# CODE GENERATION

Baishakhi Ray

Stack Machine

- A simple evaluation model

- No variables or registers

- A stack of values for intermediate results

- 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

Example of Stack Machine Operation

- The addition operation on a stack machine

pop   add   push
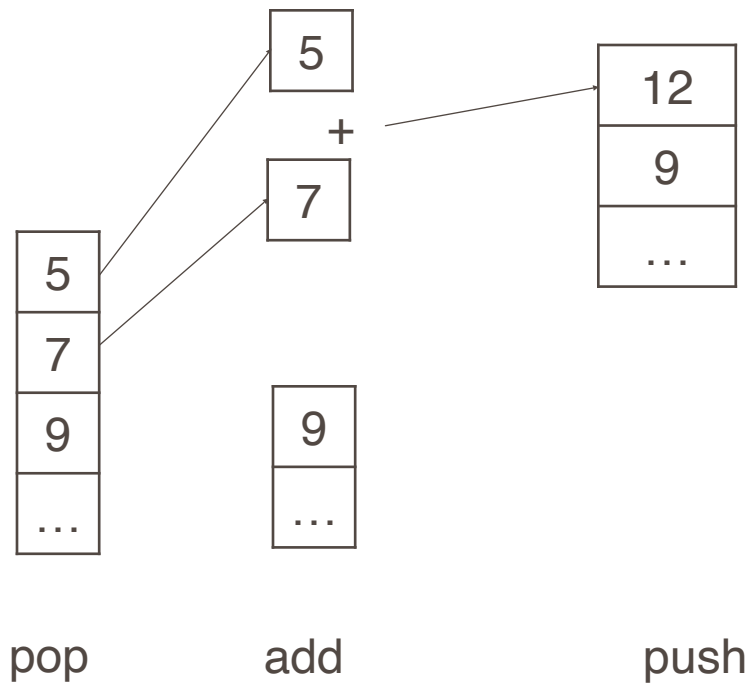
Example of a Stack Machine Program

- 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

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

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 r1, r2"
  - ⇒ 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

  acc ← acc + top_of_stack
  - Only one memory operation!

Stack Machine with Accumulator

- **Invariants**
    - The result of an expression is in the accumulator

    - For $op(e_1,\ldots,e_n)$ push the accumulator on the stack after computing $e_1,\ldots,e_{n-1}$
        - After the operation pops n-1 values

    - Expression evaluation preserves the stack

Stack Machine with Accumulator. Example

- Compute 7 + 5 using an accumulator

1. acc ← 7; push acc

2. acc ← 5

3. acc ← acc + top_of_stack

4. pop

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

Code	ACC	Stack
acc ← 3	3	<init>
push acc	3	3,<init>
acc ← 7	7	3,<init>
push	7	7, 3,<init>
acc ← 5	5	7, 3,<init>
acc ← acc + top_of_stack	12	7, 3,<init>
pop	12	3,<init>
acc ← acc + top_of_stack	15	3,<init>
pop	15	<init>

It is very important evaluation of a subexpression preserves the stack
- Stack before the evaluation of 7 + 5 is 3
- Stack after the evaluation of 7 + 5 is 3
- The first operand is on top of the stack

From Stack Machines to MIPS

- The compiler generates code for a stack machine with accumulator

- Let's run the resulting code on a MIPS like processor.
  - 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

- The address of the next location on the stack is kept in MIPS register $sp (stack pointer)
  - 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)

A Sample of MIPS Instructions

- ## lw reg1 offset(reg2)
  - Load 32-bit word from the value of reg2 (which is a memory address), add a fixed value offset into reg1

- ## add reg1 reg2 reg3
  - reg1 ← reg2 + reg3

- ## sw reg1 offset(reg2)
  - Store 32-bit word in reg1 at address reg2 + offset

- ## addiu reg1 reg2 imm
  - reg1 ← reg2 + imm
  - "u" means overflow is not checked

- ## li reg imm
  - reg ← imm

MIPS Assembly, Example

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

Steps	MIPS Instruction
`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`

- Let's generalize this to a simple language

A Small Language

- A language with integers and integer operations

```
P → D; P | D
D → def id(ARGS) = E;
ARGS → id, ARGS | id
E → int | id | if E₁ = E₂ then E₃ else E₄
      | E₁ + E₂ | E₁ − E₂ | id(E₁,…,Eₙ)
```

- 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:

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

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

- 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

Code Generation for Add

```
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
```

Code Generation for Add

```
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
```

```
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 Add. Wrong!

- Optimization: Put the result of e_1 directly in $t1?

  ```
  cgen(e1 + e2) =

      cgen(e1)

      move $t1 $a0

      cgen(e2)

      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 $e_1 + e_2$ 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

Code Generation for Sub and Constants

- New instruction: `sub reg1 reg2 reg3`
  - Implements `reg1 ← reg2 - reg3`

```
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
```

From what expression the following assembly code is generated?

```
li $a0 5
sw $a0 0($sp)
addiu $sp $sp -4
li $a0 4
sw $a0 0($sp)
addiu $sp $sp -4
li $a0 3
lw $t1 4($sp)
sub $a0 $t1 $a0
addiu $sp $sp 4
lw $t1 4($sp)
add $a0 $t1 $a0
addiu $sp $sp 4
```

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.)

```
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:
```

The Activation Record

- 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 Activation Record (Cont.)

- The stack discipline guarantees that on function exit $sp is the same as it was on function entry

- We need the return address

- A pointer to the current activation is useful
    - 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 is:
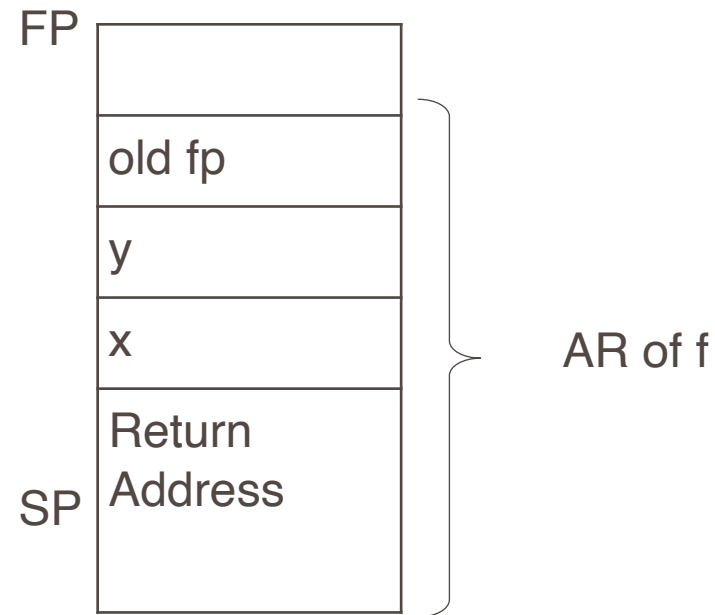
Code Generation for Function Call

- 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(e1,…,en)) =
    sw $fp 0($sp)
    addiu $sp $sp -4
    cgen(e_n)
    sw $a0 0($sp)
    addiu $sp $sp -4
    …
    cgen(e_1)
    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

- 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: jr reg
  - Jump to address in register reg

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

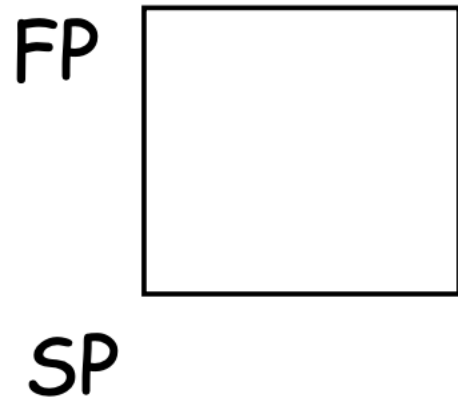Note: The frame pointer points to the top, not bottom of the frame

The callee pops the return address, the actual arguments and the saved value of the frame pointer.
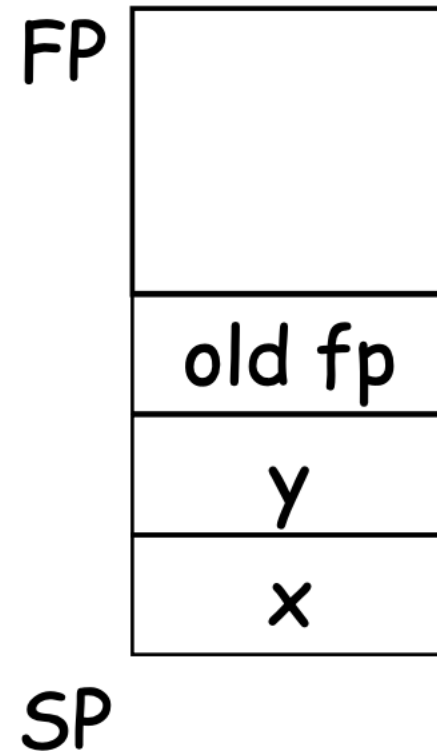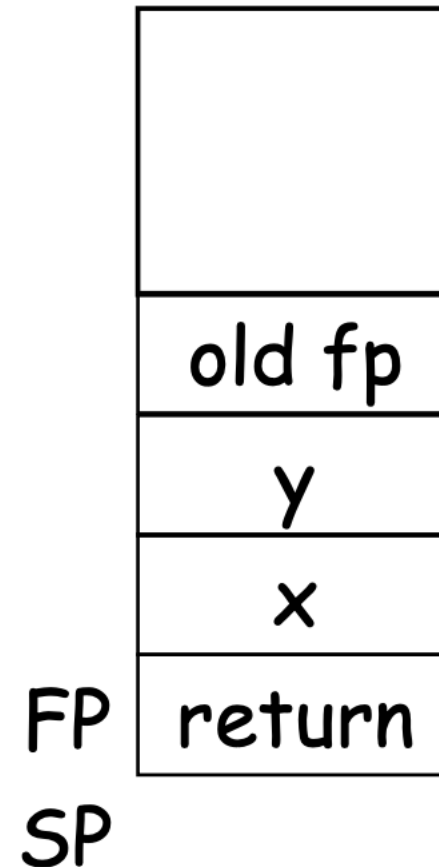
z = 4*n + 8

Calling Sequence: Example for f(x,y)

Before call

FP

SP

On entry

FP

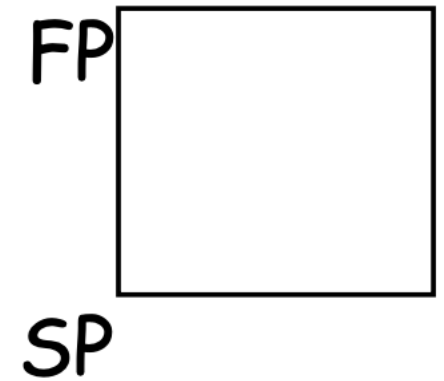| old fp |
| y |
| x |

SP

Before exit

| old fp |
| y |
| x |
| return |

FP

SP

After call

FP

SP

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 ith (i = 1,…,n) formal parameter of the function for which code is being generated

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

Code Generation for Variables (Cont.)

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

old fp
y
x
return

FP → return

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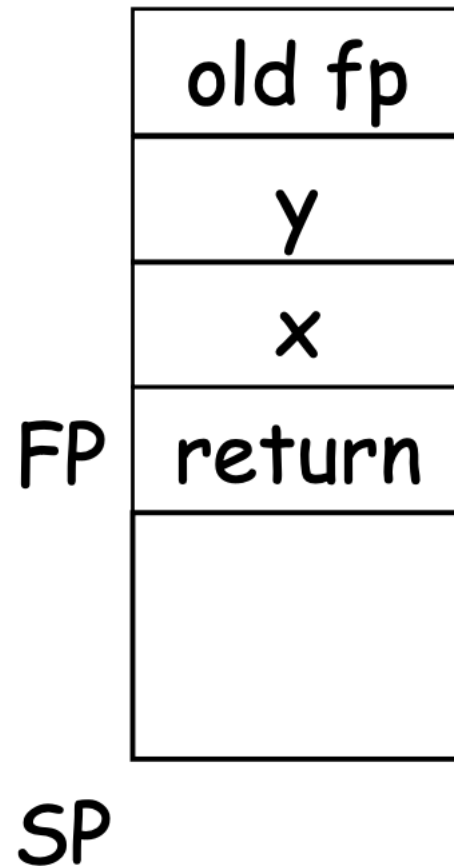
- X is at fp + 4
- Y is at fp + 8

Summary

- The activation record must be designed together with the code generator.

- Code generation can be done by recursive traversal of the AST.

- 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

Example: def cumsum(x) = if x = 0 then 0 else x + cumsum(x-1)

```
cumsumEntry:
  move $fp $sp
  sw $ra 0($sp)
  addiu $sp $sp -4
  lw $a0 4($fp)
  sw $a0 0($sp)
  addiu $sp $sp -4
  li $a0 0
  lw $t1 4($sp)
  addiu $sp $sp 4
  beq $a0 $t1 true_branch
```

```
false_branch:
  lw $a0 4($fp)
  sw $a0 0($sp)
  addiu $sp $sp -4
  sw $fp 0($sp)
  addiu $sp $sp -4
  lw $a0 4($fp)
  sw $a0 4($sp)
  addiu $sp $sp -4
  li $a0 1
  lw $t1 4($sp)
  sub $a0 $t1 $a0
  addiu $sp $sp 4
```

```
  sw $a0 0($sp)
  addiu $sp $sp -4
  jal cumsumEntry
  lw $t1 4($fp)
  add $a0 $t1 $a0
  addiu $sp $sp 4
  b endif1

true_branch:
  li $a0 0

endif1:
  lw $ra 4($sp)
  addiu $sp $sp 12
  lw $fp 0($sp)
  jr $ra
```
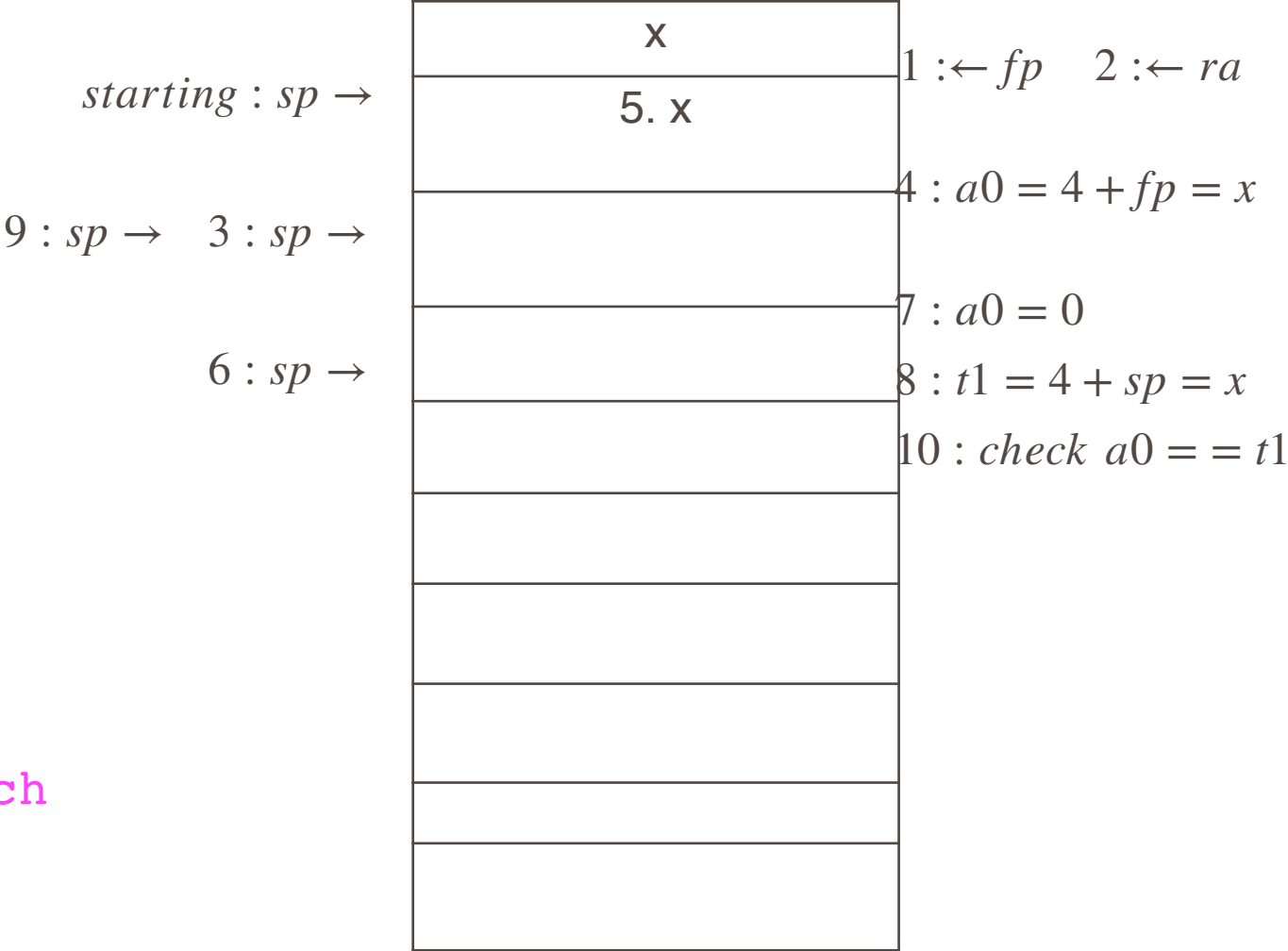
```
cumsumEntry:
  1. move $fp $sp
  2. sw $ra 0($sp)
  3. addiu $sp $sp -4
  4. lw $a0 4($fp)
  5. sw $a0 0($sp)
  6. addiu $sp $sp -4
  7. li $a0 0
  8. lw $t1 4($sp)
  9. addiu $sp $sp 4
 10. beq $a0 $t1 true_branch
```

$starting : sp \rightarrow$

$9 : sp \rightarrow$ $3 : sp \rightarrow$

$6 : sp \rightarrow$

x
5. x

$1 : \leftarrow fp$ $2 : \leftarrow ra$

$4 : a0 = 4 + fp = x$

$7 : a0 = 0$

$8 : t1 = 4 + sp = x$

$10 : check\ a0 == t1$

`

```
false_branch:
  11. lw $a0 4($fp)
  12. sw $a0 0($sp)
  13. addiu $sp $sp -4
  14. sw $fp 0($sp)
  15. addiu $sp $sp -4
  16. lw $a0 4($fp)
  17. sw $a0 4($sp)
  18. addiu $sp $sp -4
  19. li $a0 1
  20. lw $t1 4($sp)
  21. sub $a0 $t1 $a0
  22. addiu $sp $sp 4
```

$starting : sp \rightarrow$

$9 : sp \rightarrow$

$13 : sp \rightarrow$

$22 : sp \rightarrow 15 : sp \rightarrow$

$18 : sp \rightarrow$

x
12. x
14. fp
17. x

$1 :\leftarrow fp \quad 2 :\leftarrow ra$

$11 : a0 = 4 + fp = x$

$16 : a0 = 4 + fp = x$

$19 : a0 = 1$

$20 : t1 = 4 + sp = x$

$21 : a0 = t1 - a0 = x - 1$

```
23. sw $a0 0($sp)
24. addiu $sp $sp -4
25. jal cumsumEntry
26. lw $t1 4($fp)
27. add $a0 $t1 $a0
28. addiu $sp $sp 4
29. b endif1

true_branch:
30.  li $a0 0

endif1:
31.  lw $ra 4($sp)
32.   addiu $sp $sp 12
33.  lw $fp 0($sp)
34.  jr $ra
```

$starting : sp \rightarrow$

$27 : sp \rightarrow$ $22 : sp \rightarrow$

$24 : sp \rightarrow$

x
12. x
14. fp
23. x-1

$1 : \leftarrow fp \quad 2 : \leftarrow ra$

$11 : a0 = 4 + fp = x$

$16 : a0 = 4 + fp = x$

$19 : a0 = 1$

$20 : t1 = 4 + sp = x$

$21 : a0 = t1 - a0 = x - 1$

$25 : jal\ cumsumEntry\ (result\ in\ a0)$

$26 : t1 = 4 + fp = x$

$27 : a0 = t1 + a0 = x + cumsum(x - 1)$

$30 : a0 = 0$