

# Register Allocation, i

Overview & spilling

# L1

```
p ::= (label f ...)

f ::= (label nat nat i ...)

i ::= (w <- s)
| (w <- (mem x n8))
| ((mem x n8) <- s)
| (w aop= t)
| (w sop= sx)
| (w sop= num)
| (w <- t cmp t)
| label
| (goto label)
| (cjump t cmp t label label)
| (call u nat)
| (call print 1)
| (call allocate 2)
| (call array-error 2)
| (tail-call u nat0-6)
| (return)

aop ::= += | -= | *= | &=
sop ::= <<= | >>=
cmp ::= < | <= | =
u ::= w | label
t ::= x | num
s ::= x | num | label
x ::= w | rsp
w ::= a | rax | rbx | rbp | r10 | r11 | r12 | r13 | r14 | r15
a ::= rdi | rsi | rdx | sx | r8 | r9
sx ::= rcx

label ::= sequence of chars matching #rx"^[a-zA-Z_][a-zA-Z_0-9]*$"
```

# L2

```
p ::= (label f ...)

f ::= (label nat nat i ...)

i ::= (w <- s)
| (w <- (mem x n8))
| ((mem x n8) <- s)
| (w aop= t)
| (w sop= sx)
| (w sop= num)
| (w <- t cmp t)
| label
| (goto label)
| (cjump t cmp t label label)
| (call u nat)
| (call print 1)
| (call allocate 2)
| (call array-error 2)
| (tail-call u nat0-6)
| (return)
| (w <- (stack-arg n8))

aop= ::= += | -= | *= | &=
sop= ::= <<= | >>=
cmp ::= < | <= | =
u ::= w | label
t ::= x | num
s ::= x | num | label
x ::= w | rsp
w ::= a | rax | rbx | rbp | r10 | r11 | r12 | r13 | r14 | r15
a ::= rdi | rsi | rdx | sx | r8 | r9
sx ::= rcx | var

label ::= sequence of chars matching #rx"^[a-zA-Z_][a-zA-Z_0-9]*$"
var ::= sequence of chars matching #rx"^[a-zA-Z_][a-zA-Z_0-9-]*$"
```

## L2 semantics: variables

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

```
(define (f x)      ⇒ (:m
  (+ (g x) 1))           (:m 0 0
                                (rdi <- 21)
                                (tail-call :f 1))
(:f 1 0
  (temp <- 2)
  ((mem rsp -8) <- :gret)
  (call :g 1) :gret
  (rax += temp)
  (return))
(:g 1 0
  (temp <- 4)
  (rax <- rdi)
  (rax += temp)
  (return)))
```

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

## L2 semantics: stack-arg

L2 has a convenience for accessing stack-based arguments: `(stack-arg n8)`. It is equivalent to `(mem rsp ?)` where the `?` is the `n8`, plus enough space for the spills. That is, `(stack-arg 0)` is always the last stack argument, `(stack-arg 8)` is always the second to last argument, etc.

This means that if the second natural number in the function header changes, then the `stack-arg` references don't have to change – they will still be referring to the same arguments

# 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  $(\text{x} \leftarrow \text{y})$  instructions

# Example Function

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

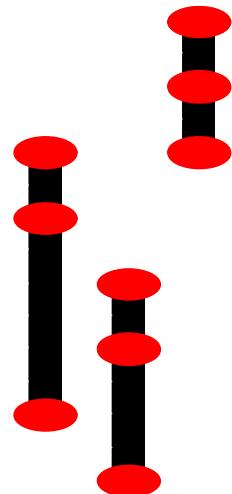
```
(:f
1 0
(x2 <- rdi)
(x2 *= x2)
(dx2 <- x2)
(dx2 *= 2)
(tx <- rdi)
(tx *= 3)
(rax <- dx2)
(rax += tx)
(rax += 4)
(return))
```

# Example Function: live ranges

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

dx2 tx x2

```
(:f
1 0
(x2 <- rdi)
(x2 *= x2)
(dx2 <- x2)
(dx2 *= 2)
(tx <- rdi)
(tx *= 3)
(rax <- dx2)
(rax += tx)
(rax += 4)
(return))
```

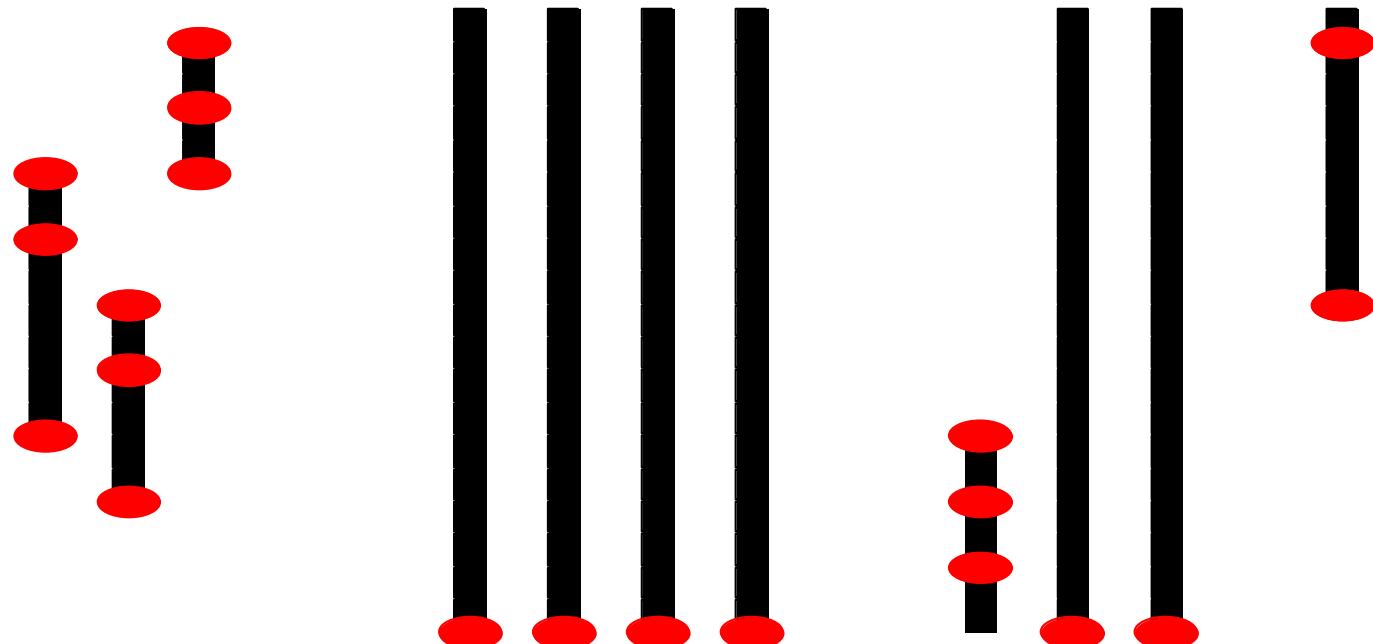


# Example Function: live ranges

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

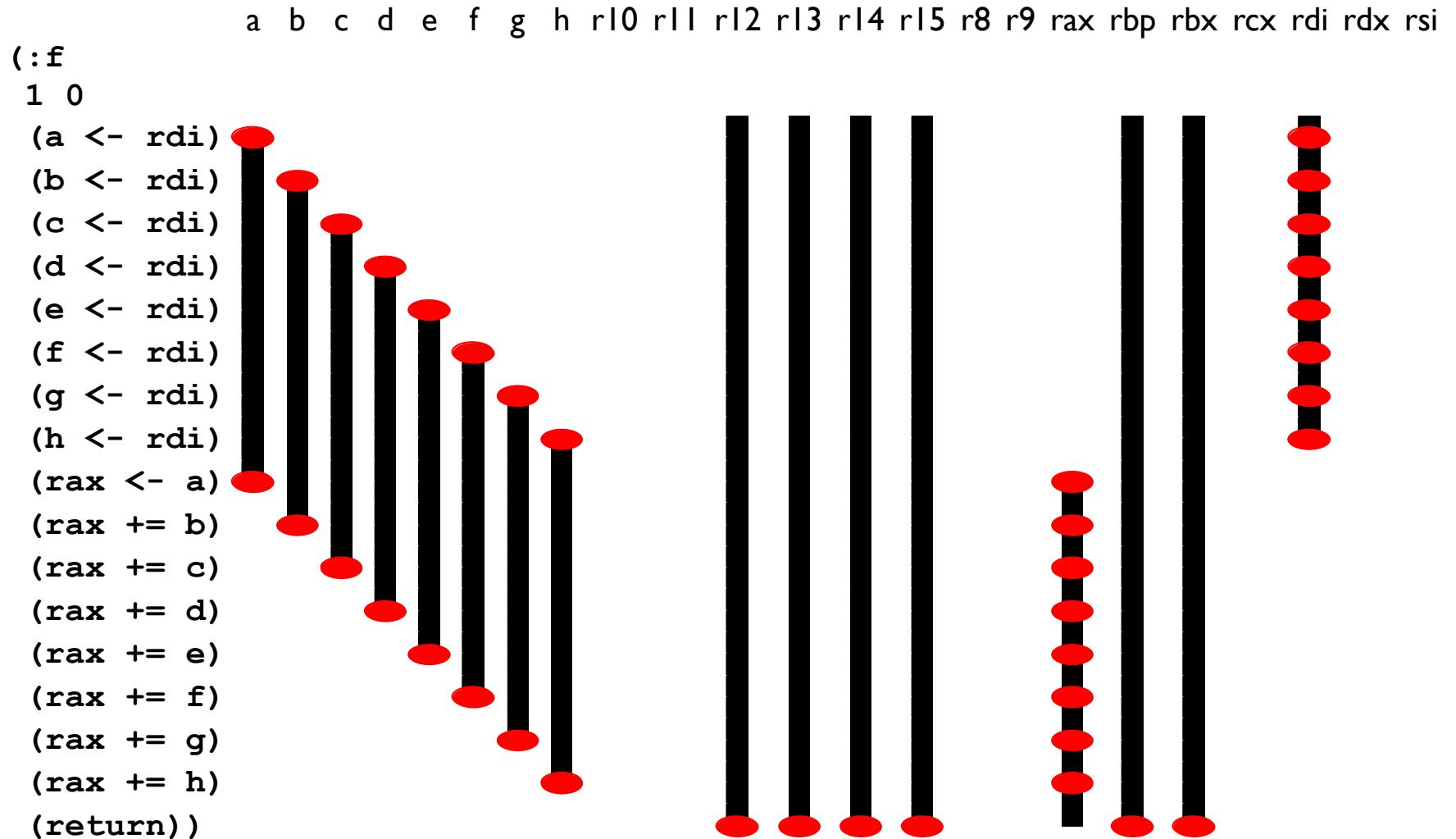
```
dx2 tx x2 r10 r11 r12 r13 r14 r15 r8 r9 rax rbp rbx rcx rdi rdx rsi
```

```
(:f  
1 0  
(x2 <- rdi)  
(x2 *= x2)  
(dx2 <- x2)  
(dx2 *= 2)  
(tx <- rdi)  
(tx *= 3)  
(rax <- dx2)  
(rax += tx)  
(rax += 4)  
(return))
```



# Example Function 2

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



No way to get all of **a**, **b**, **c**, **d**, **e**, **f**, **g**, and **h** 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
- Make a new location on the stack (increment the second `nat` in the function definition)
- Replace all writes to the variable with writes to the new stack location
- Replace all reads from the variable with reads from the new stack location

Sometimes that means introducing new temporaries

# Spilling Example

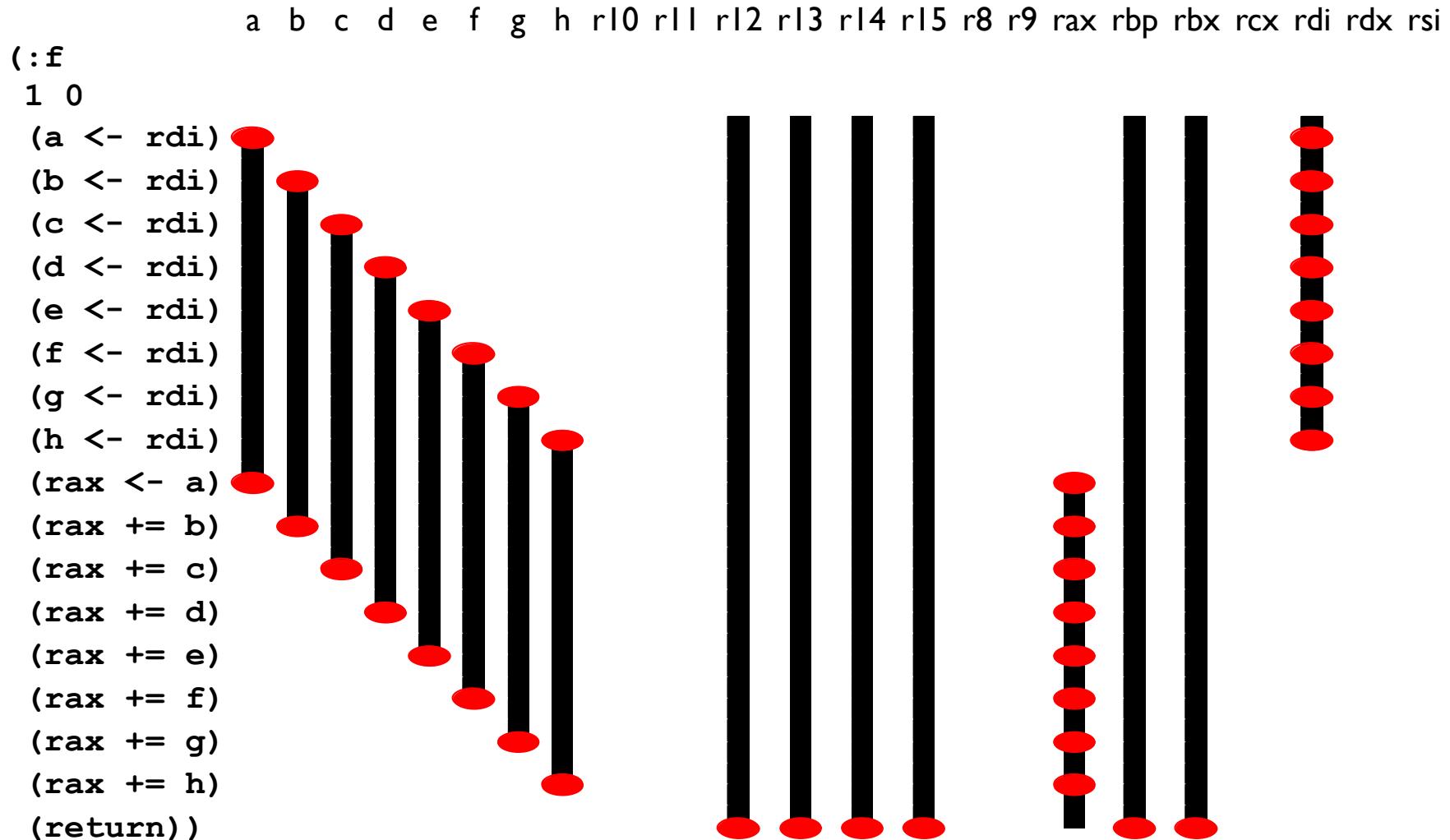
Say we want to spill **a** to first location on the stack,  
**(mem rsp 0)**; two easy cases:

**(a <- 1) ⇒ ((mem rsp 0) <- 1)**

**(x <- a) ⇒ (x <- (mem rsp 0))**

# Example Function 2, need to spill

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



## Example Function 2, spilling a

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

b c d e f g h r10 r11 r12 r13 r14 r15 r8 r9 rax rbp rbx rcx rdi rdx rsi

```
(:f
1 1
((mem rsp 0) <- rdi)
(b <- rdi)
(c <- rdi)
(d <- rdi)
(e <- rdi)
(f <- rdi)
(g <- rdi)
(h <- rdi)
(rax <- (mem rsp 0))
(rax += b)
(rax += c)
(rax += d)
(rax += e)
(rax += f)
(rax += g)
(rax += h)
(return))
```

The diagram illustrates the stack layout and memory writes for the function f. The stack grows downwards from the top. Red dots at the top of each bar indicate the current stack pointer position. The bars represent memory locations: rdi (8 bars), r10-r15 (5 bars), r8-r9 (2 bars), rax (1 bar), and rbp/rbx/rcx (3 bars). The stack grows from the bottom of the diagram towards the top, with red dots appearing at the top of each bar as they are written to.

# Spilling Example

A trickier case:

```
(a *= a) ⇒ (a_new <- (mem rsp 0))  
           (a_new *= a_new)  
           ((mem rsp 0) <- a_new)
```

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+x+x+x+x (in a stupid compiler)**

```
(:f
1 1
(a <- rdi)
((mem rsp 0) <- rdi)
(c <- rdi)
(d <- rdi)
(e <- rdi)
(f <- rdi)
(g <- rdi)
(h <- rdi)
(rax <- a)
(s0 <- (mem rsp 0))
(rax += s0)
(rax += c)
(rax += d)
(rax += e)
(rax += f)
(rax += g)
(rax += h)
(return))
```

The diagram illustrates the execution flow of the function f. It shows two columns of vertical bars representing the state of variables a through h. In the first column (left), each variable has a black bar above it and a red dot at its base, indicating they are currently in registers. In the second column (right), after the addition loop has completed, each variable still has a black bar above it and a red dot at its base, indicating they have been spilled to memory.

## Example Function 2, spilling b

Even though we still have eight temporaries, we can still allocate them to our seven 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 : (label nat nat i ...) ;; original func
  var          ;; to spill
  var          ;; prefix for temporaries
-> (label nat nat i ...) ;; spilled func
```

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

(spill «*the original program*»

' a

' s)

(spill «*the original program*»

' b

' s)

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