# Register Allocation, i

Overview & spilling

# L1

```
      p ::= ((i ...) (label i ...) ...)
      i ::= (x <- s)
          | (x <- (mem x n4))
          | ((mem x n4) <- s)
          | (x aop= t)
          | (x sop= sx)
          | (x sop= num)
          | (cx <- t cmp t)
          | label
          | (goto label)
          | (cjump t cmp t label label)
          | (call u)
          | (tail-call u)
          | (return)
          | (eax <- (print t))
          | (eax <- (allocate t t))
          | (eax <- (array-error t t))
   aop= ::= += | -= | *= | &=
    sop ::= <<= | >>=
    cmp ::= < | <= | =
      s ::= x | num | label
      t ::= x | num
      u ::= x | 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 x n4))
        | ((mem x n4) <- s)
        | (x aop= t)
        | (x sop= sx)
        | (x sop= num)
        | (cx <- t cmp t)
        | label
        | (goto label)
        | (cjump t cmp t label label)
        | (call u)
        | (tail-call u)
        | (return)
        | (eax <- (print t))
        | (eax <- (allocate t t))
        | (eax <- (array-error t t))
 aop= ::= += | -= | *= | &=
  sop ::= <<= | >>=
  cmp ::= < | <= | =
    s ::= x | num | label
    t ::= x | num
    u ::= x | label
 x, y ::= cx | esi | edi | ebp | esp
   cx ::= eax | ecx | edx | ebx | var
   sx ::= ecx | var
  var ::= variable matching regexp ^[a-zA-Z_][a-zA-Z_0-9-]*$,
          except registers and keywords (e.g., print, call, cjump, ...)
```

# 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)
(f 10)                       (return))
                         (: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** ⇒ interference graph (nodes are variables; edges indicate “cannot be in the same register”)
- **Graph coloring** ⇒ register assignments
- **Spilling:** coping with too few registers
- Bonus part, **coalescing** eliminating redundant `(x <- y)` instructions

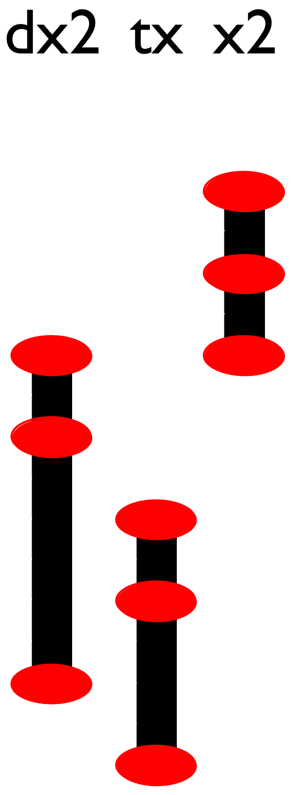
## Example Function

```
int f(int x) = 2x² + 3x + 4

:f
(x2 <- eax)
(x2 *= x2)
(dx2 <- x2)
(dx2 *= 2)
(tx <- eax)
(tx *= 3)
(eax <- dx2)
(eax += tx)
(eax += 4)
(return)
```

# Example Function: live ranges

**`int f(int x)`** $= 2x^2 + 3x + 4$

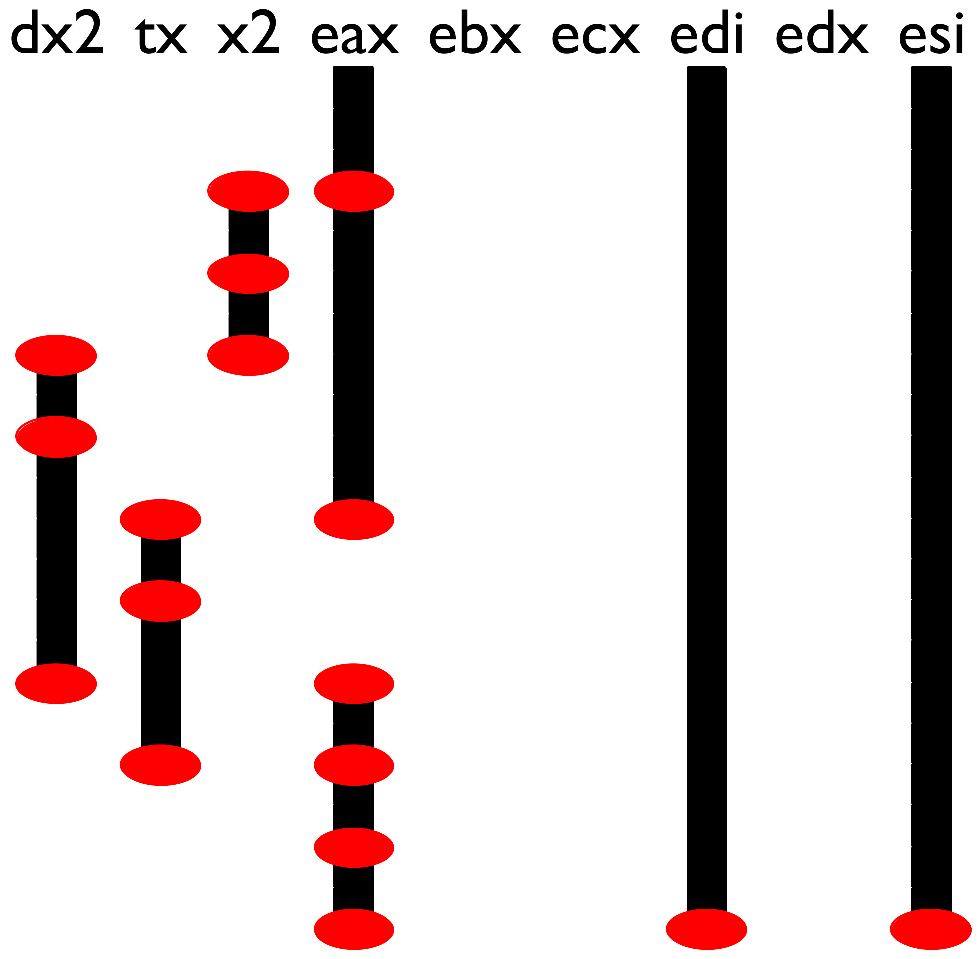


```
:f
(x2 <- eax)
(x2 *= x2)
(dx2 <- x2)
(dx2 *= 2)
(tx <- eax)
(tx *= 3)
(eax <- dx2)
(eax += tx)
(eax += 4)
(return)
```

# Example Function: live ranges

**`int f(int x) = 2x`**$^2$ **`+ 3x + 4`**

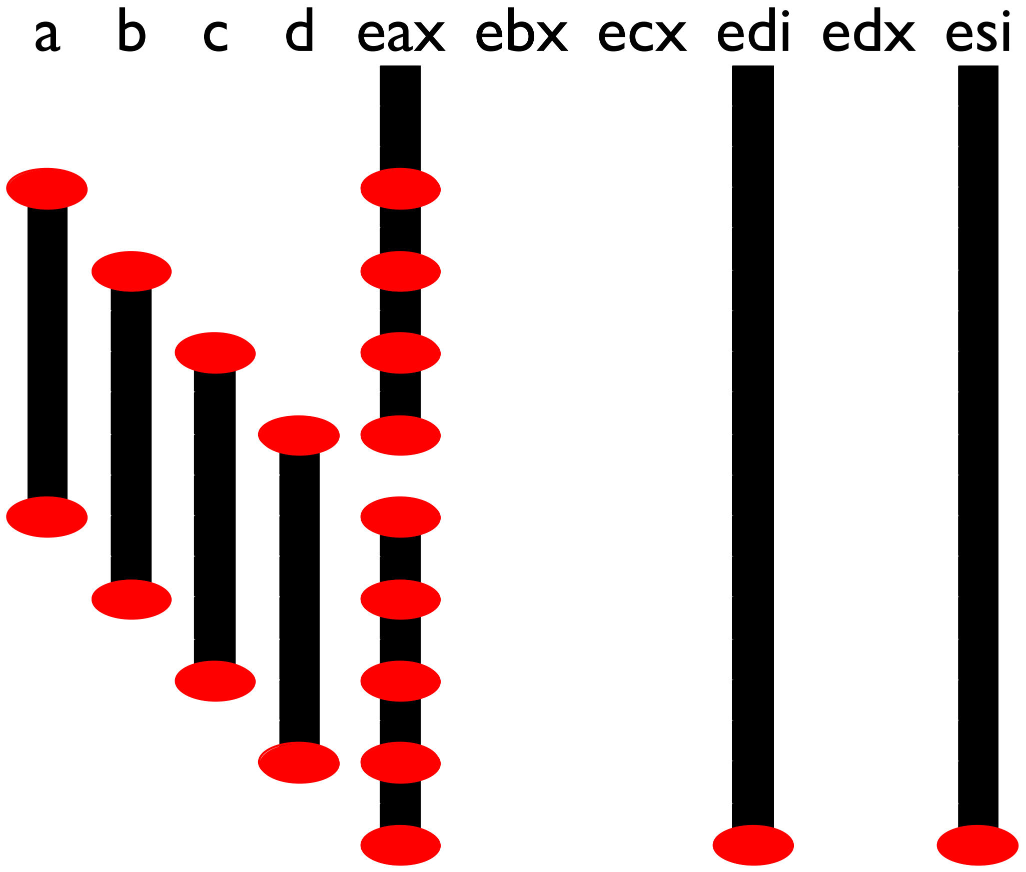
```
:f
(x2 <- eax)
(x2 *= x2)
(dx2 <- x2)
(dx2 *= 2)
(tx <- eax)
(tx *= 3)
(eax <- dx2)
(eax += tx)
(eax += 4)
(return)
```



# Example Function 2

**`int f(int x) = x+x+x+x`** (in a stupid compiler)

```
:f
(a <- eax)
(b <- eax)
(c <- eax)
(d <- eax)
(eax <- a)
(eax += b)
(eax += c)
(eax += d)
(return)
```



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:

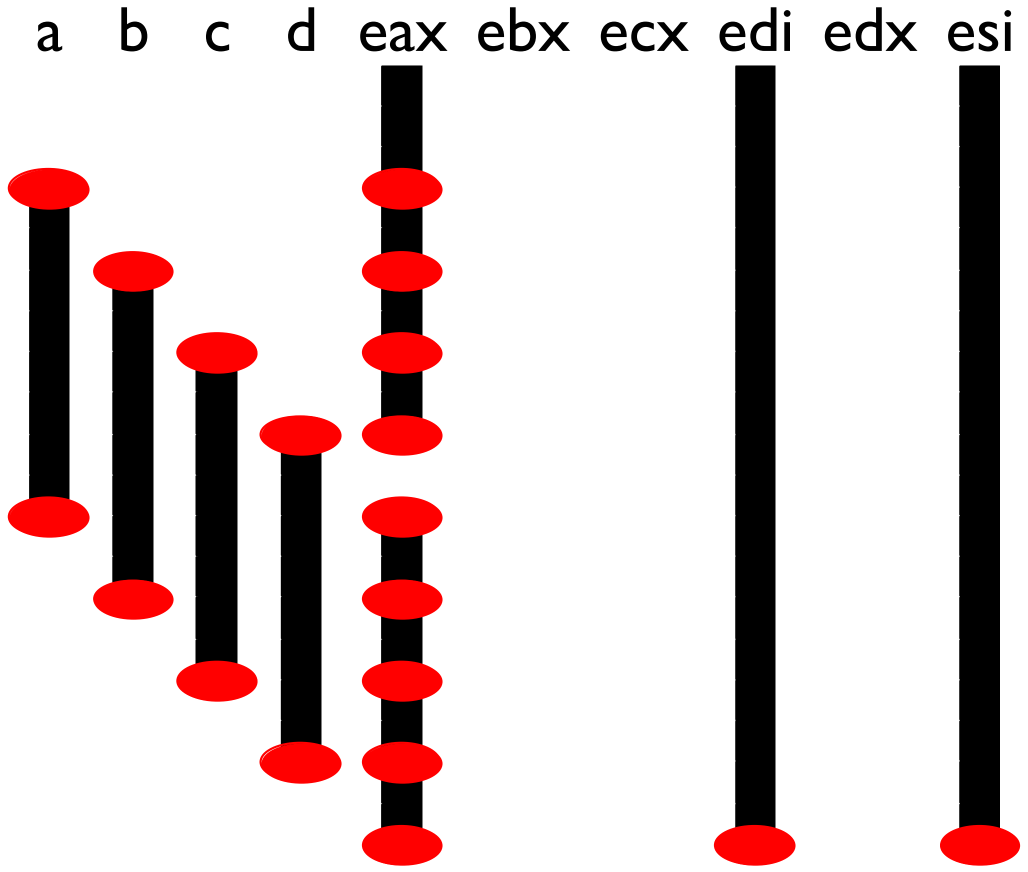
```
(a <- 1)  ⇒  ((mem ebp -4) <- 1)

(x <- a)  ⇒  (x <- (mem ebp -4))
```

# Example Function 2, need to spill

`int f(int x) = x+x+x+x` (in a stupid compiler)

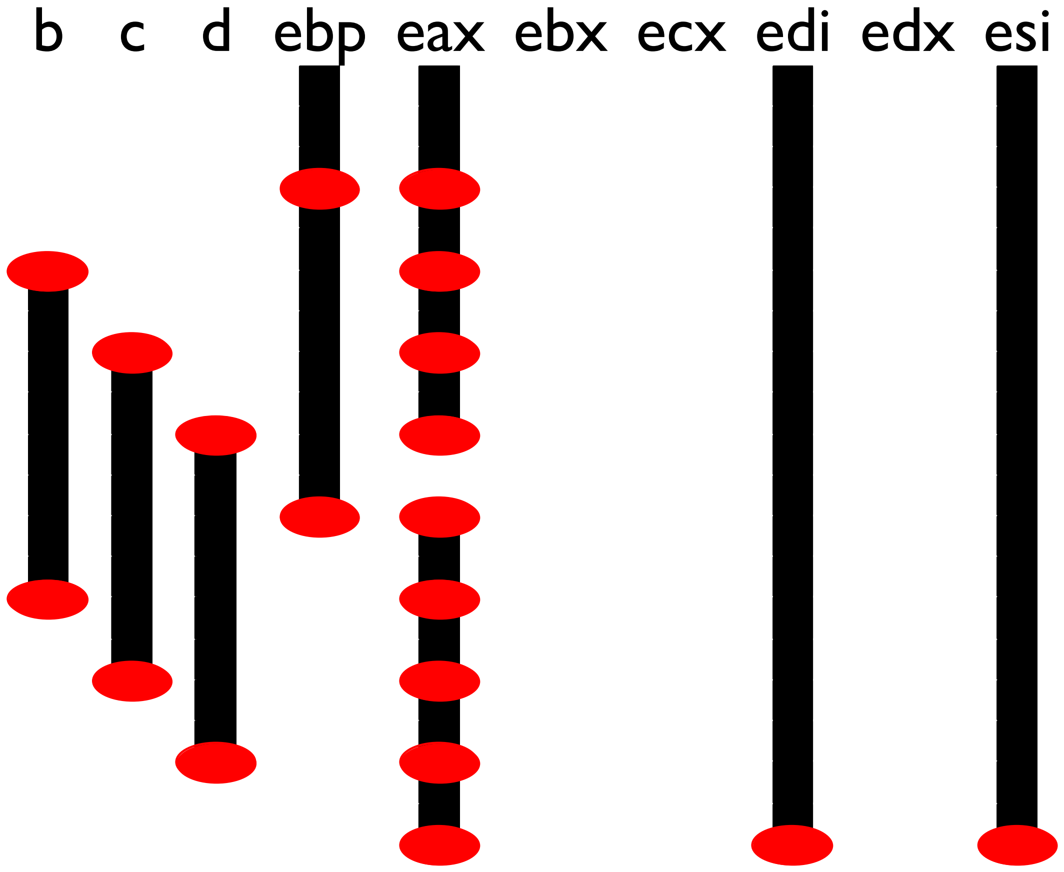
```
:f
(a <- eax)
(b <- eax)
(c <- eax)
(d <- eax)
(eax <- a)
(eax += b)
(eax += c)
(eax += d)
(return)
```



# Example Function 2, spilling a

**`int f(int x) = x+x+x+x`** (in a stupid compiler)

```
:f
((mem ebp -4) <- eax)
(b <- eax)
(c <- eax)
(d <- eax)
(eax <- (mem ebp -4))
(eax += b)
(eax += c)
(eax += d)
(return)
```



## Spilling Example

A trickier case:

$$\texttt{(a *= a)} \Rightarrow \begin{array}{l}\texttt{(a}_{\texttt{new}}\texttt{ <- (mem ebp -4))}\\ \texttt{(a}_{\texttt{new}}\texttt{ *= a}_{\texttt{new}}\texttt{)}\\ \texttt{((mem ebp -4) <- a}_{\texttt{new}}\texttt{)}\end{array}$$

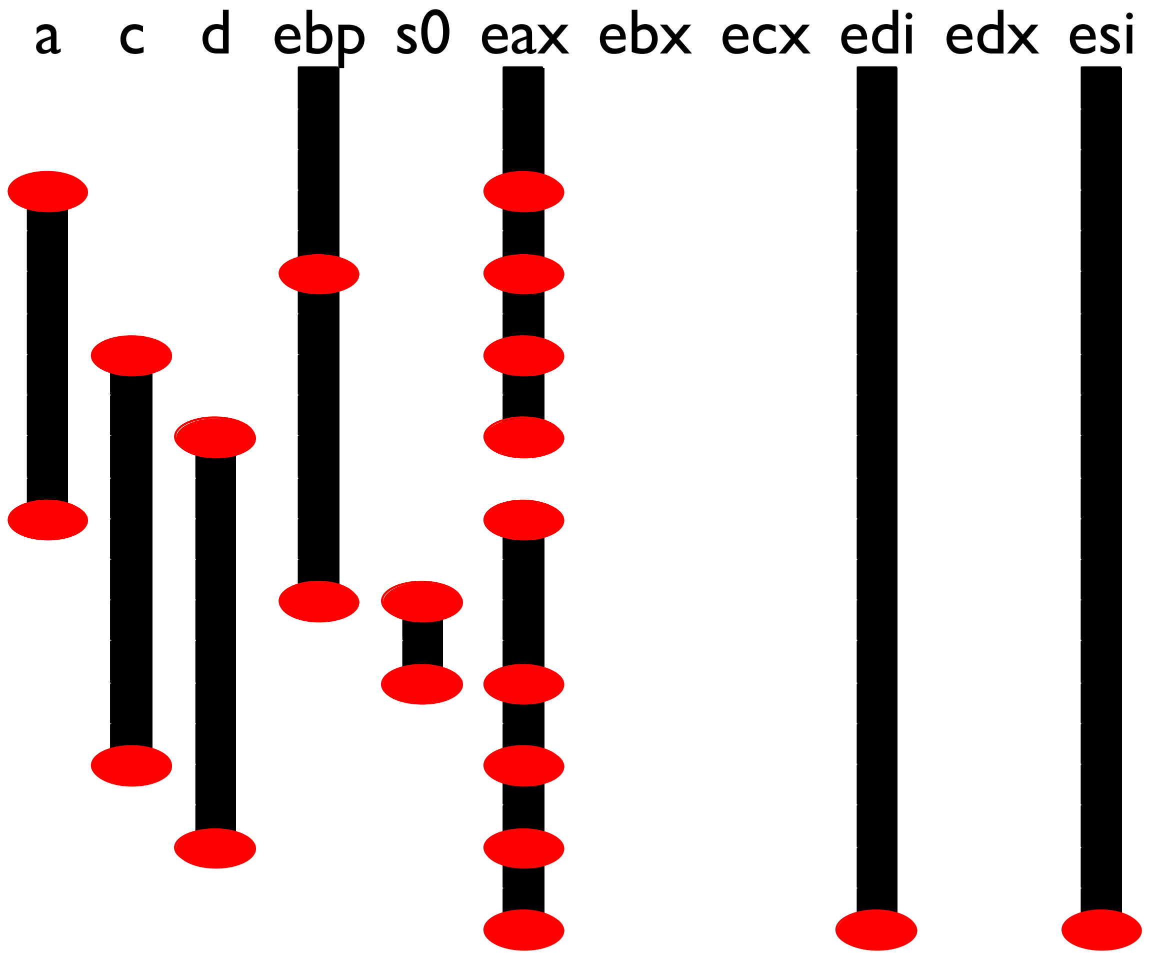
In general, make up a new temporary for each instruction that uses the variable to be spilled

This makes for very short live ranges.

# Example Function 2, spilling b

`int f(int x) = x+x+x+x` (in a stupid compiler)

```
:f
(a <- eax)
((mem ebp -4) <- eax)
(c <- eax)
(d <- eax)
(eax <- a)
(s0 <- (mem ebp -4))
(eax += s0)
(eax += c)
(eax += d)
(return)
```



## 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
```

Here’s how to two example spilled functions from the earlier slides would look like as calls to spill:

```
(spill «the original program»
       'a
       -4
       's)

(spill «the original program»
       'b
       -4
       's)
```

See the assignment handout for more details on the precise spec for test cases and your spill function’s interface