Code Generation

CS143

Lecture 12

Instructor: Fredrik Kjolstad

Slide design by Prof. Alex Aiken, with modifications

Lecture Outline

- Topic 1: Basic Code Generation
  - The MIPS assembly language
  - A simple source language
  - Stack-machine implementation of the simple language
- Topic 2: Code Generation for Objects

From Stack Machines to MIPS

- The compiler generates code for a stack machine with accumulator
- We want to run the resulting code on the MIPS processor (or simulator)
- We simulate stack machine instructions using MIPS instructions and registers

Simulating a Stack Machine…

- The accumulator is kept in MIPS register $a0

- The stack is kept in memory
  - The stack grows towards lower addresses
  - Standard convention on the MIPS architecture

- The address of the next location on the stack is kept in MIPS register $sp
  - The top of the stack is at address $sp + 4

MIPS Assembly

MIPS architecture
- Prototypical Reduced Instruction Set Computer (RISC) architecture
- Arithmetic operations use registers for operands and results
- Must use load and store instructions to use operands and results in memory
- 32 general purpose registers (32 bits each)
  - We will use $sp, $a0 and $t1 (a temporary register)

- Read the SPIM documentation for details

A Sample of MIPS Instructions

- lw reg_1 offset(reg_2)
  - Load 32-bit word from address reg_2 + offset into reg_1
- add reg_1 reg_2 reg_3
  - $reg_1 \leftarrow reg_2 + reg_3$
- sw reg_1 offset(reg_2)
  - Store 32-bit word in reg_1 at address reg_2 + offset
- addiu reg_1 reg_2 imm
  - $reg_1 \leftarrow reg_2 + imm$
  - "u" means overflow is not checked
- li reg imm
  - $reg \leftarrow imm$

MIPS Assembly. Example.

- The stack-machine code for 7 + 5 in MIPS:

acc ← 7	li $a0 7
push acc	sw $a0 0($sp)
	addiu $sp $sp -4
acc ← 5	li $a0 5
acc ← acc + top_of_stack	lw $t1 4($sp)
	add $a0 $a0 $t1
pop	addiu $sp $sp 4

- We now generalize this to a simple language…

A Small Language

- A language with integers and integer operations

$$
\begin{aligned}
P &\rightarrow D; P \mid D \\
D &\rightarrow \text{def id(ARGS)} = E; \\
ARGS &\rightarrow \text{id}, ARGS \mid \text{id} \\
E &\rightarrow \text{int} \mid \text{id} \mid \text{if } E_1 = E_2 \text{ then } E_3 \text{ else } E_4 \\
&\quad \mid E_1 + E_2 \mid E_1 - E_2 \mid \text{id}(E_1,\ldots,E_n)
\end{aligned}
$$

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:

```
def fib(x) = if x = 1 then 0 else
             if x = 2 then 1 else
               fib(x - 1) + 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

- This preserves the stack, as required

- Color key:
  - RED: compile time
  - BLUE: run time

Code Generation for Add

$cgen(e_1 + e_2) =$

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

$cgen(e_1 + e_2) =$

```
    cgen(e1)
    print "sw $a0 0($sp)"
    print "addiu $sp $sp -4"
    cgen(e2)
    print "lw $t1 4($sp)"
    print "add $a0 $t1 $a0"
    print "addiu $sp $sp 4"
```

Code Generation for Add. Wrong!

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

  cgen(e_1 + e_2) =

  cgen(e_1)

  move $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 $e_1 + e_2$ is 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 reg_1 reg_2 reg_3
  - Implements $reg_1 \leftarrow reg_2 - reg_3$

cgen(e_1 - e_2) =

```
cgen(e1)
sw $a0 0($sp)
addiu $sp $sp -4
cgen(e2)
lw $t1 4($sp)
sub $a0 $t1 $a0
addiu $sp $sp 4
```

Code Generation for Conditional

- We need flow control instructions

- New instruction: beq reg_1 reg_2 label
  - Branch to label if $reg_1 = reg_2$

- New instruction: b label
  - Unconditional jump to label

Code Generation for If (Cont.)

cgen(if $e_1 = e_2$ then e_3 else e_4) =

cgen(e_1)

sw $a0 0($sp)

addiu $sp $sp -4

cgen(e_2)

lw $t1 4($sp)

addiu $sp $sp 4

beq $a0 $t1 true_branch

false_branch:

cgen(e_4)

b end_if

true_branch:

cgen(e_3)

end_if:

The Activation Record

- Code for function calls and function definitions depends on the layout of the AR

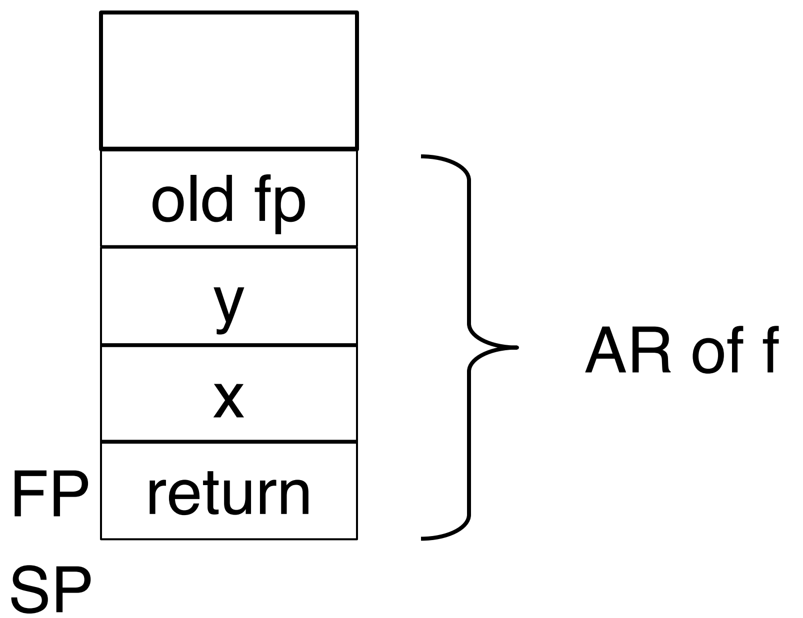
- A very simple AR suffices for this language:
  - The result is always in the accumulator
    - No need to store the result in the AR
  - The activation record holds actual parameters
    - For $f(x_1,\ldots,x_n)$ push $x_n,\ldots,x_1$ on the stack
    - These are the only variables in this language

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
- A pointer to the current activation is useful
  - –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 is:



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.)

$cgen(f(e_1,\ldots,e_n)) =$

```
sw $fp 0($sp)
addiu $sp $sp -4
```

$cgen(e_n)$

```
sw $a0 0($sp)
addiu $sp $sp -4
…
```

$cgen(e_1)$

```
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

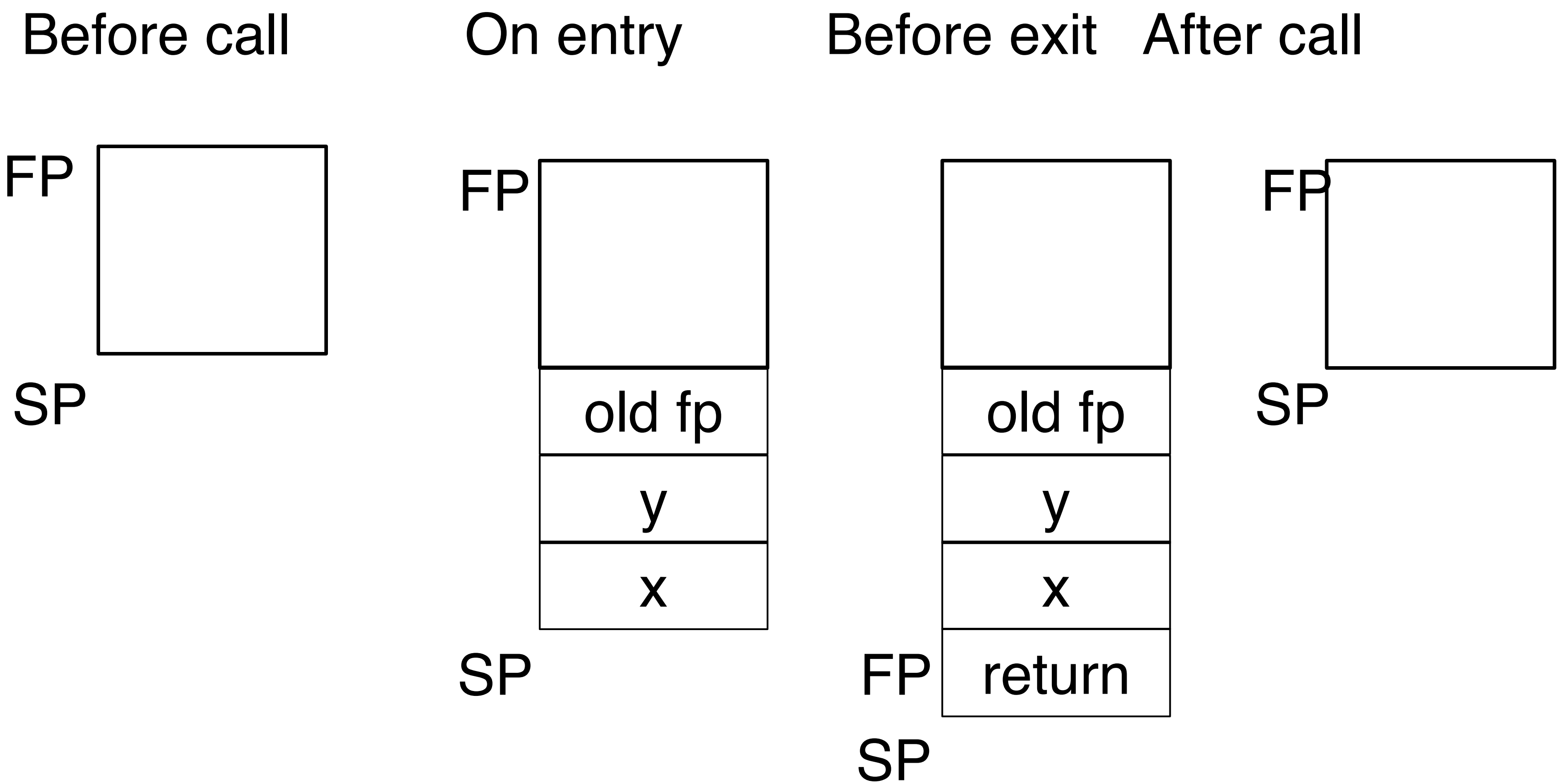
- New instruction: jr reg
  – Jump to address in register reg

cgen(def f(x_1,…,x_n) = e) =

```
move $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 top, not 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)



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

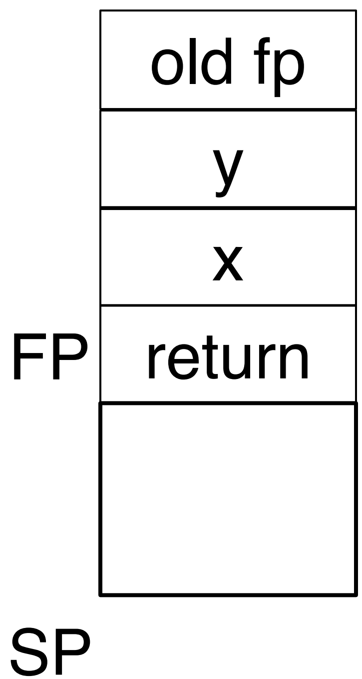
Code Generation for Variables (Cont.)

- 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} ($i = 1,\dots,n$) formal parameter of the function for which code is being generated

$$\text{cgen}(x_i) = \text{lw \$a0 } z(\$fp) \qquad (\, z = 4*i \,)$$

Code Generation for Variables (Cont.)

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



- X is at fp + 4
- Y is at fp + 8

Summary

- The activation record must be designed together with the code generator

- Code generation can be done by recursive traversal of the AST

- We recommend you use a stack machine for your Cool compiler (it’s simple)

Summary

- 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

An Improvement

- Idea: Keep temporaries in the AR

- The code generator must assign a location in the AR for each temporary

Example

```
def fib(x) = if x = 1 then 0 else
    if x = 2 then 1 else
        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

$$NT(e_1 + e_2) = \max(NT(e_1), 1 + NT(e_2))$$

$$NT(e_1 - e_2) = \max(NT(e_1), 1 + NT(e_2))$$

$$NT(\text{if } e_1 = e_2 \text{ then } e_3 \text{ else } e_4) = \max(NT(e_1), 1 + NT(e_2), NT(e_3), NT(e_4))$$

$$NT(id(e_1,\ldots,e_n) = \max(NT(e_1),\ldots,NT(e_n))$$

$$NT(int) = 0$$

$$NT(id) = 0$$

Is this bottom-up or top-down?

What is NT(…code for fib…)?

The Revised AR

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

Picture

Old FP
x_n
. . .
x_1
Return Addr.
Temp NT(e)
. . .
Temp 1

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

Code Generation for + (original)

$cgen(e_1 + e_2) =$

```
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)

$\text{cgen}(e_1 + e_2, \text{nt}) =$

```
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

Code Generation for OO Languages

Topic II

Object Layout

- OO implementation = Stuff from last part + more stuff
- OO Slogan: If B is a subclass of A, then 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

Two Issues

- How are objects represented in memory?

- How is dynamic dispatch implemented?

Object Layout Example

```
Class A {
    a: Int;
    d: Int;
    f(): Int {…};
};

Class  B inherits A {
    b: Int;
    f(): Int {…};
    g(): Int {…};
};

Class  C inherits A {
    c: Int;
    h(): Int {…};
};
```

Object Layout (Cont.)

- Attributes 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, attribute 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 Cool are implemented similarly

- Objects are laid out in contiguous memory
- Each attribute stored at a fixed offset in object
- When a method is invoked, the object is self and the fields are the object's attributes

Cool Object Layout

- The first 3 words of Cool objects contain header information:

	Offset
Class Tag	0
Object Size	4
Dispatch Ptr	8
Attribute 1	12
Attribute 2	16
. . .	

Cool 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
- Attributes 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 attributes of B

Leaves the layout of A unchanged

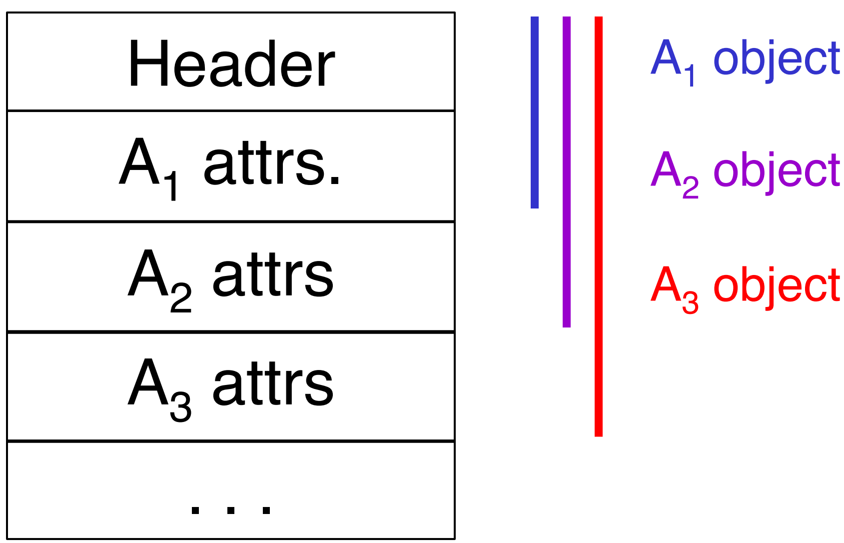
(B is an extension)

Layout Picture

Offset Class	0	4	8	12	16	20
A	Atag	5	*	a	d	
B	Btag	6	*	a	d	b
C	Ctag	6	*	a	d	c

Subclasses (Cont.)

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



Dynamic Dispatch

- Consider the following dispatches
  (using the same example)

Object Layout Example (Repeat)

```
Class A {
    a: Int;
    d: Int;
    f(): Int {…};
};

Class  B inherits A {
    b: Int;
    f(): Int {…};
    g(): Int {…};
};

Class  C inherits A {
    c: Int;
    h(): Int {…};
};
```

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

- The implementation of methods and dynamic dispatch strongly resembles the implementation of attributes

Dispatch Tables

- Every class has a fixed set of methods (including inherited methods)

- A dispatch table indexes these methods
  - 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

Offset Class	0	4
A	fA	
B	fB	g
C	fA	h

- 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, giving an object x
  - Call $D[O_f]$
    - D is the dispatch table for x
    - In the call, self is bound to x