

## Code Generation

### Lecture 12

Prof. Aiken CS 143 Lecture 12

1

## 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

Prof. Aiken CS 143 Lecture 12

2

## 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

Prof. Aiken CS 143 Lecture 12

3

## 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**

Prof. Aiken CS 143 Lecture 12

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

Prof. Aiken CS 143 Lecture 12

5

## 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**
  - $reg \leftarrow imm$

Prof. Aiken CS 143 Lecture 12

6

### 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...

Prof. Aiken CS 143 Lecture 12

7

### A Small Language

- A language with integers and integer operations

```
P → D; P | D
D → def id(ARGS) = E;
ARGS → id, ARGS | id
E → int | id | if E1 = E2 then E3 else E4
    | E1 + E2 | E1 - E2 | id(E1, ..., En)
```

Prof. Aiken CS 143 Lecture 12

8

### 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)
```

Prof. Aiken CS 143 Lecture 12

9

### 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  $e$

Prof. Aiken CS 143 Lecture 12

10

### 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

Prof. Aiken CS 143 Lecture 12

11

### Code Generation for Add

```
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

cgen(e1 + e2) =
  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"
```

Prof. Aiken CS 143 Lecture 12

12

### 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 :  $3 + (7 + 5)$

Prof. Aiken CS 143 Lecture 12

13

### 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

Prof. Aiken CS 143 Lecture 12

14

### 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( $e_1$ )  
  sw  $\$a0$  0( $\$sp$ )  
  addiu  $\$sp$   $\$sp$  -4  
  cgen( $e_2$ )  
  lw  $\$t1$  4( $\$sp$ )  
  sub  $\$a0$   $\$t1$   $\$a0$   
  addiu  $\$sp$   $\$sp$  4
```

Prof. Aiken CS 143 Lecture 12

15

### 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

Prof. Aiken CS 143 Lecture 12

16

### 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:
```

Prof. Aiken CS 143 Lecture 12

17

### 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  $f(x_1, \dots, x_n)$  push  $x_n, \dots, x_1$  on the stack
    - These are the only variables in this language

Prof. Aiken CS 143 Lecture 12

18

### The Activation Record (Cont.)

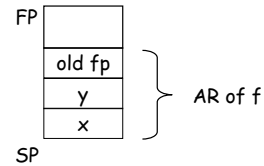
- The stack discipline guarantees that on function exit  $\$sp$  is the same as it was on function entry
  - No need for a control link
- We need the return address
- A pointer to the current activation is useful
  - This pointer lives in register  $\$fp$  (frame pointer)
  - Reason for frame pointer will be clear shortly

Prof. Aiken CS 143 Lecture 12

19

### 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:



Prof. Aiken CS 143 Lecture 12

20

### 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

Prof. Aiken CS 143 Lecture 12

21

### Code Generation for Function Call (Cont.)

```
cgen(f(e1,...,en)) =
  sw $fp 0($sp)
  addiu $sp $sp -4
  cgen(en)
  sw $a0 0($sp)
  addiu $sp $sp -4
  ...
  cgen(e1)
  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

Prof. Aiken CS 143 Lecture 12

22

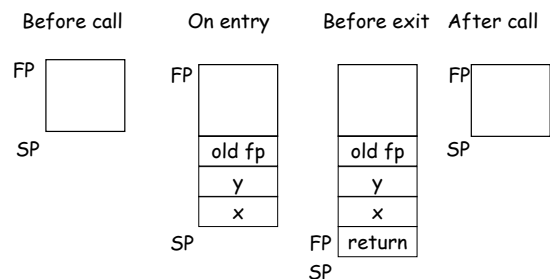
### Code Generation for Function Definition

- New instruction: `jr reg`
    - Jump to address in register `reg`
- ```
cgen(def f(x1,...,xn) = 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
```

Prof. Aiken CS 143 Lecture 12

23

### Calling Sequence: Example for $f(x,y)$



Prof. Aiken CS 143 Lecture 12

24

### 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  $\$sp$

Prof. Aiken CS 143 Lecture 12

25

### 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  $i^{\text{th}}$  ( $i = 1, \dots, n$ ) formal parameter of the function for which code is being generated

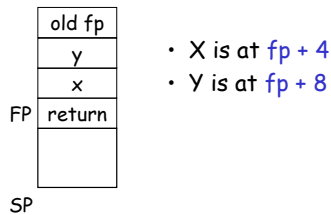
$cgen(x_i) = lw \$a0 z(\$fp) \quad (z = 4*i)$

Prof. Aiken CS 143 Lecture 12

26

### Code Generation for Variables (Cont.)

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



Prof. Aiken CS 143 Lecture 12

27

### 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)

Prof. Aiken CS 143 Lecture 12

28

### 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

Prof. Aiken CS 143 Lecture 12

29

### An Improvement

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

Prof. Aiken CS 143 Lecture 12

30

### 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?

Prof. Aiken CS 143 Lecture 12

31

### 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  $e_2$

Prof. Aiken CS 143 Lecture 12

32

### 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(if  $e_1 = e_2$  then  $e_3$  else  $e_4$ ) = max(NT( $e_1$ ), 1 + NT( $e_2$ ), NT( $e_3$ ), NT( $e_4$ ))
NT(id( $e_1, \dots, e_n$ )) = max(NT( $e_1$ ), ..., NT( $e_n$ ))
NT(int) = 0
NT(id) = 0
```

Is this bottom-up or top-down?  
What is  $NT(\dots \text{code for fib} \dots)$ ?

Prof. Aiken CS 143 Lecture 12

33

### The Revised AR

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

Prof. Aiken CS 143 Lecture 12

34

### Picture

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

Prof. Aiken CS 143 Lecture 12

35

### 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

Prof. Aiken CS 143 Lecture 12

36

### Code Generation for + (original)

```
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
```

Prof. Aiken CS 143 Lecture 12

37

### Code Generation for + (revised)

```
cgen( $e_1 + e_2$ , nt) =  
    cgen( $e_1$ , nt)  
    sw $a0 nt($fp)  
    cgen( $e_2$ , nt + 4)  
    lw $t1 nt($fp)  
    add $a0 $t1 $a0
```

Prof. Aiken CS 143 Lecture 12

38

### Notes

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

Prof. Aiken CS 143 Lecture 12

39

### Code Generation for OO Languages

#### Topic II

Prof. Aiken CS 143 Lecture 12

40

### 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`

Prof. Aiken CS 143 Lecture 12

41

### Two Issues

- How are objects represented in memory?
- How is dynamic dispatch implemented?

Prof. Aiken CS 143 Lecture 12

42

## Object Layout Example

```
Class A {
  a: Int <- 0;
  d: Int <- 1;
  f(): Int { a <- a + d; };
}

Class B inherits A {
  b: Int <- 2;
  f(): Int { a };
  g(): Int { a <- a - b; };
}

Class C inherits A {
  c: Int <- 3;
  h(): Int { a <- a * c; };
}
```

Prof. Aiken CS 143 Lecture 12

43

## 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

Prof. Aiken CS 143 Lecture 12

44

## 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

Prof. Aiken CS 143 Lecture 12

45

## 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     |
| ...          |        |

Prof. Aiken CS 143 Lecture 12

46

## 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

Prof. Aiken CS 143 Lecture 12

47

## 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)

Prof. Aiken CS 143 Lecture 12

48



## Layout Picture

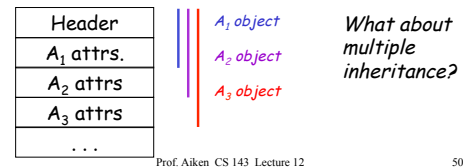
| Offset<br>Class | 0    | 4 | 8 | 12 | 16 | 20 |
|-----------------|------|---|---|----|----|----|
| A               | Atag | 5 | * | a  | d  |    |
| B               | Btag | 6 | * | a  | d  | b  |
| C               | Ctag | 6 | * | a  | d  | c  |

Prof. Aiken CS 143 Lecture 12

49

## 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  $A_n < \dots < A_3 < A_2 < A_1$



Prof. Aiken CS 143 Lecture 12

50

## Dynamic Dispatch

- Consider the following dispatches (using the same example)

Prof. Aiken CS 143 Lecture 12

51

## Object Layout Example (Repeat)

```

Class A {
  a: Int ← 0;
  d: Int ← 1;
  f(): Int { a ← a + d };
};

Class C inherits A {
  c: Int ← 3;
  h(): Int { a ← a * c };
};

Class B inherits A {
  b: Int ← 2;
  f(): Int { a };
  g(): Int { a ← a - b };
};
    
```

Prof. Aiken CS 143 Lecture 12

52

## 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 attributes

Prof. Aiken CS 143 Lecture 12

53

## 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

Prof. Aiken CS 143 Lecture 12

54

### Dispatch Table Example

| Offset<br>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

Prof. Aiken CS 143 Lecture 12

55

### 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

Prof. Aiken CS 143 Lecture 12

56

### 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

Prof. Aiken CS 143 Lecture 12

57