CS143: Code Generation

David L. Dill

Stanford University

Code Generation

- Semantic Analysis (wrap up)
  - MIPS Architecture
  - Stack Machine on MIPS
  - Generating Code for Expressions
  - Function Calls
  - Objects

MIPS Architecture

MIPS Architecture

One of the first "RISC" CPU chips

From Stanford research project

Led by Prof. John L. Hennessy – now Stanford President

RISC = "Reduced Instruction Set Computer"

Idea: Simple, very fast instructions

Still used in embedded processors, synthesizable cores, system on a chip.

MIPS Architecture

8 bit bytes

32 general-purpose registers (32 bits each)

Arithmetic instructions

$reg_1 \leftarrow reg_2$ op reg_3

Load & Store instructions to move data between registers & memory

Documentation: SPIM manual

Some MIPS Instructions

lw reg_1 offset(reg_2) "load word"

Load 32-bit word from memory address in reg_2 + offset

add reg_1 reg_2 reg_3

$reg_1 \leftarrow reg_2 + reg_3$

sw reg_1 offset(reg_2) "store word"

Store 32 bit word in reg_1 to memory address in reg_2 + offset

Some MIPS Instructions

addiu reg_1 reg_2 imm ("add immediate")

$reg_1 \leftarrow reg_2 + imm$ ← constant

"u" means no check for overflow

li reg imm ("load immediate")

reg ← imm ← constant

move reg_1 reg_2

$reg_1 \leftarrow reg_2$

Stack Machine on MIPS

Stack Machine on MIPS

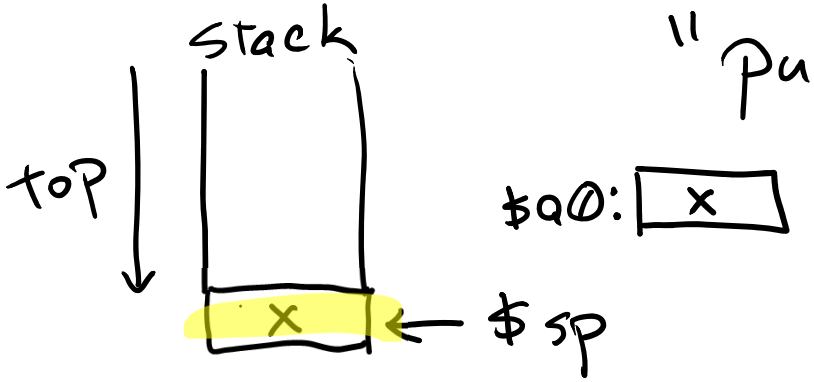
To simplify code generation, we use MIPS like a stack machine.

32-bit registers

- $a0 - accumulator
- $sp - stack pointer
- $t1 - temporary register
- $fp - frame pointer

Stack in MIPS

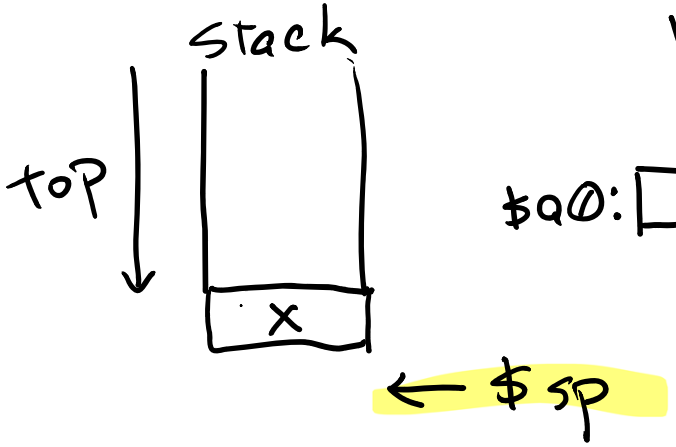
Stack grows from larger to smaller addresses,
$sp contains address of next pushed value.



"push ace" ⇒ sw $a0 0($sp)

Stack in MIPS

Stack grows from larger to smaller addresses,
$sp contains address of next pushed value.

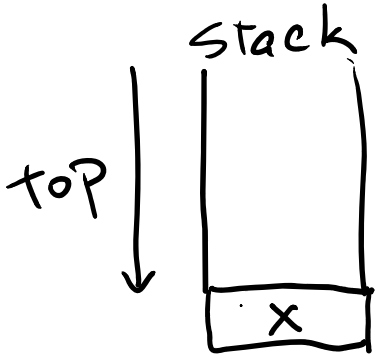


"push ace" ⇒

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

Stack in MIPS

Stack grows from larger to smaller addresses,
$sp contains address of next pushed value.



← $sp

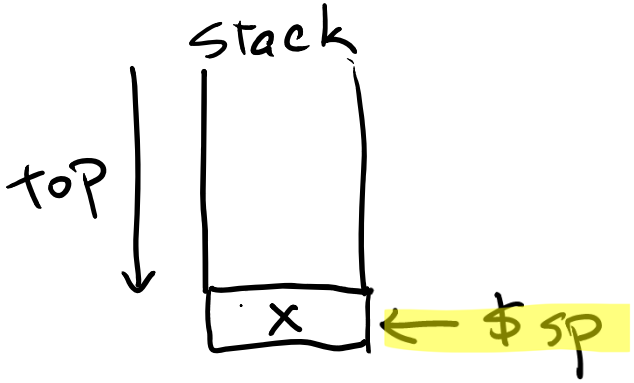
"push ace" ⇒ sw $a0 0($sp)
addiu $sp $sp -4

$t1: X

$t1 ← top_of_stack
⇒ lw $t1 4($sp)

Stack in MIPS

Stack grows from larger to smaller addresses,
$sp contains address of next pushed value.



"push ace" ⇒ sw $a0 0($sp)
addiu $sp $sp -4

$t1 ← top_of_stack
⇒ lw $t1 4($sp)

"pop" ⇒ addiu $sp $sp 4

Generating Code for Expressions

Simple Programming Language

Program → Def$^+$

Def → 'def' ID (ID$^+$)

Expr → Int | ID | Expr + Expr

| Expr − Expr

| if Expr = Expr then Expr else Expr

| ID (Expr$^+$)

Example Program

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

Generated Code for Expression

$sp and valid contents of stack before and after evaluation should be the same.

$a0 should contain the value of the expression.

Code for Constant Int

```
cgen(i):
    li $a0 i   — load i into $a0
```

Properties

- Stack does not change
- Result is in $a0

Generated Code for Expression

code for "$e_1 + e_2$":

```
<code for e1>      ← value of e1 is in $a0
sw $a0 0($sp)      ┐ push acc
addiu $sp $sp -4   ┘
<code for e2>      ← value of e2 in acc
lw $t1 4($sp)      ← $t1 ← top of stack
add $a0 $t1 $a0    ← $a0 ← $t1 + $a0
addiu $sp $sp 4    ← pop
```

Generating Code for $e_1 + e_2$

```
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"
```

Code Generation for Conditional

New instruction: beq reg_1, reg_2 label

If $reg_1 = reg_2$, jump to label

(label is a name. Assembler figures out the numerical address)

New instruction: b label

Jump to label

Code Generation for Conditional

code for "if $e_1 = e_2$ then e_3 else e_4"

```
<code for e1>
sw $a0 0($sp)
addiu $sp $sp -4
<code for e2>
lw $t1 4($sp)
add iu $sp $sp 4
```

push e_1 val on stack

pop e_1 val into $t1

(At this point, we have

$a0 = e_2

$t1 = e_1)

Code Generation for Conditional

code for "if $e_1 = e_2$ then e_3 else e_4"

```
<code for e1>
sw $a0 0($sp)
addiu $sp $sp -4
<code for e2>
lw $t1 4($sp)
add iu $sp $sp 4
beq $a0 $t1 ltrue
```

(If $a0 = $t1, jump to label "ltrue"
else, go to next instruction)

Code Generation for Conditional

code for "if $e_1 = e_2$ then e_3 else e_4"

```
<code for e1>
sw $a0 0($sp)
addiu $sp $sp -4
<code for e2>
lw $t1 4($sp)
add iu $sp $sp 4
beq $a0 $t1 ltrue
  <code for e4>
  b lend
```

does this if $a0 ≠ $t1

Code Generation for Conditional

code for "if $e_1 = e_2$ then e_3 else e_4"

```
<code for e1>
sw $a0 0($sp)
addiu $sp $sp -4
<code for e2>
lw $t1 4($sp)
add iu $sp $sp 4
beq $a0 $t1 ltrue
  <code for e4>
  b lend
```

```
ltrue: <code for e3>
```

Jumps here if $a0 = $t1

Code Generation for Conditional

code for "if $e_1 = e_2$ then e_3 else e_4"

```
<code for e1>
sw $a0 0($sp)
addiu $sp $sp -4
<code for e2>
lw $t1 4($sp)
add in $sp $sp 4
beq $a0 $t1 ltrue
  <code for e4>
  b lend
```

```
ltrue: <code for e3>
lend:
```

After executing "else" code (e_4), it jumps here, skipping "then" code.

After executing "then" code, "lend" is next.

Function Calls

Activation Record

Result is always in $a0

AR doesn't need a space

Need:

- Old frame pointer
- Actual parameter values
- Return address

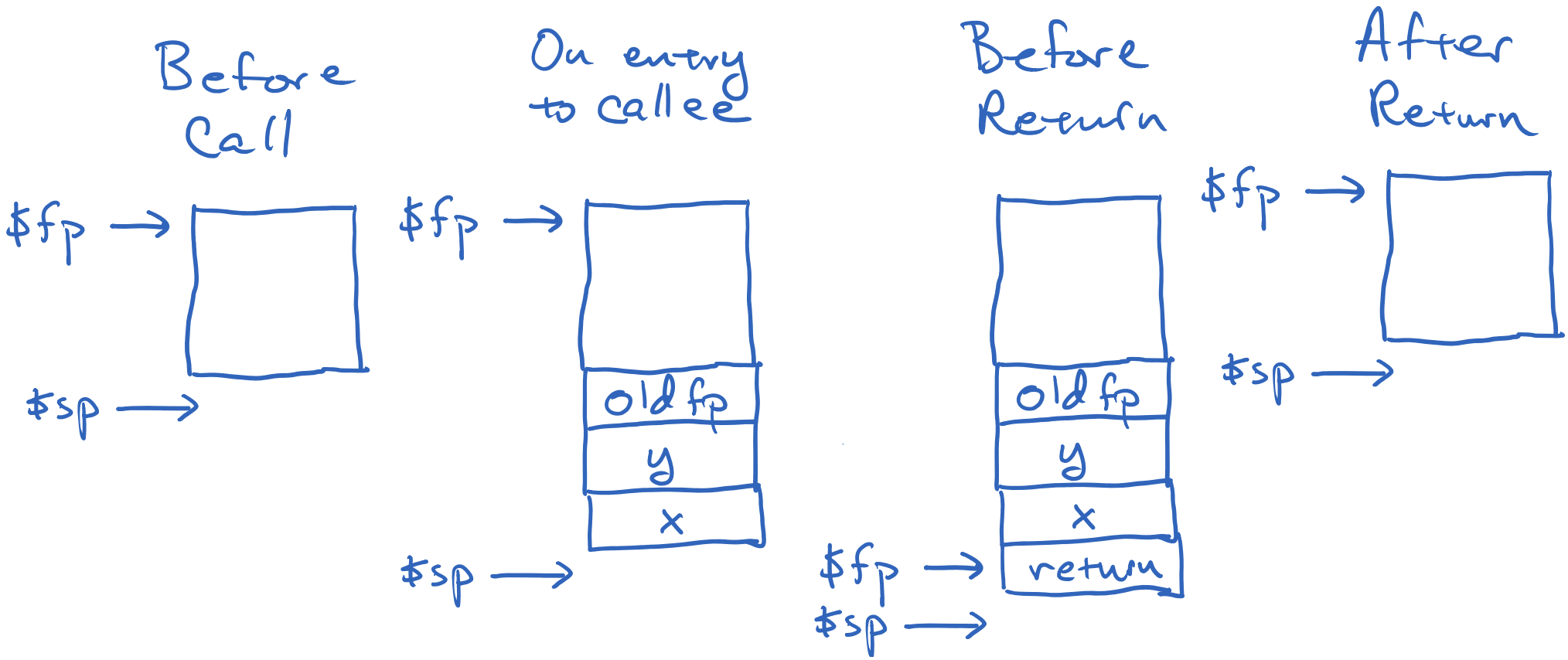
New instruction: jal label

"jump and link" – for function calls

- save address of next instruction in $ra
- jump to label

Constructing the Activation Record

Call f(x,y)

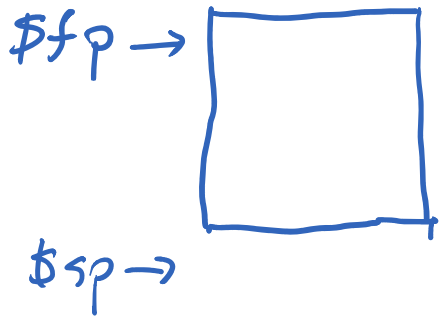


Call-side Code

code for $f(e_1, \ldots e_n)$

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

save fp

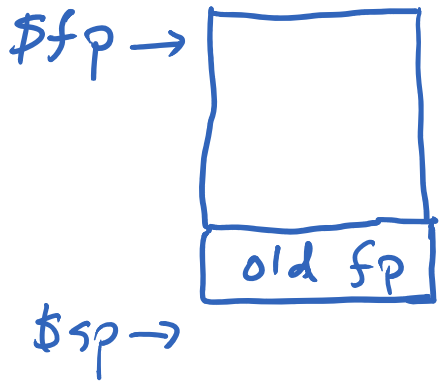


Call-side Code

code for $f(e_1, \ldots e_n)$

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

] save fp

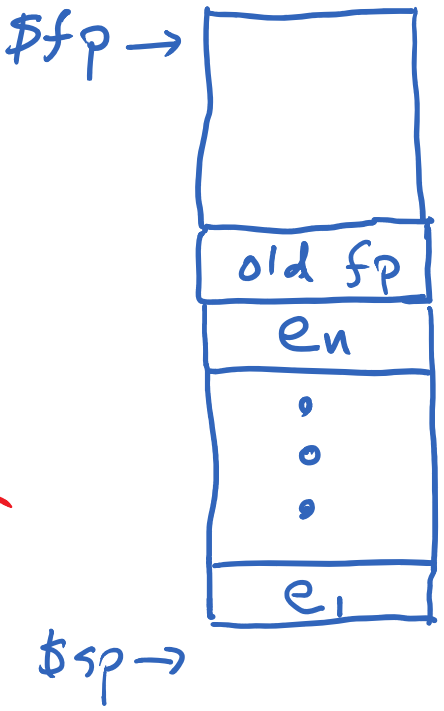


Call-side Code

```
code for f(e1, ... en)
    sw $fp 0($sp)
    addiu $sp $sp -4
    <code for en>
    sw $fp 0($sp)
    addiu $sp $sp -4
    ...
    <code for e1>
    sw $a0 0($sp)
    addiu $sp $sp -4
```

save fp

save actuals in reverse order



Call-side Code

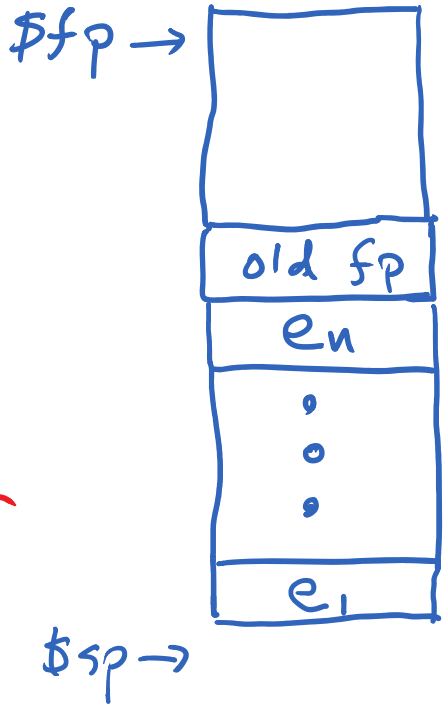
```
code for f(e1, ... en)
  sw $fp 0($sp)
  addiu $sp $sp -4
  <code for en>
  sw $fp 0($sp)
  addiu $sp $sp -4
  ...
  <code for e1>
  sw $a0 0($sp)
  addiu $sp $sp -4
  jal flabel
```

save fp

save actuals in reverse order

jump to code for f

Return address will be in $ra



New instruction: jr reg

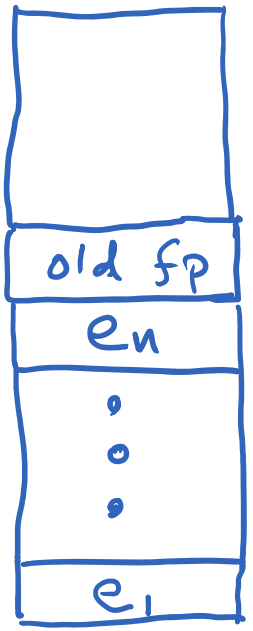
Jump to address in register

Used for return from function

Function-side Code

def $f(x_1, \ldots, x_n) = e$

move $fp $sp — fp becomes current sp



$fp = $sp →

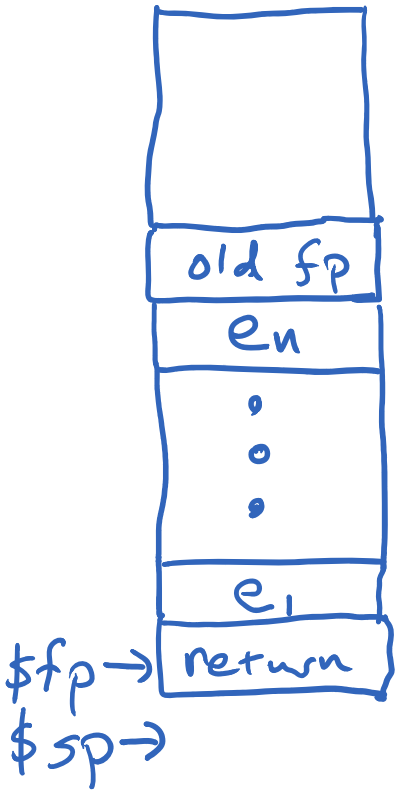
Function-side Code

def $f(x_1, \ldots, x_n) = e$

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

— fp becomes current sp

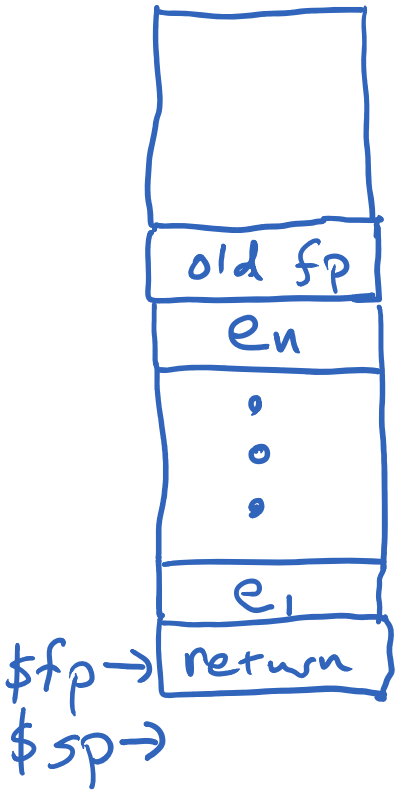
] push return address (fp points to return)



Function-side Code

def $f(x_1, \ldots, x_n) = e$

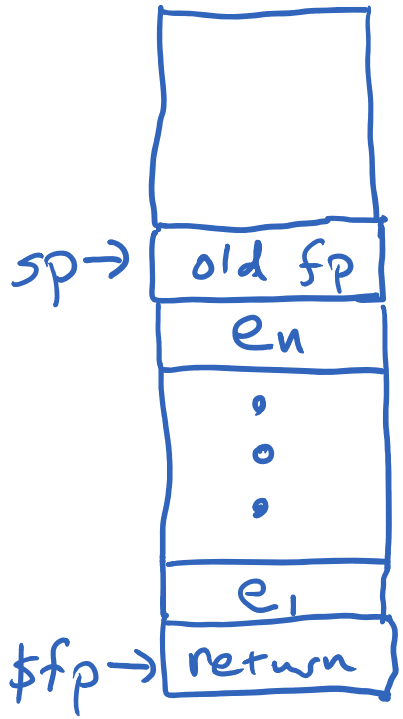
```
move $fp $sp        — fp becomes current sp
sw $ra 0($sp)       ] push return address
addiu $sp $sp -4    ]   (fp points to return)
<code for e>
lw $ra 4($sp)       — return goes in $ra
```



Function-side Code

def $f(x_1, \ldots, x_n) = e$

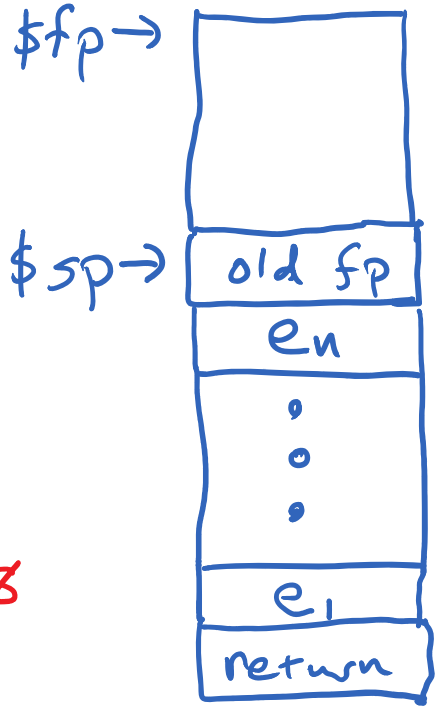
```
move $fp $sp        — fp becomes current sp
sw $ra 0($sp)       ] push return address
addiu $sp $sp -4    ]   (fp points to return)
<code for e>
lw $ra 4($sp)       — return goes in $ra
addiu $sp $sp z     — z = size of frame 4n+8
```



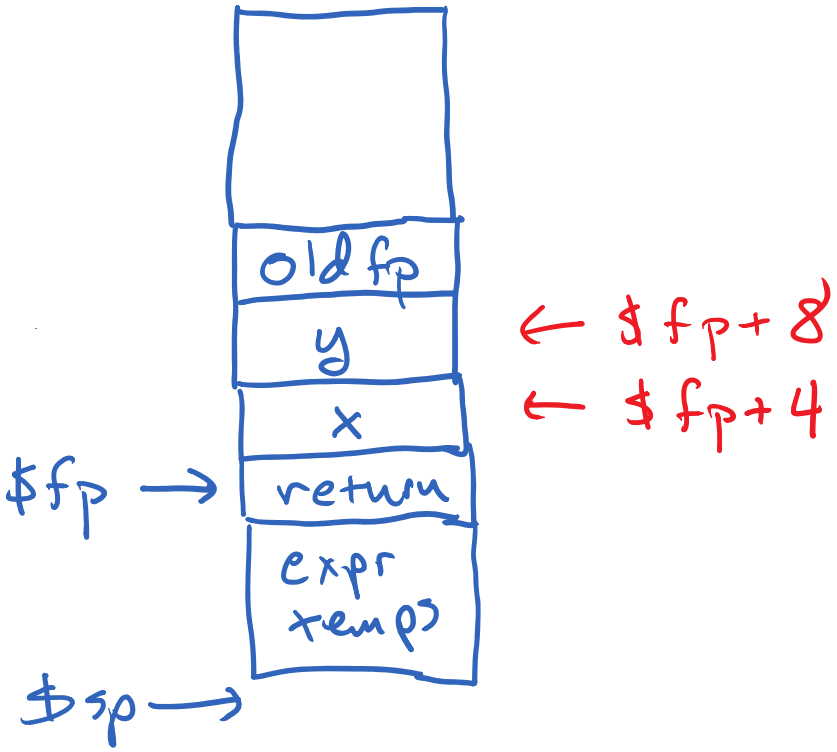
Function-side Code

def $f(x_1, \ldots, x_n) = e$

```
move $fp $sp        — fp becomes current sp
sw $ra 0($sp)       ] push return address
addiu $sp $sp -4    ]   (fp points to return)
<code for e>
lw $ra 4($sp)       — return goes in $ra
addiu $sp $sp z     — z = size of frame 4n+8
lw $fp 0($sp)       — restore old fp
jr $ra              — jump to return address
```



Accessing Variables



Code to access value of parameter #i:

```
lw $a0 4i($fp)
```

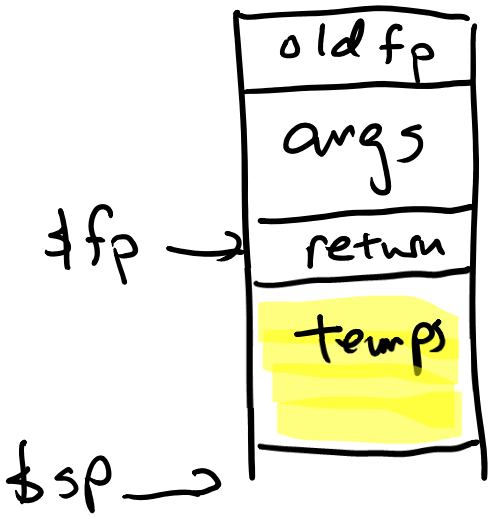
↑ compile-time constant

Allocated Temporaries

Allocated Temporaries for Expression Evaluation

Idea:

Reserve space for temps in a block.
Assign each temp a location to use



Saves instructions because MIPS takes 2 instructions to push.

Idea is useful for smarter register allocation

How Many Temporaries?

$NT(e)$ — # of temporaries required to evaluate e.

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

need to preserve value of e_1 in a temp while we use all the temps for e_2.

Number of Temps

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

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

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

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

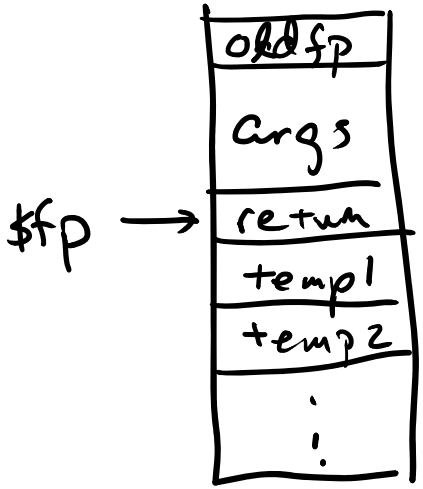
$$NT(\text{Int}) = \emptyset$$

$$NT(\text{ID}) = \emptyset$$

Code Gen with Temps

New parameter: index of next available temp

```
cgen(e1 + e2, nt)
    cgen(e1, nt)          $a0 ← e1 value
    sw $a0 nt($fp)        save in temp
    cgen(e2, nt + 4)      $a0 ← e2 value
    lw $t1 nt($fp)        $t1 ← e1 value
    add $a0 $t1 $a0       $a0 ← $t1 + $a0
```



temps are heavily reused.

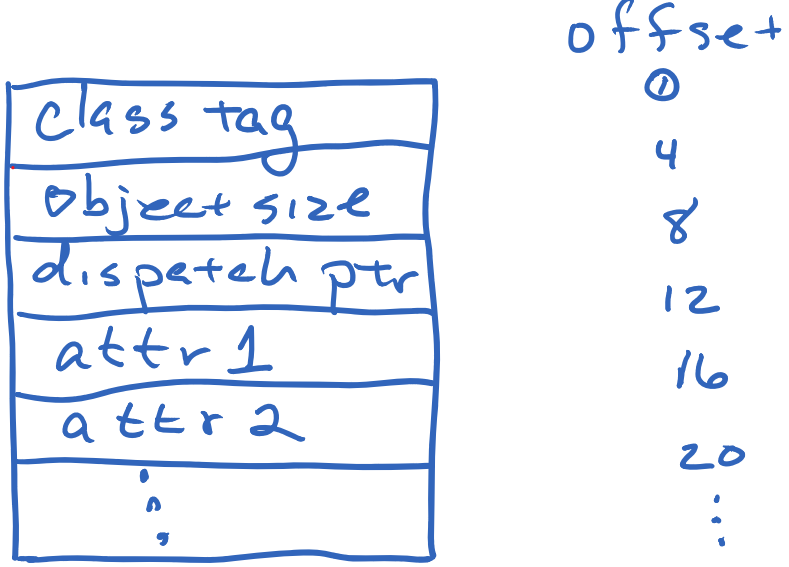
Objects

Principle

If class B $\leq$ class A, then any code that operates on an object of type A must work on an object of type B.

- Attributes inherited from A must be in same position
- Dispatch must find correct method (even if method is redefined in B)

Object Layout



Offsets are known at compile time

Size & offset of each attribute are computed by analyzing class definition

Example

```
class A {
    a : Int ← 0;
    d : Int ← 1;
    f() : { ~~ };
};
```

```
class B inherits A {
    b : Int ← 2;
    f() : { ~~ };
    g() : { ~~ };
};
```

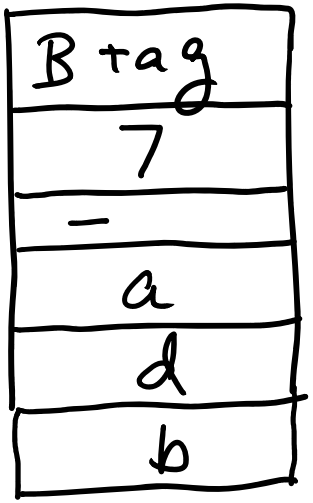
```
class C inherits A {
    c : Int ← 3;
    h() : Int { ~~ };
};
```

Layout and Inheritance

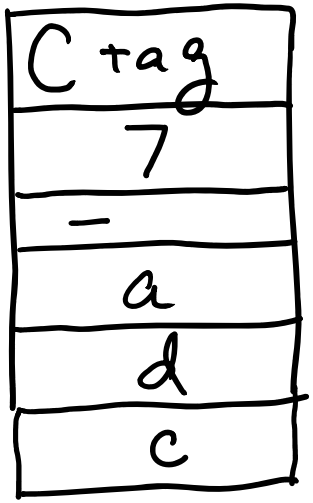
Class B inherits A ...

Instance of A

Instance of B



Instance of C



Example

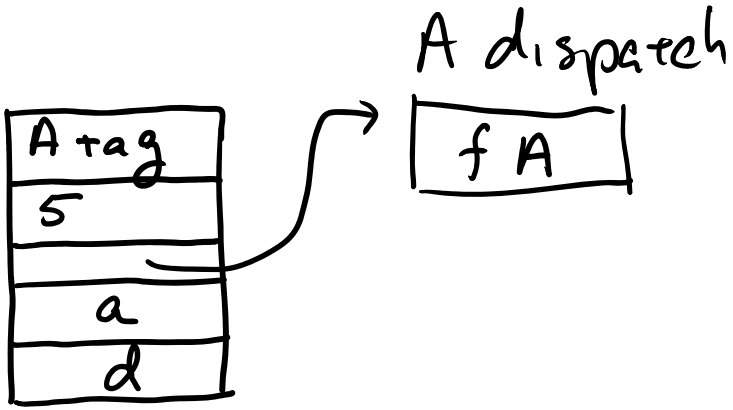
```
class A {
    a : Int ← 0;
    d : Int ← 1;
    f() : { ~~ };
};
```

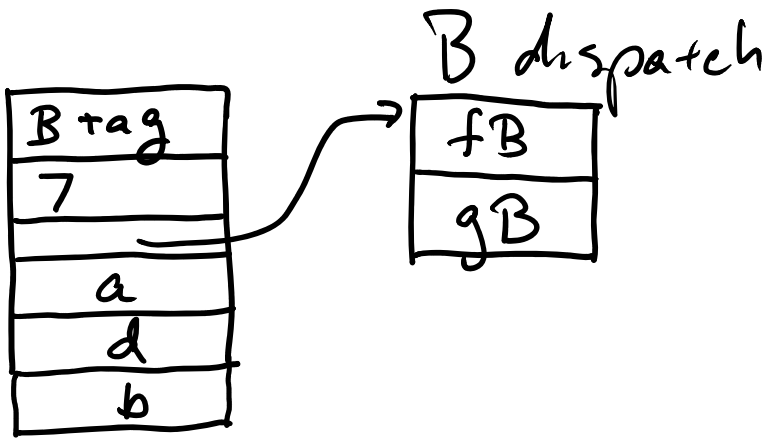
```
class B inherits A {
    b : Int ← 2;
    f() : { ~~ };
    g() : { ~~ };
};
```

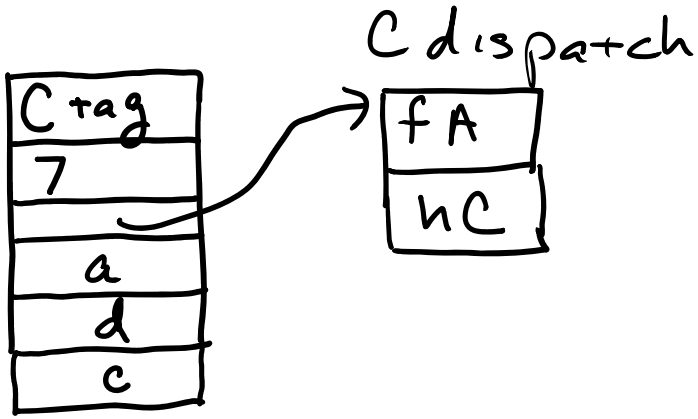
```
class C inherits A {
    c : Int ← 3;
    h() : Int { ~~ };
};
```

Dispatch

Want f at same offset in dispatch table, whether inherited or redefined.







Using Dispatch Tables

Dynamic dispatch e.f()

Code:

evaluate e → ptr to object
get ptr to dispatch table
get ptr to method from dispatch table
call method with self = e value