ostream ss(str); ss << *i;
41             std::string::size_type pos = 0;
42             while((pos = str.find("\n", pos)) != std::string::npos)
43                 str.replace(pos++, 1, "\n;");

```

```

44         out << ';' << str << '\n';
45     }
46     if(debug >= 1)
47         printSimpleInstruction(".line", utostr(instNum));
48
49     if(i->getOpcode() == Instruction::PHI)
50         // don't handle phi instruction in current block
51         continue;
52     printInstruction(i);
53     if(i->getType() != Type::getVoidTy(block->getContext())
54        && i->getOpcode() != Instruction::Invoke)
55         // instruction doesn't return anything, or is an invoke instruction
56         // which handles storing the return value itself
57         printValueStore(i);
58     }
59 }

```

2.1.3.17 void JVMWriter::printBinaryInstruction (const std::string & name, const Value * left, const Value * right) [private]

Print the given binary instruction.

Parameters:

- name* the name of the instruction
- left* the first operand
- right* the second operand

Definition at line 47 of file printinst.cpp.

References out, and printValueLoad().

```

49                                     {
50     printValueLoad(left);
51     printValueLoad(right);
52     out << '\t' << name << '\n';
53 }

```

2.1.3.18 void JVMWriter::printBinaryInstruction (const char * name, const Value * left, const Value * right) [private]

Print the given binary instruction.

Parameters:

- name* the name of the instruction
- left* the first operand
- right* the second operand

Definition at line 32 of file printinst.cpp.

References out, and printValueLoad().

```

34                                     {
35     printValueLoad(left);
36     printValueLoad(right);
37     out << '\t' << name << '\n';
38 }

```

2.1.3.19 void JVMWriter::printBitCastInstruction (const Type * ty, const Type * srcTy) [private]

Print a bitcast instruction.

Parameters:

ty the destination type

srcTy the source type

Definition at line 144 of file instruction.cpp.

References `getTypeID()`, and `printSimpleInstruction()`.

Referenced by `printCastInstruction()`.

```

144                                     {
145     char typeID = getTypeID(ty);
146     char srcTypeID = getTypeID(srcTy);
147     if(srcTypeID == 'J' && typeID == 'D')
148         printSimpleInstruction("invokestatic",
149                               "java/lang/Double/longBitsToDouble(J)D");
150     else if(srcTypeID == 'I' && typeID == 'F')
151         printSimpleInstruction("invokestatic",
152                               "java/lang/Float/intBitsToFloat(I)F");
153     if(srcTypeID == 'D' && typeID == 'J')
154         printSimpleInstruction("invokestatic",
155                               "java/lang/Double/doubleToRawLongBits(D)J");
156     else if(srcTypeID == 'F' && typeID == 'I')
157         printSimpleInstruction("invokestatic",
158                               "java/lang/Float/floatToRawIntBits(F)I");
159 }
```

2.1.3.20 void JVMWriter::printBitIntrinsic (const IntrinsicInst * inst) [private]

Print a bit manipulation intrinsic function.

Parameters:

inst the instruction

Definition at line 427 of file instruction.cpp.

References `getTypeDescriptor()`, and `printVirtualInstruction()`.

Referenced by `printIntrinsicCall()`.

```

427                                     {
428     // TODO: ctpop, ctlz, cttz
429     const Value *value = inst->getOperand(1);
430     const std::string typeDescriptor = getTypeDescriptor(value->getType());
431     switch(inst->getIntrinsicID()) {
432     case Intrinsic::bswap:
433         printVirtualInstruction(
434             "bswap(" + typeDescriptor + ")" + typeDescriptor, value); break;
435     }
436 }
```

2.1.3.21 void JVMWriter::printBranchInstruction (const BranchInst * *inst*) [private]

Print a branch instruction.

Parameters:

inst the branch instruction

Definition at line 105 of file branch.cpp.

References printBranchInstruction(), and printValueLoad().

```

105                                     {
106     if(inst->isUnconditional()) {
107         printBranchInstruction(inst->getParent(), inst->getSuccessor(0));
108     } else {
109         printValueLoad(inst->getCondition());
110         printBranchInstruction(
111             inst->getParent(), inst->getSuccessor(0), inst->getSuccessor(1));
112     }
113 }
```

2.1.3.22 void JVMWriter::printBranchInstruction (const BasicBlock * *curBlock*, const BasicBlock * *trueBlock*, const BasicBlock * *falseBlock*) [private]

Print a conditional branch instruction.

Parameters:

curBlock the current block

trueBlock the destination block if the value on top of the stack is non-zero

falseBlock the destination block if the value on top of the stack is zero

Definition at line 73 of file branch.cpp.

References getLabelName(), printBranchInstruction(), printLabel(), printPHICopy(), and printSimpleInstruction().

```

75                                     {
76     if(trueBlock == falseBlock) {
77         printSimpleInstruction("pop");
78         printBranchInstruction(curBlock, trueBlock);
79     } else if(!falseBlock) {
80         printPHICopy(curBlock, trueBlock);
81         printSimpleInstruction("ifne", getLabelName(trueBlock));
82     } else {
83         std::string labelname = getLabelName(trueBlock);
84         if(isa<PHINode>(trueBlock->begin()))
85             labelname += "$phi" + toString(getUID());
86         printSimpleInstruction("ifne", labelname);
87
88         if(isa<PHINode>(falseBlock->begin()))
89             printPHICopy(curBlock, falseBlock);
90         printSimpleInstruction("goto", getLabelName(falseBlock));
91
92         if(isa<PHINode>(trueBlock->begin())) {
93             printLabel(labelname);
94             printPHICopy(curBlock, trueBlock);
95             printSimpleInstruction("goto", getLabelName(trueBlock));
96         }
97     }
98 }
```

2.1.3.23 void JVMWriter::printBranchInstruction (const BasicBlock * *curBlock*, const BasicBlock * *destBlock*) [private]

Print an unconditional branch instruction.

Parameters:

curBlock the current block

destBlock the destination block

Definition at line 58 of file branch.cpp.

References getLabelName(), printPHICopy(), and printSimpleInstruction().

Referenced by printBranchInstruction(), printInstruction(), and printInvokeInstruction().

```

59                                     {
60     printPHICopy(curBlock, destBlock);
61     printSimpleInstruction("goto", getLabelName(destBlock));
62 }
```

2.1.3.24 void JVMWriter::printCallInstruction (const Instruction * *inst*) [private]

Print a call instruction.

Parameters:

inst the instruction

Definition at line 176 of file function.cpp.

References printFunctionCall(), and printIntrinsicCall().

Referenced by printInstruction().

```

176                                     {
177     if (isa<IntrinsicInst>(inst))
178         printIntrinsicCall(cast<IntrinsicInst>(inst));
179     else
180         printFunctionCall(inst->getOperand(0), inst);
181 }
```

2.1.3.25 void JVMWriter::printCastInstruction (unsigned int *op*, const Value * *v*, const Type * *ty*, const Type * *srcTy*) [private]

Print a cast instruction.

Parameters:

op the opcode for the instruction

v the value to be casted

ty the destination type

srcTy the source type

Definition at line 181 of file instruction.cpp.

References `getBitWidth()`, `getTypeDescriptor()`, `getTypePostfix()`, `getTypePrefix()`, `printBitCastInstruction()`, `printCastInstruction()`, `printSimpleInstruction()`, `printValueLoad()`, and `printVirtualInstruction()`.

```

182                                     {
183     printValueLoad(v);
184     switch(op) {
185     case Instruction::SIToFP:
186     case Instruction::FPToSI:
187     case Instruction::FPTrunc:
188     case Instruction::FPExt:
189     case Instruction::SExt:
190         if(getBitWidth(srcTy) < 32)
191             printCastInstruction(getTypePrefix(srcTy), "i");
192         printCastInstruction(getTypePrefix(ty, true),
193                             getTypePrefix(srcTy, true)); break;
194     case Instruction::Trunc:
195         if(getBitWidth(srcTy) == 64 && getBitWidth(ty) < 32) {
196             printSimpleInstruction("l2i");
197             printCastInstruction(getTypePrefix(ty), "i");
198         } else
199             printCastInstruction(getTypePrefix(ty),
200                                 getTypePrefix(srcTy, true));
201         break;
202     case Instruction::IntToPtr:
203         printCastInstruction("i", getTypePrefix(srcTy, true)); break;
204     case Instruction::PtrToInt:
205         printCastInstruction(getTypePrefix(ty), "i"); break;
206     case Instruction::ZExt:
207         printVirtualInstruction("zext_" + getTypePostfix(ty, true)
208                                 + "(" + getTypeDescriptor(srcTy) + ")"
209                                 + getTypeDescriptor(ty, true));
210         break;
211     case Instruction::UIToFP:
212         printVirtualInstruction("uitofp_" + getTypePostfix(ty)
213                                 + "(" + getTypeDescriptor(srcTy) + ")"
214                                 + getTypeDescriptor(ty));
215         break;
216     case Instruction::FPToUI:
217         printVirtualInstruction("fptoui_" + getTypePostfix(ty)
218                                 + "(" + getTypeDescriptor(srcTy) + ")"
219                                 + getTypeDescriptor(ty));
220         break;
221     case Instruction::BitCast:
222         printBitCastInstruction(ty, srcTy); break;
223     default:
224         errs() << "Opcode = " << op << '\n';
225         llvm_unreachable("Invalid cast instruction");
226     }
227 }

```

2.1.3.26 void JVMWriter::printCastInstruction (const std::string & typePrefix, const std::string & srcTypePrefix) [private]

Print a cast instruction.

Parameters:

typePrefix the type prefix of the destination type

srcTypePrefix the type prefix of the source type

Definition at line 167 of file instruction.cpp.

References `printSimpleInstruction()`.

Referenced by `printCastInstruction()`, `printConstantExpr()`, `printGepInstruction()`, and `printInstruction()`.

```

168                                     {
169     if(srcTypePrefix != typePrefix)
170         printSimpleInstruction(srcTypePrefix + "2" + typePrefix);
171 }
```

2.1.3.27 void JVMWriter::printCatchJump (unsigned int *numJumps*) [private]

Print the block to catch Jump objects (thrown by `longjmp`).

Parameters:

numJumps the number of `setjmp` calls made by the current function

Definition at line 267 of file `function.cpp`.

References `debug`, `printLabel()`, `printSimpleInstruction()`, and `usedRegisters`.

Referenced by `printFunction()`.

```

267                                     {
268     unsigned int jumpVarNum = usedRegisters++;
269     printSimpleInstruction(".catch lljvm/runtime/Jump "
270         "from begin_method to catch_jump using catch_jump");
271     printLabel("catch_jump");
272     printSimpleInstruction("astore", utostr(jumpVarNum));
273     printSimpleInstruction("aload", utostr(jumpVarNum));
274     printSimpleInstruction("getfield", "lljvm/runtime/Jump/value I");
275     for(unsigned int i = usedRegisters-1 - numJumps,
276         e = usedRegisters-1; i < e; i++) {
277         if(debug >= 2)
278             printSimpleInstruction(".var " + utostr(i) + " is setjmp_id_"
279                 + utostr(i) + " I from begin_method to end_method");
280         printSimpleInstruction("aload", utostr(jumpVarNum));
281         printSimpleInstruction("getfield", "lljvm/runtime/Jump/id I");
282         printSimpleInstruction("iload", utostr(i));
283         printSimpleInstruction("if_icmpeq", "setjmp$" + utostr(i));
284     }
285     printSimpleInstruction("pop");
286     printSimpleInstruction("aload", utostr(jumpVarNum));
287     printSimpleInstruction("athrow");
288     if(debug >= 2)
289         printSimpleInstruction(".var " + utostr(jumpVarNum) + " is jump "
290             "Llljvm/runtime/Jump; from begin_method to end_method");
291 }
```

2.1.3.28 void JVMWriter::printCmpInstruction (unsigned int *predicate*, const Value * *left*, const Value * *right*) [private]

Print an `icmp/fcmp` instruction.

Parameters:

predicate the predicate for the instruction

left the first operand of the instruction

right the second operand of the instruction

Definition at line 42 of file instruction.cpp.

References `getTypeDescriptor()`, and `printVirtualInstruction()`.

Referenced by `printConstantExpr()`, and `printInstruction()`.

```

44                                     {
45     std::string inst;
46     switch(predicate) {
47     case ICmpInst::ICMP_EQ:  inst = "icmp_eq";  break;
48     case ICmpInst::ICMP_NE:  inst = "icmp_ne";  break;
49     case ICmpInst::ICMP_ULE: inst = "icmp_ule"; break;
50     case ICmpInst::ICMP_SLE: inst = "icmp_sle"; break;
51     case ICmpInst::ICMP_UGE: inst = "icmp_uge"; break;
52     case ICmpInst::ICMP_SGE: inst = "icmp_sge"; break;
53     case ICmpInst::ICMP_ULT: inst = "icmp_ult"; break;
54     case ICmpInst::ICMP_SLT: inst = "icmp_slt"; break;
55     case ICmpInst::ICMP_UGT: inst = "icmp_ugt"; break;
56     case ICmpInst::ICMP_SGT: inst = "icmp_sgt"; break;
57     case FCmpInst::FCMP_UGT: inst = "fcmp_ugt"; break;
58     case FCmpInst::FCMP_OGT: inst = "fcmp_ogt"; break;
59     case FCmpInst::FCMP_UGE: inst = "fcmp_uge"; break;
60     case FCmpInst::FCMP_OGE: inst = "fcmp_oge"; break;
61     case FCmpInst::FCMP_ULT: inst = "fcmp_ult"; break;
62     case FCmpInst::FCMP_OLT: inst = "fcmp_olt"; break;
63     case FCmpInst::FCMP_ULE: inst = "fcmp_ule"; break;
64     case FCmpInst::FCMP_OLE: inst = "fcmp_ole"; break;
65     case FCmpInst::FCMP_UEQ: inst = "fcmp_ueq"; break;
66     case FCmpInst::FCMP_OEQ: inst = "fcmp_oeq"; break;
67     case FCmpInst::FCMP_UNE: inst = "fcmp_une"; break;
68     case FCmpInst::FCMP_ONE: inst = "fcmp_one"; break;
69     case FCmpInst::FCMP_ORD: inst = "fcmp_ord"; break;
70     case FCmpInst::FCMP_UNO: inst = "fcmp_uno"; break;
71     default:
72         errs() << "Predicate = " << predicate << '\n';
73         llvm_unreachable("Invalid cmp predicate");
74     }
75     printVirtualInstruction(
76         inst + "("
77         + getTypeDescriptor(left->getType(), true)
78         + getTypeDescriptor(right->getType(), true)
79         + ")Z", left, right);
80 }

```

2.1.3.29 void JVMWriter::printConstantExpr (const ConstantExpr * ce) [private]

Print the given constant expression.

Parameters:

ce the constant expression

Definition at line 237 of file const.cpp.

References `printArithmeticInstruction()`, `printCastInstruction()`, `printCmpInstruction()`, `printGepInstruction()`, and `printSelectInstruction()`.

Referenced by `printStaticConstant()`, and `printValueLoad()`.

```

237                                     {
238     const Value *left, *right;
239     if(ce->getNumOperands() >= 1) left = ce->getOperand(0);
240     if(ce->getNumOperands() >= 2) right = ce->getOperand(1);

```

```

241     switch(ce->getOpcode()) {
242     case Instruction::Trunc:
243     case Instruction::ZExt:
244     case Instruction::SExt:
245     case Instruction::FPTrunc:
246     case Instruction::FPExt:
247     case Instruction::UIToFP:
248     case Instruction::SIToFP:
249     case Instruction::FPToUI:
250     case Instruction::FPToSI:
251     case Instruction::PtrToInt:
252     case Instruction::IntToPtr:
253     case Instruction::BitCast:
254         printCastInstruction(ce->getOpcode(), left,
255                             ce->getType(), left->getType()); break;
256     case Instruction::Add:
257     case Instruction::FAdd:
258     case Instruction::Sub:
259     case Instruction::FSub:
260     case Instruction::Mul:
261     case Instruction::FMul:
262     case Instruction::UDiv:
263     case Instruction::SDiv:
264     case Instruction::FDiv:
265     case Instruction::URem:
266     case Instruction::SRem:
267     case Instruction::FRem:
268     case Instruction::And:
269     case Instruction::Or:
270     case Instruction::Xor:
271     case Instruction::Shl:
272     case Instruction::LShr:
273     case Instruction::AShr:
274         printArithmeticInstruction(ce->getOpcode(), left, right); break;
275     case Instruction::ICmp:
276     case Instruction::FCmp:
277         printCmpInstruction(ce->getPredicate(), left, right); break;
278     case Instruction::GetElementPtr:
279         printGepInstruction(ce->getOperand(0),
280                             gep_type_begin(ce),
281                             gep_type_end(ce)); break;
282     case Instruction::Select:
283         printSelectInstruction(ce->getOperand(0),
284                                ce->getOperand(1),
285                                ce->getOperand(2)); break;
286     default:
287         errs() << "Expression = " << *ce << '\n';
288         llvm_unreachable("Invalid constant expression");
289     }
290 }

```

2.1.3.30 void JVMWriter::printConstLoad (const std::string & *str*, bool *cstring*) [private]

Load the given string.

Parameters:

str the string

cstring true iff the string contains a single null character at the end

Definition at line 138 of file const.cpp.

References out.

```

138                                     {
139   out << "\tldc \"";
140   if(cstring)
141     for(std::string::const_iterator i = str.begin(),
142         e = str.end()-1; i != e; i++)
143       switch(*i) {
144         case '\\': out << "\\\\"; break;
145         case '\b': out << "\\b"; break;
146         case '\t': out << "\\t"; break;
147         case '\n': out << "\\n"; break;
148         case '\f': out << "\\f"; break;
149         case '\r': out << "\\r"; break;
150         case '\"': out << "\\\""; break;
151         case '\': out << "\\'"; break;
152         default:  out << *i;    break;
153       }
154   else
155     for(std::string::const_iterator i = str.begin(),
156         e = str.end(); i != e; i++) {
157       const char c = *i;
158       out << "\\u00" << hexdigit((c>>4) & 0xf) << hexdigit(c & 0xf);
159     }
160   out << "\\n";
161 }

```

2.1.3.31 void JVMWriter::printConstLoad (const Constant *c) [private]

Load the given constant.

Parameters:

c the constant

Definition at line 115 of file const.cpp.

References printConstLoad(), and printPtrLoad().

```

115                                     {
116   if(const ConstantInt *i = dyn_cast<ConstantInt>(c)) {
117     printConstLoad(i->getValue());
118   } else if(const ConstantFP *fp = dyn_cast<ConstantFP>(c)) {
119     if(fp->getType()->getTypeID() == Type::FloatTyID)
120       printConstLoad(fp->getValueAPF().convertToFloat());
121     else
122       printConstLoad(fp->getValueAPF().convertToDouble());
123   } else if(isa<UndefValue>(c)) {
124     printPtrLoad(0);
125   } else {
126     errs() << "Constant = " << *c << '\n';
127     llvm_unreachable("Invalid constant value");
128   }
129 }

```

2.1.3.32 void JVMWriter::printConstLoad (double d) [private]

Load the given double-precision floating point value.

Parameters:

d the value

Definition at line 93 of file const.cpp.

References printSimpleInstruction().

```

93                                     {
94     if(d == 0.0)
95         printSimpleInstruction("dconst_0");
96     else if(d == 1.0)
97         printSimpleInstruction("dconst_1");
98     else if(IsNaN(d))
99         printSimpleInstruction("getstatic", "java/lang/Double/NaN D");
100    else if(IsInf(d) > 0)
101        printSimpleInstruction("getstatic",
102                               "java/lang/Double/POSITIVE_INFINITY D");
103    else if(IsInf(d) < 0)
104        printSimpleInstruction("getstatic",
105                               "java/lang/Double/NEGATIVE_INFINITY D");
106    else
107        printSimpleInstruction("ldc2_w", ftostr(d));
108 }
```

2.1.3.33 void JVMWriter::printConstLoad (float *f*) [private]

Load the given single-precision floating point value.

Parameters:

f the value

Definition at line 69 of file const.cpp.

References printSimpleInstruction().

```

69                                     {
70     if(f == 0.0)
71         printSimpleInstruction("fconst_0");
72     else if(f == 1.0)
73         printSimpleInstruction("fconst_1");
74     else if(f == 2.0)
75         printSimpleInstruction("fconst_2");
76     else if(IsNaN(f))
77         printSimpleInstruction("getstatic", "java/lang/Float/NaN F");
78     else if(IsInf(f) > 0)
79         printSimpleInstruction("getstatic",
80                               "java/lang/Float/POSITIVE_INFINITY F");
81     else if(IsInf(f) < 0)
82         printSimpleInstruction("getstatic",
83                               "java/lang/Float/NEGATIVE_INFINITY F");
84     else
85         printSimpleInstruction("ldc", ftostr(f));
86 }
```

2.1.3.34 void JVMWriter::printConstLoad (const APInt & *i*) [private]

Load the given integer.

Parameters:

i the integer

Definition at line 41 of file const.cpp.

References `printSimpleInstruction()`.

Referenced by `printCInit()`, `printConstLoad()`, `printMemIntrinsic()`, `printPtrLoad()`, `printStaticConstant()`, `printVAArgInstruction()`, and `printValueLoad()`.

```

41                                     {
42     if(i.getBitWidth() <= 32) {
43         int64_t value = i.getSExtValue();
44         if(value == -1)
45             printSimpleInstruction("iconst_m1");
46         else if(0 <= value && value <= 5)
47             printSimpleInstruction("iconst_" + i.toString(10, true));
48         else if(-0x80 <= value && value <= 0x7f)
49             printSimpleInstruction("bipush", i.toString(10, true));
50         else if(-0x8000 <= value && value <= 0x7fff)
51             printSimpleInstruction("sipush", i.toString(10, true));
52         else
53             printSimpleInstruction("ldc", i.toString(10, true));
54     } else {
55         if(i == 0)
56             printSimpleInstruction("lconst_0");
57         else if(i == 1)
58             printSimpleInstruction("lconst_1");
59         else
60             printSimpleInstruction("ldc2_w", i.toString(10, true));
61     }
62 }

```

2.1.3.35 void JVMWriter::printFunction (const Function &f) [private]

Print the given function.

Parameters:

f the function

Definition at line 298 of file function.cpp.

References `debug`, `getLocalVarNumber()`, `getTypeDescriptor()`, `getValueName()`, `localVars`, `out`, `printCatchJump()`, `printFunctionBody()`, `printLabel()`, `printLocalVariable()`, `printSimpleInstruction()`, `usedRegisters`, and `vaArgNum`.

Referenced by `runOnFunction()`.

```

298                                     {
299     localVars.clear();
300     usedRegisters = 0;
301
302     out << '\n';
303     out << ".method " << (f.hasLocalLinkage() ? "private " : "public ")
304         << "static " << getValueName(&f) << ' (';
305     for(Function::const_arg_iterator i = f.arg_begin(), e = f.arg_end();
306         i != e; i++)
307         out << getTypeDescriptor(i->getType());
308     if(f.isVarArg())
309         out << "I";
310     out << ')' << getTypeDescriptor(f.getReturnType()) << '\n';
311
312     for(Function::const_arg_iterator i = f.arg_begin(), e = f.arg_end();
313         i != e; i++) {

```

```

314     // getLocalVarNumber must be called at least once in each iteration
315     unsigned int varNum = getLocalVarNumber(i);
316     if(debug >= 2)
317         printSimpleInstruction(".var " + utostr(varNum) + " is "
318             + getValueName(i) + ' ' + getTypeDescriptor(i->getType())
319             + " from begin_method to end_method");
320     }
321     if(f.isVarArg()) {
322         vaArgNum = usedRegisters++;
323         if(debug >= 2)
324             printSimpleInstruction(".var " + utostr(vaArgNum)
325                 + " is varargptr I from begin_method to end_method");
326     }
327
328     // TODO: better stack depth analysis
329     unsigned int stackDepth = 8;
330     unsigned int numJumps = 0;
331     for(const_inst_iterator i = inst_begin(&f), e = inst_end(&f);
332         i != e; i++) {
333         if(stackDepth < i->getNumOperands())
334             stackDepth = i->getNumOperands();
335         if(i->getType() != Type::getVoidTy(f.getContext()))
336             printLocalVariable(f, &*i);
337         if(const CallInst *inst = dyn_cast<CallInst>(&*i))
338             if(!isa<IntrinsicInst>(inst)
339                 && getValueName(inst->getOperand(0)) == "setjmp")
340                 numJumps++;
341     }
342
343     for(unsigned int i = 0; i < numJumps; i++) {
344         // initialise jump IDs to prevent class verification errors
345         printSimpleInstruction("iconst_0");
346         printSimpleInstruction("istore", utostr(usedRegisters + i));
347     }
348
349     printLabel("begin_method");
350     printSimpleInstruction("invokestatic",
351         "lljvm/runtime/Memory/createStackFrame()V");
352     printFunctionBody(f);
353     if(numJumps) printCatchJump(numJumps);
354     printSimpleInstruction(".limit stack", utostr(stackDepth * 2));
355     printSimpleInstruction(".limit locals", utostr(usedRegisters));
356     printLabel("end_method");
357     out << ".end method\n";
358 }

```

2.1.3.36 void JVMWriter::printFunctionBody (const Function &f) [private]

Print the body of the given function.

Parameters:

f the function

Definition at line 236 of file function.cpp.

References printBasicBlock(), and printLoop().

Referenced by printFunction().

```

236         {
237     for(Function::const_iterator i = f.begin(), e = f.end(); i != e; i++) {
238         if(Loop *l = getAnalysis<LoopInfo>().getLoopFor(i)) {

```

```

239         if(l->getHeader() == i && l->getParentLoop() == 0)
240             printLoop(l);
241     } else
242         printBasicBlock(i);
243     }
244 }

```

2.1.3.37 void JVMWriter::printFunctionCall (const Value *functionVal, const Instruction *inst) [private]

Print a call/invoke instruction.

Parameters:

functionVal the function to call

inst the instruction

Definition at line 93 of file function.cpp.

References `classname`, `externRefs`, `getCallSignature()`, `getTypeDescriptor()`, `getTypePostfix()`, `getValueName()`, `printLabel()`, `printOperandPack()`, `printSimpleInstruction()`, `printValueLoad()`, and `usedRegisters`.

Referenced by `printCallInstruction()`, and `printInvokeInstruction()`.

```

94                                     {
95     unsigned int origin = isa<InvokeInst>(inst) ? 3 : 1;
96     if(const Function *f = dyn_cast<Function>(functionVal)) { // direct call
97         const FunctionType *ty = f->getFunctionType();
98
99         //for(unsigned int i = origin, e = inst->getNumOperands(); i < e; i++)
100        //    printValueLoad(inst->getOperand(i));
101
102        for(unsigned int i = 0, e = ty->getNumParams(); i < e; i++)
103            printValueLoad(inst->getOperand(i + origin));
104        if(ty->isVarArg() && inst)
105            printOperandPack(inst, ty->getNumParams() + origin,
106                            inst->getNumOperands());
107
108        if(externRefs.count(f))
109            printSimpleInstruction("invokestatic",
110                                  getValueName(f) + getCallSignature(ty));
111        else
112            printSimpleInstruction("invokestatic",
113                                  classname + "/" + getValueName(f) + getCallSignature(ty));
114
115        if(getValueName(f) == "setjmp") {
116            unsigned int varNum = usedRegisters++;
117            printSimpleInstruction("istore", toString(varNum));
118            printSimpleInstruction("iconst_0");
119            printLabel("setjmp$" + toString(varNum));
120        }
121    } else { // indirect call
122        printValueLoad(functionVal);
123        const FunctionType *ty = cast<FunctionType>(
124            cast<PointerType>(functionVal->getType())->getElementType());
125        printOperandPack(inst, origin, inst->getNumOperands());
126        printSimpleInstruction("invokestatic",
127                                "lljvm/runtime/Function/invoke_"
128                                + getTypePostfix(ty->getReturnType()) + "(II)"
129                                + getTypeDescriptor(ty->getReturnType()));
130    }
131 }

```

2.1.3.38 void JVMWriter::printGepInstruction (const Value * v, gep_type_iterator i, gep_type_iterator e) [private]

Print a getelementptr instruction.

Parameters:

- v* the aggregate data structure to index
- i* an iterator to the first type indexed by the instruction
- e* an iterator specifying the upper bound on the types indexed by the instruction

Definition at line 235 of file instruction.cpp.

References printCastInstruction(), printPtrLoad(), printSimpleInstruction(), and targetData.

Referenced by printConstantExpr(), and printInstruction().

```

237                                     {
238     // load address
239     printCastInstruction(Instruction::IntToPtr, v, NULL, v->getType());
240
241     // calculate offset
242     for(; i != e; i++){
243         unsigned int size = 0;
244         const Value *indexValue = i.getOperand();
245
246         if(const StructType *structTy = dyn_cast<StructType>(*i)) {
247             for(unsigned int f = 0,
248                 fieldIndex = cast<ConstantInt>(indexValue)->getZExtValue();
249                 f < fieldIndex; f++)
250                 size = alignOffset (
251                     size + targetData->getTypeAllocSize (
252                         structTy->getContainedType(f)),
253                     targetData->getABITypeAlignment (
254                         structTy->getContainedType(f + 1)));
255             printPtrLoad(size);
256             printSimpleInstruction("iadd");
257         } else {
258             if(const SequentialType *seqTy = dyn_cast<SequentialType>(*i))
259                 size = targetData->getTypeAllocSize(seqTy->getElementType());
260             else
261                 size = targetData->getTypeAllocSize(*i);
262
263             if(const ConstantInt *c = dyn_cast<ConstantInt>(indexValue)) {
264                 // constant optimisation
265                 if(c->isNullValue()) {
266                     // do nothing
267                 } else if(c->getValue().isNegative()) {
268                     printPtrLoad(c->getValue().abs().getZExtValue() * size);
269                     printSimpleInstruction("isub");
270                 } else {
271                     printPtrLoad(c->getZExtValue() * size);
272                     printSimpleInstruction("iadd");
273                 }
274             } else {
275                 printPtrLoad(size);
276                 printCastInstruction(Instruction::IntToPtr, indexValue,
277                                     NULL, indexValue->getType());
278                 printSimpleInstruction("imul");
279                 printSimpleInstruction("iadd");
280             }
281         }
282     }
283 }

```

2.1.3.39 void JVMWriter::printIndirectLoad (const Type * ty) [private]

Load a value of the given type from the address curenly on top of the stack.

Parameters:

ty the type of the value

Definition at line 120 of file loadstore.cpp.

References getTypeDescriptor(), getTypePostfix(), and printSimpleInstruction().

```

120                                     {
121     printSimpleInstruction("invokestatic", "lljvm/runtime/Memory/load_"
122         + getTypePostfix(ty) + "(I)" + getTypeDescriptor(ty));
123 }
```

2.1.3.40 void JVMWriter::printIndirectLoad (const Value * v) [private]

Load a value from the given address.

Parameters:

v the address

Definition at line 106 of file loadstore.cpp.

References printValueLoad().

Referenced by printInstruction(), printVArgInstruction(), and printVAIntrinsic().

```

106                                     {
107     printValueLoad(v);
108     const Type *ty = v->getType();
109     if(const PointerType *p = dyn_cast<PointerType>(ty))
110         ty = p->getElementType();
111     printIndirectLoad(ty);
112 }
```

2.1.3.41 void JVMWriter::printIndirectStore (const Type * ty) [private]

Indirectly store a value of the given type.

Parameters:

ty the type of the value

Definition at line 142 of file loadstore.cpp.

References getTypeDescriptor(), and printSimpleInstruction().

```

142                                     {
143     printSimpleInstruction("invokestatic",
144         "lljvm/runtime/Memory/store(I" + getTypeDescriptor(ty) + ")V");
145 }
```

2.1.3.42 void JVMWriter::printIndirectStore (const Value * ptr, const Value * val) [private]

Store a value at the given address.

Parameters:

- ptr* the address at which to store the value
- val* the value to store

Definition at line 131 of file loadstore.cpp.

References printValueLoad().

Referenced by printInstruction(), printVArgInstruction(), and printVAIntrinsic().

```

131                                     {
132     printValueLoad(ptr);
133     printValueLoad(val);
134     printIndirectStore(val->getType());
135 }
```

2.1.3.43 void JVMWriter::printInstruction (const Instruction * inst) [private]

Print the given instruction.

Parameters:

- inst* the instruction

Definition at line 66 of file block.cpp.

References getTypePrefix(), printAllocInstruction(), printArithmeticInstruction(), printBranchInstruction(), printCallInstruction(), printCastInstruction(), printCmpInstruction(), printGepInstruction(), printIndirectLoad(), printIndirectStore(), printInvokeInstruction(), printMallocInstruction(), printSelectInstruction(), printSimpleInstruction(), printSwitchInstruction(), printVArgInstruction(), and printValueLoad().

Referenced by printBasicBlock().

```

66                                     {
67     const Value *left, *right;
68     if(inst->getNumOperands() >= 1) left = inst->getOperand(0);
69     if(inst->getNumOperands() >= 2) right = inst->getOperand(1);
70     switch(inst->getOpcode()) {
71     case Instruction::Ret:
72         printSimpleInstruction("invokestatic",
73                               "lljvm/runtime/Memory/destroyStackFrame(V)");
74         if(inst->getNumOperands() >= 1) {
75             printValueLoad(left);
76             printSimpleInstruction(
77                 getTypePrefix(left->getType(), true) + "return");
78         } else {
79             printSimpleInstruction("return");
80         }
81         break;
82     case Instruction::Unwind:
83         printSimpleInstruction("getstatic",
84                               "lljvm/runtime/Instruction$Unwind/instance "
85                               "Llljvm/runtime/Instruction$Unwind;");
```

```

86     printSimpleInstruction("athrow");
87     // TODO: need to destroy stack frames
88     break;
89 case Instruction::Unreachable:
90     printSimpleInstruction("getstatic",
91         "lljvm/runtime/Instruction$Unreachable/instance "
92         "Llljvm/runtime/Instruction$Unreachable;");
93     printSimpleInstruction("athrow");
94     break;
95 case Instruction::Free:
96     // TODO: lowering pass? <http://llvm.org/docs/Passes.html#lowerallocs>
97     printValueLoad(inst->getOperand(0));
98     printSimpleInstruction("invokestatic", "lljvm/lib/c/free(I)V");
99     break;
100 case Instruction::Add:
101 case Instruction::FAdd:
102 case Instruction::Sub:
103 case Instruction::FSub:
104 case Instruction::Mul:
105 case Instruction::FMul:
106 case Instruction::UDiv:
107 case Instruction::SDiv:
108 case Instruction::FDiv:
109 case Instruction::URem:
110 case Instruction::SRem:
111 case Instruction::FRem:
112 case Instruction::And:
113 case Instruction::Or:
114 case Instruction::Xor:
115 case Instruction::Shl:
116 case Instruction::LShr:
117 case Instruction::AShr:
118     printArithmeticInstruction(inst->getOpcode(), left, right);
119     break;
120 case Instruction::SExt:
121 case Instruction::Trunc:
122 case Instruction::ZExt:
123 case Instruction::FPTrunc:
124 case Instruction::FPExt:
125 case Instruction::UIToFP:
126 case Instruction::SIToFP:
127 case Instruction::FPToUI:
128 case Instruction::FPToSI:
129 case Instruction::PtrToInt:
130 case Instruction::IntToPtr:
131 case Instruction::BitCast:
132     printCastInstruction(inst->getOpcode(), left,
133         cast<CastInst>(inst)->getDestTy(),
134         cast<CastInst>(inst)->getSrcTy()); break;
135 case Instruction::ICmp:
136 case Instruction::FCmp:
137     printCmpInstruction(cast<CmpInst>(inst)->getPredicate(),
138         left, right); break;
139 case Instruction::Malloc:
140     // TODO: lowering pass? <http://llvm.org/docs/Passes.html#lowerallocs>
141     printMallocInstruction(cast<MallocInst>(inst)); break;
142 case Instruction::Br:
143     printBranchInstruction(cast<BranchInst>(inst)); break;
144 case Instruction::Select:
145     printSelectInstruction(inst->getOperand(0),
146         inst->getOperand(1),
147         inst->getOperand(2)); break;
148 case Instruction::Load:
149     printIndirectLoad(inst->getOperand(0)); break;
150 case Instruction::Store:
151     printIndirectStore(inst->getOperand(1), inst->getOperand(0)); break;
152 case Instruction::GetElementPtr:

```

```

153     printGepInstruction(inst->getOperand(0),
154                       gep_type_begin(inst),
155                       gep_type_end(inst)); break;
156 case Instruction::Call:
157     printCallInstruction(cast<CallInst>(inst)); break;
158 case Instruction::Invoke:
159     printInvokeInstruction(cast<InvokeInst>(inst)); break;
160 case Instruction::Switch:
161     printSwitchInstruction(cast<SwitchInst>(inst)); break;
162 case Instruction::Alloca:
163     printAllocaInstruction(cast<AllocaInst>(inst)); break;
164 case Instruction::VAArg:
165     printVAArgInstruction(cast<VAArgInst>(inst)); break;
166 default:
167     errs() << "Instruction = " << *inst << '\n';
168     llvm_unreachable("Unsupported instruction");
169 }
170 }

```

2.1.3.44 void JVMWriter::printIntrinsicCall (const IntrinsicInst * *inst*) [private]

Print a call to an intrinsic function.

Parameters:

inst the instruction

Definition at line 138 of file function.cpp.

References printBitIntrinsic(), printMathIntrinsic(), printMemIntrinsic(), printSimpleInstruction(), and printVAIntrinsic().

Referenced by printCallInstruction().

```

138                                     {
139     switch(inst->getIntrinsicID()) {
140     case Intrinsic::vastart:
141     case Intrinsic::vacopy:
142     case Intrinsic::vaend:
143         printVAIntrinsic(inst); break;
144     case Intrinsic::memcpy:
145     case Intrinsic::memmove:
146     case Intrinsic::memset:
147         printMemIntrinsic(cast<MemIntrinsic>(inst)); break;
148     case Intrinsic::flt_rounds:
149         printSimpleInstruction("iconst_m1"); break;
150     case Intrinsic::dbg_declare:
151     case Intrinsic::dbg_func_start:
152     case Intrinsic::dbg_region_end:
153     case Intrinsic::dbg_region_start:
154     case Intrinsic::dbg_stoppoint:
155         // ignore debugging intrinsics
156         break;
157     case Intrinsic::pow:
158     case Intrinsic::exp:
159     case Intrinsic::log10:
160     case Intrinsic::log:
161     case Intrinsic::sqrt:
162         printMathIntrinsic(inst); break;
163     case Intrinsic::bswap:
164         printBitIntrinsic(inst); break;
165     default:
166         errs() << "Intrinsic = " << *inst << '\n';

```

```

167         llvm_unreachable("Invalid intrinsic function");
168     }
169 }

```

2.1.3.45 void JVMWriter::printInvokeInstruction (const InvokeInst * *inst*) [private]

Print an invoke instruction.

Parameters:

inst the instruction

Definition at line 188 of file function.cpp.

References printBranchInstruction(), printFunctionCall(), printLabel(), printSimpleInstruction(), and printValueStore().

Referenced by printInstruction().

```

188                                     {
189     std::string labelname = getUID() + "$invoke";
190     printLabel(labelname + "_begin");
191     printFunctionCall(inst->getOperand(0), inst);
192     printValueStore(inst); // save return value
193     printLabel(labelname + "_end");
194     printBranchInstruction(inst->getParent(), inst->getNormalDest());
195     printLabel(labelname + "_catch");
196     printSimpleInstruction("pop");
197     printBranchInstruction(inst->getParent(), inst->getUnwindDest());
198     printSimpleInstruction(".catch lljvm/runtime/System$Unwind",
199         "from " + labelname + "_begin "
200         + "to " + labelname + "_end "
201         + "using " + labelname + "_catch");
202 }

```

2.1.3.46 void JVMWriter::printLabel (const std::string & *label*) [private]

Print the given label.

Parameters:

label the label

Definition at line 180 of file printinst.cpp.

References out.

```

180                                     {
181     out << label << ":\n";
182 }

```

2.1.3.47 void JVMWriter::printLabel (const char * *label*) [private]

Print the given label.

Parameters:

label the label

Definition at line 171 of file printinst.cpp.

References out.

Referenced by printBasicBlock(), printBranchInstruction(), printCatchJump(), printFunction(), printFunctionCall(), printInvokeInstruction(), printLoop(), and printSelectInstruction().

```
171                                     {
172     out << label << ":\n";
173 }
```

2.1.3.48 void JVMWriter::printLocalVariable (const Function &f, const Instruction *inst) [private]

Allocate a local variable for the given function.

Variable initialisation and any applicable debugging information is printed.

Parameters:

f the parent function of the variable

inst the instruction assigned to the variable

Definition at line 211 of file function.cpp.

References debug, getLocalVarNumber(), getTypeDescriptor(), getTypePrefix(), getValueName(), and printSimpleInstruction().

Referenced by printFunction().

```
212                                     {
213     const Type *ty;
214     if(isa<AllocaInst>(inst) && !isa<GlobalVariable>(inst))
215         // local variable allocation
216         ty = PointerType::getUnqual(
217             cast<AllocaInst>(inst)->getAllocatedType());
218     else // operation result
219         ty = inst->getType();
220     // getLocalVarNumber must be called at least once in this method
221     unsigned int varNum = getLocalVarNumber(inst);
222     if(debug >= 2)
223         printSimpleInstruction(".var " + utostr(varNum) + " is "
224             + getValueName(inst) + ' ' + getTypeDescriptor(ty)
225             + " from begin_method to end_method");
226     // initialise variable to avoid class verification errors
227     printSimpleInstruction(getTypePrefix(ty, true) + "const_0");
228     printSimpleInstruction(getTypePrefix(ty, true) + "store", utostr(varNum));
229 }
```

2.1.3.49 void JVMWriter::printLoop (const Loop *l) [private]

Print a loop.

Parameters:*l* the loop

Definition at line 169 of file branch.cpp.

References `getLabelName()`, `printBasicBlock()`, `printLabel()`, and `printSimpleInstruction()`.Referenced by `printFunctionBody()`.

```

169                                     {
170     printLabel(getLabelName(l->getHeader()));
171     for(Loop::block_iterator i = l->block_begin(),
172         e = l->block_end(); i != e; i++) {
173         const BasicBlock *block = *i;
174         Loop *blockLoop = getAnalysis<LoopInfo>().getLoopFor(block);
175         if(l == blockLoop)
176             // the loop is the innermost parent of this block
177             printBasicBlock(block);
178         else if(block == blockLoop->getHeader()
179             && l == blockLoop->getParentLoop())
180             // this block is the header of its innermost parent loop,
181             // and the loop is the parent of that loop
182             printLoop(blockLoop);
183     }
184     printSimpleInstruction("goto", getLabelName(l->getHeader()));
185 }

```

2.1.3.50 void JVMWriter::printMallocInstruction (const MallocInst * *inst*) [private]

Print a malloc instruction.

Parameters:*inst* the instruction

Definition at line 382 of file instruction.cpp.

References `printPtrLoad()`, `printSimpleInstruction()`, `printValueLoad()`, and `targetData`.Referenced by `printInstruction()`.

```

382                                     {
383     printPtrLoad(targetData->getTypeAllocSize(inst->getAllocatedType()));
384     printValueLoad(inst->getArraySize());
385     printSimpleInstruction("imul");
386     printSimpleInstruction("invokestatic", "lljvm/lib/c/malloc(I)I");
387 }

```

2.1.3.51 void JVMWriter::printMathIntrinsic (const IntrinsicInst * *inst*) [private]

Print a mathematical intrinsic function.

Parameters:*inst* the instruction

Definition at line 394 of file instruction.cpp.

References `getBitWidth()`, `printSimpleInstruction()`, and `printValueLoad()`.Referenced by `printIntrinsicCall()`.

```

394                                     {
395     bool f32 = (getBitWidth(inst->getOperand(1)->getType()) == 32);
396     printValueLoad(inst->getOperand(1));
397     if(f32) printSimpleInstruction("f2d");
398     if(inst->getNumOperands() >= 3) {
399         printValueLoad(inst->getOperand(2));
400         if(f32) printSimpleInstruction("f2d");
401     }
402     switch(inst->getIntrinsicID()) {
403     case Intrinsic::exp:
404         printSimpleInstruction("invokestatic", "java/lang/Math/exp(D)D");
405         break;
406     case Intrinsic::log:
407         printSimpleInstruction("invokestatic", "java/lang/Math/log(D)D");
408         break;
409     case Intrinsic::log10:
410         printSimpleInstruction("invokestatic", "java/lang/Math/log10(D)D");
411         break;
412     case Intrinsic::sqrt:
413         printSimpleInstruction("invokestatic", "java/lang/Math/sqrt(D)D");
414         break;
415     case Intrinsic::pow:
416         printSimpleInstruction("invokestatic", "java/lang/Math/pow(DD)D");
417         break;
418     }
419     if(f32) printSimpleInstruction("d2f");
420 }

```

2.1.3.52 void JVMWriter::printMemIntrinsic (const MemIntrinsic * *inst*) [private]

Print a memory intrinsic function.

Parameters:

inst the instruction

Definition at line 353 of file instruction.cpp.

References `getTypeDescriptor()`, `printConstLoad()`, `printSimpleInstruction()`, and `printValueLoad()`.

Referenced by `printIntrinsicCall()`.

```

353                                     {
354     printValueLoad(inst->getDest());
355     if(const MemTransferInst *minst = dyn_cast<MemTransferInst>(inst))
356         printValueLoad(minst->getSource());
357     else if(const MemSetInst *minst = dyn_cast<MemSetInst>(inst))
358         printValueLoad(minst->getValue());
359     printValueLoad(inst->getLength());
360     printConstLoad(inst->getAlignmentCst());
361
362     std::string lenDescriptor = getTypeDescriptor(
363         inst->getLength()->getType(), true);
364     switch(inst->getIntrinsicID()) {
365     case Intrinsic::memcpy:
366         printSimpleInstruction("invokestatic",
367             "lljvm/runtime/Memory/memcpy(II" + lenDescriptor + "I)V"); break;
368     case Intrinsic::memmove:
369         printSimpleInstruction("invokestatic",
370             "lljvm/runtime/Memory/memmove(II" + lenDescriptor + "I)V"); break;
371     case Intrinsic::memset:
372         printSimpleInstruction("invokestatic",
373             "lljvm/runtime/Memory/memset(IB" + lenDescriptor + "I)V"); break;

```

```

374     }
375 }

```

2.1.3.53 void JVMWriter::printOperandPack (const Instruction * *inst*, unsigned int *minOperand*, unsigned int *maxOperand*) [private]

Pack the specified operands of the given instruction into memory.

The address of the packed values is left on the top of the stack.

Parameters:

inst the given instruction

minOperand the lower bound on the operands to pack (inclusive)

maxOperand the upper bound on the operands to pack (exclusive)

Definition at line 64 of file function.cpp.

References getTypeDescriptor(), printSimpleInstruction(), printValueLoad(), and targetData.

Referenced by printFunctionCall().

```

66                                     {
67     unsigned int size = 0;
68     for(unsigned int i = minOperand; i < maxOperand; i++)
69         size += targetData->getTypeAllocSize(
70             inst->getOperand(i)->getType());
71
72     printSimpleInstruction("bipush", utostr(size));
73     printSimpleInstruction("invokestatic",
74         "lljvm/runtime/Memory/allocateStack(I)I");
75     printSimpleInstruction("dup");
76
77     for(unsigned int i = minOperand; i < maxOperand; i++) {
78         const Value *v = inst->getOperand(i);
79         printValueLoad(v);
80         printSimpleInstruction("invokestatic",
81             "lljvm/runtime/Memory/pack(I"
82             + getTypeDescriptor(v->getType()) + ")I");
83     }
84     printSimpleInstruction("pop");
85 }

```

2.1.3.54 void JVMWriter::printPHICopy (const BasicBlock * *src*, const BasicBlock * *dest*) [private]

Replace PHI instructions with copy instructions (load-store pairs).

Parameters:

src the predecessor block

dest the destination block

Definition at line 41 of file branch.cpp.

References printValueLoad(), and printValueStore().

Referenced by printBranchInstruction().

```

41
42     for(BasicBlock::const_iterator i = dest->begin(); isa<PHINode>(i); i++) {
43         const PHINode *phi = cast<PHINode>(i);
44         const Value *val = phi->getIncomingValueForBlock(src);
45         if(isa<UndefValue>(val))
46             continue;
47         printValueLoad(val);
48         printValueStore(phi);
49     }
50 }

```

2.1.3.55 void JVMWriter::printPtrLoad (uint64_t n) [private]

Load the given pointer.

Parameters:

n the value of the pointer

Definition at line 30 of file const.cpp.

References module, and printConstLoad().

Referenced by printAllocaInstruction(), printConstLoad(), printGepInstruction(), printMallocInstruction(), printStaticConstant(), and printValueLoad().

```

30
31     if(module->getPointerSize() != Module::Pointer32)
32         llvm_unreachable("Only 32-bit pointers are allowed");
33     printConstLoad(APInt(32, n, false));
34 }

```

2.1.3.56 void JVMWriter::printSelectInstruction (const Value *cond, const Value *trueVal, const Value *falseVal) [private]

Print a select instruction.

Parameters:

cond the condition

trueVal the return value of the instruction if the condition is non-zero

falseVal the return value of the instruction if the condition is zero

Definition at line 124 of file branch.cpp.

References printLabel(), printSimpleInstruction(), and printValueLoad().

Referenced by printConstantExpr(), and printInstruction().

```

126
127     std::string labelname = "select" + utostr(getUID());
128     printValueLoad(cond);
129     printSimpleInstruction("ifeq", labelname + "a");
130     printValueLoad(trueVal);
131     printSimpleInstruction("goto", labelname + "b");
132     printLabel(labelname + "a");
133     printValueLoad(falseVal);
134     printLabel(labelname + "b");
135 }

```

2.1.3.57 void JVMWriter::printSimpleInstruction (const std::string & *inst*, const std::string & *operand*) [private]

Print the given instruction.

Parameters:

inst the instruction

operand the operand to the instruction

Definition at line 89 of file printinst.cpp.

References out.

```
90                                     {
91     out << '\t' << inst << ' ' << operand << '\n';
92 }
```

2.1.3.58 void JVMWriter::printSimpleInstruction (const std::string & *inst*) [private]

Print the given instruction.

Parameters:

inst the instruction

Definition at line 79 of file printinst.cpp.

References out.

```
79                                     {
80     out << '\t' << inst << '\n';
81 }
```

2.1.3.59 void JVMWriter::printSimpleInstruction (const char * *inst*, const char * *operand*) [private]

Print the given instruction.

Parameters:

inst the instruction

operand the operand to the instruction

Definition at line 70 of file printinst.cpp.

References out.

```
70                                     {
71     out << '\t' << inst << ' ' << operand << '\n';
72 }
```

2.1.3.60 void JVMWriter::printSimpleInstruction (const char * *inst*) [private]

Print the given instruction.

Parameters:

inst the instruction

Definition at line 60 of file printinst.cpp.

References out.

Referenced by printAllocInstruction(), printArithmeticInstruction(), printBasicBlock(), printBitCastInstruction(), printBranchInstruction(), printCastInstruction(), printCatchJump(), printCInit(), printConstLoad(), printFunction(), printFunctionCall(), printGepInstruction(), printIndirectLoad(), printIndirectStore(), printInstruction(), printIntrinsicCall(), printInvokeInstruction(), printLocalVariable(), printLoop(), printMainMethod(), printMallocInstruction(), printMathIntrinsic(), printMemIntrinsic(), printOperandPack(), printSelectInstruction(), printStaticConstant(), printVAArgInstruction(), printVAIntrinsic(), printValueLoad(), and printValueStore().

```
60                                     {
61     out << '\t' << inst << '\n';
62 }
```

2.1.3.61 void JVMWriter::printStaticConstant (const Constant * *c*) [private]

Store the given static constant.

The constant is stored to the address currently on top of the stack, pushing the first address following the constant onto the stack afterwards.

Parameters:

c the constant

Definition at line 170 of file const.cpp.

References getTypeDescriptor(), printConstantExpr(), printConstLoad(), printPtrLoad(), printSimpleInstruction(), printValueLoad(), and targetData.

Referenced by printCInit().

```
170                                     {
171     if (isa<ConstantAggregateZero>(c) || c->isNullValue()) {
172         // zero initialised constant
173         printPtrLoad(targetData->getTypeAllocSize(c->getType()));
174         printSimpleInstruction("invokestatic",
175                               "lljvm/runtime/Memory/zero(II)I");
176         return;
177     }
178     std::string typeDescriptor = getTypeDescriptor(c->getType());
179     switch(c->getType()->getTypeID()) {
180     case Type::IntegerTyID:
181     case Type::FloatTyID:
182     case Type::DoubleTyID:
183         printConstLoad(c);
184         printSimpleInstruction("invokestatic",
185                               "lljvm/runtime/Memory/pack(I" + typeDescriptor + ")I");
186         break;
```

```

187     case Type::ArrayTyID:
188         if(const ConstantArray *ca = dyn_cast<ConstantArray>(c))
189             if(ca->isString()) {
190                 bool cstring = ca->isCString();
191                 printConstLoad(ca->getAsString(), cstring);
192                 if(cstring)
193                     printSimpleInstruction("invokestatic",
194                                             "lljvm/runtime/Memory/pack(ILjava/lang/String;)I");
195                 else {
196                     printSimpleInstruction("invokevirtual",
197                                             "java/lang/String/toCharArray()[C];
198                     printSimpleInstruction("invokestatic",
199                                             "lljvm/runtime/Memory/pack(I[C]I");
200                 }
201                 break;
202             }
203         // else fall through
204     case Type::VectorTyID:
205     case Type::StructTyID:
206         for(unsigned int i = 0, e = c->getNumOperands(); i < e; i++)
207             printStaticConstant(cast<Constant>(c->getOperand(i)));
208         break;
209     case Type::PointerTyID:
210         if(const Function *f = dyn_cast<Function>(c))
211             printValueLoad(f);
212         else if(const GlobalVariable *g = dyn_cast<GlobalVariable>(c))
213             // initialise with address of global variable
214             printValueLoad(g);
215         else if(isa<ConstantPointerNull>(c) || c->isNullValue())
216             printSimpleInstruction("iconst_0");
217         else if(const ConstantExpr *ce = dyn_cast<ConstantExpr>(c))
218             printConstantExpr(ce);
219         else {
220             errs() << "Constant = " << *c << '\n';
221             llvm_unreachable("Invalid static initializer");
222         }
223         printSimpleInstruction("invokestatic",
224                                 "lljvm/runtime/Memory/pack(I" + typeDescriptor + ")I");
225         break;
226     default:
227         errs() << "TypeID = " << c->getType()->getTypeID() << '\n';
228         llvm_unreachable("Invalid type in printStaticConstant()");
229     }
230 }

```

2.1.3.62 void JVMWriter::printSwitchInstruction (const SwitchInst * *inst*) [private]

Print a switch instruction.

Parameters:

inst the switch instruction

Definition at line 142 of file branch.cpp.

References [getLabelName\(\)](#), [out](#), and [printValueLoad\(\)](#).

Referenced by [printInstruction\(\)](#).

```

142     {
143         // TODO: This method does not handle switch statements when the
144         // successor contains phi instructions (the value of the phi instruction
145         // should be set before branching to the successor). Therefore, it has

```

```

146 // been replaced by the switch lowering pass. Once this method is
147 // fixed the switch lowering pass should be removed.
148
149 std::map<int, unsigned int> cases;
150 for(unsigned int i = 1, e = inst->getNumCases(); i < e; i++)
151     cases[(int) inst->getCaseValue(i)->getValue().getSExtValue()] = i;
152
153 // TODO: tableswitch in cases where it won't increase the size of the
154 //       class file
155 printValueLoad(inst->getCondition());
156 out << "\tlookupswitch\n";
157 for(std::map<int, unsigned int>::const_iterator
158     i = cases.begin(), e = cases.end(); i != e; i++)
159     out << "\t\t" << i->first << " : "
160         << getLabelName(inst->getSuccessor(i->second)) << '\n';
161 out << "\t\tdefault : " << getLabelName(inst->getDefaultDest()) << '\n';
162 }

```

2.1.3.63 void JVMWriter::printVAArgInstruction (const VAArgInst * *inst*) [private]

Print a `va_arg` instruction.

Parameters:

inst the instruction

Definition at line 310 of file `instruction.cpp`.

References `printConstLoad()`, `printIndirectLoad()`, `printIndirectStore()`, `printSimpleInstruction()`, `printValueLoad()`, and `targetData`.

Referenced by `printInstruction()`.

```

310                                                                 {
311     printIndirectLoad(inst->getOperand(0));
312     printSimpleInstruction("dup");
313     printConstLoad(
314         APInt(32, targetData->getTypeAllocSize(inst->getType()), false));
315     printSimpleInstruction("iadd");
316     printValueLoad(inst->getOperand(0));
317     printSimpleInstruction("swap");
318     printIndirectStore(PointerType::getUnqual(
319         IntegerType::get(inst->getContext(), 8)));
320     printIndirectLoad(inst->getType());
321 }

```

2.1.3.64 void JVMWriter::printVAIntrinsic (const IntrinsicInst * *inst*) [private]

Print a `vararg` intrinsic function.

Parameters:

inst the instruction

Definition at line 328 of file `instruction.cpp`.

References `printIndirectLoad()`, `printIndirectStore()`, `printSimpleInstruction()`, `printValueLoad()`, and `vaArgNum`.

Referenced by `printIntrinsicCall()`.

```

328                                     {
329     const Type *valistTy = PointerType::getUnqual(
330         IntegerType::get(inst->getContext(), 8));
331     switch(inst->getIntrinsicID()) {
332     case Intrinsic::vastart:
333         printValueLoad(inst->getOperand(1));
334         printSimpleInstruction("iload", utostr(vaArgNum) + " ; varargptr");
335         printIndirectStore(valistTy);
336         break;
337     case Intrinsic::vacopy:
338         printValueLoad(inst->getOperand(1));
339         printValueLoad(inst->getOperand(2));
340         printIndirectLoad(valistTy);
341         printIndirectStore(valistTy);
342         break;
343     case Intrinsic::vaend:
344         break;
345     }
346 }

```

2.1.3.65 void JVMWriter::printValueLoad(const Value *v) [private]

Load the given value.

Parameters:

v the value to load

Definition at line 30 of file loadstore.cpp.

References `classname`, `externRefs`, `getCallSignature()`, `getLocalVarNumber()`, `getTypePrefix()`, `getValueName()`, `printConstantExpr()`, `printConstLoad()`, `printPtrLoad()`, and `printSimpleInstruction()`.

Referenced by `printAllocaInstruction()`, `printArithmeticInstruction()`, `printBinaryInstruction()`, `printBranchInstruction()`, `printCastInstruction()`, `printFunctionCall()`, `printIndirectLoad()`, `printIndirectStore()`, `printInstruction()`, `printMallocInstruction()`, `printMathIntrinsic()`, `printMemIntrinsic()`, `printOperandPack()`, `printPHICopy()`, `printSelectInstruction()`, `printStaticConstant()`, `printSwitchInstruction()`, `printVAArgInstruction()`, `printVAIntrinsic()`, and `printVirtualInstruction()`.

```

30                                     {
31     if(const Function *f = dyn_cast<Function>(v)) {
32         std::string sig = getValueName(f)
33             + getCallSignature(f->getFunctionType());
34         if(externRefs.count(v))
35             printSimpleInstruction("CLASSFORMETHOD", sig);
36         else
37             printSimpleInstruction("ldc", '"' + classname + '"');
38         printSimpleInstruction("ldc", '"' + sig + '"');
39         printSimpleInstruction("invokestatic",
40             "lljvm/runtime/Function/getFunctionPointer"
41             "(Ljava/lang/String;Ljava/lang/String;)I");
42     } else if(isa<GlobalVariable>(v)) {
43         const Type *ty = cast<PointerType>(v->getType())->getElementType();
44         if(externRefs.count(v))
45             printSimpleInstruction("getstatic", getValueName(v) + " I");
46         else
47             printSimpleInstruction("getstatic",
48                 classname + "/" + getValueName(v) + " I");
49     } else if(isa<ConstantPointerNull>(v)) {
50         printPtrLoad(0);
51     } else if(const ConstantExpr *ce = dyn_cast<ConstantExpr>(v)) {
52         printConstantExpr(ce);

```

```

53     } else if(const Constant *c = dyn_cast<Constant>(v)) {
54         printConstLoad(c);
55     } else {
56         if(getLocalVarNumber(v) <= 3)
57             printSimpleInstruction(
58                 getTypePrefix(v->getType(), true) + "load_"
59                 + utostr(getLocalVarNumber(v))
60                 + " ; " + getValueName(v));
61         else
62             printSimpleInstruction(
63                 getTypePrefix(v->getType(), true) + "load",
64                 utostr(getLocalVarNumber(v))
65                 + " ; " + getValueName(v));
66     }
67 }

```

2.1.3.66 void JVMWriter::printValueStore (const Value * v) [private]

Store the value currently on top of the stack to the given local variable.

Parameters:

v the Value representing the local variable

Definition at line 74 of file loadstore.cpp.

References `getBitWidth()`, `getLocalVarNumber()`, `getTypePrefix()`, `getValueName()`, and `printSimpleInstruction()`.

Referenced by `printBasicBlock()`, `printInvokeInstruction()`, and `printPHICopy()`.

```

74     {
75     if(isa<Function>(v) || isa<GlobalVariable>(v) || isa<Constant>(v)) {
76         errs() << "Value = " << *v << '\n';
77         llvm_unreachable("Invalid value");
78     }
79     unsigned int bitWidth = getBitWidth(v->getType());
80     // truncate int
81     if(bitWidth == 16)
82         printSimpleInstruction("i2s");
83     else if(bitWidth == 8)
84         printSimpleInstruction("i2b");
85     else if(bitWidth == 1) {
86         printSimpleInstruction("iconst_1");
87         printSimpleInstruction("iand");
88     }
89     if(getLocalVarNumber(v) <= 3)
90         printSimpleInstruction(
91             getTypePrefix(v->getType(), true) + "store_"
92             + utostr(getLocalVarNumber(v))
93             + " ; " + getValueName(v));
94     else
95         printSimpleInstruction(
96             getTypePrefix(v->getType(), true) + "store",
97             utostr(getLocalVarNumber(v))
98             + " ; " + getValueName(v));
99 }

```

2.1.3.67 void JVMWriter::printVirtualInstruction (const std::string & sig, const Value * left, const Value * right) [private]

Print the virtual instruction with the given signature.

Parameters:

sig the signature of the instruction

left the first operand

right the second operand

Definition at line 158 of file printinst.cpp.

References printValueLoad(), and printVirtualInstruction().

```
160                                     {
161     printValueLoad(left);
162     printValueLoad(right);
163     printVirtualInstruction(sig);
164 }
```

2.1.3.68 void JVMWriter::printVirtualInstruction (const std::string & sig, const Value * operand) [private]

Print the virtual instruction with the given signature.

Parameters:

sig the signature of the instruction

operand the operand to the instruction

Definition at line 145 of file printinst.cpp.

References printValueLoad(), and printVirtualInstruction().

```
146                                     {
147     printValueLoad(operand);
148     printVirtualInstruction(sig);
149 }
```

2.1.3.69 void JVMWriter::printVirtualInstruction (const std::string & sig) [private]

Print the virtual instruction with the given signature.

Parameters:

sig the signature of the instruction

Definition at line 135 of file printinst.cpp.

References printVirtualInstruction().

```
135                                     {
136     printVirtualInstruction(sig.c_str());
137 }
```

2.1.3.70 void JVMWriter::printVirtualInstruction (const char * *sig*, const Value * *left*, const Value * *right*) [private]

Print the virtual instruction with the given signature.

Parameters:

sig the signature of the instruction

left the first operand

right the second operand

Definition at line 122 of file printinst.cpp.

References printValueLoad(), and printVirtualInstruction().

```

124                                     {
125     printValueLoad(left);
126     printValueLoad(right);
127     printVirtualInstruction(sig);
128 }
```

2.1.3.71 void JVMWriter::printVirtualInstruction (const char * *sig*, const Value * *operand*) [private]

Print the virtual instruction with the given signature.

Parameters:

sig the signature of the instruction

operand the operand to the instruction

Definition at line 109 of file printinst.cpp.

References printValueLoad(), and printVirtualInstruction().

```

110                                     {
111     printValueLoad(operand);
112     printVirtualInstruction(sig);
113 }
```

2.1.3.72 void JVMWriter::printVirtualInstruction (const char * *sig*) [private]

Print the virtual instruction with the given signature.

Parameters:

sig the signature of the instruction

Definition at line 99 of file printinst.cpp.

References out.

Referenced by printArithmeticInstruction(), printBitIntrinsic(), printCastInstruction(), printCmpInstruction(), and printVirtualInstruction().

```

99                                     {
100     out << '\t' << "invokestatic lljvm/runtime/Instruction/" << sig << '\n';
101 }
```

2.1.3.73 bool JVMWriter::runOnFunction (Function &f) [private]

Process the given function.

Parameters:

f the function to process

Returns:

whether the function was modified (always false)

Definition at line 56 of file backend.cpp.

References `printFunction()`.

```
56                                     {
57     if(!f.isDeclaration() && !f.hasAvailableExternallyLinkage())
58         printFunction(f);
59     return false;
60 }
```

2.1.3.74 std::string JVMWriter::sanitizeName (std::string name) [private]

Replace any non-alphanumeric characters with underscores.

Parameters:

name the name to sanitize

Returns:

the sanitized name

Definition at line 33 of file name.cpp.

Referenced by `getLabelName()`, and `getValueName()`.

```
33                                     {
34     for(std::string::iterator i = name.begin(), e = name.end(); i != e; i++)
35         if(!isalnum(*i))
36             *i = '_';
37     return name;
38 }
```

The documentation for this class was generated from the following files:

- backend.h
- backend.cpp
- block.cpp
- branch.cpp
- const.cpp
- function.cpp
- instruction.cpp
- loadstore.cpp
- name.cpp
- printinst.cpp
- sections.cpp
- types.cpp

Index

- doFinalization
 - JVMWriter, 10
- doInitialization
 - JVMWriter, 10
- getAnalysisUsage
 - JVMWriter, 11
- getBitWidth
 - JVMWriter, 11
- getCallSignature
 - JVMWriter, 12
- getLabelName
 - JVMWriter, 12
- getLocalVarNumber
 - JVMWriter, 13
- getTypeDescriptor
 - JVMWriter, 13
- getTypeID
 - JVMWriter, 14
- getTypeName
 - JVMWriter, 14
- getTypePostfix
 - JVMWriter, 15
- getTypePrefix
 - JVMWriter, 15
- getValueName
 - JVMWriter, 16
- JVMWriter, 3
 - doFinalization, 10
 - doInitialization, 10
 - getAnalysisUsage, 11
 - getBitWidth, 11
 - getCallSignature, 12
 - getLabelName, 12
 - getLocalVarNumber, 13
 - getTypeDescriptor, 13
 - getTypeID, 14
 - getTypeName, 14
 - getTypePostfix, 15
 - getTypePrefix, 15
 - getValueName, 16
 - JVMWriter, 9
 - printAllocaInstruction, 16
 - printArithmeticInstruction, 17
 - printBasicBlock, 18
 - printBinaryInstruction, 19
 - printBitCastInstruction, 19
 - printBitIntrinsic, 20
 - printBranchInstruction, 20, 21
 - printCallInstruction, 22
 - printCastInstruction, 22, 23
 - printCatchJump, 24
 - printCmpInstruction, 24
 - printConstantExpr, 25
 - printConstLoad, 26–28
 - printFunction, 29
 - printFunctionBody, 30
 - printFunctionCall, 31
 - printGepInstruction, 31
 - printIndirectLoad, 32, 33
 - printIndirectStore, 33
 - printInstruction, 34
 - printIntrinsicCall, 36
 - printInvokeInstruction, 37
 - printLabel, 37
 - printLocalVariable, 38
 - printLoop, 38
 - printMallocInstruction, 39
 - printMathIntrinsic, 39
 - printMemIntrinsic, 40
 - printOperandPack, 41
 - printPHICopy, 41
 - printPtrLoad, 42
 - printSelectInstruction, 42
 - printSimpleInstruction, 42, 43
 - printStaticConstant, 44
 - printSwitchInstruction, 45
 - printVAArgInstruction, 46
 - printVAIntrinsic, 46
 - printValueLoad, 47
 - printValueStore, 48
 - printVirtualInstruction, 48–50
 - runOnFunction, 50
 - sanitizeName, 51
- printAllocaInstruction
 - JVMWriter, 16
- printArithmeticInstruction
 - JVMWriter, 17

- printBasicBlock
 - JVMWriter, 18
- printBinaryInstruction
 - JVMWriter, 19
- printBitCastInstruction
 - JVMWriter, 19
- printBitIntrinsic
 - JVMWriter, 20
- printBranchInstruction
 - JVMWriter, 20, 21
- printCallInstruction
 - JVMWriter, 22
- printCastInstruction
 - JVMWriter, 22, 23
- printCatchJump
 - JVMWriter, 24
- printCmpInstruction
 - JVMWriter, 24
- printConstantExpr
 - JVMWriter, 25
- printConstLoad
 - JVMWriter, 26–28
- printFunction
 - JVMWriter, 29
- printFunctionBody
 - JVMWriter, 30
- printFunctionCall
 - JVMWriter, 31
- printGepInstruction
 - JVMWriter, 31
- printIndirectLoad
 - JVMWriter, 32, 33
- printIndirectStore
 - JVMWriter, 33
- printInstruction
 - JVMWriter, 34
- printIntrinsicCall
 - JVMWriter, 36
- printInvokeInstruction
 - JVMWriter, 37
- printLabel
 - JVMWriter, 37
- printLocalVariable
 - JVMWriter, 38
- printLoop
 - JVMWriter, 38
- printMallocInstruction
 - JVMWriter, 39
- printMathIntrinsic
 - JVMWriter, 39
- printMemIntrinsic
 - JVMWriter, 40
- printOperandPack
 - JVMWriter, 41
- printPHICopy
 - JVMWriter, 41
- printPtrLoad
 - JVMWriter, 42
- printSelectInstruction
 - JVMWriter, 42
- printSimpleInstruction
 - JVMWriter, 42, 43
- printStaticConstant
 - JVMWriter, 44
- printSwitchInstruction
 - JVMWriter, 45
- printVAArgInstruction
 - JVMWriter, 46
- printVAIntrinsic
 - JVMWriter, 46
- printValueLoad
 - JVMWriter, 47
- printValueStore
 - JVMWriter, 48
- printVirtualInstruction
 - JVMWriter, 48–50
- runOnFunction
 - JVMWriter, 50
- sanitizeName
 - JVMWriter, 51