8. Code Generation

Oscar Nierstrasz

Thanks to Jens Palsberg and Tony Hosking for their kind permission to reuse and adapt the CS132 and CS502 lecture notes. 
http://www.cs.ucla.edu/~palsberg/
http://www.cs.purdue.edu/homes/hosking/
Roadmap

- Runtime storage organization
- Procedure call conventions
- Instruction selection
- Register allocation
- Example: generating Java bytecode

Roadmap

- Runtime storage organization
- Procedure call conventions
- Instruction selection
- Register allocation
- Example: generating Java bytecode
Heap grows “up”, stack grows “down”.

- Allows both stack and heap maximal freedom.
- Code and static data may be separate or intermingled.
Procedures as abstractions

function foo() {
  int a, b;
  ...
  bar(a);
  ...
}

function bar(int a) {
  int x;
  ...
  bar(x);
  ...
}

bar() must preserve foo()'s state while executing.
what if bar() is recursive?
Activation records

Diagram showing activation records with input and output arguments.
Registers

> Typical machine has many of them

> Caller-save vs. Callee-save
  — Convention depending on architecture
  — Used for nifty optimizations
    - When value is not needed after call the caller puts the value in a caller-save register
    - When value is needed in multiple called functions the callers saves it only once

> Parameter passing put first $k$ arguments in registers ($k=4..6$)
  — avoids needless memory traffic because of
    - leaf procedures (many)
    - interprocedural register allocation
  — same with the return address
Procedures as control abstractions

- **On entry**, establish $p$’s environment
- **During a call**, preserve $p$’s environment
- **On exit**, tear down $p$’s environment
# Procedure linkage contract

<table>
<thead>
<tr>
<th><strong>Caller</strong></th>
<th><strong>Callee</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pre-call</strong></td>
<td><strong>prologue</strong></td>
</tr>
<tr>
<td>1. allocate basic frame</td>
<td>1. save registers, state</td>
</tr>
<tr>
<td>2. evaluate &amp; store parameters</td>
<td>2. store FP (dynamic link)</td>
</tr>
<tr>
<td>3. store return address</td>
<td>3. set new FP</td>
</tr>
<tr>
<td>4. jump to child</td>
<td>4. store static link to outer scope</td>
</tr>
<tr>
<td></td>
<td>5. extend basic frame for local data</td>
</tr>
<tr>
<td></td>
<td>6. initialize locals</td>
</tr>
<tr>
<td></td>
<td>7. fall through to code</td>
</tr>
<tr>
<td><strong>post-call</strong></td>
<td><strong>epilogue</strong></td>
</tr>
<tr>
<td>1. copy return value</td>
<td>1. store return value</td>
</tr>
<tr>
<td>2. de-allocate basic frame</td>
<td>2. restore state</td>
</tr>
<tr>
<td>3. restore parameters (if copy out)</td>
<td>3. cut back to basic frame</td>
</tr>
<tr>
<td></td>
<td>4. restore parent’s FP</td>
</tr>
<tr>
<td></td>
<td>5. jump to return address</td>
</tr>
</tbody>
</table>
Variable scoping

Who sees local variables? Where can they be allocated?

**Downward exposure**
- called procedures see caller variables
- dynamic scoping
- lexical scoping

**Upward exposure**
- procedures can return references to variables
- functions that return functions

*With downward exposure can the compiler allocate local variables in frames on the run-time stack.*
Higher-order functions

fun f(x)
    let fun g(y) = x+y
    return g
end

val a = f(1)
val b = f(-1)

val x = a(5)
val y = b(6)

Nested functions
+ Functions returned as values
= Higher-order functions
Access to non-local data

> How does code find non-local data at \textit{run-time}?
> globals are visible everywhere
> lexical nesting
> \textit{view variables as (level, offset) pairs}
  — reflects scoping
  — helps look up name to find most recent declaration
    — If \textit{level} = \textit{current level} then variable is local,
    — else must generate code to look up stack
  — Must maintain
    — \textit{access links} to previous stack frame
    — table of access links (display)

http://en.wikipedia.org/wiki/Call_stack
The Procedure Abstraction

> The *procedure abstraction* supports separate compilation
  — build large programs
  — keep compile times reasonable
  — independent procedures

> The linkage convention (calling convention):
  — *a social contract* — procedures inherit a valid run-time environment
    *and* restore one for their parents
  — *platform dependent* — code generated at compile time
Roadmap

- Runtime storage organization
- Procedure call conventions
- Instruction selection
- Register allocation
- Example: generating Java bytecode
## Calls: Saving and restoring registers

<table>
<thead>
<tr>
<th>caller’s registers</th>
<th>callee saves</th>
<th>caller saves</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call includes bitmap of caller’s registers to be saved/restored. <em>Best: saves fewer registers, compact call sequences</em></td>
<td>Caller saves and restores own registers. Unstructured returns (e.g., exceptions) cause some problems to locate and execute restore code.</td>
<td></td>
</tr>
<tr>
<td>Backpatch code to save registers used in callee on entry, restore on exit. Non-local gotos/exceptions must unwind dynamic chain to restore callee-saved registers.</td>
<td>Bitmap in callee’s stack frame is used by caller to save/restore. Unwind dynamic chain as at left.</td>
<td></td>
</tr>
<tr>
<td>all registers</td>
<td>Easy. Non-local gotos/exceptions must restore all registers from “outermost callee”</td>
<td>Easy. (Use utility routine to keep calls compact.) Non-local gotos/exceptions need only restore original registers.</td>
</tr>
</tbody>
</table>
Call/return (callee saves)

1. caller pushes space for return value
2. caller pushes SP (stack pointer)
3. caller pushes space for: return address, static chain, saved registers
4. caller evaluates and pushes actuals onto stack
5. caller sets return address, callee’s static chain, performs call
6. callee saves registers in register-save area
7. callee copies by-value arrays/records using addresses passed as actuals
8. callee allocates dynamic arrays as needed
9. on return, callee restores saved registers
10. callee jumps to return address
## MIPS registers

<table>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
<th>Use</th>
<th>Callee must preserve?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>$0</td>
<td>constant 0</td>
<td>N/A</td>
</tr>
<tr>
<td>$at</td>
<td>$1</td>
<td>assembler temporary</td>
<td>no</td>
</tr>
<tr>
<td>$v0–$v1</td>
<td>$2–$3</td>
<td>Values for function returns and expression evaluation</td>
<td>no</td>
</tr>
<tr>
<td>$a0–$a3</td>
<td>$4–$7</td>
<td>function arguments</td>
<td>no</td>
</tr>
<tr>
<td>$t0–$t7</td>
<td>$8–$15</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$s0–$s7</td>
<td>$16–$23</td>
<td>saved temporaries</td>
<td>yes</td>
</tr>
<tr>
<td>$t8–$t9</td>
<td>$24–$25</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$k0–$k1</td>
<td>$26–$27</td>
<td>reserved for OS kernel</td>
<td>no</td>
</tr>
<tr>
<td>$gp</td>
<td>$28</td>
<td>global pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$sp</td>
<td>$29</td>
<td>stack pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$fp</td>
<td>$30</td>
<td>frame pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$ra</td>
<td>$31</td>
<td>return address</td>
<td>N/A</td>
</tr>
</tbody>
</table>

MIPS procedure call convention

> **Philosophy:**
  — Use full, general calling sequence only when necessary
  — Omit portions of it where possible (e.g., avoid using FP register whenever possible)

> **Classify routines:**
  — *non-leaf routines* call other routines
  — *leaf routines* don’t
    - identify those that require stack storage for locals
    - and those that don’t
MIPS procedure call convention

> **Pre-call:**

1. Pass arguments: use registers a0 . . . a3; remaining arguments are pushed on the stack along with save space for a0 . . . a3
2. Save caller-saved registers if necessary
3. Execute a jal instruction:
   - *jumps to target address (callee’s first instruction), saves return address in register ra*
MIPS procedure call convention

> **Prologue:**

1. Leaf procedures that use the stack and non-leaf procedures:
   a) *Allocate all stack space needed by routine:*
      - local variables
      - saved registers
      - arguments to routines called by this routine
        
        ```
        subu $sp, framesize
        ```
   b) *Save registers (ra etc.), e.g.:
      
      ```
      sw $31, framesize+frameoffset($sp)
      sw $17, framesize+frameoffset-4($sp)
      sw $16, framesize+frameoffset-8($sp)
      ```
      where *framesize* and *frameoffset* (usually negative) are compile-time constants

2. Emit code for routine
MIPS procedure call convention

Epilogue:
1. Copy return values into result registers (if not already there)
2. Restore saved registers
   \[ \text{lw } $31, \text{framesize+frameoffset}-N($sp) \]
3. Get return address
   \[ \text{lw } $31, \text{framesize+frameoffset}($sp) \]
4. Clean up stack
   \[ \text{addu } $sp, \text{framesize} \]
5. Return
   \[ j \ $31 \]
Roadmap

> Runtime storage organization
> Procedure call conventions
> **Instruction selection**
> Register allocation
> Example: generating Java bytecode
Instruction selection

> **Simple approach:**
  — Macro-expand each IR tuple/subtree to machine instructions
  — Expanding independently leads to poor code quality
  — Mapping may be many-to-one
  — “Maximal munch” works well with RISC

> **Interpretive approach:**
  — Model target machine state as IR is expanded
Register and temporary allocation

> Limited # hard registers
  — assume *pseudo-register* for each temporary
  — register allocator chooses temporaries to spill
  — allocator generates mapping
  — allocator inserts code to spill/restore pseudo-registers to/from storage as needed
> A *tree pattern* characterizes a fragment of the IR corresponding to a machine instruction

—Instruction selection means *tiling* the IR tree with a minimal set of tree patterns
### MIPS Tree Patterns (Example)

<table>
<thead>
<tr>
<th></th>
<th>( r_i )</th>
<th>TEMP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( r_0 )</td>
<td>CONST 0</td>
</tr>
<tr>
<td>li</td>
<td>Rd Rd I</td>
<td>CONST</td>
</tr>
<tr>
<td>la</td>
<td>Rd Rd label</td>
<td>NAME</td>
</tr>
<tr>
<td>move</td>
<td>Rd Rs</td>
<td>MOVE((\cdot, \cdot))</td>
</tr>
<tr>
<td>add</td>
<td>Rd Rs Rs I_{16}</td>
<td>(+(\cdot, \cdot), +(\cdot,\text{CONST}<em>{16}), +(\cdot,\text{CONST}</em>{16}, \cdot))</td>
</tr>
<tr>
<td>mulo</td>
<td>Rd Rs Rs</td>
<td>(\times(\cdot, \cdot), \times(\cdot, \text{CONST}<em>{16}), \times(\cdot, \text{CONST}</em>{16}, \cdot))</td>
</tr>
<tr>
<td>and</td>
<td>Rd Rs Rs</td>
<td>(\text{AND}(\cdot, \cdot), \text{AND}(\cdot, \text{CONST}<em>{16}), \text{AND}(\cdot, \text{CONST}</em>{16}, \cdot))</td>
</tr>
<tr>
<td>or</td>
<td>Rd Rs Rs</td>
<td>(\text{OR}(\cdot, \cdot), \text{OR}(\cdot, \text{CONST}<em>{16}), \text{OR}(\cdot, \text{CONST}</em>{16}, \cdot))</td>
</tr>
<tr>
<td>xor</td>
<td>Rd Rs Rs</td>
<td>(\text{XOR}(\cdot, \cdot), \text{XOR}(\cdot, \text{CONST}<em>{16}), \text{XOR}(\cdot, \text{CONST}</em>{16}, \cdot))</td>
</tr>
<tr>
<td>sub</td>
<td>Rd Rs Rs</td>
<td>(-(\cdot, \cdot), -(\cdot, \text{CONST}_{16}))</td>
</tr>
<tr>
<td>div</td>
<td>Rd Rs Rs</td>
<td>(/(\cdot, \cdot), /(\cdot, \text{CONST}_{16}))</td>
</tr>
<tr>
<td>srl</td>
<td>Rd Rs Rs</td>
<td>RSHIFT((\cdot, \cdot))</td>
</tr>
<tr>
<td>sll</td>
<td>Rd Rs Rs</td>
<td>RSHIFT((\cdot, \text{CONST}_{16}))</td>
</tr>
<tr>
<td>sra</td>
<td>Rd Rs Rs</td>
<td>ARSHIFT((\cdot, \cdot))</td>
</tr>
<tr>
<td>lw</td>
<td>Rd I_{16}(Rb)</td>
<td>MEM((\cdot, \text{CONST}<em>{16})), MEM((+(\cdot, \text{CONST}</em>{16}, \cdot)), MEM((\cdot, \text{CONST}_{16})), MEM((\cdot))</td>
</tr>
</tbody>
</table>

#### Notation:
- \( r_i \): register \( i \)
- Rd: destination register
- Rs: source register
- Rb: base register
- \( I \): 32-bit immediate
- \( I_{16} \): 16-bit immediate
- label: code label

#### Addressing Modes:
- register: \( R \)
- indexed: \( I_{16}(Rb) \)
- immediate: \( I_{16} \)
Optimal tiling

> “Maximal munch”
  — Start at root of tree
  — Tile root with largest tile that fits
  — Repeat for each subtree

> NB: (locally) optimal ≠ (global) optimum
  — optimum: least cost instructions sequence (shortest, fewest cycles)
  — optimal: no two adjacent tiles combine to a lower cost tile
  — CISC instructions have complex tiles ⇒ optimal ≠ optimum
  — RISC instructions have small tiles ⇒ optimal ≈ optimum
Optimum tiling

> *Dynamic programming*

— Assign cost to each tree node — sum of instruction costs of best tiling for that node (including best tilings for children)

![Diagram of tiling](http://en.wikipedia.org/wiki/Dynamic_programming)
Roadmap

> Runtime storage organization
> Procedure call conventions
> Instruction selection
> **Register allocation**
> Example: generating Java bytecode
Register allocation

- Want to have value in register when used
  - limited resources
  - changes instruction choices
  - can move loads and stores
  - optimal allocation is difficult (NP-complete)
Liveness analysis

> **Problem:**
  — IR has unbounded # temporaries
  — Machines has bounded # registers

> **Approach:**
  — Temporaries with disjoint *live* ranges can map to same register
  — If not enough registers, then *spill* some temporaries (i.e., keep in memory)

> The compiler must perform *liveness analysis* for each temporary
  — It is *live* if it holds a value that may still be needed
Control flow analysis

Liveness information is a form of data flow analysis over the control flow graph (CFG):

— Nodes may be individual program statements or basic blocks
— Edges represent potential flow of control

\[
L_1: \quad b \leftarrow a + 1 \\
c \leftarrow c + b \\
a \leftarrow b \times 2 \\
\text{if } a < N \text{ goto } L_1 \\
\text{return } c
\]
A variable \( v \) is **live** on edge \( e \) if there is a path from \( e \) to a use of \( v \) not passing through a definition of \( v \).

\[
\begin{align*}
a &:= 0 \\
b &:= a + 1 \\
c &:= c + b \\
a &:= b * 2 \\
a &< N \\
\text{return } c
\end{align*}
\]

\[
\begin{align*}
a &:= 0 \\
b &:= a + 1 \\
c &:= c + b \\
a &:= b * 2 \\
a &< N \\
\text{return } c
\end{align*}
\]

\[
\begin{align*}
a &:= 0 \\
b &:= a + 1 \\
c &:= c + b \\
a &:= b * 2 \\
a &< N \\
\text{return } c
\end{align*}
\]

\[a \text{ and } b \text{ are never live at the same time, so two registers suffice to hold } a, b \text{ and } c\]
Roadmap

- Runtime storage organization
- Procedure call conventions
- Instruction selection
- Register allocation
- **Example: generating Java bytecode**
Straightline Compiler Files

Source files

- "Grammar spec"
  - slpl.jj

- "Compiler source"
  - CompilerVisitor ...

Generated files

- "Grammar spec with actions"
  - jtb.out.jj

- "Default visitors and interfaces"
  - Visitor ...

- "Syntax Tree Nodes"
  - Goal ...

- "Parser source"
  - StraightLineParser ...

- "Bytecode"
  - StraightLineParser ...

Key

- generates
Straightline Compiler Runtime

```
«Straightline source code»
Examples

StraightLineParser

«Syntax Tree»
Goal ...

visits

CompilerVisitor ...

uses

«Bytecode generation library»
bcel.jar

bytecode

output

Key
generates
```
package compiler;
...
public class CompilerVisitor extends DepthFirstVisitor {
    
    Generator gen;
    
    public CompilerVisitor(String className) {
        gen = new Generator(className);
    }
    
    public void visit(Assignment n) {
        n.f0.accept(this);
        n.f1.accept(this);
        n.f2.accept(this);
        String id = n.f0.f0.tokenImage;
        gen.assignValue(id);
    }
    
    public void visit(PrintStm n) {
        n.f0.accept(this);
        gen.prepareToPrint();
        n.f1.accept(this);
        n.f2.accept(this);
        n.f3.accept(this);
        gen.stopPrinting();
    }
    ...
}
We introduce a separate class to introduce a higher-level interface for generating bytecode.

Creates a class with a static main!
Invoking print methods

```java
private void genPrintTopNum() {
    il.append(factory.createInvoke("java.io.PrintStream", "print",
        Type.VOID, new Type[] { Type.INT }, Constants.INVOKEVIRTUAL));
}
private void genPrintString(String s) {
    pushSystemOut();
    il.append(new PUSH(cp, s));
    il.append(factory.createInvoke("java.io.PrintStream", "print",
        Type.VOID, new Type[] { Type.STRING }, Constants.INVOKEVIRTUAL));
}
private void pushSystemOut() {
    il.append(factory.createFieldAccess("java.lang.System", "out",
        new ObjectType("java.io.PrintStream"), Constants.GETSTATIC));
}
public void prepareToPrint() {
    pushSystemOut();
}
public void printValue() {
    genPrintTopNum();
    genPrintString(" ");
}
public void stopPrinting() {
    genPrintTopNum();
    genPrintString("\n");
}
```

To print, we must push System.out on the stack, push the arguments, then invoke print.
Binary operators

```java
public void add() {
    il.append(new IADD());
}

public void subtract() {
    il.append(new ISUB());
}

public void multiply() {
    il.append(new IMUL());
}

public void divide() {
    il.append(new IDIV());
}

public void pushInt(int val) {
    il.append(new PUSH(cp, val));
}
```

Operators simply consume the top stack items and push the result back on the stack.
public void assignValue(String id) {
    il.append(factory.createStore(Type.INT, getLocation(id)));
}

public void pushId(String id) {
    il.append(factory.createLoad(Type.INT, getLocation(id)));
}

private int getLocation(String id) {
    if(!symbolTable.containsKey(id)) {
        symbolTable.put(id, 1+symbolTable.size());
    }
    return symbolTable.get(id);
}

Variables must be translated to locations. BCEL keeps track of the needed space.
public void generate(File folder) throws IOException {
    il.append(InstructionFactory.createReturn(Type.VOID));
    method.setMaxStack();
    method.setMaxLocals();
    cg.addMethod(method.getMethod());
    il.dispose();
    OutputStream out =
        new FileOutputStream(new File(folder, className + ".class"));
    cg.getJavaClass().dump(out);
}
public class Eg3 {
    public static void main(java.lang.String[] arg0);
        0  getstatic java.lang.System.out : java.io.PrintStream [12]
        3  iconst_1
        4  istore_1
        5  iload_1
        6  iload_1
        7  iload_1
        8  imul
        9  iadd
       10  iload_1
       11  iadd
       12  istore_1
       13  iload_1
       14  invokevirtual java.io.PrintStream.print(int) : void [18]
       20  ldc <String " "> [20]
       22  invokevirtual java.io.PrintStream.print(java.lang.String) : void [23]
       28  iload_1
       29  iconst_1
       30  iadd
       31  invokevirtual java.io.PrintStream.print(int) : void [18]
       37  ldc <String \n> [25]
       39  invokevirtual java.io.PrintStream.print(java.lang.String) : void [23]
       42  return
    }

Generated from:
"print((a := 1; a := a+a*a+a, a),a+1)"
Decompiling the generated class files

```java
import java.io.PrintStream;

public class Eg3
{
    public static void main(String[] arg0)
    {
        int i = 1;
        i = i + i * i + i;
        System.out.print(i);
        System.out.print(" ");
        System.out.print(i + 1);
        System.out.print("\n");
    }
}
```
What you should know!

- How is the run-time stack typically organized?
- What is the “procedure linkage contract”?
- What is the difference between the FP and the SP?
- What are storage classes for variables?
- What is “maximal munch”?
- Why is liveness analysis useful to allocate registers?
- How does BCEL simplify code generation?
Can you answer these questions?

✎ Why does the run-time stack grow down and not up?
✎ In Java, which variables are stored on the stack?
✎ Does Java support downward or upward exposure of local variables?
✎ Why is optimal tiling not necessarily the optimum?
✎ What semantic analysis have we forgotten to perform in our straightline to bytecode compiler?
Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

You are free to:
- **Share** — copy and redistribute the material in any medium or format
- **Adapt** — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

- **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

- **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

[http://creativecommons.org/licenses/by-sa/4.0/](http://creativecommons.org/licenses/by-sa/4.0/)