A Small Language

- A language with integers and integer operations

\[
\begin{align*}
P & \Rightarrow D; P | D \\
D & \Rightarrow \text{def id(ARGS)} = E; \\
\text{ARGS} & \Rightarrow \text{id}, \text{ARGS} | \text{id} \\
E & \Rightarrow \text{int} | \text{id} | \text{if } E_1 = E_2 \text{ then } E_3 \text{ else } E_4 \\
& \quad | E_1 + E_2 | E_1 - E_2 | \text{id}(E_1, \ldots, E_n)
\end{align*}
\]

A Small Language (Cont.)

- The first function definition \( f \) is the “main” routine
- Running the program on input \( i \) means computing \( f(i) \)
- Program for computing the Fibonacci numbers:
  \[
  \text{def } \text{fib}(x) = \text{if } x = 1 \text{ then } 0 \text{ else} \\
  \quad \text{if } x = 2 \text{ then } 1 \text{ else} \\
  \quad \text{fib}(x - 1) + \text{fib}(x - 2)
  \]
Code Generation Strategy

- For each expression \( e \) we generate MIPS code that:
  - Computes the value of \( e \) in \( a0 \)
  - Preserves \( sp \) and the contents of the stack

- We define a code generation function \( cgen(e) \) whose result is the code generated for \( e \)

Code Generation for Constants

- The code to evaluate a constant simply copies it into the accumulator:

\[
cgen(i) = li \ a0 \ i
\]

- Note that this also preserves the stack, as required

Code Generation for Add

\[
cgen(e_1 + e_2) =
\]
\[
cgen(e_1)
sw \ a0 \ 0(sp)
addiu \ sp \ sp -4
cgen(e_2)
lw \ t1 \ 4(sp)
add \ a0 \ t1 \ a0
addiu \ sp \ sp 4
\]

- Possible optimization: Put the result of \( e_1 \) directly in register \( t1 \) ?
Code Generation for Add.

Wrong!

- Optimization: Put the result of $e_1$ directly in $t1$?

\[
cgen(e_1 + e_2) = \\
cgen(e_1) \\
mov $t1 $a0 \\
cgen(e_2) \\
add $a0 $t1 $a0
\]

- Try to generate code for: $3 + (7 + 5)$

---

Code Generation Notes

- The code for + is a template with “holes” for code for evaluating $e_1$ and $e_2$
- Stack machine code generation is recursive
- Code for $e_1 + e_2$ consists of code for $e_1$ and $e_2$ glued together
- Code generation can be written as a recursive-descent of the AST
  - At least for expressions

---

Code Generation for Sub and Constants

- New instruction: sub reg₁, reg₂, reg₃
- Implements $reg₁ = reg₂ - reg₃$

\[
cgen(e_1 - e_2) = \\
cgen(e_1) \\
sw $a0 0($sp) \\
addiu $sp $sp -4 \\
cgen(e_2) \\
lw $t1 4($sp) \\
sub $a0 $t1 $a0 \\
addiu $sp $sp 4
\]
Code Generation for Conditional

- We need flow control instructions
- New instruction: `beq reg₁, reg₂ label`
  - Branch to label if `reg₁ = reg₂`
- New instruction: `b label`
  - Unconditional jump to label

Code Generation for If (Cont.)
```
cgen(if e₁ = e₂ then e₃ else e₄) =
  false_branch:
  cgen(e₂)
  sw $a0 0($sp)
  addiu $sp $sp -4
  true_branch:
  cgen(e₁)
  lw $t1 4($sp)
  addiu $sp $sp 4
  beq $a0 $t1 true_branch
```

The Activation Record

- Code for function calls and function definitions depends on the layout of the activation record
- A very simple AR suffices for this language:
  - The result is always in the accumulator
  - No need to store the result in the AR
  - The activation record holds actual parameters
    - For `f(x₁,...,xₙ)` push `xₙ,...,x₁` on the stack
    - These are the only variables in this language
The Activation Record (Cont.)
- The stack discipline guarantees that on function exit $sp$ is the same as it was on function entry
- We need the return address
- It’s handy to have a pointer to the current activation
  - This pointer lives in register $fp$ (frame pointer)
  - Reason for frame pointer will be clear shortly

The Activation Record
Summary: For this language, an AR with the caller’s frame pointer, the actual parameters, and the return address suffices
- Picture: Consider a call to $f(x,y)$, The AR will be:

```
<table>
<thead>
<tr>
<th>FP</th>
<th>old fp</th>
<th>y</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

AR of previous call
AR of $f$

Code Generation for Function Call
- The calling sequence is the instructions (of both caller and callee) to set up a function invocation
- New instruction: jal label
  - Jump to label, save address of next instruction in $ra$
  - On other architectures the return address is stored on the stack by the “call” instruction
Code Generation for Function Call (Cont.)

cgen(f(e_1, ..., e_n)) =

sw $fp 0($sp)
addiu $sp $sp -4
cgen(e_1)
sw $a0 0($sp)
addiu $sp $sp -4
...
cgen(e_n)
sw $a0 0($sp)
addiu $sp $sp -4
jal f_entry

• The caller saves its value of the frame pointer
• Then it saves the actual parameters in reverse order
• The caller saves the return address in register $ra
• The AR so far is 4*n+4 bytes long

Code Generation for Function Definition

• New instruction: jr reg
  • Jump to address in register reg

cgen(def f(x_1, ..., x_n) = e) =
mov $fp $sp
sw $ra 0($sp)
addiu $sp $sp -4
cgen(e)
lw $ra 4($sp)
addiu $sp $sp $z
lw $fp 0($sp)
jr $ra

• Note: The frame pointer points to the bottom of the frame
• The callee pops the return address, the actual arguments and the saved value of the frame pointer
• \( z = 4*n + 8 \)

Calling Sequence. Example for f(x,y).

<table>
<thead>
<tr>
<th>Before call</th>
<th>After call</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP</td>
<td>FP</td>
</tr>
<tr>
<td>SP</td>
<td>FP</td>
</tr>
<tr>
<td>old fp</td>
<td>old fp</td>
</tr>
<tr>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>On entry</th>
<th>Before exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP</td>
<td>FP</td>
</tr>
<tr>
<td>SP</td>
<td>SP</td>
</tr>
<tr>
<td>old fp</td>
<td>old fp</td>
</tr>
<tr>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>return</td>
<td>return</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SP</th>
<th>SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>old fp</td>
<td>old fp</td>
</tr>
<tr>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>return</td>
<td>return</td>
</tr>
</tbody>
</table>
Code Generation for Variables

- Variable references are the last construct
- The “variables” of a function are just its parameters
  - They are all in the AR
  - Pushed by the caller
- Problem: Because the stack grows when intermediate results are saved, the variables are not at a fixed offset from $sp$

Code Generation for Variables (Cont.)

- Solution: use a frame pointer
  - Always points to the return address on the stack
  - Since it does not move it can be used to find the variables
- Let $x_i$ be the $i^{th}$ ($i = 1, \ldots, n$) formal parameter of the function for which code is being generated

\[
cgen(x_i) = lw $a0 z($fp) \quad (z = 4*i)
\]

Example: For a function def $f(x,y) = e$ the activation and frame pointer are set up as follows:

```
old fp
  \cdot X is at fp + 4
y
  \cdot Y is at fp + 8
x
  \cdot FP
return
SP
```
Temporaries
- The stack machine has activation records and intermediate results interleaved on the stack

<table>
<thead>
<tr>
<th>AR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temporaries</td>
</tr>
<tr>
<td>AR</td>
</tr>
<tr>
<td>Temporaries</td>
</tr>
</tbody>
</table>

Review (Cont.)
- Advantage: Very simple code generation
- Disadvantage: Very slow code
  - Storing/loading temporaries requires a store/load and $sp adjustment

A Better Way
- Idea: Keep temporaries in the AR
- The code generator must assign a location in the AR for each temporary
Example

```python
def fib(x):
    if x == 1:
        return 0
    elif x == 2:
        return 1
    else:
        return fib(x - 1) + fib(x - 2)
```

- What intermediate values are placed on the stack?
- How many slots are needed in the AR to hold these values?

How Many Temporaries?

- Let $NT(e) =$ # of temps needed to evaluate $e$
- $NT(e_1 + e_2)$
  - Needs at least as many temporaries as $NT(e_1)$
  - Needs at least as many temporaries as $NT(e_2)$ + 1
- Space used for temporaries in $e_1$ can be reused for temporaries in $e_2$

The Equations

- $NT(e_1 + e_2) = \max(NT(e_1), 1 + NT(e_2))$
- $NT(e_1 - e_2) = \max(NT(e_1), 1 + NT(e_2))$
- $NT(\text{if } e_1 = e_2 \text{ then } e_3 \text{ else } e_4) = \max(NT(e_1), 1 + NT(e_2), NT(e_3), NT(e_4))$
- $NT(id(e_1, \ldots, e_n)) = \max(NT(e_1), \ldots, NT(e_n))$
- $NT(int) = 0$
- $NT(id) = 0$
The Revised AR

- For a function definition $f(x_1, \ldots, x_n) = e$ the AR has $2 + n + \text{NT}(e)$ elements
  - Return address
  - Frame pointer
  - $n$ arguments
  - NT(e) locations for intermediate results

Revised Code Generation

- Code generation must know how many temporaries are in use at each point

- Add a new argument to code generation: the position of the next available temporary

Picture

<table>
<thead>
<tr>
<th>Old FP</th>
<th>(x_1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\ldots</td>
<td>\ldots</td>
</tr>
<tr>
<td>(x_n)</td>
<td>Return Addr</td>
</tr>
<tr>
<td></td>
<td>Temp NT(e)</td>
</tr>
<tr>
<td></td>
<td>Temp 1</td>
</tr>
</tbody>
</table>
Code Generation for +
(original)

cgen(e₁ + e₂) =
cgen(e₁)
sw $a0 0($sp)
addiu $sp $sp -4
cgen(e₂)
lw $t1 4($sp)
add $a0 $t1 $a0
addiu $sp $sp 4

Code Generation for +
(revised)

cgen(e₁ + e₂, nt) =
cgen(e₁, nt)
sw $a0 nt($fp)
cgen(e₂, nt + 4)
lw $t1 nt($fp)
add $a0 $t1 $a0

Notes

- The temporary area is used like a small, fixed-size stack
- Exercise: Write out cgen for other constructs
Summary

- The activation record must be designed together with the code generator
- Code generation can be done by recursive traversal of the AST
- Production compilers do different things
  - Emphasis is on keeping values (esp. current stack frame) in registers
  - Intermediate results are laid out in the AR, not pushed and popped from the stack

Code Generation for OO Languages

Issues:

- How to lay out objects?
- How is dynamic dispatch handled?

Object Layout

- OO implementation = Code generation issues already talked about plus others
- OO Slogan: If B is a subclass of A, than an object of class B can be used wherever an object of class A is expected
- This means that code in class A works unmodified for an object of class B
Object Layout Example

```java
class A {
    int a = 0;
    int d = 1;
    int f() { a = a + d; return a; }
};

class B extends A {
    int b = 2;
    int f() { return a; }
    int g() { a = a - b; return a; }
};

class C extends A {
    int c = 3;
    int h() { a = a * c; return a; }
};
```

Object Layout (Cont.)

- Fields a and d are inherited by classes B and C
- All methods in all classes refer to a
- For A methods to work correctly in A, B, and C objects, field a must be in the same “place” in each object

Object Layout (Cont.)

An object is like a struct in C. The reference `foo.field` is an index into a foo struct at an offset corresponding to field

Objects in Java/C++ are implemented similarly
- Objects are laid out in contiguous memory
- Each field is stored at a fixed offset in object
A Sample Object Layout

- The first 3 words of an object contain header information:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Class Tag</td>
</tr>
<tr>
<td>4</td>
<td>Object Size</td>
</tr>
<tr>
<td>8</td>
<td>Dispatch Ptr</td>
</tr>
<tr>
<td>12</td>
<td>Field 1</td>
</tr>
<tr>
<td>16</td>
<td>Field 2</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Sample Object Layout (Cont.)

- Class tag is an integer
  - Identifies class of the object
- Object size is an integer
  - Size of the object in words
- Dispatch ptr is a pointer to a table of methods
  - More later
- Fields in subsequent slots
- Lay out in contiguous memory

Subclasses

**Observation:** Given a layout for class A, a layout for subclass B can be defined by extending the layout of A with additional slots for the additional field of B

Leaves the layout of A unchanged
(B is an extension of A)
Subclasses (Cont.)

- The offset for a field is the same in a class and all of its subclasses
- Any method for an $A_i$ can be used on a subclass $A_j$
- Consider layout for $A_n < ... < A_3 < A_2 < A_1$

<table>
<thead>
<tr>
<th>Offset</th>
<th>Class</th>
<th>A1 attrs</th>
<th>A2 attrs</th>
<th>A3 attrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Atag</td>
<td>a</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Btag</td>
<td>a</td>
<td>d</td>
<td>b</td>
</tr>
<tr>
<td>8</td>
<td>Ctag</td>
<td>a</td>
<td>d</td>
<td>c</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What about multiple inheritance?

Dynamic Dispatch

- Consider the following dispatches (using the same example)
Object Layout Example
(Repeat)

```java
class A {
    int a = 0;
    int d = 1;
    int f() { a = a + d; return a; }
};

class B extends A {
    int b = 2;
    int f() { return a; }
    int g() { a = a - b; return a; }
};

class C extends A {
    int c = 3;
    int h() { a = a * c; return a; }
};
```

Dynamic Dispatch Example

- `e.g()`
  - `g` refers to method in `B` if `e` is a `B`
- `e.f()`
  - `f` refers to method in `A` if `f` is an `A` or `C` (inherited in the case of `C`)
  - `f` refers to method in `B` for a `B` object

The implementation of methods and dynamic dispatch strongly resembles the implementation of fields

Dispatch Tables

- Every class has a fixed set of methods (including inherited methods)
  - A dispatch table indexes these methods
    - dispatch table = an array of method entry points
    - A method `f` lives at a fixed offset in the dispatch table for a class and all of its subclasses
Dispatch Table Example

<table>
<thead>
<tr>
<th>Offset</th>
<th>Class</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>fA</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>fB</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>g</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>fA</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>h</td>
</tr>
</tbody>
</table>

- The dispatch table for class A has only 1 method
- The tables for B and C extend the table for A to the right
- Because methods can be overridden, the method for f is not the same in every class, but is always at the same offset

Using Dispatch Tables

- The dispatch pointer in an object of class X points to the dispatch table for class X
- Every method f of class X is assigned an offset $O_f$ in the dispatch table at compile time

Using Dispatch Tables (Cont.)

- To implement a dynamic dispatch e.f() we
  - Evaluate e. The result is a pointer to an object x
  - Call $D[O_f]$
    - D is the dispatch table for x
    - In the call, this is bound to x