The Activation Record (AR)

[Notes adapted from R. Bodik]

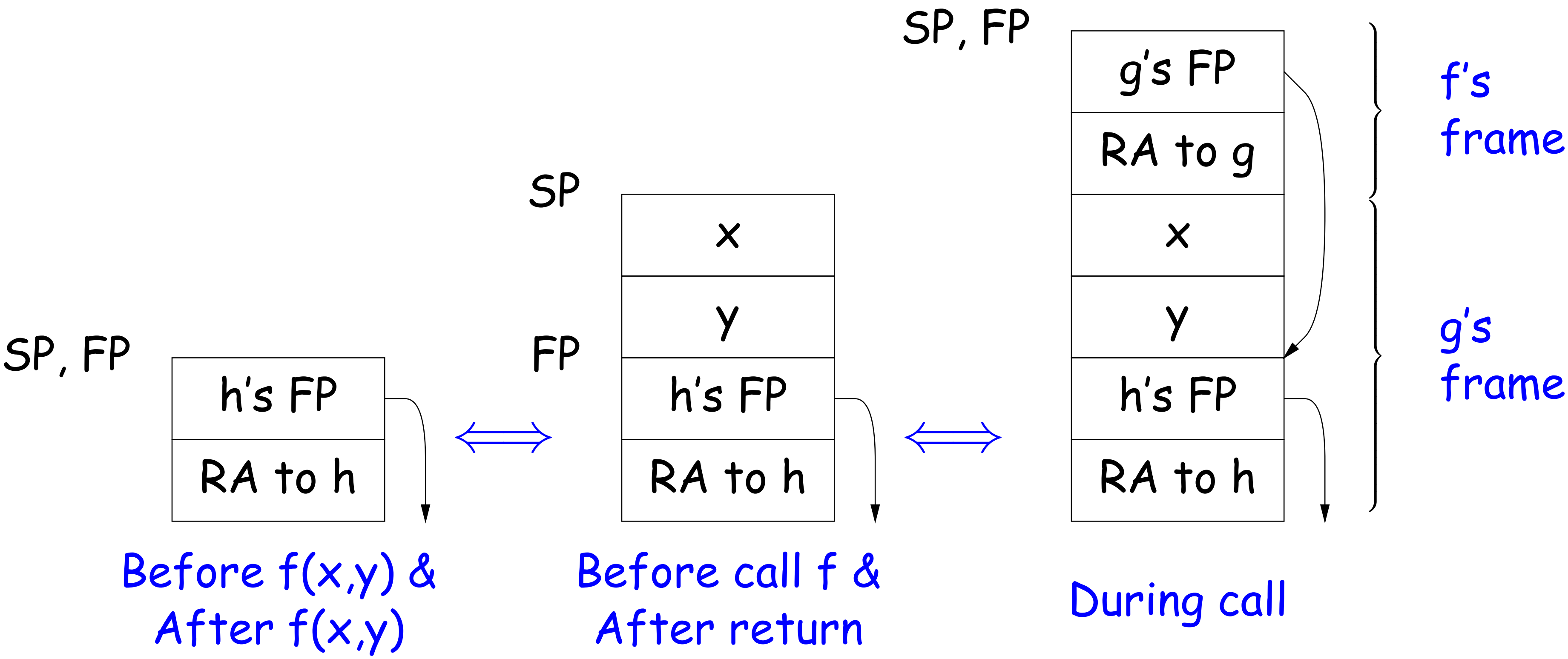
- Code for function calls and function definitions depends on the layout of the activation record
- Very simple AR suffices for this language:
  - The result is always in the accumulator; no need to store the result in the AR.
  - The activation record of the caller holds actual parameters just below callee's AR.
    - For f(x1,...,xn), push xn,...,x1 on the stack
    - These are the only variables in this language
  - AR must also save return address.

The Frame Pointer

- The stack discipline guarantees that on function exit \$sp is the same as it was on function entry.
- No need to save \$sp
- But it's handy to have a pointer to start of the current AR.
  - Lives in register \$fp (frame pointer)
  - Useful for giving addresses of variables and parameters fixed offsets while manipulating \$sp.

Layout of Frame

- For our simple language, if h calls g, which calls f(x,y), then
  - g's AR will contain x and y,
  - f's AR will contain return address (back to g) and g's frame pointer.



Basic Tools for Calling

- The *calling sequence* is the instructions to set up a function invocation and restore state on return.
- The *function prologue* is the code in the function definition that sets up the AR.
- The *function epilogue* is the code in the function that returns and deletes the activation record.
- Most machines have special instructions for calls:
  - On MIPS, jal *LABEL*, jumps to *LABEL* and saves address of next instruction after the jal in $ra.
  - On ia32, the return address is stored on the stack by the *call LABEL* instruction
- And returns:
  - On MIPS, jr *REG* jumps to address in *REG*.
  - On ia32, ret pops return address from stack and goes there.

Code Generation Strategy for Call

```
cgen (f(e_1, ..., e_n)):
      cgen (e_n)                  # Evaluate and push
      push $acc                   # parameters in reverse
      ...
      cgen (e_1)
      push $acc
      jal f                       # Jump to f and save return
      addiu $sp, $sp, 4*n         # Pop parameters from stack
```

Code Generation for Function Prologue and Epilogue

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

```
push $ra            # Save return address
push $fp            # Save frame pointer
move $fp, $sp       # Set new frame pointer
cgen (e)
lw $ra, 8($fp)      # Restore return address
lw $fp, 4($fp)      # Restore frame pointer
addiu $sp, $fp, 8   # Restore the stack pointer
jr $ra              # And return to caller
```

IA32 Version of Function Prologue and Epilogue

The last slide not a typical MIPS sequence: biased to look like the ia32:

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

```
                                # (Call instruction has already
                                # pushed return address.)
    pushl %ebp                  # Save frame pointer
    movl %esp,%ebp              # Set new frame pointer
    cgen (e)
    leave                       # Pop frame pointer from stack.
    ret                         # Pop return address and return
```

Code Generation for Local Variables

- Local variables are stored on the stack (thus not at fixed location).
- One possibility: access relative to the stack pointer.
  - Problem: stack pointer changes in strategy we've been using for cgen.
- Solution: use frame pointer, which is constant over execution of function.
- For simple language, use fact that parameter i is at location $fp $+ 4(i+2)$:
  - cgen (x_i) = lw $a0, K($fp), where $K = 4(i+2)$.
- If we had local variables other than parameters, they would be at negative offsets from $fp.

Passing Static Links (I)

- When using static links, the link can be treated as a parameter.
- In the Pyth runtime, for example, a function value consists of a code address followed by a static link.
- So, if we have a function-valued variable at, say, offset -8 from frame pointer, can call it with

```
lw $t1, -8($fp)          # Fetch address of code
lw $t2, -4($fp)          # Fetch static link
push $t2                 # And pass as first parameter
jalr $t1                 # Jump to address in $t1.
```

Accessing Non-Local Variables

- In program on left, how does f3 access x1?
- f3 will have been passed a static link as its first parameter.
- The static link passed to f3 will be f2's frame pointer

```
def f1 (x1):
   def f2 (x2):
       def f3 (x3):
           ... x1 ...
       ...
       f3 (12)
   ...
   f2 (9)
```

```
lw $t, 8($fp)   # Fetch FP for f2
lw $t, 8($t)    # Fetch FP for f1
lw $a0, 12($t)  # Fetch x1
```

- In general, for a function at nesting level n to access a variable at nesting level $m < n$, perform $n - m$ loads of static links.

Passing Static Links (II)

- In previous example, how do we call f2 from f3? f3 from f2? f2 from f3?

```
def f1 (x1):
    def f2 (x2):
        def f3 (x3):
            ... f2 (9) ...
        ...
        f3 (12)
        f2 (10) # (recursively)
    ...
```

To get static link for f2(9):

```
lw $t 8($fp)   # Fetch FP for f2
lw $t 8($t)    # Fetch FP for f1
push $t        # Push static link
```

To get static link for f3 (12):

```
push $fp       # f2's FP is static link
```

To get static link for f2(10):

```
lw $t 8($fp)
push $t
```

- Could create a function value, and call as in previous slide.
- Can do better. Functions and their nesting levels are known.
- If in a function at nesting level n, calling another at known nesting level $m \leq n+1$, get correct static link in $t with:
  - Set $t to $fp.
  - Perform 'lw $t, 8($t)' $n-m+1$ times.