Code Generation

- We focus on generating code for a stack machine with accumulator
- We want to run the resulting code on a real machine
  - e.g., the MIPS processor (or simulator)
- We simulate stack machine instructions using MIPS instructions and registers

- The accumulator is kept in MIPS register $a0
- The stack is kept in memory
  - The stack grows towards lower addresses
  - Standard convention on MIPS
- 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

Notice that $sp points to the next unused memory, and since the stack grows towards lower addresses the top of the stack is at $sp+4

MIPS architecture

- Prototypical Reduced Instruction Set Computer (RISC)
- Most operations use registers for operands & results
- Use load & store instructions to use values in memory
- 32 general purpose registers (32 bits each)
  - We use $sp, $a0 and $t1 (a temporary register)

- Read the SPIM documentation for details

SPIM is the simulator that we can use to execute our generated code.

5 MIPS instructions that we will need for our first example :

- 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
  - $reg \leftarrow imm$

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

```
acc ← 7
push acc

acc ← 5
acc ← acc + top_of_stack

pop
```

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
```

A language with integers and integer operations

$$P \rightarrow D; P \mid D$$

$$D \rightarrow \text{def id(ARGS)} = E;$$

$$\text{ARGS} \rightarrow \text{id, ARGS} \mid \text{id}$$

$$E \rightarrow \text{int} \mid \text{id} \mid \text{if } E_1 = E_2 \text{ then } E_3 \text{ else } E_4$$

$$\mid E_1 + E_2 \mid E_1 - E_2 \mid \text{id}(E_1,\ldots,E_n)$$

- The first function definition f is the entry point
  - The "main" routine

- 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)
```

- 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

- 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

$cgen(e_1 + e_2) =$

```
    cgen(e1)
    sw $a0 0($sp)
    addiu $sp $sp -4
    cgen(e2)
    lw $t1 4($sp)
    add $a0 $t1 $a0
    addiu $sp $sp 4
```

$cgen(e_1 + e_2) =$

```
cgen(e1)
sw $a0 0($sp)
addiu $sp $sp -4
cgen(e2)
lw $t1 4($sp)
add $a0 $t1 $a0
addiu $sp $sp 4
```

These two statements store the result of e1 into the stack

```
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
```

These two statements store the result of e1 into the stack

This instruction load the result of e1 into the temporary reg t1

```
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
```

These two statements store the result of e1 into the stack

This instruction load the result of e1 into the temporary reg t1

accu = e1+e1

```
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
```

These two statements store the result of e1 into the stack

This instruction load the result of e1 into the temporary reg t1

accu = e1+e2

pops the stack

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

These two statements store the result of e1 into the stack

This instruction load the result of e1 into the temporary reg t1

accu = e1+e1

pops the stack

Actually the code generator creates a file containing the instructions that will be executed at run time.

```
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"
```

- 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

- New instruction: sub reg_1 reg_2 reg_3
  - Implements $reg_1 \leftarrow reg_2 - reg_3$

```
cgen(e1 - e2) =
      cgen(e1)
      sw $a0 0($sp)
      addiu $sp $sp -4
      cgen(e2)
      lw $t1 4($sp)
      sub $a0 $t1 $a0
      addiu $sp $sp 4
```

- New instruction: sub reg_1 reg_2 reg_3
  - Implements $reg_1 \leftarrow reg_2 - reg_3$

```
cgen(e1 - e2) =
    cgen(e1)
    sw $a0 0($sp)
    addiu $sp $sp -4
    cgen(e2)
    lw $t1 4($sp)
    sub $a0 $t1 $a0
    addiu $sp $sp 4
```

What is the difference between this and the code for e1+e2?

- New instruction: sub reg_1 reg_2 reg_3
  - Implements $reg_1 \leftarrow reg_2 - reg_3$

cgen(e_1 - e_2) =

```
cgen(e1)
sw $a0 0($sp)
addiu $sp $sp -4
cgen(e2)
lw $t1 4($sp)
sub $a0 $t1 $a0
addiu $sp $sp 4
```

This difference is only in this instruction (sub instead of add)

To be able to generate code for **if expressions** we need two more MIPS 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

```
cgen(if e1 = e2 then e3 else e4) =
  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 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

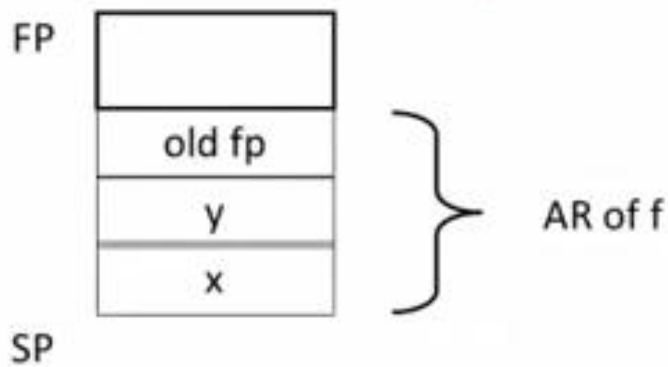
- The stack discipline guarantees that on function exit $sp is the same as it was on function entry
  - No need for a control link

- We need the return address

- A pointer to the current activation is useful
  - This pointer lives in register $fp (frame pointer)

- 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:



- 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

The code that will execute on the Caller Side

```
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

The code that will execute on the Callee Side

- New instruction: jr reg
  - Jump to address in register reg

```
cgen(def f(x1,...,xn) = e) =
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
```

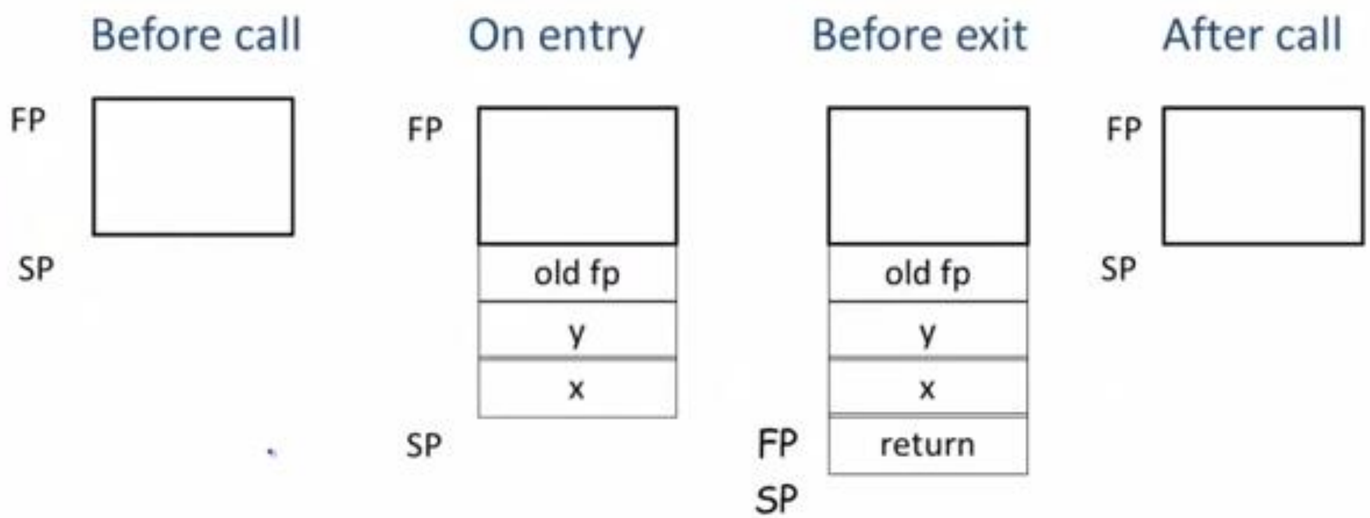
- New instruction: jr reg
  - Jump to address in register reg

cgen(def f(x_1,...,x_n) = e) =

```
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
```

- 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

Two words: the return address and the old frame address


Before call
On entry
Before exit
After call
FP
SP
FP
old fp
y
x
SP
old fp
y
x
FP
return
SP
FP
SP

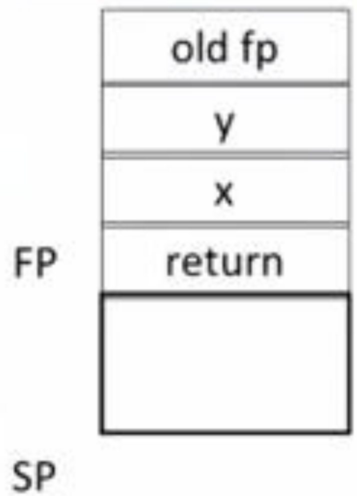
Generating Code for Reference Variables: i.e. generate code that determines their place in the activation record.

- 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

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

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

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



- X is at fp + 4
- Y is at fp + 8

Code Generation Example

def sumto(X) = if x=0 then 0 else x+sumto(x-1)

Sumto_entry:

```
    move    $fp    $sp           % fp points to AR of sumto
    sw      $ra    0($sp)
    addiu   $sp    $sp    -4
    lw      $a0    4($fp)        //load  x
    sw      $a0    0($sp)
    addiu   $sp    $sp   -4
    li      $a0    0             //code for 0
    lw      $t1    4($sp)        //pop x
    addiu   $sp    $sp    4
    beq     $a0    $t1   true1   //goto then part
```

false1:

false1:

```
        lw      $a0     4($fp)      % x
        sw      $a0     0($sp)
        addiu   $sp     $sp  -4
```

% call sumto(x-1) here we insert what happens at the caller side

% store old fp on the new activation record

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

% compute x-1 and store it in the new AR

```
        lw      $a0     4($fp)
        sw      $a0     0($sp)
        addiu   $sp     $sp  -4
        li      $a0     1           //code for 1
        lw      $t1     4($sp)      // x
        sub     $a0     $t1    $a0
        addiu   $sp     0($sp)      //pop stack
```

%store x-1 in the new AR

```
        sw      $a0     0($sp)
```

% jump to sumto(x-1)

```
              jal       Sumto_entry
% add X
              lw        $t1       4($sp)      % x was stored in the stack
                                              because it is part of the
                                              expression x+sumto(x-1)
              add       $a0       $t1   $a0
              addiu     $sp       $sp   4     %pop x
              b         endif
% the true part    0
true1:
              li        $a0       0
% function ends so we return
endif:        lw        $ra       4($sp)
% pop AR
              addiu     $sp       $sp   12
              lw        $fp       0($sp)
% jump back to the caller
              jr        $ra
```

Two Observations:

1. Although the code is correct it is extremely inefficient, notice that we load x form the stack (memory op) then immediately after that we store it back in the stack.
2. Temporary values are stored in the AR; a better alternative might be to store them in temporary registers.

Summary

- The activation record must be designed together with the code generator
- Code generation can be done by recursive traversal of the AST
- The stack model simplifies things.
- 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

An Improvement

- Idea: Keep temporaries in the AR
- The code generator must assign a fixed location in the AR for each temporary (this saves the need to keep pushing an popping them).

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? i.e. how many intermediate values are needed?

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

- $NT(e_1 + e_2) = max(NT(e_1), 1 + NT(e_2))$
- $NT(e_1 - e_2) = max(NT(e_1), 1 + NT(e_2))$
- $NT(\text{if } e_1 = e_2 \text{ then } e_3 \text{ else } e_4) = max(NT(e_1), 1 + NT(e_2), NT(e_3), NT(e_4))$
- $NT(id(e_1,\ldots,e_n) = max(NT(e_1),\ldots,NT(e_n))$
- $NT(int) = 0$
- $NT(id) = 0$

- What is NT(…code for fib…)?

- The answer is 2

def fib(x) = if x = 1 then 0 else

if x = 2 then 1 else

fib(x - 1) + fib(x – 2)

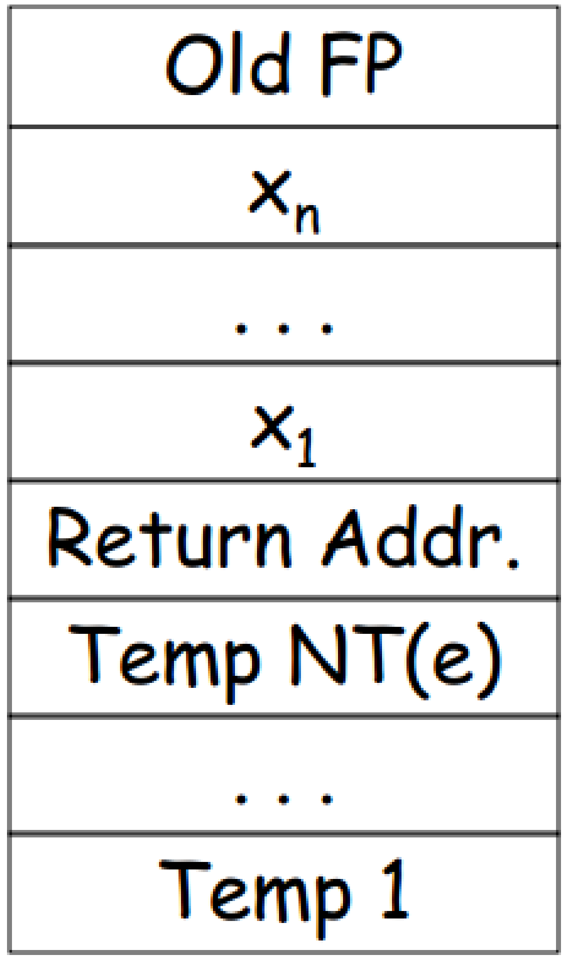
= 2

temp

The Revised AR

- For a function definition $f(x_1,\ldots,x_n) = e$ 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

Code Generation for + (original)

cgen(e_1+ e_2) =

```
cgen(e1)
sw $a0 0($sp)
addiu $sp $sp -4
cgen(e2)
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

- Two sentences shorter and substantially more efficient.
- The temporary area is used like a small, fixed-size stack