Lecture #27

Code Generation

Code Generation

- MIPS instruction: `beq` reg_1 reg_2 `label`
  - Branch to label if $reg_1 = reg_2$
- MIPS instruction: `b label`
  - Unconditional branch to label

- `cgen(if` e_1 `=` e_2 `then` e_3 `else` e_4`) =`

```
    cgen(e1)
    sw $a0 0($sp)
    addiu $sp $sp - 4
    cgen(e2)
    lw $t1 4($sp)
    addiu $sp $sp 4
    beq $a0 $t1 True_branch
False_branch:
    cgen(e4)
    b end_if
True_branch:
    cgen(e3)
End_if:
```

Code Generation – Function Calls

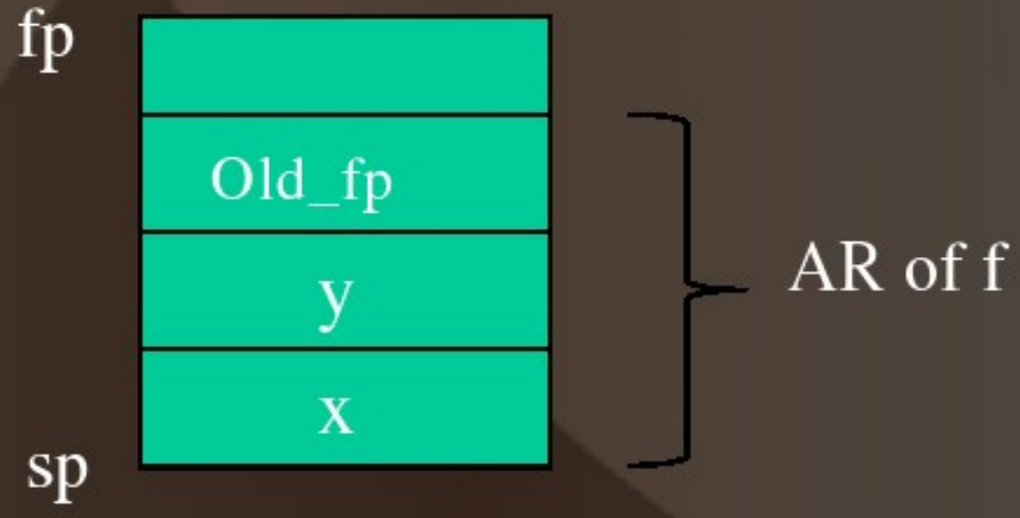
- Code for function calls and function definitions depends on the layout of the AR

- 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 stack discipline guarantees that on function exit $sp is the same as it was on function entry
  - No need for a control link

Code Generation – Function Calls

- A pointer to the current activation is useful
  - This pointer lives in register $fp (frame pointer)
- So, for this language, an AR with the caller's frame pointer, the actual parameters, and the return address suffices

- Consider a call to f(x,y), the AR is:



Code Generation – Function Calls

- The calling sequence is the instructions (of both caller and callee) to set up a function invocation

- New MIPS instruction: jal label
  - Jump to label, save address of next instruction in $ra
  - To be used in Caller

- New MIPS instruction: jr reg
  - Jump to address in register reg
  - To be used in Callee

Code Generation – Function Calls

Code in Caller

```
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
- Finally the caller saves the return address in register $ra
- The AR so far is 4*n+4 bytes long

Code Generation – Function Calls

Code in Callee

```
cgen(def f(x1,…,xn)= e) =
F_entry:
      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
```

- 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

Code Generation – Function Calls

- 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**
- Solution: use a frame pointer
  - Always points to the return address on the stack

- Let x_i be the i^{th} (i = 1,…,n) formal parameter of the function for which code is being generated

```
cgen(xi) = lw $a0 z($fp)  (z = 4*i)
```

Code Generation

- In production compilers:
  - Emphasis is on keeping values in registers
    - Especially the current stack frame
  - Intermediate results are laid out in the AR, not pushed and popped from the stack