

# **Code Generation**

## Lecture 14-15

# Lecture Outline

---

- Stack machines
- The MIPS assembly language
- A simple source language
- Stack-machine implementation of the simple language

# Stack Machines

---

- A simple evaluation model
- No variables or registers
- A stack of values for intermediate results

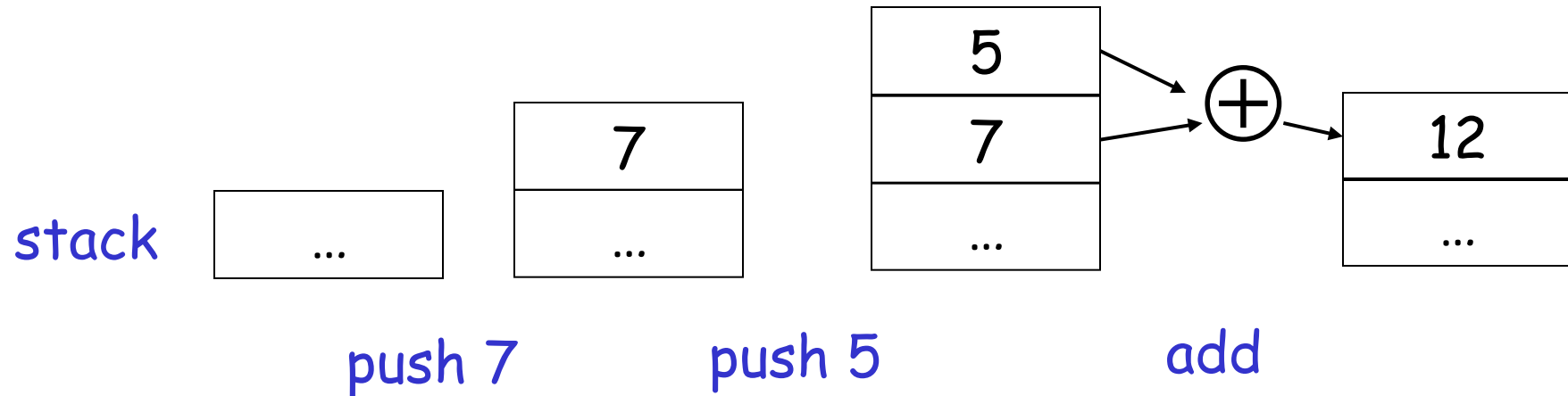
# Example of a Stack Machine Program

---

- Consider two instructions
  - `push i` - place the integer `i` on top of the stack
  - `add` - pop two elements, add them and put the result back on the stack
- A program to compute  $7 + 5$ :
  - `push 7`
  - `push 5`
  - `add`

# Stack Machine. Example

---



- Each instruction:
  - Takes its operands from the top of the stack
  - Removes those operands from the stack
  - Computes the required operation on them
  - Pushes the result on the stack

# Why Use a Stack Machine ?

---

- Each operation takes operands from the same place and puts results in the same place
- This means a uniform compilation scheme
- And therefore a simpler compiler
  - This is what you have to do for PA5

# Why Use a Stack Machine ?

---

- Location of the operands is implicit
  - Always on the top of the stack
- No need to specify operands explicitly
- No need to specify the location of the result
- Instruction “add” as opposed to “add  $r_1, r_2$ ”
  - ⇒ Smaller encoding of instructions
  - ⇒ More compact programs
- This is one reason why Java Bytecodes use a stack evaluation model

# Optimizing the Stack Machine

---

- The add instruction does 3 memory operations
  - Two reads and one write to the stack
  - The top of the stack is frequently accessed
- Idea: keep the top of the stack in a register (called accumulator)
  - Register accesses are faster
- The “add” instruction is now
$$\text{acc} \leftarrow \text{acc} + \text{top\_of\_stack}$$
  - Only one memory operation!



# Stack Machine with Accumulator

---

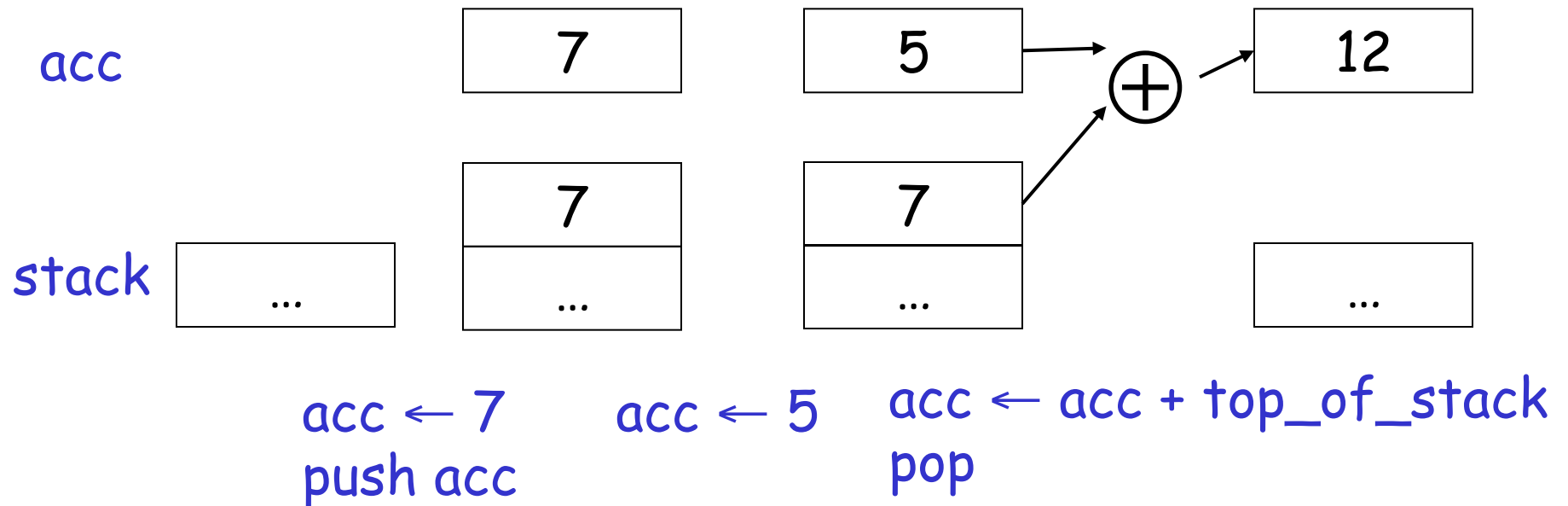
## Invariants

- The result of computing an expression is always in the accumulator
- For an operation  $op(e_1, \dots, e_n)$  push the accumulator on the stack after computing each of  $e_1, \dots, e_{n-1}$ 
  - The result of  $e_n$  is in the accumulator before  $op$
  - After the operation pop  $n-1$  values
- After computing an expression the stack is as before

# Stack Machine with Accumulator. Example

---

- Compute  $7 + 5$  using an accumulator



## A Bigger Example: $3 + (7 + 5)$

---

Code	Acc	Stack
$\text{acc} \leftarrow 3$	3	<init>
push acc	3	3, <init>
$\text{acc} \leftarrow 7$	7	3, <init>
push acc	7	7, 3, <init>
$\text{acc} \leftarrow 5$	5	7, 3, <init>
$\text{acc} \leftarrow \text{acc} + \text{top\_of\_stack}$	12	7, 3, <init>
pop	12	3, <init>
$\text{acc} \leftarrow \text{acc} + \text{top\_of\_stack}$	15	3, <init>
pop	15	<init>

# Notes

---

- It is **very important** that the stack is preserved across the evaluation of a subexpression
  - Stack before the evaluation of  $7 + 5$  is  $3, \langle \text{init} \rangle$
  - Stack after the evaluation of  $7 + 5$  is  $3, \langle \text{init} \rangle$
  - The first operand is on top of the stack

# 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 implement 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 handout for more details

# A Sample of MIPS Instructions

---

- lw  $reg_1$  offset( $reg_2$ )
  - Load 32-bit word from address  $reg_2 + \text{offset}$  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  $reg_2 + \text{offset}$
- addiu  $reg_1$   $reg_2$  imm
  - $reg_1 \leftarrow reg_2 + \text{imm}$
  - “u” means overflow is not checked
- li  $reg$  imm
  - $reg \leftarrow \text{imm}$



## MIPS Assembly. Example.

---

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

$\text{acc} \leftarrow 7$

push acc

$\text{acc} \leftarrow 5$

$\text{acc} \leftarrow \text{acc} + \text{top\_of\_stack}$

pop

li \$a0 7

sw \$a0 0(\$sp)

addiu \$sp \$sp -4

li \$a0 5

lw \$t1 4(\$sp)

add \$a0 \$a0 \$t1

addiu \$sp \$sp 4

- We now generalize this to a simple language...

## Some Useful Macros

---

- We define the following abbreviations
- `push $t`  
`sw $t 0($sp)`  
`addiu $sp $sp -4`
- `pop`  
`addiu $sp $sp 4`
- `$t  $\leftarrow$  top`  
`lw $t 4($sp)`

# A Small Language

---

- A language with integers and integer operations

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

## 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:  
$$\begin{aligned} \text{def fib}(x) = & \text{if } x = 1 \text{ then } 0 \text{ else} \\ & \text{if } x = 2 \text{ then } 1 \text{ else} \\ & \text{fib}(x - 1) + \text{fib}(x - 2) \end{aligned}$$

# 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( $e_1$ )  
    push $a0  
    cgen( $e_2$ )  
    $t1  $\leftarrow$  top  
    add $a0 $t1 $a0  
    pop
```

- 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( $e_1$ )  
    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  $e_1 + e_2$  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  $reg_1 \leftarrow reg_2 - reg_3$   
`cgen(e1 - e2) =`  
`cgen(e1)`  
`push $a0`  
`cgen(e2)`  
`$t1 ← top`  
`sub $a0 $t1 $a0`  
`pop`

## 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.)

---

$\text{cgen}(\text{if } e_1 = e_2 \text{ then } e_3 \text{ else } e_4) =$

$\text{cgen}(e_1)$

push \$a0

$\text{cgen}(e_2)$

$\$t1 \leftarrow \text{top}$

pop

beq \$a0 \$t1 true\_branch

false\_branch:

$\text{cgen}(e_4)$

b end\_if

true\_branch:

$\text{cgen}(e_3)$

end\_if:

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

## The Activation Record (Cont.)

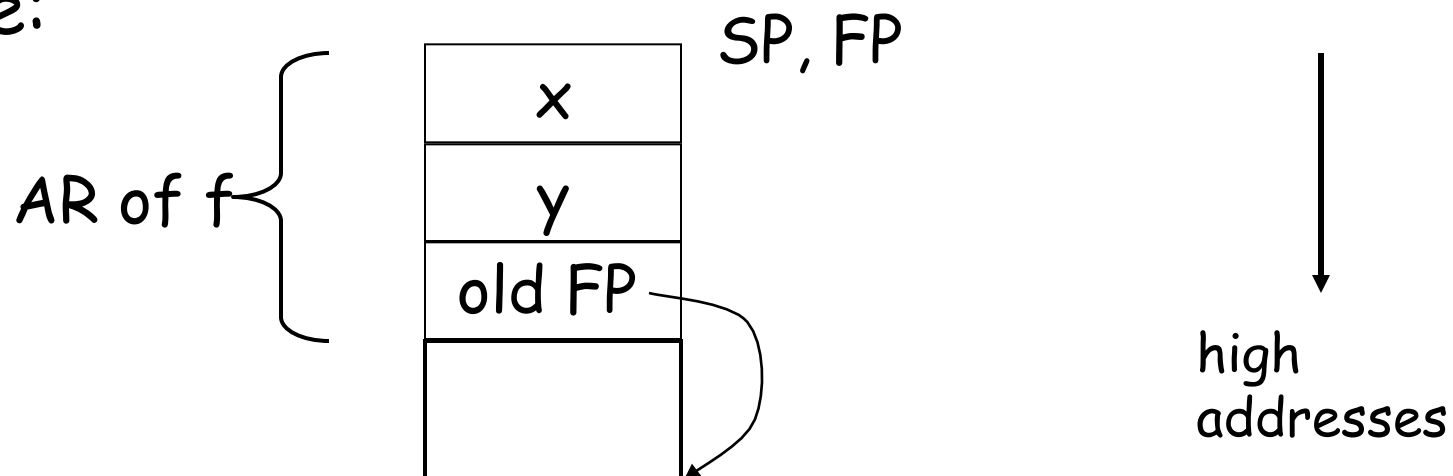
---

- The stack discipline guarantees that on function exit `$sp` is the same as it was on function entry
  - No need to save `$sp`
- We need the return address
- It's handy to have a pointer to start of 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:



## 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( $e_1, \dots, e_n$ )) =  
  push $fp  
  cgen( $e_n$ )  
  push $a0  
  ...  
  cgen( $e_1$ )  
  push $a0  
  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: `j reg`
  - Jump to address in register `reg`

`cgen(def f( $x_1, \dots, x_n$ ) = e) =`

`f_entry:`

`move $fp $sp`

`push $ra`

`cgen(e)`

`$ra  $\leftarrow$  top`

`addiu $sp $sp z`

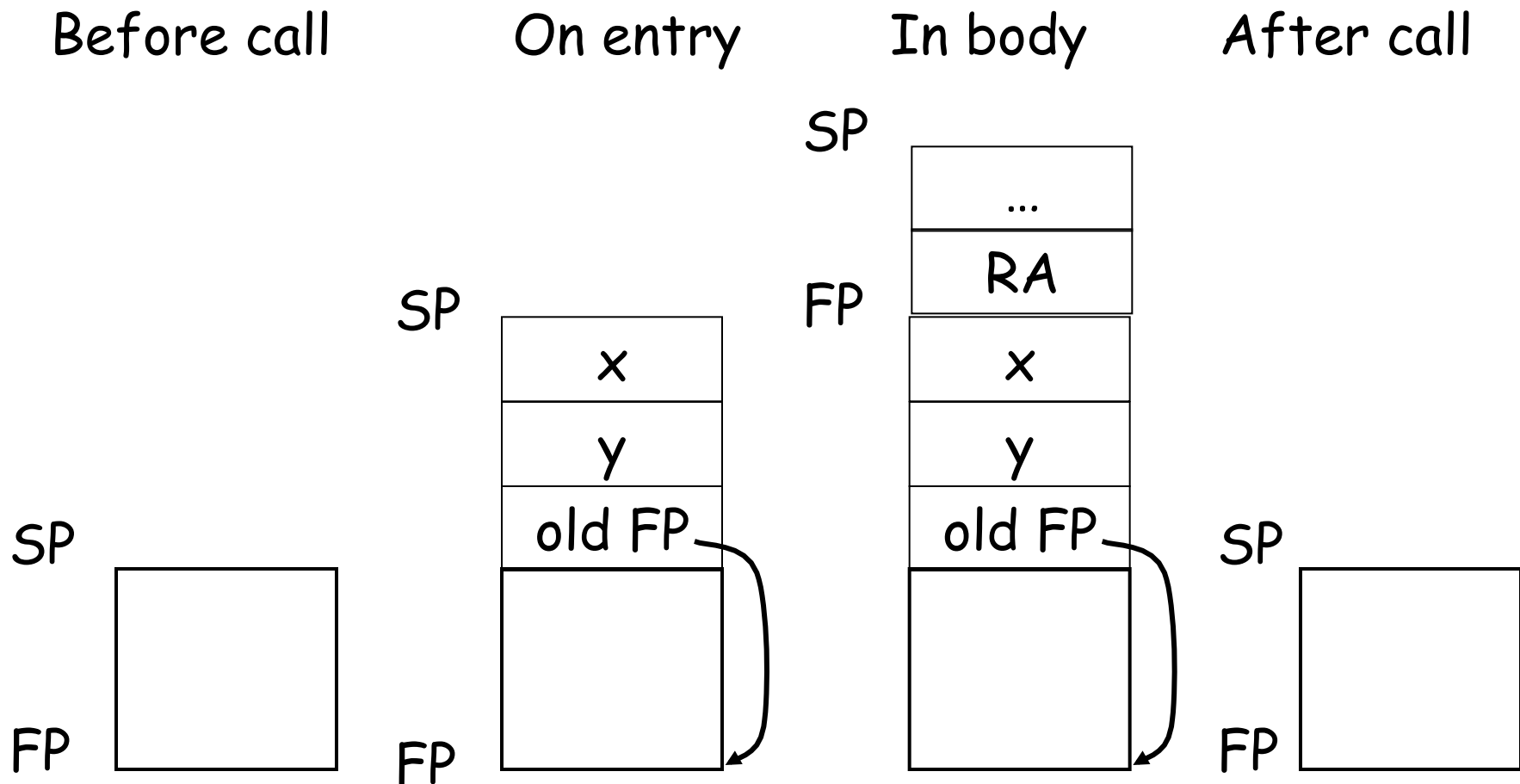
`lw $fp 0($sp)`

`j $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.)

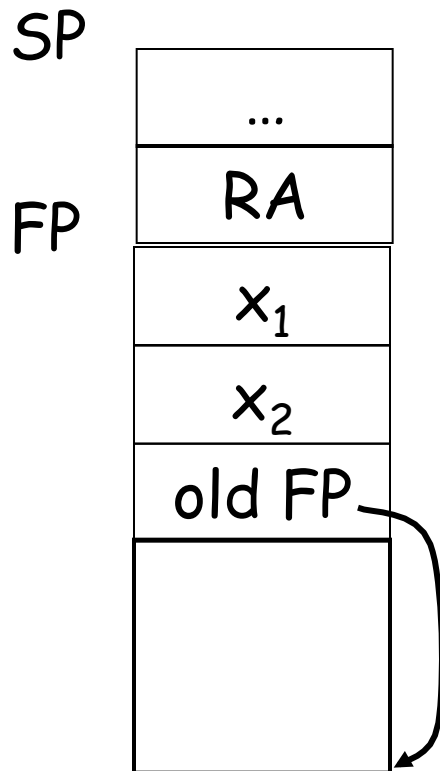
---

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

## Code Generation for Variables (Cont.)

---

- Example: For a function `def f(x1,x2) = e` the activation and frame pointer are set up as follows:



$x_1$  is at  $fp + 4$

$x_2$  is at  $fp + 8$

- Thus:

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

## 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

---

- See the Web page for a large code generation example
- 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



# **Allocating Temporaries in the AR**

# Review

---

- The stack machine has activation records and intermediate results interleaved on the stack

AR
Intermediates
AR
Intermediates

## 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

---

```
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

---

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

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

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

$$NT(\text{id}(e_1, \dots, e_n)) = \max(NT(e_1) + n - 1, \dots, NT(e_n))$$

$$NT(\text{int}) = 0$$

$$NT(\text{id}) = 0$$

Is this bottom-up or top-down?

What is  $NT(\dots \text{code for fib} \dots)$ ?

## The Revised AR

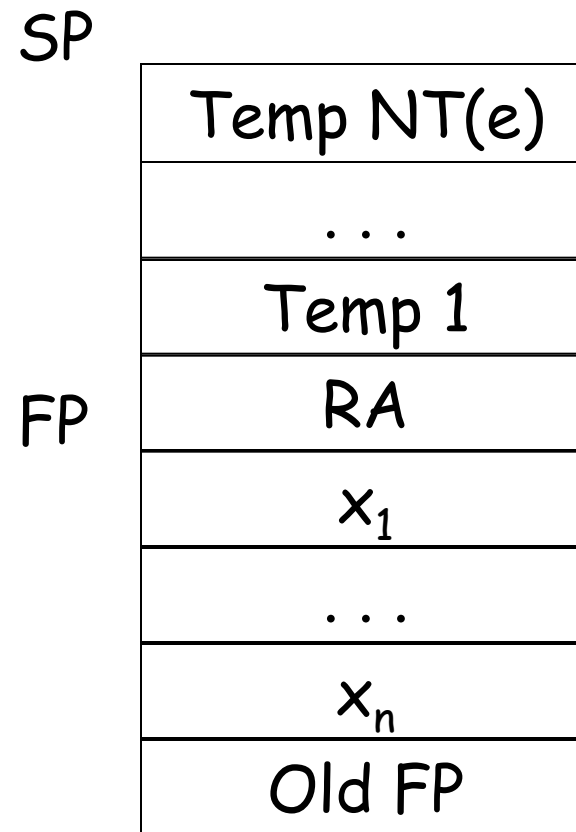
---

- 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



# Picture

---



## 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  
 $cgen(e, n)$  : generate code for  $e$  and use temporaries whose address is  $(\$fp - n)$  or lower

## Code Generation for + (original)

---

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

$\text{cgen}(e_1)$

sw \$a0 0(\$sp)

addiu \$sp \$sp -4

$\text{cgen}(e_2)$

lw \$t1 4(\$sp)

add \$a0 \$t1 \$a0

addiu \$sp \$sp 4

## 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
```

# Notes

---

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

# Code Generation for Object-Oriented Languages

# Object Layout

---

- OO implementation = Stuff from last lecture + 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 (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:

	<i>Offset</i>
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

# 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 }; // Override  
  g(): Int { a <- a - b };  
};
```

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

## 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

# 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

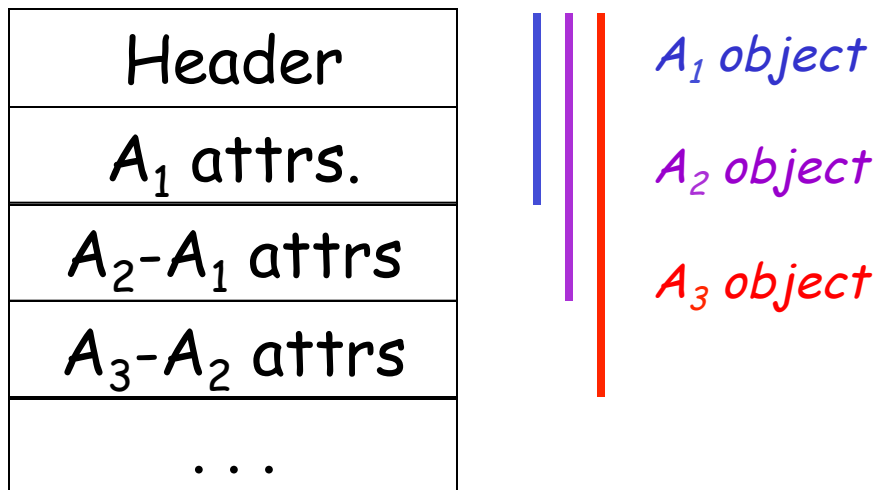
---

<div>Class Offset</div>	A	B	C
0	Atag	Btag	Ctag
4	5	6	6
8	*	*	*
12	a	a	a
16	d	d	d
20		b	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  $A_n \leq \dots \leq A_3 \leq A_2 \leq A_1$





# Dynamic Dispatch

---

- Consider again our 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 };  
};
```

# 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

# 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

---

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

- The dispatch table for class **A** has only 1 method
- The tables for **B** and **C** extend the table for **A** with more methods
- 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.)

---

- Every method must know what object is “self”
  - “self” is passed as the first argument to all methods
- To implement a dynamic dispatch  $e.f()$  we
  - Evaluate  $e$ , obtaining an object  $x$
  - Find  $D$  by reading the dispatch-table field of  $x$
  - Call  $D[\text{Offset}_f](x)$ 
    - $D$  is the dispatch table for  $x$
    - In the call,  $\text{self}$  is bound to  $x$

# Multiple Inheritance

- supplement -

# Example

---

- Assume that we extend Cool with multiple inheritance
- Consider the following 3 classes:

Class A { a : Int; m1() : Int { a }; }

Class B { b: Int; m2() : Int { b }; }

Class C inherit A, B { c : Int; m1() : Int { c }; }

- class C inherits attribute a and overrides method m1 from A, inherits attribute b and method m2 from B



# Object Layout

