Introduction to Compiler Design

Lesson 17:

Code Generation, part 2

How to be a MIPS Master

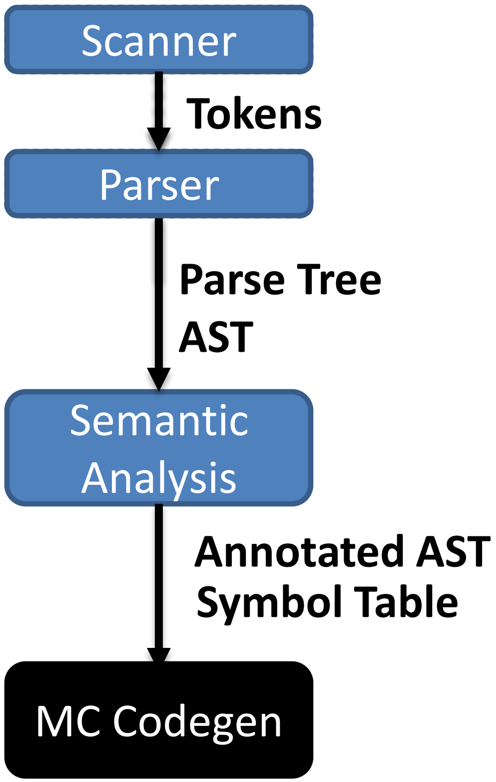
- It’s really easy to get confused with assembly
- Some suggestions
  – Start simple: main procedure with “print(1);”
    - Get procedure main to compile and run
      – Function prologue and epilog
    - Trivial case of expressions: evaluating the constant 1, which pushes a 1 on the stack
    - Printing: print(1);
  – Then grow your compiler incrementally
    - Expressions
    - Control constructes
    - Call/return

How to be a MIPS Master

- More suggestions
  - Try writing the desired assembly code by hand before having the compiler generate it
  - Draw pictures of program flow
  - Have your compiler put in detailed comments in the assembly code it emits!
- It's really easy to get confused with assembly

Roadmap

- Last:
  - Talked about compiler backend design points
  - Decided to go directly from AST to machine code for our language
- Now:
  - Discuss what the actual codegen pass should look like



Review: Global Variables

- Showed you one way to do declaration last time:

  .data

  .align 2

  _name: .space 4

- Simpler form for primitives:

  .data

  _name: .word <value>

Review: Functions

- Preamble
  - Sort of like the function signature
- Prologue
  - Set up the AR
- Body
  - Do the thing
- Epilogue
  - Tear down the AR

Function Preambles

```
int f(int a, int
b){
    int c = a + b;
    int d = c - 7;
    return c;
}
```

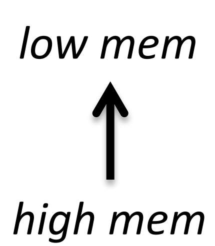
```
.text
_f:
#... Function
body ...
```

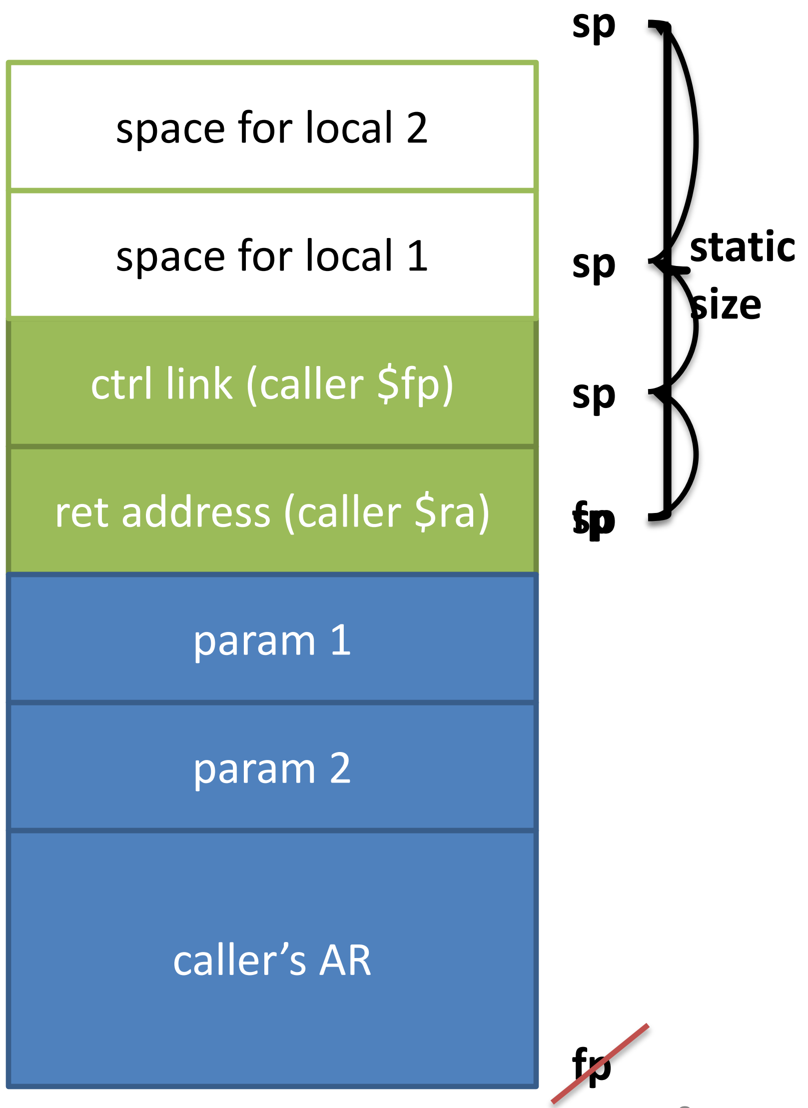
This label gives us something to jump to

```
jal _f
```

Function Prologue

- Recall our view of the Activation Record
  1. save the return address
  2. save the frame pointer
  3. make space for locals
  4. update the frame ptr





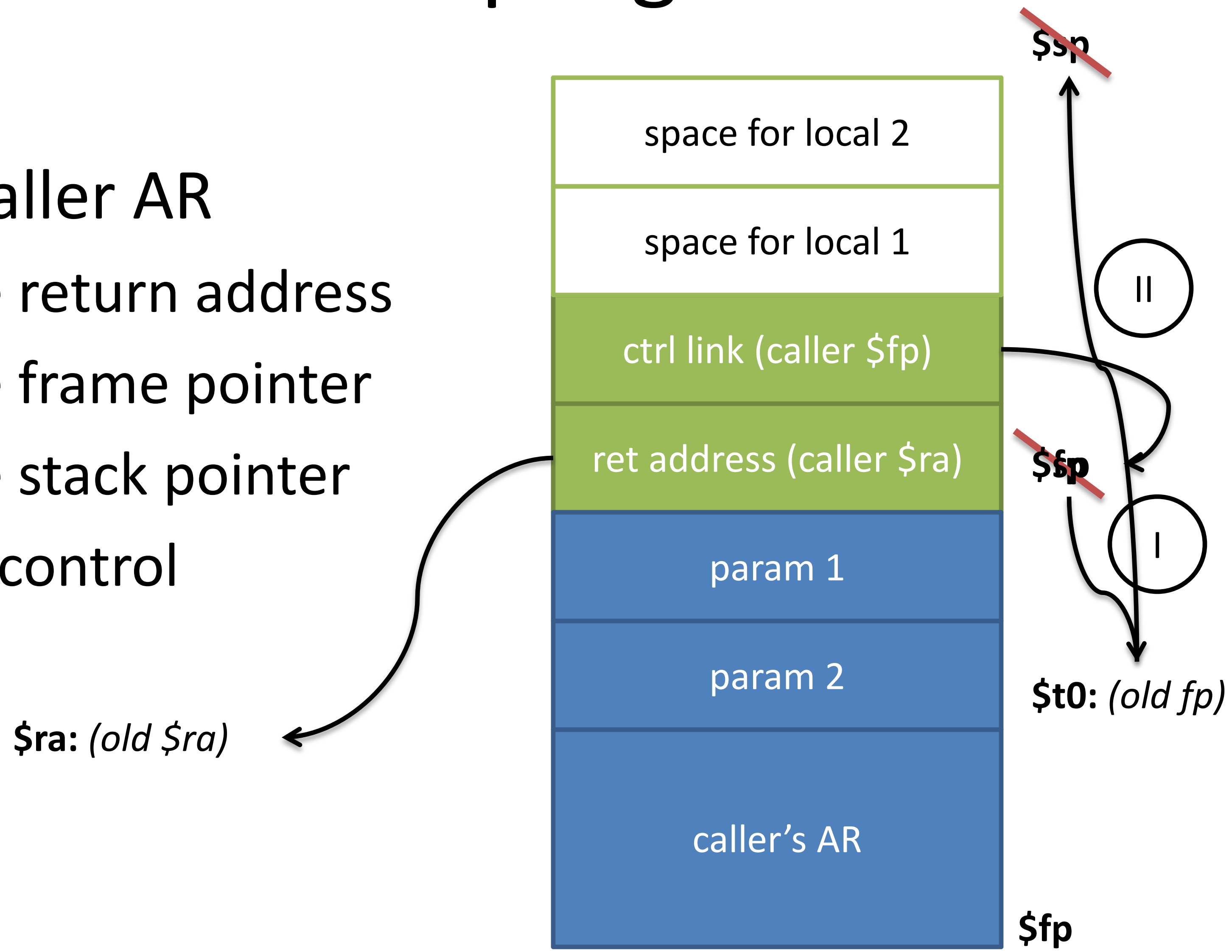
Function Prologue: MIPS

- Recall our view of the Activation Record
  1. save the return address
  2. save the frame pointer
  3. make space for locals
  4. update the frame pointer

```
.text
_f:
  sw $ra 0($sp)     #call lnk (*sp = ra)
  subu $sp $sp 4    #push (sp -= 4)
  sw $fp 0($sp)     #ctrl lnk (*sp = fp)
  subu $sp $sp 4    #push (sp -= 4)
  subu $sp $sp 8    #locals (sp -= 8)
  addu $fp $sp 16   #update(fp = sp+16)
```

Function Epilogue

- Restore Caller AR
  1. restore return address
  2. restore frame pointer
  3. restore stack pointer
  4. return control



Function Epilogue: MIPS

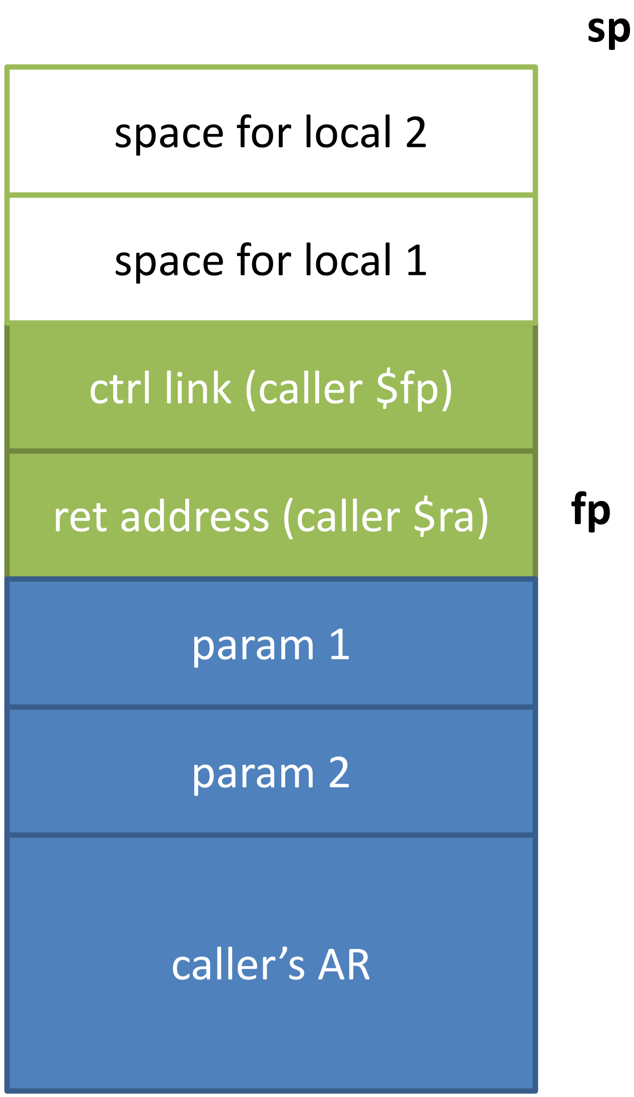
- Restore Caller AR
  1. restore return address
  2. restore frame pointer
  3. restore stack pointer
  4. return control

```
.text
_f:
  sw $ra 0($sp)
  subu $sp $sp 4
  sw $fp 0($sp)
  subu $sp $sp 4
  subu $sp $sp 8
  addu $fp $sp 16
  #... Function body ...
  lw $ra, 0($fp) #ra = *fp
  move $t0, $fp  #t0 = fp
  lw $fp, -4($fp)#fp = *(fp-4)
  move $sp, $t0  #sp = t0
  jr $ra
```

Function Body

- Obviously, quite different based on content
  - Higher-level data constructs
    - Loading parameters, setting return
    - Evaluating expressions
  - Higher-level control constructs
    - Performing a call
    - While loops
    - If-then and if-then-else statements

Function Locals

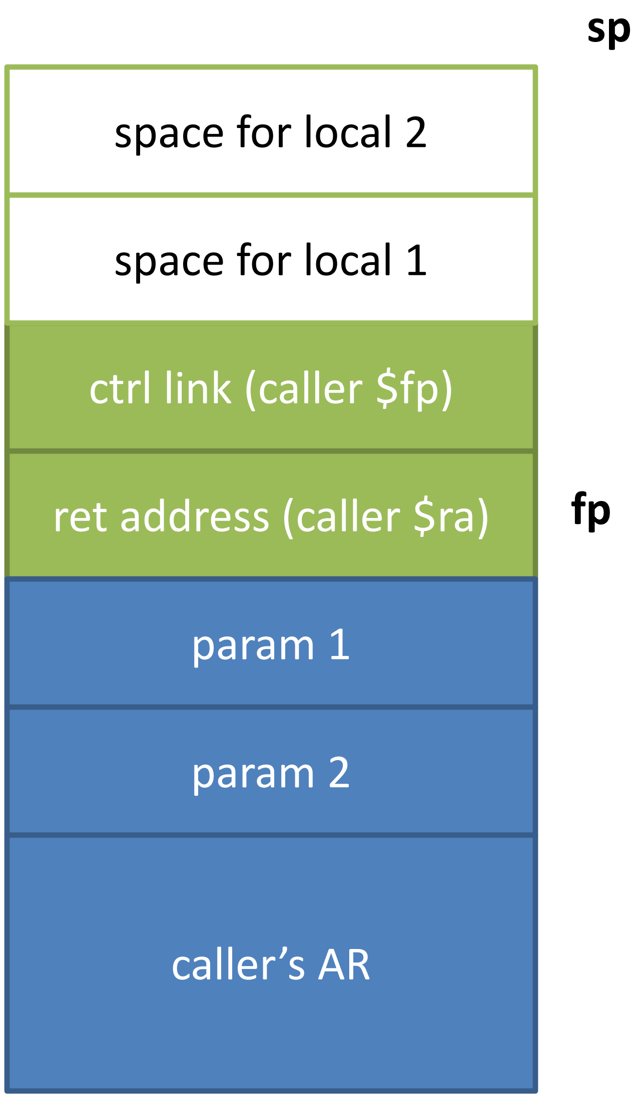



```
.text
_f:
    # … prologue … #
    lw $t0, -8($fp)
    lw $t1, -12($fp)



    # … epilogue … #
```

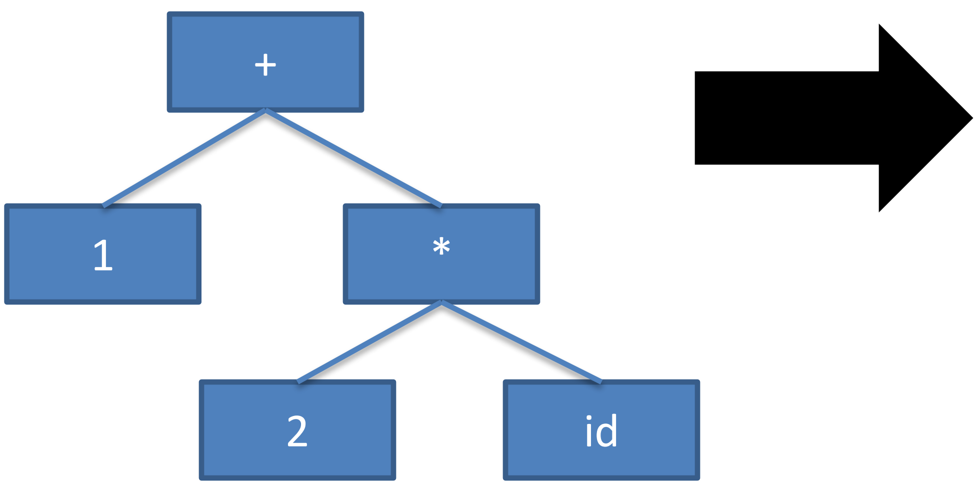
Function Returns




```
.text
_f:
    # … prologue … #
    lw $t0, -8($fp)
    lw $t1, -12($fp)
    lw $v0, -8($fp)
    j _f_exit
_f_exit:
    # … epilogue … #
```

Function Body: Expressions

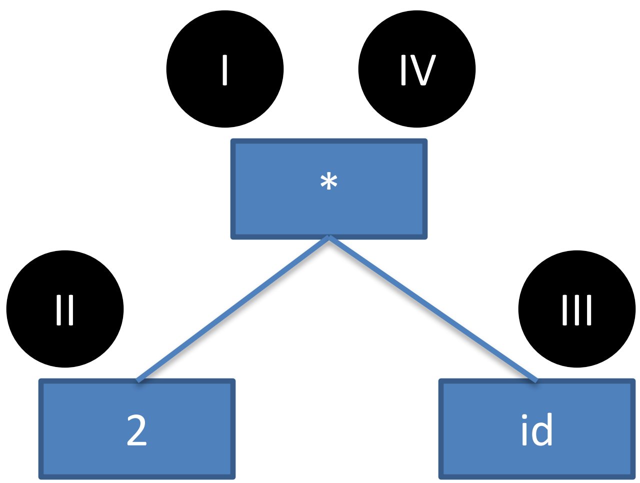
- Goal
  - Linearize ("flatten") an expression tree
- Use the same insight as SDT during parsing
  - Use a work stack and a post-order traversal



Visit 1
Visit 2
Visit id
Visit *
Visit +

Linearized Pseudocode

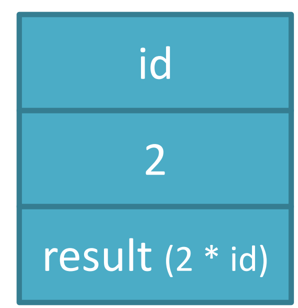
- Key insights
  - Use the stack-pointer location as “scratch space”
  - At operands: push value onto the stack
  - At operators: pop source values from stack, push result



Push the *value* of id!

```
$t1 = id
$t0 = 2    2 * id
```

```
push 2
push id
pop id into t1
pop 2 into t0
mult t0 * t1 into t0
push t0
```



Linearized MIPS

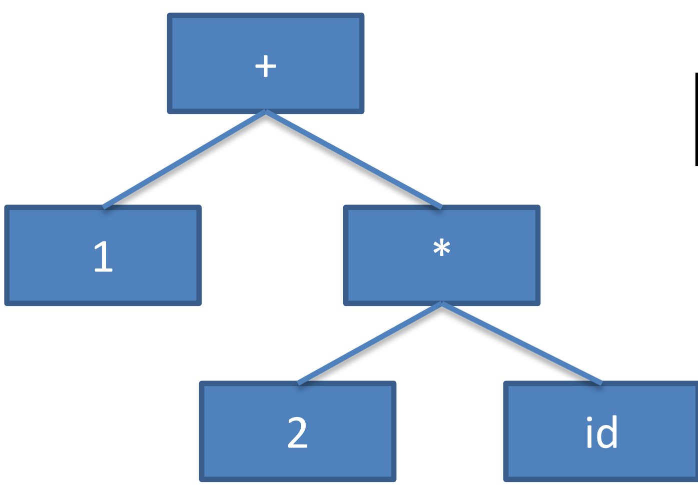
```
.data
    _id: .word <value>

.text
L1: push 2
L2: push id
L3: pop id into t1
L4: pop 2 into t0
L5: mult t0 * t1 into t0
L6: push t0
```

```
L1: li $t0 2
    sw $t0 0($sp)
    subu $sp $sp 4
L2: lw $t0 _id
    sw $t0 0($sp)
    subu $sp $sp 4
L3: lw $t1 4($sp)
    addu $sp $sp 4
L4: lw $t0 4($sp)
    addu $sp $sp 4
L5: mult $t0 $t0 $t1
L6: sw $t0 0($sp)
    subu $sp $sp 4
```

Function Body: Expressions

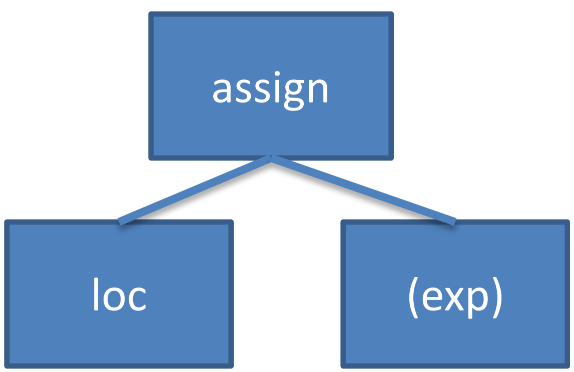
- Goal
  - Linearize ("flatten") an expression tree
- Use the same insight as SDT during parsing
  - Use a work stack and a post-order traversal



Visit 1	push 1
Visit 2	push 2
Visit id	push value of id
Visit *	pop id into t1
	pop 2 into t0
	mult t0 * t1 into t0
	push t0
Visit +	pop into t1
	pop 1 into t0
	add t0 + t1 into t0
	push t0

Assignment Statements

- By the end of the expression, the stack isn't exactly as we found it
  - Contains the value of the expression
  - This organization is intentional

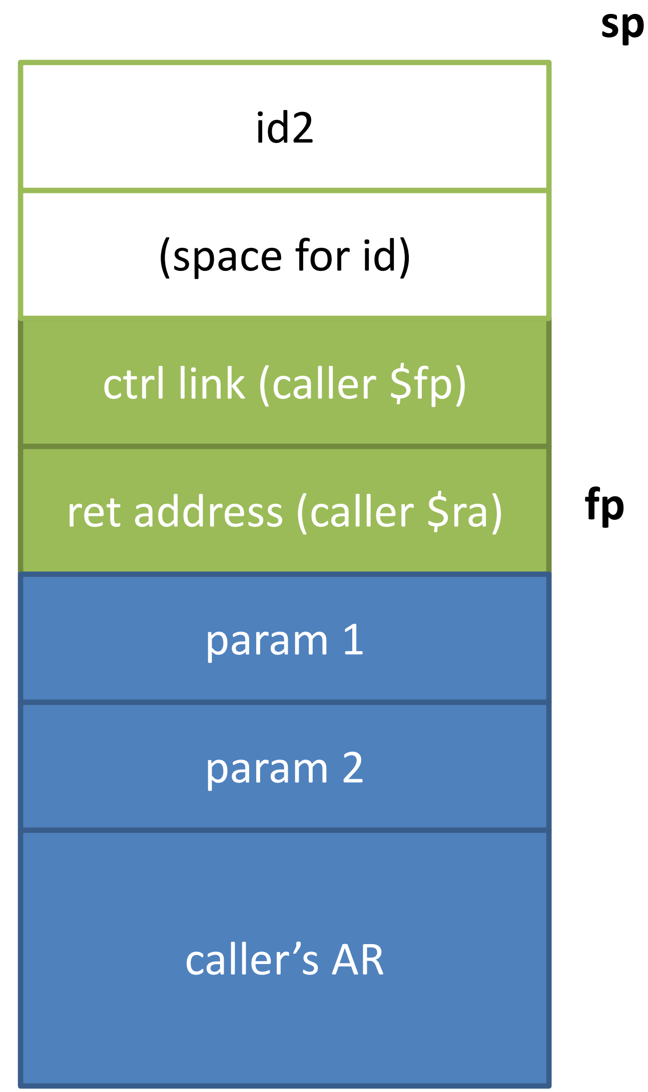


1) Compute address of LHS *location*; leave result on stack
2) Compute value of RHS expr; leave result on stack
3) Pop RHS into $t1
4) Pop LHS into $t0
5) Store value $t1 at the address held in $t0

Simple Assignment

- Generate stack-machine style MIPS code for

  id = 1 + 2;



Algorithm

1) Compute address of LHS *location*; leave result on stack
2) Compute value of RHS expr; leave result on stack
3) Pop RHS into $t1
4) Pop LHS into $t0
5) Store value $t1 at the address held in $t0

Dot Access

- Fortunately, we know the offset from the base of a `struct` to a certain field statically
  - The compiler can do the math for the slot address
  - This isn't true for languages with pointers!

```
struct Inner{
  bool hi;
  int there;
  int c;
};

struct Demo{
  struct Inner b;
  int val;
};
```

```
struct Demo inst;
struct Demo inst2;
inst.b.c = inst2.b.c + 1;
```

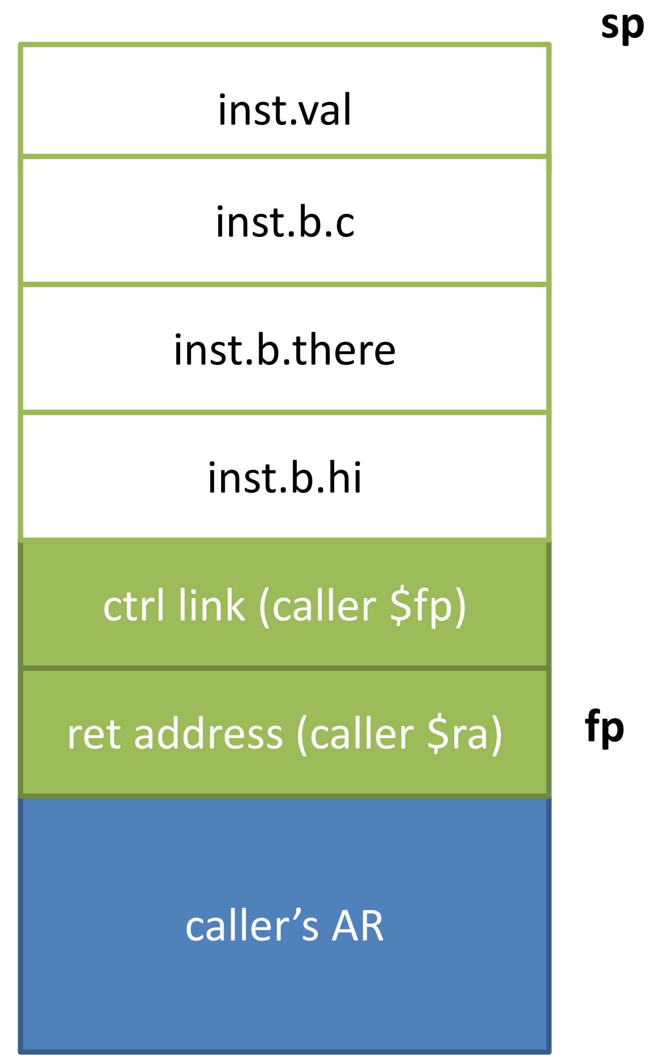
load this address   load this value

Dot Access Example

```
void v(){
   struct Inner{
     bool hi;
     int there;
     int c;
   };
   struct Demo{
     struct Inner b;
     int val;
   };
   struct Demo inst;
   … = inst.b.c;
}  inst.b.c = …;
```

inst is based at $fp-8

field b.c is -8 off the base




```
LHS
subu $t0 $fp 16
sw $t0 0($sp)
subu $sp $sp 4
```

```
RHS
lw $t0 -16($fp)
sw $t0 0($sp)
subu $sp $sp 4
```

Control-Flow Constructs

- Function Calls
- Loops
- Ifs

Function Call

- Two tasks:
  - Put argument *values* on the stack (pass-by-value semantics)
  - Jump to the callee preamble label
  - Bonus 3rd task: save *live* registers
    - (We don't have any in a stack machine)
- On return
  - Tear down the actual parameters
  - Retrieve and push the result value

Function-Call Example

```
int f(int arg1, int arg2){
  return 2;
}
int main(){
  int a;
  a = f(a,4);
}
```

```
li $t0 4           # push arg 2
sw $t0 0($sp)      #
subu $sp $sp 4     #
lw $t0 -8($fp)     # push arg 1
sw $t0 0($sp)      #
subu $sp $sp 4     #
jal _f             # call f (via jump and link)
addu $sp $sp 8     # tear down actual parameters
sw $v0 0($sp)      # retrieve and push the result
subu $sp $sp 4     #
```

Generating If-Then[-Else] Statements

- First, obtain names to use for the labels of the
  - [false branch]
  - successor
- Generate code for the branch condition
  - Can emit a jump to the (not-yet placed!) false-branch label
- Generate code for the true branch
  - Emit the code for the body of the true branch
  - [Emit a jump to the (not-yet placed!) successor label]
- [Generate code for the false branch (similar to the true branch)
  - Emit the false-branch label
  - Emit the code for the body of the false branch]
- Emit the successor label

If-Then Statement Example

```
…
if (val == 1){
   val = 2;
}
…
```

```
  lw $t0 _val          # evaluate condition LHS
  sw $t0 0($sp)        # push onto stack
  subu $sp $sp 4       #
  li $t0 1             # evaluate condition RHS
  sw $t0 0($sp)        # push onto stack
  subu $sp $sp 4       #
  lw $t1 4($sp)        # pop RHS into $t1
  addu $sp $sp 4       #
  lw $t0 4($sp)        # pop LHS into $t0
  addu $sp $sp 4       #
  bne $t0 $t1 L_0      # branch if condition false
  li $t0 2             # true branch
  sw $t0 _val
  j L_0                # end true branch
L_0:                   # successor label
  …
```

If-Then-Else Statement Example

```
…
if (val == 1){
   val = 2;
} else {
   val = 3;
}
…
```

```
  lw $t0 _val         # evaluate condition LHS
  sw $t0 0($sp)       # push onto stack
  subu $sp $sp 4      #
  li $t0 1            # evaluate condition RHS
  sw $t0 0($sp)       # push onto stack
  subu $sp $sp 4      #
  lw $t1 4($sp)       # pop RHS into $t1
  addu $sp $sp 4      #
  lw $t0 4($sp)       # pop LHS into $t0
  addu $sp $sp 4      #
  bne $t0 $t1 L_1     # branch if condition false
  li $t0 2            # true branch
  sw $t0 _val
  j L_0               # end true branch
L_1:
  li $t0 3            # false branch
  sw $t0 _val
L_0:                  # successor label
```

Generating While Loops

- Very similar to if-then statements
  - Obtain several labels to use for the
    - Head of the loop
    - Successor of the loop
- At the end of the loop body
  - Unconditionally jump back to the head

While-Loop Example

```
while (val == 1){
   val = 2;
}
```

```
L_0:
  lw $t0 _val        # evaluate condition LHS
  sw $t0 0($sp)      # push onto stack
  subu $sp $sp 4     #
  li $t0 1           # evaluate condition RHS
  sw $t0 0($sp)      # push onto stack
  subu $sp $sp 4     #
  lw $t1 4($sp)      # pop RHS into $t1
  addu $sp $sp 4     #
  lw $t0 4($sp)      # pop LHS into $t0
  addu $sp $sp 4     #
  bne $t0 $t1 L_1    # branch if condition false
  li $t0 2           # Loop body
  sw $t0 _val
  j L_0              # jump to loop head
L_1:                 # Loop successor
  …
```

Helper Functions

- Generate (opcode, …args…)
  - Generate("add", "T0", "T0", "T1")
    - writes out `add $t0, $t0, $t1`
  - Versions for fewer args as well
- Generate indexed (opcode, "Reg1", "Reg2", offset)
- GenPush(reg) / GenPop(reg)
- NextLabel() – Used to obtain a unique label
- GenLabel(L) – Places a label

MIPS System Calls

(from SPIM S20: A MIPS R2000 Simulator, James J. Larus, University of Wisconsin-Madison)

SPIM provides a small set of operating-system-like services through the MIPS system call (syscall) instruction. To request a service, a program loads the system call code (see Table below) into register $v0 and the arguments into registers $a0, ..., $a3 (or $f12 for floating point values). System calls that return values put their result in register $v0 (or $f0 for floating point results).

Service	**System Call Code**	**Arguments**	**Result**
print integer	1	$a0 = value	(none)
print float	2	$f12 = float value	(none)
print double	3	$f12 = double value	(none)
print string	4	$a0 = address of string	(none)
read integer	5	(none)	$v0 = value read
read float	6	(none)	$f0 = value read
read double	7	(none)	$f0 = value read
read string	8	$a0 = address where string to be stored $a1 = number of characters to read + 1	(none)
memory allocation	9	$a0 = number of bytes of storage desired	$v0 = address of block
exit (end of program)	10	(none)	(none)
print character	11	$a0 = integer	(none)
read character	12	(none)	char in $v0

MIPS System Calls

To print "the answer = 5", use the commands:

```
.data str: .asciiz "the answer = “
.text
   li $v0, 4          # $system call code for print_str
   la $a0, str        # $address of string to print
   syscall            # print the string
   li $v0, 1          # $system call code for print_int
   li $a0, 5          # $integer to print
   syscall            # print it
```

- **print int** passes an integer and prints it on the console
- **print float** prints a single floating point number
- **print double** prints a double precision number
- **print string** passes a pointer to a null-terminated string
- **read int**, **read float**, and **read double** read an entire line of input up to and including a newline.
- **read string** has the same semantics as the Unix library routine fgets. It reads up to n - 1 characters into a buffer and terminates the string with a null byte. If there are fewer characters on the current line, it reads through the newline and again null-terminates the string.
- **sbrk** returns a pointer to a block of memory containing *n* additional bytes
- **exit** stops a program from running

(from SPIM S20: A MIPS R2000 Simulator, James J. Larus, University of Wisconsin-Madison)

Summary

- Now:
  - Got the basics of MIPS
  - CodeGen for *most* AST node types
- Next:
  - Do the rest of the AST nodes
  - Introduce control-flow graphs