

# Register Allocation, i

Overview & spilling

# L1

```
p ::= ((i ...) (label i ...) ...)
i ::= (x <- s)
      | (x <- (mem s n4))
      | ((mem x n4) <- s)
      | (x aop= s)
      | (x sop= sx)
      | (x sop= num)
      | (cx <- s cmp s)
      | label
      | (goto label)
      | (cjump s cmp s label label)
      | (call s)
      | (tail-call s)
      | (return)
      | (eax <- (print s))
      | (eax <- (allocate s s))
      | (eax <- (array-error s s))
aop= ::= += | -= | *= | &=
sop ::= <<= | >>=
cmp ::= < | <= | =
s ::= x | num | label
x, y ::= cx | esi | edi | ebp | esp
cx ::= eax | ecx | edx | ebx
sx ::= ecx
```

# L2

```
p ::= ((i ...) (label i ...) ...)
i ::= (x <- s)
      | (x <- (mem s n4))
      | ((mem x n4) <- s)
      | (x aop= s)
      | (x sop= sx)
      | (x sop= num)
      | (cx <- s cmp s)
      | label
      | (goto label)
      | (cjump s cmp s label label)
      | (call s)
      | (tail-call s)
      | (return)
      | (eax <- (print s))
      | (eax <- (allocate s s))
      | (eax <- (array-error s s))
aop= ::= += | -= | *= | &=
sop ::= <<= | >>=
cmp ::= < | <= | =
s ::= x | num | label
x, y ::= any-variable-at-all | reg
cx ::= any-variable-at-all | reg
sx ::= any-variable-at-all | reg
reg ::= eax | ecx | edx | ebx | esi | edi | ebp | esp
```

## L2 semantics: variables

L2 behaves just like L1, except that non-reg variables are function local, e.g.,

```
(define (f x)      ⇒  ((; :main
  (+ (g x) 1))    (eax <- 10)
                    (call :f))
(define (g x)      (:f (temp <- 1)
  (+ x 2))         (call :g)
                    (eax += temp)
                    (return))
(f 10)            (:g (temp <- 2)
                    (eax += temp)
                    (return)))
```

The assignment to `temp` in `g` does not break `f`, but if `temp` were a register, it would.

## L2 semantics: esp & ebp

L2 programs must use neither **esp** nor **ebp**. They are in L2 to facilitate register allocation only, *not* for the L3 → L2 compiler's use.

# From L2 to L1

Register allocation, in three parts; for each function body we do:

- **Liveness analysis**  $\Rightarrow$  interference graph (nodes are variables; edges indicate “cannot be in the same register”)
- **Graph coloring**  $\Rightarrow$  register assignments
- **Spilling**: coping with too few registers
- Bonus part, **coalescing** eliminating redundant `(x <- y)` instructions

# Example Function

```
int f(int x) = 2x2 + 3x + 4
```

```
:f
```

```
(x2 <- eax)
```

```
(x2 *= x2)
```

```
(2x2 <- x2)
```

```
(2x2 *= 2)
```

```
(3x <- eax)
```

```
(3x *= 3)
```

```
(eax <- 2x2)
```

```
(eax += 3x)
```

```
(eax += 4)
```

```
(return)
```

# Example Function: live ranges

```
int f(int x) = 2x2 + 3x + 4
```

2x2 3x x2

```
:f
```

```
(x2 <- eax)
```

```
(x2 *= x2)
```

```
(2x2 <- x2)
```

```
(2x2 *= 2)
```

```
(3x <- eax)
```

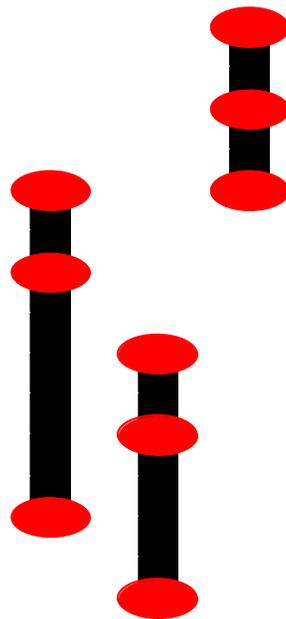
```
(3x *= 3)
```

```
(eax <- 2x2)
```

```
(eax += 3x)
```

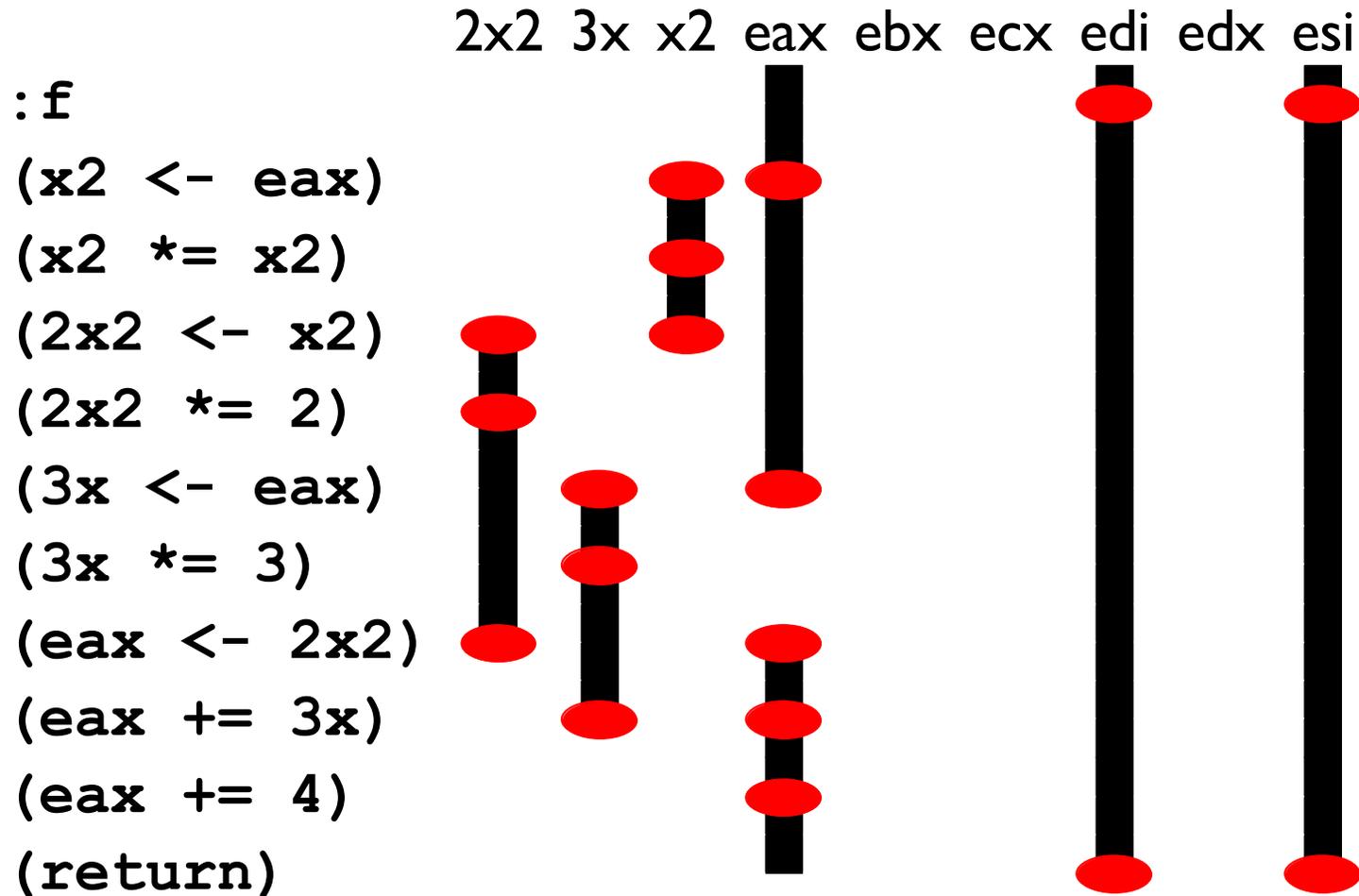
```
(eax += 4)
```

```
(return)
```



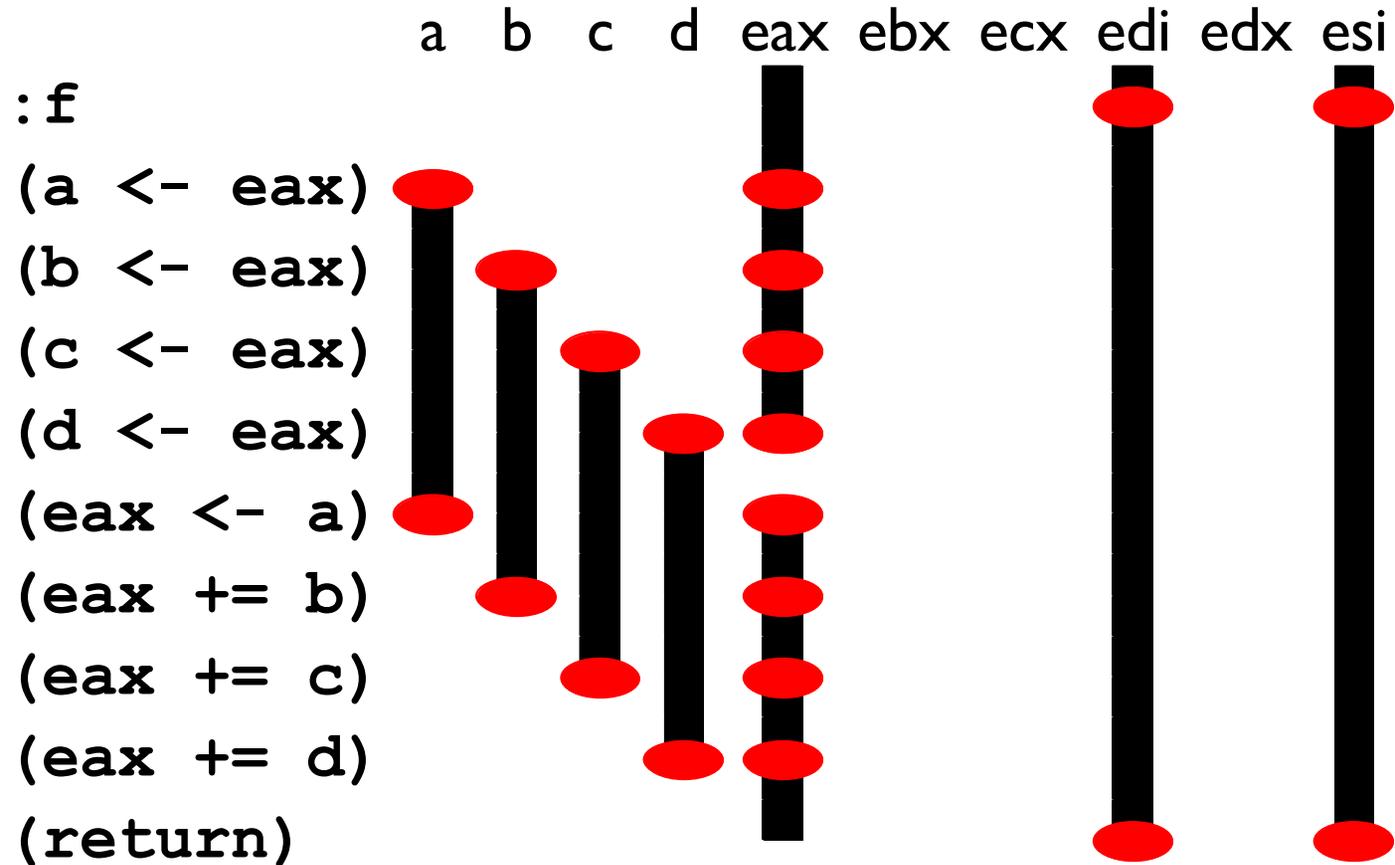
# Example Function: live ranges

```
int f(int x) = 2x2 + 3x + 4
```



## Example Function 2

`int f(int x) = 4x` (in a bad compiler)



No way to get all of **a**, **b**, **c**, and **d** into their own registers; so we need to *spill* one of them.

# Spilling

**Spilling** is a program rewrite to make it easier to allocate registers

- Pick a variable and a location on the stack for it
- Replace all writes to the variable with writes to the stack
- Replace all reads from the variable with reads from the stack

Sometimes that means introducing new temporaries

# Spilling Example

Say we want to spill **a** to the location **(mem ebp -4)**.

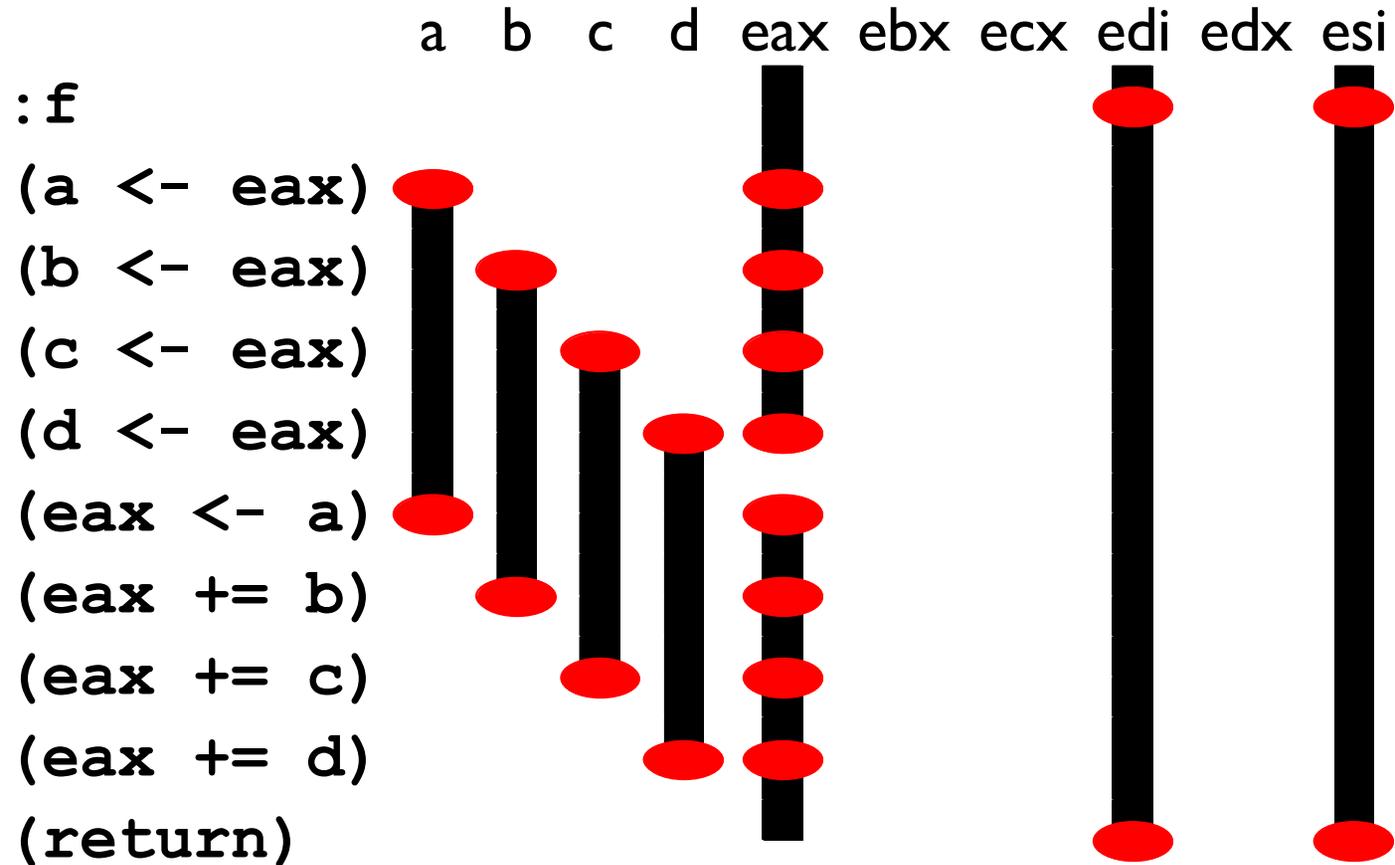
Two easy cases:

**(a <- 1)**  $\Rightarrow$  **((mem ebp -4) <- 1)**

**(x <- a)**  $\Rightarrow$  **(x <- (mem ebp -4))**

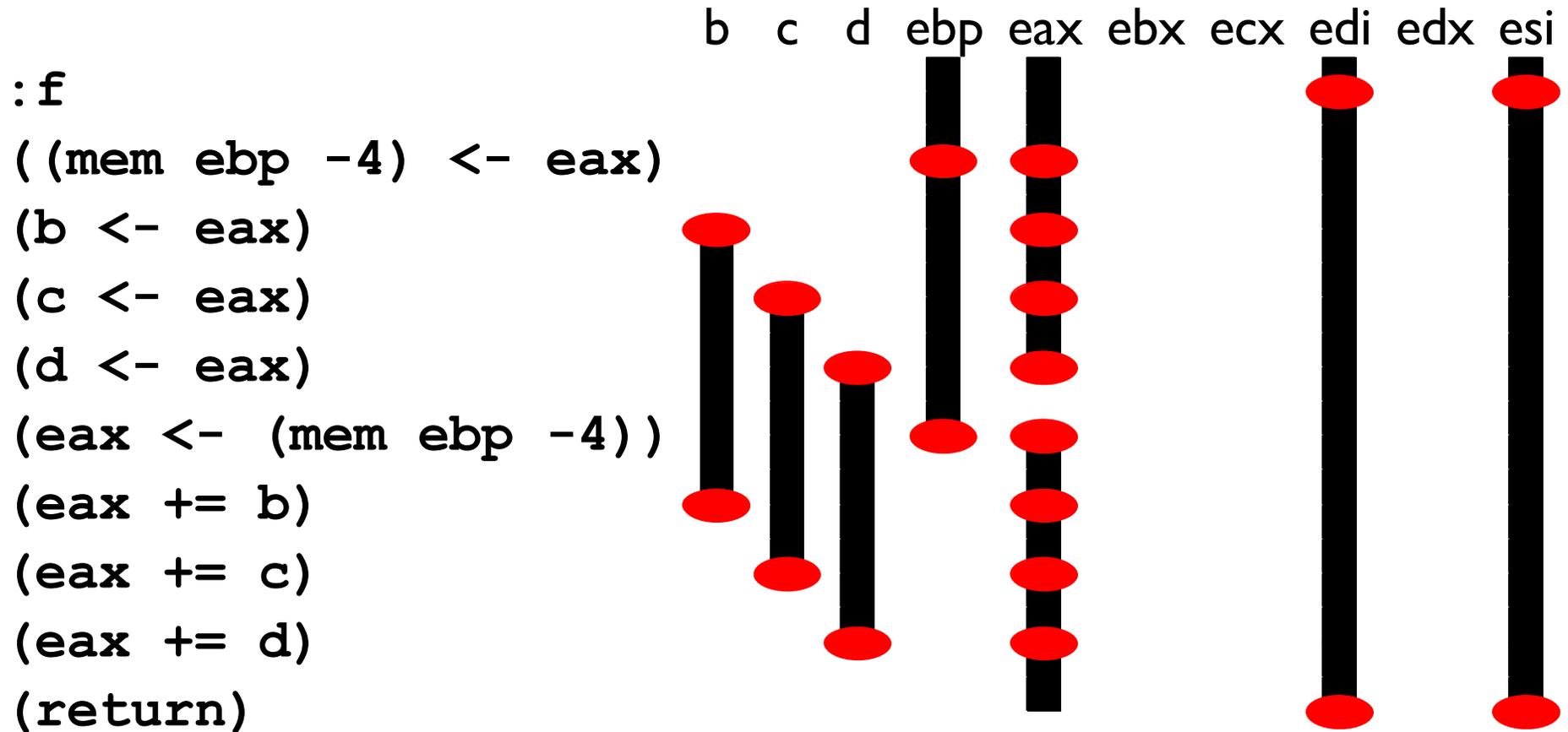
## Example Function 2, need to spill

`int f(int x) = 4x` (in a bad compiler)



## Example Function 2, spilling a

`int f(int x) = 4x` (in a bad compiler)



# Spilling Example

A trickier case:

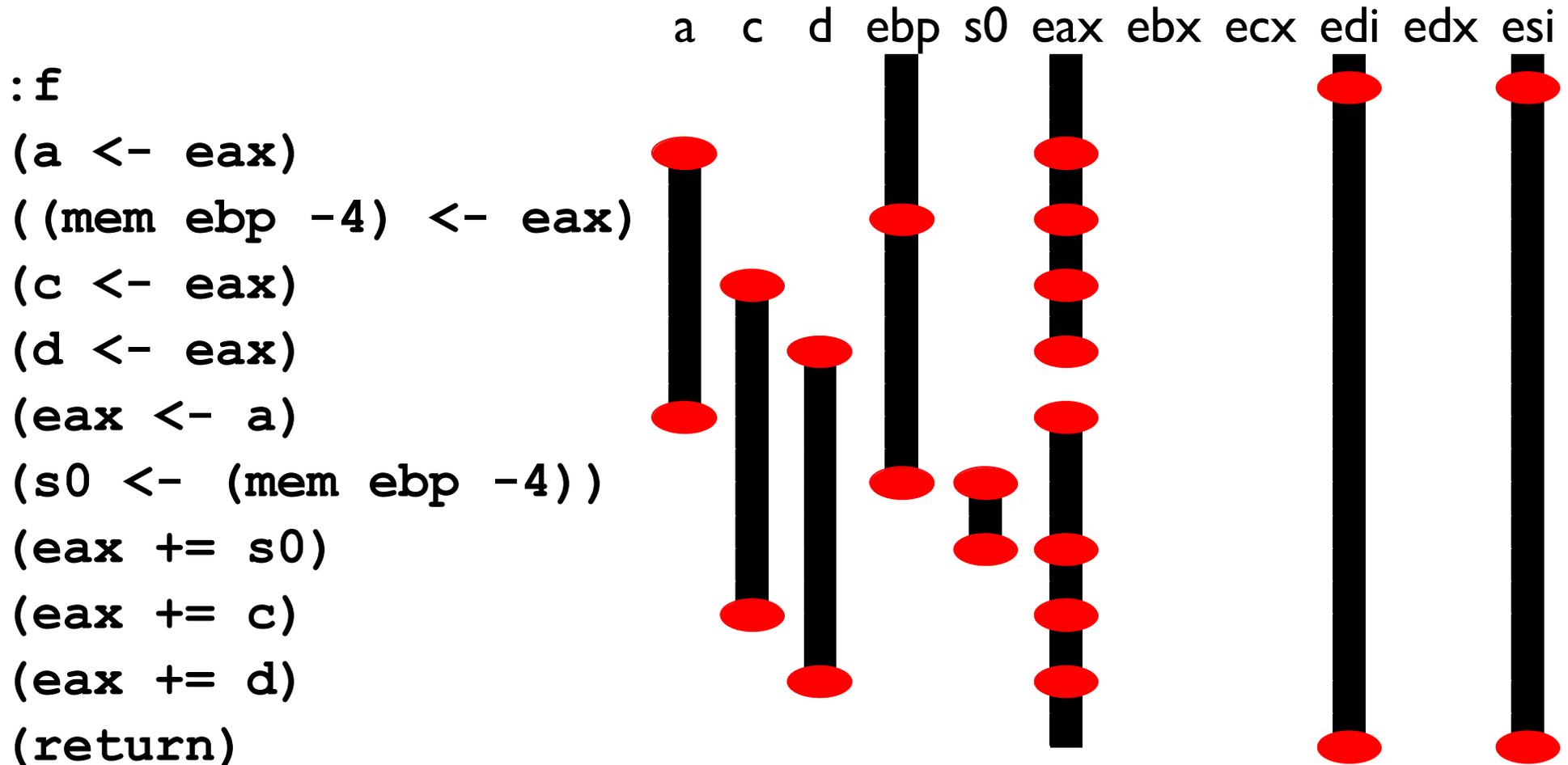
$$\begin{aligned} (a \ *= \ a) \Rightarrow & (a_{\text{new}} \leftarrow (\text{mem } \text{ebp} \ -4)) \\ & (a_{\text{new}} \ *= \ a_{\text{new}}) \\ & ((\text{mem } \text{ebp} \ -4) \leftarrow a_{\text{new}}) \end{aligned}$$

In general, make up a new temporary for each instruction that uses the variable to be spilled

This makes for very short live ranges.

## Example Function 2, spilling b

`int f(int x) = 4x` (in a bad compiler)



## Example Function 2, spilling b

Even though we still have four temporaries, we can still allocate them to our three unused registers because the live ranges of `s0` and `a` don't overlap and so they can go into the same register.

# Your job

Implement:

```
spill : (i ...) ;; original function
      var      ;; to spill
      offset   ;; multiple of 4
      var      ;; prefix for temporaries
-> (i ...) ;; spilled version
```