Design and Implementation of GCC Register Allocation

HelloGCC'2014

Date : Sep 13th, 2014

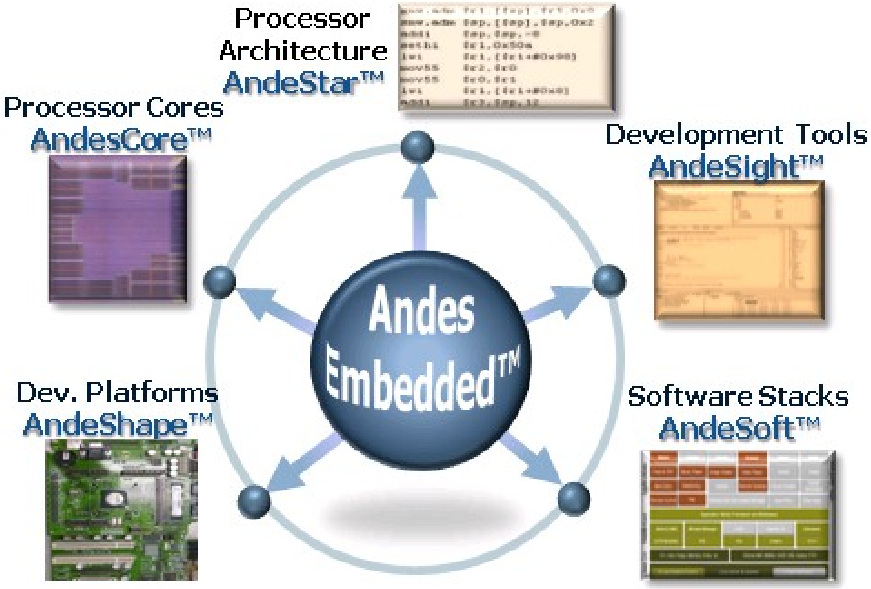
Kito Cheng

kito.cheng@gmail.com

500,000,000 Andes-Embedded SoC

100 Agreements

9.5 Year



Processor Architecture
AndeStar™

Processor Cores
AndesCore™

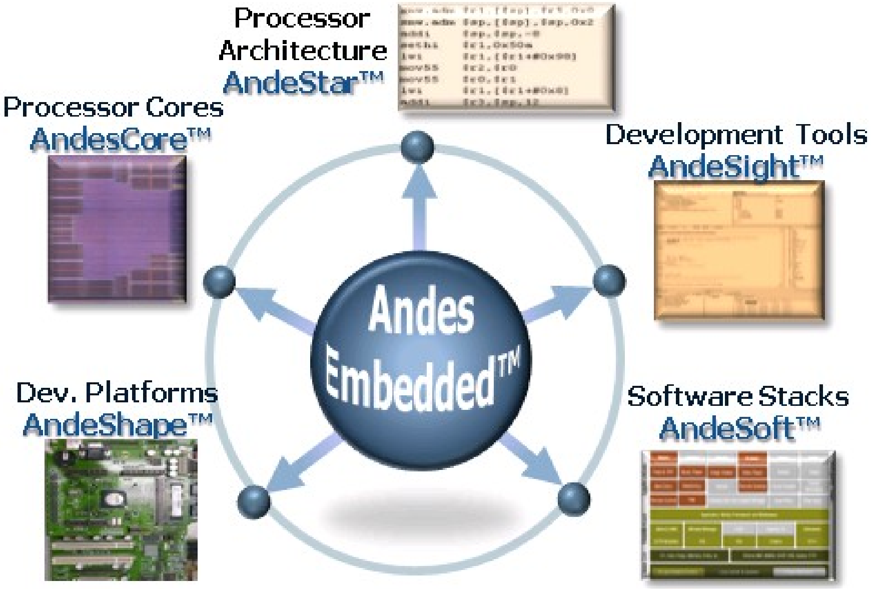
Development Tools
AndeSight™

Andes Embedded™

Dev. Platforms
AndeShape™

Software Stacks
AndeSoft™

- **7** AndesCores, applications ranging from **8-bit CPU upgrade to Linux-based Networking**
- Customers mainly in **Taiwan** and **China**, but we’re growing into **Korea**, **Japan** and **North America.**



Register Allocation

```
int foo(int n)
{
  int i, y, t, z;
  int x = n * 2;
  z = x;
  int sum = 0;
  for (i=0; i<n; ++i) {
    y = i;
    t = z + y;
    sum = sum + t;
  }
  return sum;
}
```

```
foo:
          addi      $sp, $sp, -8
          slli      $r1, $r0, 1
          swi       $r1, [$sp]
          movi      $r1, 0
          movi      $r2, 0
          swi       $r0, [$sp + (4)]
          j         .L2
.L3:
          mov   $r3, $r2
          lwi   $r0, [$sp]
          add   $r3, $r0, $r3
          add   $r1, $r1, $r3
          addi  $r2, $r2, 1
.L2:
          lwi   $r0, [$sp + (4)]
          slts  $ta, $r2, $r0
          bnez  $ta, .L3
          mov   $r0, $r1
          addi  $sp, $sp, 8
          ret
```

Register Allocation

```
int foo(int n)
{
  int i, y, t, z;
  int x = n * 2;
  z = x;
  int sum = 0;
  for (i=0; i<n; ++i) {
    y = i;
    t = z + y;
    sum = sum + t;
  }
  return sum;
}
```

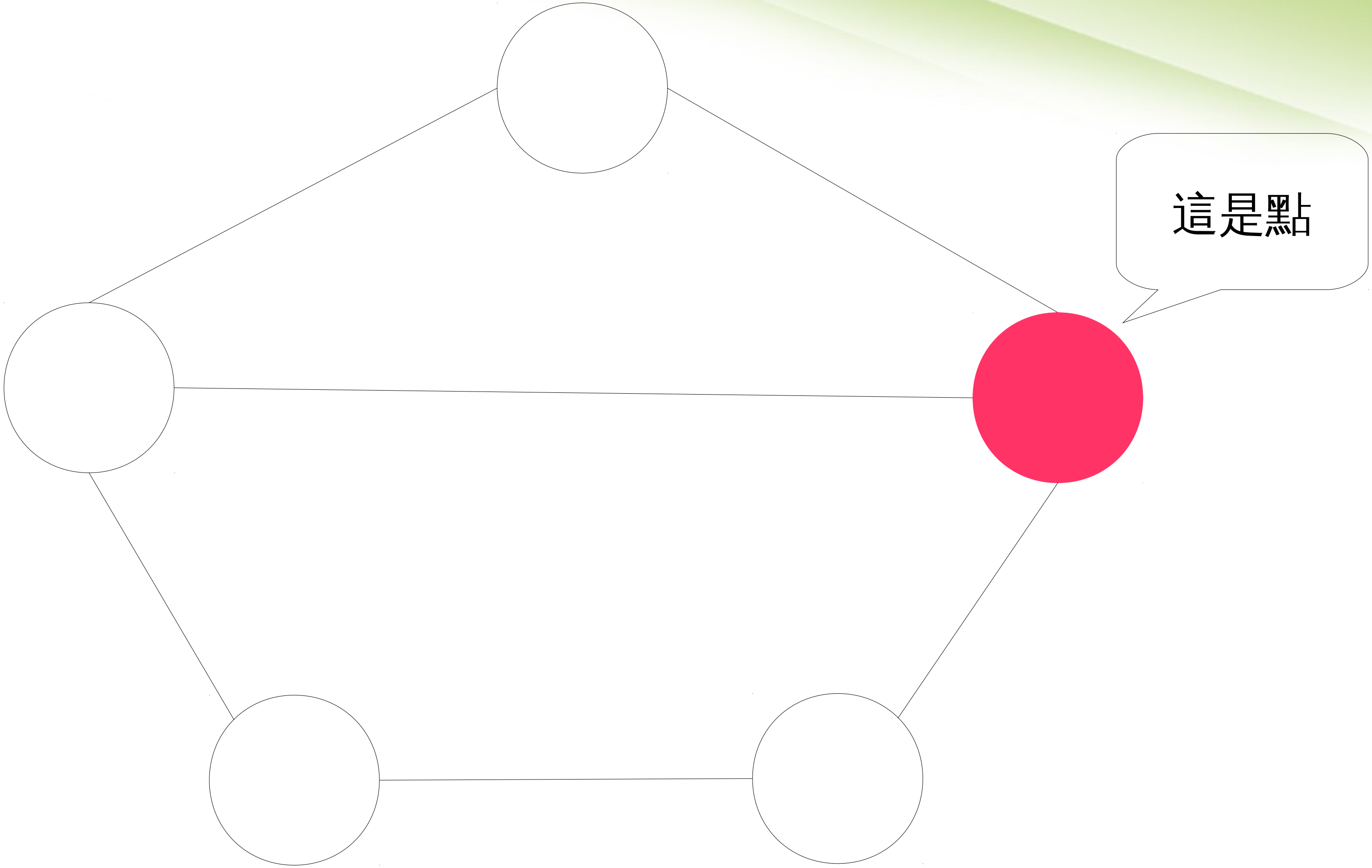
```
foo:
          addi      $sp, $sp, -8
          slli      $r1, $r0, 1
          swi       $r1, [$sp]
          movi      $r1, 0
          movi      $r2, 0
          swi       $r0, [$sp + (4)]
          j         .L2
.L3:
          mov   $r3, $r2
          lwi   $r0, [$sp]
          add   $r3, $r0, $r3
          add   $r1, $r1, $r3
          addi  $r2, $r2, 1
.L2:
          lwi   $r0, [$sp + (4)]
          slts  $ta, $r2, $r0
          bnez  $ta, .L3
          mov   $r0, $r1
          addi  $sp, $sp, 8
          ret
```

Graph Coloring
基礎理論

Graph-Coloring

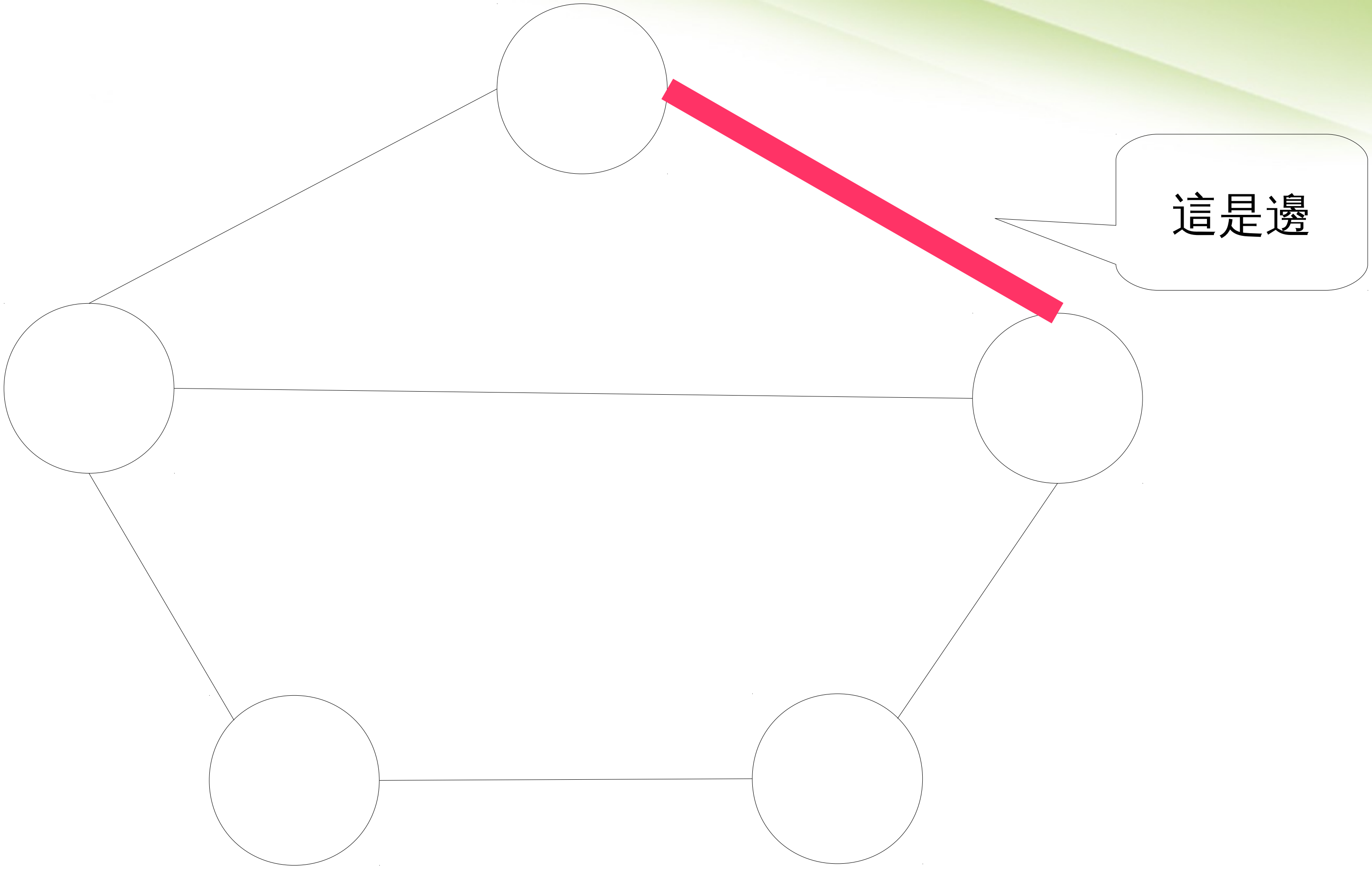
Graph（圖） ＝ Vertex/Node（點）+ Edge（邊）

Graph-Coloring



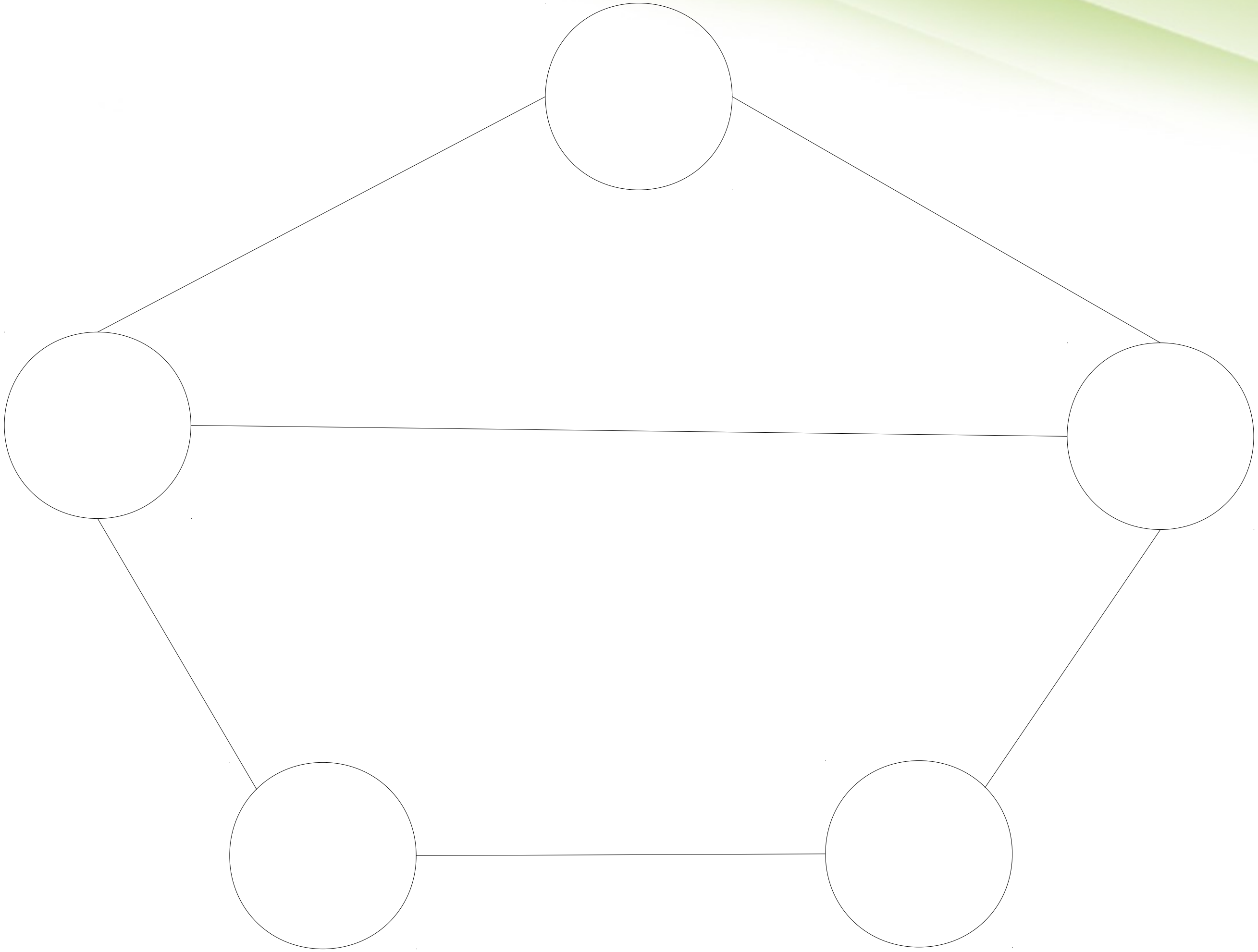
Graph（圖） ＝ Vertex/Node（點）+ Edge（邊）

Graph-Coloring



Graph（圖） ＝ Vertex/Node（點）+ Edge（邊）

Graph-Coloring

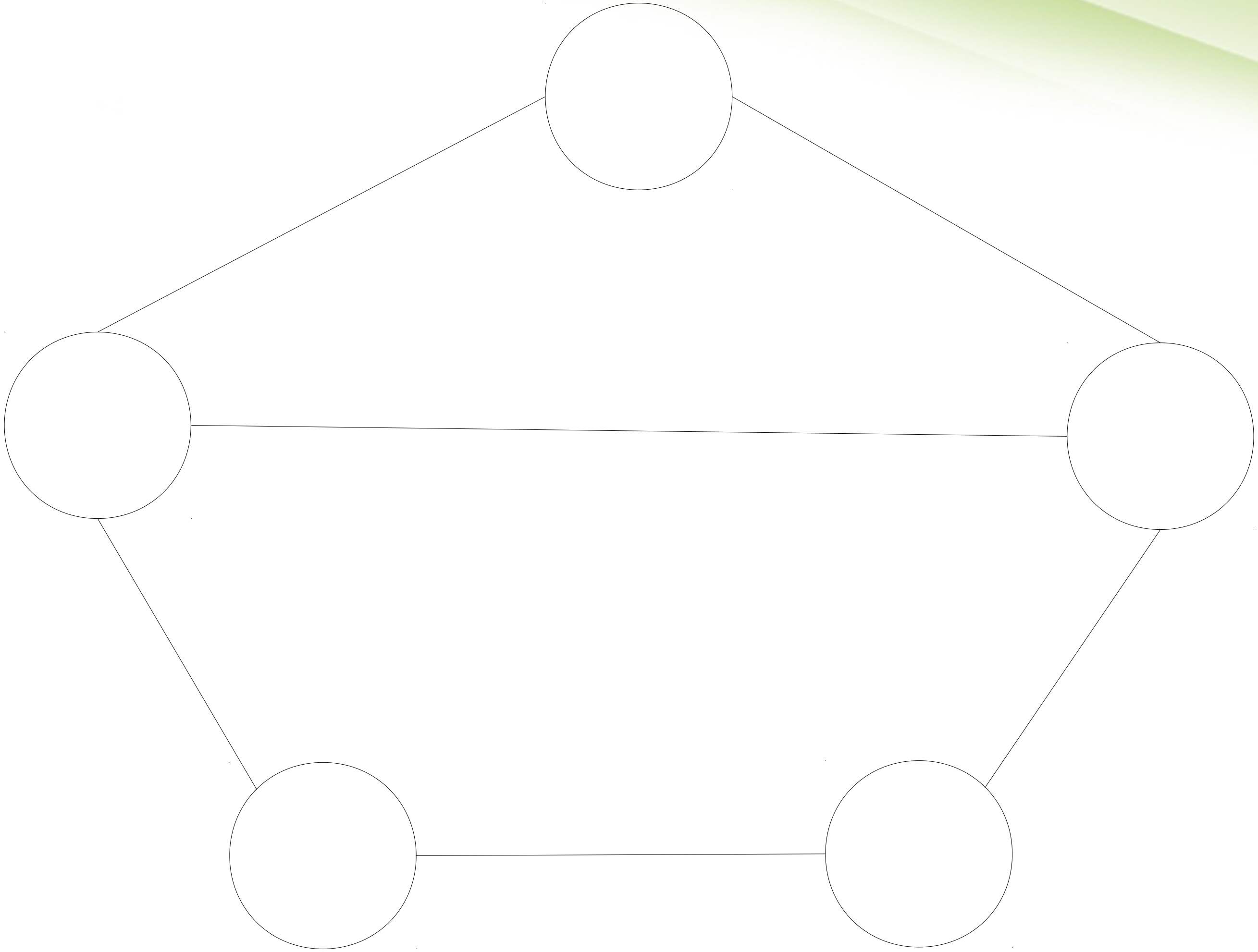


上色問題就是把**點**上顏色，
並且不能與相鄰的**點**同顏色

Graph-Coloring

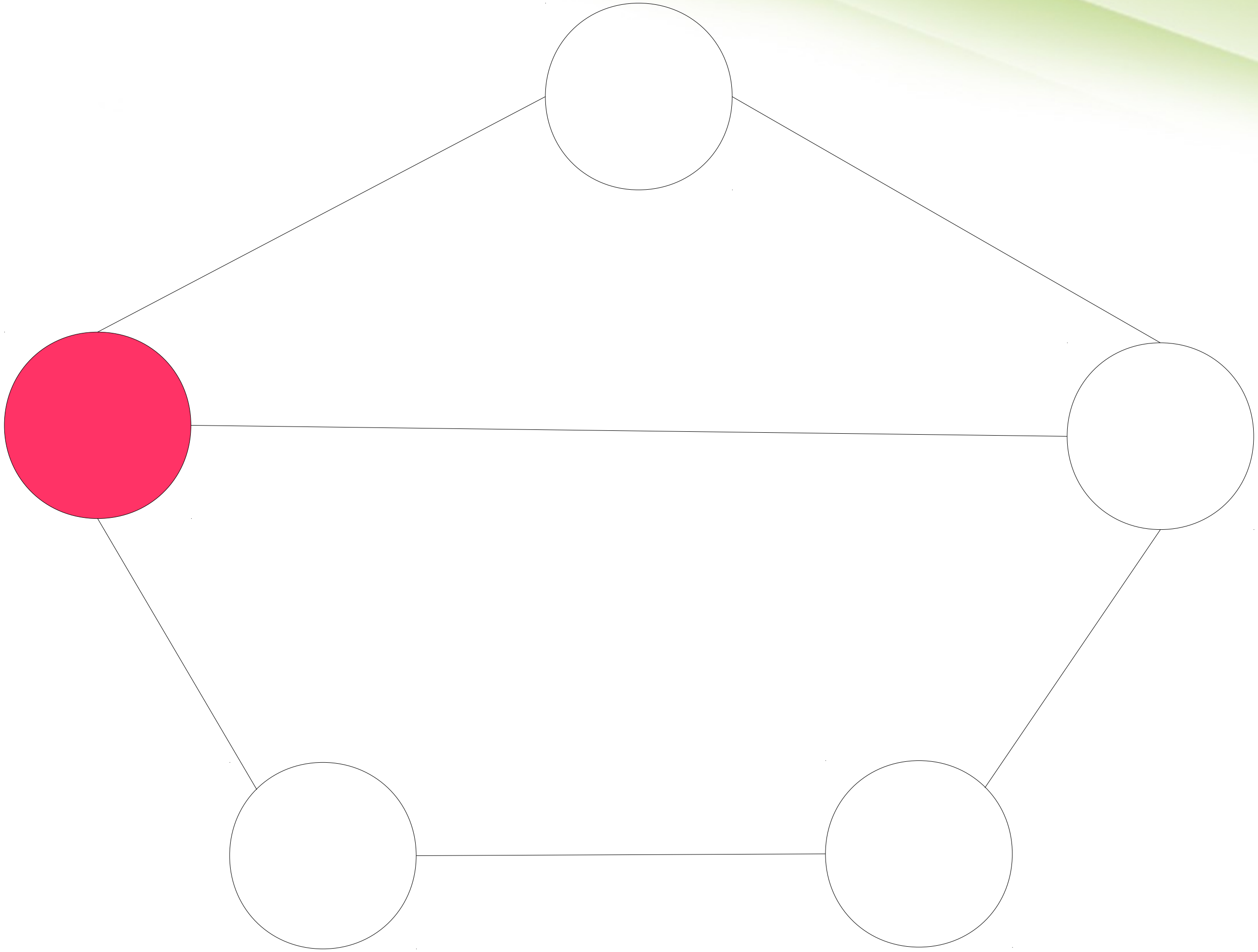
用最經典的演算法上色：
假設有 N 個顏色,
若邊小於 N 個則可隨意上色

Graph-Coloring



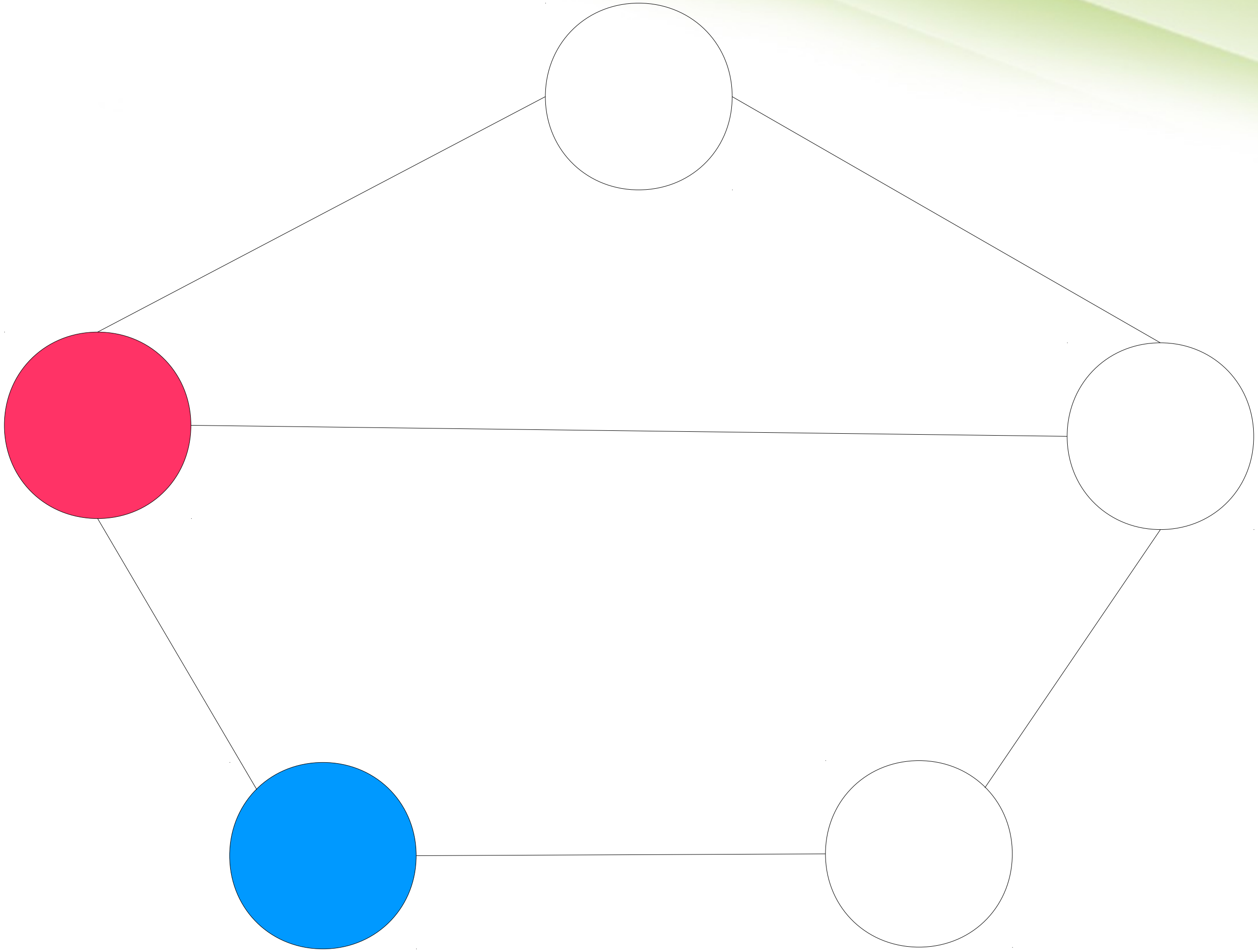
假設有四個顏色，若邊小於四則可隨意上色

Graph-Coloring



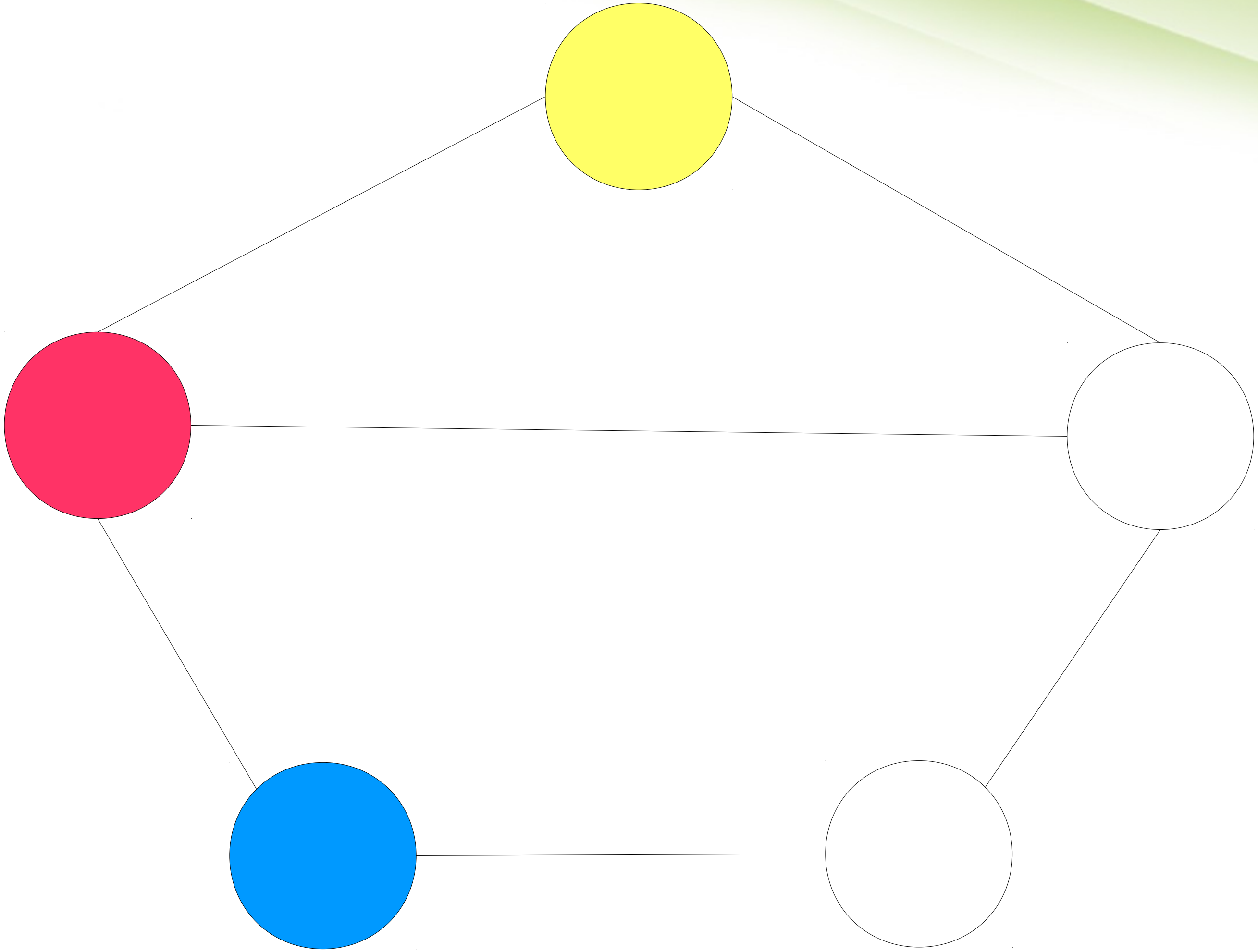
假設有四個顏色，若邊小於四則可隨意上色

Graph-Coloring



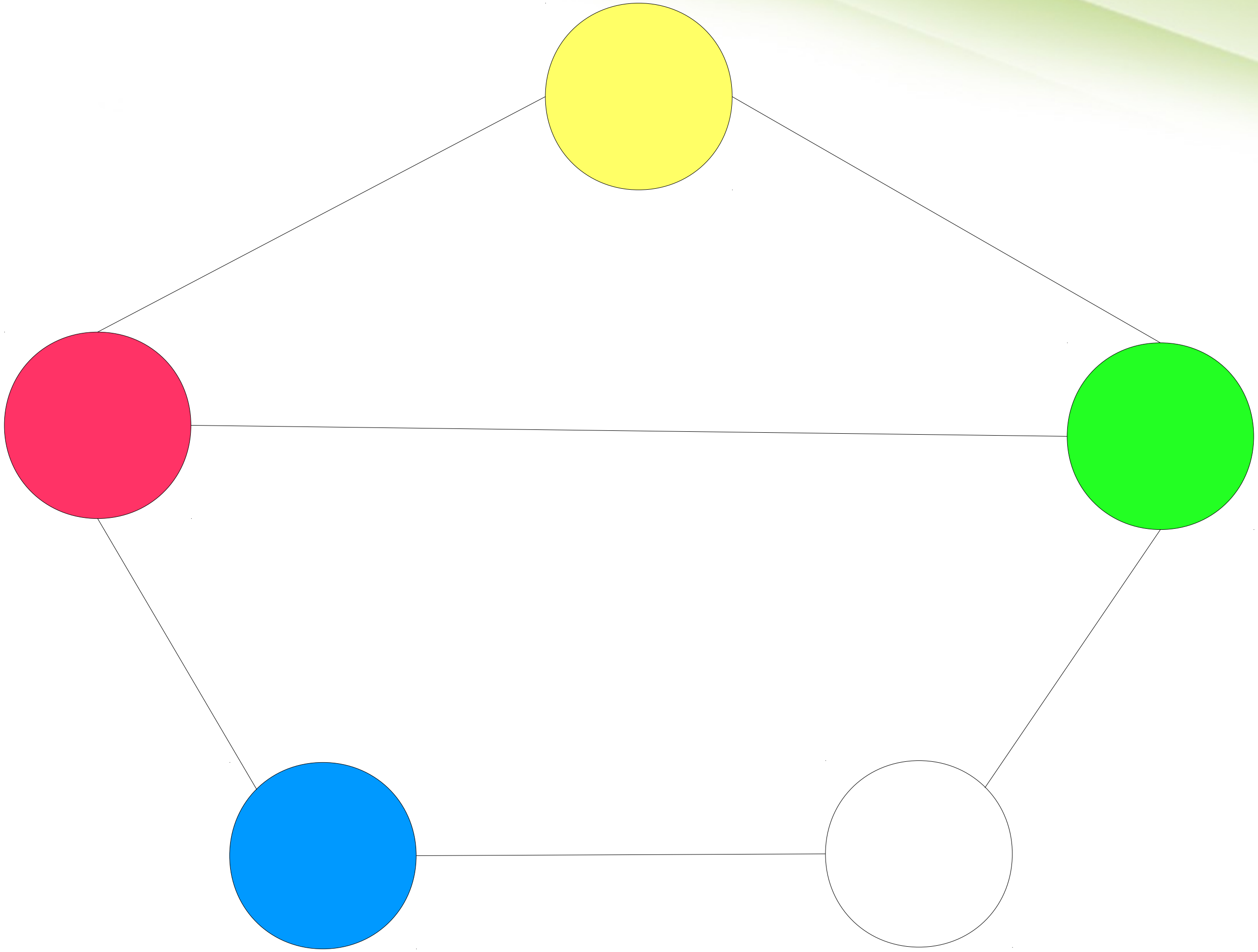
假設有四個顏色，若邊小於四則可隨意上色

Graph-Coloring



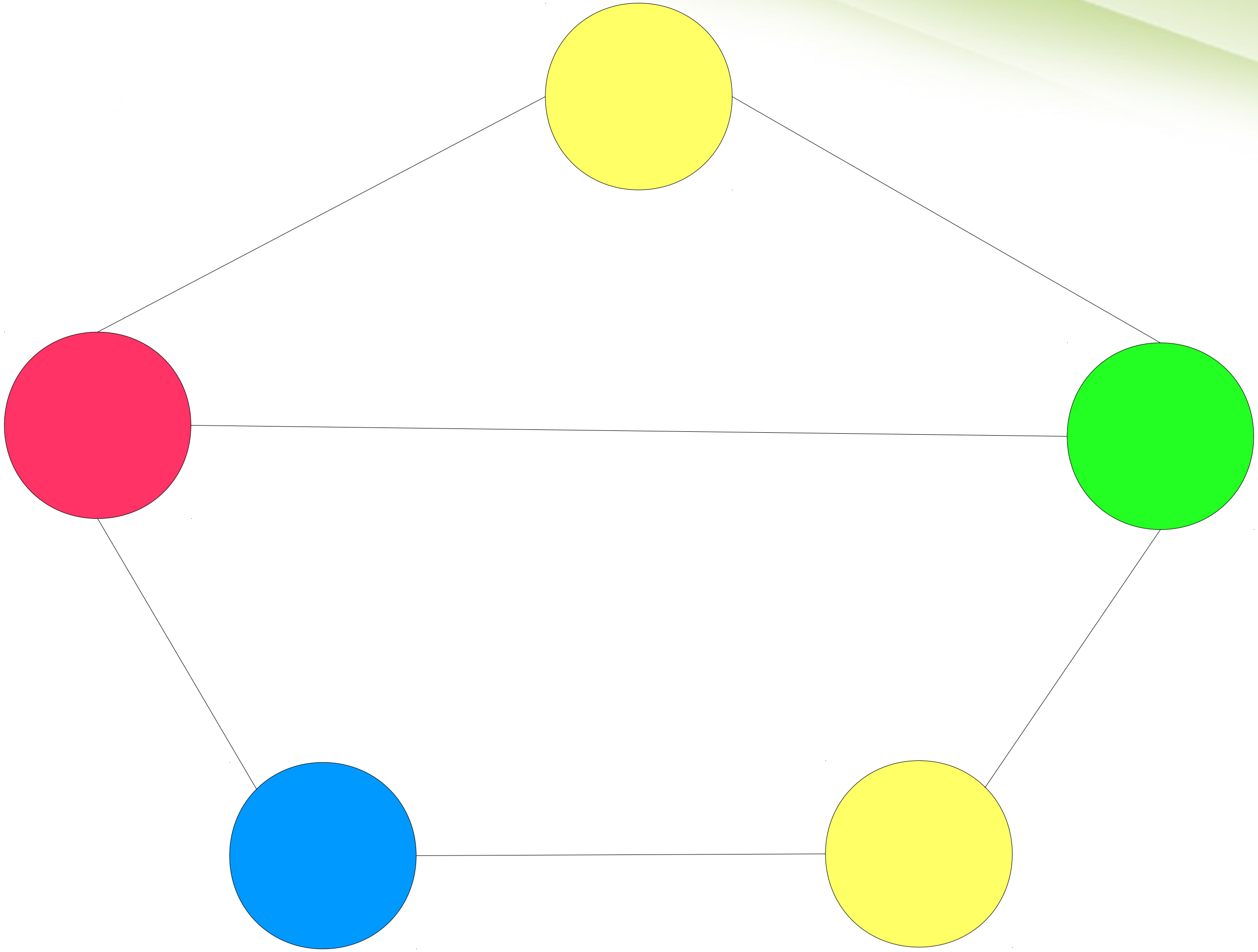
假設有四個顏色，若邊小於四則可隨意上色

Graph-Coloring



假設有四個顏色，若邊小於四則可隨意上色

Graph-Coloring



假設有四個顏色，若邊小於四則可隨意上色

Graph-Coloring

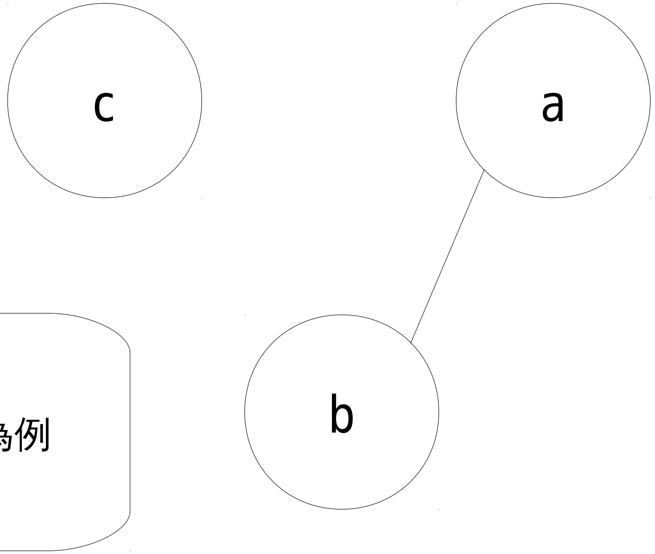
那跟 Register Allocation 的關聯？

Interference Graph

那跟 Register Allocation 的關聯？

```
int a = 10, b = 20;
int c = a + b;
```

以 c = a + b 為例



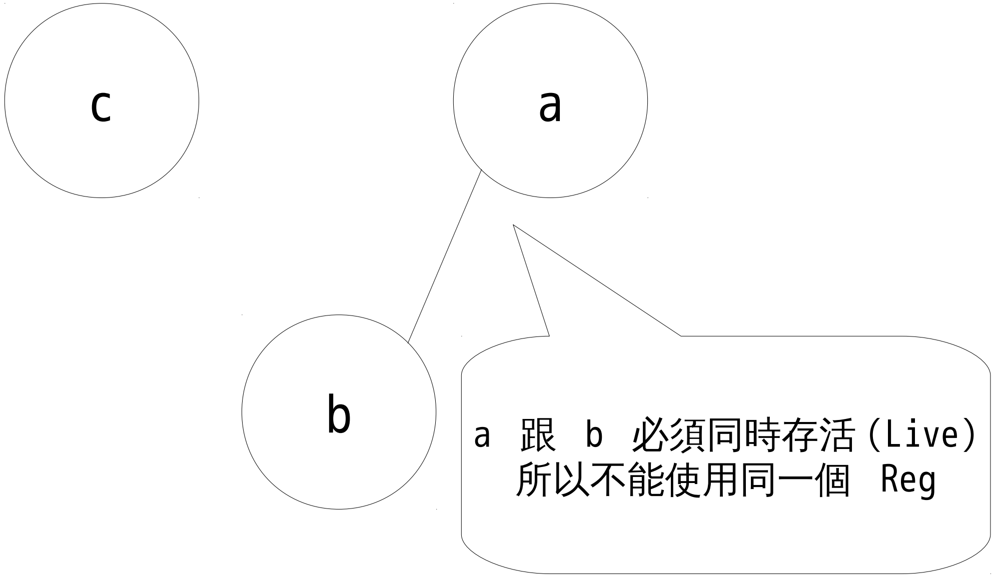
Interference Graph

那跟 Register Allocation 的關聯？

```
int a = 10, b = 20;
int c = a + b;
```

Live Range

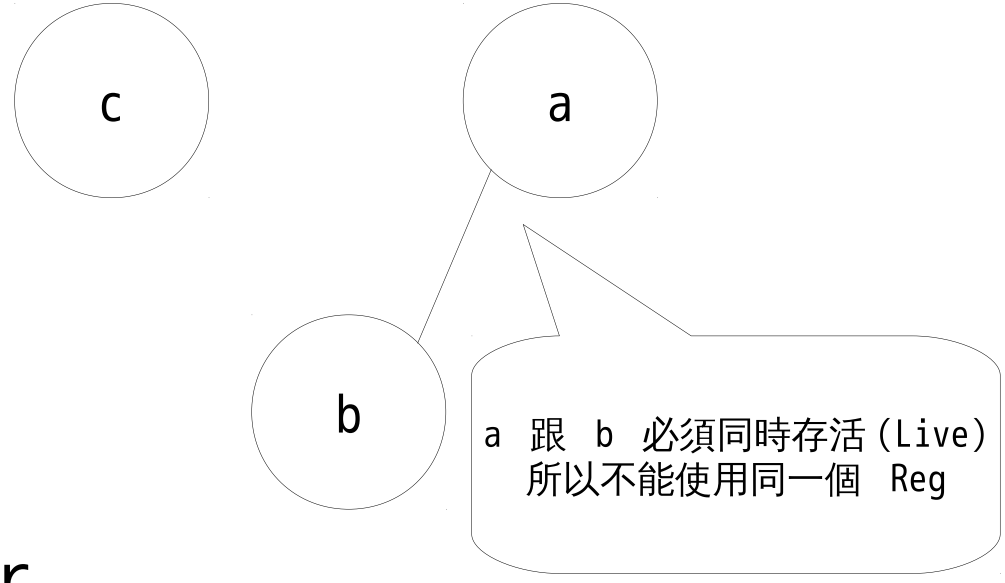
	a	b	c
a=10, b=20			
	🟩	🟩	
c = a + b	🟩	🟩	
			🟩



a 跟 b 必須同時存活(Live)
所以不能使用同一個 Reg

Interference Graph

```
int a = 10, b = 20;
int c = a + b;
```



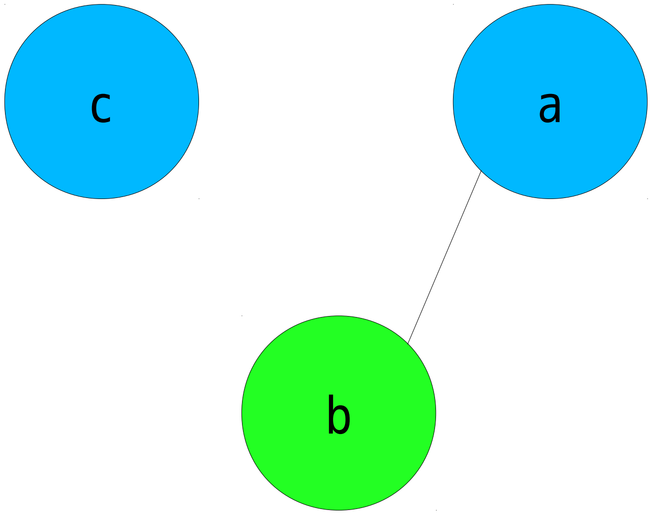
a 跟 b 必須同時存活(Live)
所以不能使用同一個 Reg

顏色 = Register

n 個顏色可用 = n 個暫存器可用

Interference Graph

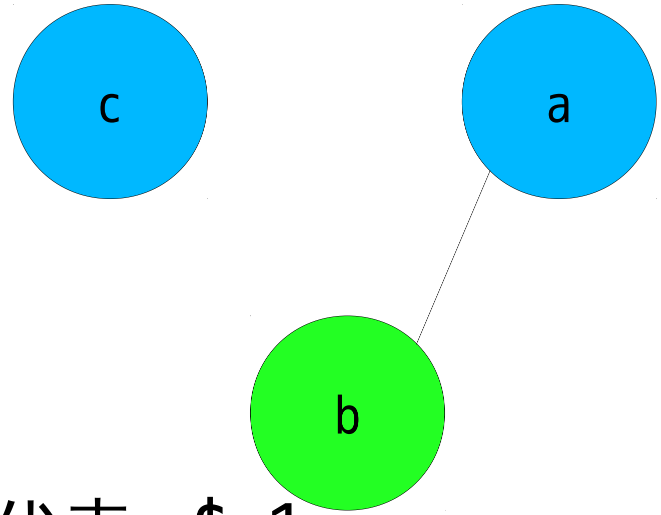
```
int a = 10, b = 20;
int c = a + b;
```



任意上色一下

Interference Graph

```
int a = 10, b = 20;
int c = a + b;
```



假設藍色代表 $r0，綠色代表 $r1
亦即此上色結果所產生的程式碼為
add $r0, $r0, $r1

假設有 N 個顏色，
若邊小於 N 個則可隨意上色

假設有 N 個顏色,
若邊小於 N 個則可隨意上色

↓

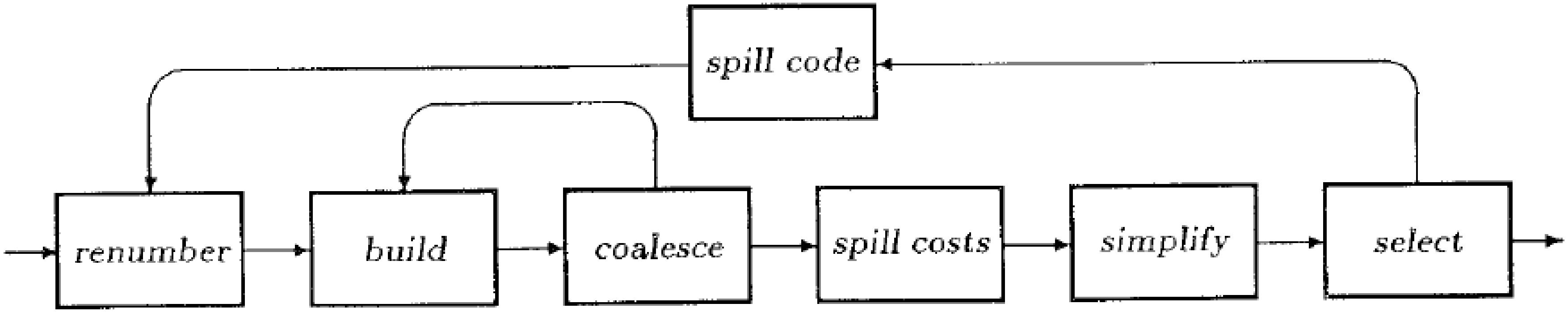
假設有 N 個Register,
若邊小於 N 個則可隨意Allocation

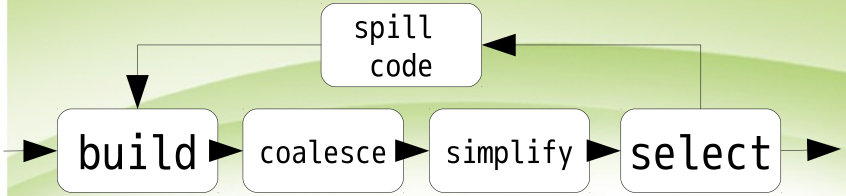
以上就是
Graph Coloring-based
Register Allocation
的概念

經典
Graph Coloring
Register Allocation

Chaitin-Briggs Algorithm

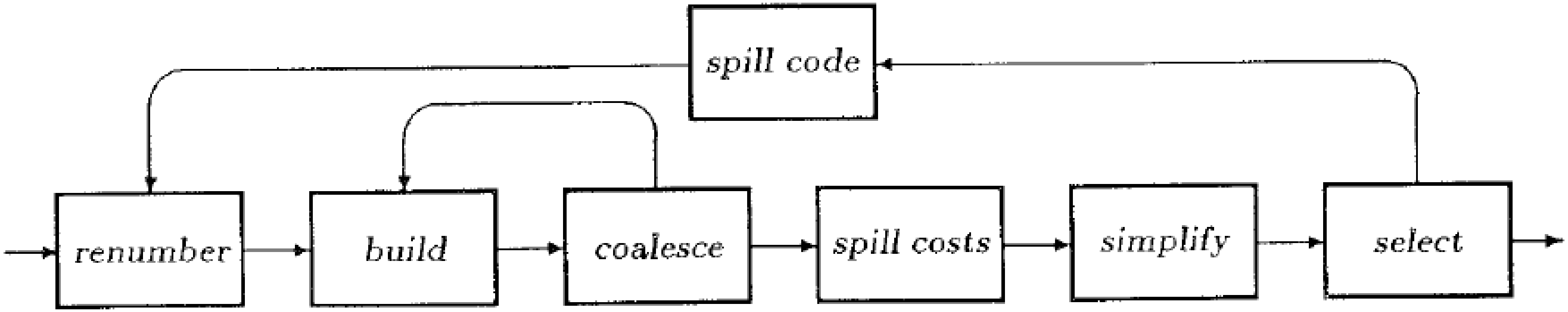
- 在深入 GCC 前先複習下最經典的演算法：

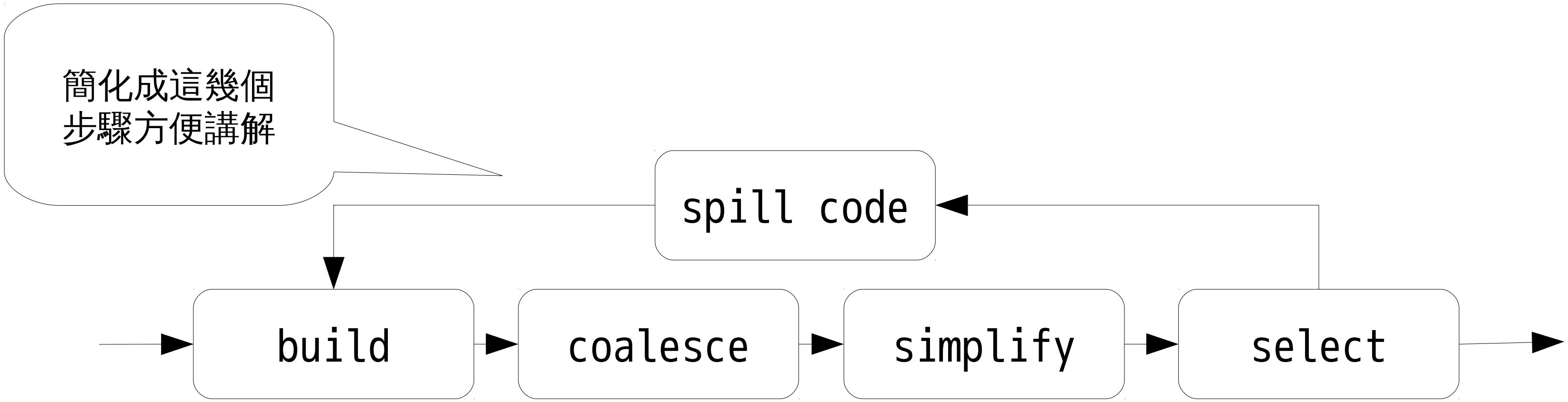


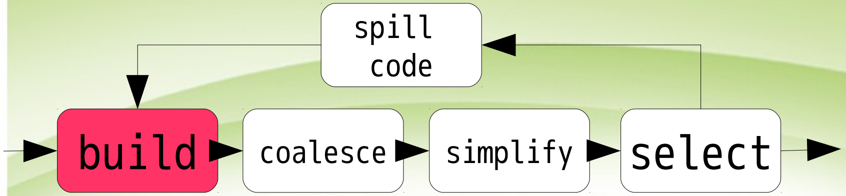


Chaitin-Briggs Algorithm

- 在深入 GCC 前先複習下最經典的演算法：

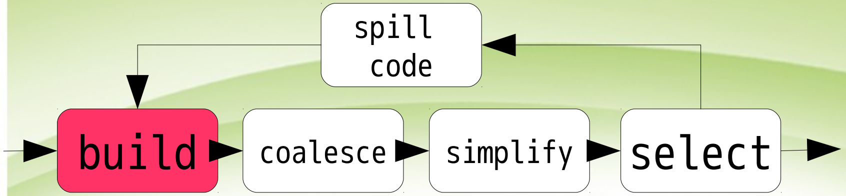



簡化成這幾個步驟方便講解



Example

```
int foo(int n)
{
  int i, y, t, z;
  int x = n * 2;
  z = x;
  int sum = 0;
  for (i=0; i<n; ++i) {
    y = i;
    t = z + y;
    sum = sum + t;
  }
  return sum;
}
```



spill code

build

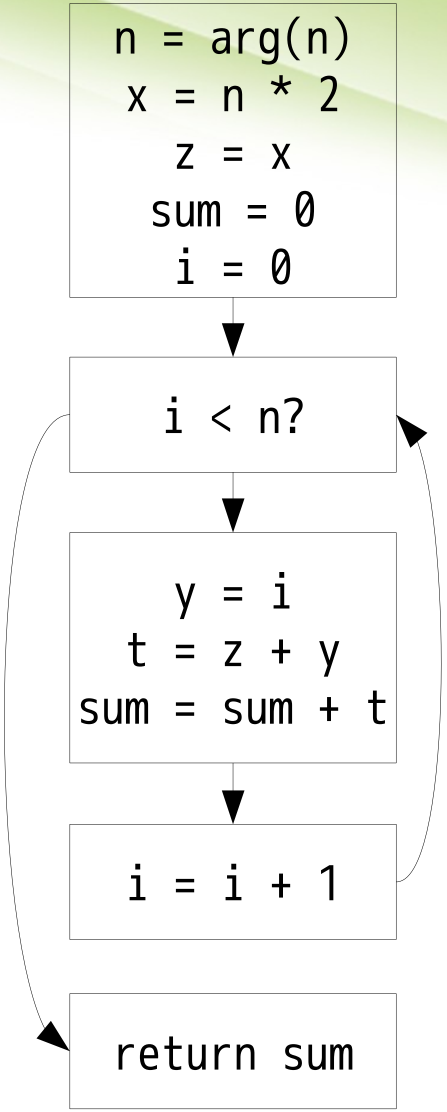
coalesce

simplify

select

CFG

```
int foo(int n)
{
  int i, y, t, z;
  int x = n * 2;
  z = x;
  int sum = 0;
  for (i=0; i<n; ++i) {
    y = i;
    t = z + y;
    sum = sum + t;
  }
  return sum;
}
```



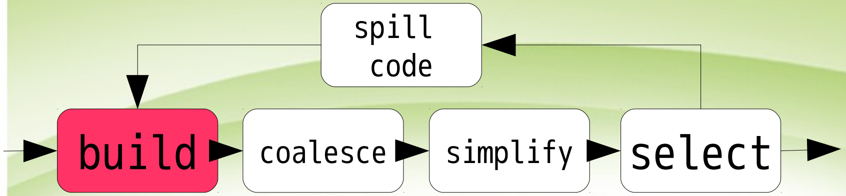
```
n = arg(n)
x = n * 2
z = x
sum = 0
i = 0
```

```
i < n?
```

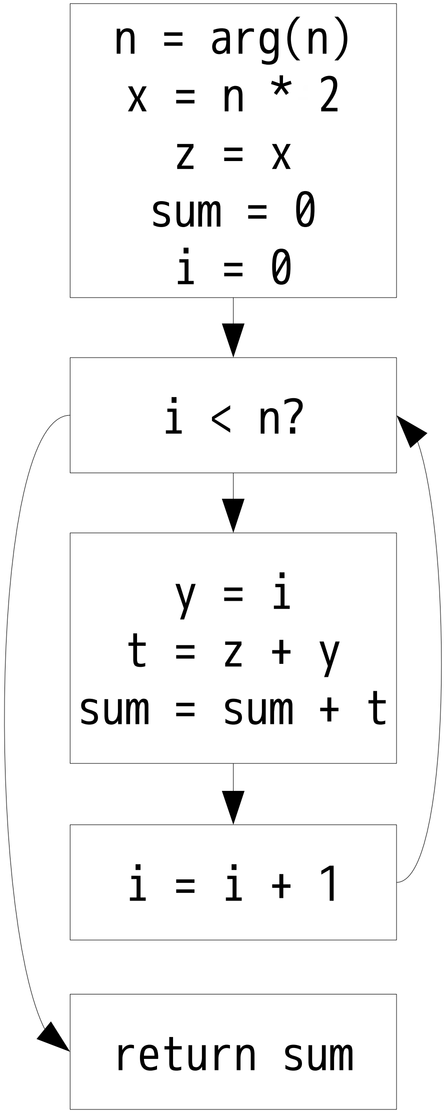
```
y = i
t = z + y
sum = sum + t
```

```
i = i + 1
```

```
return sum
```



Live Range



n = arg(n)
x = n * 2
z = x
sum = 0
i = 0

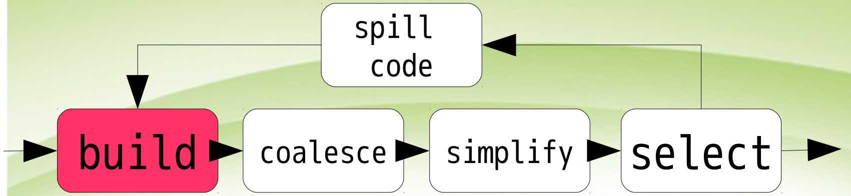
i < n?

y = i
t = z + y
sum = sum + t

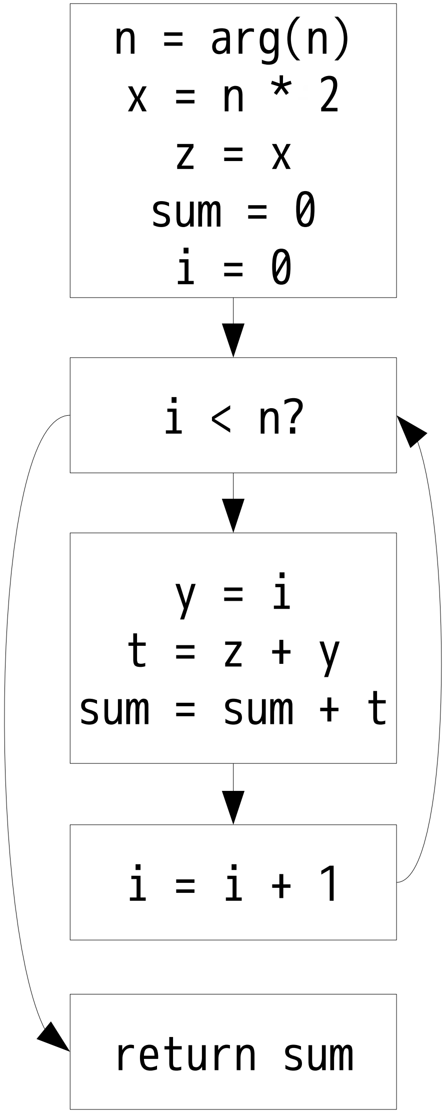
i = i + 1

return sum

	n	x	z	sum	i	y	t
n = arg (n)							
x = n * 2							
z = x							
sum = 0							
i = 0							
i < n?							
y = i							
t = z + y							
sum = sum + t							
i = i + 1							
return sum							

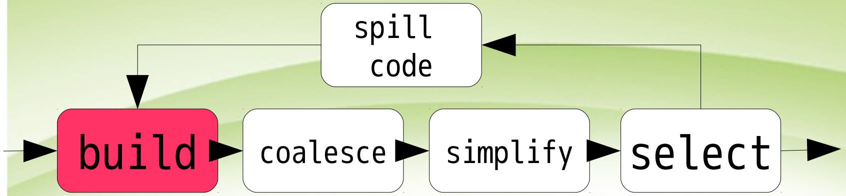


Live Range

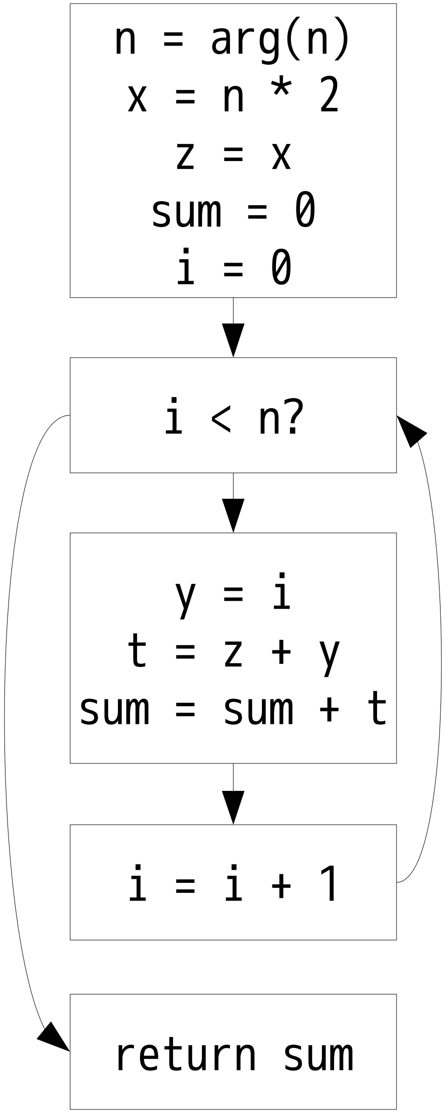


	n	x	z	sum	i	y	t
n = arg (n)							
x = n * 2	🟦	🟦	🟦	🟦	🟦	🟦	🟦
	🟧	🟧	🟧	🟧	🟧	🟧	🟧
z = x							
sum = 0							
i = 0							
i < n?							
y = i							
t = z + y							
sum = sum + t							
i = i + 1							
return sum							

每道指令有兩個
Program point,
Read, Write



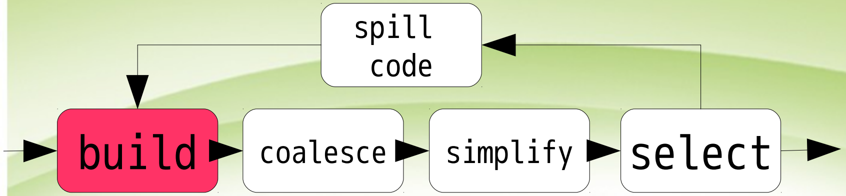
Live Range



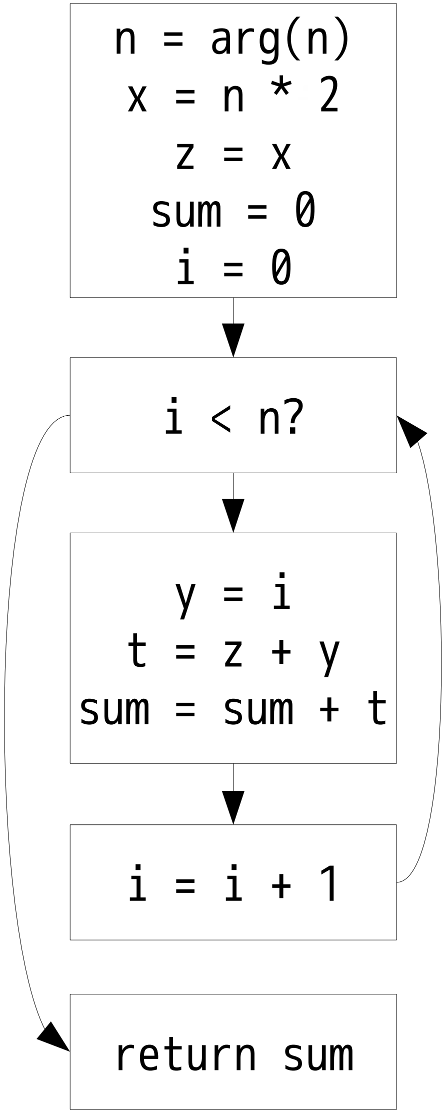
	n	x	z	sum	i	y	t
n = arg (n)							
x = n * 2							
z = x							
sum = 0							
i = 0							
i < n?							
y = i							
						🟩	
t = z + y						🟩	
sum = sum + t							
i = i + 1							
return sum							

y 於 y = i 的 Write 時 Live

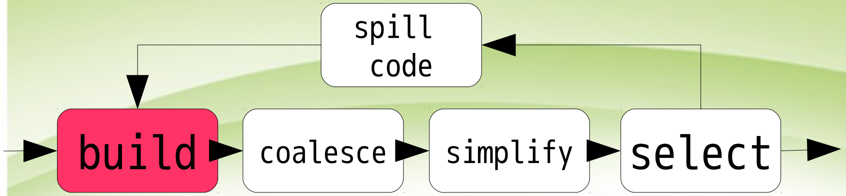
於 t = z + y 的 Read 時 最後使用



Live Range



	n	x	z	sum	i	y	t
n = arg (n)							
	🟩						
x = n * 2	🟩						
	🟩	🟩					
z = x	🟩	🟩					
	🟩		🟩				
sum = 0	🟩		🟩				
	🟩		🟩	🟩			
i = 0	🟩		🟩	🟩			
	🟩		🟩	🟩	🟩		
i < n?	🟩		🟩	🟩	🟩		
	🟩		🟩	🟩	🟩		
y = i	🟩		🟩	🟩	🟩		
	🟩		🟩	🟩	🟩	🟩	
t = z + y	🟩		🟩	🟩	🟩	🟩	
	🟩		🟩	🟩	🟩		🟩
sum = sum + t	🟩		🟩	🟩	🟩		🟩
	🟩		🟩	🟩	🟩		
i = i + 1	🟩		🟩	🟩	🟩		
	🟩		🟩	🟩	🟩		
return sum				🟩			
				🟩			



spill code

build

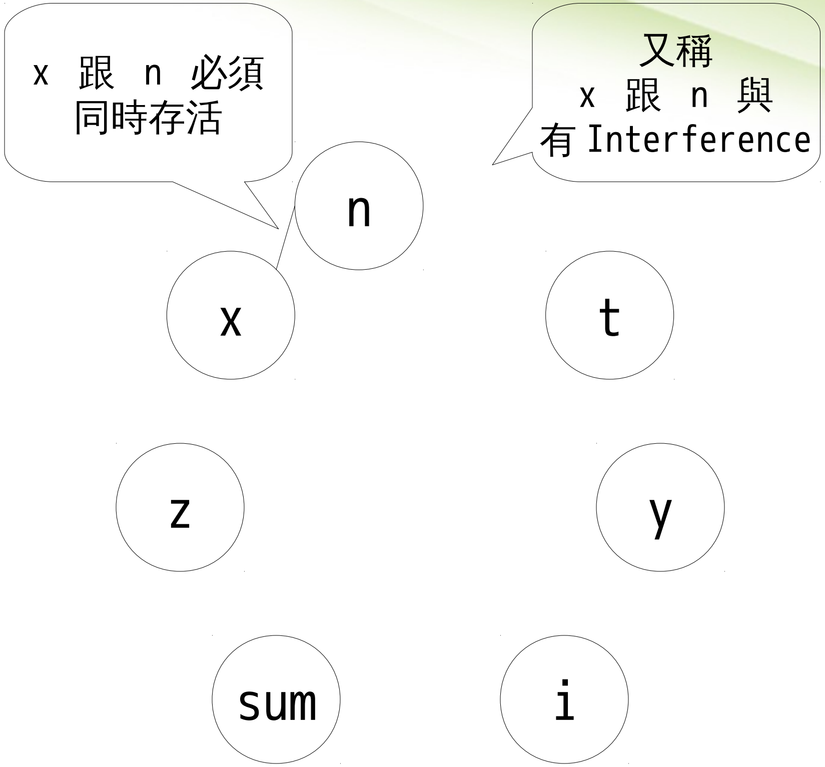
coalesce

simplify

select

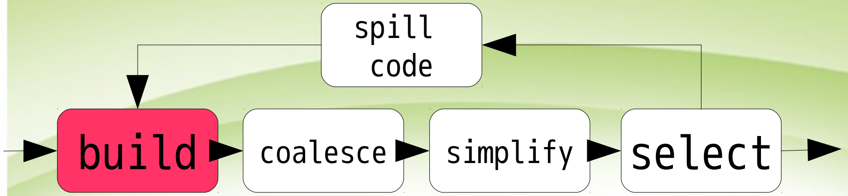
Interference Graph

	n	x	z	sum	i	y	t
n = arg (n)							
	🟩						
x = n * 2	🟩						
	🟩	🟩					
z = x	🟩	🟩					
	🟩		🟩				
sum = 0	🟩		🟩				
	🟩		🟩	🟩			
i = 0	🟩		🟩	🟩			
	🟩		🟩	🟩	🟩		
i < n?	🟩		🟩	🟩	🟩		
	🟩		🟩	🟩	🟩		
y = i	🟩		🟩	🟩	🟩		
	🟩		🟩	🟩	🟩	🟩	
t = z + y	🟩		🟩	🟩	🟩	🟩	
	🟩		🟩	🟩	🟩		🟩
sum = sum + t	🟩		🟩	🟩	🟩		🟩
	🟩		🟩	🟩	🟩		
i = i + 1	🟩		🟩	🟩	🟩		
	🟩		🟩	🟩	🟩		
return sum				🟩			
				🟩			



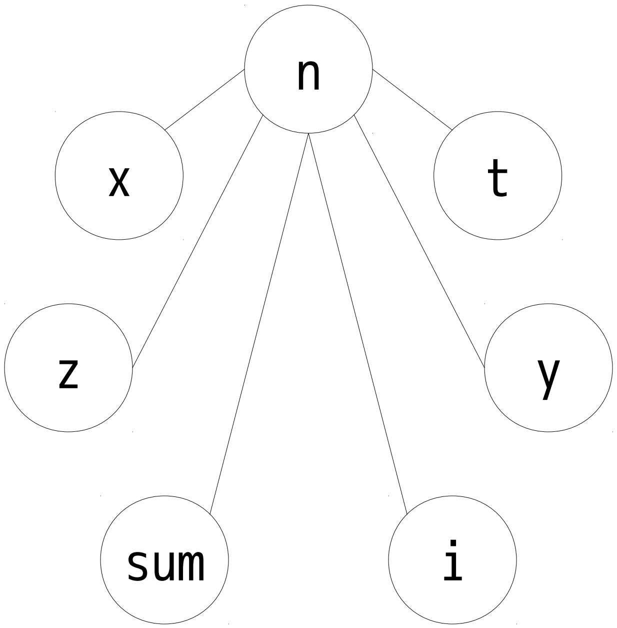
x 跟 n 必須同時存活

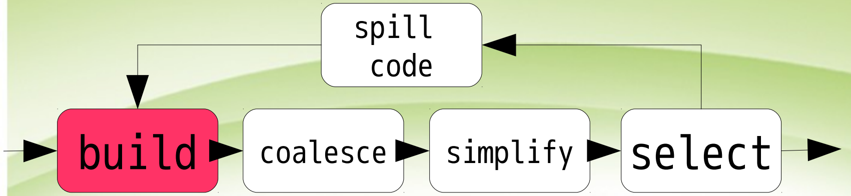
又稱
x 跟 n 與
有 Interference



Interference Graph

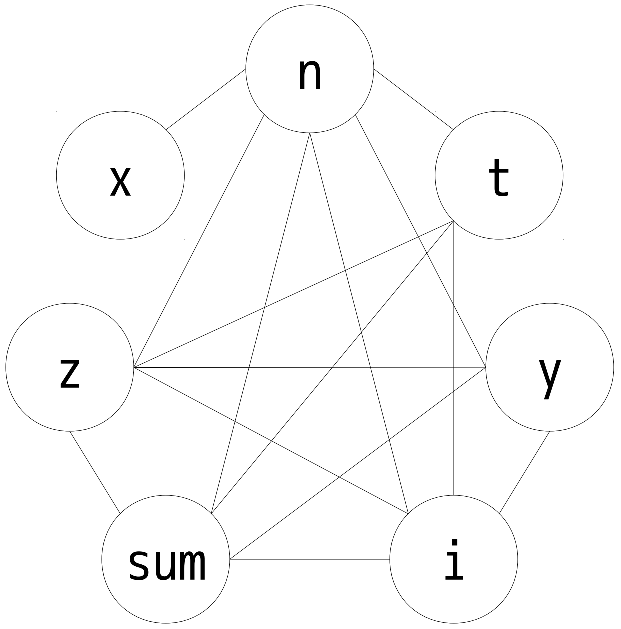
	n	x	z	sum	i	y	t
n = arg (n)							
	🟩						
x = n * 2	🟩						
	🟩	🟩					
z = x	🟩	🟩					
	🟩		🟩				
sum = 0	🟩		🟩				
	🟩		🟩	🟩			
i = 0	🟩		🟩	🟩			
	🟩		🟩	🟩	🟩		
i < n?	🟩		🟩	🟩	🟩		
	🟩		🟩	🟩	🟩		
y = i	🟩		🟩	🟩	🟩		
	🟩		🟩	🟩	🟩	🟩	
t = z + y	🟩		🟩	🟩	🟩	🟩	
	🟩		🟩	🟩	🟩		🟩
sum = sum + t	🟩		🟩	🟩	🟩		🟩
	🟩		🟩	🟩	🟩		
i = i + 1	🟩		🟩	🟩	🟩		
	🟩		🟩	🟩	🟩		
return sum				🟩			
				🟩			

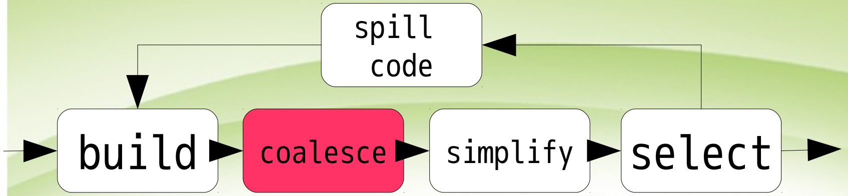




Interference Graph

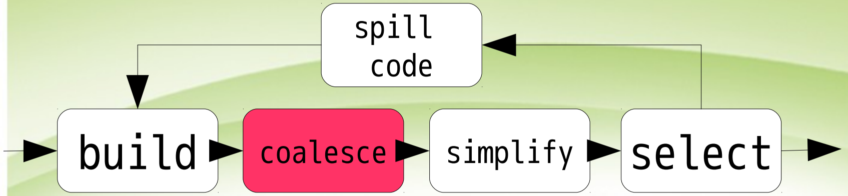
	n	x	z	sum	i	y	t
n = arg (n)							
	🟩						
x = n * 2	🟩						
	🟩	🟩					
z = x	🟩	🟩					
	🟩		🟩				
sum = 0	🟩		🟩				
	🟩		🟩	🟩			
i = 0	🟩		🟩	🟩			
	🟩		🟩	🟩	🟩		
i < n?	🟩		🟩	🟩	🟩		
	🟩		🟩	🟩	🟩		
y = i	🟩		🟩	🟩	🟩		
	🟩		🟩	🟩	🟩	🟩	
t = z + y	🟩		🟩	🟩	🟩	🟩	
	🟩		🟩	🟩	🟩		🟩
sum = sum + t	🟩		🟩	🟩	🟩		🟩
	🟩		🟩	🟩	🟩		
i = i + 1	🟩		🟩	🟩	🟩		
	🟩		🟩	🟩	🟩		
return sum				🟩			
				🟩			





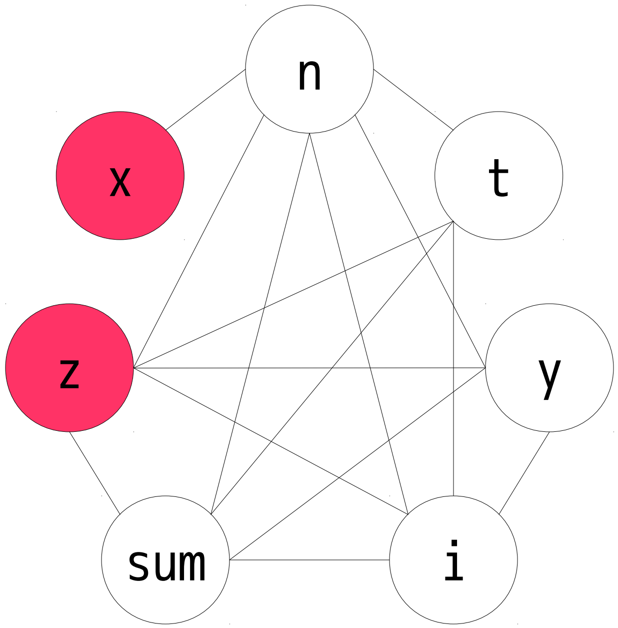
Coalesce

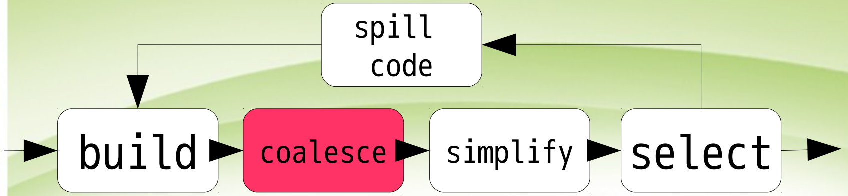
- Coalesce: 合併
- 若一道 move 指令的**來源**與**目的**無互相干擾則可進行 Coalesce, 並將圖上對應的點合併
  - 保證兩者分配到相同 Register
  - move $r0, $r0
    - 明顯可移除的指令



Coalesce

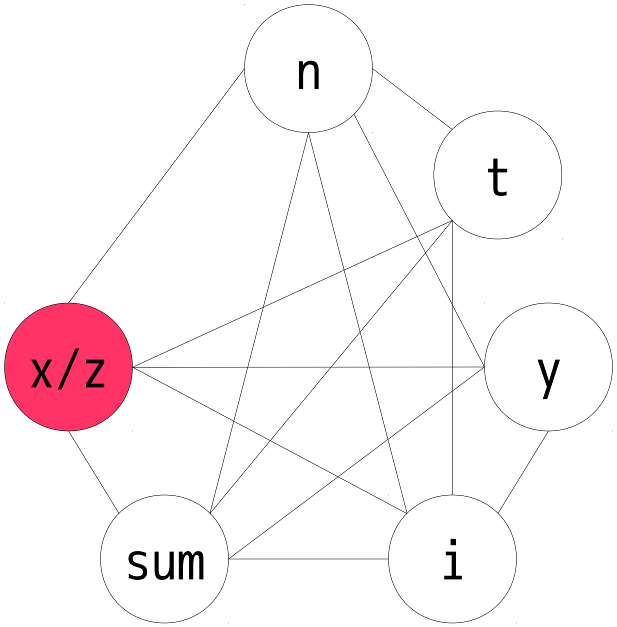
	n	x	z	sum	i	y	t
n = arg (n)							
	🟩						
x = n * 2	🟩						
	🟩	🟩					
z = x	🟩	🟩					
	🟩		🟩				
sum = 0	🟩		🟩				
	🟩		🟩	🟩			
i = 0	🟩		🟩	🟩			
	🟩		🟩	🟩	🟩		
i < n?	🟩		🟩	🟩	🟩		
	🟩		🟩	🟩	🟩		
y = i	🟩		🟩	🟩	🟩		
	🟩		🟩	🟩	🟩	🟩	
t = z + y	🟩		🟩	🟩	🟩	🟩	
	🟩		🟩	🟩	🟩		🟩
sum = sum + t	🟩		🟩	🟩	🟩		🟩
	🟩		🟩	🟩	🟩		
i = i + 1	🟩		🟩	🟩	🟩		
	🟩		🟩	🟩	🟩		
return sum				🟩			
				🟩			

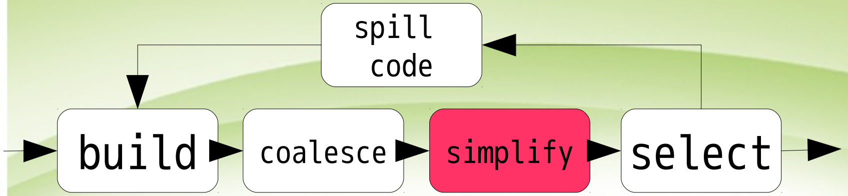




Coalesce

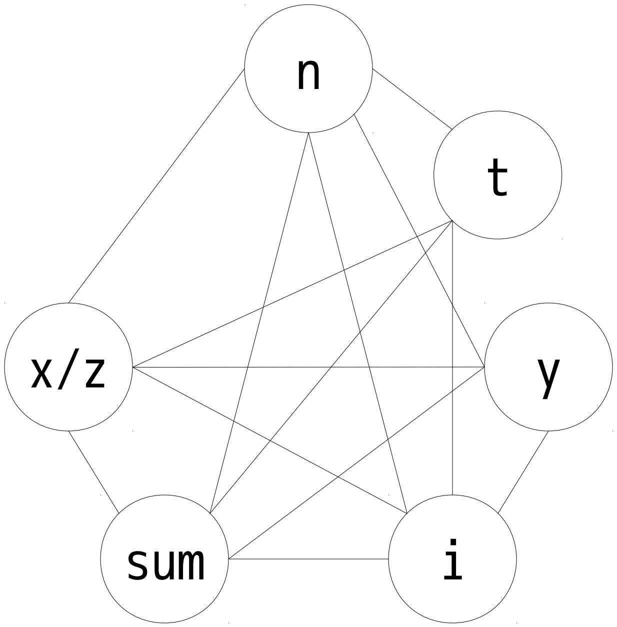
	n	x	z	sum	i	y	t
n = arg (n)							
	🟩						
x = n * 2	🟩						
	🟩	🟩					
z = x	🟩	🟩					
	🟩		🟩				
sum = 0	🟩		🟩				
	🟩		🟩	🟩			
i = 0	🟩		🟩	🟩			
	🟩		🟩	🟩	🟩		
i < n?	🟩		🟩	🟩	🟩		
	🟩		🟩	🟩	🟩		
y = i	🟩		🟩	🟩	🟩		
	🟩		🟩	🟩	🟩	🟩	
t = z + y	🟩		🟩	🟩	🟩	🟩	
	🟩		🟩	🟩	🟩		🟩
sum = sum + t	🟩		🟩	🟩	🟩		🟩
	🟩		🟩	🟩	🟩		
i = i + 1	🟩		🟩	🟩	🟩		
	🟩		🟩	🟩	🟩		
return sum				🟩			
				🟩			

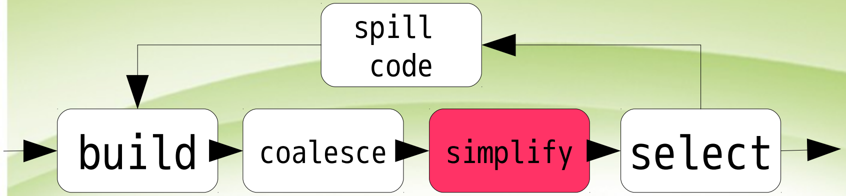




Simplify

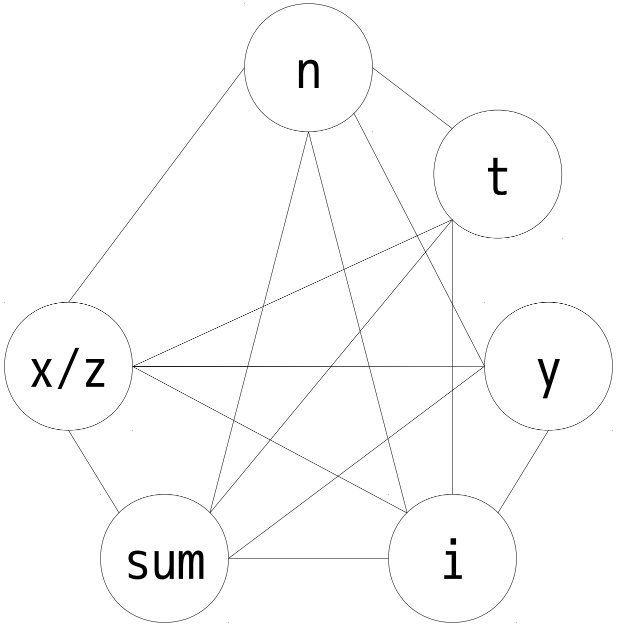
假設有 4 個 Register 可用

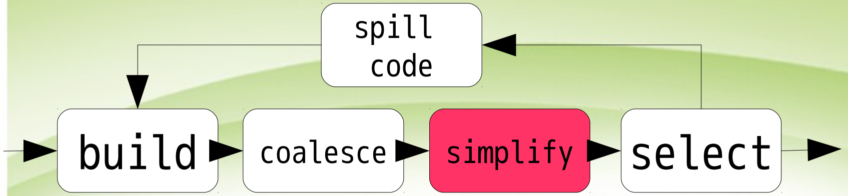




Simplify

假設有 4 個 Register 可用
=> 邊少於 4 都可隨意著色

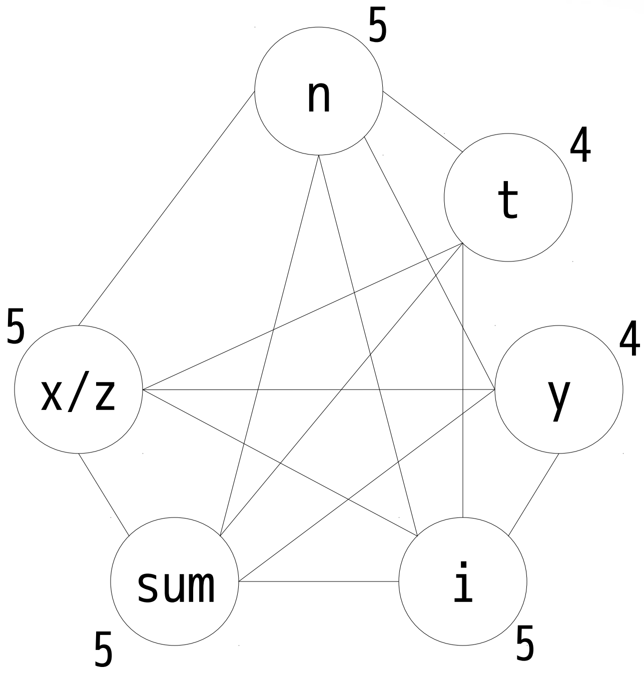


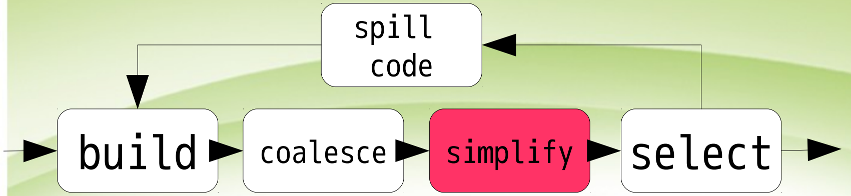


Simplify

假設有 4 個 Register 可用

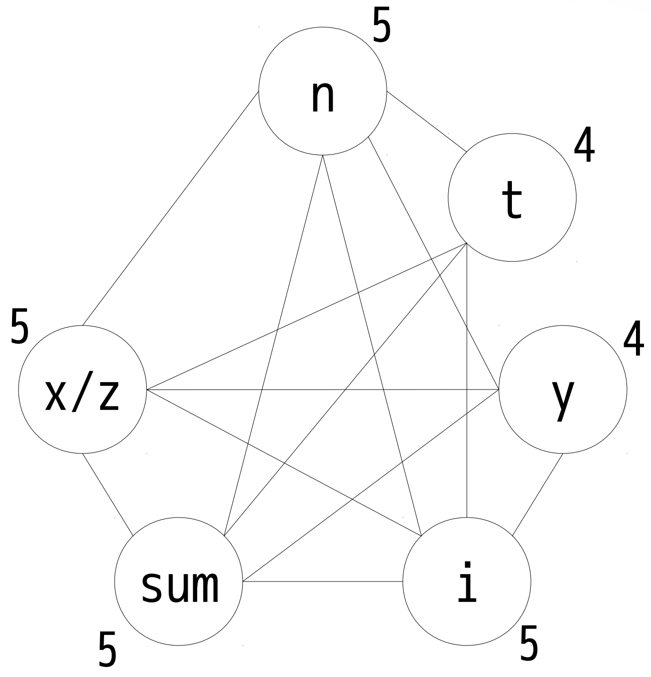
=> 邊少於 4 都可隨意著色



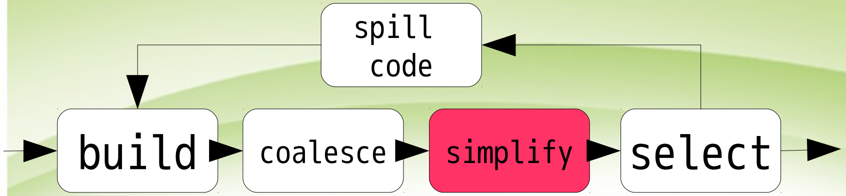


Simplify

假設有 4 個　Register　可用
=>　邊少於　4　都可隨意著色



發現沒邊少於 4 的點！



spill code

build

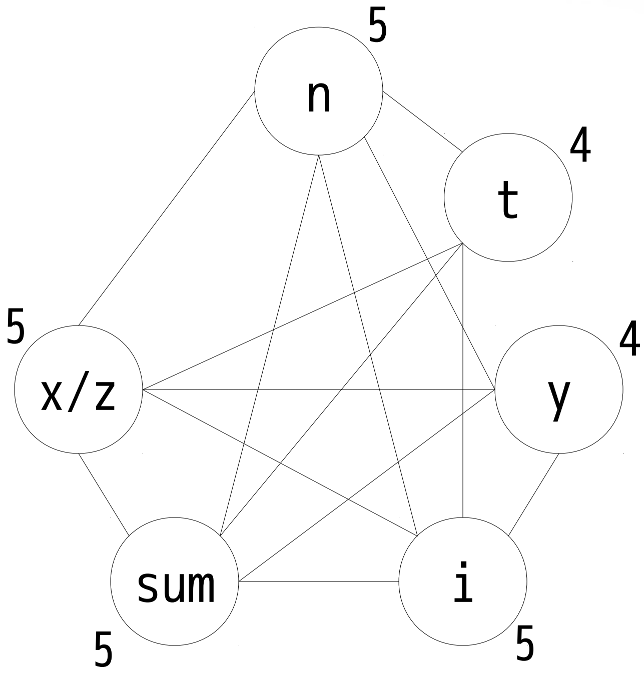
coalesce

simplify

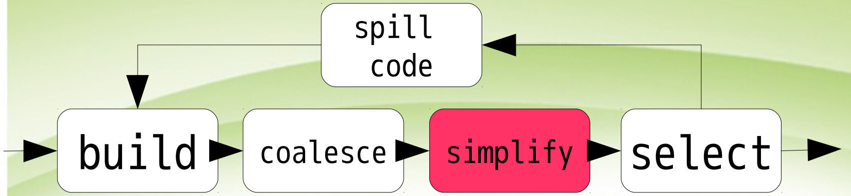
select

Simplify

假設有 4 個 Register 可用
=> 邊少於 4 都可隨意著色



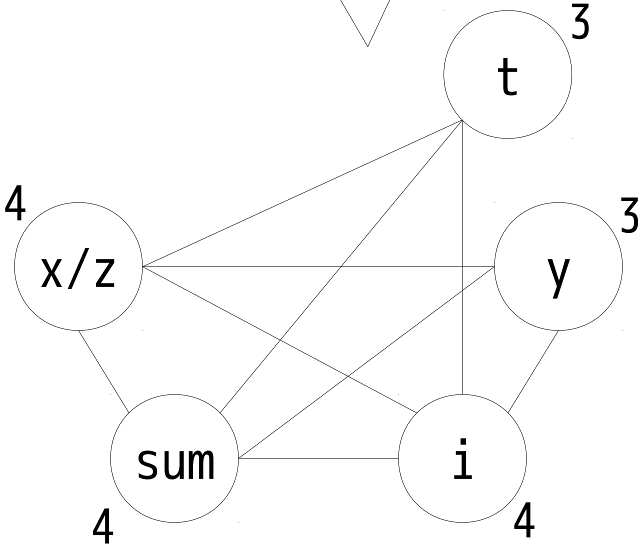
發現沒邊少於 4 的點！
=> 挑一個 Cost 最低的拿

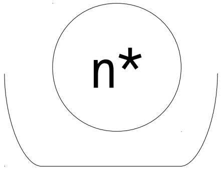


Simplify

假設有 4 個 Register 可用
=> 邊少於 4 都可隨意著色

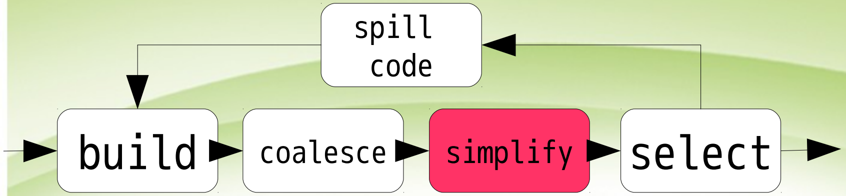
更新每個點的邊數





拿到旁邊的 Stack
並標上記號

發現沒邊少於 4 的點！
=> 挑一個 Cost 最低的拿



spill code

build

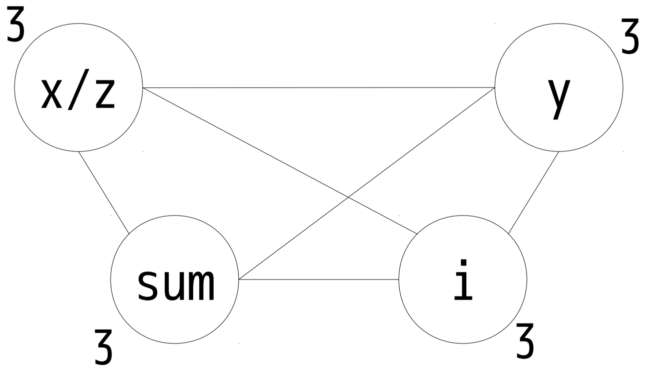
coalesce

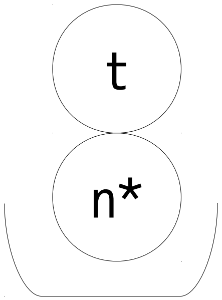
simplify

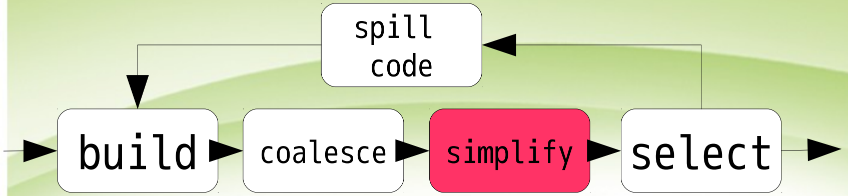
select

Simplify

假設有 4 個 Register 可用
=> 邊少於 4 都可隨意著色



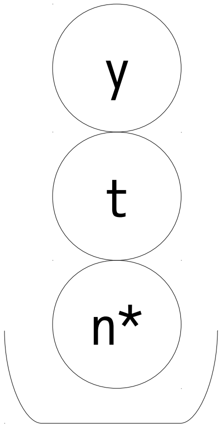


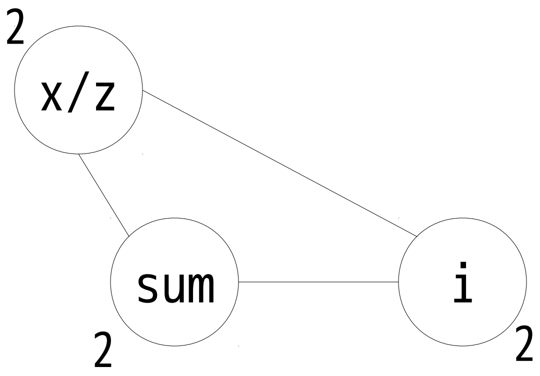


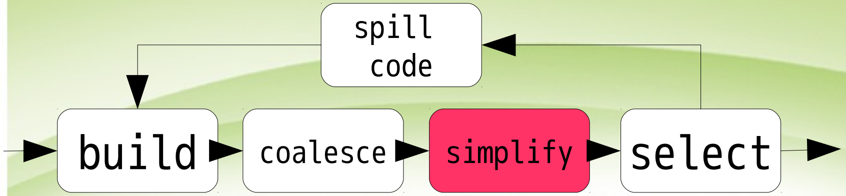
Simplify

假設有 4 個 Register 可用

=> 邊少於 4 都可隨意著色

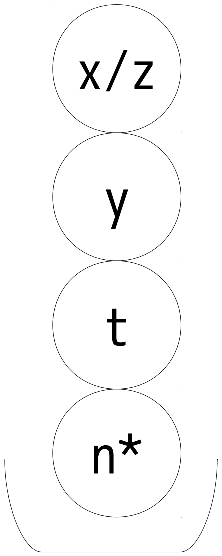


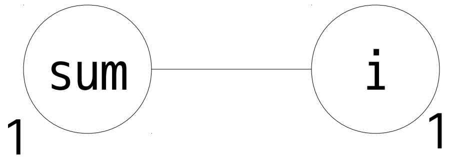


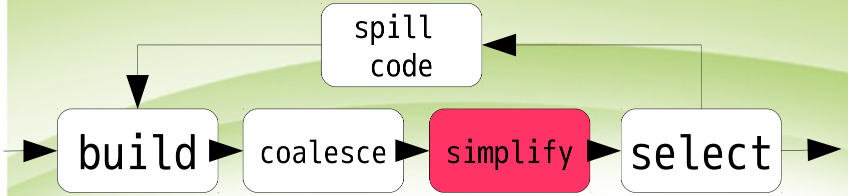


Simplify

假設有 4 個 Register 可用
=> 邊少於 4 都可隨意著色



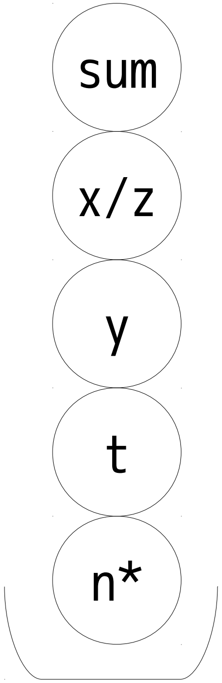


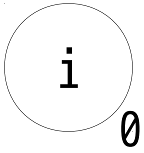


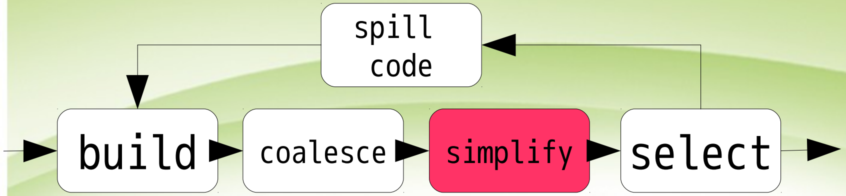
Simplify

假設有 4 個 Register 可用

=> 邊少於 4 都可隨意著色



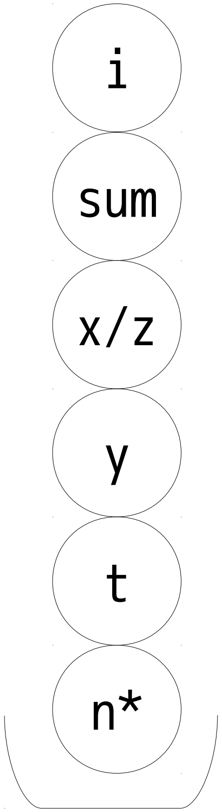


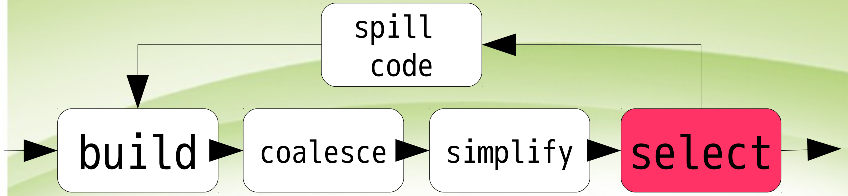


Simplify

假設有 4 個 Register 可用

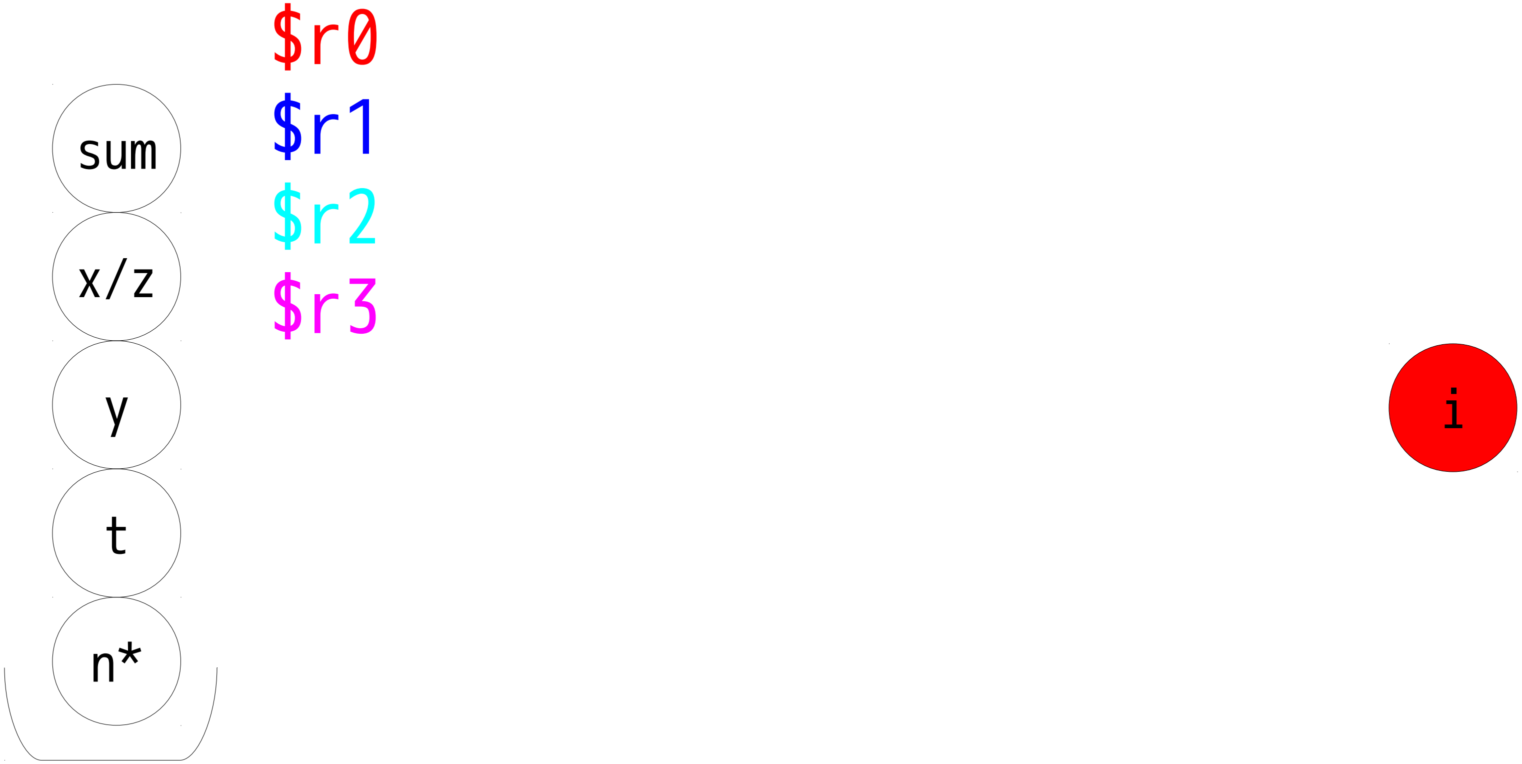
=> 邊少於 4 都可隨意著色

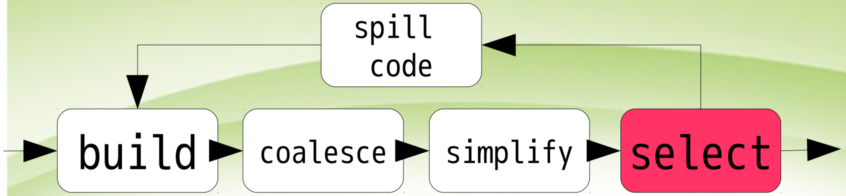




Select

進入著色階段， 從 Stack 中 Pop 出來並著上隨意顏色





Select

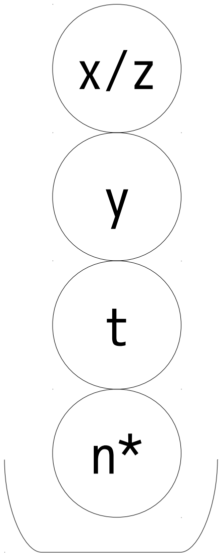
進入著色階段， 從 Stack 中 Pop 出來並著上隨意顏色

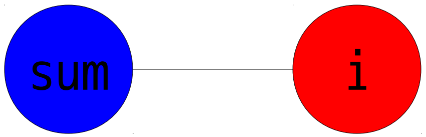
$r0

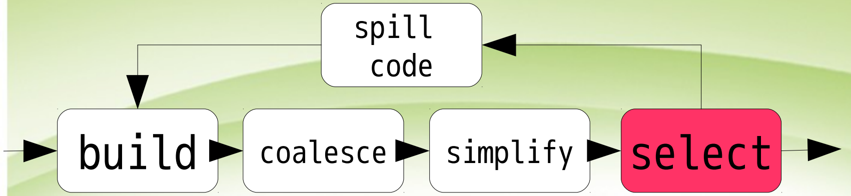
$r1

$r2

$r3







spill code

build

coalesce

simplify

select

Select

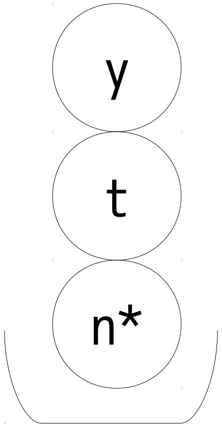
進入著色階段， 從 Stack 中 Pop 出來並著上隨意顏色

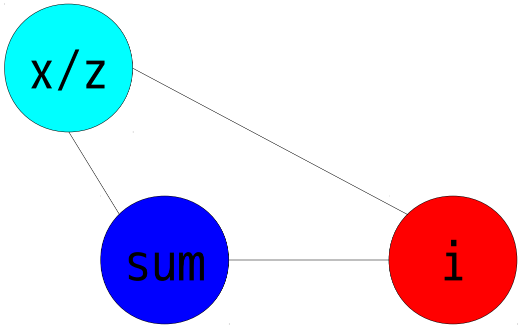
$r0

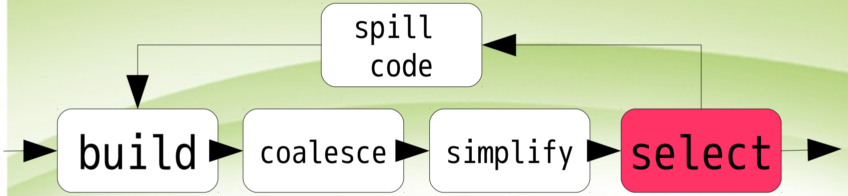
$r1

$r2

$r3






spill code

build

coalesce

simplify

select

Select

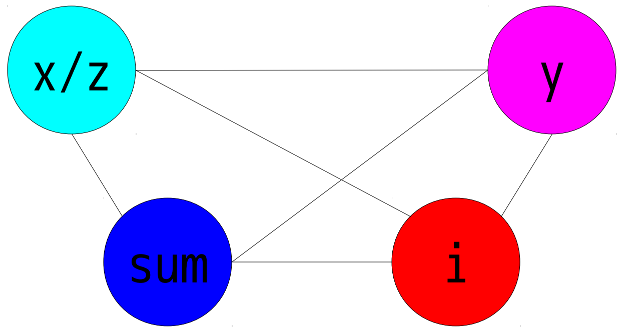
進入著色階段， 從 Stack 中 Pop 出來並著上隨意顏色

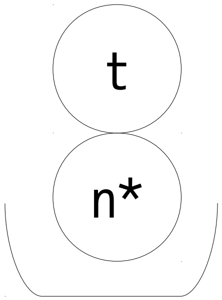
$r0

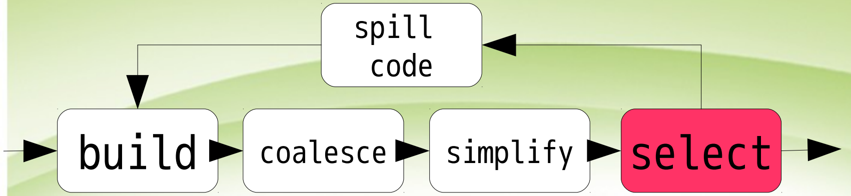
$r1

$r2

$r3



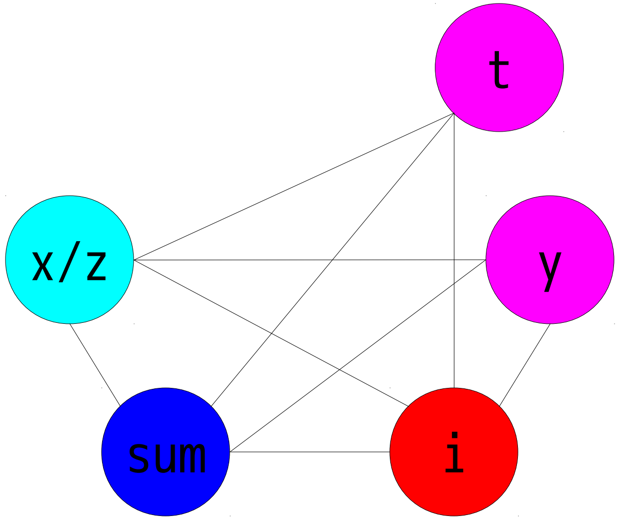


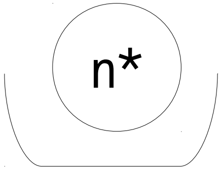


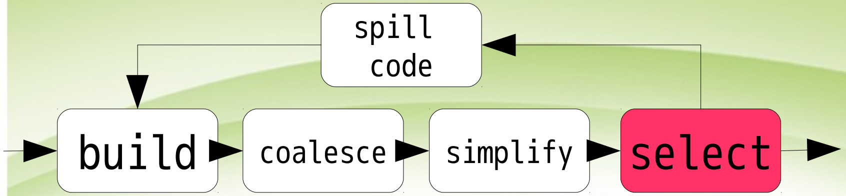
Select

進入著色階段， 從 Stack 中 Pop
出來並著上隨意顏色

$r0
$r1
$r2
$r3



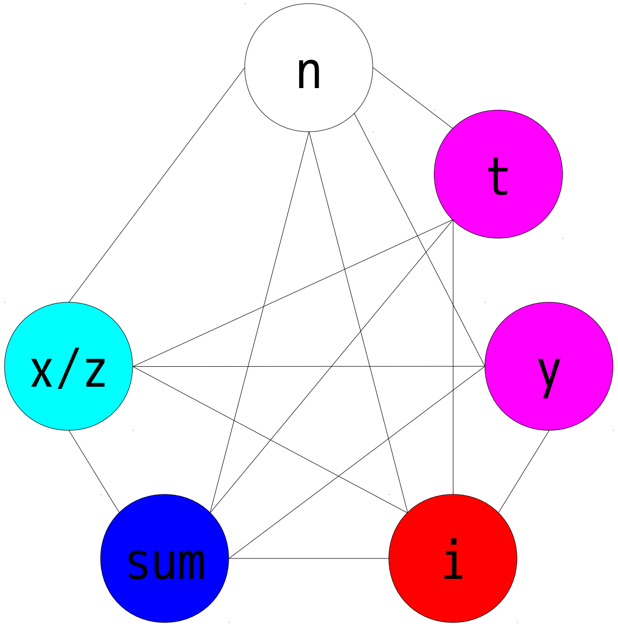


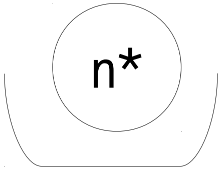


Select

進入著色階段， 從 Stack 中 Pop 出來並著上隨意顏色

$r0
$r1
$r2
$r3



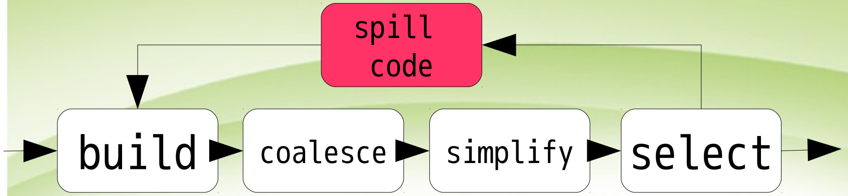


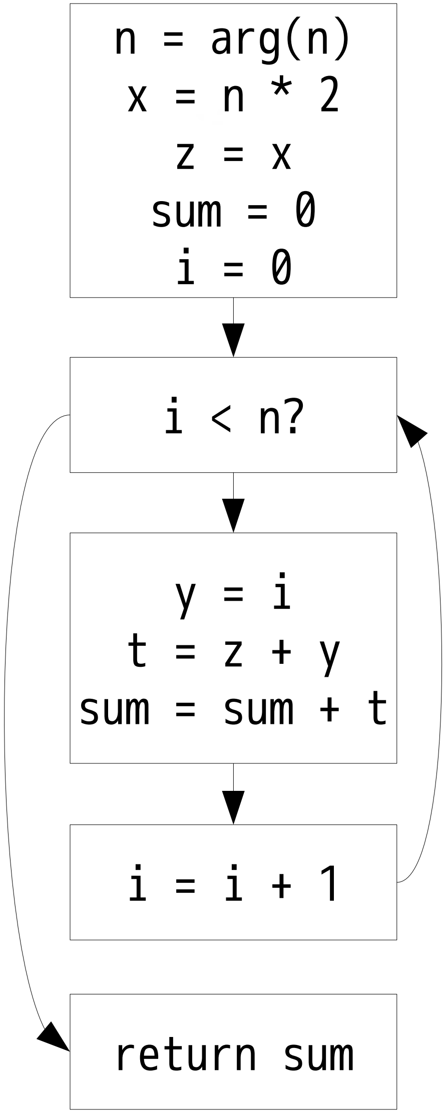
發現 ***n*** 得不到任何可用顏色！
進入 Spill 階段

Spill

- 當 Register 不足時， 則必須將值暫存到記憶體中， 必要時再從記憶體中取回

Spill





```
n = arg(n)
x = n * 2
z = x
sum = 0
i = 0
```

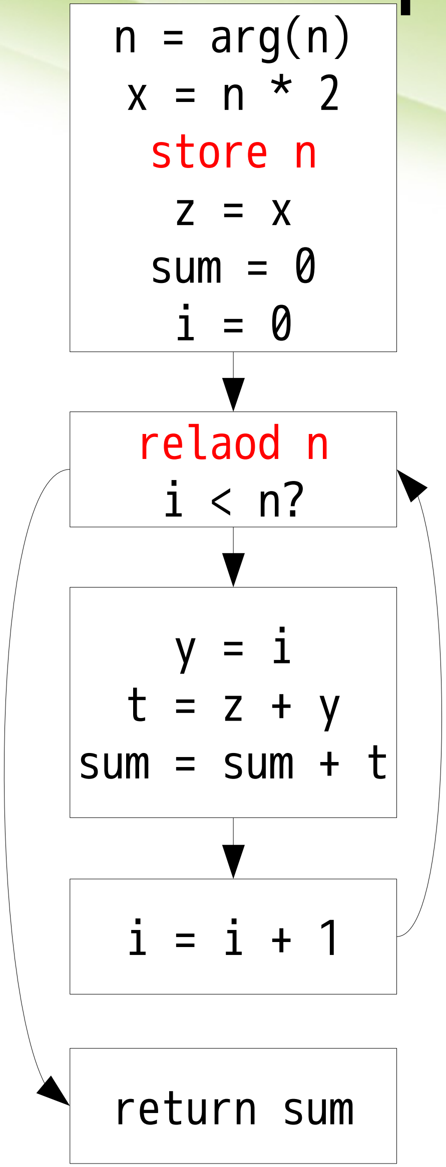
```
i < n?
```

```
y = i
t = z + y
sum = sum + t
```

```
i = i + 1
```

```
return sum
```

插入
Spill Code



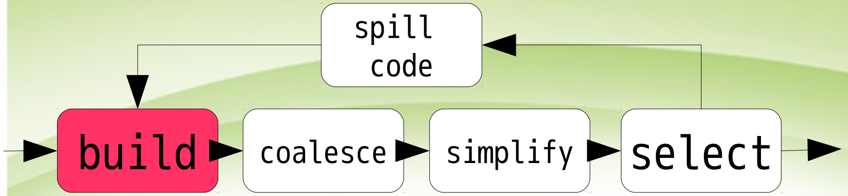
```
n = arg(n)
x = n * 2
store n
z = x
sum = 0
i = 0
```

```
relaod n
i < n?
```

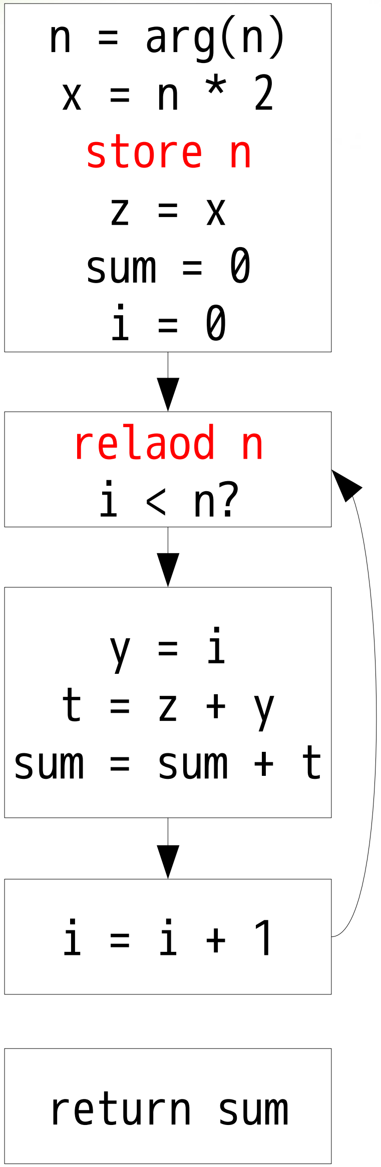
```
y = i
t = z + y
sum = sum + t
```

```
i = i + 1
```

```
return sum
```

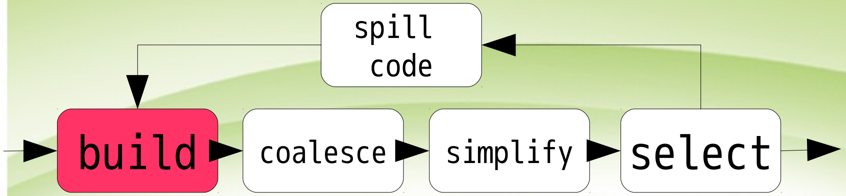


Live Range

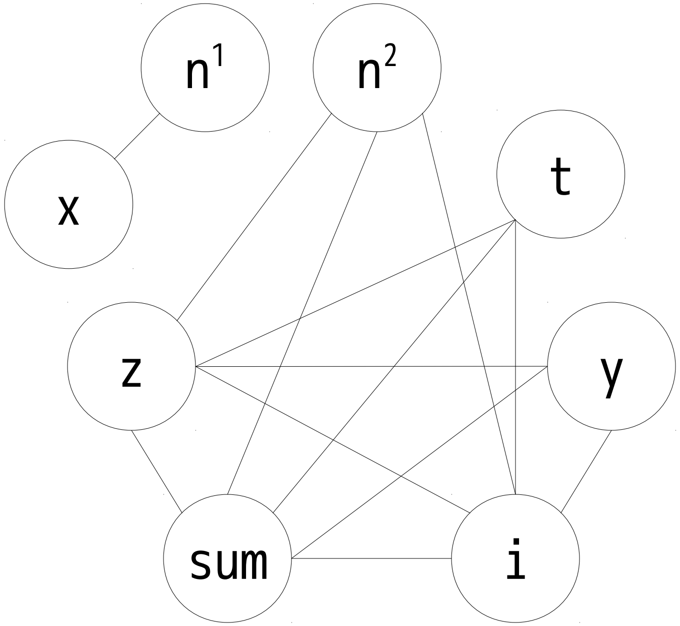


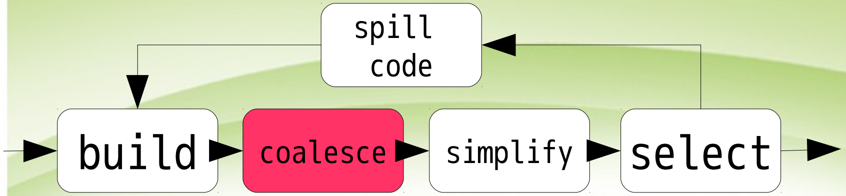
	n	x	z	sum	i	y	t
n = arg (n)							
	🟩						
x = n * 2	🟩						
	🟩 n^1	🟩					
store n	🟩	🟩					
		🟩					
z = x		🟩					
			🟩				
sum = 0			🟩				
			🟩	🟩			
i = 0			🟩	🟩			
			🟩	🟩	🟩		
reload n			🟩	🟩	🟩		
	🟩		🟩	🟩	🟩		
i < n?	🟩 n^2		🟩	🟩	🟩		
			🟩	🟩	🟩		
y = i			🟩	🟩	🟩		
			🟩	🟩	🟩	🟩	
t = z + y			🟩	🟩	🟩	🟩	
			🟩	🟩	🟩		🟩
sum = sum + t			🟩	🟩	🟩		🟩
			🟩	🟩	🟩		
i = i +1			🟩	🟩	🟩		
			🟩	🟩	🟩		
return sum				🟩			

Interference Graph



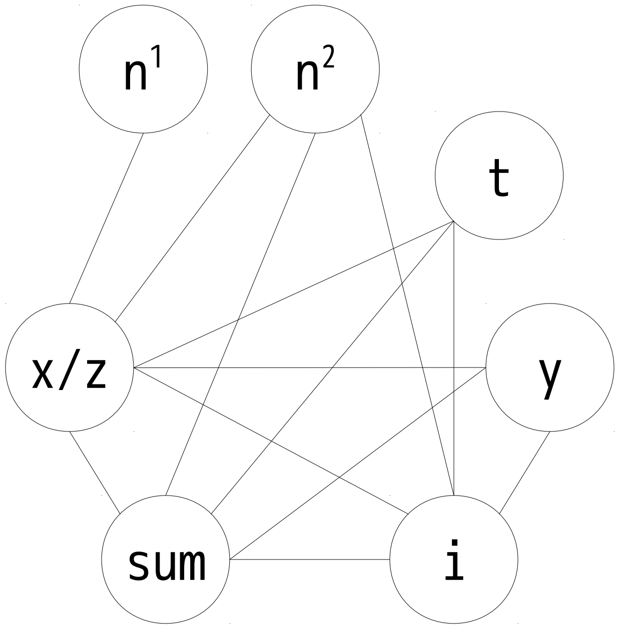
	n	x	z	sum	i	y	t
n = arg (n)							
	🟩						
x = n * 2	🟩						
	🟩 n^1	🟩					
store n	🟩	🟩					
		🟩					
z = x		🟩					
			🟩				
sum = 0			🟩				
			🟩	🟩			
i = 0			🟩	🟩			
			🟩	🟩	🟩		
reload n			🟩	🟩	🟩		
	🟩		🟩	🟩	🟩		
i < n?	🟩 n^2		🟩	🟩	🟩		
			🟩	🟩	🟩		
y = i			🟩	🟩	🟩		
			🟩	🟩	🟩	🟩	
t = z + y			🟩	🟩	🟩	🟩	
			🟩	🟩	🟩		🟩
sum = sum + t			🟩	🟩	🟩		🟩
			🟩	🟩	🟩		
i = i +1			🟩	🟩	🟩		
			🟩	🟩	🟩		
return sum				🟩			



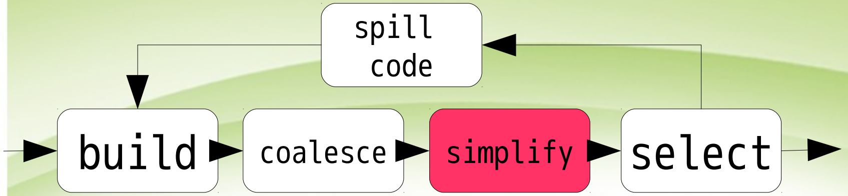


Coalesce

	n	x	z	sum	i	y	t
n = arg (n)							
	🟩						
x = n * 2	🟩 n^1						
	🟩	🟩					
store n	🟩	🟩					
		🟩					
z = x		🟩					
			🟩				
sum = 0			🟩				
			🟩	🟩			
i = 0			🟩	🟩			
			🟩	🟩	🟩		
reload n			🟩	🟩	🟩		
	🟩 n^2		🟩	🟩	🟩		
i < n?	🟩		🟩	🟩	🟩		
			🟩	🟩	🟩		
y = i			🟩	🟩	🟩		
			🟩	🟩	🟩	🟩	
t = z + y			🟩	🟩	🟩	🟩	
			🟩	🟩	🟩		🟩
sum = sum + t			🟩	🟩	🟩		🟩
			🟩	🟩	🟩		
i = i +1			🟩	🟩	🟩		
			🟩	🟩	🟩		
return sum				🟩			



Simplify

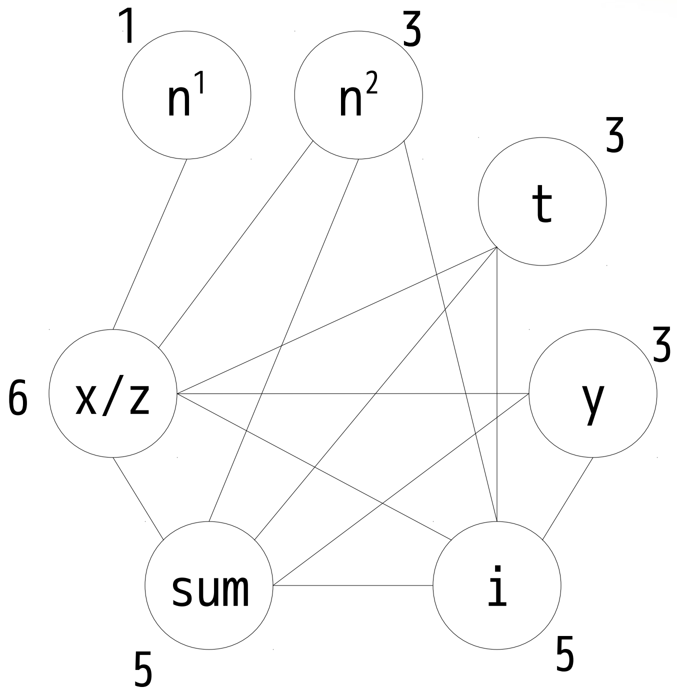

spill code

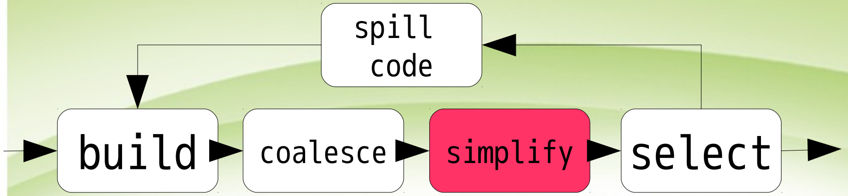
build

coalesce

simplify

select




spill code

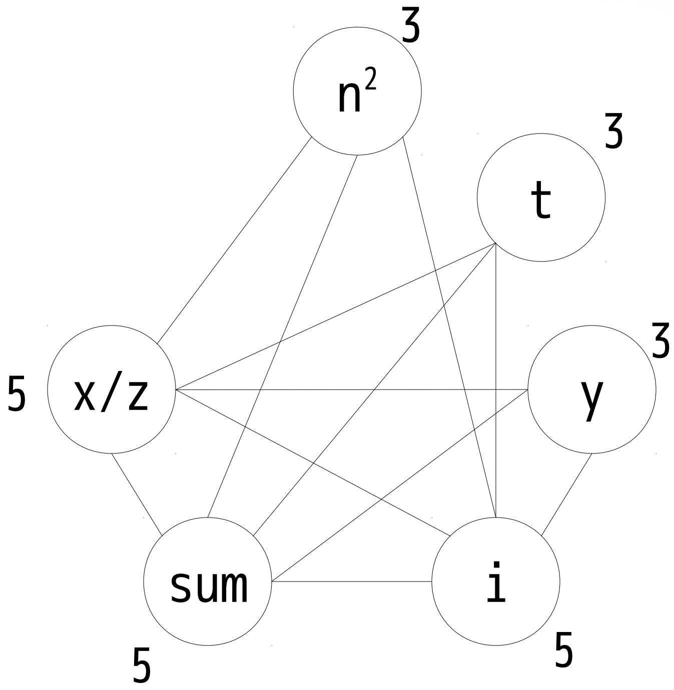
build

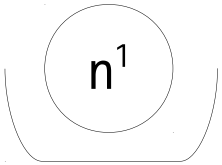
coalesce

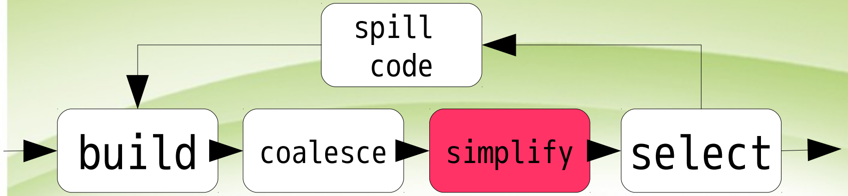
simplify

select

Simplify

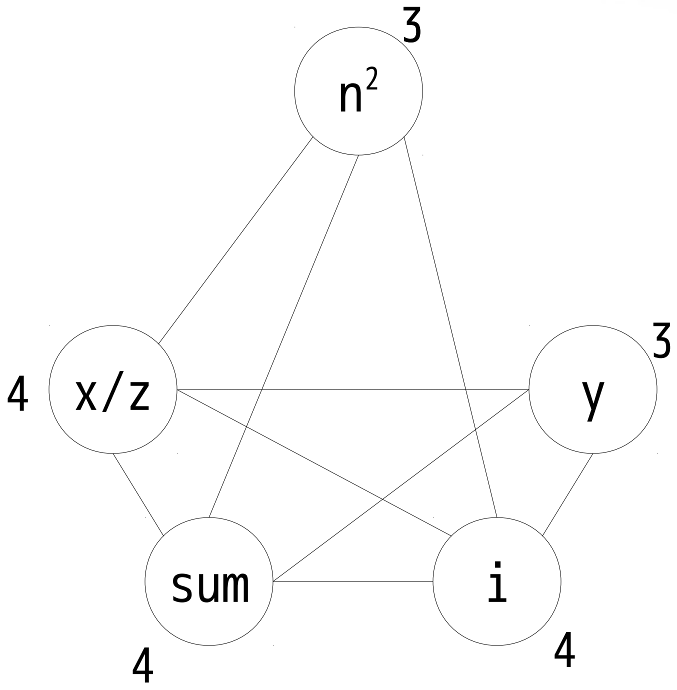




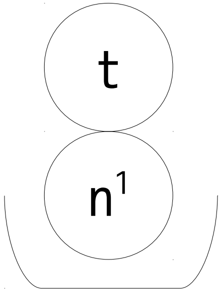


spill code
build
coalesce
simplify
select

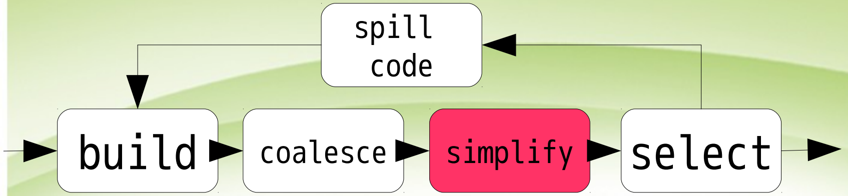
Simplify



n2
3
x/z
4
y
3
sum
4
i
4



t
n1



spill code

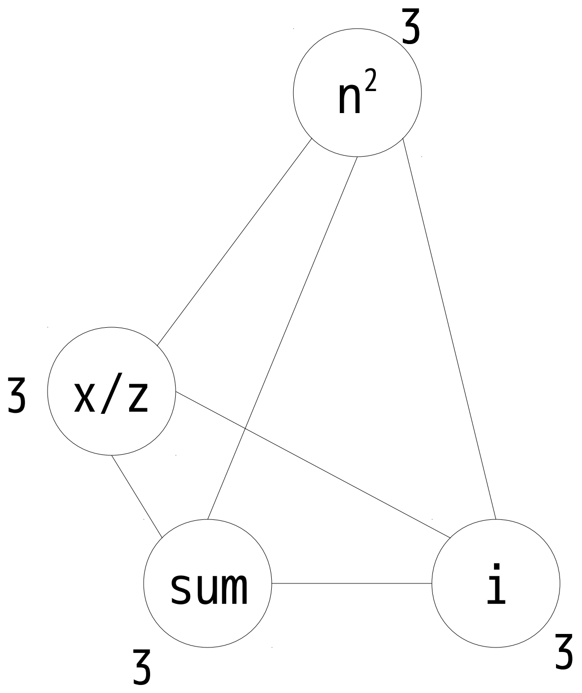
build

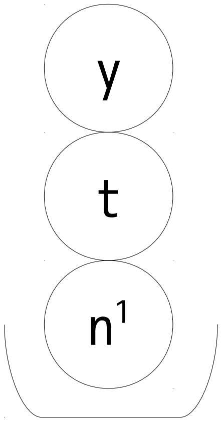
coalesce

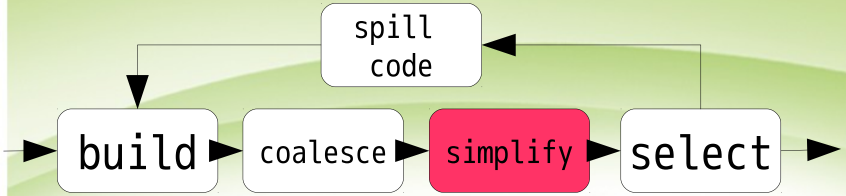
simplify

select

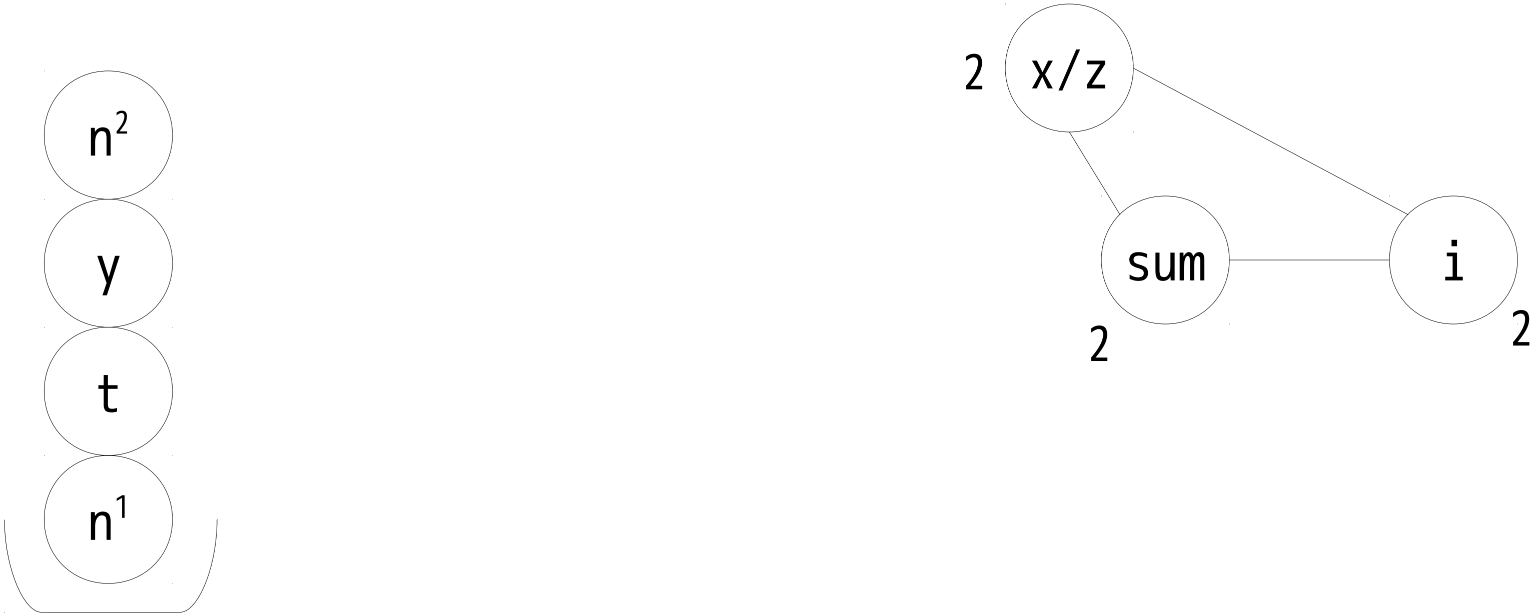
Simplify

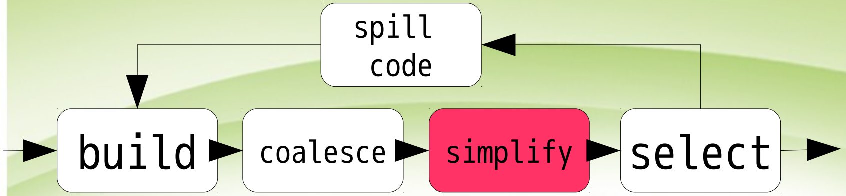





spill code
build
coalesce
simplify
select

Simplify


n²
y
t
n¹
2
x/z
sum
2
i
2



spill code

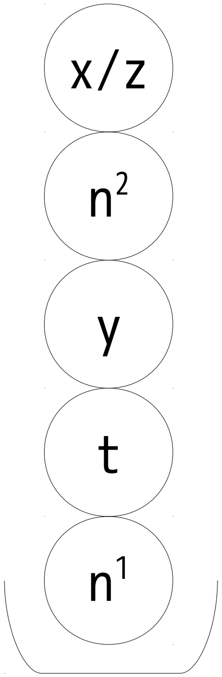
build

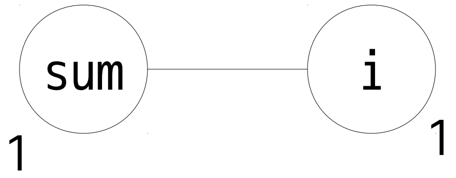
coalesce

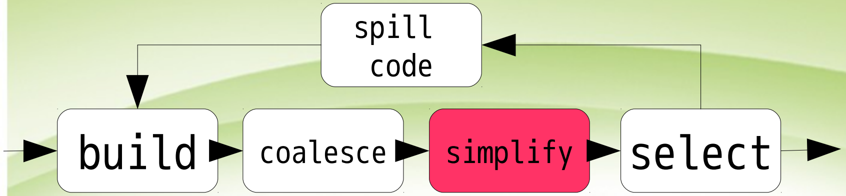
simplify

select

Simplify






spill code

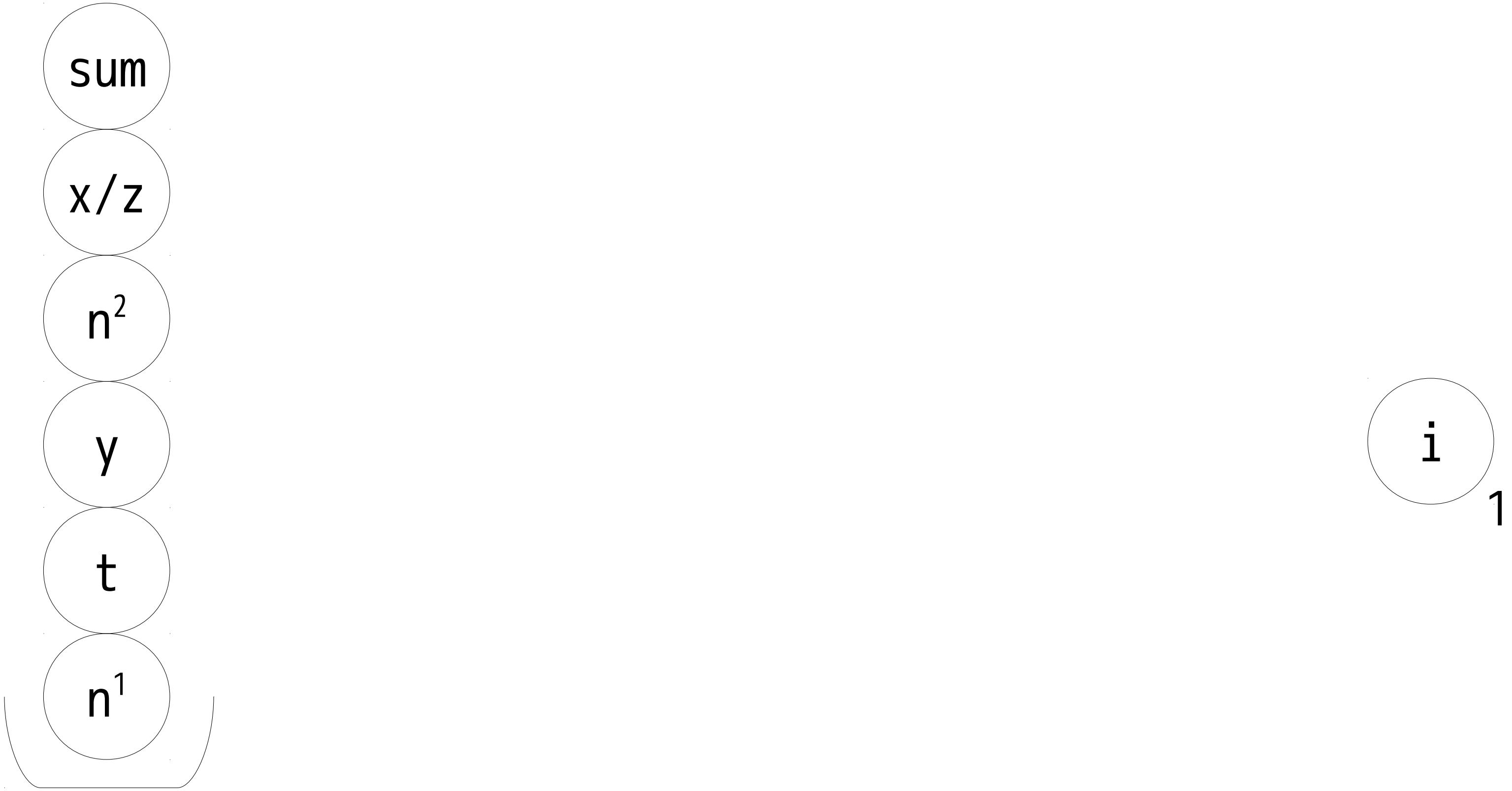
build

coalesce

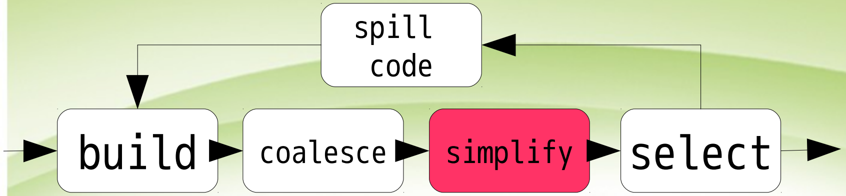
simplify

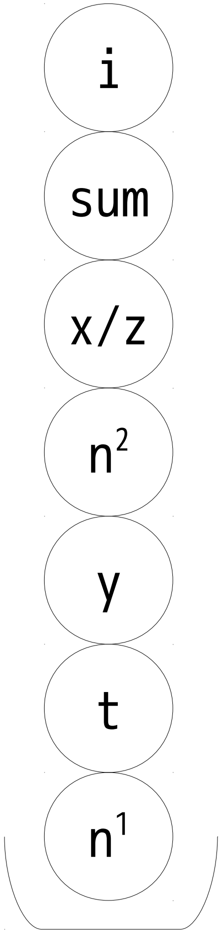
select

Simplify

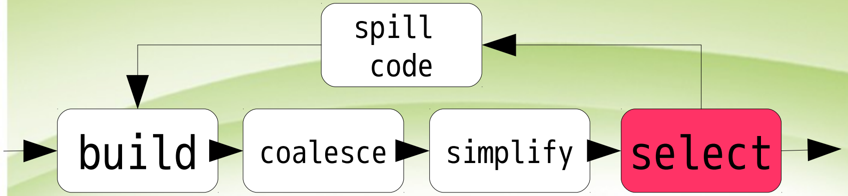


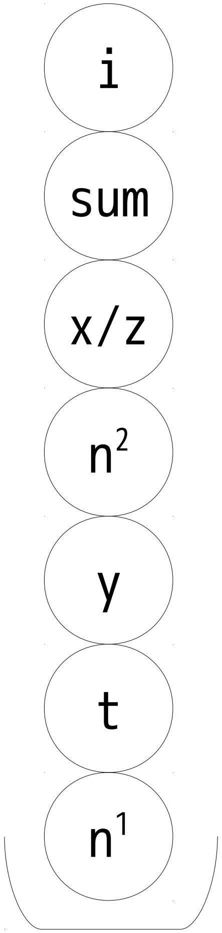
Simplify





Select



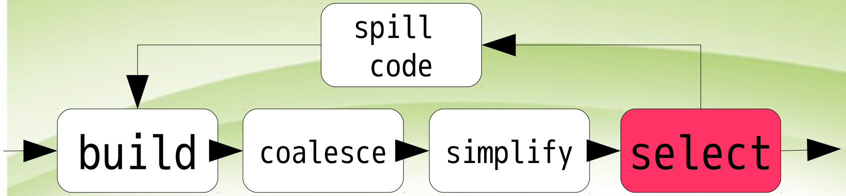


$r0

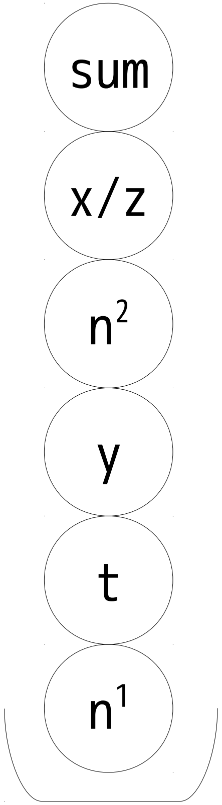
$r1

$r2

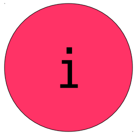
$r3

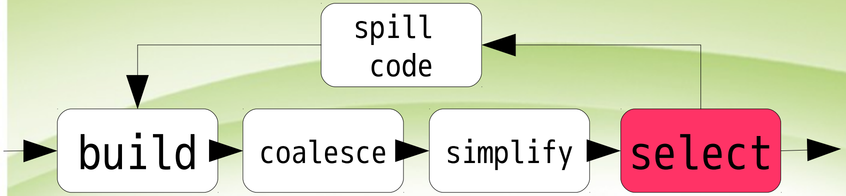


Select

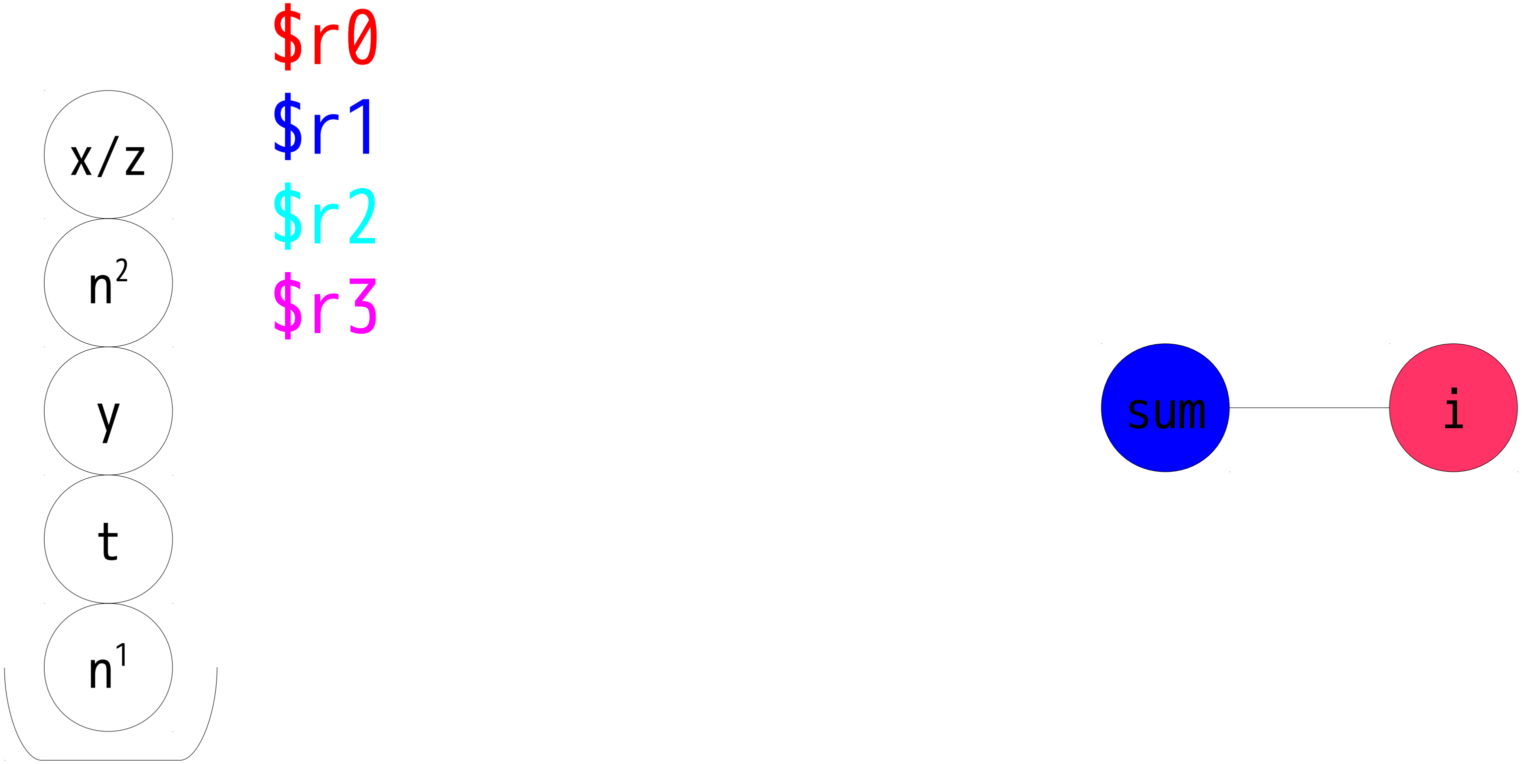


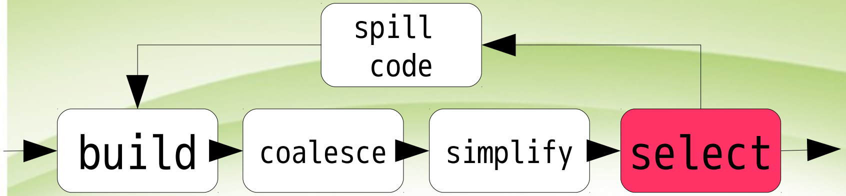
$r0
$r1
$r2
$r3




spill code
build
coalesce
simplify
select

Select


$r0
$r1
$r2
$r3
x/z
n2
y
t
n1
sum
i



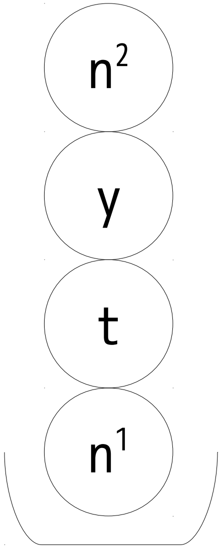
Select

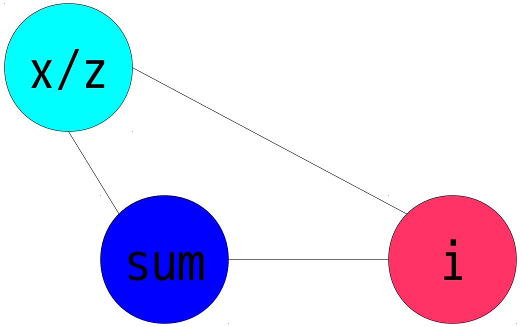
$r0

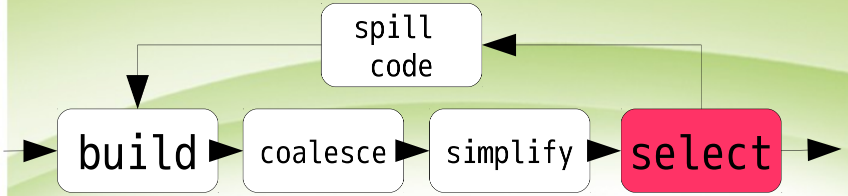
$r1

$r2

$r3

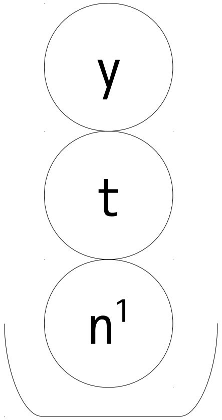


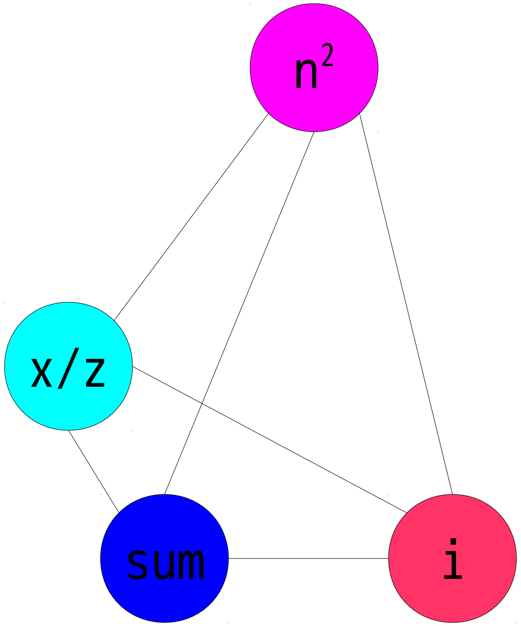


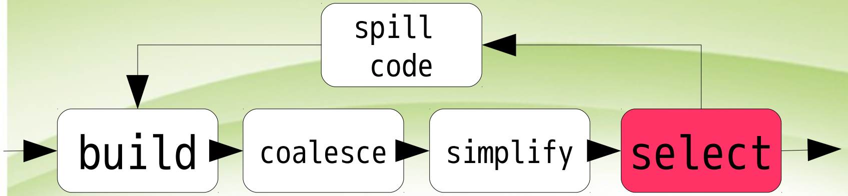


Select

$r0
$r1
$r2
$r3







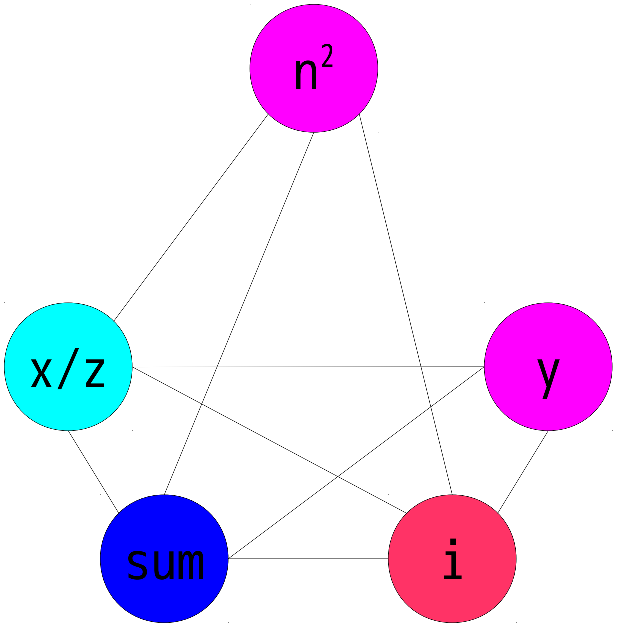
Select

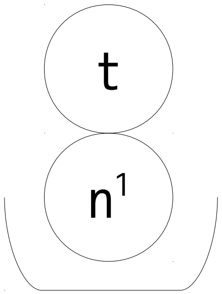
$r0

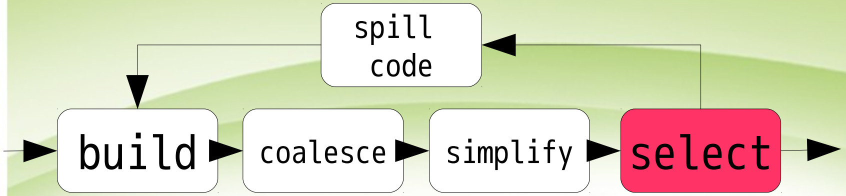
$r1

$r2

$r3

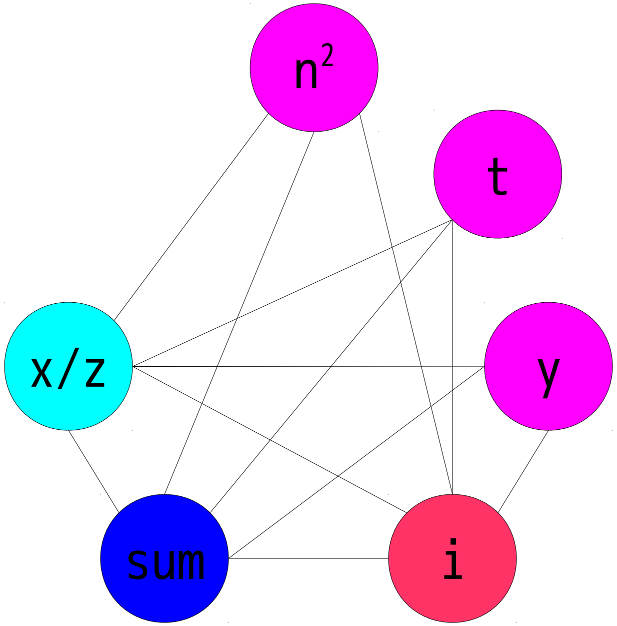


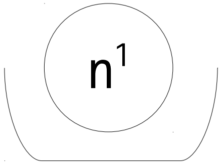


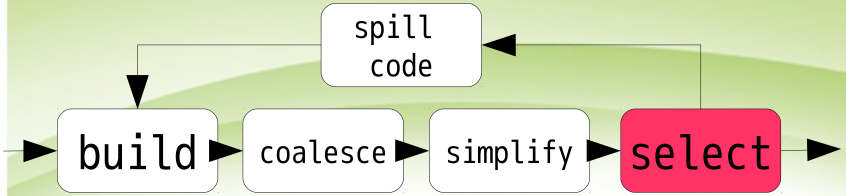


Select

$r0
$r1
$r2
$r3

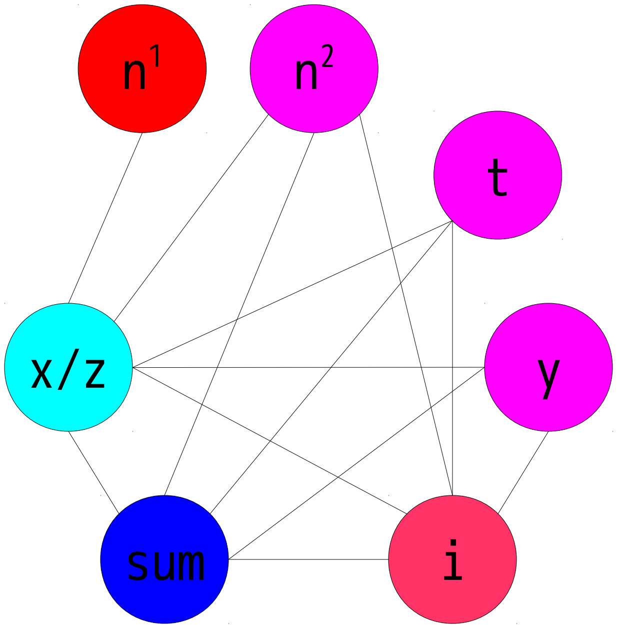


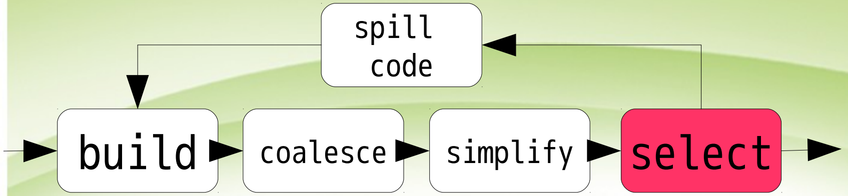




Select

$r0
$r1
$r2
$r3





Select

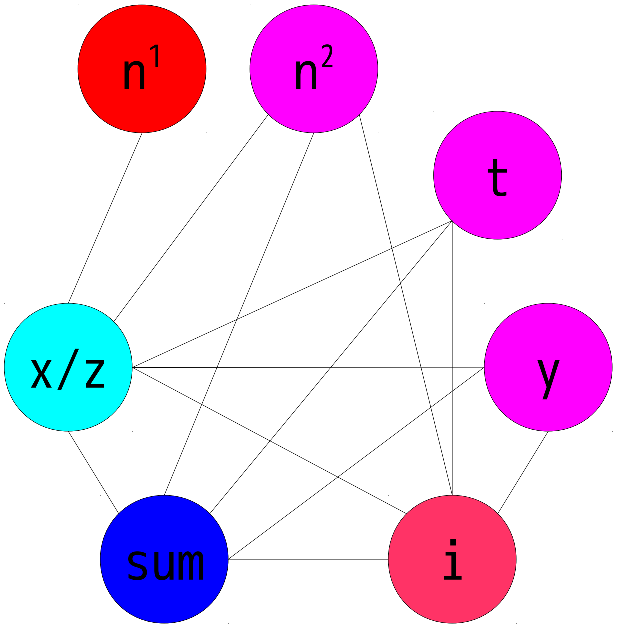
\$r0

\$r1

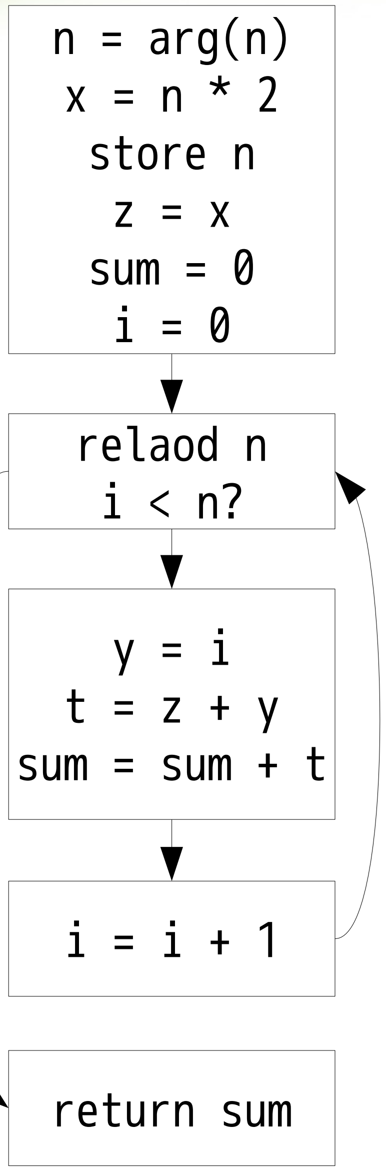
\$r2

\$r3

$n^1 = \$r0$	sum = \$r1
$n^2 = \$r3$	t = \$r3
x = \$r2	y = \$r3
z = \$r2	i = \$r0



Code Gen



```
n = arg(n)
x = n * 2
store n
z = x
sum = 0
i = 0
```

```
relaod n
i < n?
```

```
y = i
t = z + y
sum = sum + t
```

```
i = i + 1
```

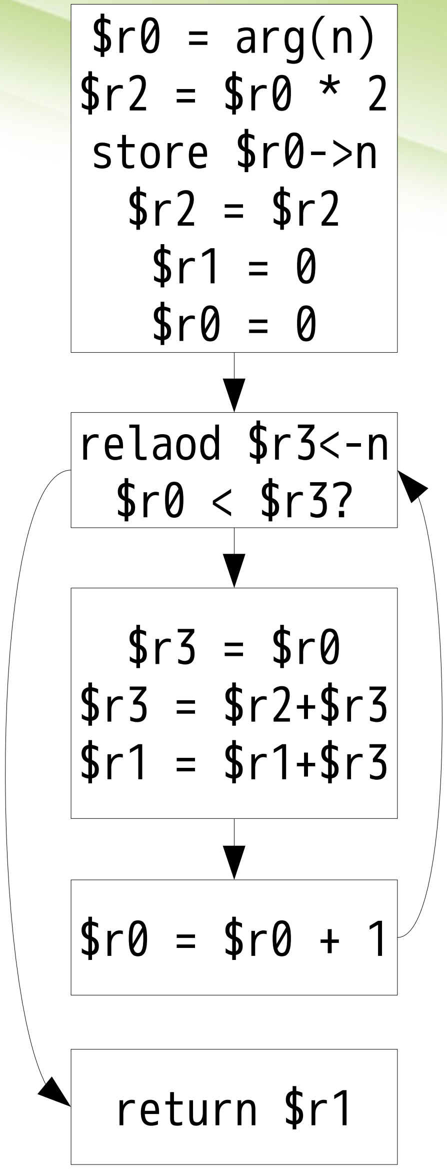
```
return sum
```

n^1 = \$r0   sum = \$r1

n^2 = \$r3   t = \$r3

x = \$r2   y = \$r3

z = \$r2   i = \$r0



```
$r0 = arg(n)
$r2 = $r0 * 2
store $r0->n
$r2 = $r2
$r1 = 0
$r0 = 0
```

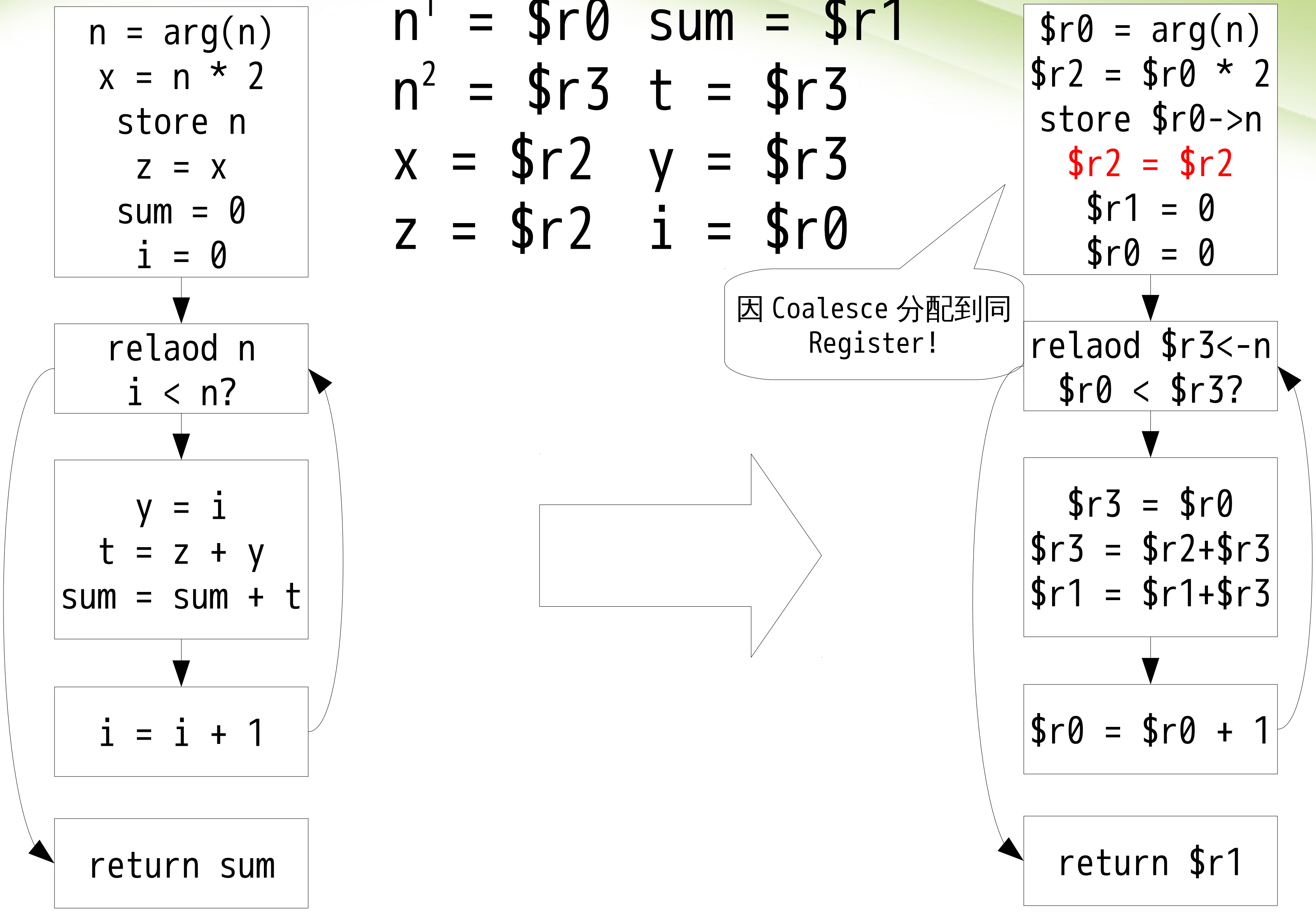
```
relaod $r3<-n
$r0 < $r3?
```

```
$r3 = $r0
$r3 = $r2+$r3
$r1 = $r1+$r3
```

```
$r0 = $r0 + 1
```

```
return $r1
```

Code Gen



```
n = arg(n)
x = n * 2
store n
z = x
sum = 0
i = 0
```

```
relaod n
i < n?
```

```
y = i
t = z + y
sum = sum + t
```

```
i = i + 1
```

```
return sum
```

n^1	= \$r0	sum	= \$r1
n^2	= \$r3	t	= \$r3
x	= \$r2	y	= \$r3
z	= \$r2	i	= \$r0

因 Coalesce 分配到同 Register!

```
$r0 = arg(n)
$r2 = $r0 * 2
store $r0->n
$r2 = $r2
$r1 = 0
$r0 = 0
```

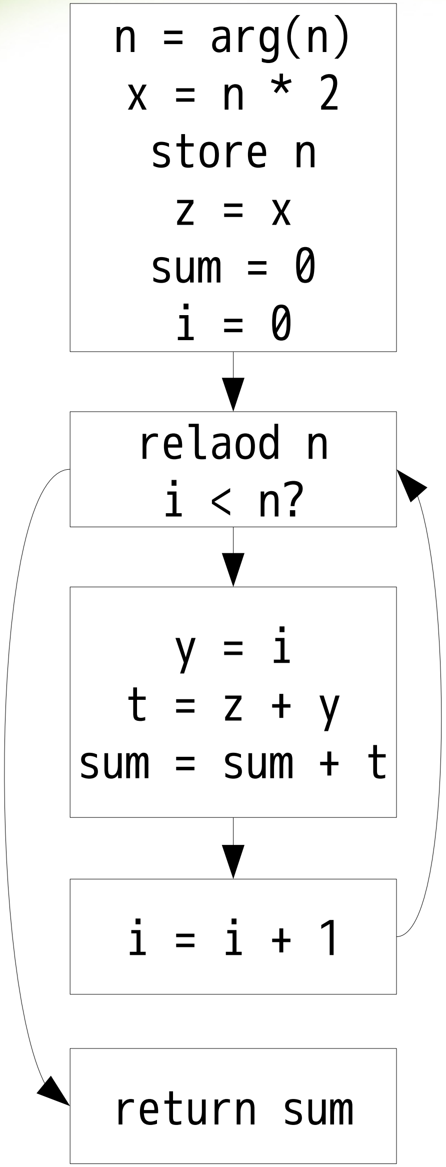
```
relaod $r3<-n
$r0 < $r3?
```

```
$r3 = $r0
$r3 = $r2+$r3
$r1 = $r1+$r3
```

```
$r0 = $r0 + 1
```

```
return $r1
```

Code Gen



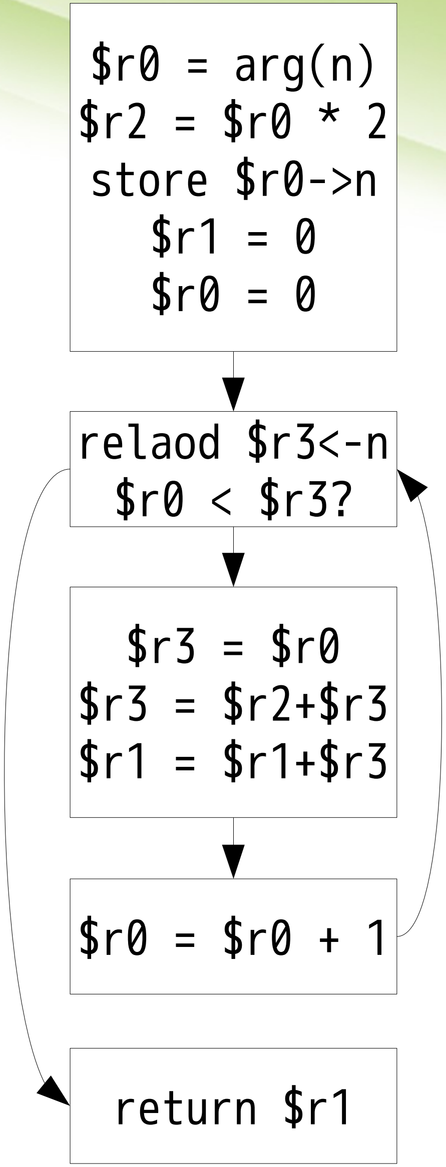
```
n = arg(n)
x = n * 2
store n
z = x
sum = 0
i = 0

relaod n
i < n?

y = i
t = z + y
sum = sum + t

i = i + 1

return sum
```

$n^1 = \$r0$ sum = \$r1

$n^2 = \$r3$ t = \$r3

x = \$r2 y = \$r3

z = \$r2 i = \$r0



```
$r0 = arg(n)
$r2 = $r0 * 2
store $r0->n
$r1 = 0
$r0 = 0

relaod $r3<-n
$r0 < $r3?

$r3 = $r0
$r3 = $r2+$r3
$r1 = $r1+$r3

$r0 = $r0 + 1

return $r1
```

Rematerialization

- Materialization: 實現化、具體化

Rematerialization

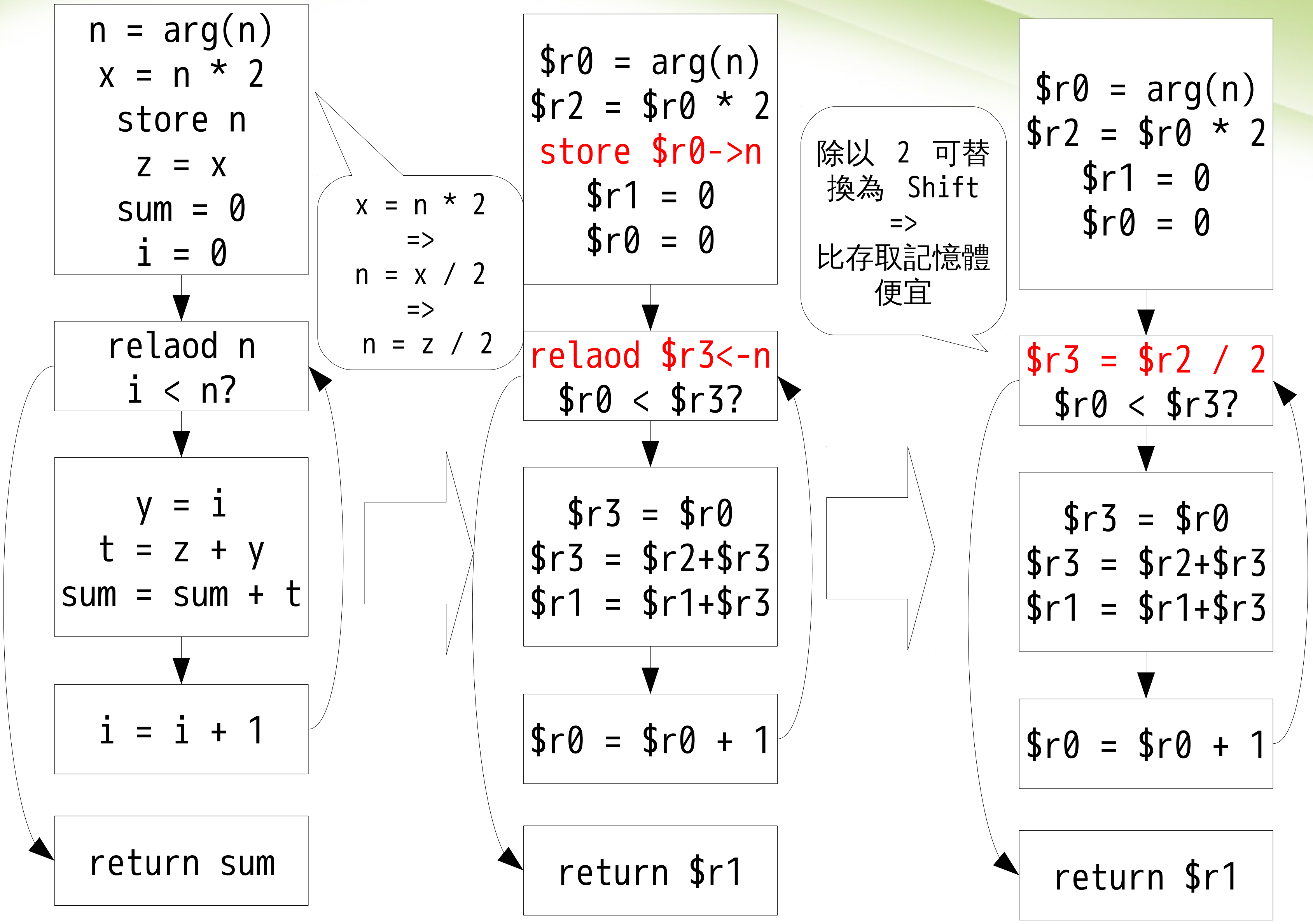
- Materialization: 實現化、具體化
- ReMaterialization: 重現化

Rematerialization

- Materialization: 實現化、具體化
- ReMaterialization: 重現化

- 在 Briggs 論文中所提出的技術：
  - 若其值可以很便宜的算出來， 則不進行 Spill， 取而代之直接重算其結果
  - *便宜的定義因目標不同而意義不同：
    - -O3: Cycle 數少即便宜
    - -Os: Code size 小即便宜

Rematerialization



```
n = arg(n)
x = n * 2
store n
z = x
sum = 0
i = 0

relaod n
i < n?

y = i
t = z + y
sum = sum + t

i = i + 1

return sum
```

```
x = n * 2
=>
n = x / 2
=>
n = z / 2
```

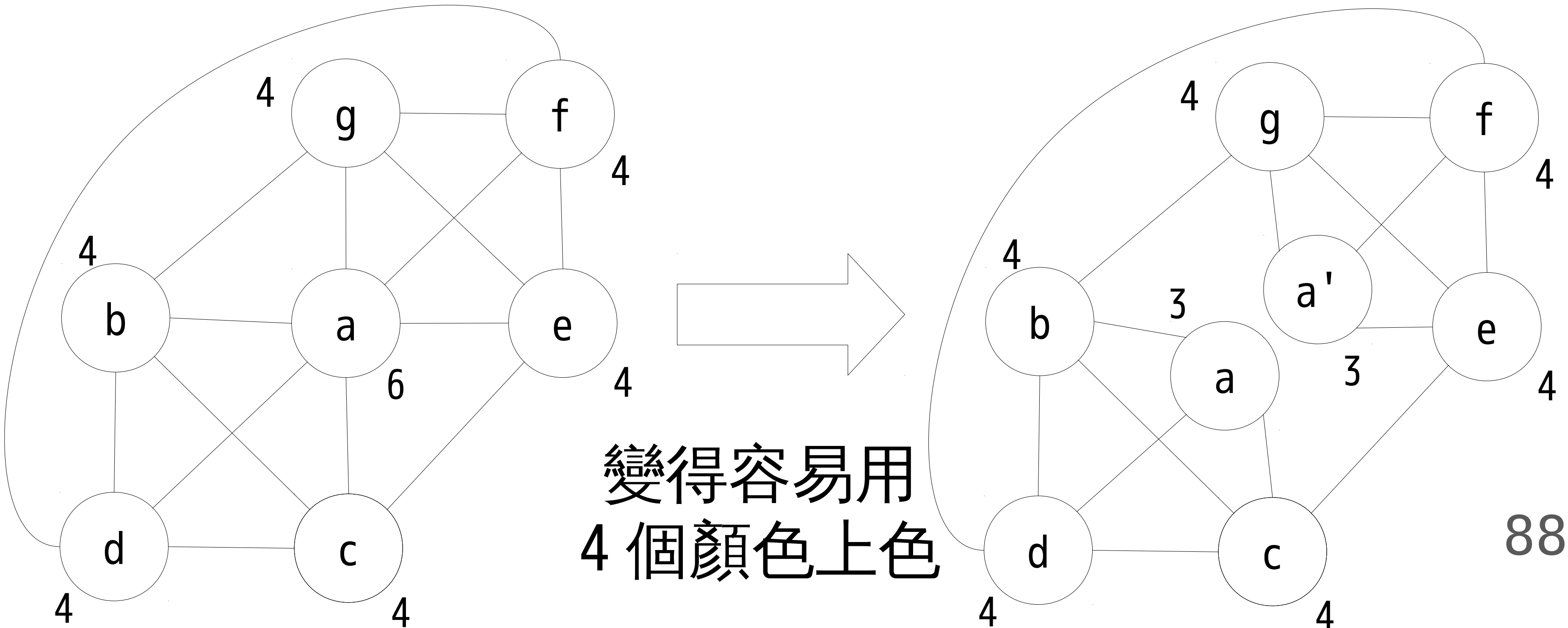
```
$r0 = arg(n)
$r2 = $r0 * 2
store $r0->n
$r1 = 0
$r0 = 0

relaod $r3<-n
$r0 < $r3?

$r3 = $r0
$r3 = $r2+$r3
$r1 = $r1+$r3

$r0 = $r0 + 1

return $r1
```

除以 2 可替換為 Shift
=>
比存取記憶體便宜

```
$r0 = arg(n)
$r2 = $r0 * 2
$r1 = 0
$r0 = 0

$r3 = $r2 / 2
$r0 < $r3?

$r3 = $r0
$r3 = $r2+$r3
$r1 = $r1+$r3

$r0 = $r0 + 1

return $r1
```

Split

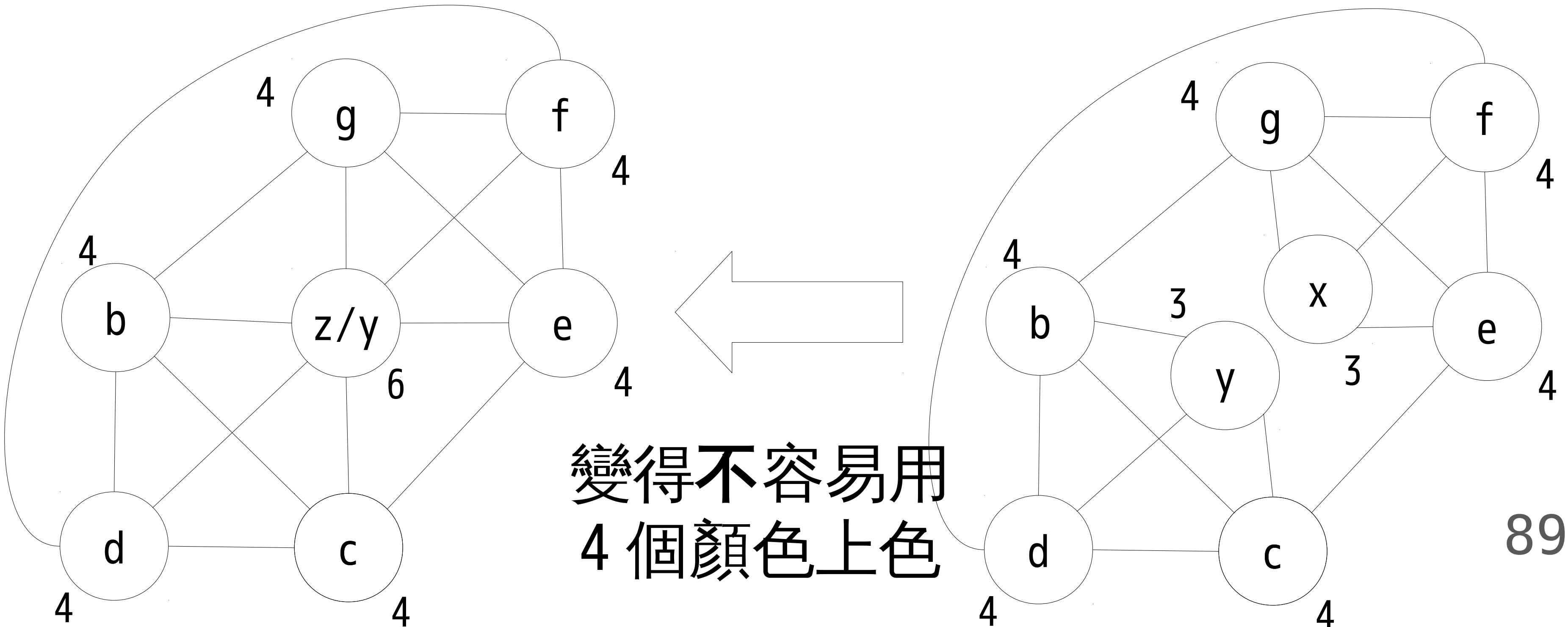
- 跟 Coalesce 相反， 也有將一個點拆成兩個的方法



變得容易用
4 個顏色上色

反思 Coalesce

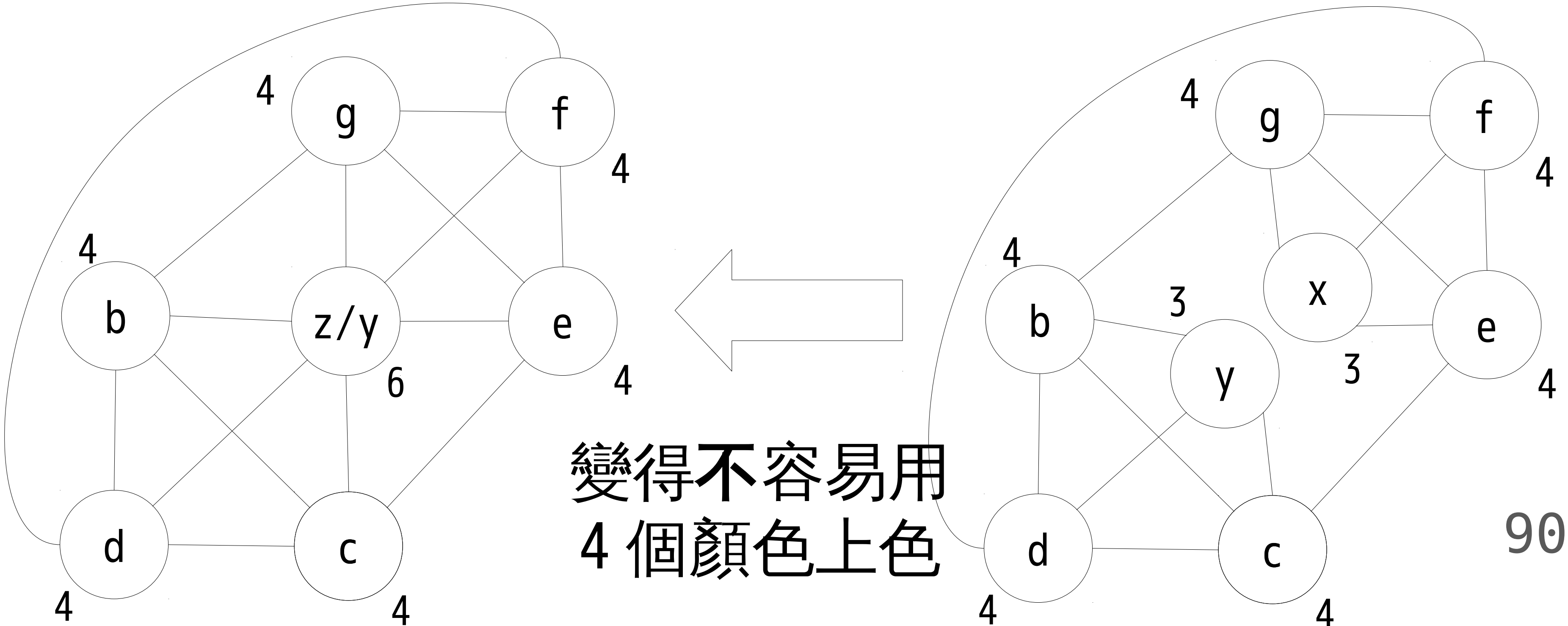
- 不好的 Coalesce 決策會造成圖難以著色，並生出額外的 Spill Code



變得**不**容易用
4 個顏色上色

反思 Coalesce

- 不好的 Coalesce 決策會造成圖難以著色，並生出額外的 Spill Code
- 但不好的 Split 決策相對會造成多餘 Move



變得**不**容易用
4 個顏色上色

Split or Coalesce

To be or not to be, that is the question

- RA 的學術研究上， 如何 Split 及 Coalesce 都可單獨成為一篇論文 ...

- Coalesce 在 CGO'07[1] 時被證明本身也是 NP-Complete

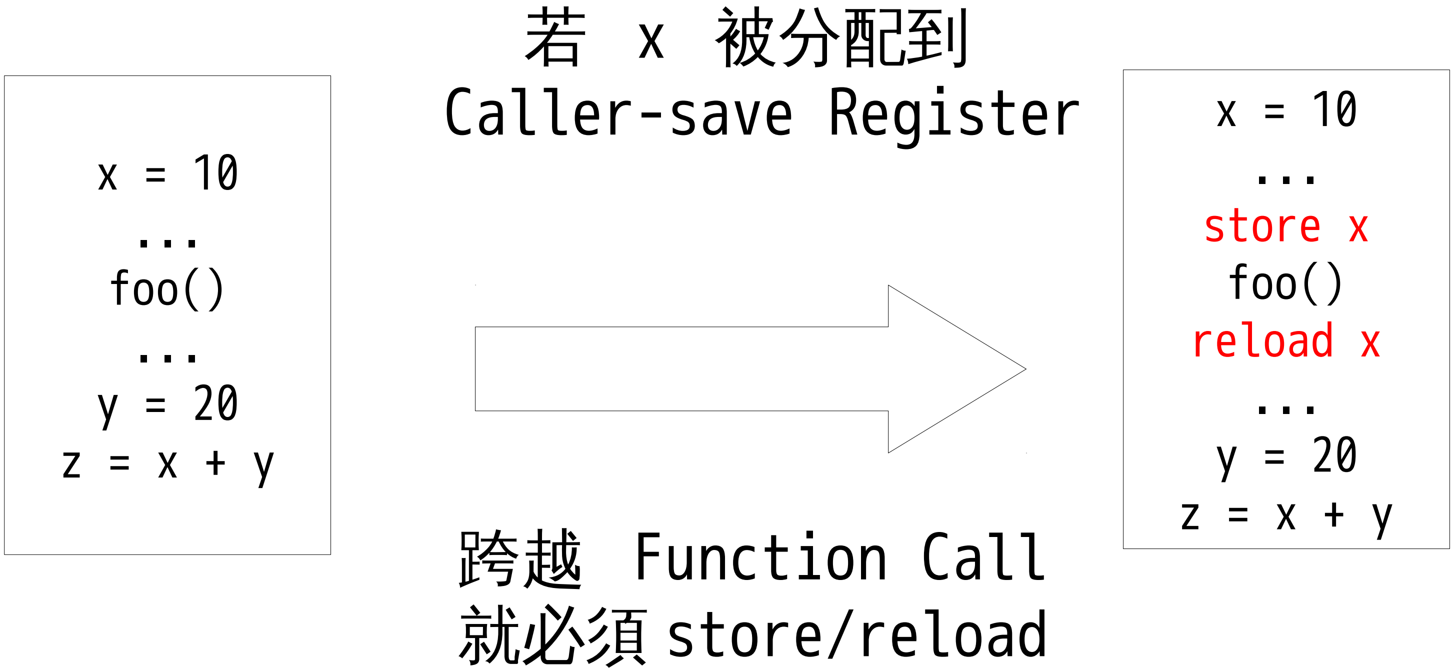
[1] Florent Bouchez, Alain Darte, and Fabrice Rastello. On the Complexity of Register Coalescing. In Proceedings of the International Symposium on Code Generation and Optimization, CGO ’07, pages 102-114, Washington, DC, USA, 2007. IEEE Computer Society.

從理論到現實

從理論到現實

- 演算法面看似很容易理解
  - 帶入現實情況就會發現基礎的Graph Coloring沒涵蓋一些問題...
    - Caller-save reg, Callee-save reg
    - 參數/回傳值放置位置
    - 不同的Register Class
    - Register Pair
    - 累加器(Accumulator)
    - Memory Operand

Caller-save Register or Callee-save Register



```
x = 10
...
foo()
...
y = 20
z = x + y
```

若 x 被分配到 Caller-save Register

```
x = 10
...
store x
foo()
reload x
...
y = 20
z = x + y
```

跨越 Function Call 就必須 store/reload

Caller-save Register or Callee-save Register

但用到
Callee-save Register
也必須在 Function 開頭
及結尾儲存跟復原

```
foo:
   store $r3
        ...
        ...
        ...
        ...
 restore $r3
      return
```

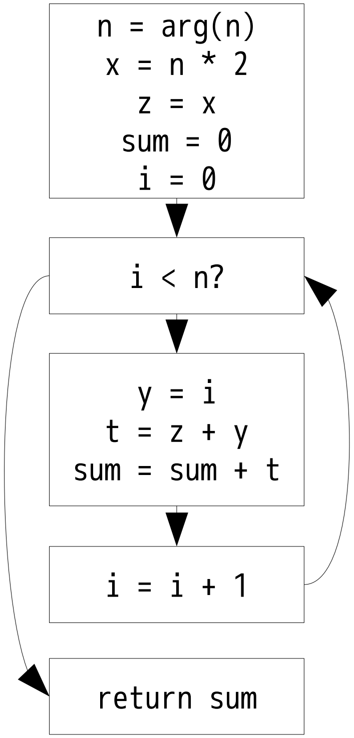
Caller-save Register or Callee-save Register

- 每個 Register 的使用成本隨著使用情境不同

- 著色時並非只看能否成功著色， 選合適的 Register 也是相當重要的一件事

參數 / 回傳值放置位置

- ABI 會規定參數與回傳值放置的方式
  - 以 nds32 為例：第一個參數放 $r0, 回傳值也放 $r0



```
n = arg(n)
x = n * 2
z = x
sum = 0
i = 0
```

```
i < n?
```

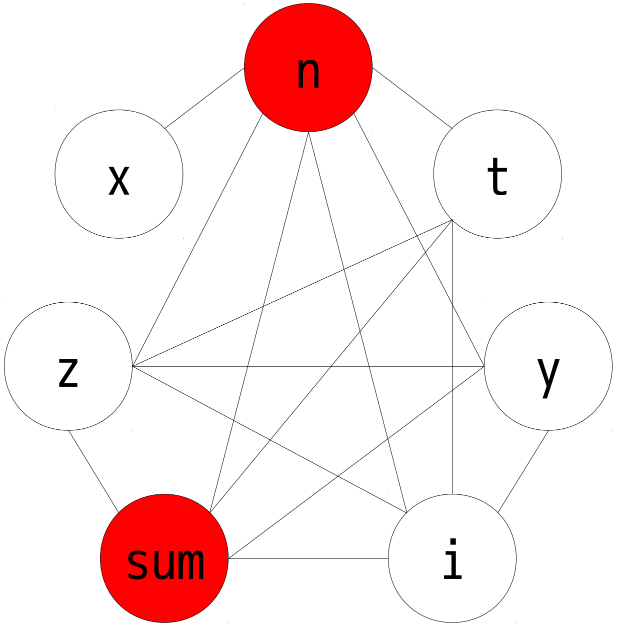
```
y = i
t = z + y
sum = sum + t
```

```
i = i + 1
```

```
return sum
```

以前面範例來看：
間接的使得 n 以及 sum 都必須使用 $r0, 否則需要額外 move 指令

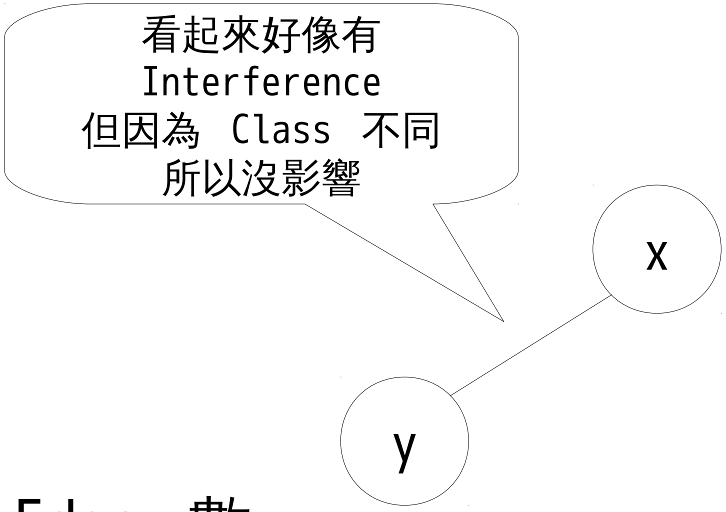
這項限制將會使得上色更為困難， 以這張圖為例，目前已經是不合法的著色



不同的 Register Class

- 最常見是分為兩大類：
  - Floating Point Register
  - General Purpose Register

```
int x = 10;
float y = 3.144444;
...
...
```


看起來好像有
Interference
但因為 Class 不同
所以沒影響

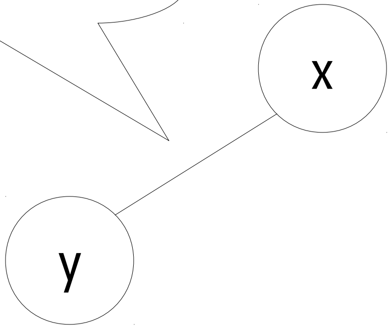
衍生思考： 只算帳面 Edge 數
無法正確判定是否可簡單著色

Register Pair

- 在許多架構中兩個 Floating Point Register 可合併成一個 double precision floating point

```
double x = 1.6188888;
float y = 3.144444;
...
...
```

看起來好像只有一條 Edge 但 x 會吃掉兩個 Register



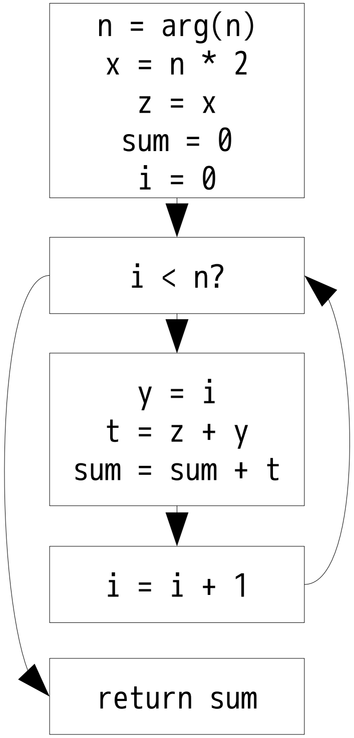
再次思考： 只算帳面 Edge 數真的不太可靠

累加器（Accumulator）

- 在 x86 架構中有累加器，其常見格式為
  - a = a op b
  - 回想前面例子，其分配結果無法直接套用在累加器架構下...
  - CB 演算法提出背景是 RISC GPR 架構！

Memory Operand

- 在 CISC 架構中有許多指令 Operand 可間接定址並且運算
  - addl   %eax, -4(%ebp) ! *(%ebp-4) += %eax



n = arg(n)
x = n * 2
z = x
sum = 0
i = 0

i < n?

y = i
t = z + y
sum = sum + t

i = i + 1

return sum

sum 若採用 memory operand, 建構出來的 Graph 會相當不一樣

Graph Coloring 外的選擇

- Linear Scan
- PBQP
- Puzzle
- Network Flow
- ILP

Register Allocation in **GCC**

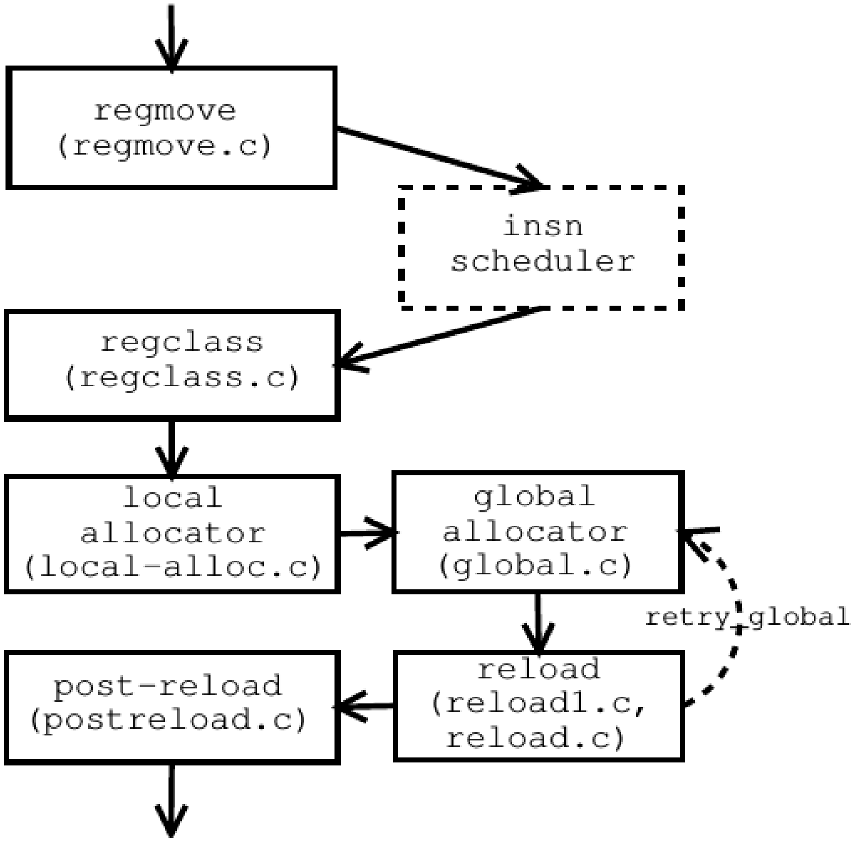
GCC RA 演進

- Local + Global + Reload

- IRA + Reload
  - GCC 4.4

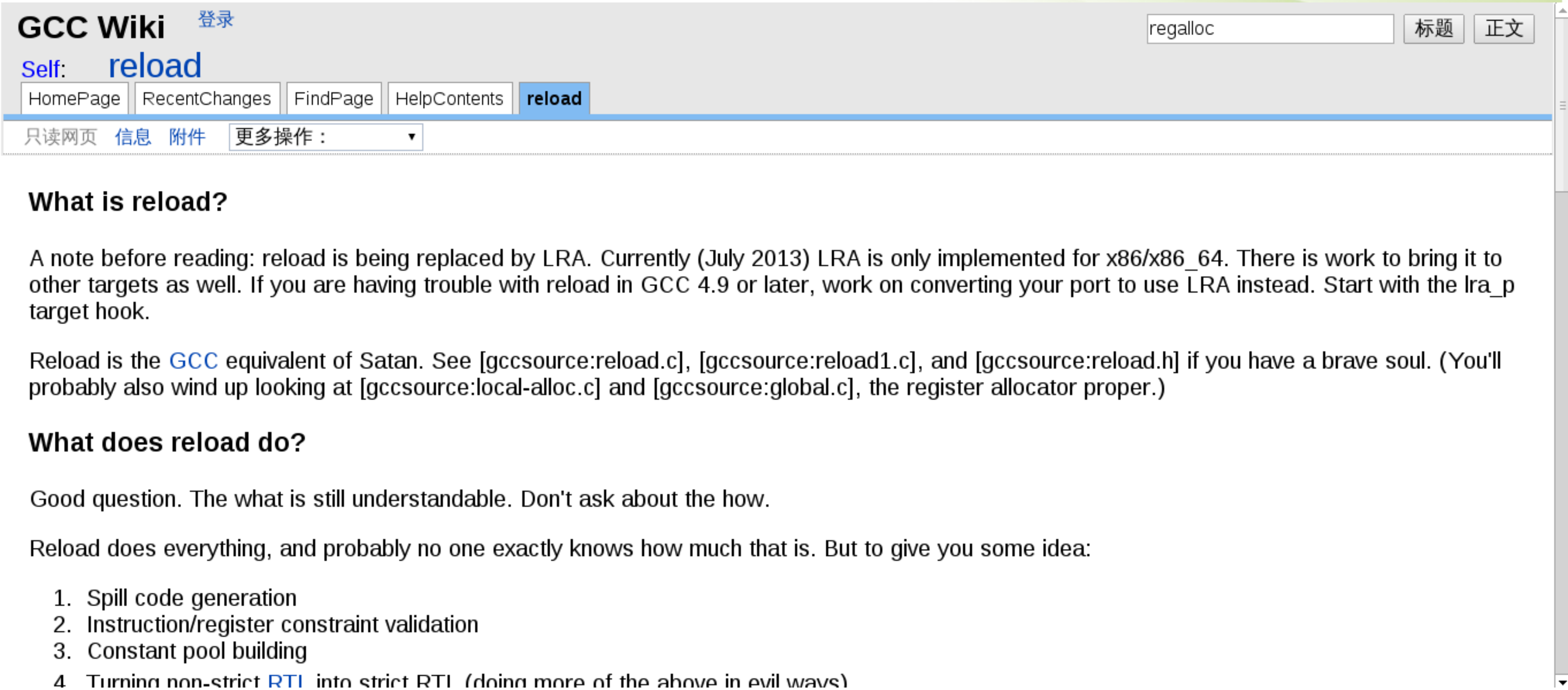
- IRA + LRA
  - GCC 4.8

Local + Global + Reload



Ref: Vladimir. N. .Makarov “Fighting register pressure in GCC”, GCC Submit 2004

Reload



What is reload?

A note before reading: reload is being replaced by LRA. Currently (July 2013) LRA is only implemented for x86/x86_64. There is work to bring it to other targets as well. If you are having trouble with reload in GCC 4.9 or later, work on converting your port to use LRA instead. Start with the lra_p target hook.

Reload is the GCC equivalent of Satan. See [gccsource:reload.c], [gccsource:reload1.c], and [gccsource:reload.h] if you have a brave soul. (You'll probably also wind up looking at [gccsource:local-alloc.c] and [gccsource:global.c], the register allocator proper.)

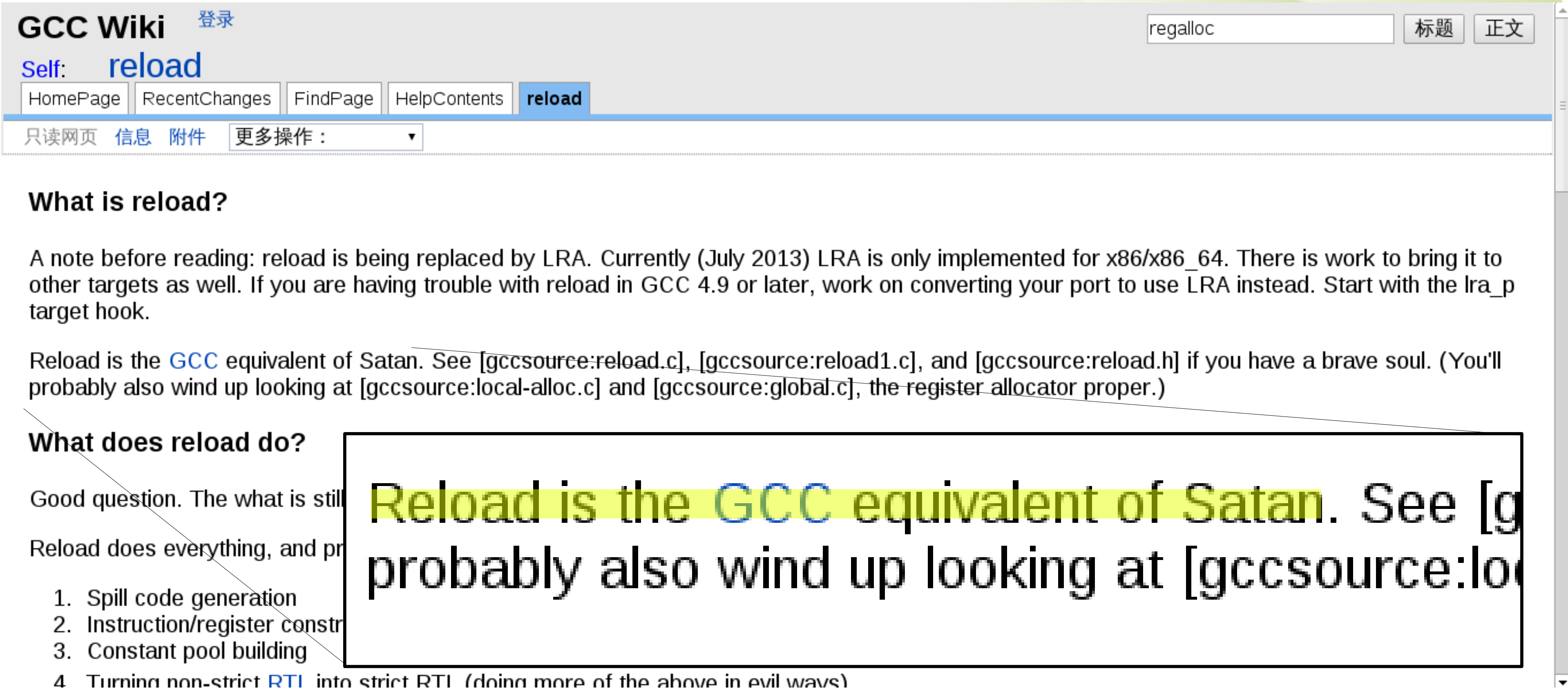
What does reload do?

Good question. The what is still understandable. Don't ask about the how.

Reload does everything, and probably no one exactly knows how much that is. But to give you some idea:

1. Spill code generation
2. Instruction/register constraint validation
3. Constant pool building
4. Turning non-strict RTL into strict RTL (doing more of the above in evil ways)

Reload



What is reload?

A note before reading: reload is being replaced by LRA. Currently (July 2013) LRA is only implemented for x86/x86_64. There is work to bring it to other targets as well. If you are having trouble with reload in GCC 4.9 or later, work on converting your port to use LRA instead. Start with the lra_p target hook.

Reload is the GCC equivalent of Satan. See [gccsource:reload.c], [gccsource:reload1.c], and [gccsource:reload.h] if you have a brave soul. (You'll probably also wind up looking at [gccsource:local-alloc.c] and [gccsource:global.c], the register allocator proper.)

What does reload do?

Good question. The what is stil

Reload does everything, and pr

1. Spill code generation
2. Instruction/register constr
3. Constant pool building
4. Turning non-strict RTL into strict RTL (doing more of the above in evil ways)

Reload is the GCC equivalent of Satan. See [g
probably also wind up looking at [gccsource:lo

Reload

- Spill code generation
- Instruction/register constraint validation
- Constant pool building
- Turning non-strict RTL into strict RTL (doing more of the above in evil ways)
- Register elimination--changing frame pointer references to stack pointer references
- Reload inheritance--essentially a builtin CSE pass on spill code

Reload

- 產生 Spill code
- 驗證所有指令跟暫存器的 Constraint
- 建立 Constant pool
- 把所有 non-strict RTL 轉成 strict RTL
  - 用超多邪惡的方法！！！
- 針對 frame pointer 跟 stack pointer 進行大融合
- 內建小型 CSE

Reload

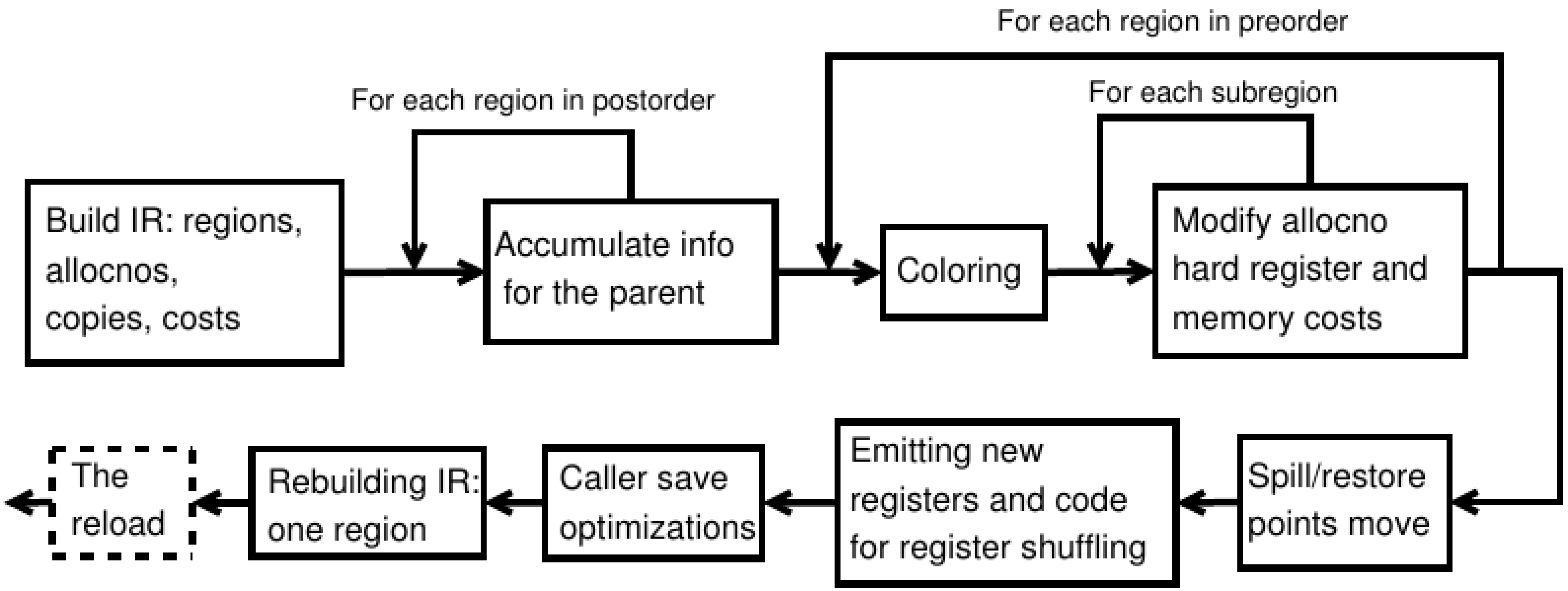
- 產生 Spill code
- 驗證所有指令跟暫存器的 Constraint
- 建立 Constant pool
- 把所有 non-strict RTL 轉成 strict RTL
  - 用超多邪惡的方法！！！
- 針對 frame pointer 跟 stack pointer 進行大融合
- 內建小型 CSE

Reload 最大問題在於所有東西黏在一團，難以修改維護 ... 加新功能？別鬧了 ...

```
git blame reload1.c |awk '{ print $3}'  | awk -F - '{ print $1}' | sort | uniq -c
git blame reload.c  |awk '{ print $3}'  | awk -F - '{ print $1}' | sort | uniq -c
```

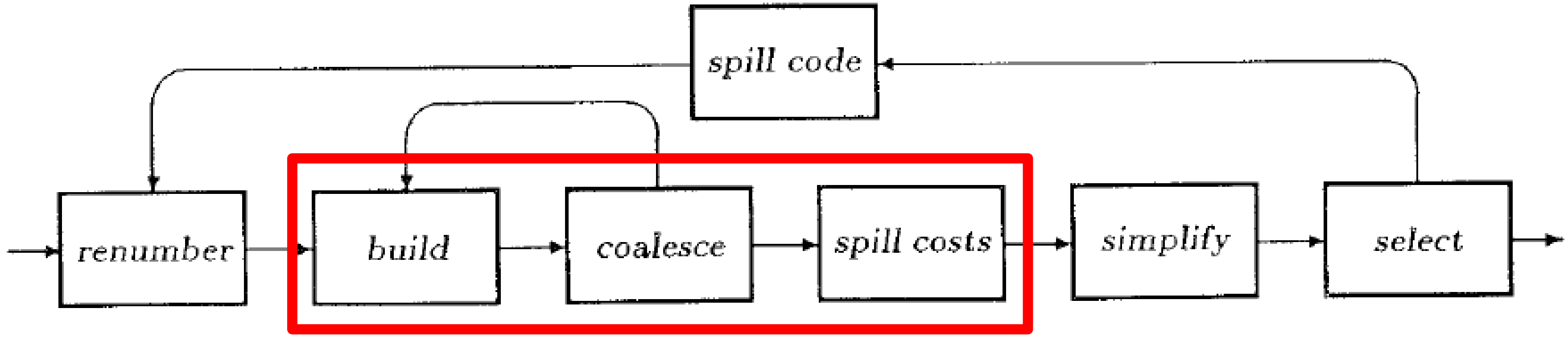
可透過這兩道指令觀察到 reload 大約都是 199x 年遺留下來的 ....

IRA + Reload


For each region in preorder
For each region in postorder
For each subregion
Build IR: regions, allocnos, copies, costs
Accumulate info for the parent
Coloring
Modify allocno hard register and memory costs
Spill/restore points move
Emitting new registers and code for register shuffling
Caller save optimizations
Rebuilding IR: one region
The reload

Ref: Vladimir. N. .Makarov, “The top-down regional register allocator for irregular register file architectures”

IRA



spill code

renumber

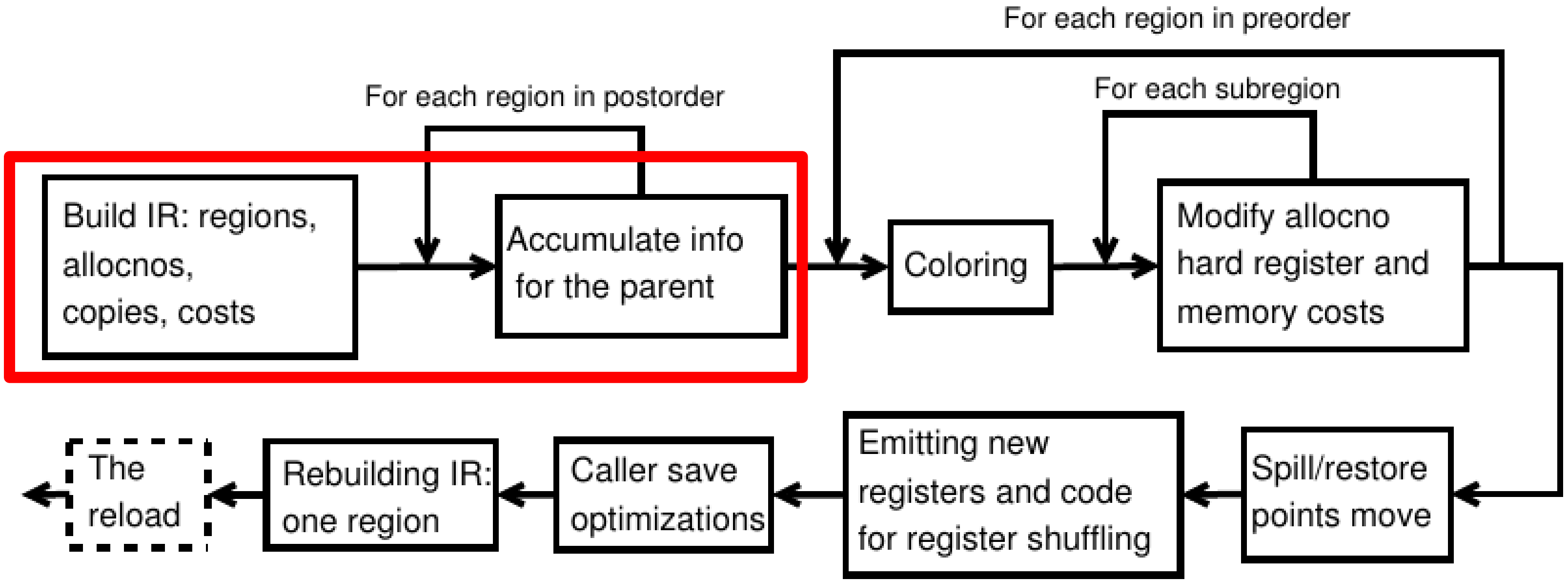
build

coalesce

spill costs

simplify

select



For each region in preorder

For each region in postorder

For each subregion

Build IR: regions, allocnos, copies, costs

Accumulate info for the parent

Coloring

Modify allocno hard register and memory costs

Spill/restore points move

Emitting new registers and code for register shuffling

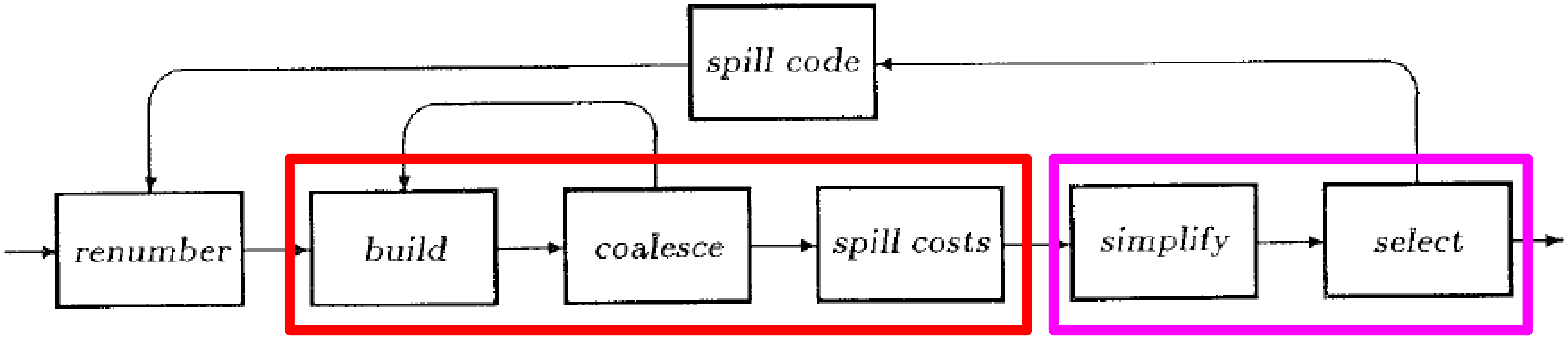
Caller save optimizations

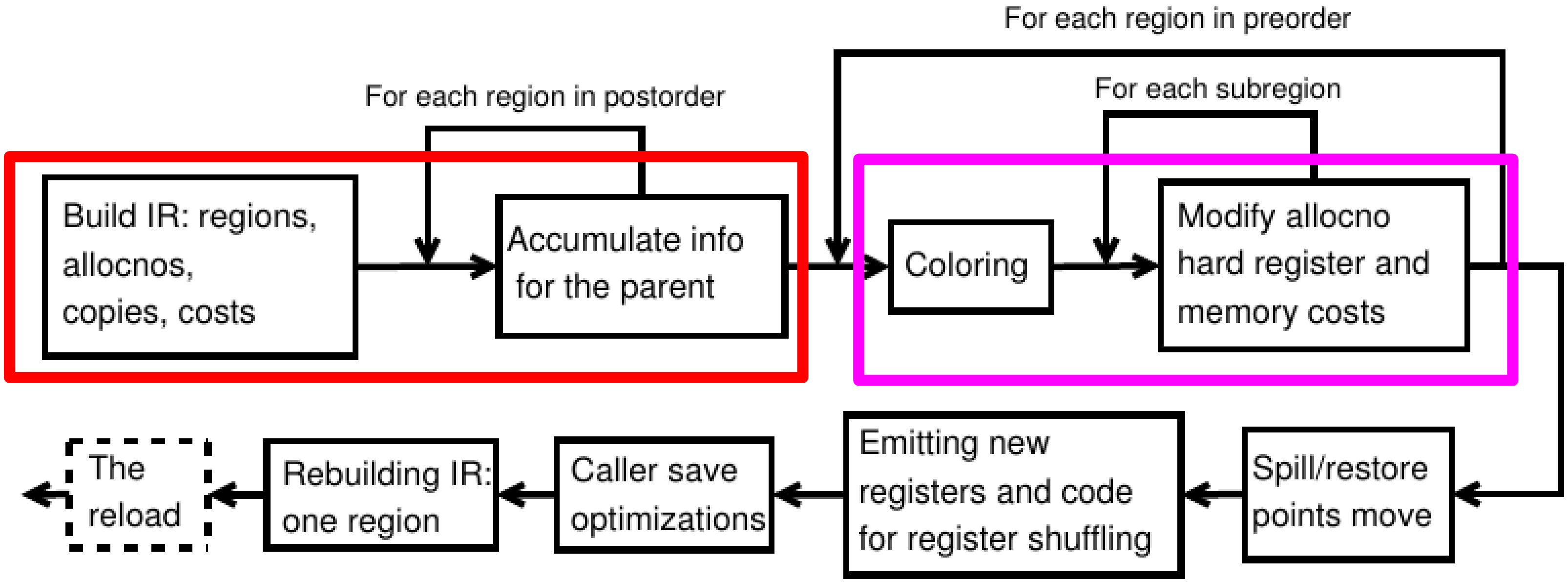
Rebuilding IR: one region

The reload

Ref: Vladimir. N. .Makarov, “The top-down regional register allocator for irregular register file architectures”

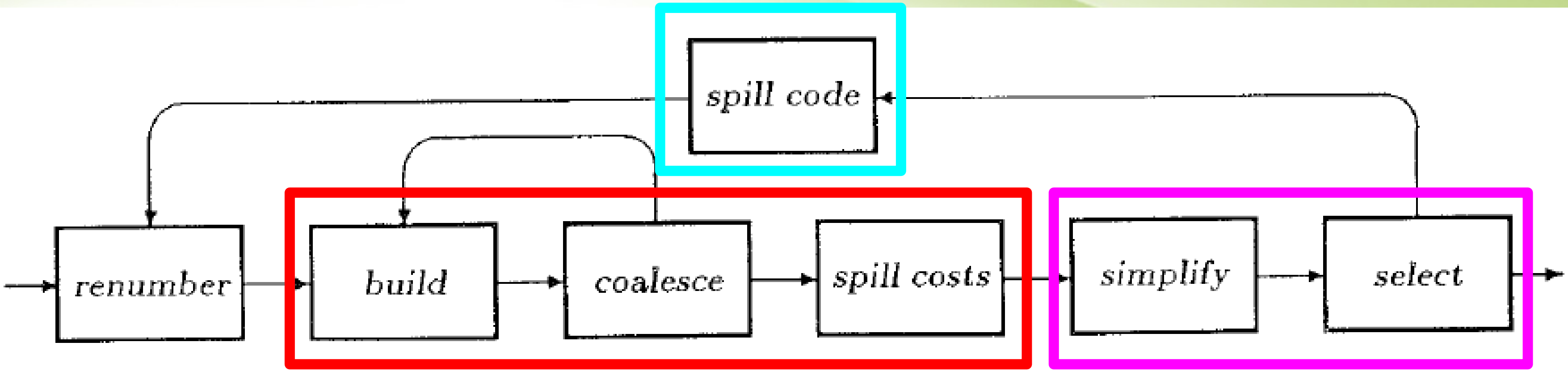
IRA

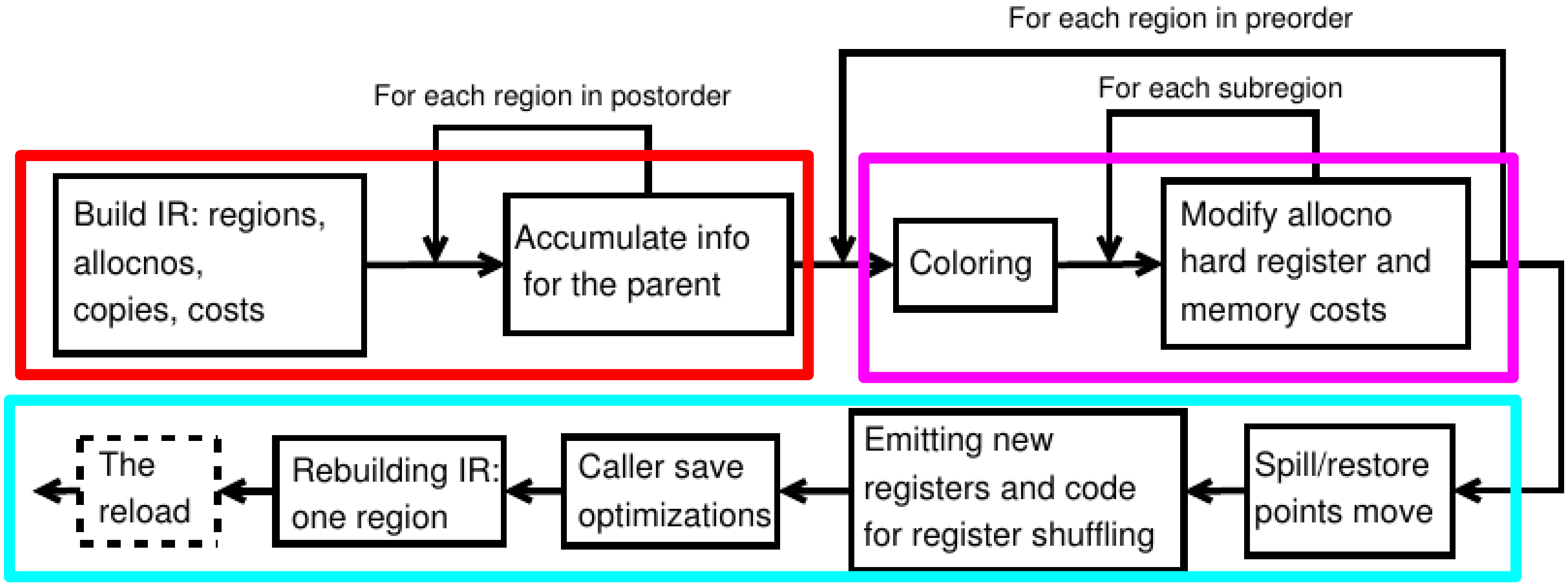



For each region in preorder
For each region in postorder
For each subregion
Build IR: regions, allocnos, copies, costs
Accumulate info for the parent
Coloring
Modify allocno hard register and memory costs
The reload
Rebuilding IR: one region
Caller save optimizations
Emitting new registers and code for register shuffling
Spill/restore points move

Ref: Vladimir. N. .Makarov, “The top-down regional register allocator for irregular register file architectures”

IRA





For each region in preorder
For each region in postorder
For each subregion
Build IR: regions, allocnos, copies, costs
Accumulate info for the parent
Coloring
Modify allocno hard register and memory costs
The reload
Rebuilding IR: one region
Caller save optimizations
Emitting new registers and code for register shuffling
Spill/restore points move

Ref: Vladimir. N. .Makarov, "The top-down regional register allocator for irregular register file architectures"

IRA

- CB 演算法為迭代式(Iterative Style), 但 IRA 整個流程只走一次
  - 時間與品質的取捨
  - 尚未完整整合 Reload

IRA

- Region-based Graph Coloring Register allocation
  - 主要參考 C ALLAHAN , D., AND KOBLENZ , B. Register allocation via hierarchical graph coloring. SIGPLAN 26, 6 (1991),192–203.
  - 要掌握 IRA 理論面， **至少**要讀完 ira.c 註解上列的幾篇參考文獻

深入觀察 IRA

- 以 nds32 target 為實驗平台
- 透過其 dump 資訊來研究 IRA:
  - nds32le-elf-gcc foo.c -O0 -fdump-rtl-ira -fira-verbose=9 -fomit-frame-pointer
    - -O0: 避免程式在 Middle-End 時， 被最佳化打亂
    - -fdump-rtl-ira: 吐出 IRA 的 dump
    - -fira-verbose=9: 吐出最多的 IRA 資訊
    - -fomit-frame-pointer: 不用 fp, 以簡化輸出

Let's hack IRA!

```
diff --git a/gcc/cfgexpand.c b/gcc/cfgexpand.c
index f6da5d6..9f96d25 100644
--- a/gcc/cfgexpand.c
+++ b/gcc/cfgexpand.c
@@ -5626,7 +5626,8 @@ pass_expand::execute (function *fun)
   edge e;
   rtx var_seq, var_ret_seq;
   unsigned i;

+  int saved_optimize = optimize;
+  optimize = 2;
   timevar_push (TV_OUT_OF_SSA);
   rewrite_out_of_ssa (&SA);
   timevar_pop (TV_OUT_OF_SSA);
@@ -5999,6 +6000,7 @@ pass_expand::execute (function *fun)

   timevar_pop (TV_POST_EXPAND);

+  optimize = saved_optimize;
   return 0;
 }
```

避免 GCC, GIMPLE->RTL 因 -O0 而產生過度冗餘的 Code

```
diff --git a/gcc/ira.c b/gcc/ira.c
index ccc6c79..16d3e55 100644
--- a/gcc/ira.c
+++ b/gcc/ira.c
@@ -5032,7 +5032,8 @@ ira (FILE *f)
   int rebuild_p;
   bool saved_flag_caller_saves = flag_caller_saves;
   enum ira_region saved_flag_ira_region = flag_ira_region;

+  optimize = 2;
+  flag_ira_region = IRA_REGION_ALL;
+  flag_expensive_optimizations = true;
   ira_conflicts_p = optimize > 0;

   ira_use_lra_p = targetm.lra_p ();
```

避免 IRA 便宜行事， 導致無法觀察

```
diff --git a/gcc/ira-build.c b/gcc/ira-build.c
index ee20c09..1a64748 100644
--- a/gcc/ira-build.c
+++ b/gcc/ira-build.c
@@ -2608,6 +2608,8 @@ remove_low_level_allocnos (void)
 static void
 remove_unnecessary_regions (bool all_p)
 {
+  if (!all_p)
+    return;
   if (current_loops == NULL)
     return;
   if (all_p)
```

避免 IRA 進行 Region 融合

Let's hack IRA!

```
diff --git a/gcc/config/nds32/nds32.h b/gcc/config/nds32/nds32.h
index bbcf100..a345638 100644
--- a/gcc/config/nds32/nds32.h
+++ b/gcc/config/nds32/nds32.h
@@ -503,13 +503,13 @@ enum nds32_builtins
    reserved for other use : $r24, $r25, $r26, $r27 */
 #define FIXED_REGISTERS                 \
 { /* r0  r1  r2  r3  r4  r5  r6  r7  */ \
-      0,  0,  0,  0,  0,  0,  0,  0,    \
+      0,  0,  0,  0,  1,  1,  1,  1,    \
   /* r8  r9  r10 r11 r12 r13 r14 r15 */ \
-      0,  0,  0,  0,  0,  0,  0,  1,    \
+      1,  1,  1,  1,  1,  1,  1,  1,    \
   /* r16 r17 r18 r19 r20 r21 r22 r23 */ \
-      0,  0,  0,  0,  0,  0,  0,  0,    \
+      1,  1,  1,  1,  1,  1,  1,  1,    \
   /* r24 r25 r26 r27 r28 r29 r30 r31 */ \
-      1,  1,  1,  1,  0,  1,  0,  1,    \
+      1,  1,  1,  1,  1,  1,  1,  1,    \
   /* ARG_POINTER:32 */                  \
       1,                                \
   /* FRAME_POINTER:33 */                \
@@ -524,13 +524,13 @@ enum nds32_builtins
    1 : caller-save registers */
 #define CALL_USED_REGISTERS             \
 { /* r0  r1  r2  r3  r4  r5  r6  r7  */ \
-      1,  1,  1,  1,  1,  1,  0,  0,    \
+      1,  1,  1,  1,  1,  1,  1,  1,    \
   /* r8  r9  r10 r11 r12 r13 r14 r15 */ \
-      0,  0,  0,  0,  0,  0,  0,  1,    \
+      1,  1,  1,  1,  1,  1,  1,  1,    \
   /* r16 r17 r18 r19 r20 r21 r22 r23 */ \
       1,  1,  1,  1,  1,  1,  1,  1,    \
   /* r24 r25 r26 r27 r28 r29 r30 r31 */ \
-      1,  1,  1,  1,  0,  1,  0,  1,    \
+      1,  1,  1,  1,  1,  1,  1,  1,    \
   /* ARG_POINTER:32 */                  \
       1,                                \
   /* FRAME_POINTER:33 */                \
```

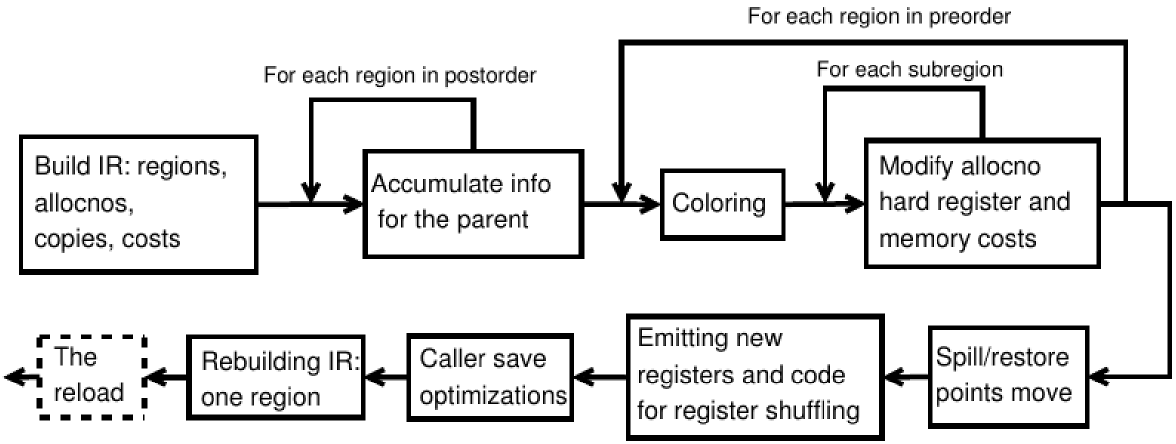
為方便小程式就會 Spill, 將 nds32 改成只有 4 個 Register

實驗環境準備

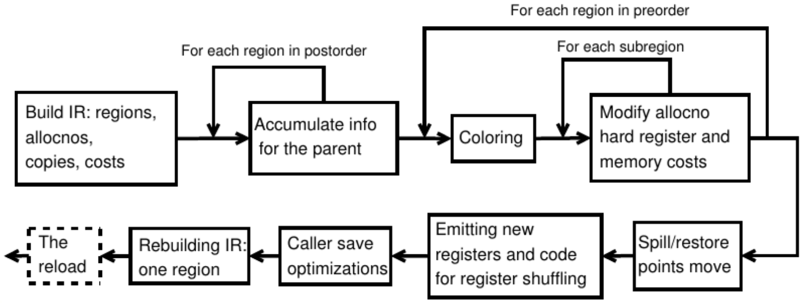
- git clone git://gcc.gnu.org/git/gcc.git ~/gcc-ra-test/src
- # 套上前兩頁的修改
- mkdir ~/build-gcc-ra-test
- cd ~/build-gcc-ra-test
- ~/gcc-ra-test/src/configure --prefix=$HOME/gcc-ra-test --target=nds32le-elf
- make all-gcc -j8 && make install-gcc

IRA

```
int foo(int n)
{
  int i, y, t, z;
  int x = n * 2;
  z = x;
  int sum = 0;
  for (i=0; i<n; ++i) {
    y = i;
    t = z + y;
    sum = sum + t;
  }
  return sum;
}
```


For each region in preorder
For each region in postorder
For each subregion
Build IR: regions, allocnos, copies, costs
Accumulate info for the parent
Coloring
Modify allocno hard register and memory costs
Spill/restore points move
Emitting new registers and code for register shuffling
Caller save optimizations
Rebuilding IR: one region
The reload

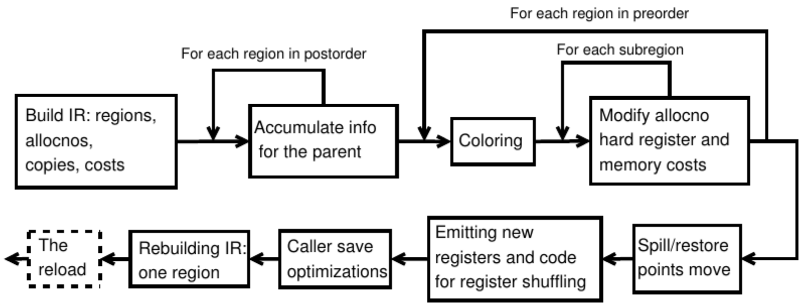
RTL


For each region in preorder
For each region in postorder
For each subregion
Build IR: regions, allocnos, copies, costs
Accumulate info for the parent
Coloring
Modify allocno hard register and memory costs
The reload
Rebuilding IR: one region
Caller save optimizations
Emitting new registers and code for register shuffling
Spill/restore points move

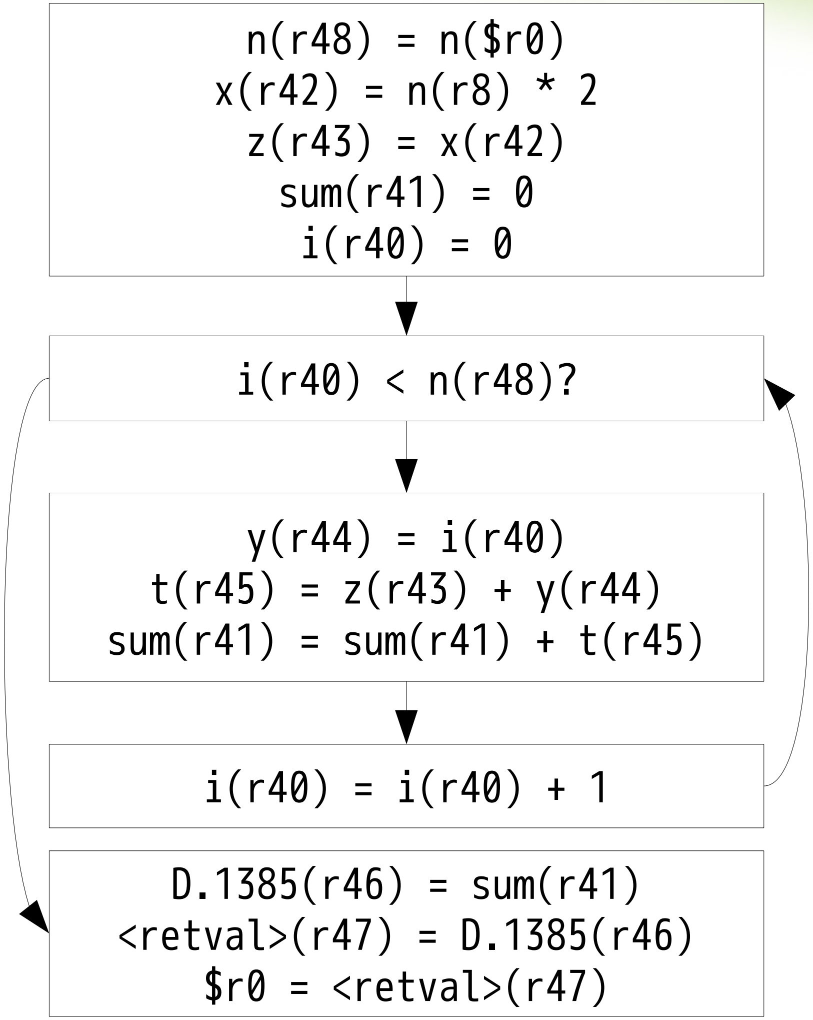
```
(insn 2 4 3 2 (set (reg/v:SI 48 [ n ])
        (reg:SI 0 $r0 [ n ])))
(insn 6 3 7 2 (set (reg/v:SI 42 [ x ])
        (ashift:SI (reg/v:SI 48 [ n ])
            (const_int 1 [0x1]))))
(insn 7 6 8 2 (set (reg/v:SI 43 [ z ])
        (reg/v:SI 42 [ x ])))
(insn 8 7 9 2 (set (reg/v:SI 41 [ sum ])
        (const_int 0 [0])))
(insn 9 8 33 2 (set (reg/v:SI 40 [ i ])
        (const_int 0 [0])))
(jump_insn 33 9 34 2 (set (pc)
        (label_ref 17)))
(code_label 19 34 12 3 3)
(insn 13 12 14 3 (set (reg/v:SI 44 [ y ])
        (reg/v:SI 40 [ i ])))
(insn 14 13 15 3 (set (reg/v:SI 45 [ t ])
        (plus:SI (reg/v:SI 43 [ z ])
            (reg/v:SI 44 [ y ]))))
(insn 15 14 16 3 (set (reg/v:SI 41 [ sum ])
        (plus:SI (reg/v:SI 41 [ sum ])
            (reg/v:SI 45 [ t ]))))
(insn 16 15 17 3 (set (reg/v:SI 40 [ i ])
        (plus:SI (reg/v:SI 40 [ i ])
            (const_int 1 [0x1]))))
```

```
(code_label 17 16 18 4 2)
(insn 20 18 21 4 (set (reg:SI 15 $ta)
        (lt:SI (reg/v:SI 40 [ i ])
            (reg/v:SI 48 [ n ]))))
(jump_insn 21 20 22 4 (set (pc)
        (if_then_else (ne (reg:SI 15 $ta)
                (const_int 0 [0]))
            (label_ref 19)
            (pc))))

(insn 23 22 26 5 (set (reg:SI 46 [ D.1385 ])
        (reg/v:SI 41 [ sum ])))
(insn 26 23 30 5 (set (reg:SI 47 [ <retval> ])
        (reg:SI 46 [ D.1385 ])))
(insn 30 26 31 5 (set (reg/i:SI 0 $r0)
        (reg:SI 47 [ <retval> ])))
(insn 31 30 0 5 (use (reg/i:SI 0 $r0)))
```

RTL


For each region in preorder
For each region in postorder
For each subregion
Build IR: regions, allocnos, copies, costs
Accumulate info for the parent
Coloring
Modify allocno hard register and memory costs
The reload
Rebuilding IR: one region
Caller save optimizations
Emitting new registers and code for register shuffling
Spill/restore points move

上一頁那沱 RTL 大致等價於右邊的 CFG



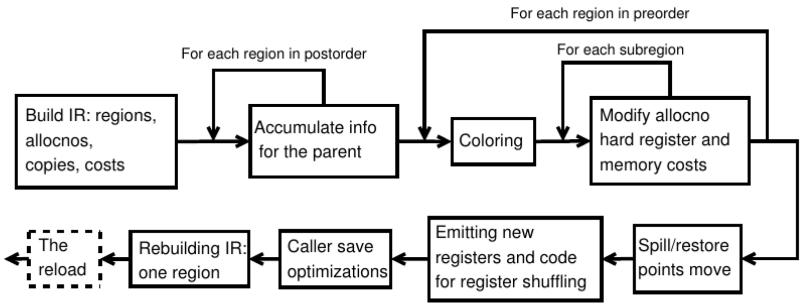
```
n(r48) = n($r0)
x(r42) = n(r8) * 2
z(r43) = x(r42)
sum(r41) = 0
i(r40) = 0
```

```
i(r40) < n(r48)?
```

```
y(r44) = i(r40)
t(r45) = z(r43) + y(r44)
sum(r41) = sum(r41) + t(r45)
```

```
i(r40) = i(r40) + 1
```

```
D.1385(r46) = sum(r41)
<retval>(r47) = D.1385(r46)
$r0 = <retval>(r47)
```


For each region in preorder

For each region in postorder

For each subregion

Build IR: regions, allocnos, copies, costs

Accumulate info for the parent

Coloring

Modify allocno hard register and memory costs

Spill/restore points move

Emitting new registers and code for register shuffling

Caller save optimizations

Rebuilding IR: one region

The reload

Pseudo Register

- GCC RTL 中在 RA 前有無限多 Pseudo Register
  - GCC Pseudo Register **不是** SSA Form

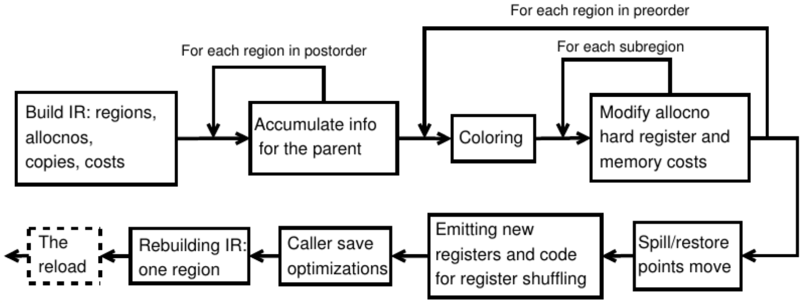
```
(set (reg:SI 45 [ t ])
     (plus:SI (reg:SI 43 [ z ])
              (reg:SI 44 [ y ])))
```

Pseudo Register Number

會保留在上層
IR(GIMPLE) 的變數名稱

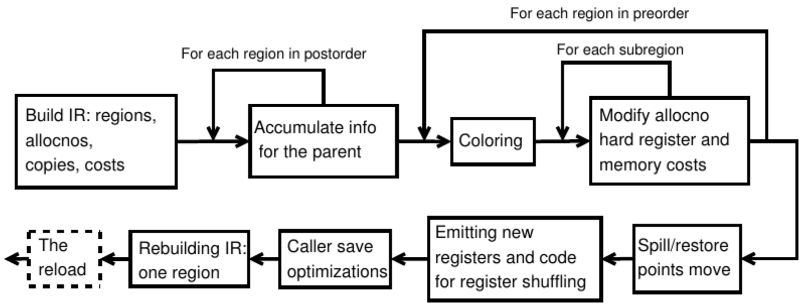
- 相對應的就是 Hard Register

```
(set (reg/v:SI 48 [ n ])
     (reg:SI 0 $r0 [ n ]))
```


For each region in preorder
For each region in postorder
For each subregion
Build IR: regions, allocnos, copies, costs
Accumulate info for the parent
Coloring
Modify allocno hard register and memory costs
Spill/restore points move
Emitting new registers and code for register shuffling
Caller save optimizations
Rebuilding IR: one region
The reload

Pseudo<->Var

Variable Name	Pseudo Register Number	Comment
i	40	
sum	41	
x	42	
y	44	
z	43	
t	45	
n	48	
D.1385	46	Temp variable create by expand
retval	47	Return Value


For each region in preorder
For each region in postorder
For each subregion
Build IR: regions, allocnos, copies, costs
Accumulate info for the parent
Coloring
Modify allocno hard register and memory costs
The reload
Rebuilding IR: one region
Caller save optimizations
Emitting new registers and code for register shuffling
Spill/restore points move

ABI

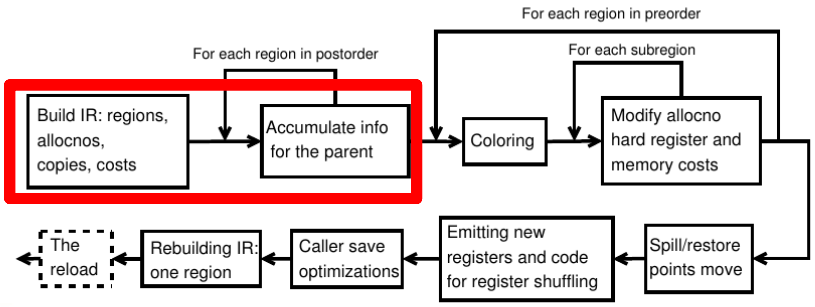
- Expand (Gimple->RTL) 階段時， 會針對參數及回傳值插入額外 move 指令

```
# 參數
(insn 2 4 3 2 (set (reg/v:SI 48 [ n ])
        (reg:SI 0 $r0 [ n ])) foo.c:2 27 {*movsi}
     (nil))
...
# 回傳值
(insn 23 22 26 5 (set (reg:SI 46 [ D.1385 ])
        (reg/v:SI 41 [ sum ])) foo.c:12 27 {*movsi}
     (nil))
(insn 26 23 30 5 (set (reg:SI 47 [ <retval> ])
        (reg:SI 46 [ D.1385 ])) foo.c:12 27 {*movsi}
     (nil))
(insn 30 26 31 5 (set (reg/i:SI 0 $r0)
        (reg:SI 47 [ <retval> ])) foo.c:13 27 {*movsi}
     (nil))
```

第一個參數放 $r0

回傳值也放 $r0

Region


For each region in postorder

For each region in preorder

For each subregion

Build IR: regions, allocnos, copies, costs

Accumulate info for the parent

Coloring

Modify allocno hard register and memory costs

Spill/restore points move

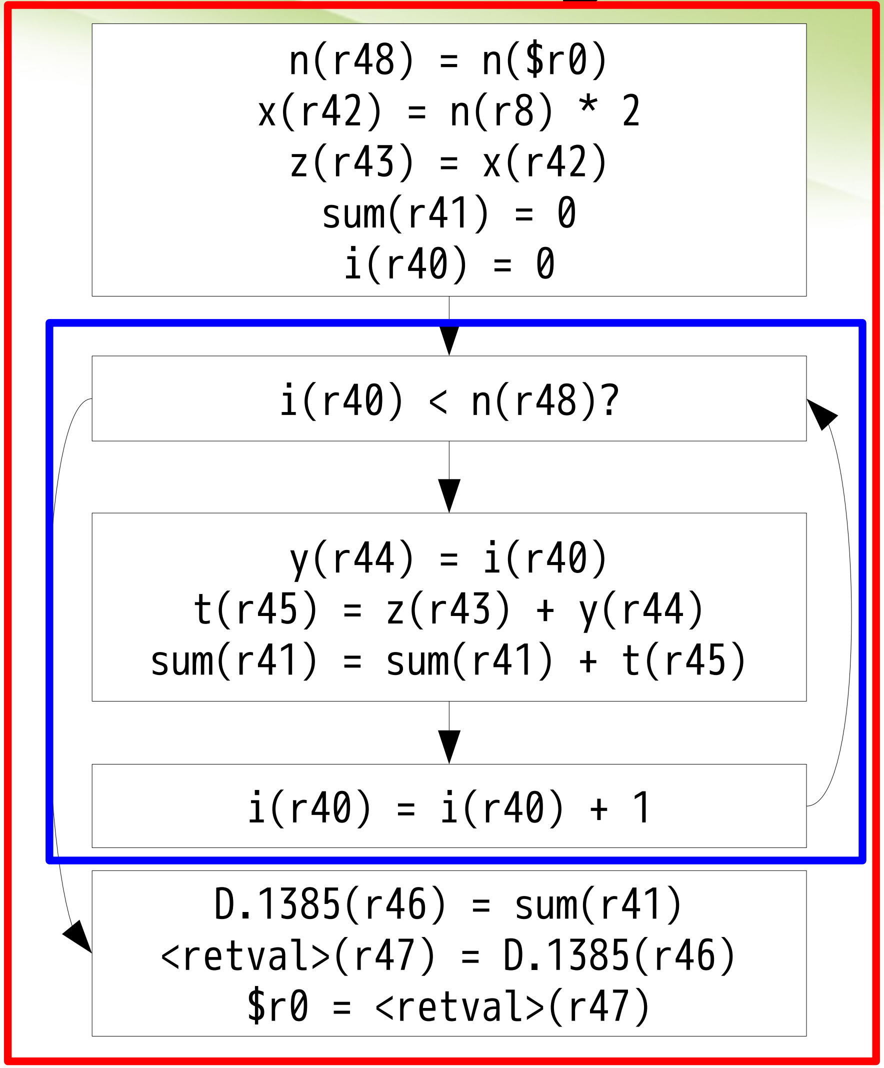
Emitting new registers and code for register shuffling

Caller save optimizations

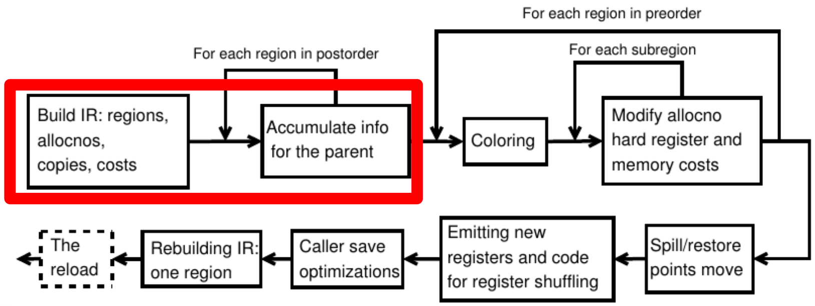
Rebuilding IR: one region

The reload

```
int foo(int n)
{
  int i, y, t, z;
  int x = n * 2;
  z = x;
  int sum = 0;
  for (i=0; i<n; ++i) {
    y = i;
    t = z + y;
    sum = sum + t;
  }          Region 1
  return sum;
}            Region 0
```

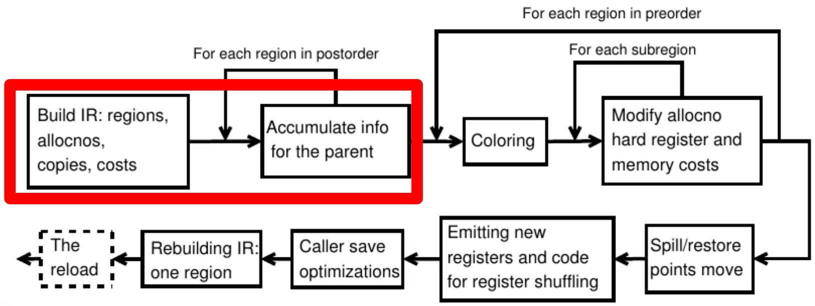

```
n(r48) = n($r0)
x(r42) = n(r8) * 2
z(r43) = x(r42)
sum(r41) = 0
i(r40) = 0

i(r40) < n(r48)?

y(r44) = i(r40)
t(r45) = z(r43) + y(r44)
sum(r41) = sum(r41) + t(r45)

i(r40) = i(r40) + 1

D.1385(r46) = sum(r41)
<retval>(r47) = D.1385(r46)
$r0 = <retval>(r47)
```

根據 Loop Structure 建立 Region, 若該 Loop Register Pressure 較低則會與上層 Region 合併


For each region in preorder
For each region in postorder
For each subregion
Build IR: regions, allocnos, copies, costs
Accumulate info for the parent
Coloring
Modify allocno hard register and memory costs
The reload
Rebuilding IR: one region
Caller save optimizations
Emitting new registers and code for register shuffling
Spill/restore points move

Allocno

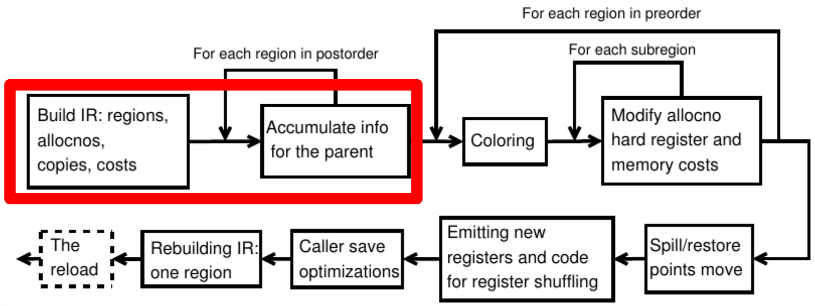
- 在 IRA 中 Pseudo Register 並不是點的單位，Allocno 在 IRA 才是等同於圖中的點
  - ira_allcno_t
- 在不同 Region 中一個 Pseudo Register 都有各自的 Allocno
  - Split by region


For each region in preorder
For each region in postorder
For each subregion
Build IR: regions, allocnos, copies, costs
Accumulate info for the parent
Coloring
Modify allocno hard register and memory costs
The reload
Rebuilding IR: one region
Caller save optimizations
Emitting new registers and code for register shuffling
Spill/restore points move

Allocno

- 一個 Allocno 由數個 Live Range 組成
- Live Range 在 IRA 中相對應的結構是 live_range_t
  - live_range_t = start/end program point

Build Allocno



For each region in postorder

Build IR: regions, allocnos, copies, costs

Accumulate info for the parent

For each region in preorder

Coloring

For each subregion

Modify allocno hard register and memory costs

Spill/restore points move

Emitting new registers and code for register shuffling

Caller save optimizations

Rebuilding IR: one region

The reload

IRA DUMP

```
...
  a0(r47,l0) costs: LOW_REGS:0,0 MIDDLE_REGS:0,0 GENERAL_REGS:0,0 ALL_REGS:0,0 MEM:2,2
  a1(r46,l0) costs: LOW_REGS:0,0 MIDDLE_REGS:0,0 GENERAL_REGS:0,0 ALL_REGS:0,0 MEM:2,2
  a2(r41,l0) costs: LOW_REGS:0,0 MIDDLE_REGS:0,0 GENERAL_REGS:0,0 ALL_REGS:0,0 MEM:6,22
  a3(r40,l0) costs: LOW_REGS:0,0 MIDDLE_REGS:0,0 GENERAL_REGS:0,0 ALL_REGS:0,0 MEM:5,30
  a4(r43,l0) costs: LOW_REGS:0,0 MIDDLE_REGS:0,0 GENERAL_REGS:0,0 ALL_REGS:0,0 MEM:1,9
  a5(r42,l0) costs: LOW_REGS:0,0 MIDDLE_REGS:0,0 GENERAL_REGS:0,0 ALL_REGS:0,0 MEM:9,9
  a6(r48,l0) costs: LOW_REGS:0,0 MIDDLE_REGS:0,0 GENERAL_REGS:0,0 ALL_REGS:0,0 MEM:9,17
  a7(r40,l1) costs: LOW_REGS:0,0 MIDDLE_REGS:0,0 GENERAL_REGS:0,0 ALL_REGS:0,0 MEM:25,25
  a8(r41,l1) costs: LOW_REGS:0,0 MIDDLE_REGS:0,0 GENERAL_REGS:0,0 ALL_REGS:0,0 MEM:16,16
  a9(r43,l1) costs: LOW_REGS:0,0 MIDDLE_REGS:0,0 GENERAL_REGS:0,0 ALL_REGS:0,0 MEM:8,8
  a10(r48,l1) costs: LOW_REGS:0,0 MIDDLE_REGS:0,0 GENERAL_REGS:0,0 ALL_REGS:0,0 MEM:8,8
  a11(r45,l1) costs: LOW_REGS:0,0 MIDDLE_REGS:0,0 GENERAL_REGS:0,0 ALL_REGS:0,0 MEM:16,16
  a12(r44,l1) costs: LOW_REGS:0,0 MIDDLE_REGS:0,0 GENERAL_REGS:0,0 ALL_REGS:0,0 MEM:9,9
...
```

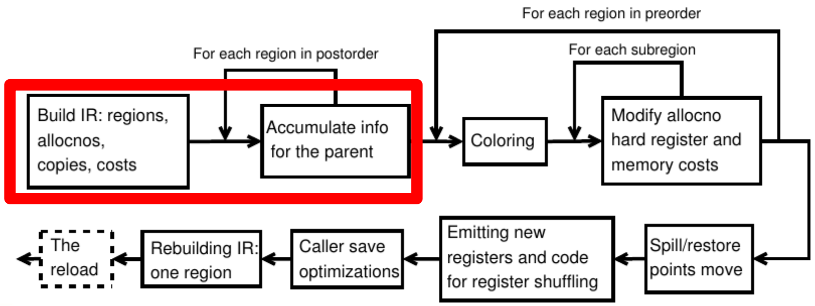
a12(r44,l1)

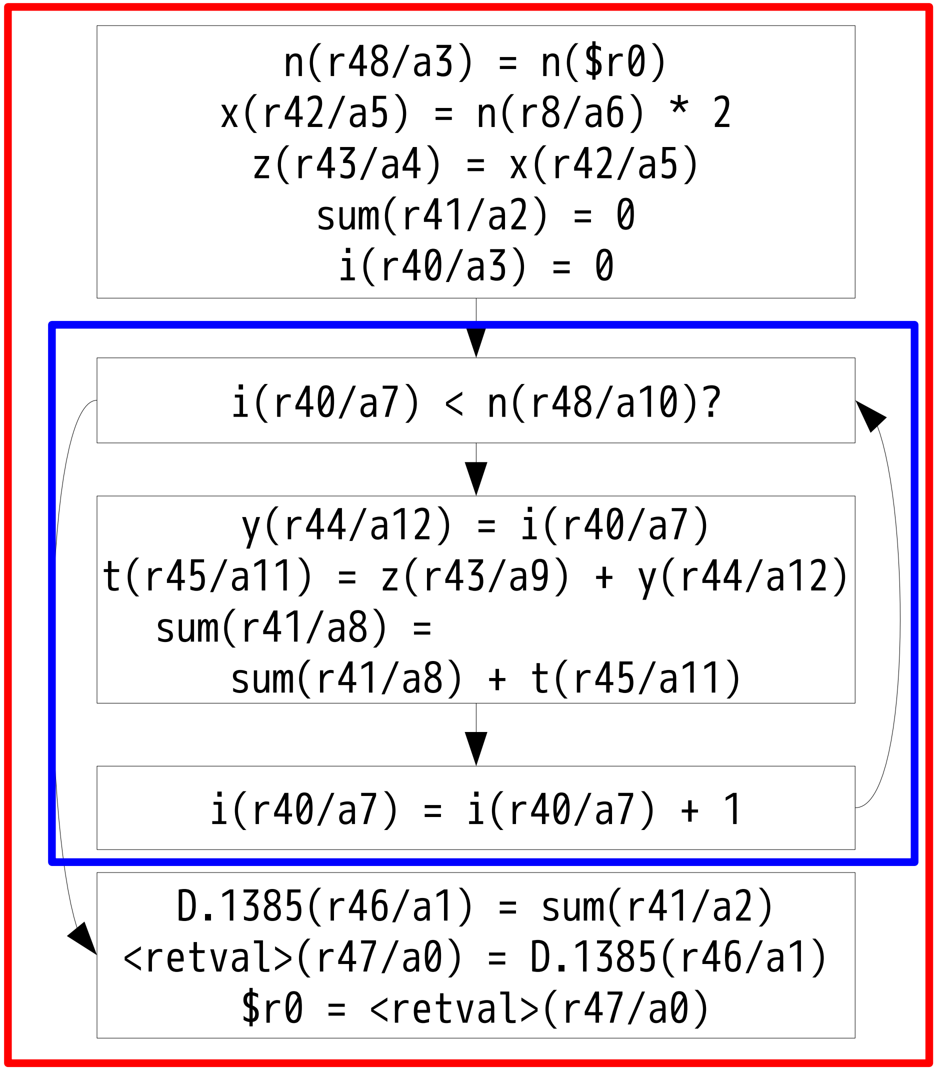
Allocno

Pseudo Register

Region

Build Allocno

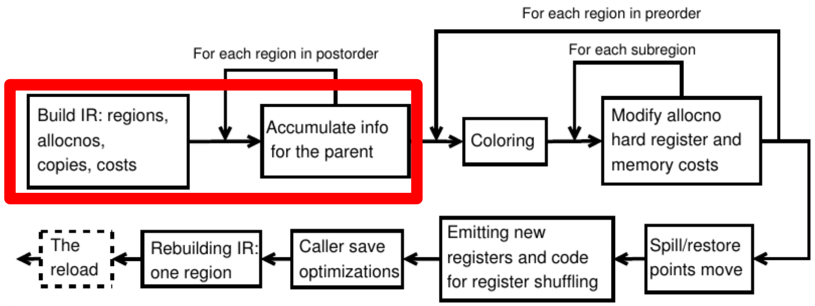

For each region in postorder
For each region in preorder
For each subregion
Build IR: regions, allocnos, copies, costs
Accumulate info for the parent
Coloring
Modify allocno hard register and memory costs
Spill/restore points move
Emitting new registers and code for register shuffling
Caller save optimizations
Rebuilding IR: one region
The reload



```
n(r48/a3) = n($r0)
x(r42/a5) = n(r8/a6) * 2
z(r43/a4) = x(r42/a5)
sum(r41/a2) = 0
i(r40/a3) = 0

i(r40/a7) < n(r48/a10)?

y(r44/a12) = i(r40/a7)
t(r45/a11) = z(r43/a9) + y(r44/a12)
sum(r41/a8) =
sum(r41/a8) + t(r45/a11)

i(r40/a7) = i(r40/a7) + 1

D.1385(r46/a1) = sum(r41/a2)
<retval>(r47/a0) = D.1385(r46/a1)
$r0 = <retval>(r47/a0)
```

Variable Name	Pseudo Register Number	Region 0	Region 1
i	40	a3	a7
sum	41	a2	a8
x	42	a5	
y	44		a12
z	43	a4	a9
t	45		a11
n	48	a6	a10
D.1385	46	a1	
retval	47	a0	

Build Cap



For each region in postorder

For each region in preorder

For each subregion

Build IR: regions, allocnos, copies, costs

Accumulate info for the parent

Coloring

Modify allocno hard register and memory costs

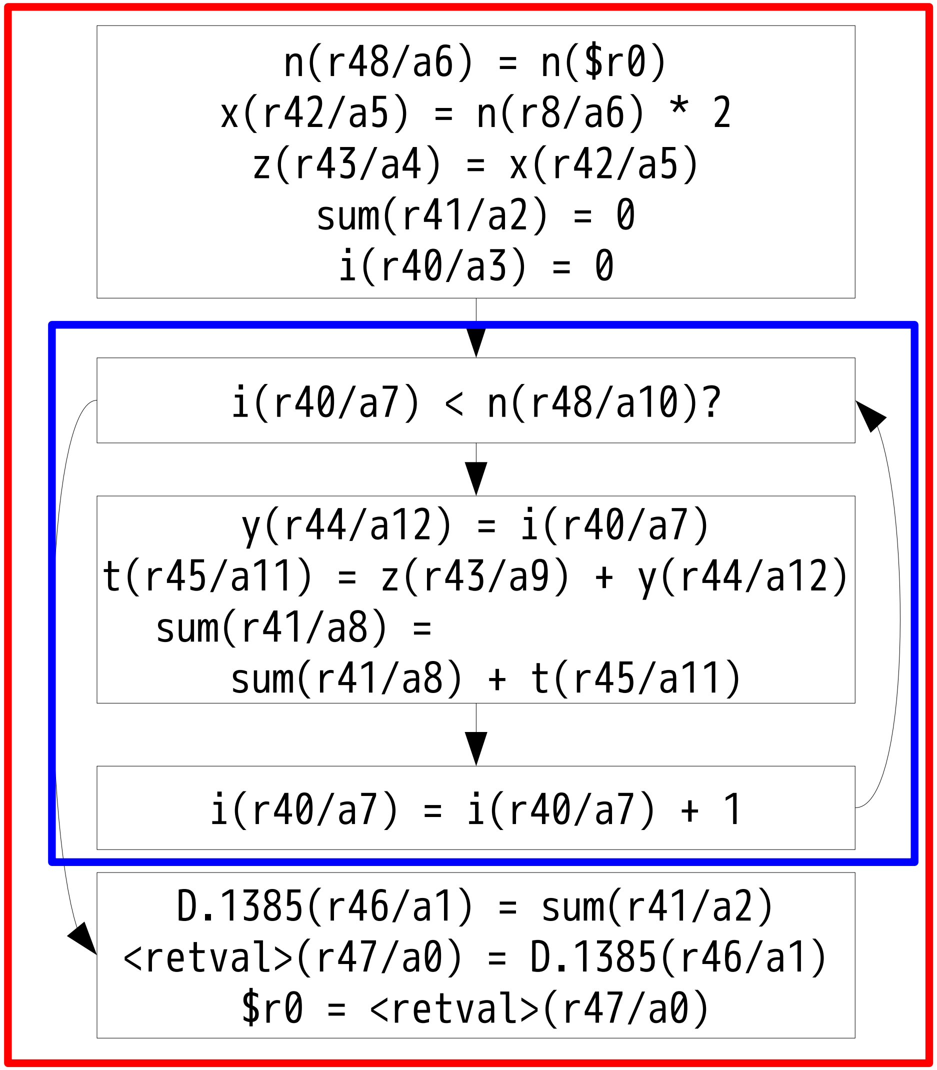
Spill/restore points move

Emitting new registers and code for register shuffling

Caller save optimizations

Rebuilding IR: one region

The reload



```
n(r48/a6) = n($r0)
x(r42/a5) = n(r8/a6) * 2
z(r43/a4) = x(r42/a5)
sum(r41/a2) = 0
i(r40/a3) = 0
```

```
i(r40/a7) < n(r48/a10)?
```

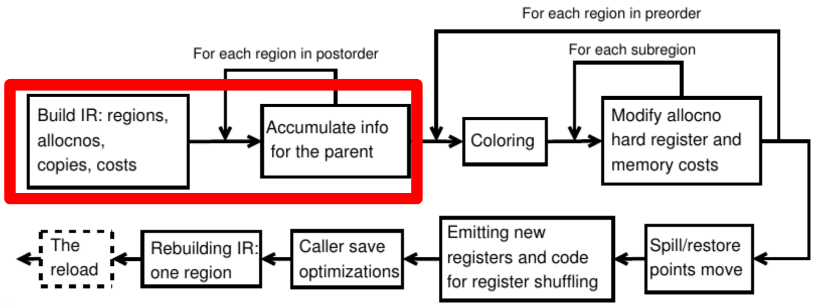
```
y(r44/a12) = i(r40/a7)
t(r45/a11) = z(r43/a9) + y(r44/a12)
sum(r41/a8) =
    sum(r41/a8) + t(r45/a11)
```

```
i(r40/a7) = i(r40/a7) + 1
```

```
D.1385(r46/a1) = sum(r41/a2)
<retval>(r47/a0) = D.1385(r46/a1)
$r0 = <retval>(r47/a0)
```

Variable Name	Pseudo Register Number	Region 0	Region 1
i	40	a3	a7
sum	41	a2	a8
x	42	a5	
y	44	a13	a12
z	43	a4	a9
t	45	a14	a11
n	48	a6	a10
D.1385	46	a1	
retval	47	a0	

Cap



For each region in preorder

For each region in postorder

For each subregion

Build IR: regions, allocnos, copies, costs

Accumulate info for the parent

Coloring

Modify allocno hard register and memory costs

The reload

Rebuilding IR: one region

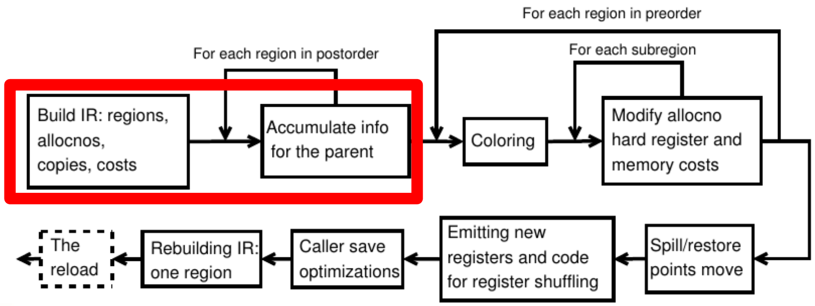
Caller save optimizations

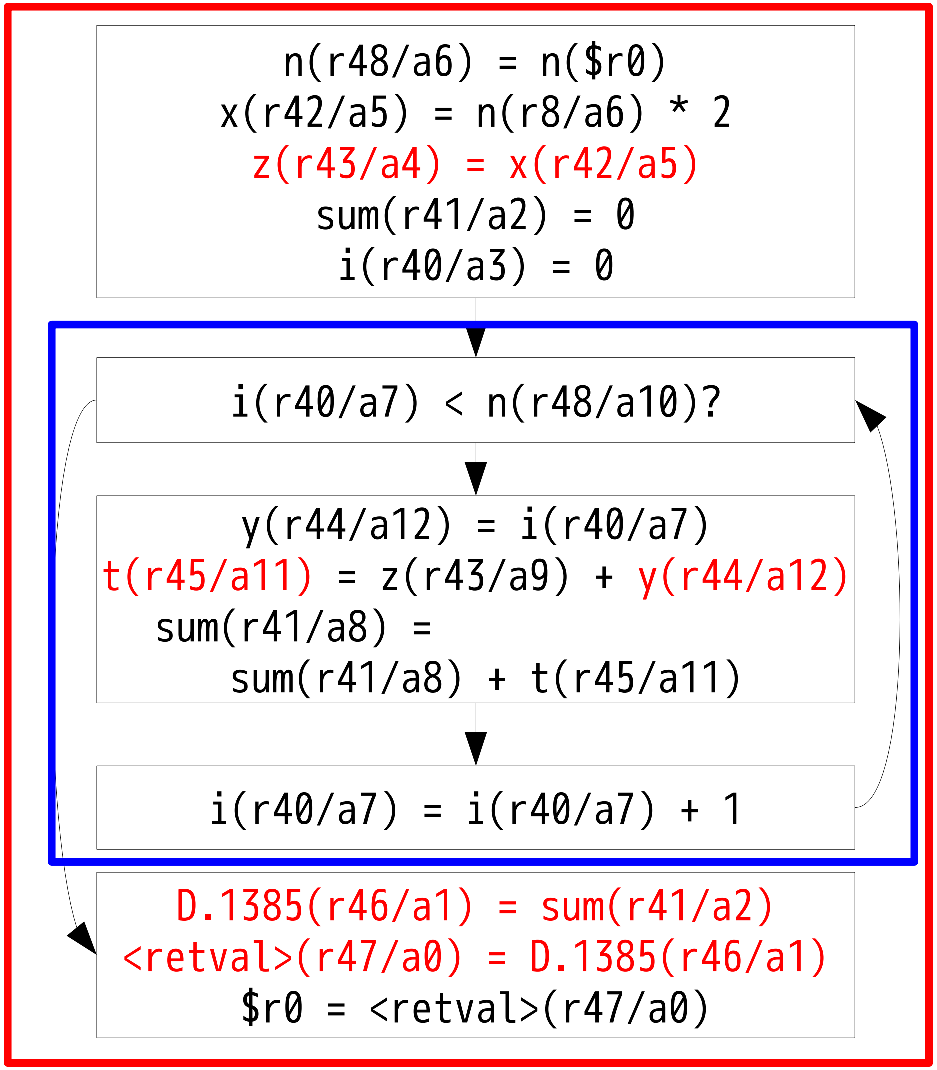
Emitting new registers and code for register shuffling

Spill/restore points move

- Allocno 的一種， 代表內層迴圈的變數
  - 用來輔助計算外層迴圈的 Allocno Cost

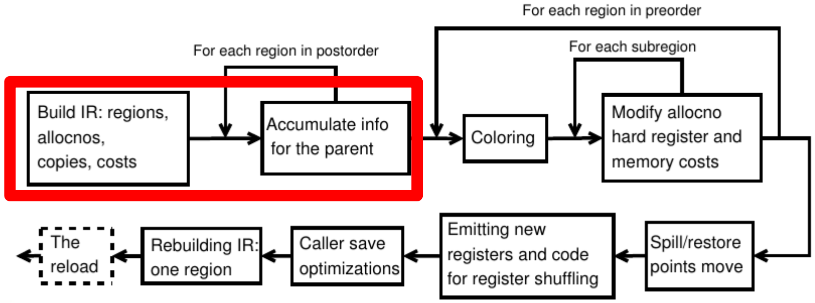
Build Copy


For each region in postorder
For each region in preorder
For each subregion
Build IR: regions, allocnos, copies, costs
Accumulate info for the parent
Coloring
Modify allocno hard register and memory costs
Spill/restore points move
Emitting new registers and code for register shuffling
Caller save optimizations
Rebuilding IR: one region
The reload



```
n(r48/a6) = n($r0)
x(r42/a5) = n(r8/a6) * 2
z(r43/a4) = x(r42/a5)
sum(r41/a2) = 0
i(r40/a3) = 0

i(r40/a7) < n(r48/a10)?

y(r44/a12) = i(r40/a7)
t(r45/a11) = z(r43/a9) + y(r44/a12)
sum(r41/a8) =
    sum(r41/a8) + t(r45/a11)

i(r40/a7) = i(r40/a7) + 1

D.1385(r46/a1) = sum(r41/a2)
<retval>(r47/a0) = D.1385(r46/a1)
$r0 = <retval>(r47/a0)
```

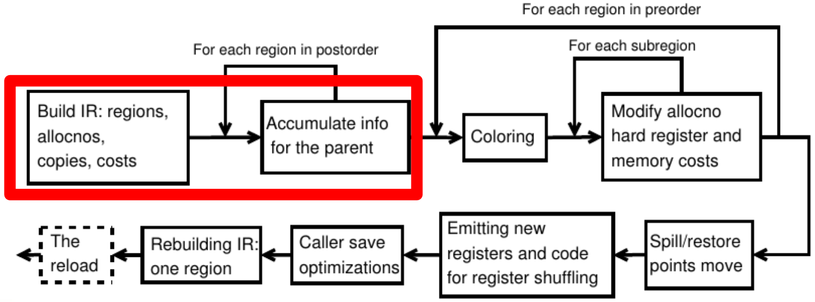
```
...
  cp0:a1(r46)<->a2(r41)@1:move
  cp1:a0(r47)<->a1(r46)@1:move
  cp2:a4(r43)<->a5(r42)@1:move
  cp3:a11(r45)<->a12(r44)@1:shuffle
  cp4:a13(r45)<->a14(r44)@1:shuffle
...
```


For each region in preorder
For each region in postorder
For each subregion
Build IR: regions, allocnos, copies, costs
Accumulate info for the parent
Coloring
Modify allocno hard register and memory costs
The reload
Rebuilding IR: one region
Caller save optimizations
Emitting new registers and code for register shuffling
Spill/restore points move

Copy

- 除了原本 `a = b` 這類指令外 IRA 也引進另一類型的 coalesce
  - `a = b op c`
  - 若 a 與 b 或 c 無 Interference 則也可 coalesce
  - 可減少 Register Pressure

Register Preference



For each region in postorder

For each region in preorder

For each subregion

Build IR: regions, allocnos, copies, costs

Accumulate info for the parent

Coloring

Modify allocno hard register and memory costs

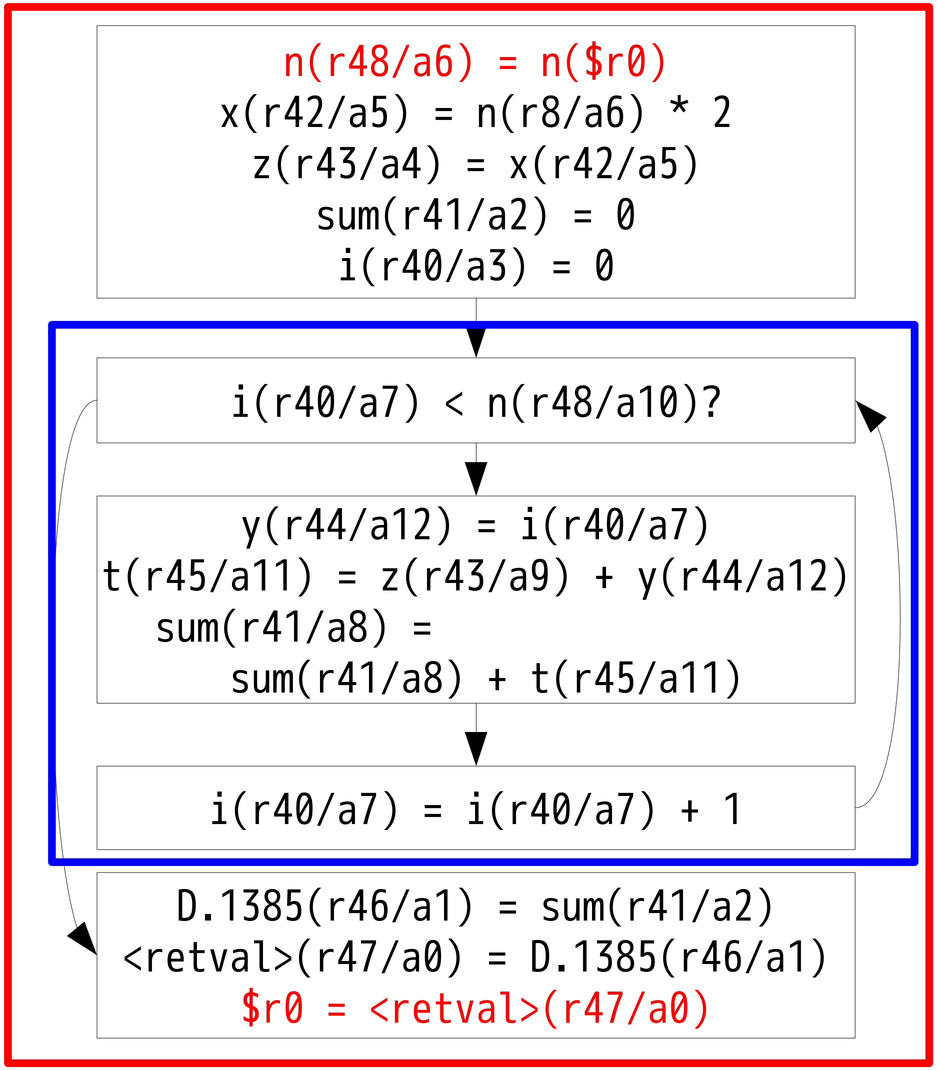
Spill/restore points move

Emitting new registers and code for register shuffling

Caller save optimizations

Rebuilding IR: one region

The reload



```
n(r48/a6) = n($r0)
x(r42/a5) = n(r8/a6) * 2
z(r43/a4) = x(r42/a5)
sum(r41/a2) = 0
i(r40/a3) = 0
```

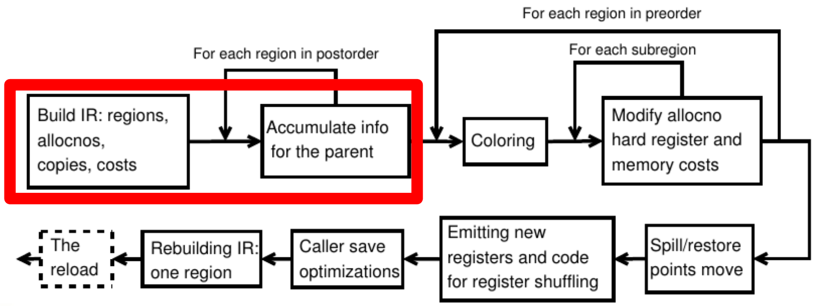
```
i(r40/a7) < n(r48/a10)?
```

```
y(r44/a12) = i(r40/a7)
t(r45/a11) = z(r43/a9) + y(r44/a12)
sum(r41/a8) =
    sum(r41/a8) + t(r45/a11)
```

```
i(r40/a7) = i(r40/a7) + 1
```

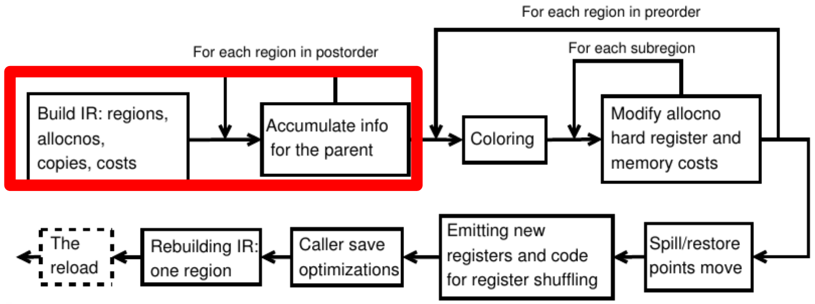
```
D.1385(r46/a1) = sum(r41/a2)
<retval>(r47/a0) = D.1385(r46/a1)
$r0 = <retval>(r47/a0)
```

```
...
  pref0:a0(r47)<-hr0@2
  pref1:a6(r48)<-hr0@2
...
```


For each region in preorder
For each region in postorder
For each subregion
Build IR: regions, allocnos, copies, costs
Accumulate info for the parent
Coloring
Modify allocno hard register and memory costs
The reload
Rebuilding IR: one region
Caller save optimizations
Emitting new registers and code for register shuffling
Spill/restore points move

Register Preference

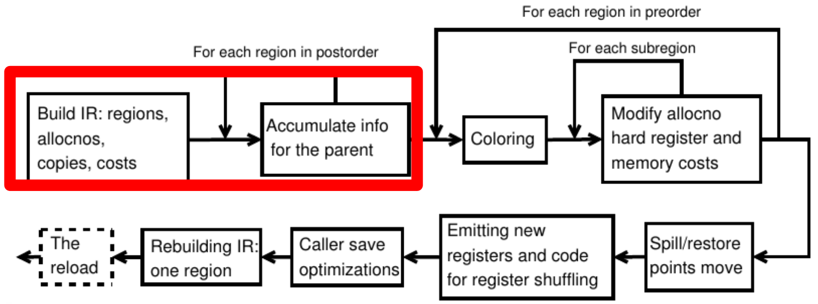
- 某些 Allocno 若分配到特定暫存器則可減少 Move 指令
  - 主要來自 ABI 的限制


For each region in preorder
For each region in postorder
For each subregion
Build IR: regions, allocnos, copies, costs
Accumulate info for the parent
Coloring
Modify allocno hard register and memory costs
The reload
Rebuilding IR: one region
Caller save optimizations
Emitting new registers and code for register shuffling
Spill/restore points move

Cost

ira-costs.c

算 Cost 的邏輯都放這！



For each region in preorder

For each region in postorder

For each subregion

Build IR: regions, allocnos, copies, costs

Accumulate info for the parent

Coloring

Modify allocno hard register and memory costs

The reload

Rebuilding IR: one region

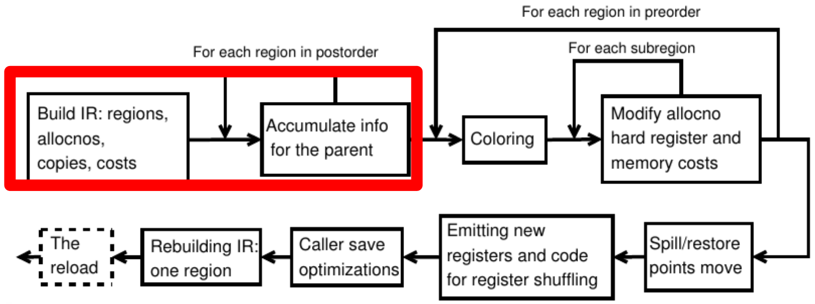
Caller save optimizations

Emitting new registers and code for register shuffling

Spill/restore points move

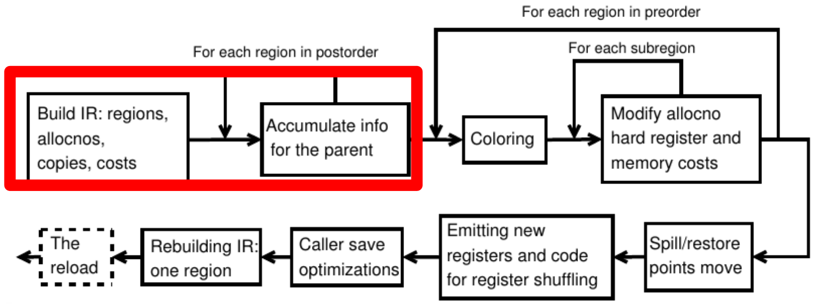
Cost

- 參考 Profile feed back 資訊
  - 沒 Profile 資訊 GCC 會自己猜機率 (predict.c)
    - [1] "Branch Prediction for Free" Ball and Larus; PLDI '93.
    - [2] "Static Branch Frequency and Program Profile Analysis" Wu and Larus; MICRO-27.
    - [3] "Corpus-based Static Branch Prediction" Calder, Grunwald, Lindsay, Martin, Mozer, and Zorn; PLDI '95. */
- 由上往下 (Top-Down) 的 Region 算下去


For each region in preorder
For each region in postorder
For each subregion
Build IR: regions, allocnos, copies, costs
Accumulate info for the parent
Coloring
Modify allocno hard register and memory costs
The reload
Rebuilding IR: one region
Caller save optimizations
Emitting new registers and code for register shuffling
Spill/restore points move

Preferred Register Class

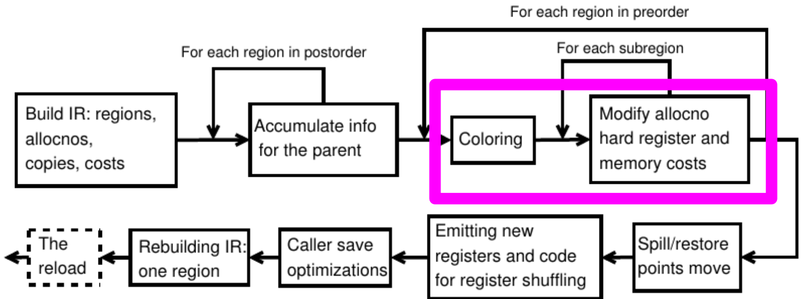
- 在記算 Cost 階段時也會計算每個 Allocno 所偏好的 Register Class
  - 參考 Register Preference

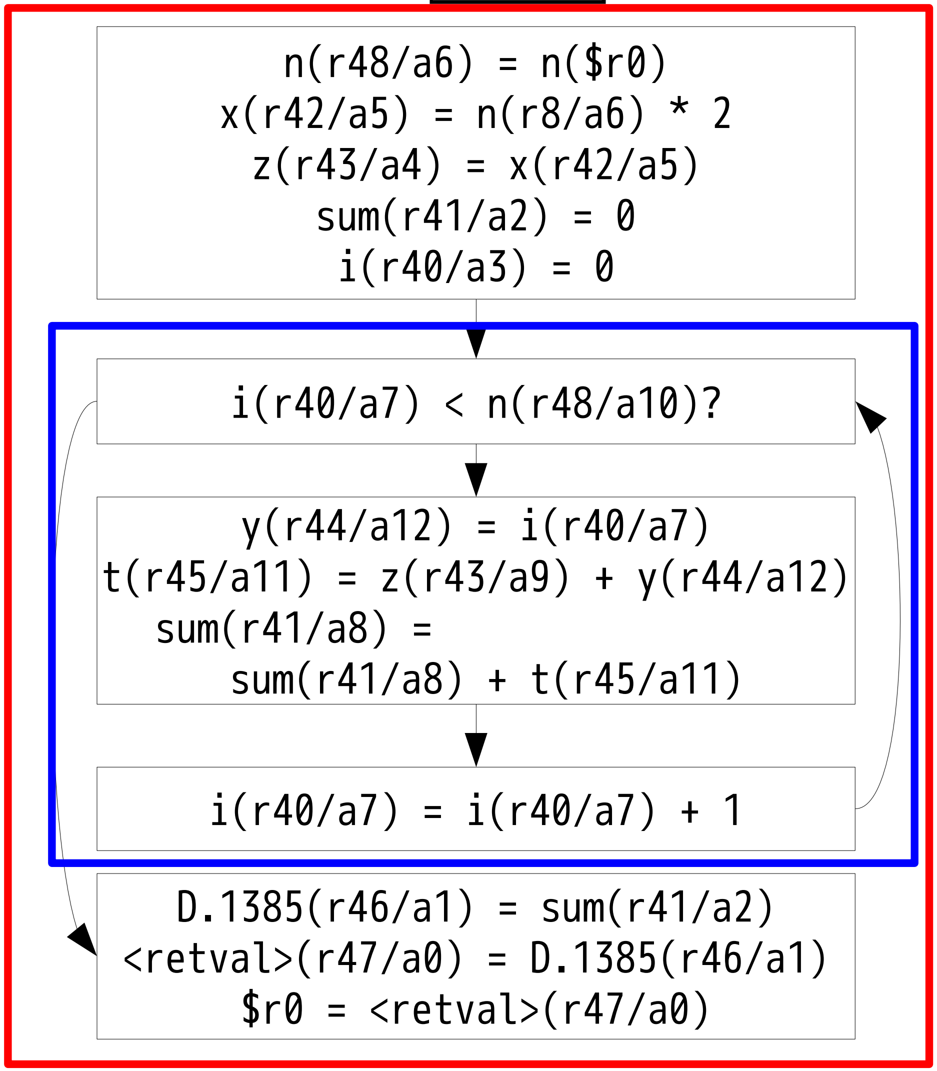

For each region in preorder
For each region in postorder
For each subregion
Build IR: regions, allocnos, copies, costs
Accumulate info for the parent
Coloring
Modify allocno hard register and memory costs
The reload
Rebuilding IR: one region
Caller save optimizations
Emitting new registers and code for register shuffling
Spill/restore points move

Preferred Register Class

- IRA 在這邊有作一些特殊處理
  - 以 x86 為例：
    - 若 Allocno a 有 8 use, 1 def: 若其中只有一個地方一定要 EBX, 而其它地方需要 EAX, 那該 Allocno 一樣會偏好 EAX
    - 需要 EBX 的地方則留給 Reload 處理
    - 在一般文獻中會採用所有使用到的 Register Class 的交集(Intersect)

Coloring Region 0

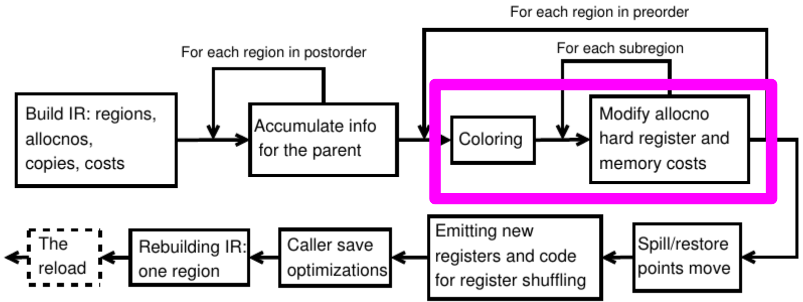

For each region in preorder
For each region in postorder
For each subregion
Build IR: regions, allocnos, copies, costs
Accumulate info for the parent
Coloring
Modify allocno hard register and memory costs
The reload
Rebuilding IR: one region
Caller save optimizations
Emitting new registers and code for register shuffling
Spill/restore points move



```
n(r48/a6) = n($r0)
x(r42/a5) = n(r8/a6) * 2
z(r43/a4) = x(r42/a5)
sum(r41/a2) = 0
i(r40/a3) = 0

i(r40/a7) < n(r48/a10)?

y(r44/a12) = i(r40/a7)
t(r45/a11) = z(r43/a9) + y(r44/a12)
sum(r41/a8) =
   sum(r41/a8) + t(r45/a11)

i(r40/a7) = i(r40/a7) + 1

D.1385(r46/a1) = sum(r41/a2)
<retval>(r47/a0) = D.1385(r46/a1)
$r0 = <retval>(r47/a0)
```

```
 Loop 0 (parent -1, header bb2, depth 0)
...
    Forming thread by copy 0:a1r46-a2r41 (freq=1):
      Result (freq=6): a1r46(2) a2r41(4)
    Forming thread by copy 1:a0r47-a1r46 (freq=1):
      Result (freq=8): a0r47(2) a1r46(2) a2r41(4)
    Pushing a5(r42,l0)(cost 0)
    Pushing a1(r46,l0)(cost 0)
    Pushing a0(r47,l0)(cost 0)
    Pushing a4(r43,l0)(potential spill: pri=3, cost=16)
      Making a13(r45,l0: a11(r45,l1)) colorable
    Forming thread by copy 4:a13r45-a14r44 (freq=1):
      Result (freq=4): a13r45(2) a14r44(2)
      Making a14(r44,l0: a12(r44,l1)) colorable
    Pushing a14(r44,l0: a12(r44,l1))(cost 16)
      Making a2(r41,l0) colorable
      Making a3(r40,l0) colorable
      Making a6(r48,l0) colorable
    Pushing a6(r48,l0)(cost 28)
    Pushing a13(r45,l0: a11(r45,l1))(cost 16)
    Pushing a3(r40,l0)(cost 40)
    Pushing a2(r41,l0)(cost 32)
    Popping a2(r41,l0)  -- assign reg 1
    Popping a3(r40,l0)  -- assign reg 2
    Popping a13(r45,l0: a11(r45,l1))  -- assign reg 3
    Popping a6(r48,l0)  -- assign reg 0
    Popping a14(r44,l0: a12(r44,l1))  -- assign reg 3
    Popping a4(r43,l0)  -- spill
    Popping a0(r47,l0)  -- assign reg 0
    Popping a1(r46,l0)  -- assign reg 0
    Popping a5(r42,l0)  -- assign reg 1
```

Coloring Region 0



For each region in preorder

For each region in postorder

For each subregion

Build IR: regions, allocnos, copies, costs

Accumulate info for the parent

Coloring

Modify allocno hard register and memory costs

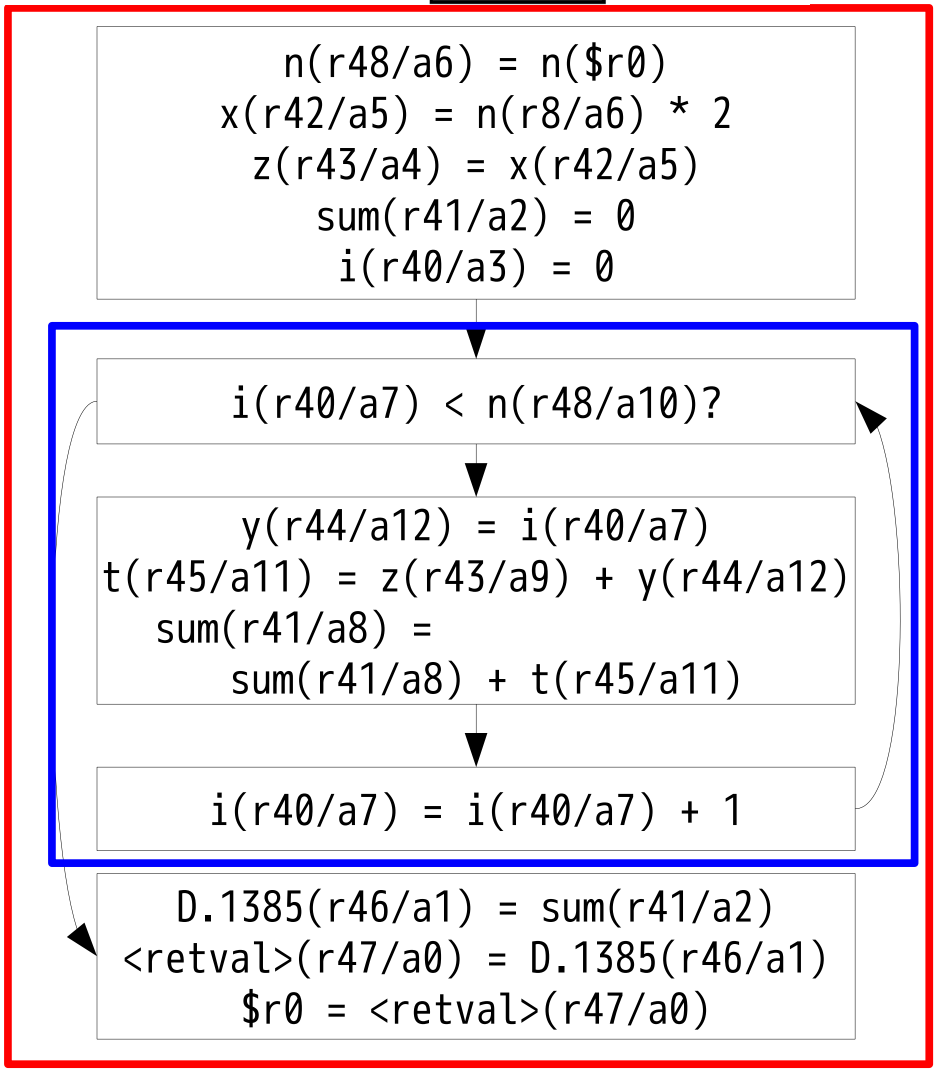
Spill/restore points move

Emitting new registers and code for register shuffling

Caller save optimizations

Rebuilding IR: one region

The reload



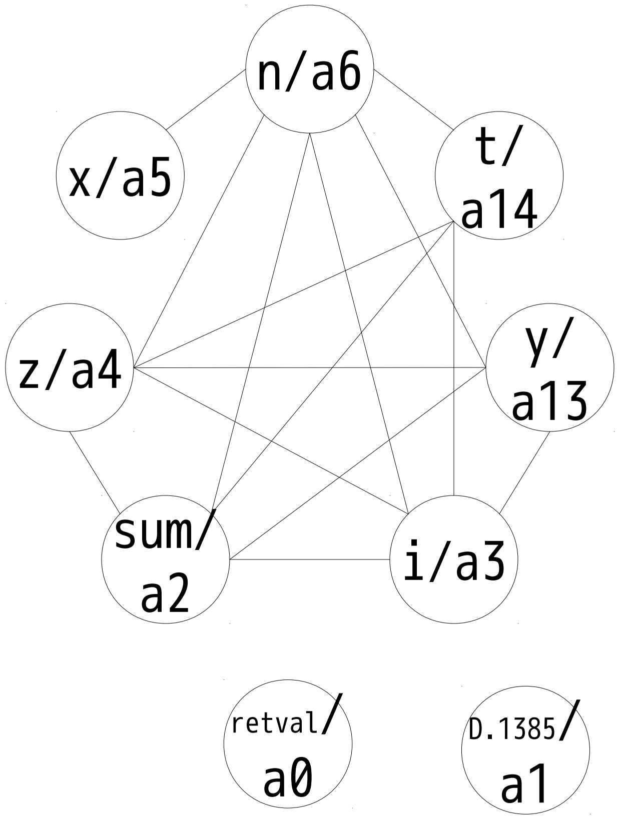
```
n(r48/a6) = n($r0)
x(r42/a5) = n(r8/a6) * 2
z(r43/a4) = x(r42/a5)
sum(r41/a2) = 0
i(r40/a3) = 0
```

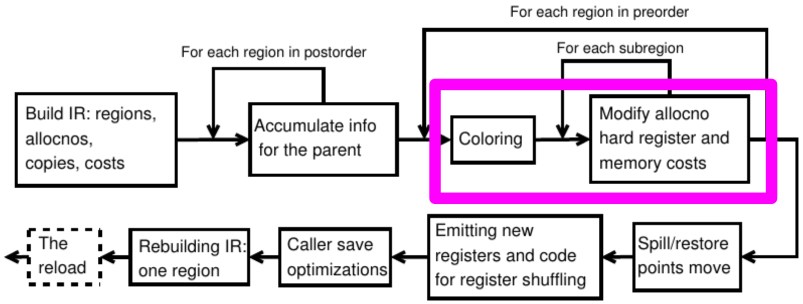
```
i(r40/a7) < n(r48/a10)?
```

```
y(r44/a12) = i(r40/a7)
t(r45/a11) = z(r43/a9) + y(r44/a12)
sum(r41/a8) =
    sum(r41/a8) + t(r45/a11)
```

```
i(r40/a7) = i(r40/a7) + 1
```

```
D.1385(r46/a1) = sum(r41/a2)
<retval>(r47/a0) = D.1385(r46/a1)
$r0 = <retval>(r47/a0)
```





For each region in preorder

For each region in postorder

For each subregion

Build IR: regions, allocnos, copies, costs

Accumulate info for the parent

Coloring

Modify allocno hard register and memory costs

The reload

Rebuilding IR: one region

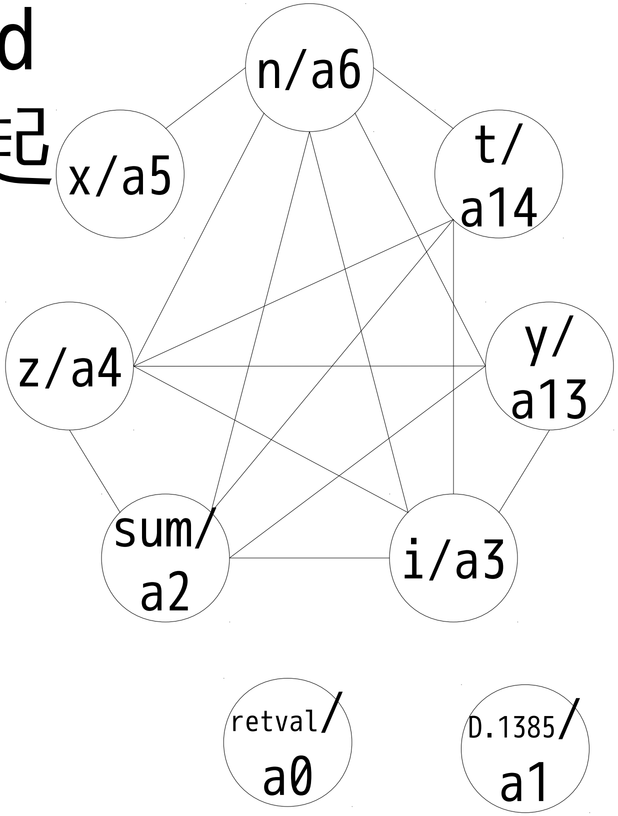
Caller save optimizations

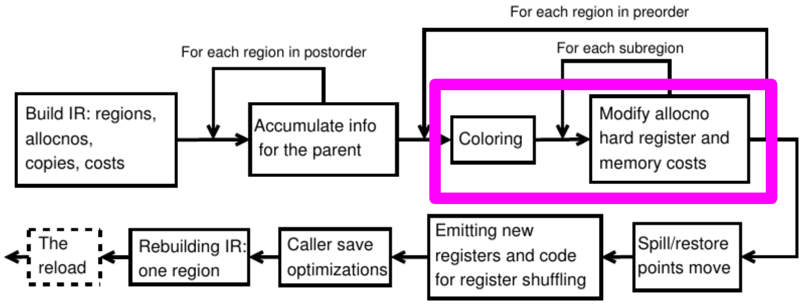
Emitting new registers and code for register shuffling

Spill/restore points move

Coloring Region 0

ira中coalesce並不會將點合併而是用一種稱為Thread的方式把copy相關的點一起推到 Stack





For each region in preorder

For each region in postorder

For each subregion

Build IR: regions, allocnos, copies, costs

Accumulate info for the parent

Coloring

Modify allocno hard register and memory costs

The reload

Rebuilding IR: one region

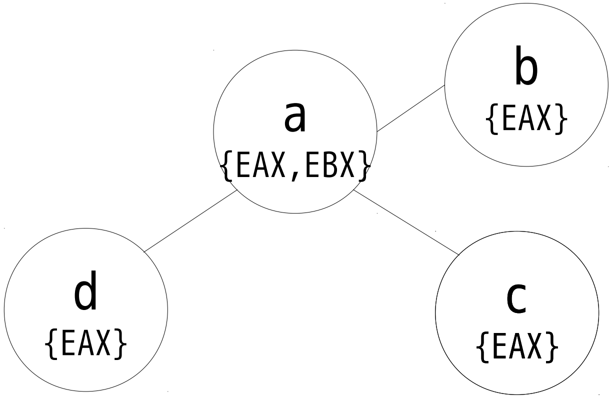
Caller save optimizations

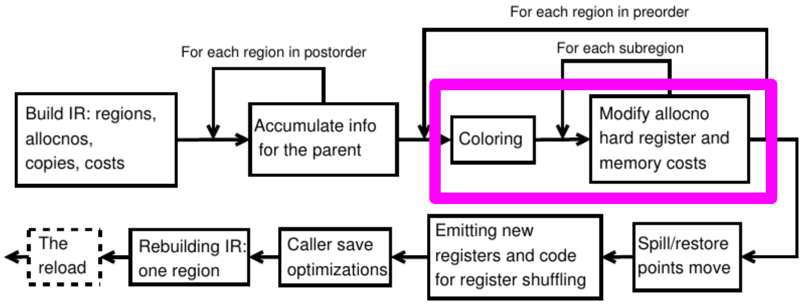
Emitting new registers and code for register shuffling

Spill/restore points move

Colorable

- 計算 Colorable 的方式並非計算**邊**數：
  - 以 x86 為例：
    - 若 Allocno a 可用 EAX 及 EBX, 且有 3 個邊，但其它相鄰的**點**都僅可使用 EAX， 這樣一樣是 Colorable





For each region in postorder

For each region in preorder

For each subregion

Build IR: regions, allocnos, copies, costs

Accumulate info for the parent

Coloring

Modify allocno hard register and memory costs

Spill/restore points move

Emitting new registers and code for register shuffling

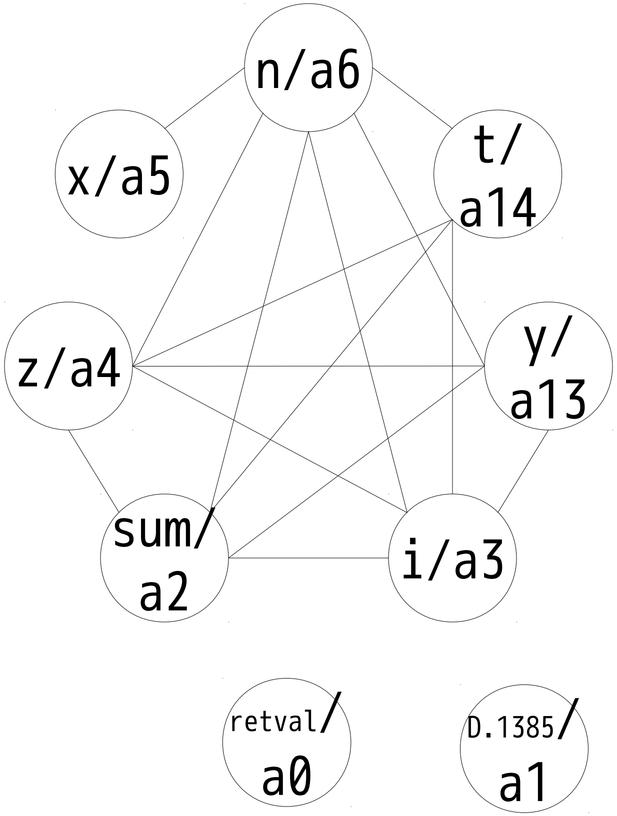
Caller save optimizations

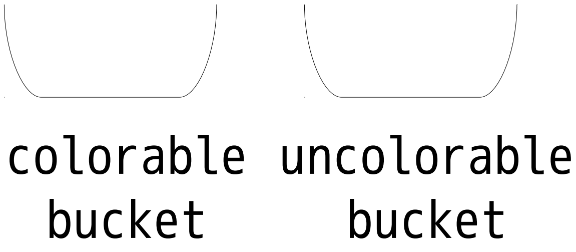
Rebuilding IR: one region

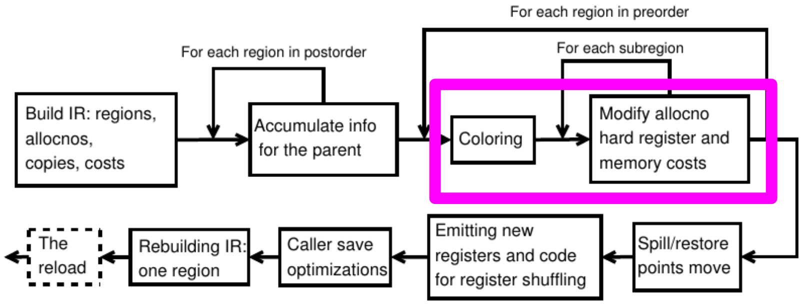
The reload

Coloring Region 0

IRA 會先依照初始的邊數
將所有 Allocno 分兩堆
可簡單上色的一堆（邊小於 4 的）
不可簡單上色的一堆（邊大於等於 4 的）







For each region in preorder

For each region in postorder

For each subregion

Build IR: regions, allocnos, copies, costs

Accumulate info for the parent

Coloring

Modify allocno hard register and memory costs

Spill/restore points move

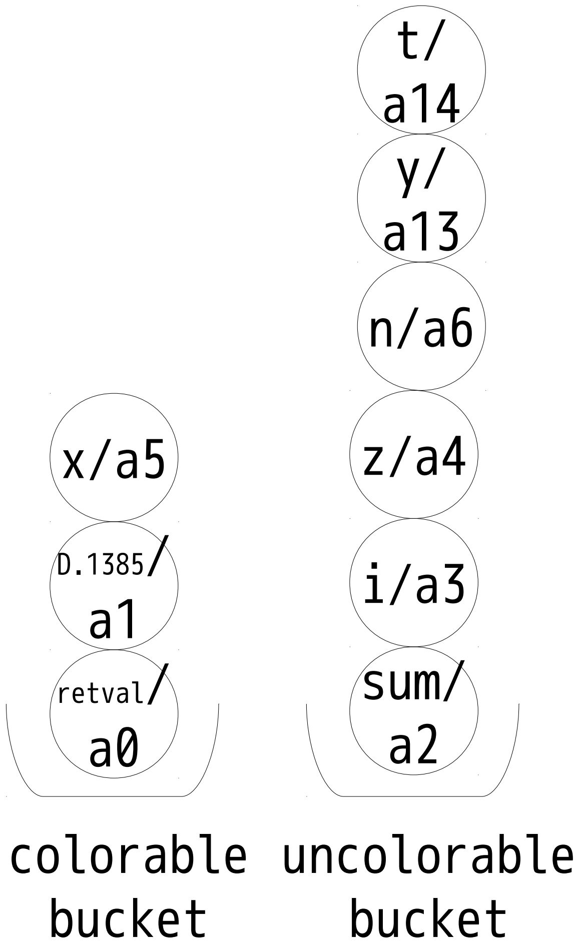
Emitting new registers and code for register shuffling

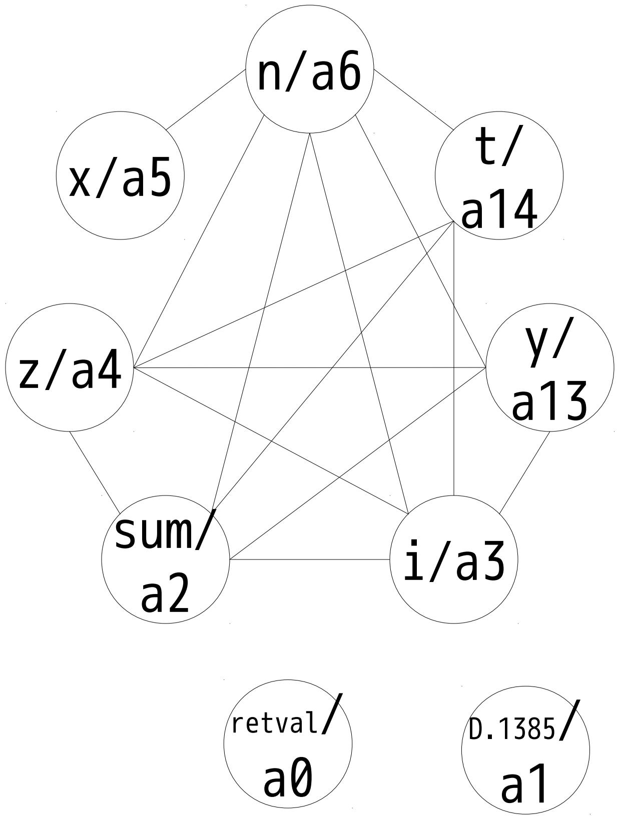
Caller save optimizations

Rebuilding IR: one region

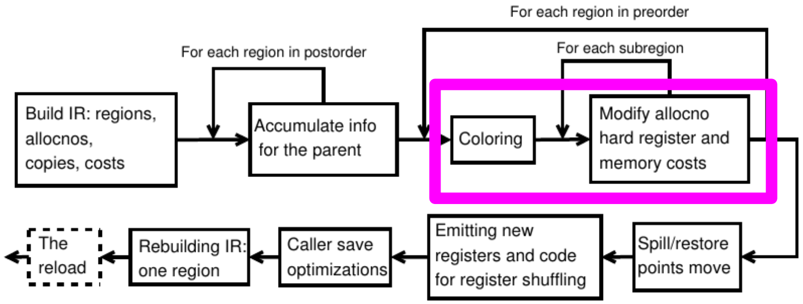
The reload

Coloring Region 0

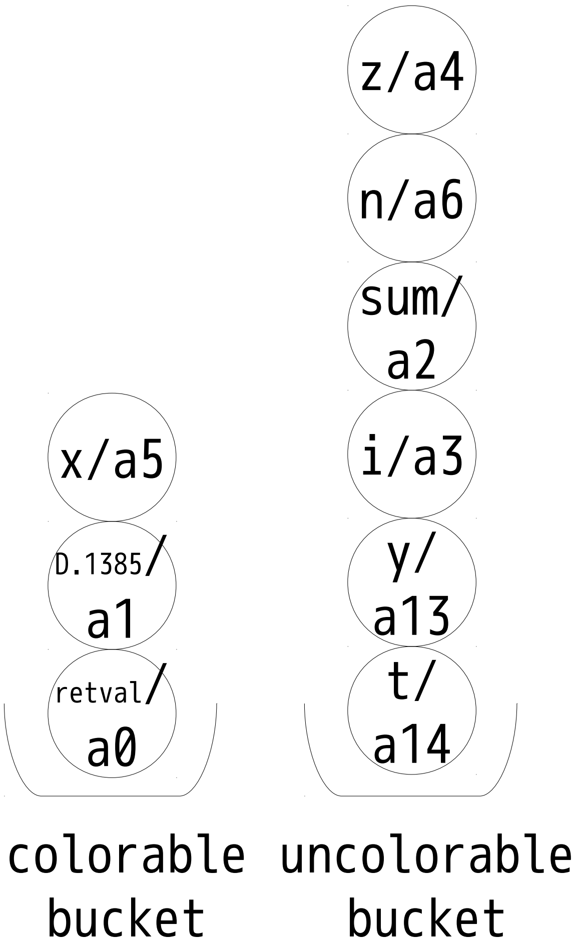


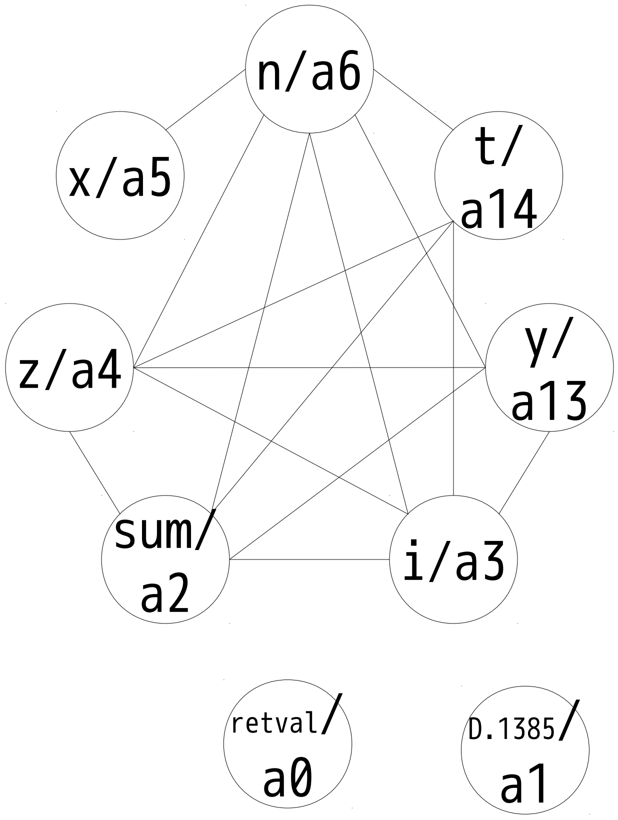


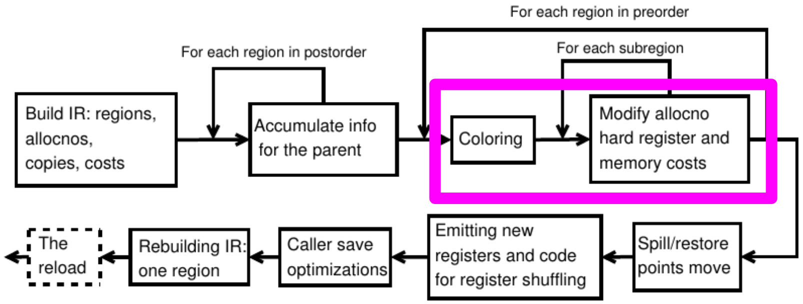
Coloring Region 0


For each region in preorder
For each region in postorder
For each subregion
Build IR: regions, allocnos, copies, costs
Accumulate info for the parent
Coloring
Modify allocno hard register and memory costs
The reload
Rebuilding IR: one region
Caller save optimizations
Emitting new registers and code for register shuffling
Spill/restore points move

sort uncolorable bucket by spill cost

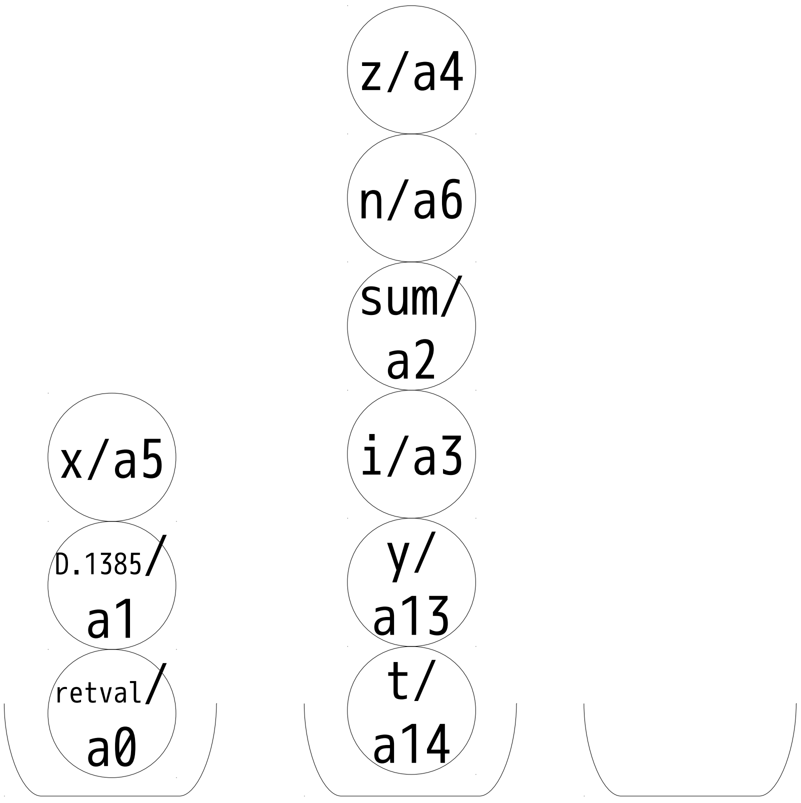





For each region in preorder
For each region in postorder
For each subregion
Build IR: regions, allocnos, copies, costs
Accumulate info for the parent
Coloring
Modify allocno hard register and memory costs
The reload
Rebuilding IR: one region
Caller save optimizations
Emitting new registers and code for register shuffling
Spill/restore points move

Coloring Region 0

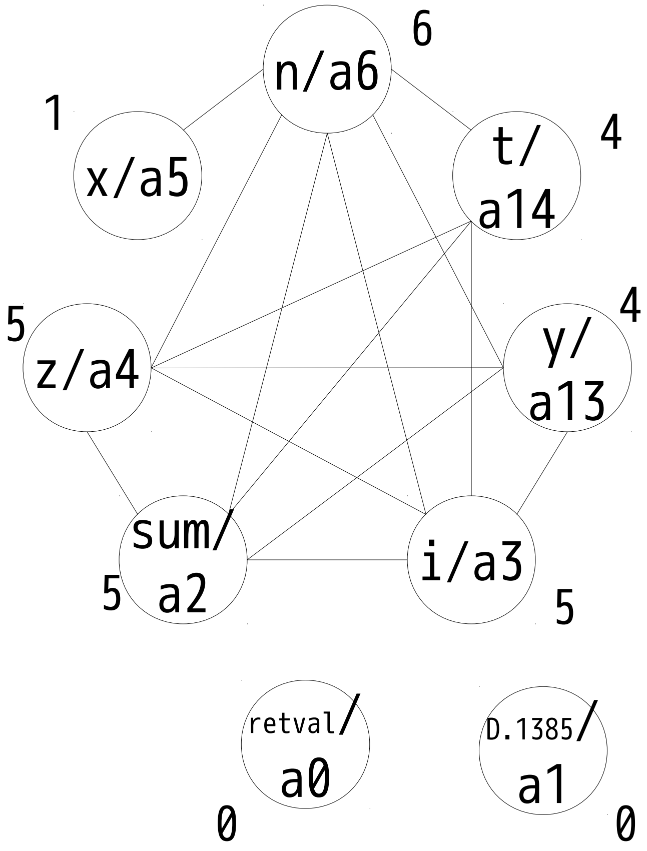
先將所有 Colorable bucket 的推進 stack

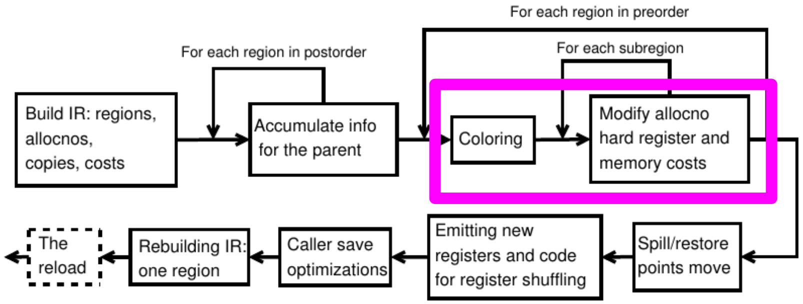


colorable bucket

uncolorable bucket

Stack




For each region in preorder

For each region in postorder

For each subregion

Build IR: regions, allocnos, copies, costs

Accumulate info for the parent

Coloring

Modify allocno hard register and memory costs

The reload

Rebuilding IR: one region

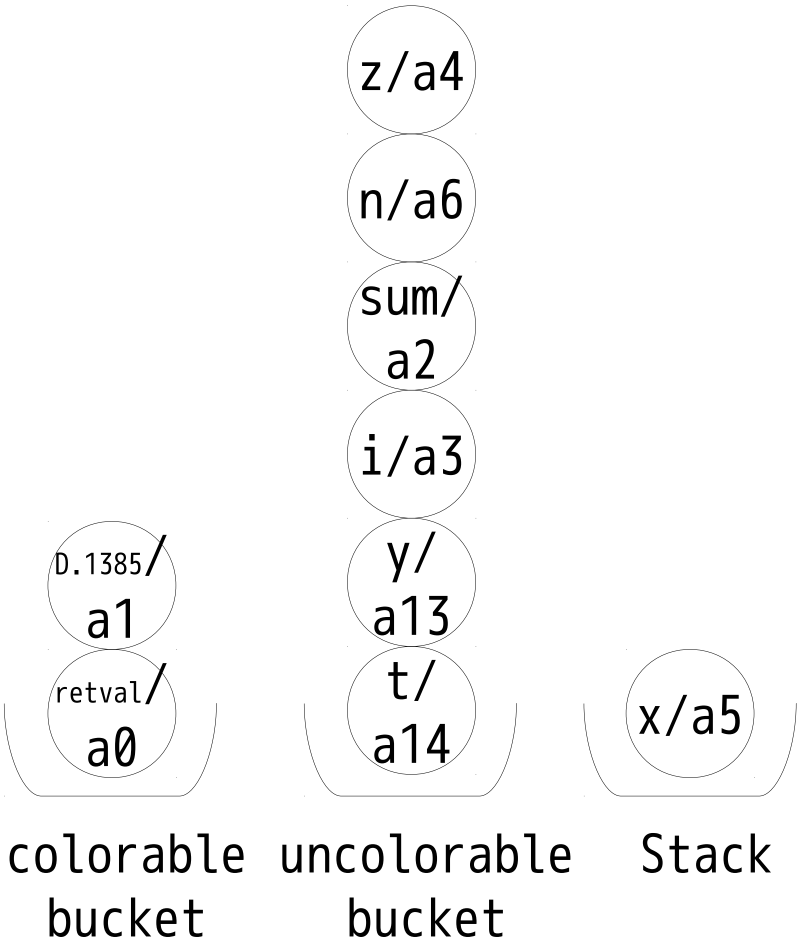
Caller save optimizations

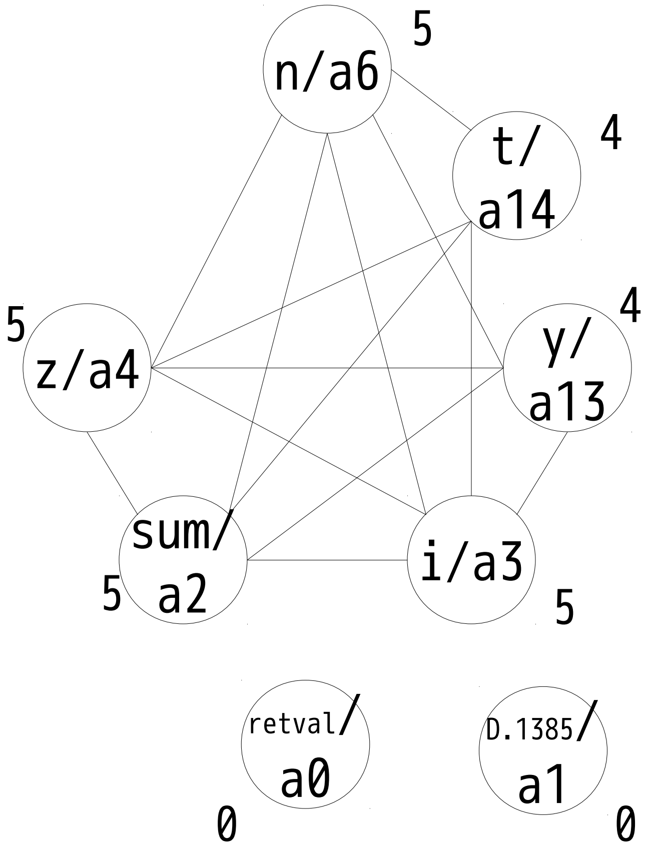
Emitting new registers and code for register shuffling

Spill/restore points move

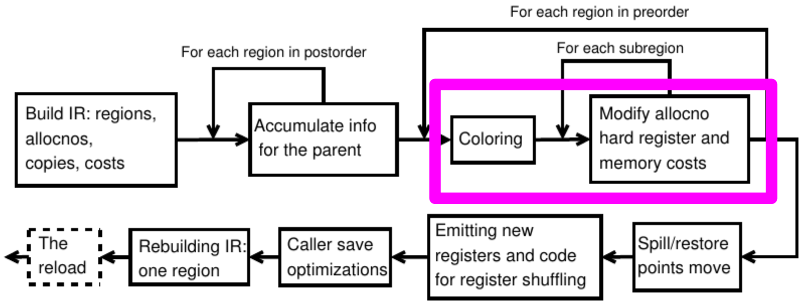
Coloring Region 0

先將所有 Colorable bucket 的推進 stack





Coloring Region 0



For each region in postorder

Build IR: regions, allocnos, copies, costs

Accumulate info for the parent

For each region in preorder

For each subregion

Coloring

Modify allocno hard register and memory costs

Spill/restore points move

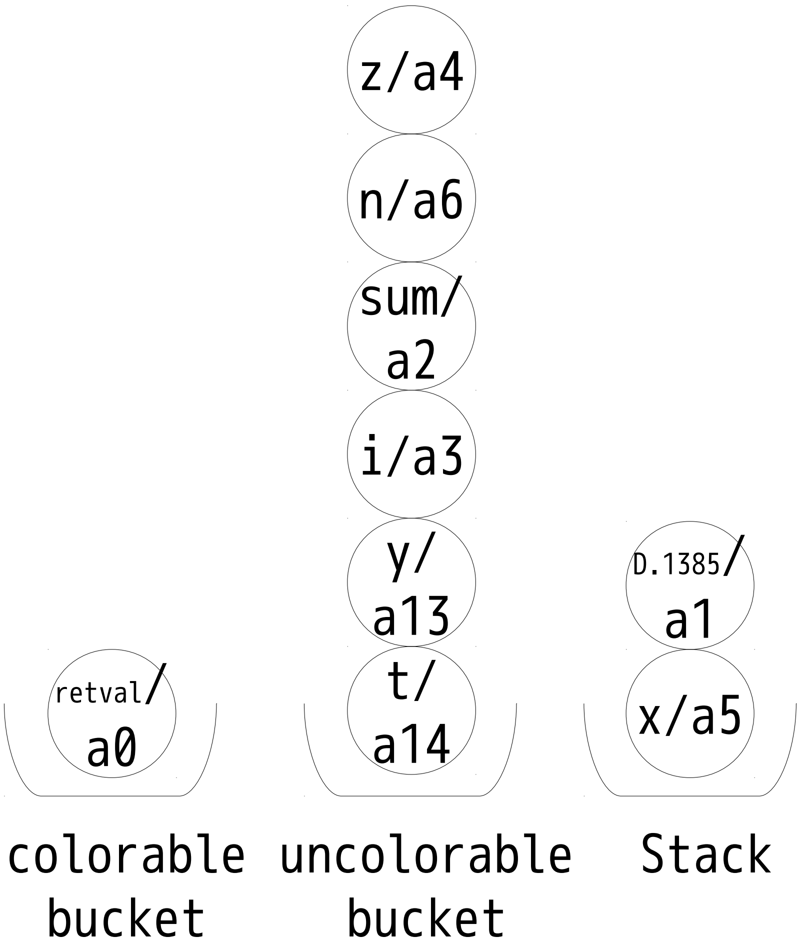
Emitting new registers and code for register shuffling

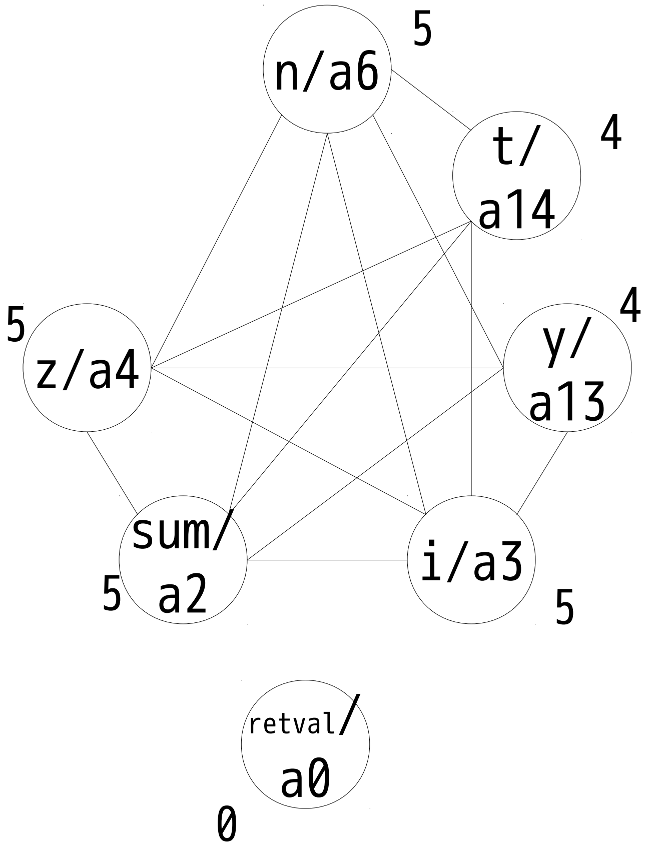
Caller save optimizations

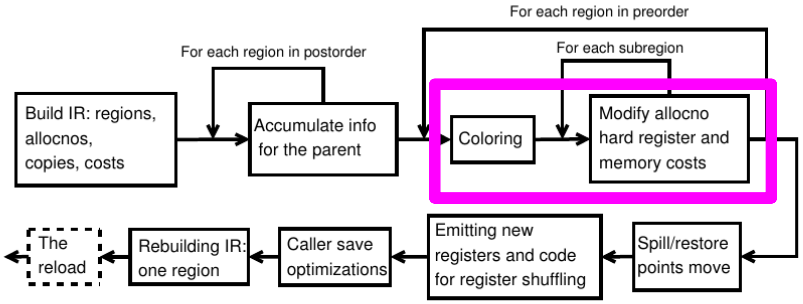
Rebuilding IR: one region

The reload

先將所有 Colorable bucket 的推進 stack






For each region in postorder

Build IR: regions, allocnos, copies, costs

Accumulate info for the parent

For each region in preorder

For each subregion

Coloring

Modify allocno hard register and memory costs

Spill/restore points move

Emitting new registers and code for register shuffling

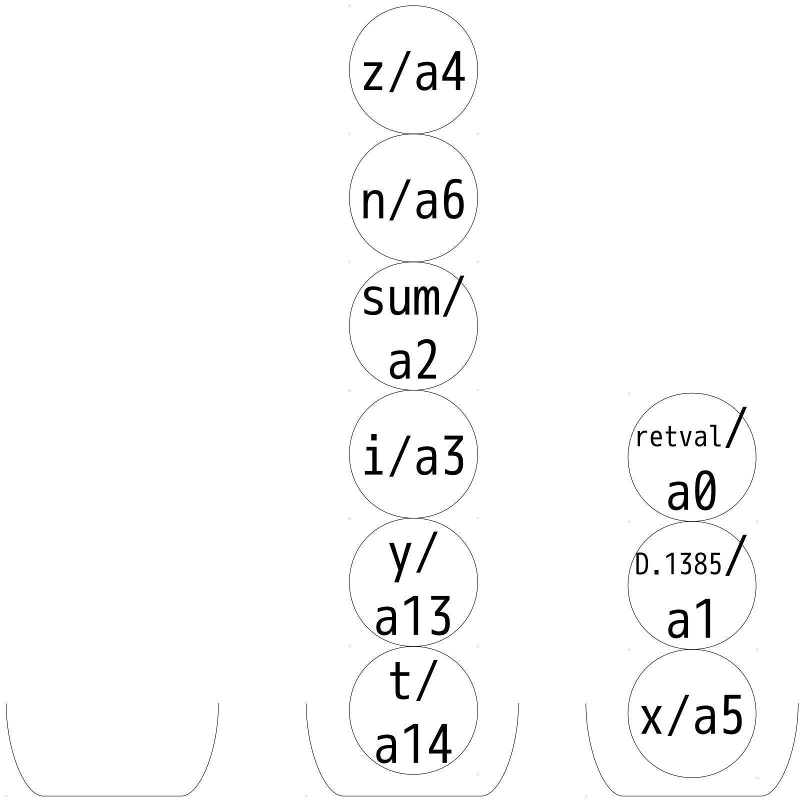
Caller save optimizations

Rebuilding IR: one region

The reload

Coloring Region 0

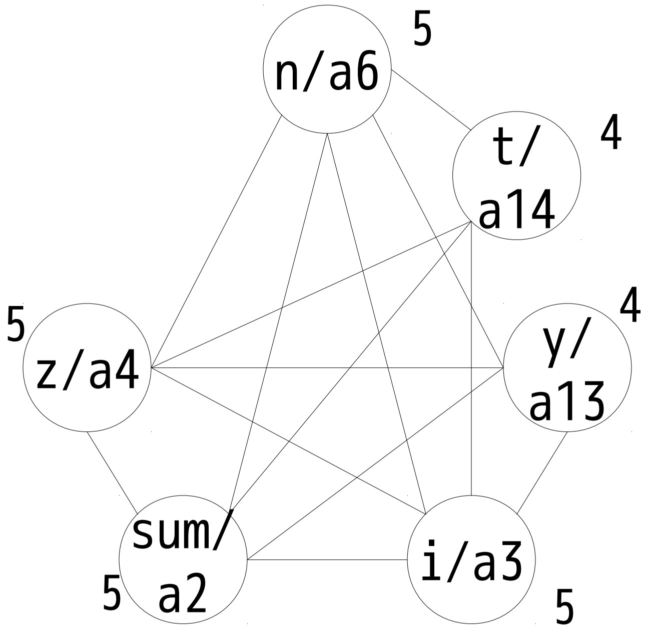
先將所有 Colorable bucket 的推進 stack



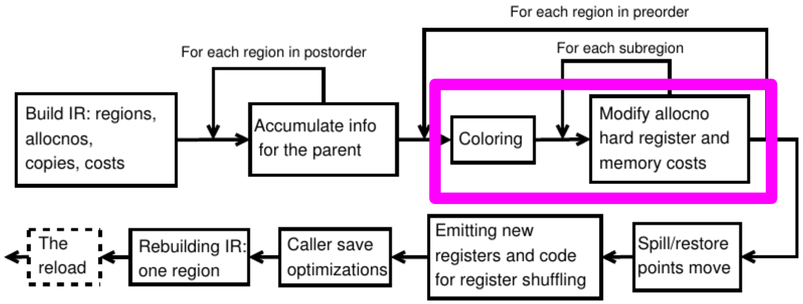
colorable bucket

uncolorable bucket

Stack



Coloring Region 0


For each region in postorder

For each region in preorder

For each subregion

Build IR: regions, allocnos, copies, costs

Accumulate info for the parent

Coloring

Modify allocno hard register and memory costs

Spill/restore points move

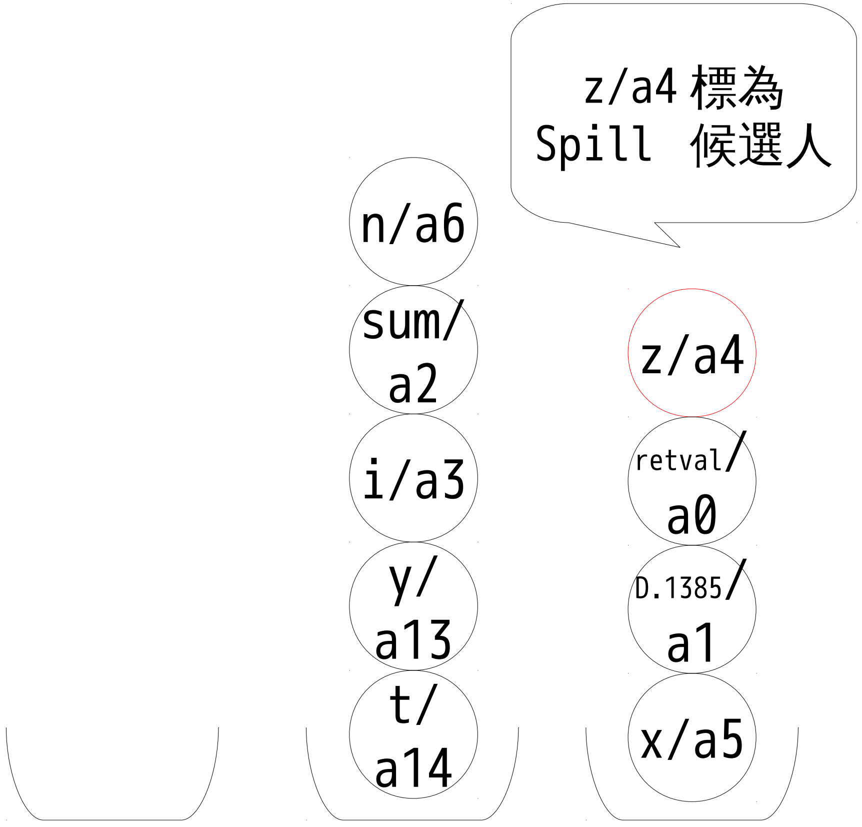
Emitting new registers and code for register shuffling

Caller save optimizations

Rebuilding IR: one region

The reload

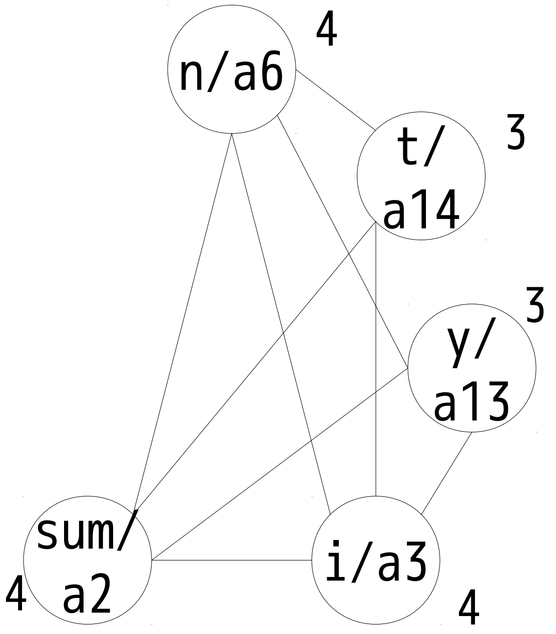
先將所有 Colorable bucket 的推進 stack


z/a4 標為 Spill 候選人

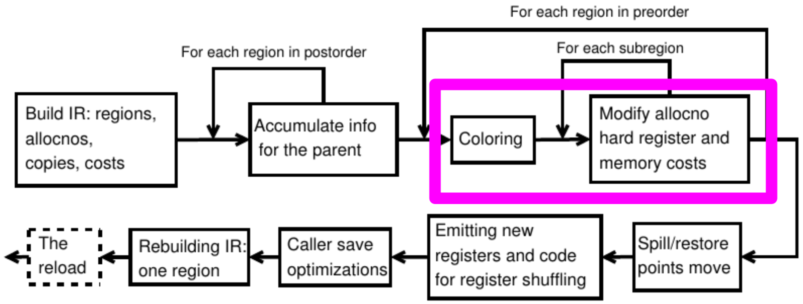
colorable bucket

uncolorable bucket

Stack



Coloring Region 0



For each region in preorder

For each region in postorder

For each subregion

Build IR: regions, allocnos, copies, costs

Accumulate info for the parent

Coloring

Modify allocno hard register and memory costs

The reload

Rebuilding IR: one region

Caller save optimizations

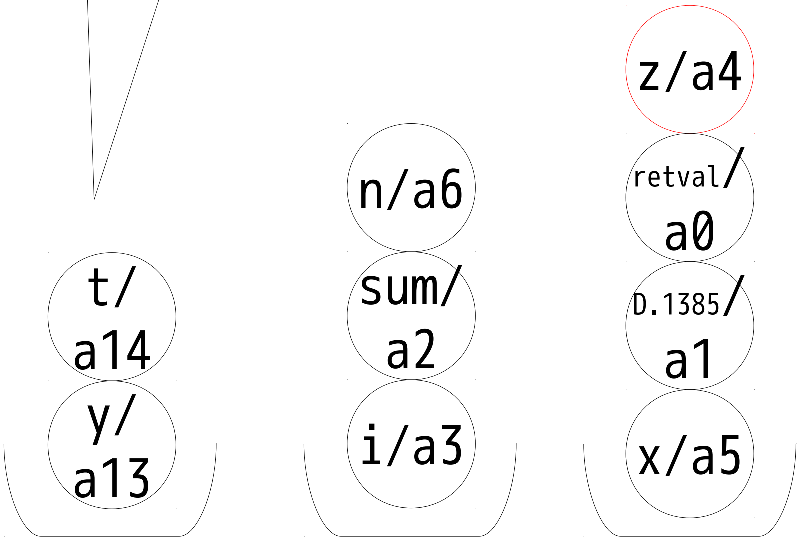
Emitting new registers and code for register shuffling

Spill/restore points move

先將所有 Colorable bucket 的推進 stack

y/a13 及 t/a14 因此變成 colorable

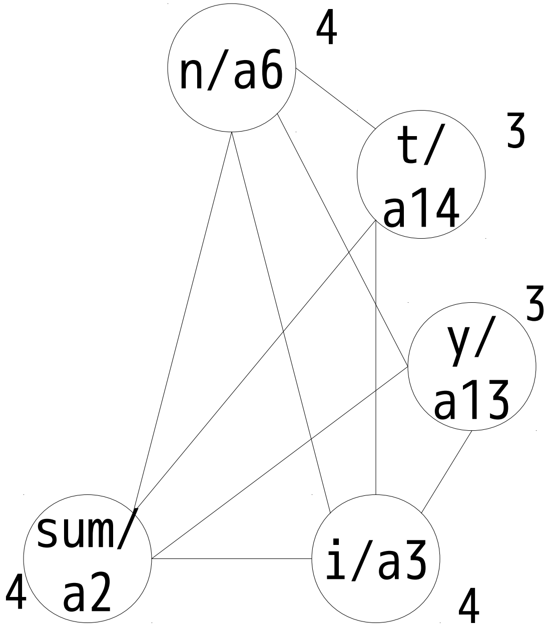
z/a4 標為 Spill 候選人

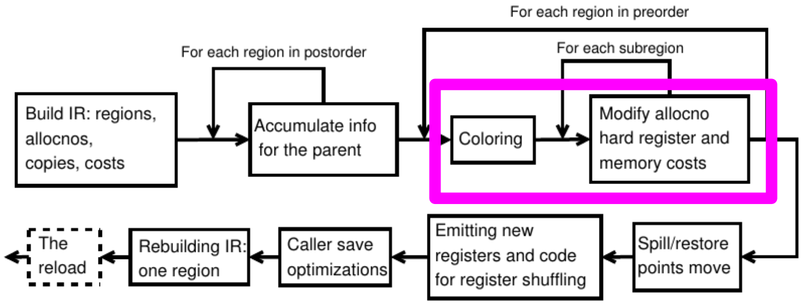


colorable bucket

uncolorable bucket

Stack





For each region in preorder

For each region in postorder

For each subregion

Build IR: regions, allocnos, copies, costs

Accumulate info for the parent

Coloring

Modify allocno hard register and memory costs

The reload

Rebuilding IR: one region

Caller save optimizations

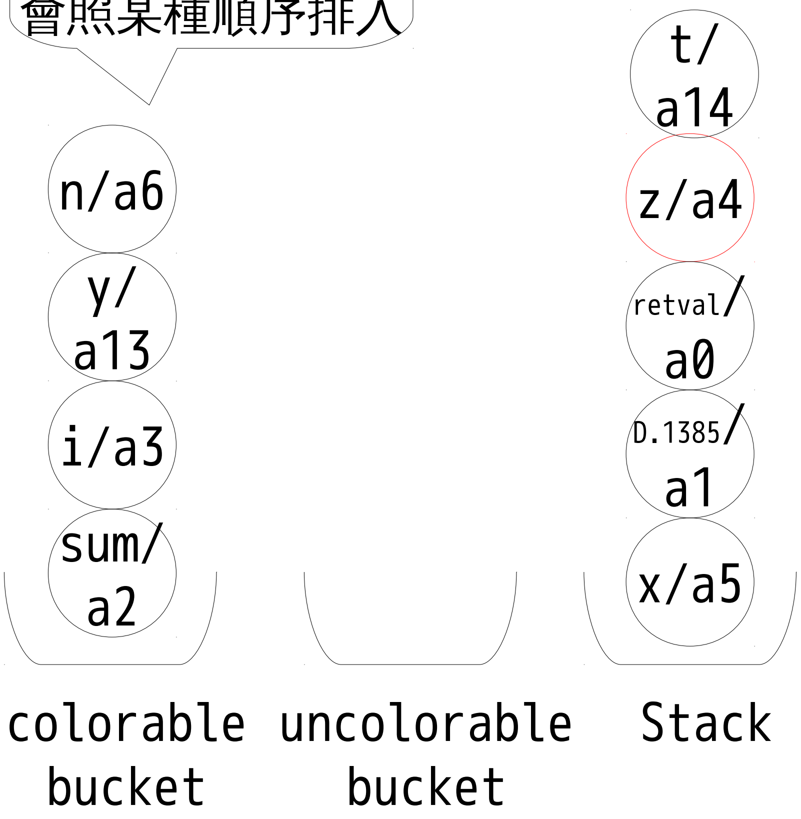
Emitting new registers and code for register shuffling

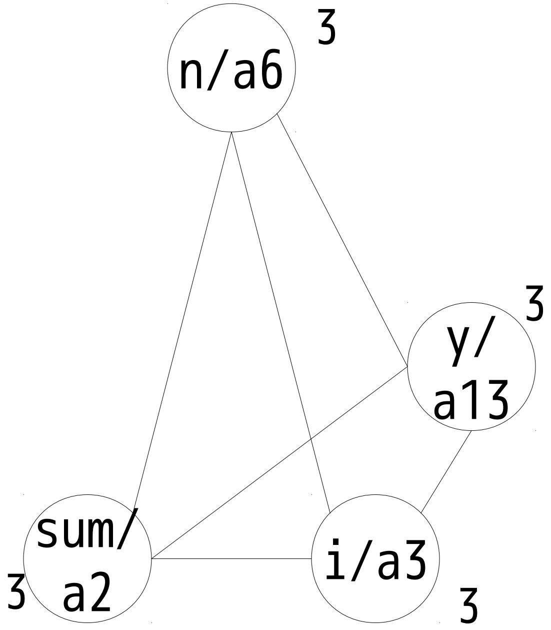
Spill/restore points move

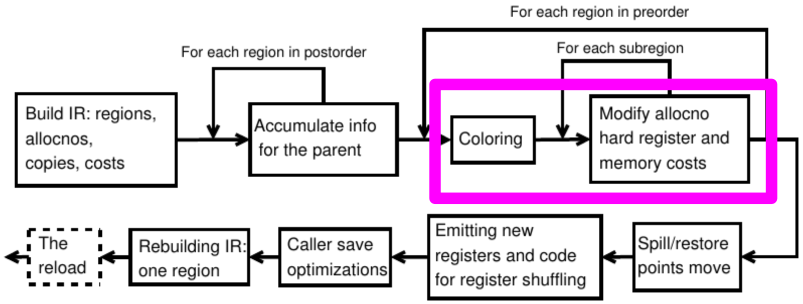
Coloring Region 0

先將所有 Colorable bucket 的推進 stack

uncolorable
移到 colorable
會照某種順序排入






For each region in preorder

For each region in postorder

For each subregion

Build IR: regions, allocnos, copies, costs

Accumulate info for the parent

Coloring

Modify allocno hard register and memory costs

Spill/restore points move

Emitting new registers and code for register shuffling

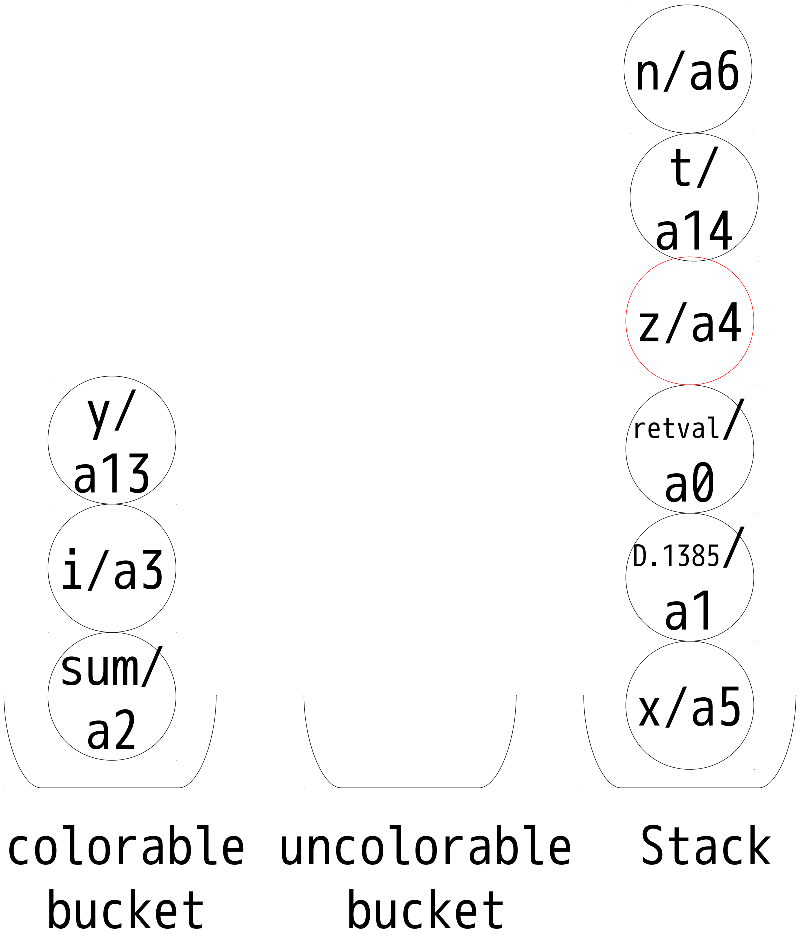
Caller save optimizations

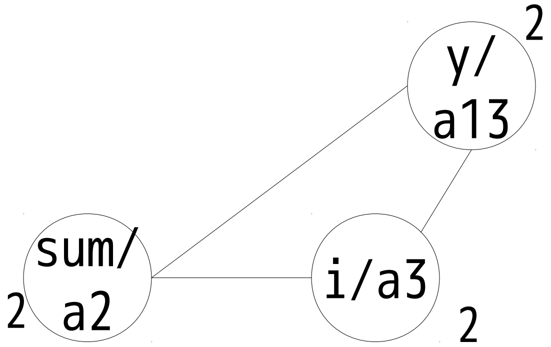
Rebuilding IR: one region

The reload

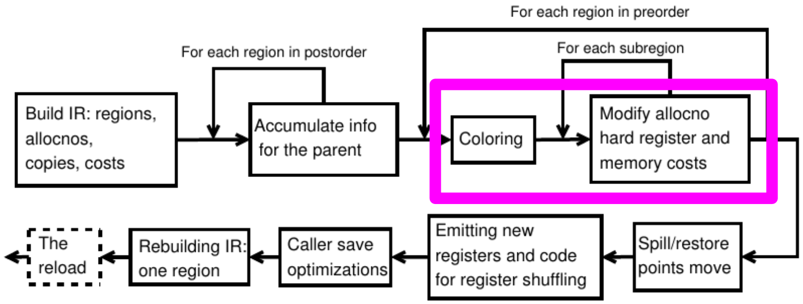
Coloring Region 0

先將所有 Colorable bucket 的推進 stack

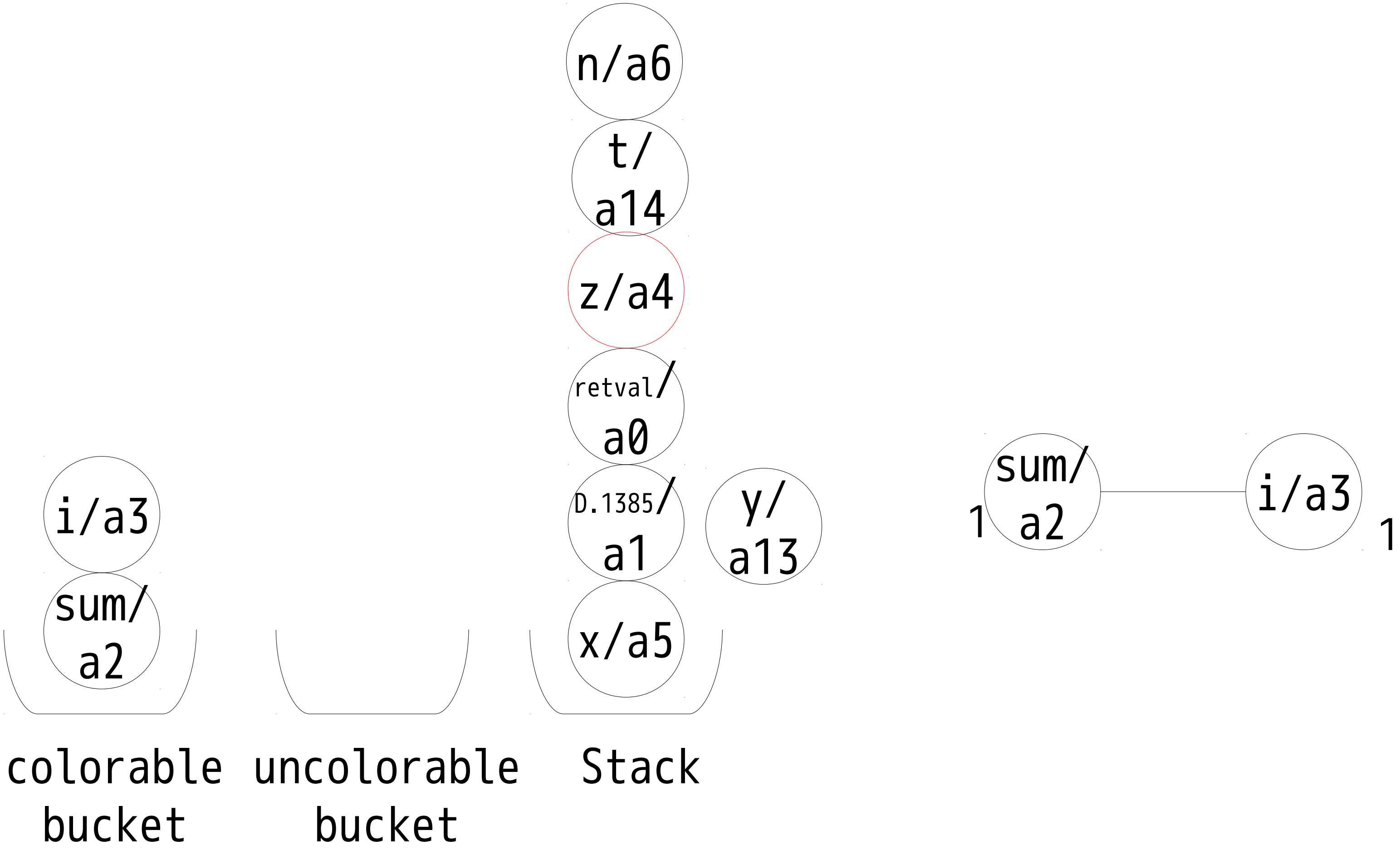


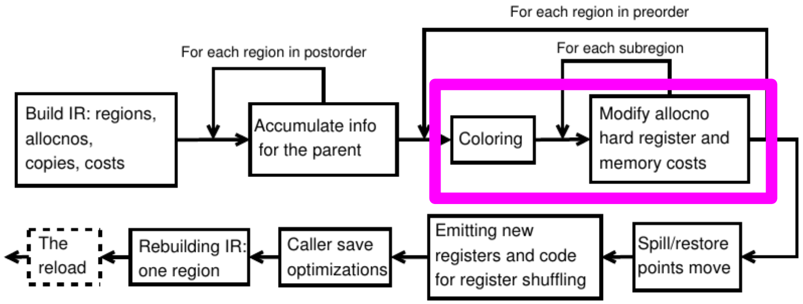


Coloring Region 0


For each region in preorder
For each region in postorder
For each subregion
Build IR: regions, allocnos, copies, costs
Accumulate info for the parent
Coloring
Modify allocno hard register and memory costs
The reload
Rebuilding IR: one region
Caller save optimizations
Emitting new registers and code for register shuffling
Spill/restore points move

先將所有 Colorable bucket 的推進 stack





For each region in postorder

For each region in preorder

For each subregion

Build IR: regions, allocnos, copies, costs

Accumulate info for the parent

Coloring

Modify allocno hard register and memory costs

Spill/restore points move

Emitting new registers and code for register shuffling

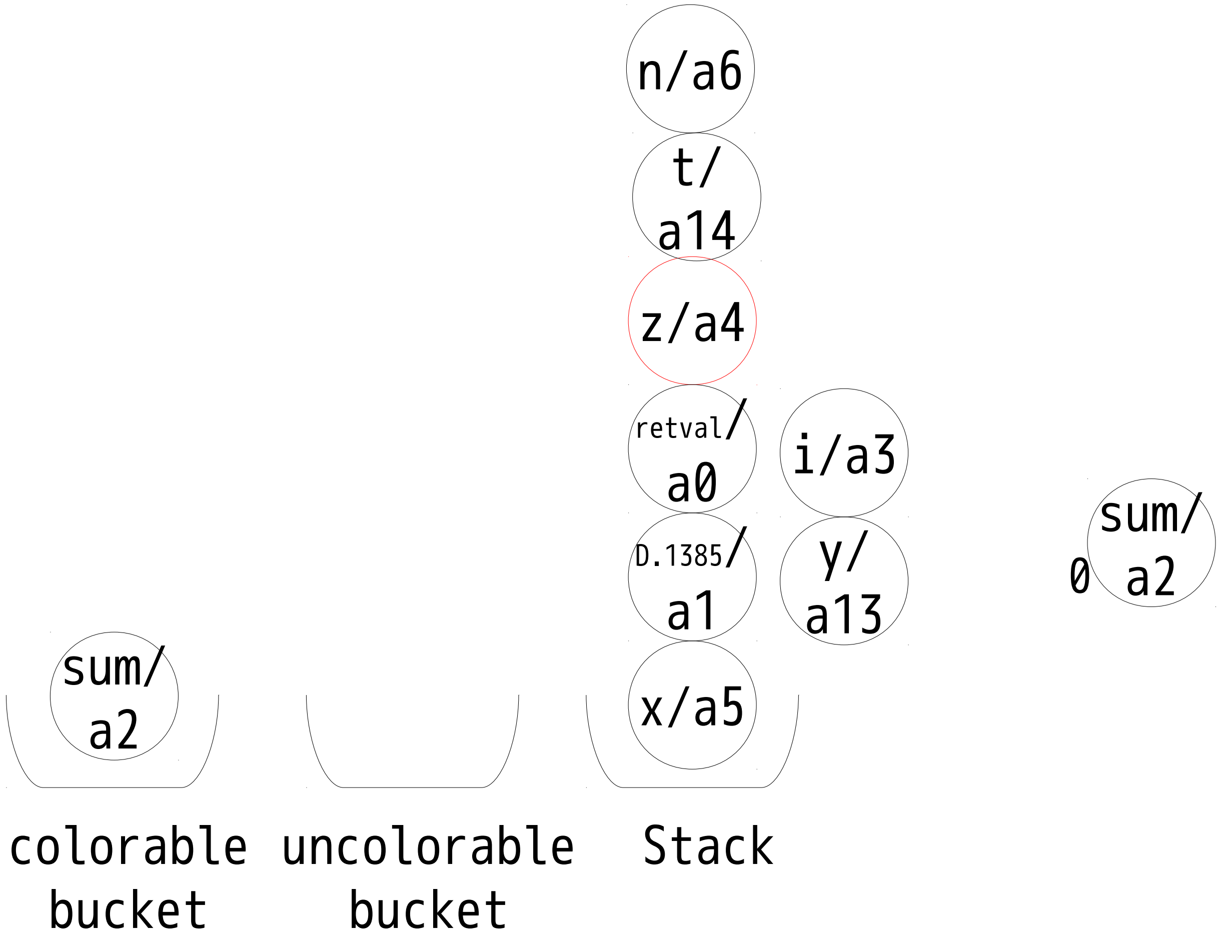
Caller save optimizations

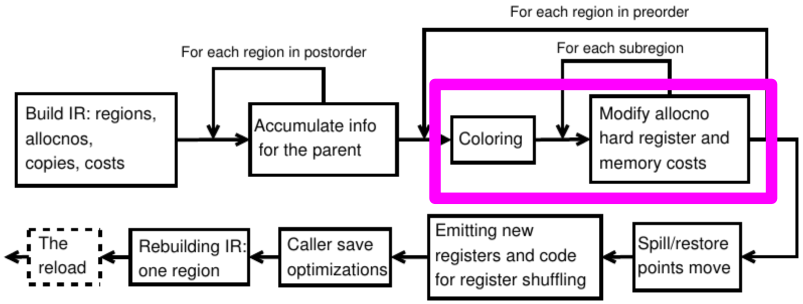
Rebuilding IR: one region

The reload

Coloring Region 0

先將所有 Colorable bucket 的推進 stack

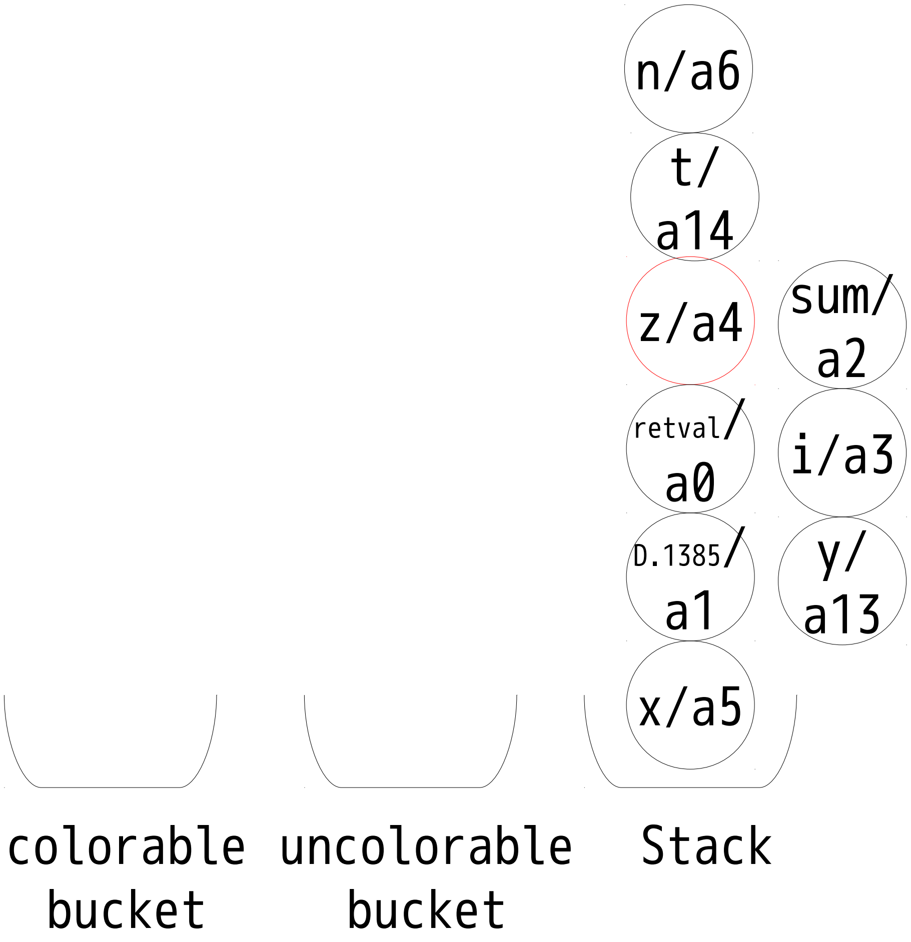




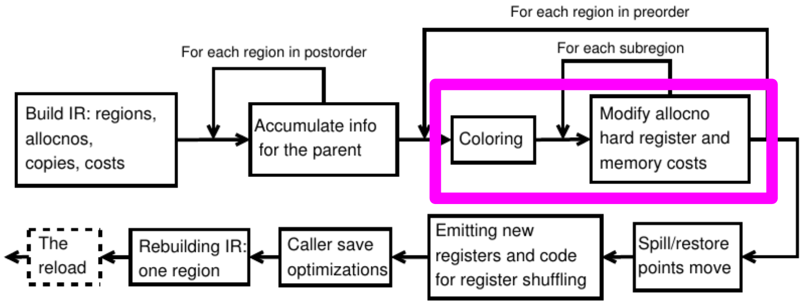
For each region in preorder
For each region in postorder
For each subregion
Build IR: regions, allocnos, copies, costs
Accumulate info for the parent
Coloring
Modify allocno hard register and memory costs
The reload
Rebuilding IR: one region
Caller save optimizations
Emitting new registers and code for register shuffling
Spill/restore points move

Coloring Region 0

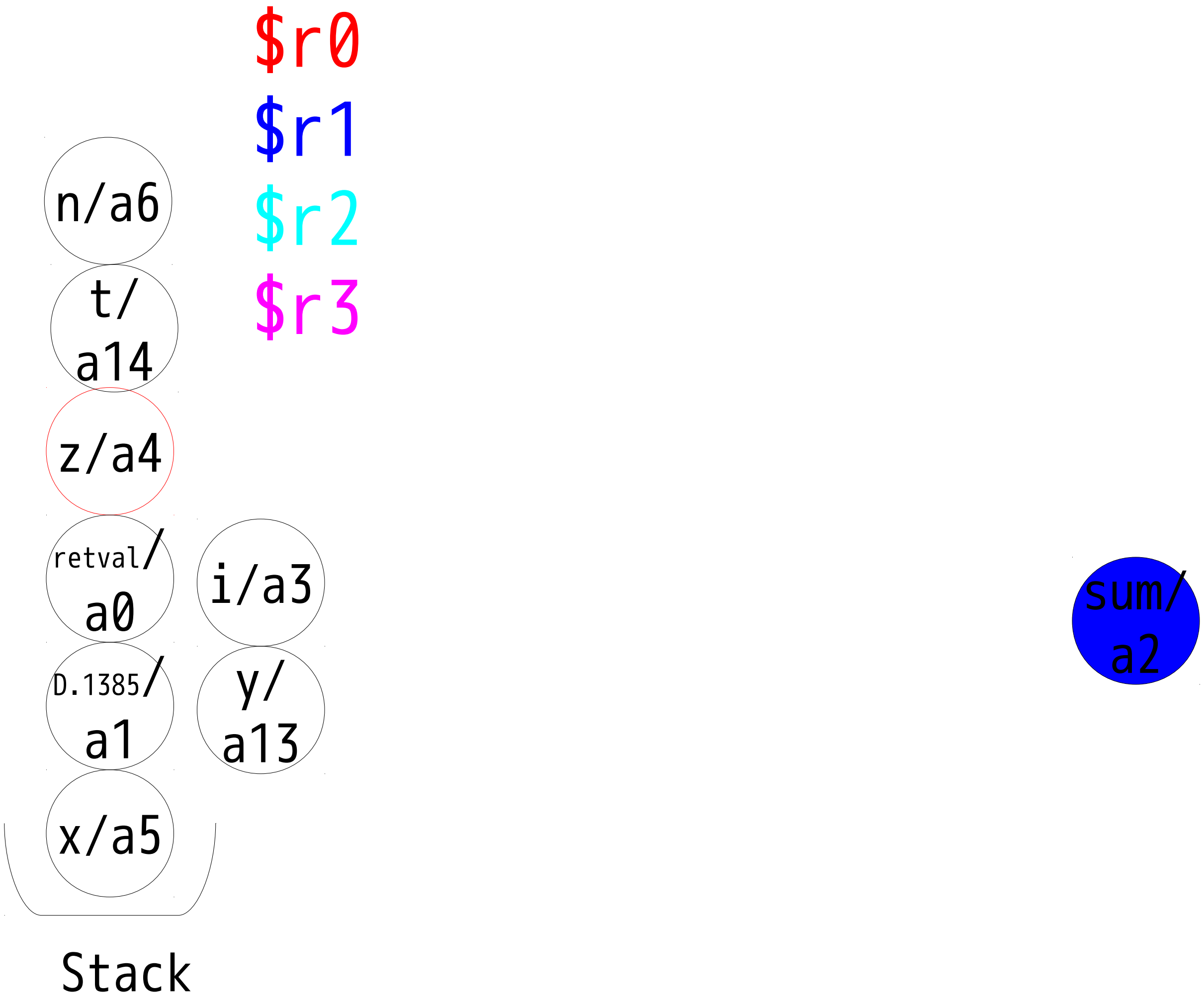
先將所有 Colorable bucket 的推進 stack



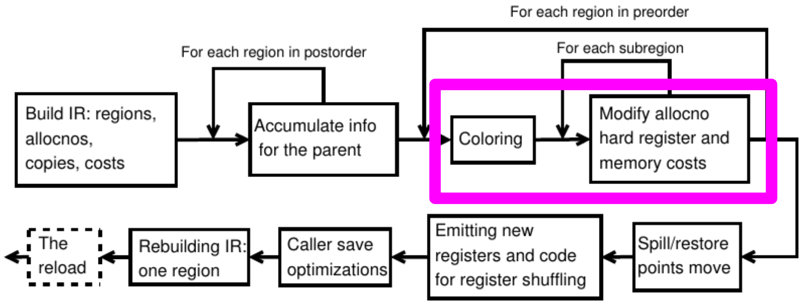
Coloring Region 0



For each region in preorder
For each region in postorder
For each subregion
Build IR: regions, allocnos, copies, costs
Accumulate info for the parent
Coloring
Modify allocno hard register and memory costs
The reload
Rebuilding IR: one region
Caller save optimizations
Emitting new registers and code for register shuffling
Spill/restore points move



Coloring Region 0

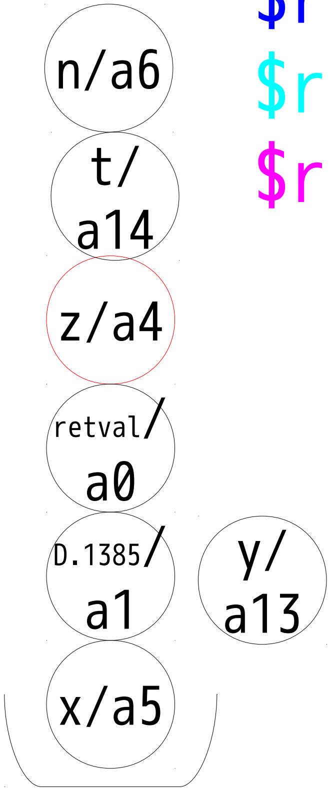

For each region in preorder
For each region in postorder
For each subregion
Build IR: regions, allocnos, copies, costs
Accumulate info for the parent
Coloring
Modify allocno hard register and memory costs
The reload
Rebuilding IR: one region
Caller save optimizations
Emitting new registers and code for register shuffling
Spill/restore points move

$r0

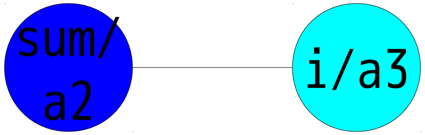
$r1

$r2

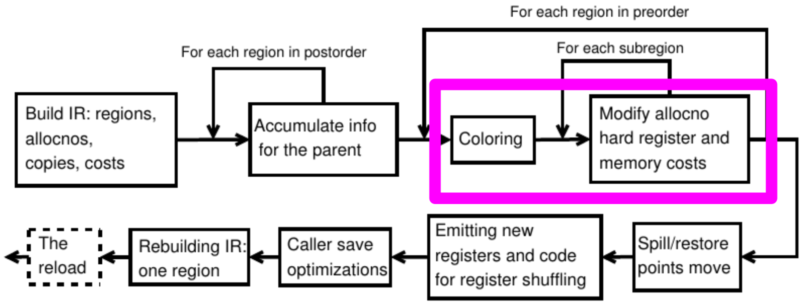
$r3



Stack



Coloring Region 0



For each region in preorder

For each region in postorder

For each subregion

Build IR: regions, allocnos, copies, costs

Accumulate info for the parent

Coloring

Modify allocno hard register and memory costs

Spill/restore points move

Emitting new registers and code for register shuffling

Caller save optimizations

Rebuilding IR: one region

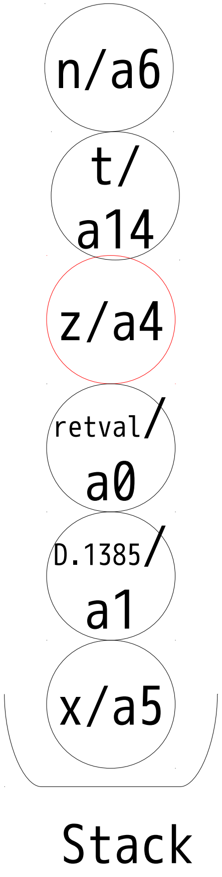
The reload

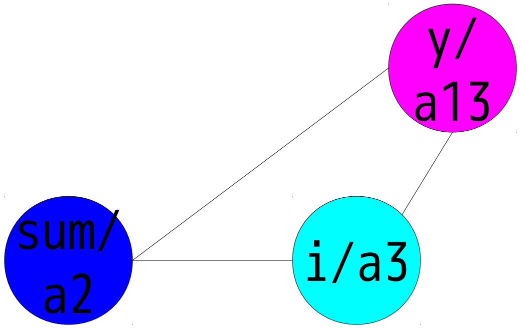
$r0

$r1

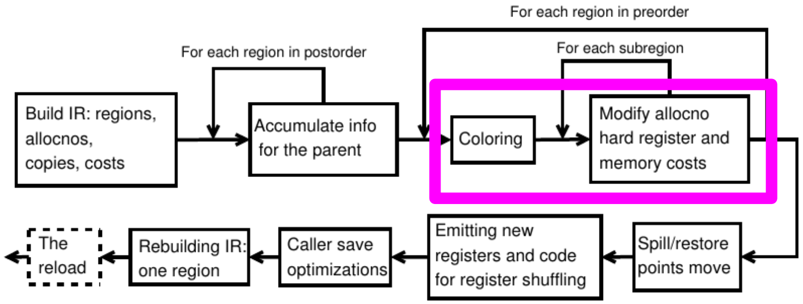
$r2

$r3





Coloring Region 0



For each region in preorder

For each region in postorder

For each subregion

Build IR: regions, allocnos, copies, costs

Accumulate info for the parent

Coloring

Modify allocno hard register and memory costs

The reload

Rebuilding IR: one region

Caller save optimizations

Emitting new registers and code for register shuffling

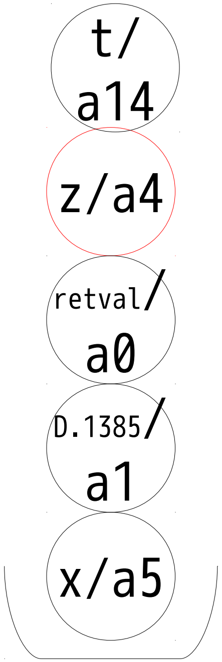
Spill/restore points move

$r0

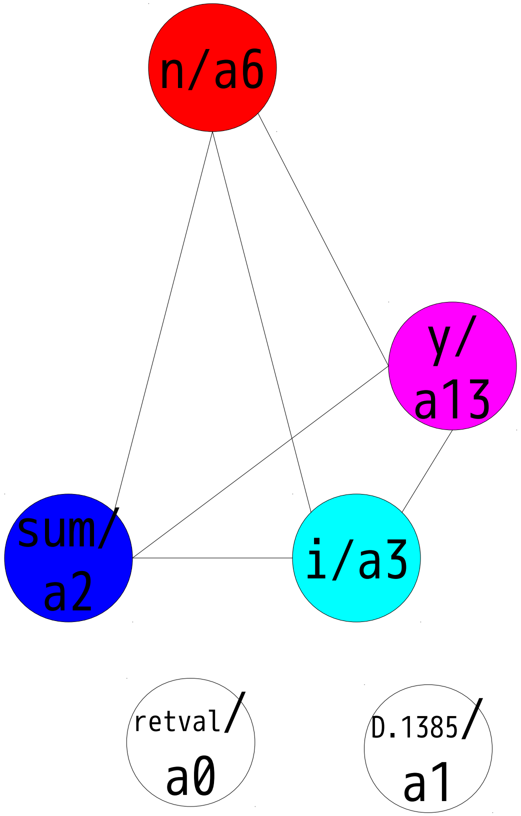
$r1

$r2

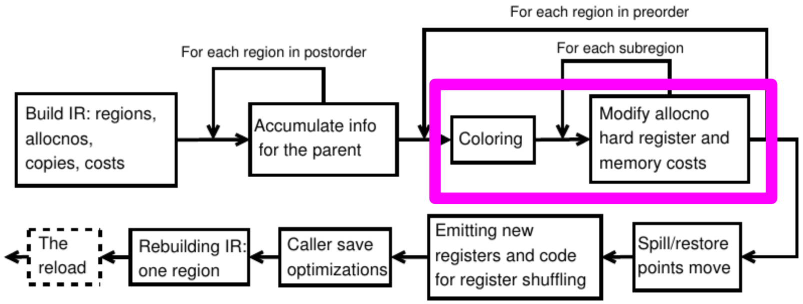
$r3



Stack



Coloring Region 0



For each region in preorder

For each region in postorder

For each subregion

Build IR: regions, allocnos, copies, costs

Accumulate info for the parent

Coloring

Modify allocno hard register and memory costs

The reload

Rebuilding IR: one region

Caller save optimizations

Emitting new registers and code for register shuffling

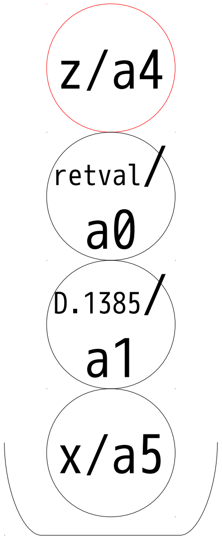
Spill/restore points move

$r0

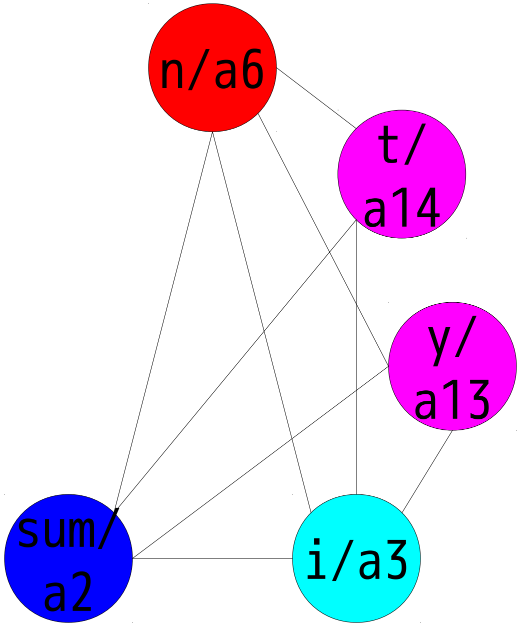
$r1

$r2

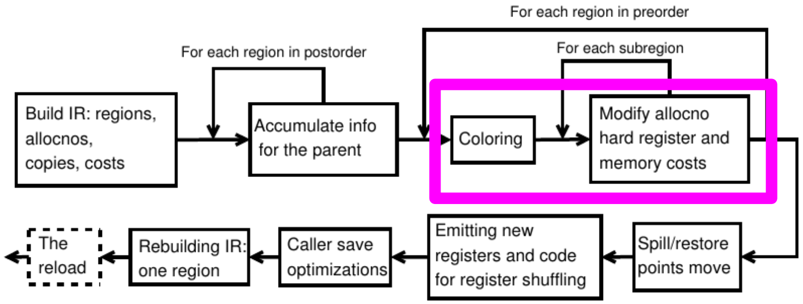
$r3



Stack



Coloring Region 0


Build IR: regions, allocnos, copies, costs

For each region in postorder

Accumulate info for the parent

For each region in preorder

For each subregion

Coloring

Modify allocno hard register and memory costs

Spill/restore points move

Emitting new registers and code for register shuffling

Caller save optimizations

Rebuilding IR: one region

The reload

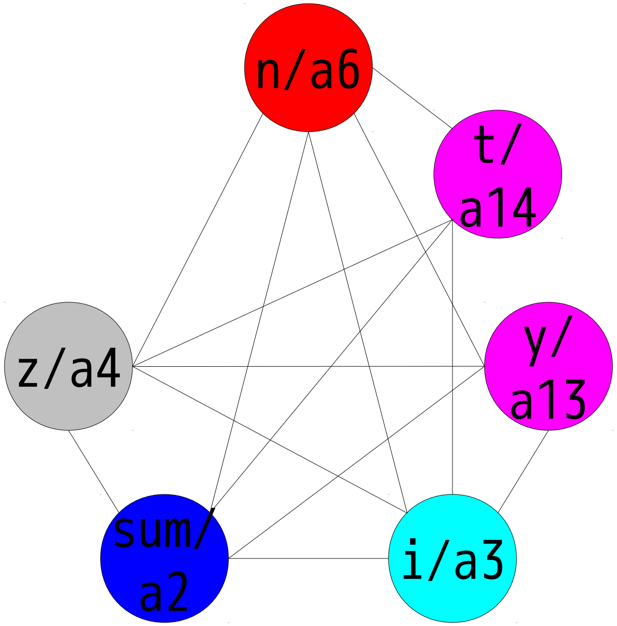
$r0

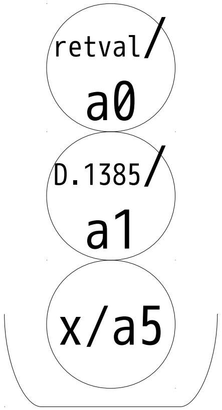
$r1

$r2

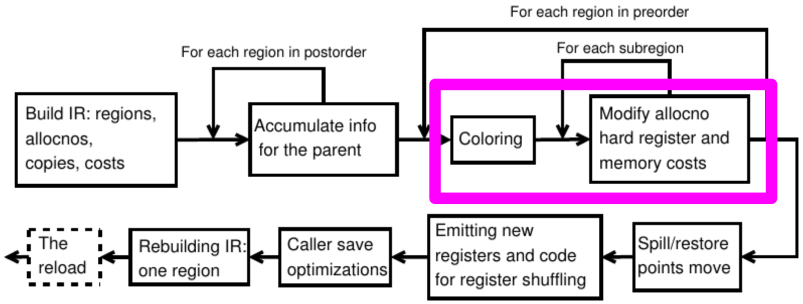
$r3

z/a4 拿不到 Register 標為 Spill 但繼續著色





Stack


For each region in postorder

Build IR: regions, allocnos, copies, costs

Accumulate info for the parent

For each region in preorder

For each subregion

Coloring

Modify allocno hard register and memory costs

Spill/restore points move

Emitting new registers and code for register shuffling

Caller save optimizations

Rebuilding IR: one region

The reload

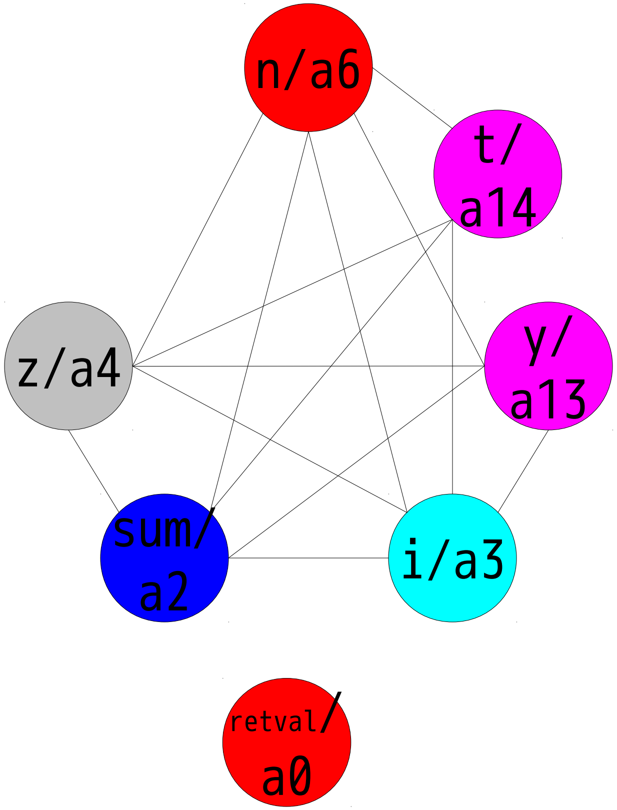
Coloring Region 0

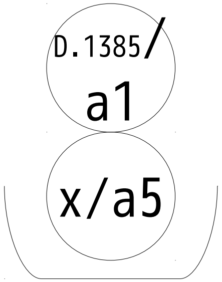
$r0

$r1

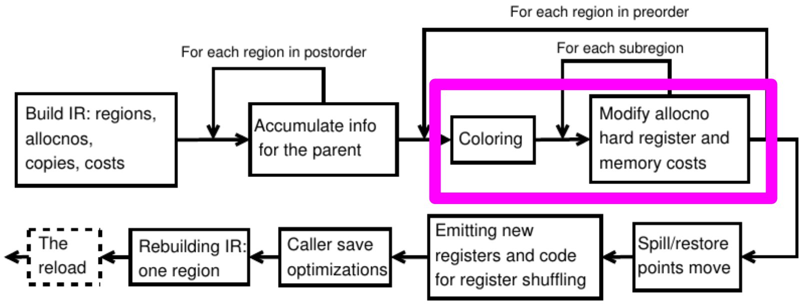
$r2

$r3





Stack



For each region in preorder

For each region in postorder

For each subregion

Build IR: regions, allocnos, copies, costs

Accumulate info for the parent

Coloring

Modify allocno hard register and memory costs

The reload

Rebuilding IR: one region

Caller save optimizations

Emitting new registers and code for register shuffling

Spill/restore points move

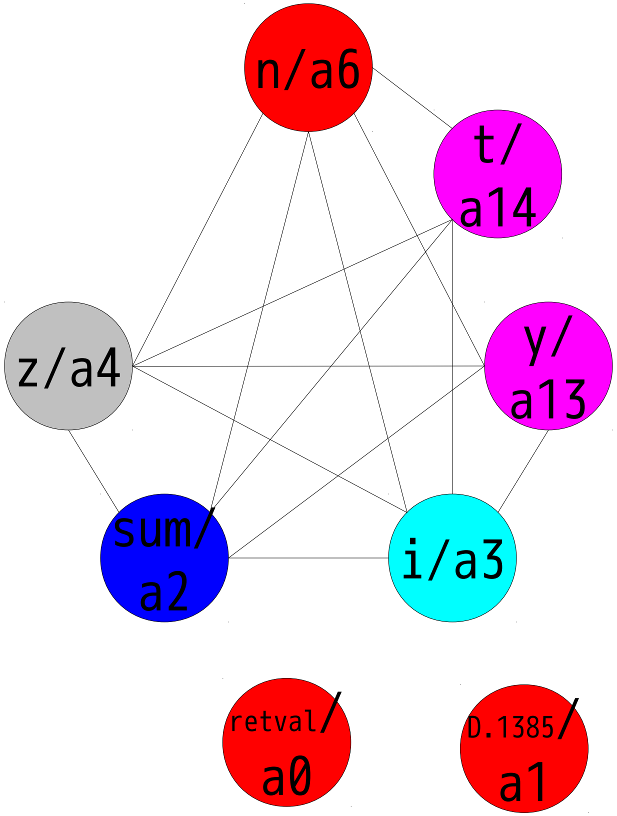
Coloring Region 0

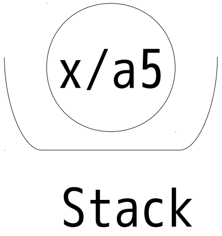
$r0

$r1

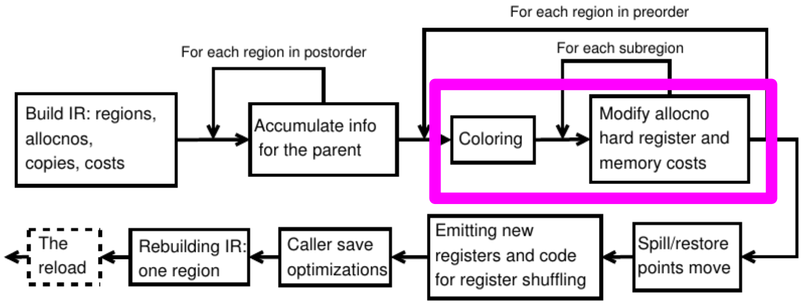
$r2

$r3





Coloring Region 0

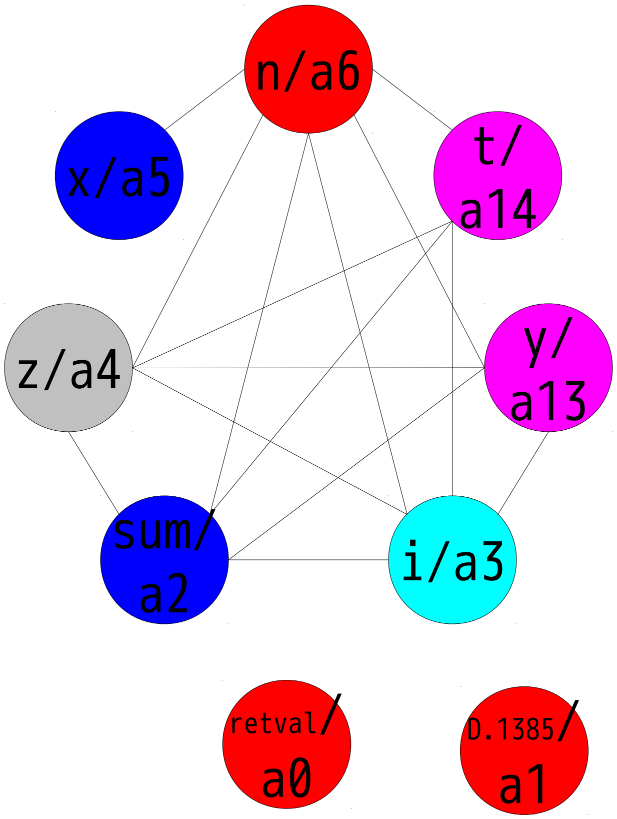

For each region in preorder
For each region in postorder
For each subregion
Build IR: regions, allocnos, copies, costs
Accumulate info for the parent
Coloring
Modify allocno hard register and memory costs
The reload
Rebuilding IR: one region
Caller save optimizations
Emitting new registers and code for register shuffling
Spill/restore points move

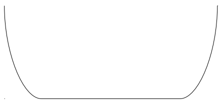
$r0

$r1

$r2

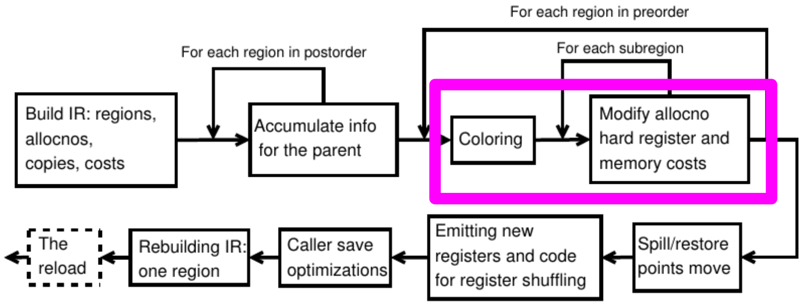
$r3

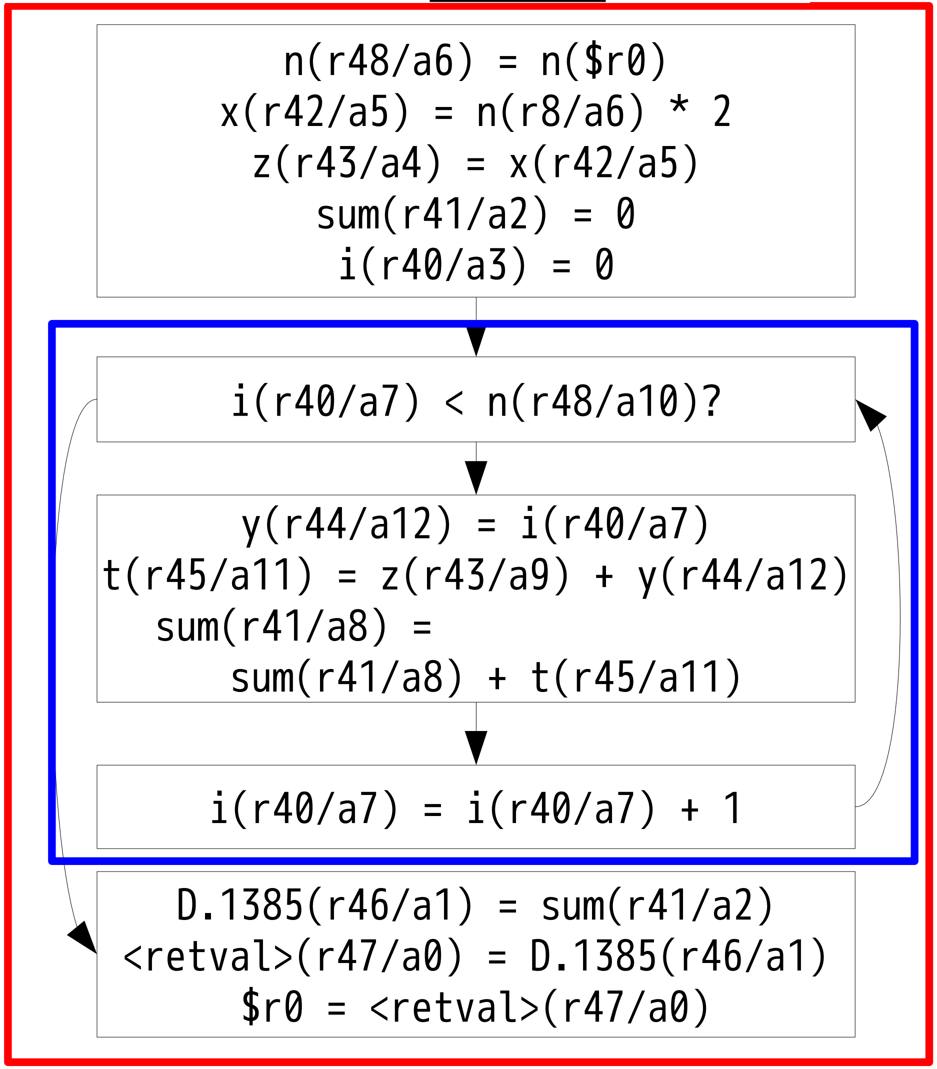




Stack

Coloring Region 1

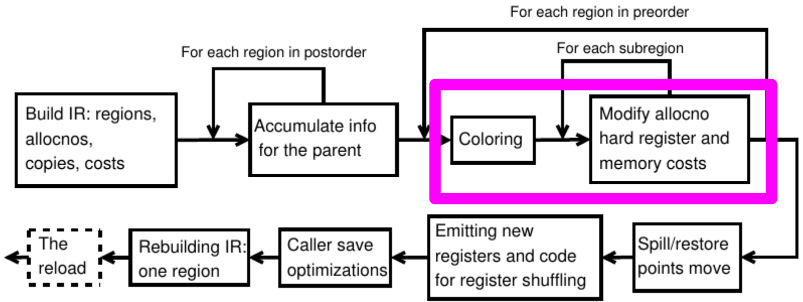

For each region in preorder
For each region in postorder
For each subregion
Build IR: regions, allocnos, copies, costs
Accumulate info for the parent
Coloring
Modify allocno hard register and memory costs
The reload
Rebuilding IR: one region
Caller save optimizations
Emitting new registers and code for register shuffling
Spill/restore points move



```
n(r48/a6) = n($r0)
x(r42/a5) = n(r8/a6) * 2
z(r43/a4) = x(r42/a5)
sum(r41/a2) = 0
i(r40/a3) = 0

i(r40/a7) < n(r48/a10)?

y(r44/a12) = i(r40/a7)
t(r45/a11) = z(r43/a9) + y(r44/a12)
sum(r41/a8) =
    sum(r41/a8) + t(r45/a11)

i(r40/a7) = i(r40/a7) + 1

D.1385(r46/a1) = sum(r41/a2)
<retval>(r47/a0) = D.1385(r46/a1)
$r0 = <retval>(r47/a0)
```

```
 Loop 1 (parent 0, header bb4, depth 1)
...
     Forming thread by copy 3:a11r45-a12r44 (freq=1):
       Result (freq=4): a11r45(2) a12r44(2)
     Pushing a12(r44,l1)(cost 0)
       Making a7(r40,l1) colorable
       Making a8(r41,l1) colorable
       Making a10(r48,l1) colorable
     Pushing a10(r48,l1)(cost 40)
     Pushing a8(r41,l1)(cost 48)
     Pushing a11(r45,l1)(cost 0)
     Pushing a7(r40,l1)(cost 64)
     Popping a7(r40,l1)  -- assign reg 2
     Popping a11(r45,l1)  -- assign reg 3
     Popping a8(r41,l1)  -- assign reg 1
     Popping a10(r48,l1)  -- assign reg 0
     Popping a12(r44,l1)  -- assign reg 3
```

Coloring Region 1



For each region in preorder

For each region in postorder

For each subregion

Build IR: regions, allocnos, copies, costs

Accumulate info for the parent

Coloring

Modify allocno hard register and memory costs

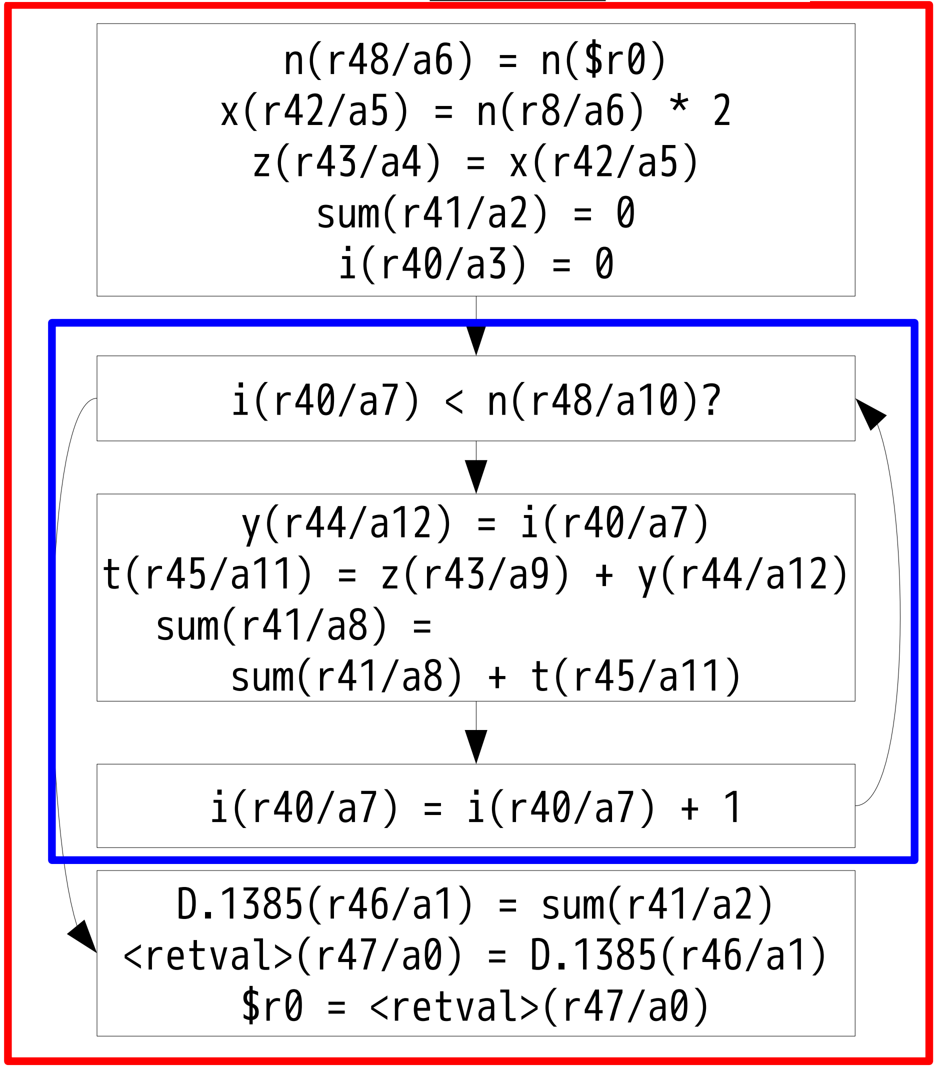
Spill/restore points move

Emitting new registers and code for register shuffling

Caller save optimizations

Rebuilding IR: one region

The reload



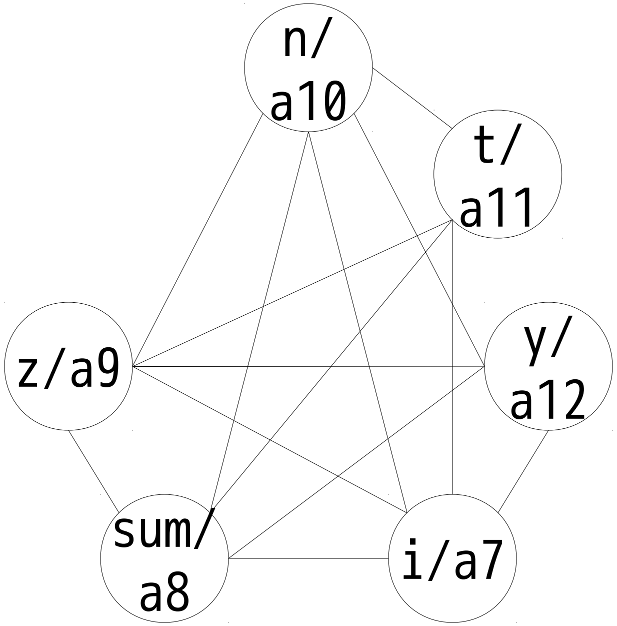
```
n(r48/a6) = n($r0)
x(r42/a5) = n(r8/a6) * 2
z(r43/a4) = x(r42/a5)
sum(r41/a2) = 0
i(r40/a3) = 0
```

```
i(r40/a7) < n(r48/a10)?
```

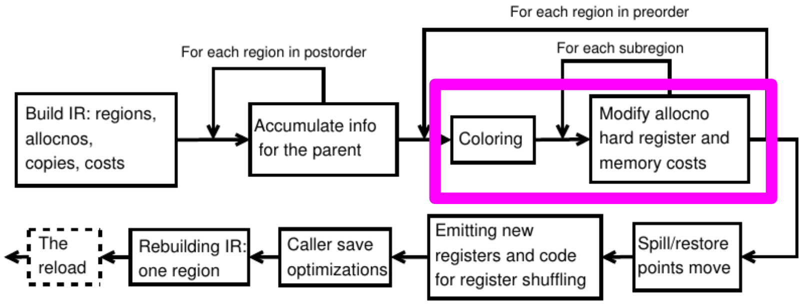
```
y(r44/a12) = i(r40/a7)
t(r45/a11) = z(r43/a9) + y(r44/a12)
sum(r41/a8) =
sum(r41/a8) + t(r45/a11)
```

```
i(r40/a7) = i(r40/a7) + 1
```

```
D.1385(r46/a1) = sum(r41/a2)
<retval>(r47/a0) = D.1385(r46/a1)
$r0 = <retval>(r47/a0)
```

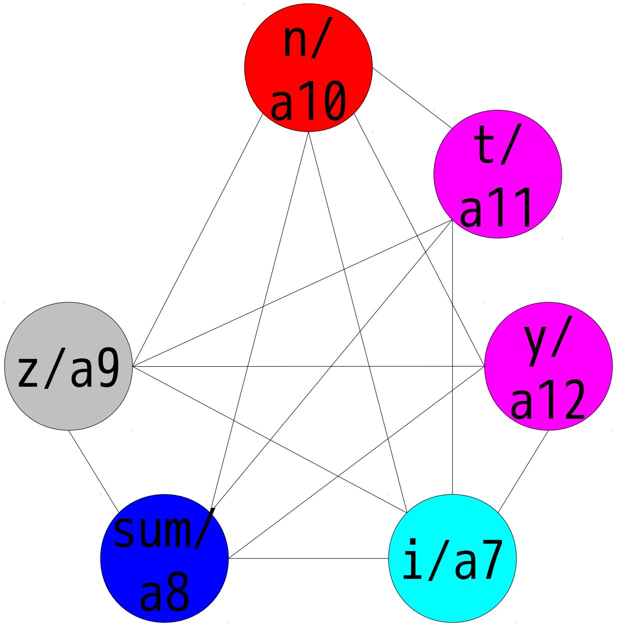


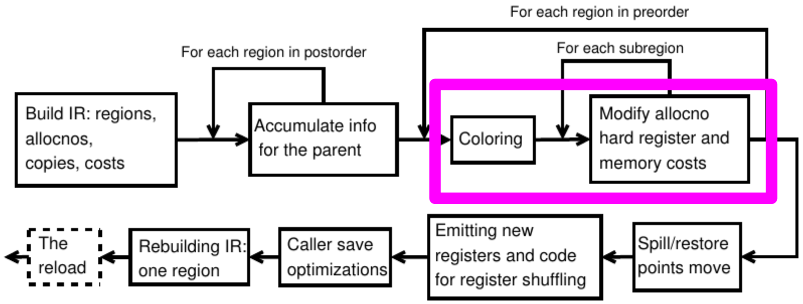
Coloring Region 1


For each region in preorder
For each region in postorder
For each subregion
Build IR: regions, allocnos, copies, costs
Accumulate info for the parent
Coloring
Modify allocno hard register and memory costs
The reload
Rebuilding IR: one region
Caller save optimizations
Emitting new registers and code for register shuffling
Spill/restore points move

$r0
$r1
$r2
$r3

z/a4 拿不到 Register,
z/a9
繼續拿不到

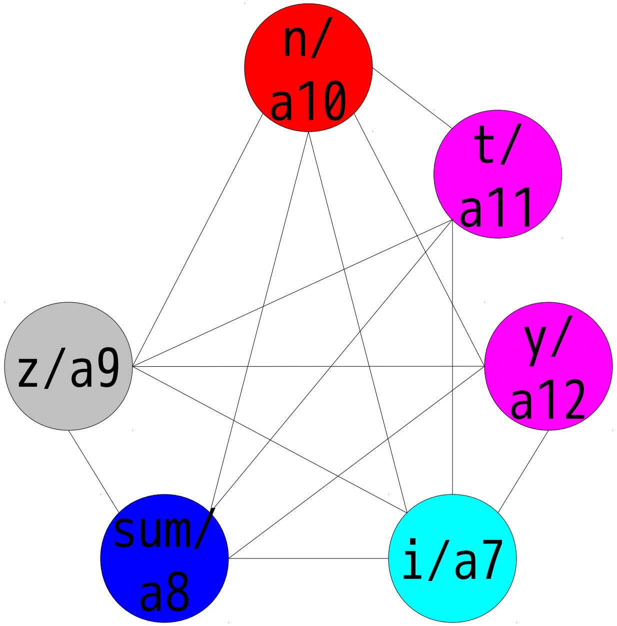



For each region in postorder
For each region in preorder
For each subregion
Build IR: regions, allocnos, copies, costs
Accumulate info for the parent
Coloring
Modify allocno hard register and memory costs
Spill/restore points move
Emitting new registers and code for register shuffling
Caller save optimizations
Rebuilding IR: one region
The reload

Coloring Region 1

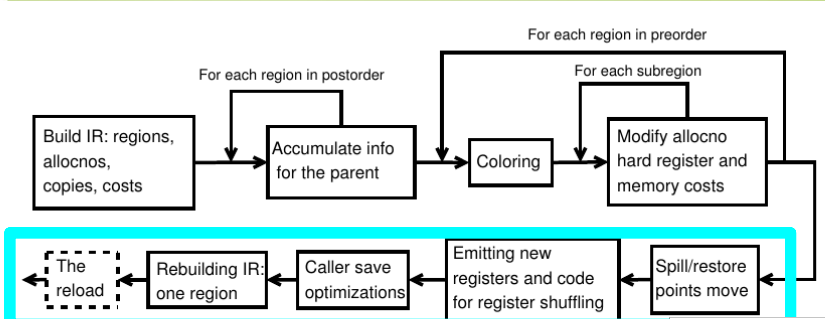
$r0
$r1
$r2
$r3

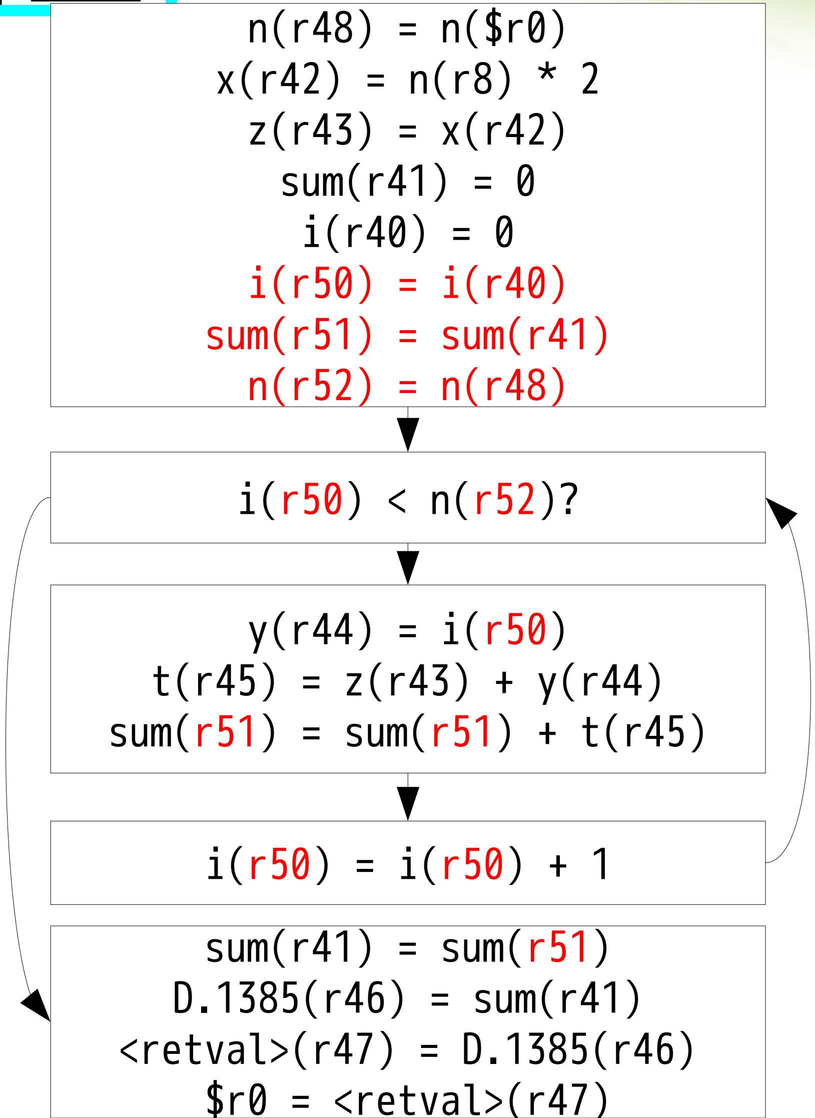
z/a4 拿不到
Register,
z/a9
繼續拿不到



所有 Spill 的 Register 會在下階段處理

Insert Move


For each region in preorder
For each region in postorder
For each subregion
Build IR: regions, allocnos, copies, costs
Accumulate info for the parent
Coloring
Modify allocno hard register and memory costs
The reload
Rebuilding IR: one region
Caller save optimizations
Emitting new registers and code for register shuffling
Spill/restore points move



```
n(r48) = n($r0)
x(r42) = n(r8) * 2
z(r43) = x(r42)
sum(r41) = 0
i(r40) = 0
i(r50) = i(r40)
sum(r51) = sum(r41)
n(r52) = n(r48)
```

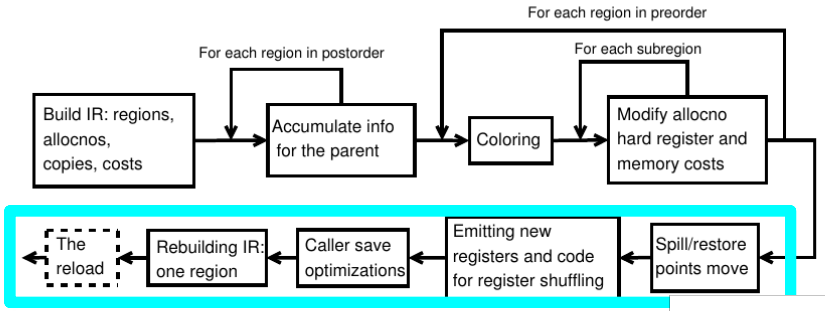
```
i(r50) < n(r52)?
```

```
y(r44) = i(r50)
t(r45) = z(r43) + y(r44)
sum(r51) = sum(r51) + t(r45)
```

```
i(r50) = i(r50) + 1
```

```
sum(r41) = sum(r51)
D.1385(r46) = sum(r41)
<retval>(r47) = D.1385(r46)
$r0 = <retval>(r47)
```

RELOAD



For each region in preorder

For each region in postorder

For each subregion

Build IR: regions, allocnos, copies, costs

Accumulate info for the parent

Coloring

Modify allocno hard register and memory costs

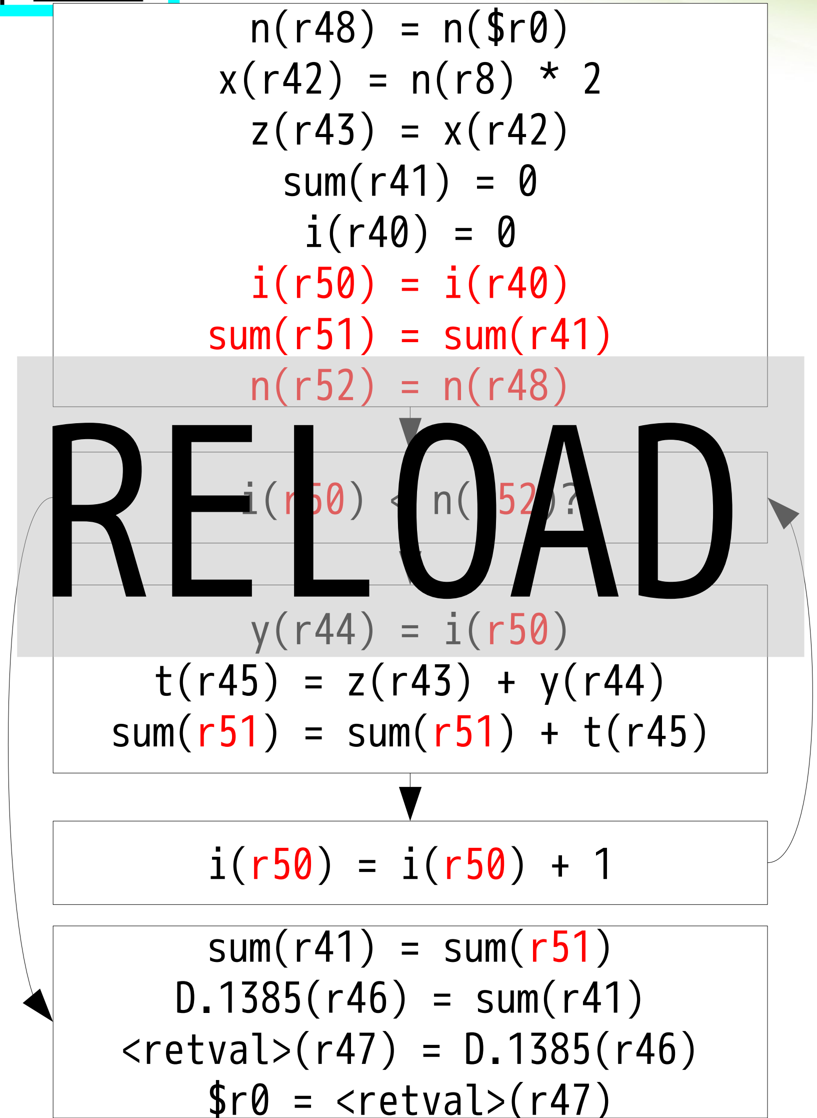
Spill/restore points move

Emitting new registers and code for register shuffling

Caller save optimizations

Rebuilding IR: one region

The reload



```
n(r48) = n($r0)
x(r42) = n(r8) * 2
z(r43) = x(r42)
sum(r41) = 0
i(r40) = 0
i(r50) = i(r40)
sum(r51) = sum(r41)
n(r52) = n(r48)
```

RELOAD

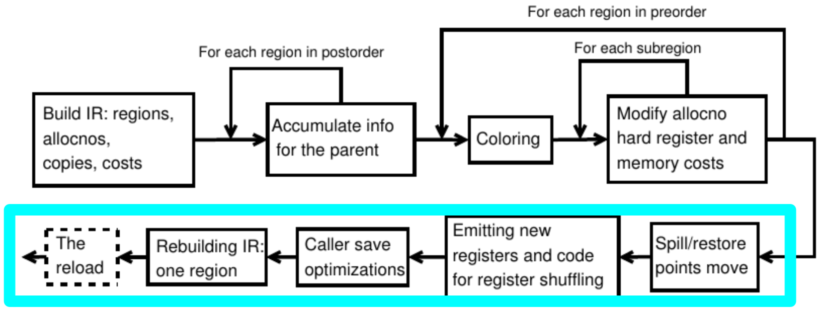
```
i(r50) < n(r52)?
```

```
y(r44) = i(r50)
t(r45) = z(r43) + y(r44)
sum(r51) = sum(r51) + t(r45)
```

```
i(r50) = i(r50) + 1
```

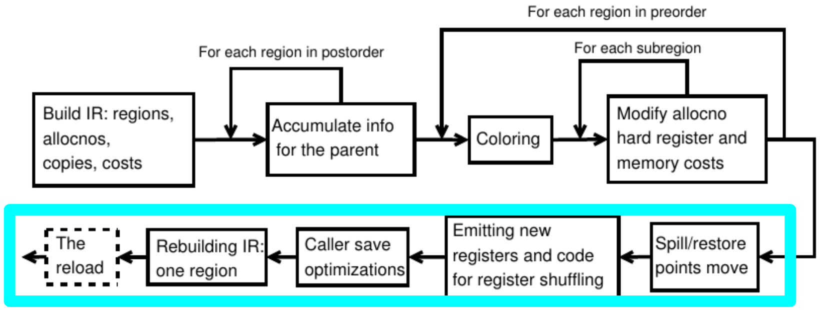
```
sum(r41) = sum(r51)
D.1385(r46) = sum(r41)
<retval>(r47) = D.1385(r46)
$r0 = <retval>(r47)
```

RELOAD


For each region in preorder
For each region in postorder
For each subregion
Build IR: regions, allocnos, copies, costs
Accumulate info for the parent
Coloring
Modify allocno hard register and memory costs
The reload
Rebuilding IR: one region
Caller save optimizations
Emitting new registers and code for register shuffling
Spill/restore points move

- nds32 無實作 Reload 相關 Hook 因此不展示 Reload 相關流程

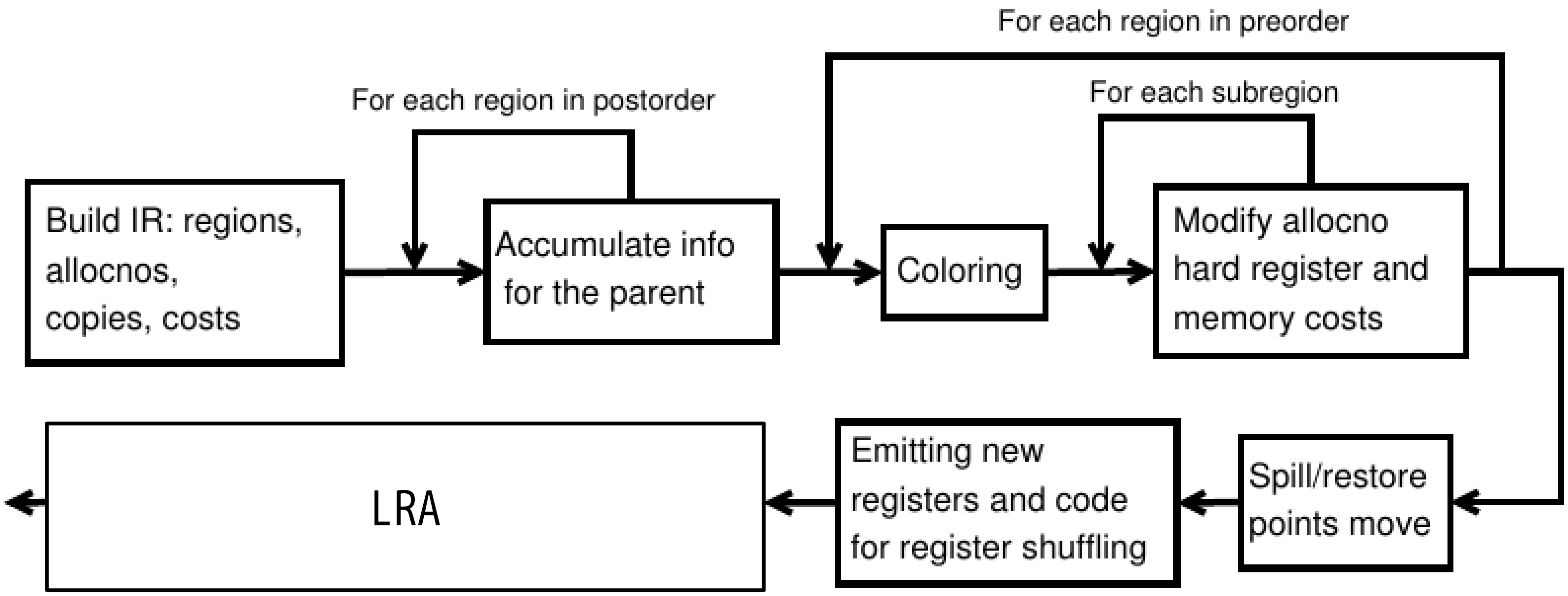
RELOAD


For each region in preorder
For each region in postorder
For each subregion
Build IR: regions, allocnos, copies, costs
Accumulate info for the parent
Coloring
Modify allocno hard register and memory costs
The reload
Rebuilding IR: one region
Caller save optimizations
Emitting new registers and code for register shuffling
Spill/restore points move

- nds32 無實作 Reload 相關 Hook 因此不展示 Reload 相關流程

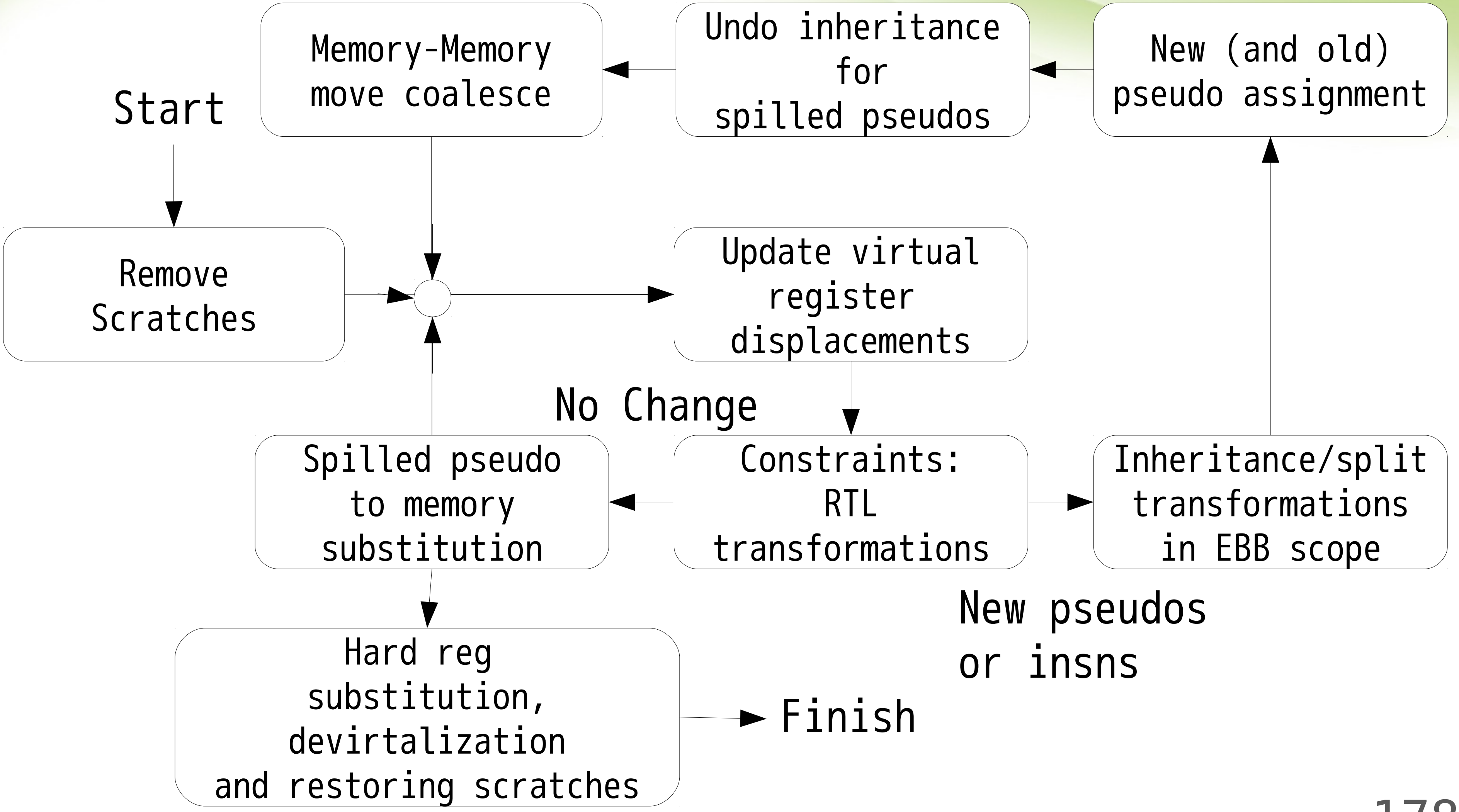
- IRA 中使用 LRA 或 Reload， 流程會有所不同

IRA + LRA


For each region in preorder
For each region in postorder
For each subregion
Build IR: regions, allocnos, copies, costs
Accumulate info for the parent
Coloring
Modify allocno hard register and memory costs
LRA
Emitting new registers and code for register shuffling
Spill/restore points move

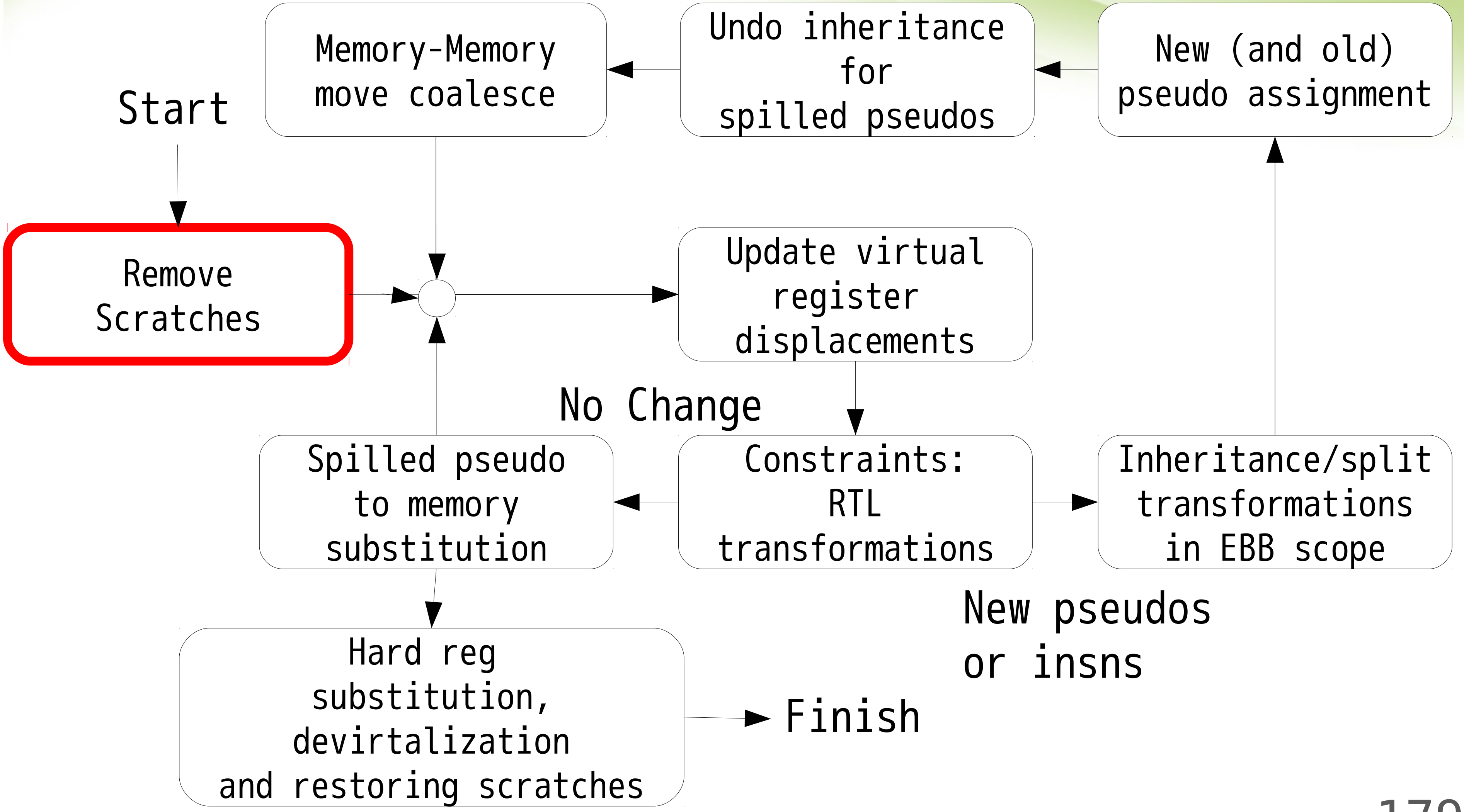
Ref: Vladimir. N. .Makarov, “The top-down regional register allocator for irregular register file architectures”

LRA


Start
Remove Scratches
Memory-Memory move coalesce
Undo inheritance for spilled pseudos
New (and old) pseudo assignment
Update virtual register displacements
No Change
Spilled pseudo to memory substitution
Constraints: RTL transformations
Inheritance/split transformations in EBB scope
New pseudos or insns
Hard reg substitution, devirtalization and restoring scratches
Finish

Ref: Vladimir. N. .Makarov, “The Local Register Allocator Project”, GNU Tools Cauldron 2012

LRA



Start

Remove Scratches

Memory-Memory move coalesce

Undo inheritance for spilled pseudos

New (and old) pseudo assignment

Update virtual register displacements

No Change

Spilled pseudo to memory substitution

Constraints: RTL transformations

Inheritance/split transformations in EBB scope

New pseudos or insns

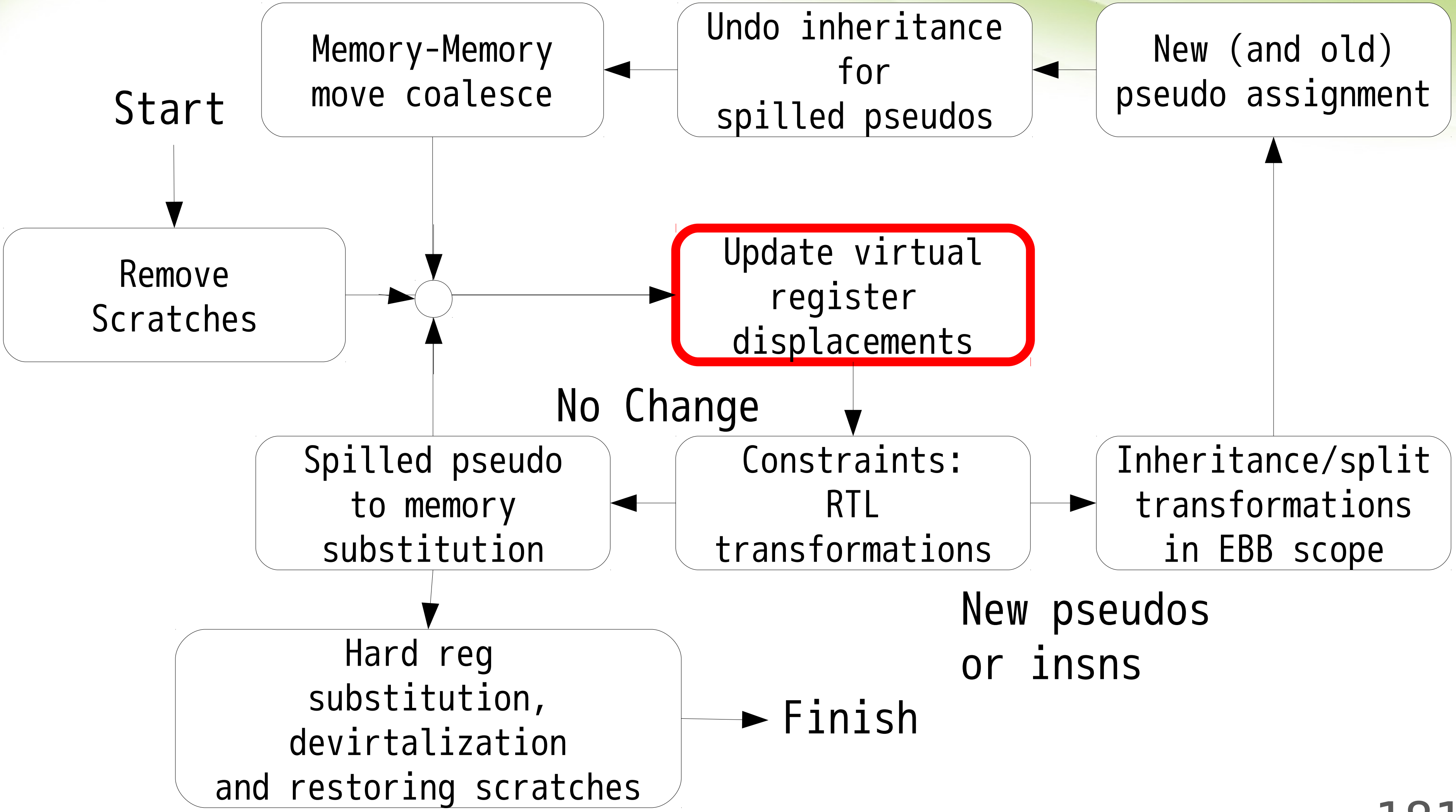
Hard reg substitution, devirtualization and restoring scratches

Finish

Remove Scratches

處理 match_scratch
讓它暫時先變 pseudo register
以利後續處理

LRA



Start

Remove Scratches

Memory-Memory move coalesce

Undo inheritance for spilled pseudos

New (and old) pseudo assignment

Update virtual register displacements

No Change

Spilled pseudo to memory substitution

Constraints: RTL transformations

Inheritance/split transformations in EBB scope

New pseudos or insns

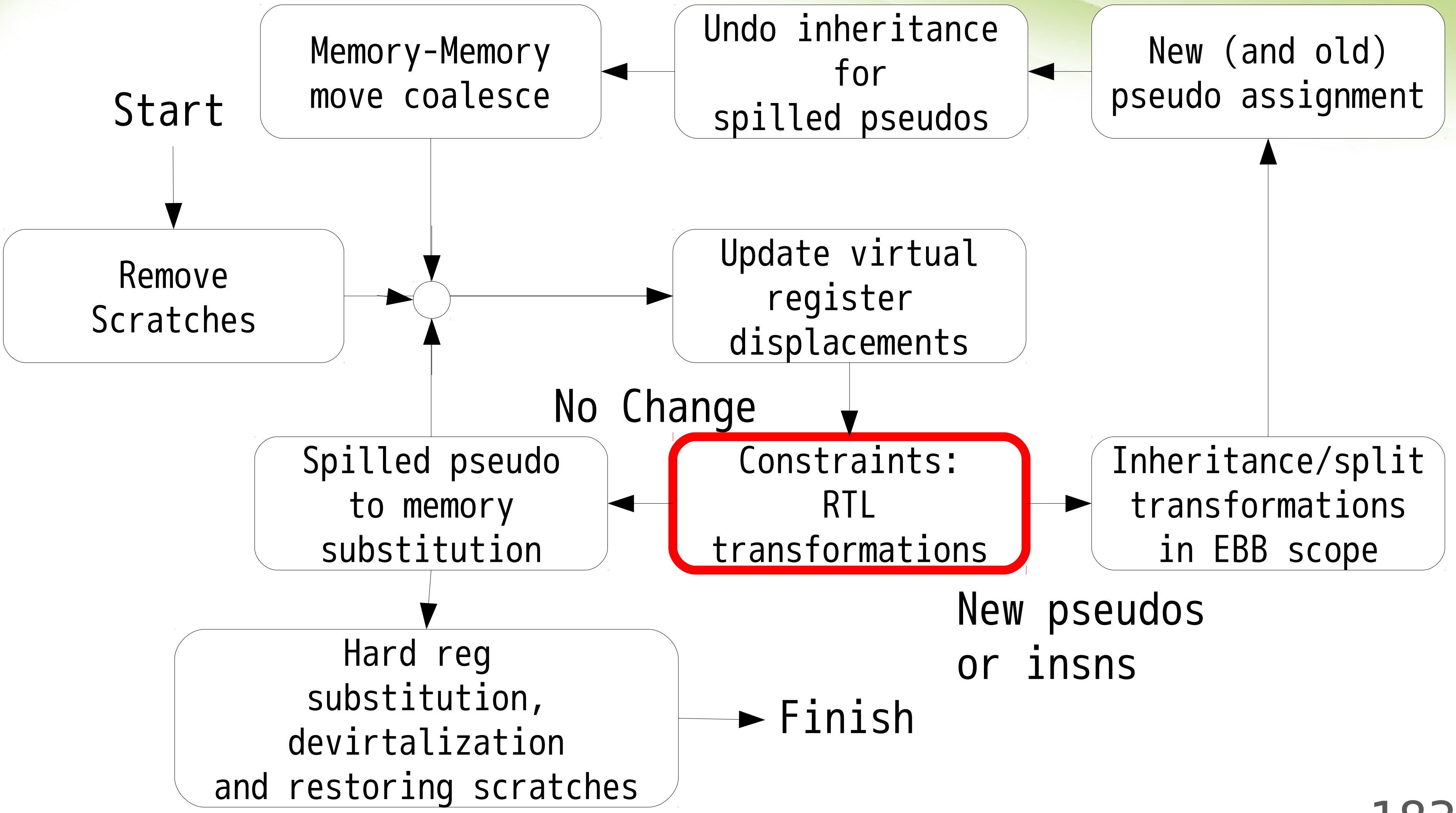
Hard reg substitution, devirtualization and restoring scratches

Finish

Update virtual register displacements

- 分配 Stack slot 空間
  - eg: $sp + 0, $sp + 4, ...
- 融合 $sp 跟 $fp

LRA



Start

Remove Scratches

Memory-Memory move coalesce

Undo inheritance for spilled pseudos

New (and old) pseudo assignment

Update virtual register displacements

No Change

Spilled pseudo to memory substitution

Constraints: RTL transformations

Inheritance/split transformations in EBB scope

New pseudos or insns

Hard reg substitution, devirtualization and restoring scratches

Finish

Constraint

```
(define_insn "addsi"
  [(set (match_operand:SI          0 "register_operand", "r,    r")
        (plus:SI (match_operand:SI 1 "register_operand", "r,    r")
                 (match_operand:SI 2 "register_operand", "r, Is15")))]
  ""
...
```

Constraint是gcc在md中用來
描述指令中Operand的資訊
以上面為例，上面描述add可接受兩種格式：
add $ra, $rb, $rc
add $ra, $rb, #simm15

LRA Constraints #0

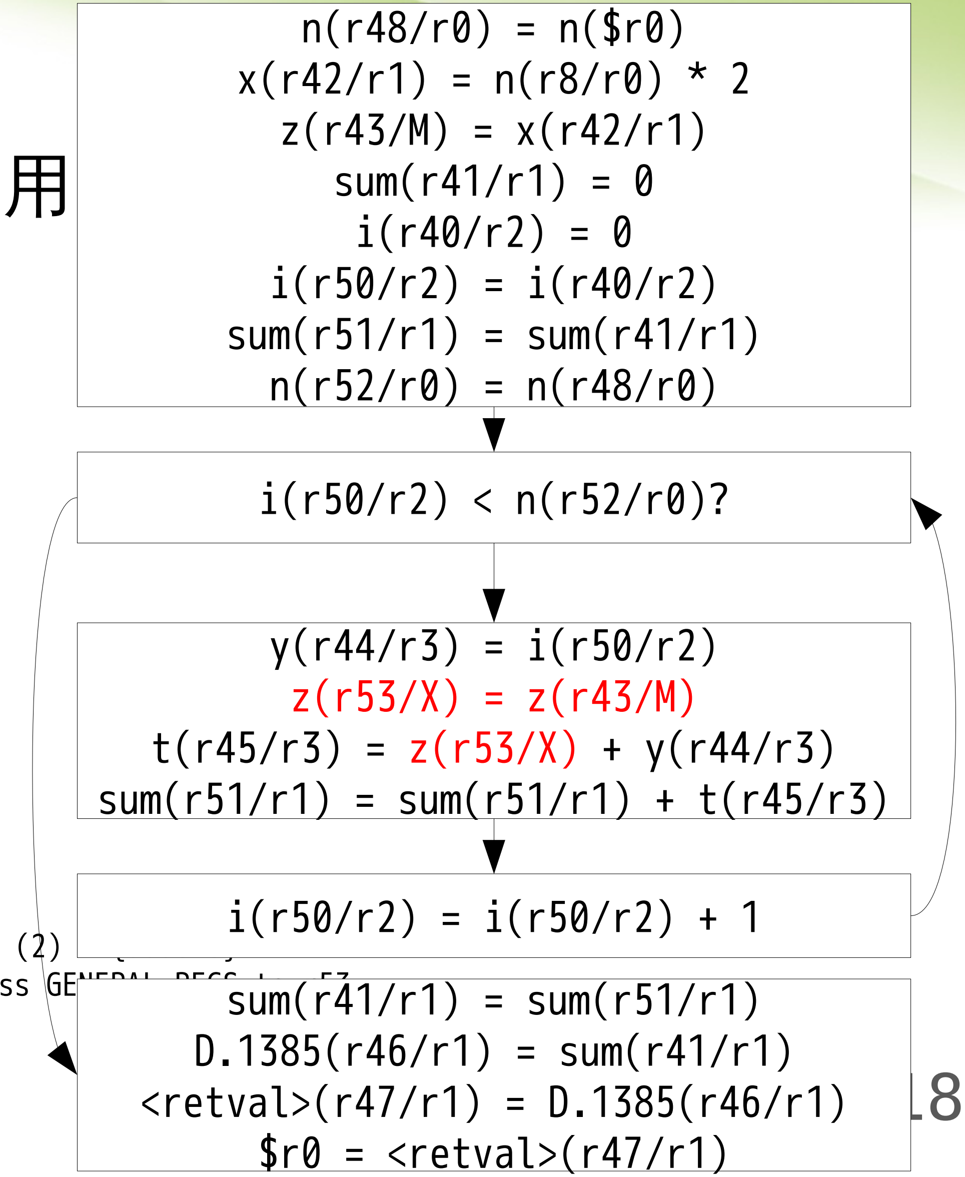
```
...
          alt=0,overall=0,losers=0,rld_nregs=0
             Choosing alt 0 in insn 6:  (0) =l  (1) l  (2) Iu03 {ashlsi3}
            0 Non input pseudo reload: reject++
          alt=0,overall=607,losers=1,rld_nregs=1
            0 Non input pseudo reload: reject++
          alt=1,overall=607,losers=1,rld_nregs=1
            0 Non pseudo reload: reject++
          alt=2,overall=1,losers=0,rld_nregs=0
             Choosing alt 2 in insn 7:  (0) U45  (1) l {*movsi}
...
```

檢查每道指令是否符合任何一組 Constraint

LRA Constraints #0

```
...
          1 Matching alt: reject+=2
          alt=0: Bad operand -- refuse
          alt=1: Bad operand -- refuse
          1 Matching alt: reject+=2
          alt=2: Bad operand -- refuse
          alt=3: Bad operand -- refuse
          1 Matching alt: reject+=2
        alt=4,overall=8,losers=1,rld_nregs=1
        alt=5,overall=6,losers=1,rld_nregs=1
          alt=6: Bad operand -- refuse
          alt=7: Bad operand -- refuse
          alt=8: Bad operand -- refuse
        alt=9,overall=6,losers=1,rld_nregs=1
          alt=0: Bad operand -- refuse
          alt=1: Bad operand -- refuse
          alt=2: Bad operand -- refuse
          alt=3: Bad operand -- refuse
        alt=4,overall=6,losers=1,rld_nregs=1
        alt=5,overall=6,losers=1,rld_nregs=1
          alt=6: Bad operand -- refuse
          alt=7: Bad operand -- refuse
          alt=8: Bad operand -- refuse
        alt=9,overall=6,losers=1,rld_nregs=1
  Commutative operand exchange in insn 14
           Choosing alt 4 in insn 14:  (0) d  (1) 0  (2)
      Creating newreg=53 from oldreg=43, assigning class GE
  14: r45:SI=r44:SI+r53:SI
      REG_DEAD r44:SI
    Inserting insn reload before:
  40: r53:SI=r43:SI
...
```

找不到一組 Constraint 可用時會進行 Split



n(r48/r0) = n($r0)
x(r42/r1) = n(r8/r0) * 2
z(r43/M) = x(r42/r1)
sum(r41/r1) = 0
i(r40/r2) = 0
i(r50/r2) = i(r40/r2)
sum(r51/r1) = sum(r41/r1)
n(r52/r0) = n(r48/r0)

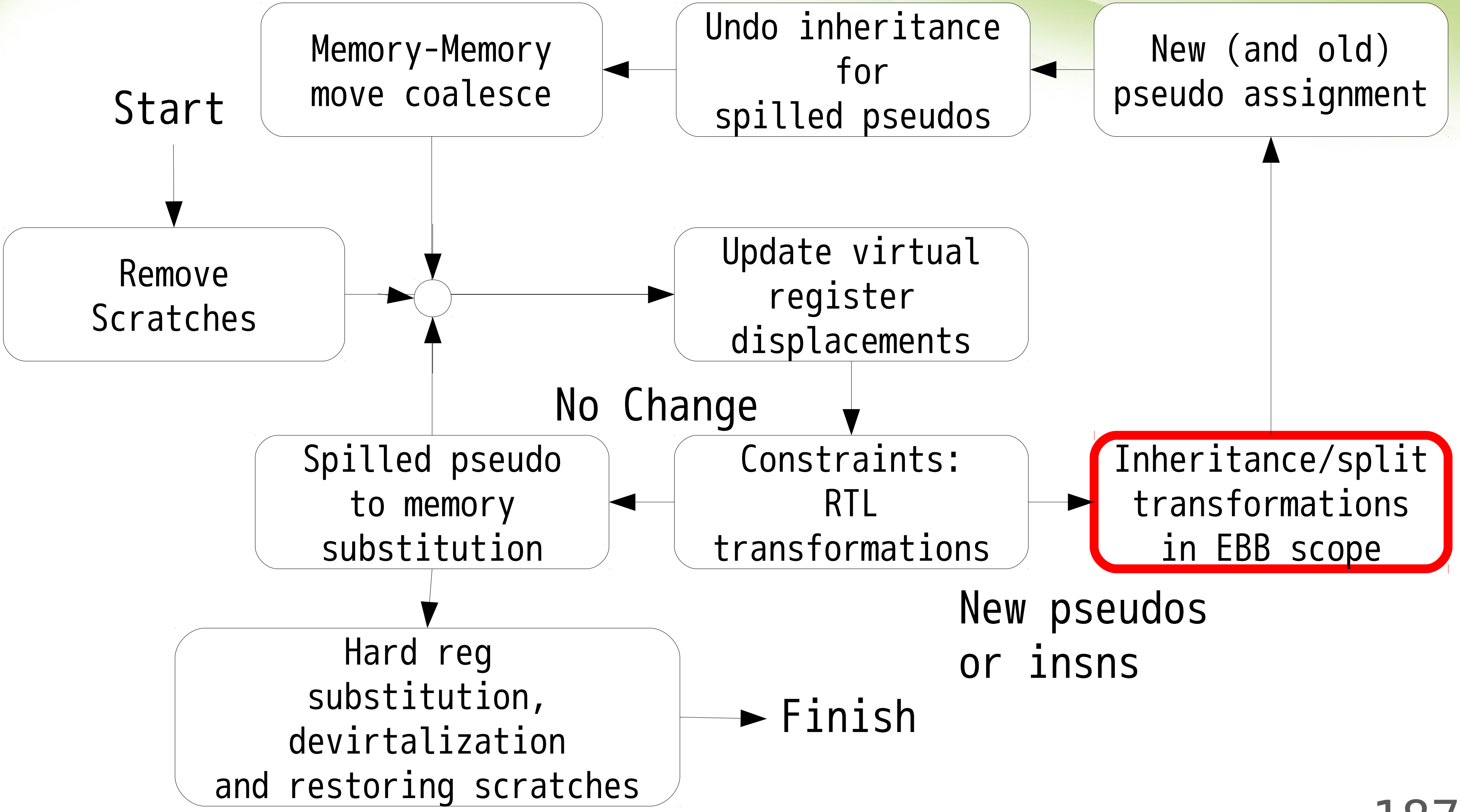
i(r50/r2) < n(r52/r0)?

y(r44/r3) = i(r50/r2)
z(r53/X) = z(r43/M)
t(r45/r3) = z(r53/X) + y(r44/r3)
sum(r51/r1) = sum(r51/r1) + t(r45/r3)

i(r50/r2) = i(r50/r2) + 1

sum(r41/r1) = sum(r51/r1)
D.1385(r46/r1) = sum(r41/r1)
<retval>(r47/r1) = D.1385(r46/r1)
$r0 = <retval>(r47/r1)

LRA



Start

Remove Scratches

Memory-Memory move coalesce

Update virtual register displacements

No Change

Constraints: RTL transformations

Inheritance/split transformations in EBB scope

New pseudos or insns

New (and old) pseudo assignment

Undo inheritance for spilled pseudos

Spilled pseudo to memory substitution

Hard reg substitution, devirtualization and restoring scratches

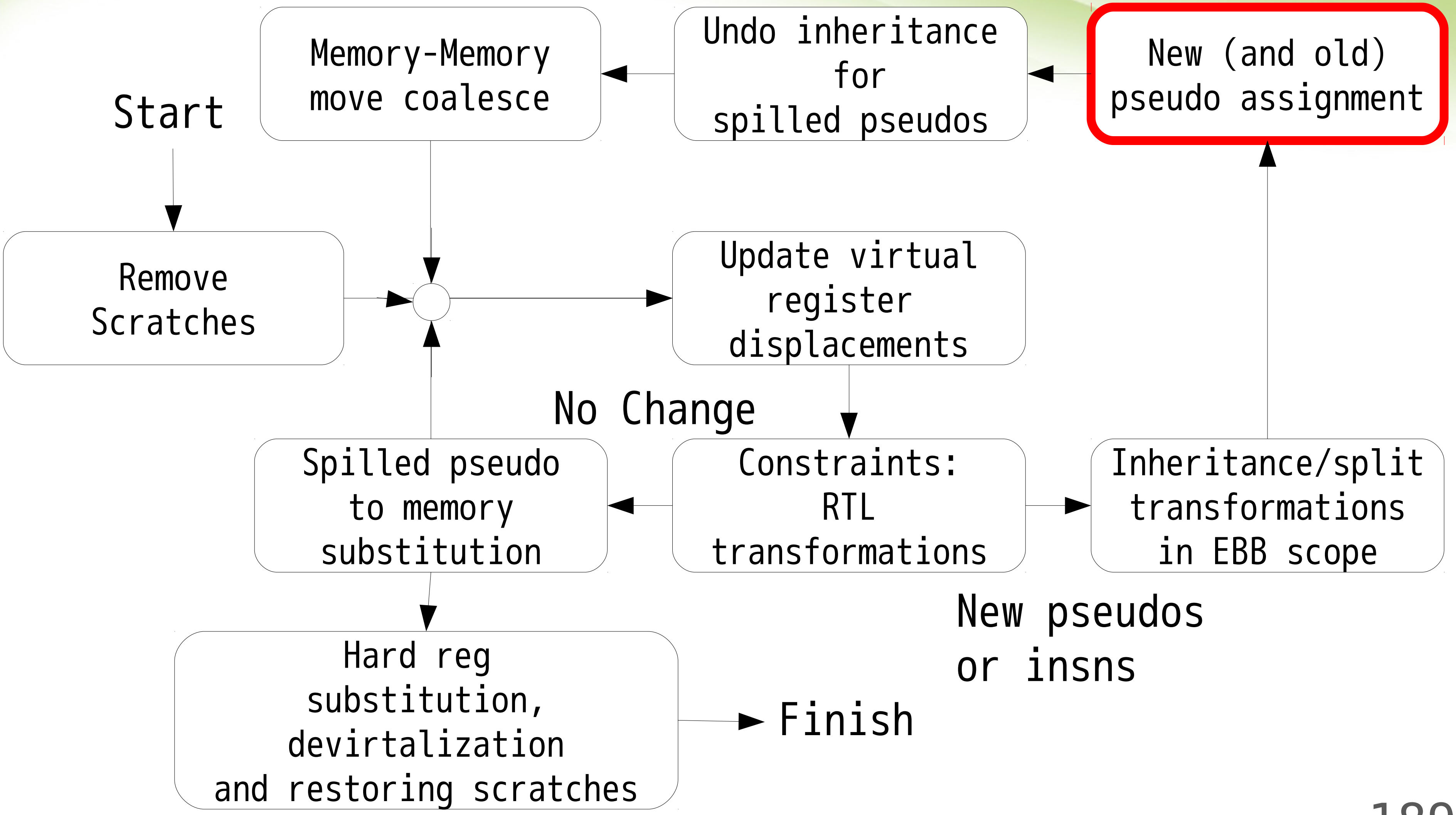
Finish

lra-constraints.c:lra_inheritance()

Inheritance/Split

- 這部份比較複雜， LRA 嘗試最佳化的部份
  - 時間關係略過
  - lra-constraints.c:lra_inheritance()

LRA



Start

Remove Scratches

Update virtual register displacements

Constraints: RTL transformations

No Change

Spilled pseudo to memory substitution

Hard reg substitution, devirtualization and restoring scratches

Finish

New pseudos or insns

Inheritance/split transformations in EBB scope

New (and old) pseudo assignment

Undo inheritance for spilled pseudos

Memory-Memory move coalesce

LRA Assign #1

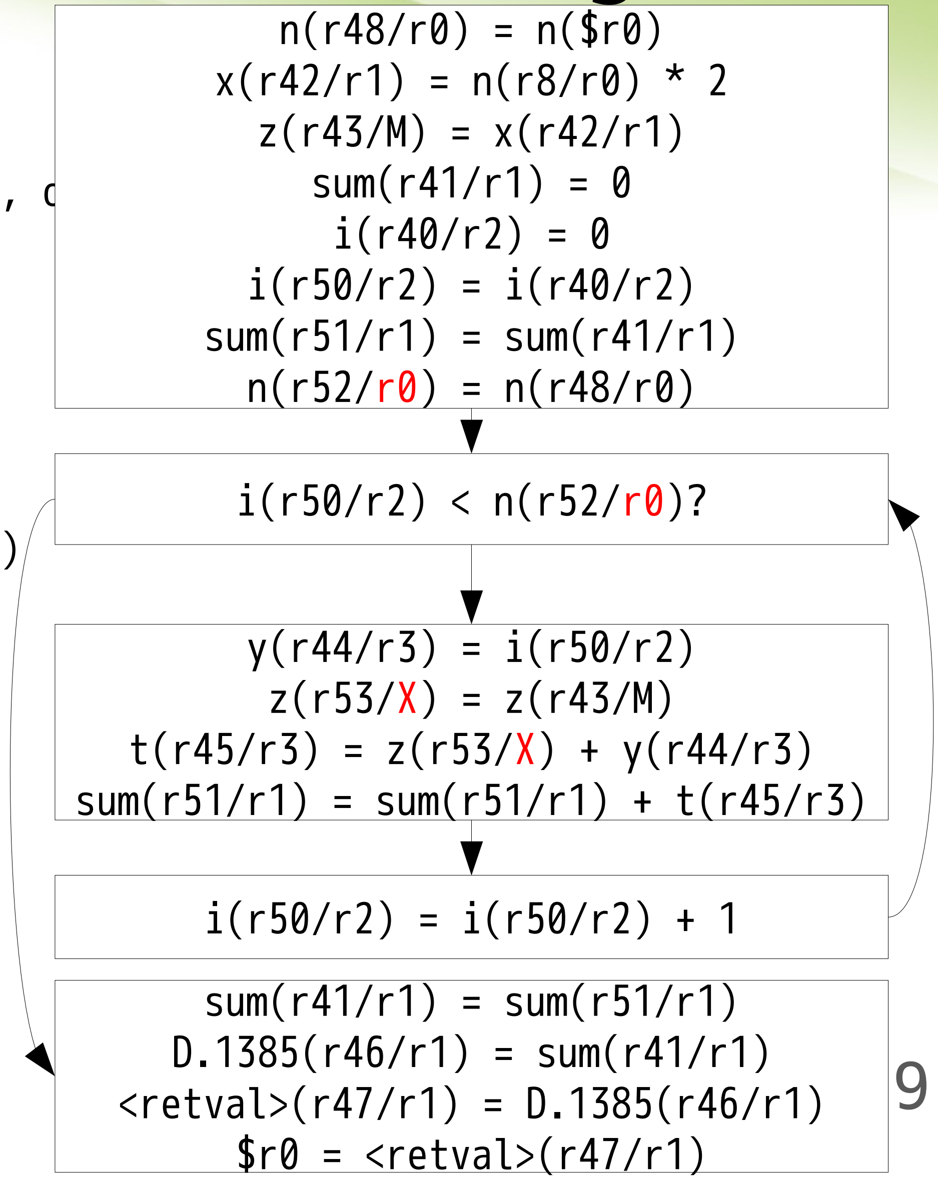
```
    Assigning to 53 (cl=GENERAL_REGS, orig=43, freq=2, tfirst=53, tfreq=2)...
    Trying 0: spill 52(freq=2)           Now best 0(cost=0)

    Trying 1: spill 51(freq=3)
    Trying 2: spill 50(freq=4)
    Trying 3: spill 44(freq=2)
Spill r52(hr=0, freq=2) for r53
      Assign 0 to reload r53 (freq=2)
```

要 Spill 時會先嘗試有沒有 Register,
沒有的時候只好找個替死鬼 ...

LRA Assign #1

```
    Assigning to 53 (cl=GENERAL_REGS, c
    Trying 0: spill 52(freq=2)

    Trying 1: spill 51(freq=3)
    Trying 2: spill 50(freq=4)
    Trying 3: spill 44(freq=2)
Spill r52(hr=0, freq=2) for r53
      Assign 0 to reload r53 (freq=2)
```

分到 $r0， 踢掉 r52



```
n(r48/r0) = n($r0)
x(r42/r1) = n(r8/r0) * 2
z(r43/M) = x(r42/r1)
sum(r41/r1) = 0
i(r40/r2) = 0
i(r50/r2) = i(r40/r2)
sum(r51/r1) = sum(r41/r1)
n(r52/r0) = n(r48/r0)
```

```
i(r50/r2) < n(r52/r0)?
```

```
y(r44/r3) = i(r50/r2)
z(r53/X) = z(r43/M)
t(r45/r3) = z(r53/X) + y(r44/r3)
sum(r51/r1) = sum(r51/r1) + t(r45/r3)
```

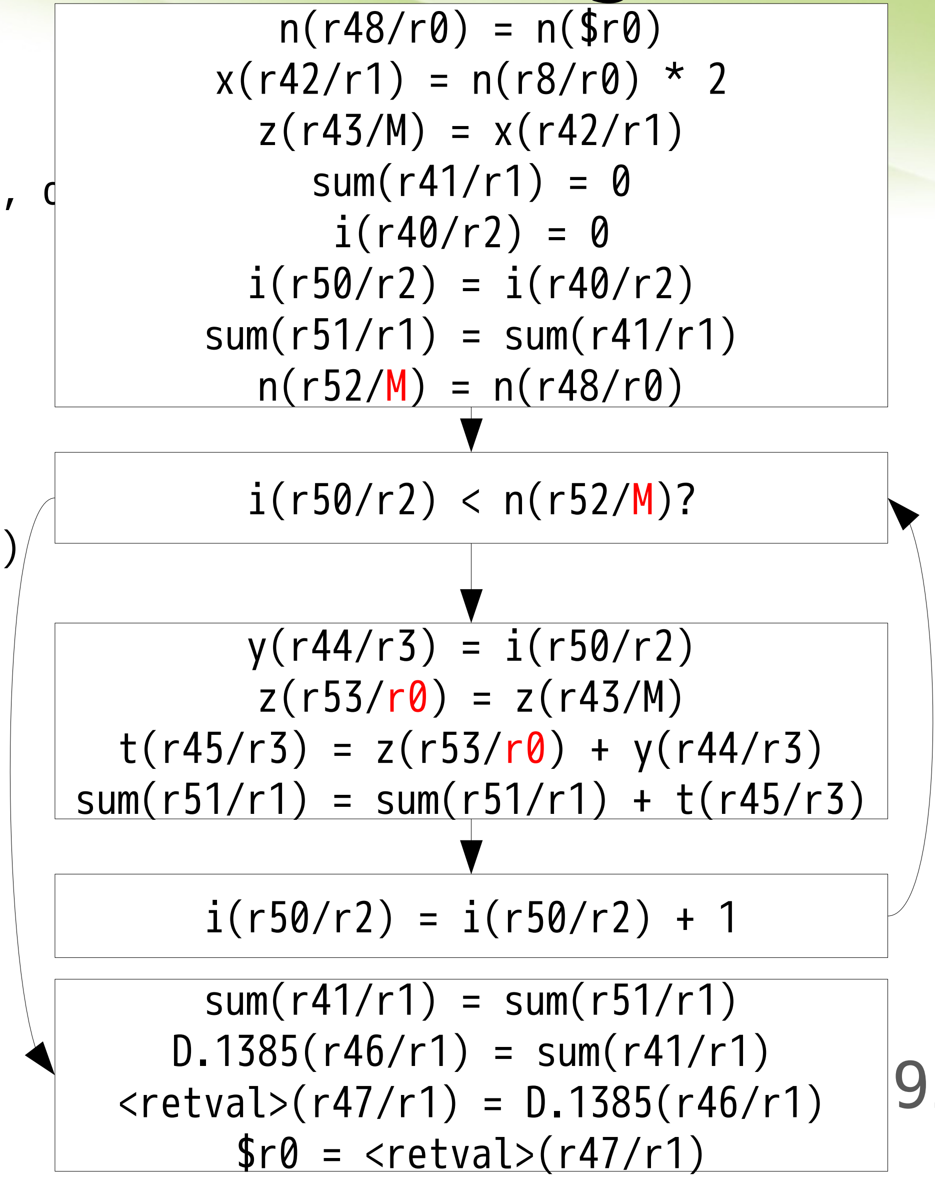
```
i(r50/r2) = i(r50/r2) + 1
```

```
sum(r41/r1) = sum(r51/r1)
D.1385(r46/r1) = sum(r41/r1)
<retval>(r47/r1) = D.1385(r46/r1)
$r0 = <retval>(r47/r1)
```

LRA Assign #1

```
    Assigning to 53 (cl=GENERAL_REGS, c
    Trying 0: spill 52(freq=2)

    Trying 1: spill 51(freq=3)
    Trying 2: spill 50(freq=4)
    Trying 3: spill 44(freq=2)
Spill r52(hr=0, freq=2) for r53
      Assign 0 to reload r53 (freq=2)
```

分到 $r0， 踢掉 r52



```
n(r48/r0) = n($r0)
x(r42/r1) = n(r8/r0) * 2
z(r43/M) = x(r42/r1)
sum(r41/r1) = 0
i(r40/r2) = 0
i(r50/r2) = i(r40/r2)
sum(r51/r1) = sum(r41/r1)
n(r52/M) = n(r48/r0)
```

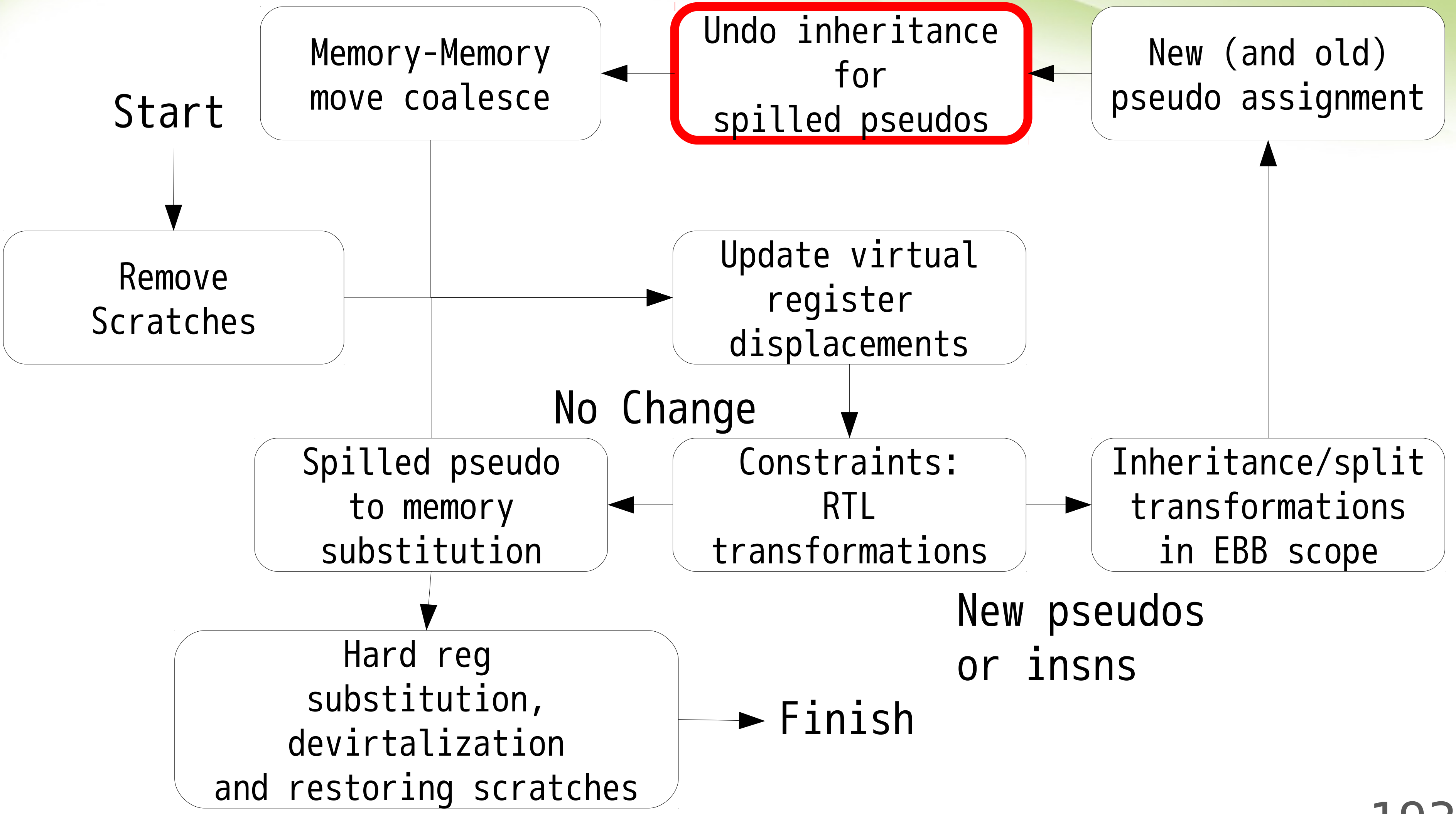
```
i(r50/r2) < n(r52/M)?
```

```
y(r44/r3) = i(r50/r2)
z(r53/r0) = z(r43/M)
t(r45/r3) = z(r53/r0) + y(r44/r3)
sum(r51/r1) = sum(r51/r1) + t(r45/r3)
```

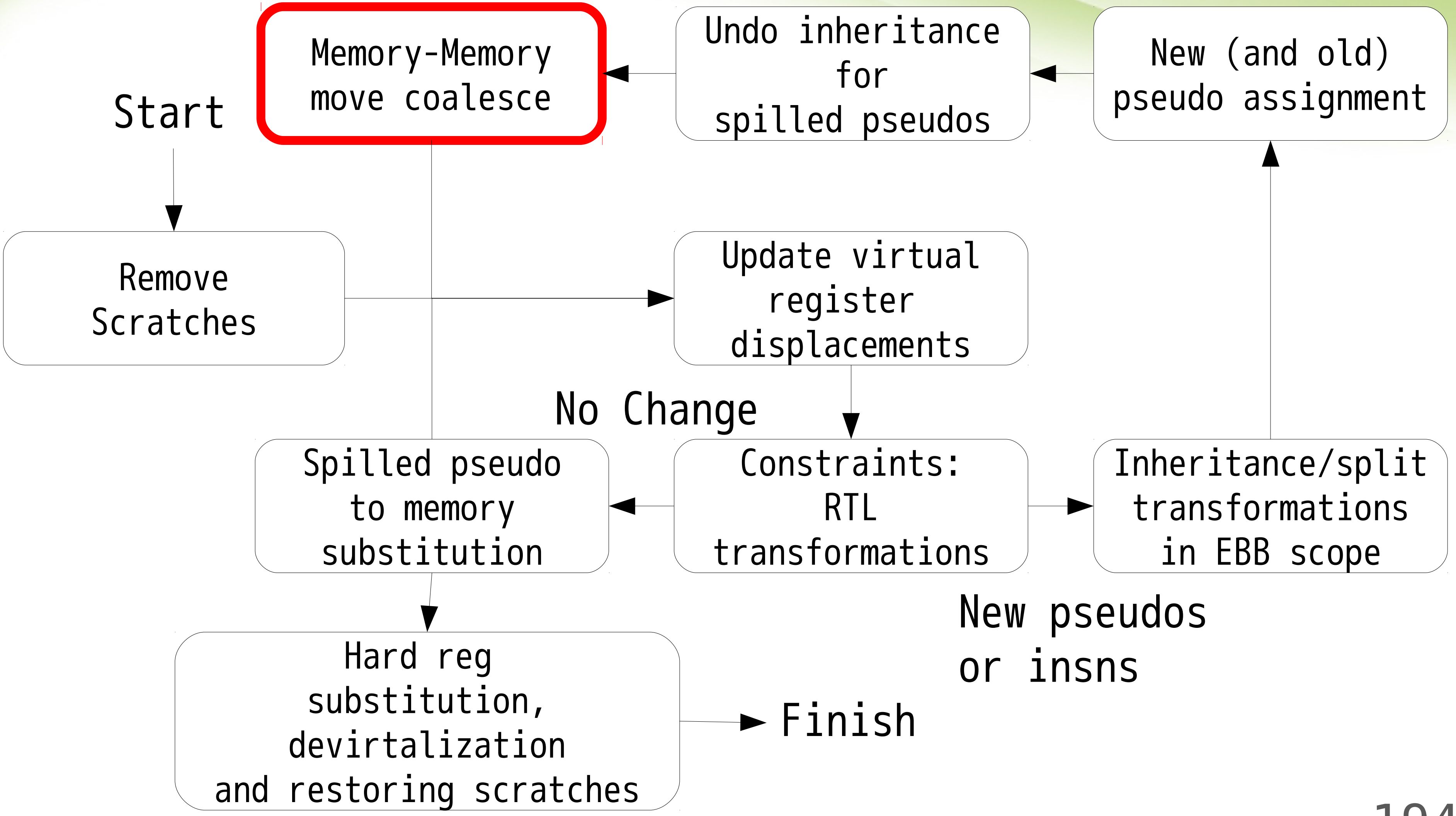
```
i(r50/r2) = i(r50/r2) + 1
```

```
sum(r41/r1) = sum(r51/r1)
D.1385(r46/r1) = sum(r41/r1)
<retval>(r47/r1) = D.1385(r46/r1)
$r0 = <retval>(r47/r1)
```

LRA


Start
Memory-Memory move coalesce
Undo inheritance for spilled pseudos
New (and old) pseudo assignment
Remove Scratches
Update virtual register displacements
No Change
Spilled pseudo to memory substitution
Constraints: RTL transformations
Inheritance/split transformations in EBB scope
New pseudos or insns
Hard reg substitution, devirtualization and restoring scratches
Finish

LRA



Start

Memory-Memory move coalesce

Undo inheritance for spilled pseudos

New (and old) pseudo assignment

Remove Scratches

Update virtual register displacements

No Change

Spilled pseudo to memory substitution

Constraints: RTL transformations

Inheritance/split transformations in EBB scope

New pseudos or insns

Hard reg substitution, devirtualization and restoring scratches

Finish

Memory-Memory move coalesce

- 假設一道 Move 指令的來源與目的的 Allocno 皆分配到 Memory（Spill）

a(r53/M_a) = b(r43/M_b)

...

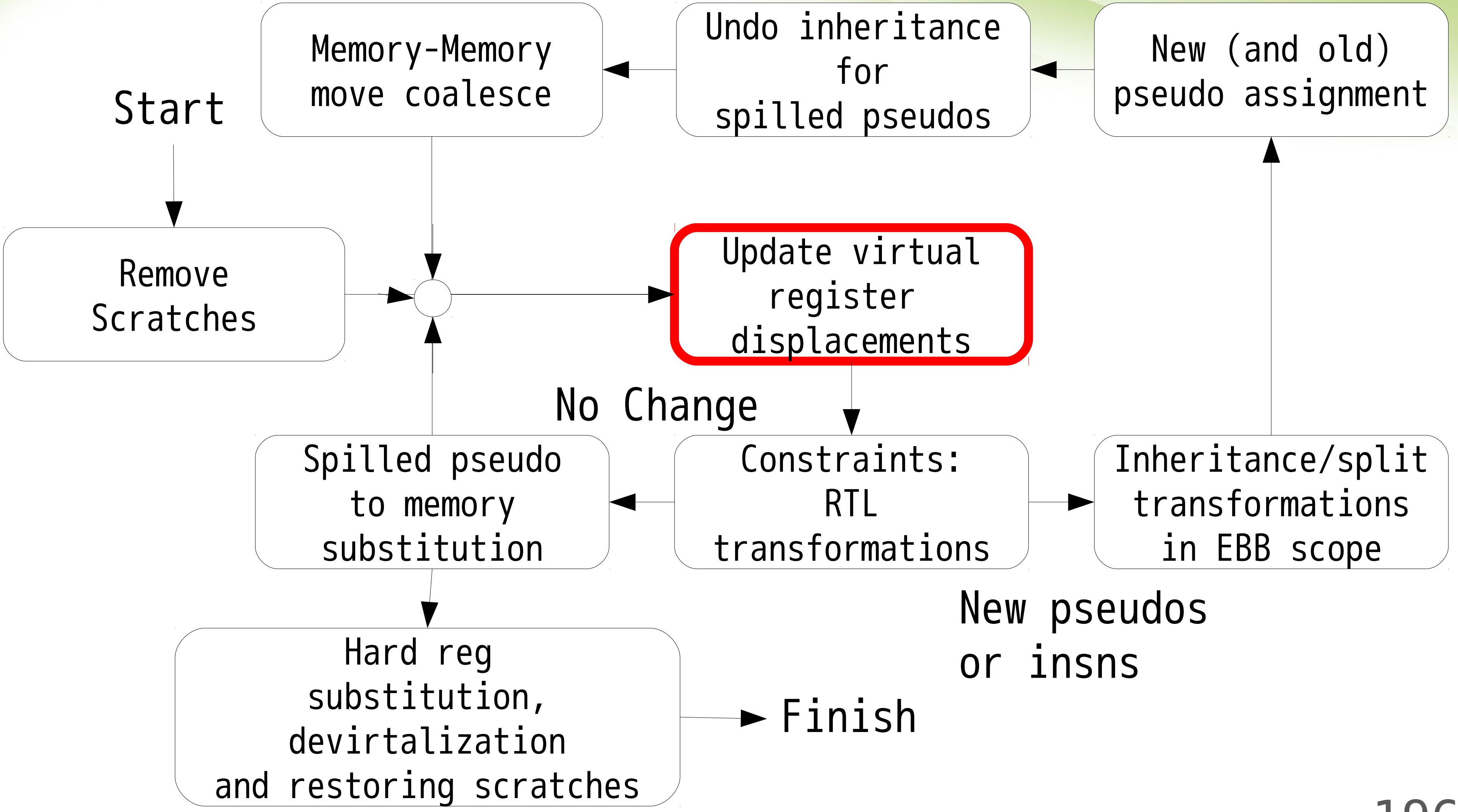
a(r55/$r1) = a(r53/$M_a$)

⇓

...

a(r55/$r1) = b(r43/$M_b$)

LRA



Start

Remove Scratches

Memory-Memory move coalesce

Undo inheritance for spilled pseudos

New (and old) pseudo assignment

Update virtual register displacements

No Change

Spilled pseudo to memory substitution

Constraints: RTL transformations

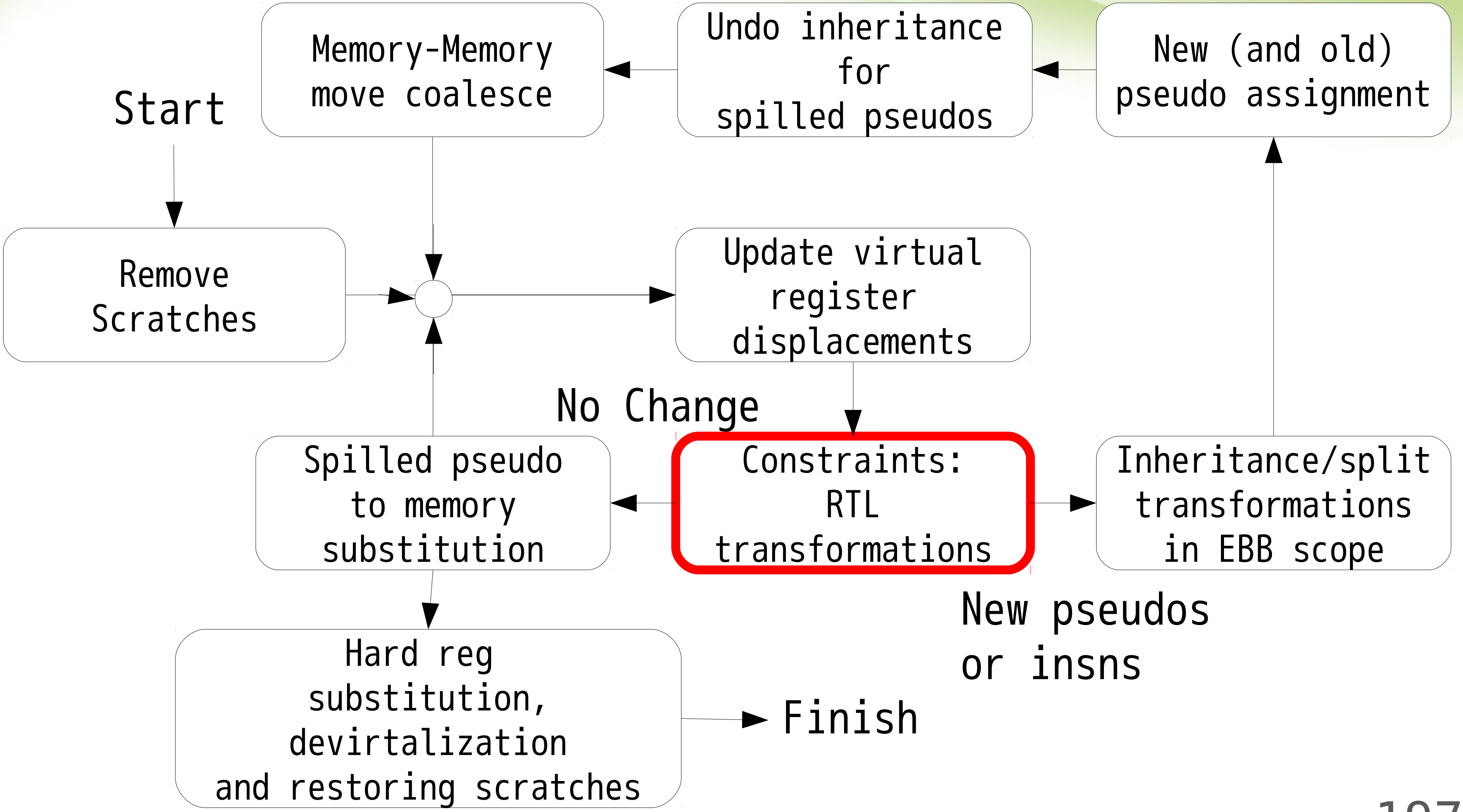
Inheritance/split transformations in EBB scope

New pseudos or insns

Hard reg substitution, devirtualization and restoring scratches

Finish

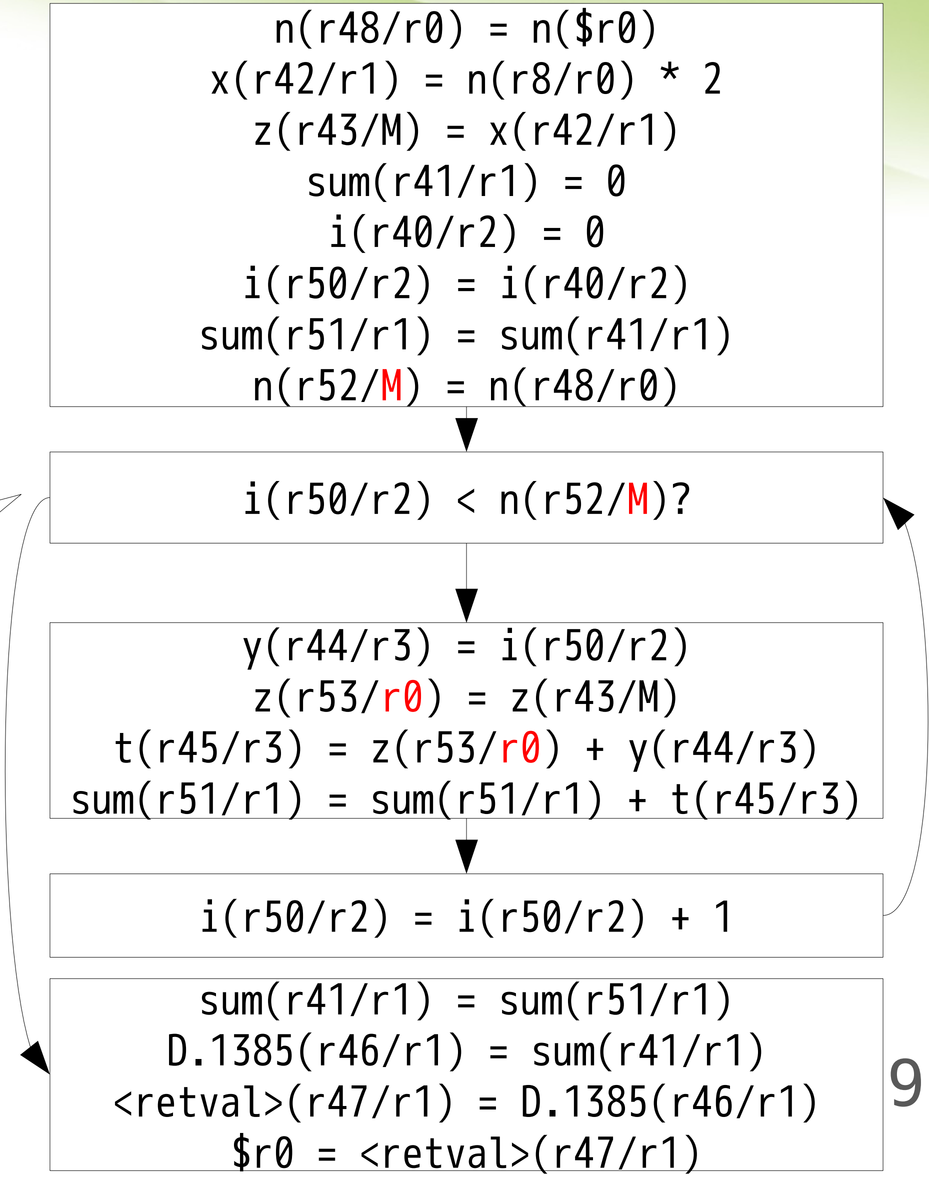
LRA


Start
Memory-Memory move coalesce
Undo inheritance for spilled pseudos
New (and old) pseudo assignment
Remove Scratches
Update virtual register displacements
No Change
Spilled pseudo to memory substitution
Constraints: RTL transformations
Inheritance/split transformations in EBB scope
New pseudos or insns
Hard reg substitution, devirtualization and restoring scratches
Finish

LRA Constraint #2

比較指令只能 Reg 跟 Reg 比
不符合 Constraint

nds32 中，比較指令只有
Reg 與 Reg 相比
無 Reg 與 Mem 比



```
n(r48/r0) = n($r0)
x(r42/r1) = n(r8/r0) * 2
z(r43/M) = x(r42/r1)
sum(r41/r1) = 0
i(r40/r2) = 0
i(r50/r2) = i(r40/r2)
sum(r51/r1) = sum(r41/r1)
n(r52/M) = n(r48/r0)
```

```
i(r50/r2) < n(r52/M)?
```

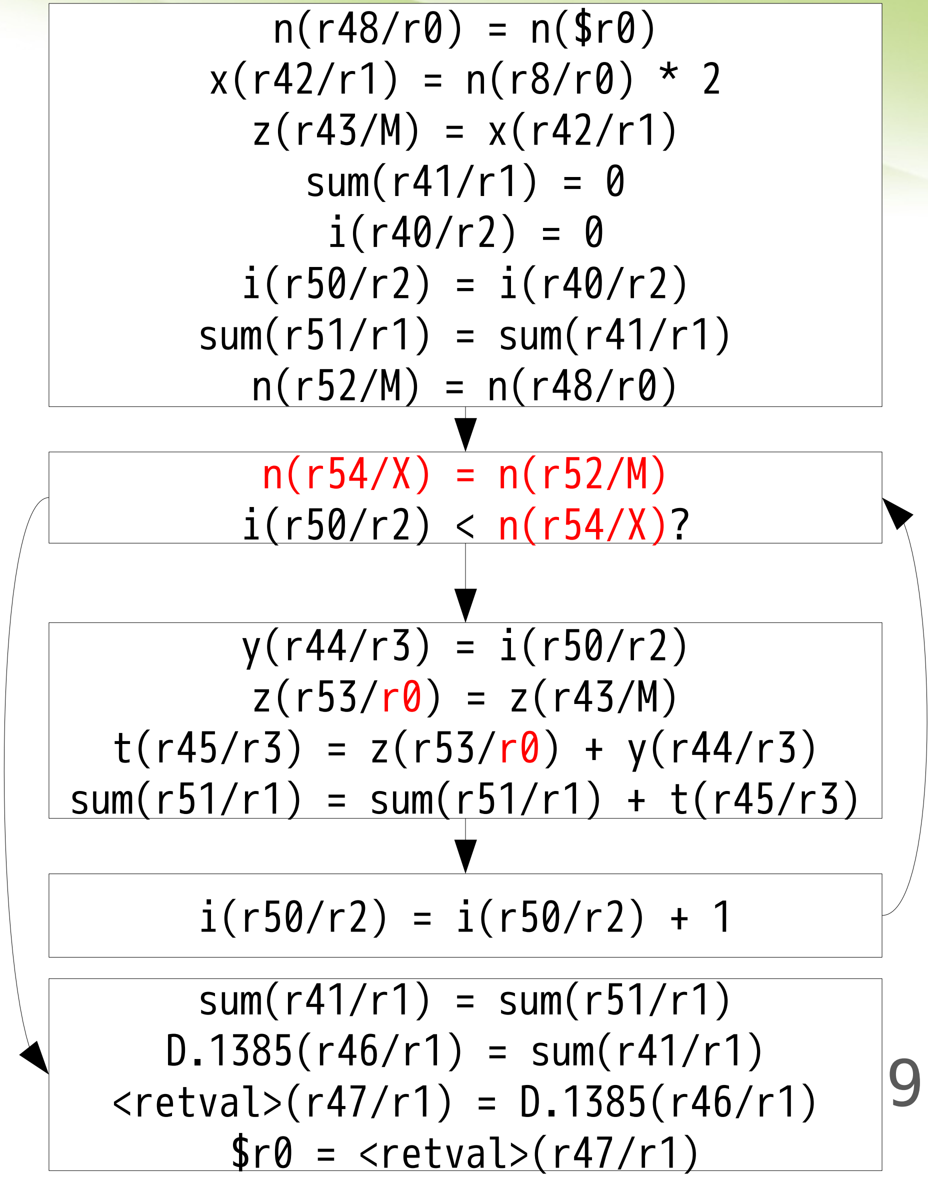
```
y(r44/r3) = i(r50/r2)
z(r53/r0) = z(r43/M)
t(r45/r3) = z(r53/r0) + y(r44/r3)
sum(r51/r1) = sum(r51/r1) + t(r45/r3)
```

```
i(r50/r2) = i(r50/r2) + 1
```

```
sum(r41/r1) = sum(r51/r1)
D.1385(r46/r1) = sum(r41/r1)
<retval>(r47/r1) = D.1385(r46/r1)
$r0 = <retval>(r47/r1)
```

LRA Constraint #2

比較指令只能 Reg 跟 Reg 比
不符合 Constraint



```
n(r48/r0) = n($r0)
x(r42/r1) = n(r8/r0) * 2
z(r43/M) = x(r42/r1)
sum(r41/r1) = 0
i(r40/r2) = 0
i(r50/r2) = i(r40/r2)
sum(r51/r1) = sum(r41/r1)
n(r52/M) = n(r48/r0)
```

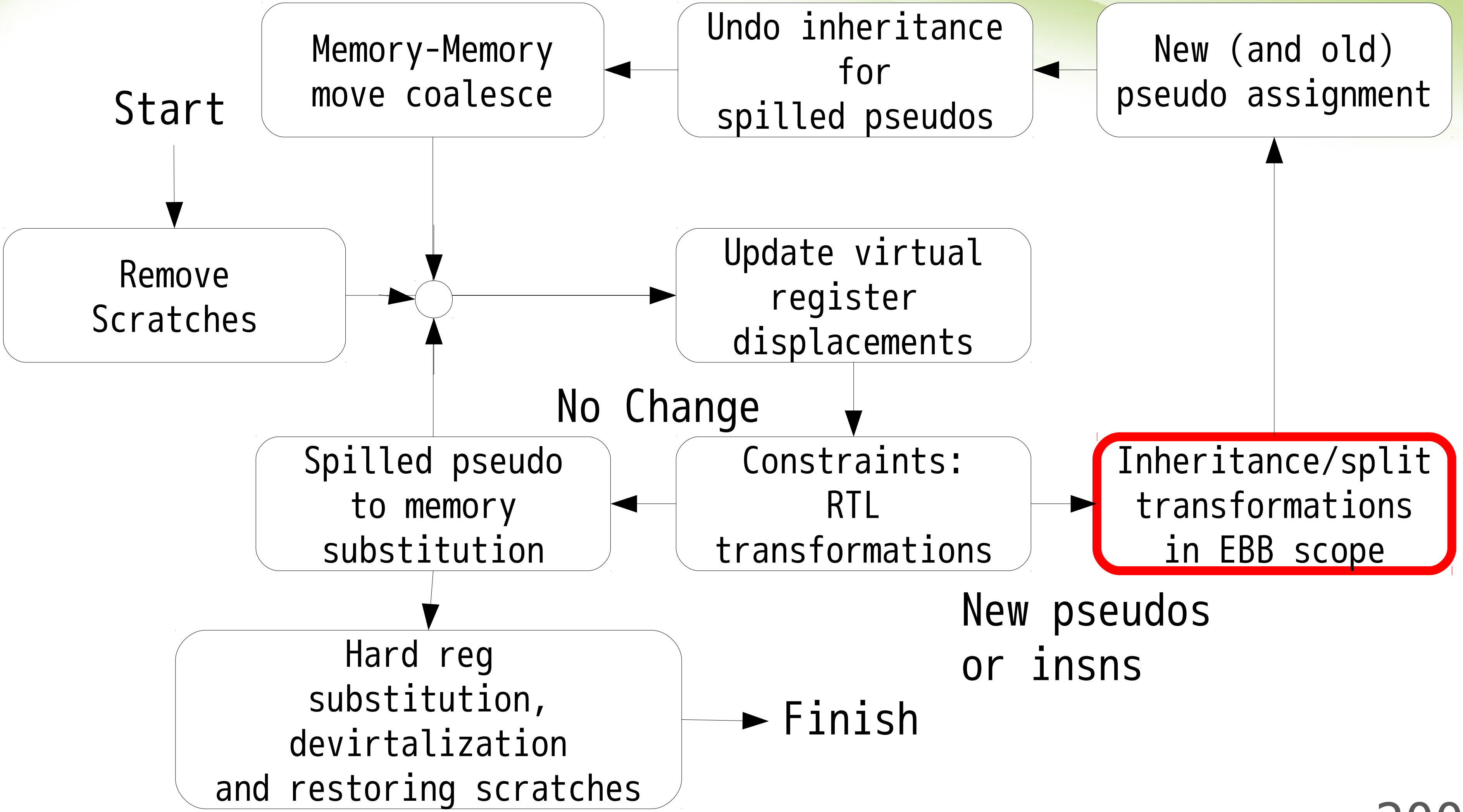
```
n(r54/X) = n(r52/M)
i(r50/r2) < n(r54/X)?
```

```
y(r44/r3) = i(r50/r2)
z(r53/r0) = z(r43/M)
t(r45/r3) = z(r53/r0) + y(r44/r3)
sum(r51/r1) = sum(r51/r1) + t(r45/r3)
```

```
i(r50/r2) = i(r50/r2) + 1
```

```
sum(r41/r1) = sum(r51/r1)
D.1385(r46/r1) = sum(r41/r1)
<retval>(r47/r1) = D.1385(r46/r1)
$r0 = <retval>(r47/r1)
```

LRA



Start

Remove Scratches

Memory-Memory move coalesce

Undo inheritance for spilled pseudos

New (and old) pseudo assignment

Update virtual register displacements

No Change

Spilled pseudo to memory substitution

Constraints: RTL transformations

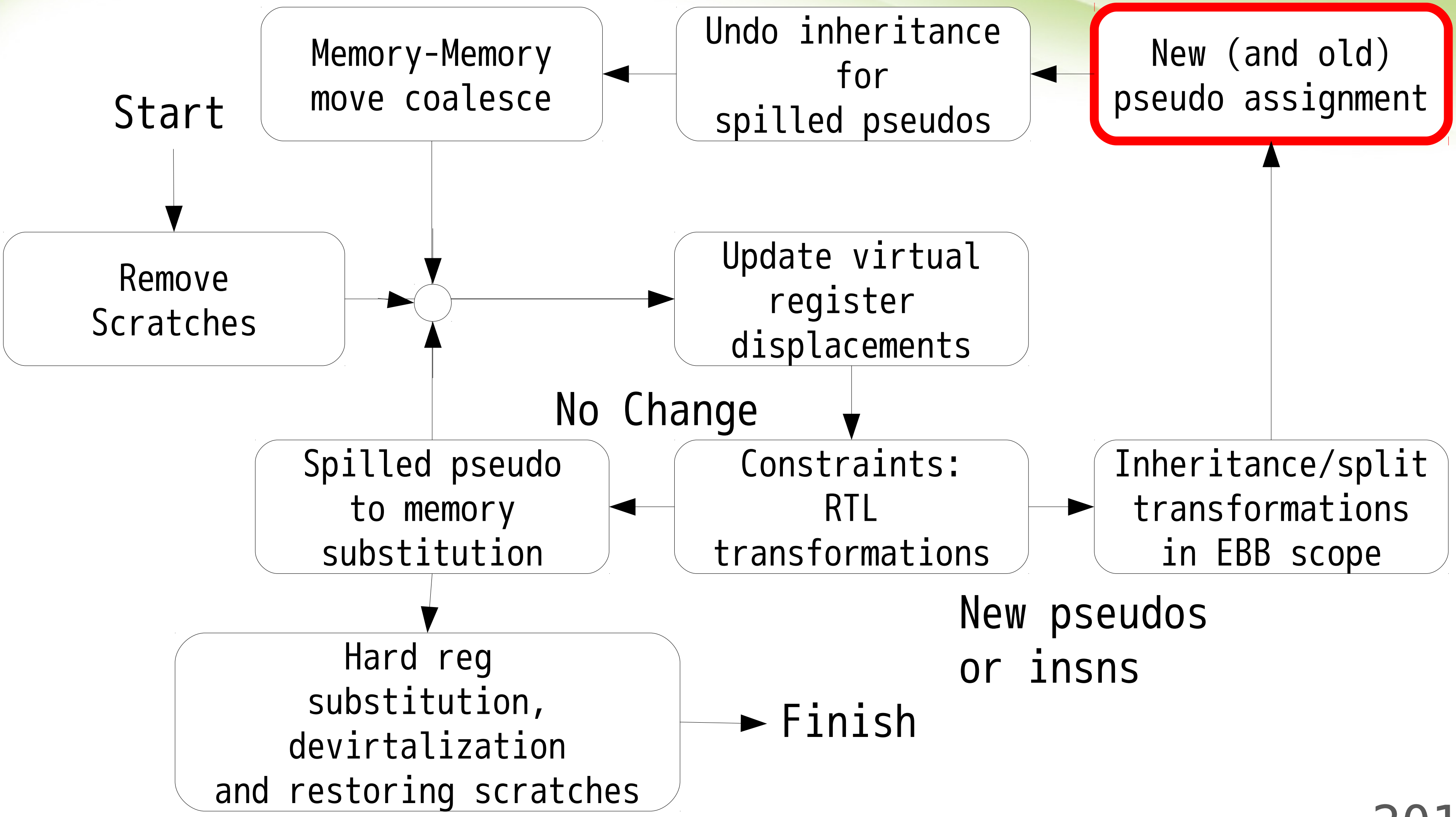
Inheritance/split transformations in EBB scope

New pseudos or insns

Hard reg substitution, devirtualization and restoring scratches

Finish

LRA



Start

Remove Scratches

Memory-Memory move coalesce

Undo inheritance for spilled pseudos

New (and old) pseudo assignment

Update virtual register displacements

No Change

Spilled pseudo to memory substitution

Constraints: RTL transformations

Inheritance/split transformations in EBB scope

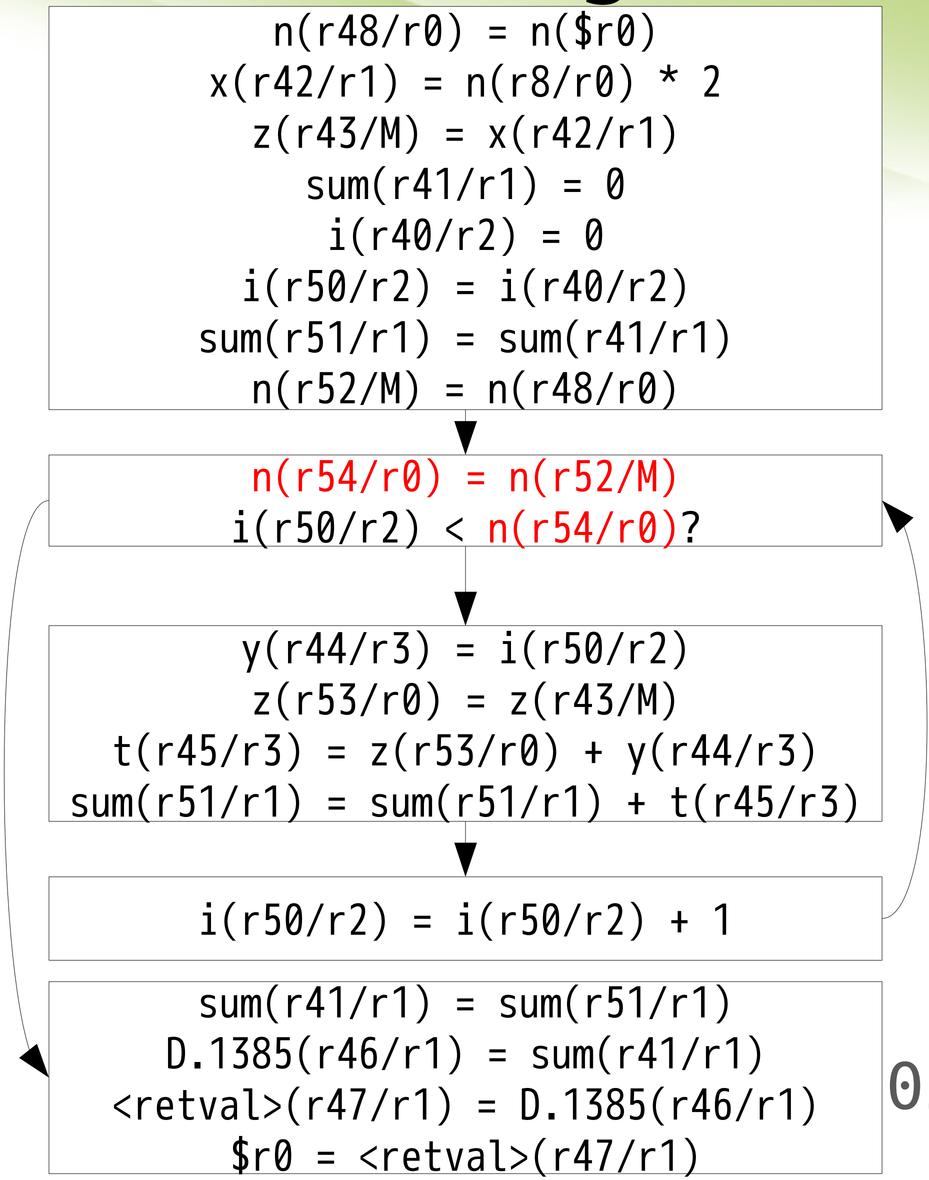
New pseudos or insns

Hard reg substitution, devirtualization and restoring scratches

Finish

LRA Assign #2

很幸運直接撿到剛被搶走的 r0 可以用



```
n(r48/r0) = n($r0)
x(r42/r1) = n(r8/r0) * 2
z(r43/M) = x(r42/r1)
sum(r41/r1) = 0
i(r40/r2) = 0
i(r50/r2) = i(r40/r2)
sum(r51/r1) = sum(r41/r1)
n(r52/M) = n(r48/r0)
```

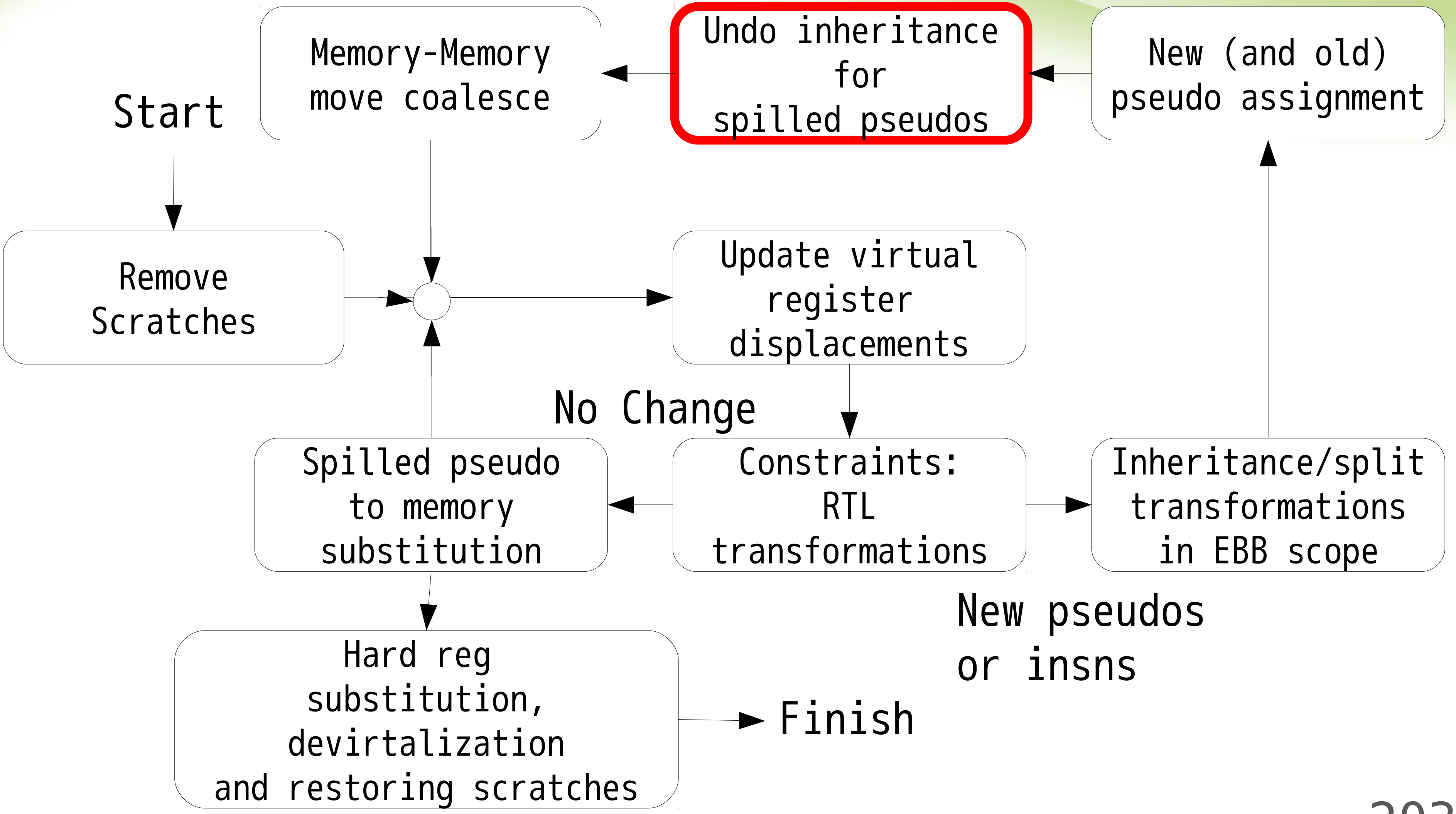
```
n(r54/r0) = n(r52/M)
i(r50/r2) < n(r54/r0)?
```

```
y(r44/r3) = i(r50/r2)
z(r53/r0) = z(r43/M)
t(r45/r3) = z(r53/r0) + y(r44/r3)
sum(r51/r1) = sum(r51/r1) + t(r45/r3)
```

```
i(r50/r2) = i(r50/r2) + 1
```

```
sum(r41/r1) = sum(r51/r1)
D.1385(r46/r1) = sum(r41/r1)
<retval>(r47/r1) = D.1385(r46/r1)
$r0 = <retval>(r47/r1)
```

LRA


Start

Remove Scratches

Memory-Memory move coalesce

Undo inheritance for spilled pseudos

New (and old) pseudo assignment

Update virtual register displacements

No Change

Spilled pseudo to memory substitution

Constraints: RTL transformations

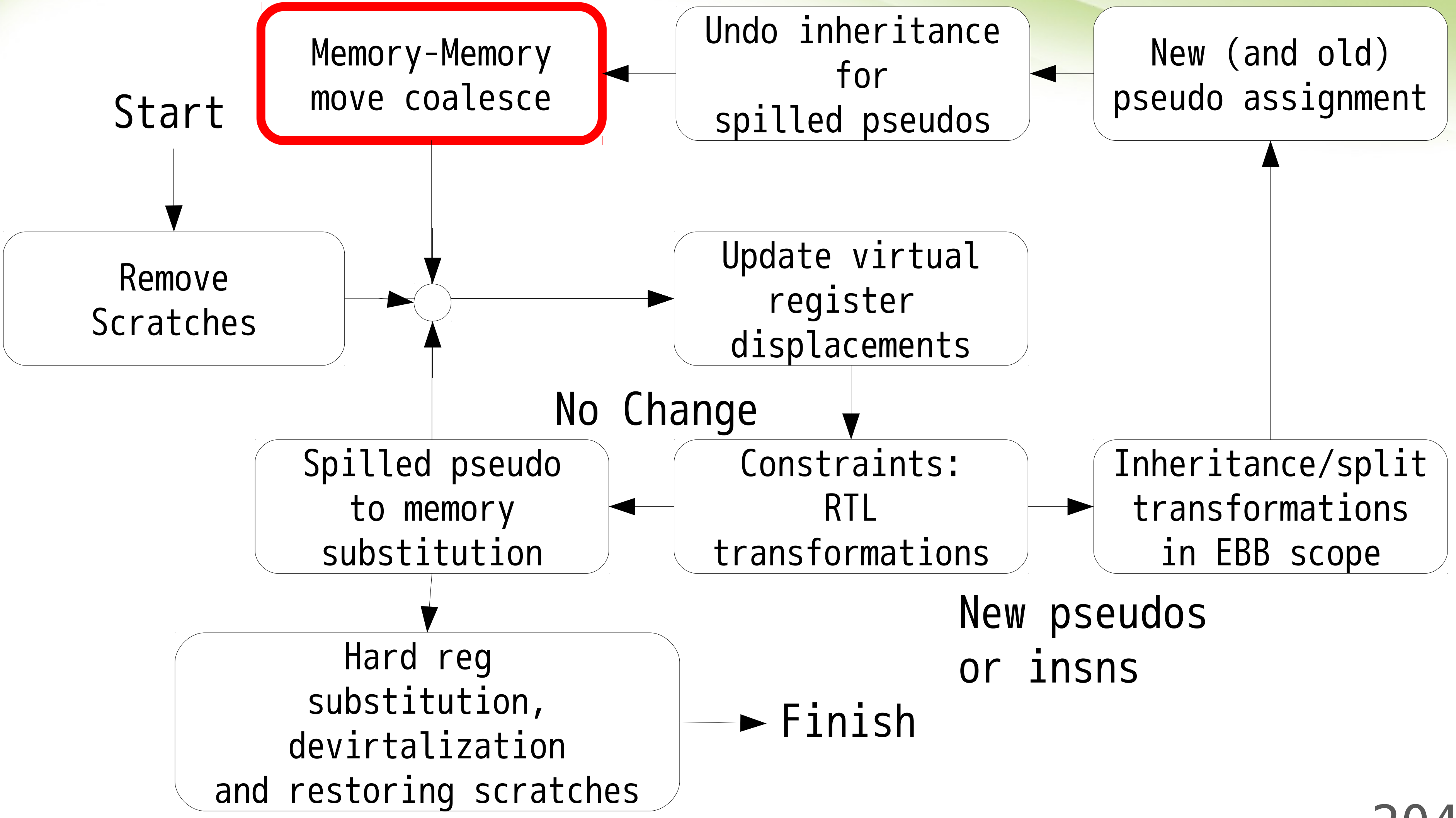
Inheritance/split transformations in EBB scope

New pseudos or insns

Hard reg substitution, devirtualization and restoring scratches

Finish

LRA



Start

Remove Scratches

Memory-Memory move coalesce

Undo inheritance for spilled pseudos

New (and old) pseudo assignment

Update virtual register displacements

No Change

Spilled pseudo to memory substitution

Constraints: RTL transformations

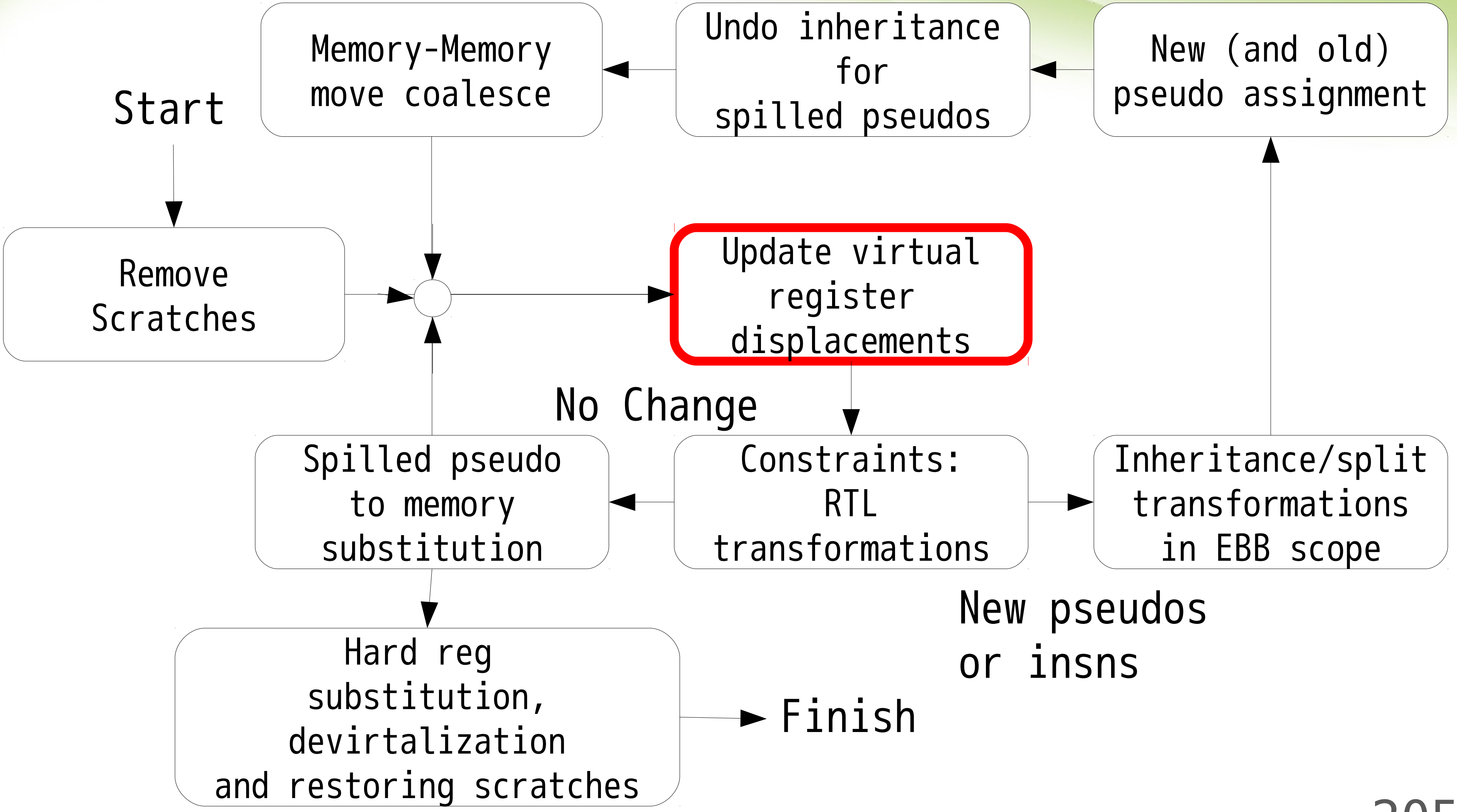
Inheritance/split transformations in EBB scope

New pseudos or insns

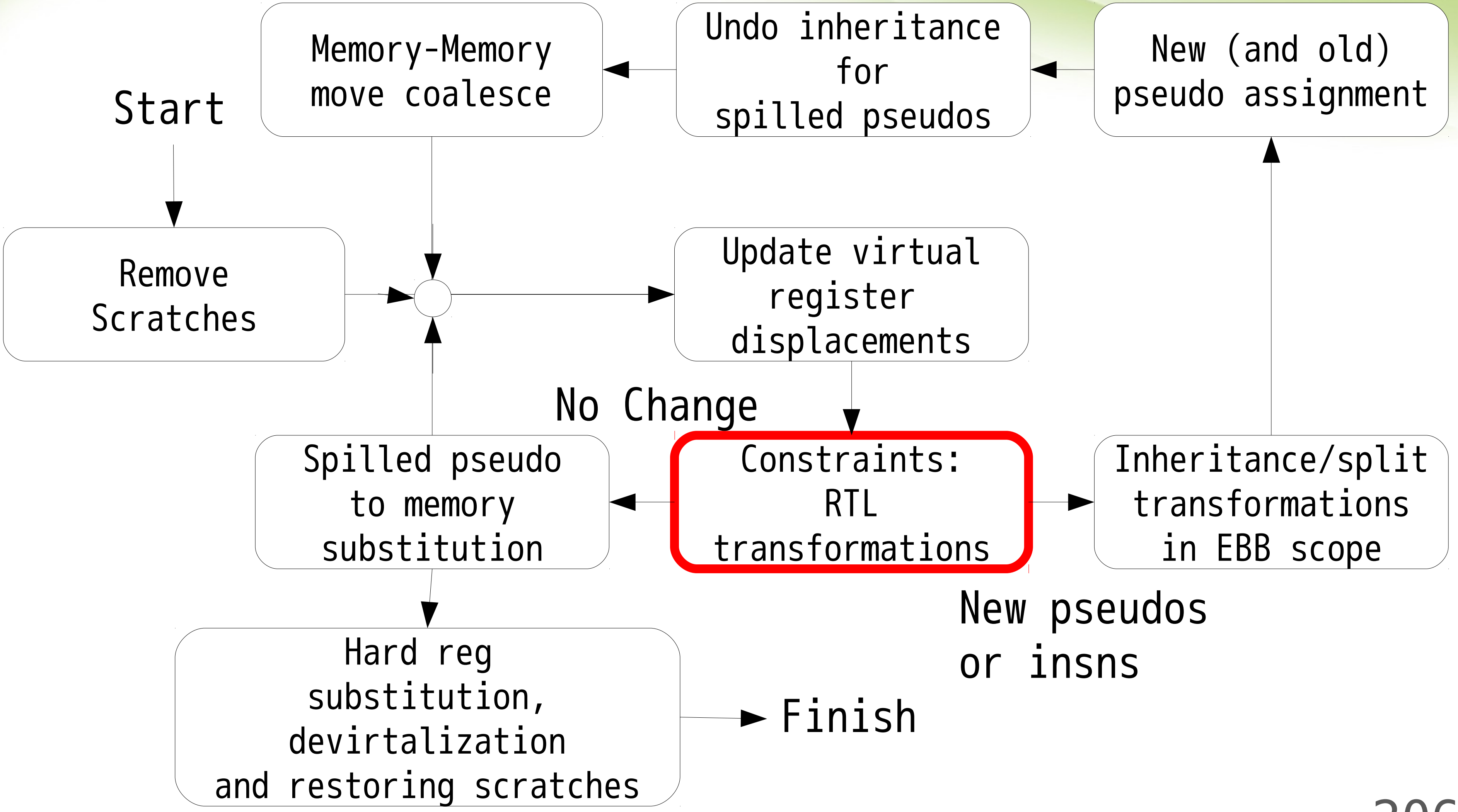
Hard reg substitution, devirtualization and restoring scratches

Finish

LRA

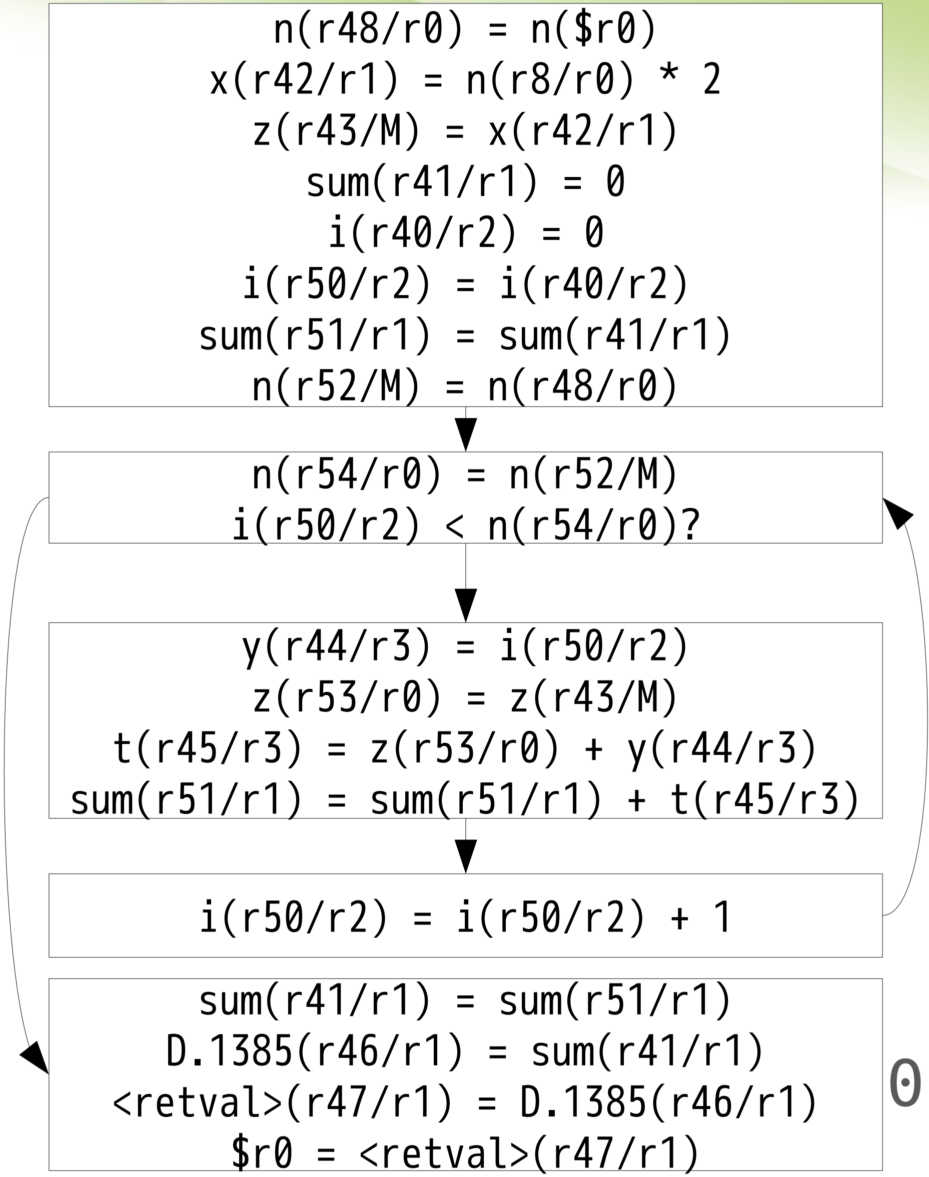

Start
Remove Scratches
Memory-Memory move coalesce
Undo inheritance for spilled pseudos
New (and old) pseudo assignment
Update virtual register displacements
No Change
Spilled pseudo to memory substitution
Constraints: RTL transformations
Inheritance/split transformations in EBB scope
New pseudos or insns
Hard reg substitution, devirtualization and restoring scratches
Finish

LRA


Start
Remove Scratches
Memory-Memory move coalesce
Undo inheritance for spilled pseudos
New (and old) pseudo assignment
Update virtual register displacements
No Change
Spilled pseudo to memory substitution
Constraints: RTL transformations
Inheritance/split transformations in EBB scope
New pseudos or insns
Hard reg substitution, devirtualization and restoring scratches
Finish

LRA Constraint #3

整個掃描過後發現所有指令符合 Constraint!



```
n(r48/r0) = n($r0)
x(r42/r1) = n(r8/r0) * 2
z(r43/M) = x(r42/r1)
sum(r41/r1) = 0
i(r40/r2) = 0
i(r50/r2) = i(r40/r2)
sum(r51/r1) = sum(r41/r1)
n(r52/M) = n(r48/r0)
```

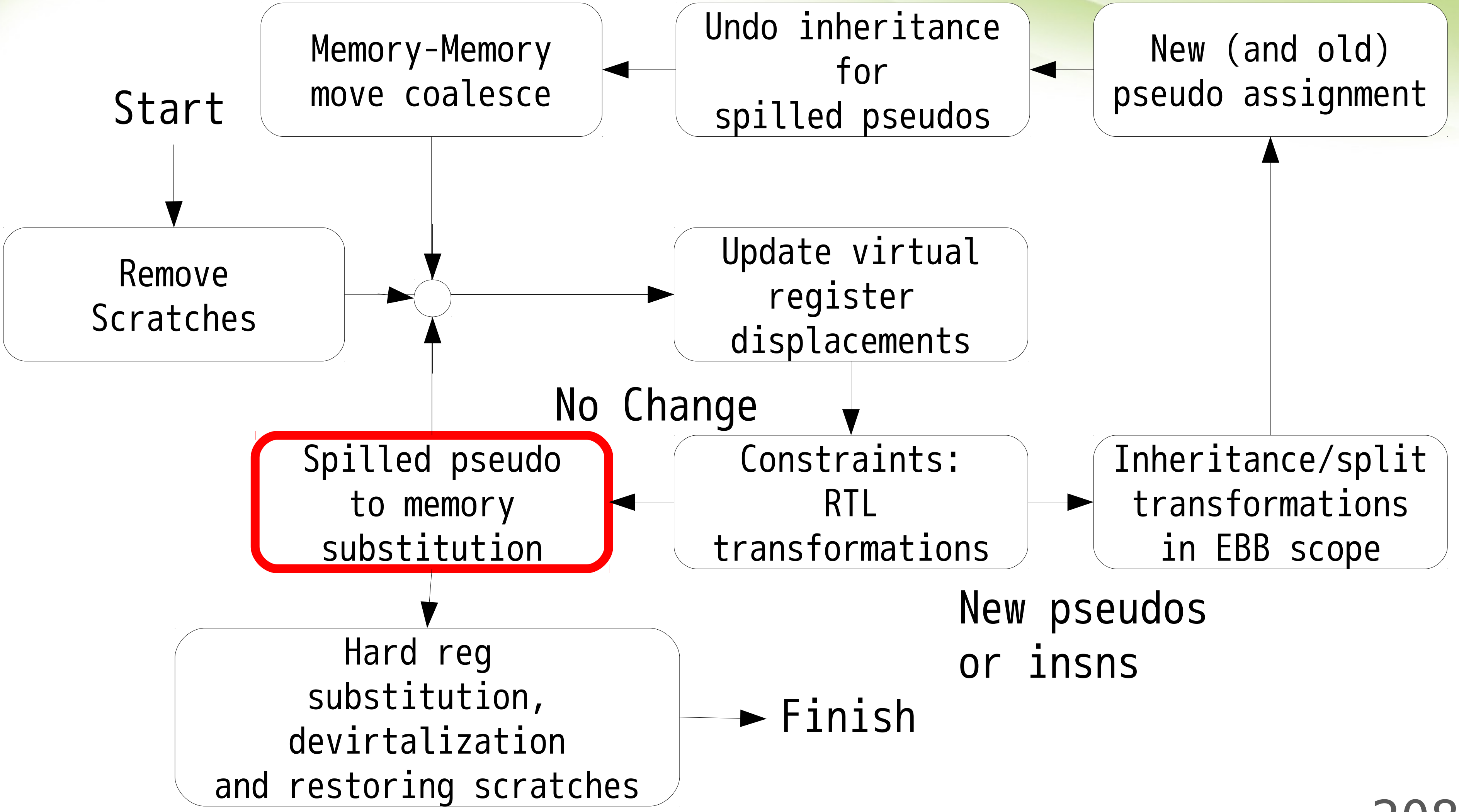
```
n(r54/r0) = n(r52/M)
i(r50/r2) < n(r54/r0)?
```

```
y(r44/r3) = i(r50/r2)
z(r53/r0) = z(r43/M)
t(r45/r3) = z(r53/r0) + y(r44/r3)
sum(r51/r1) = sum(r51/r1) + t(r45/r3)
```

```
i(r50/r2) = i(r50/r2) + 1
```

```
sum(r41/r1) = sum(r51/r1)
D.1385(r46/r1) = sum(r41/r1)
<retval>(r47/r1) = D.1385(r46/r1)
$r0 = <retval>(r47/r1)
```

LRA


Start

Memory-Memory move coalesce

Undo inheritance for spilled pseudos

New (and old) pseudo assignment

Remove Scratches

Update virtual register displacements

No Change

Spilled pseudo to memory substitution

Constraints: RTL transformations

Inheritance/split transformations in EBB scope

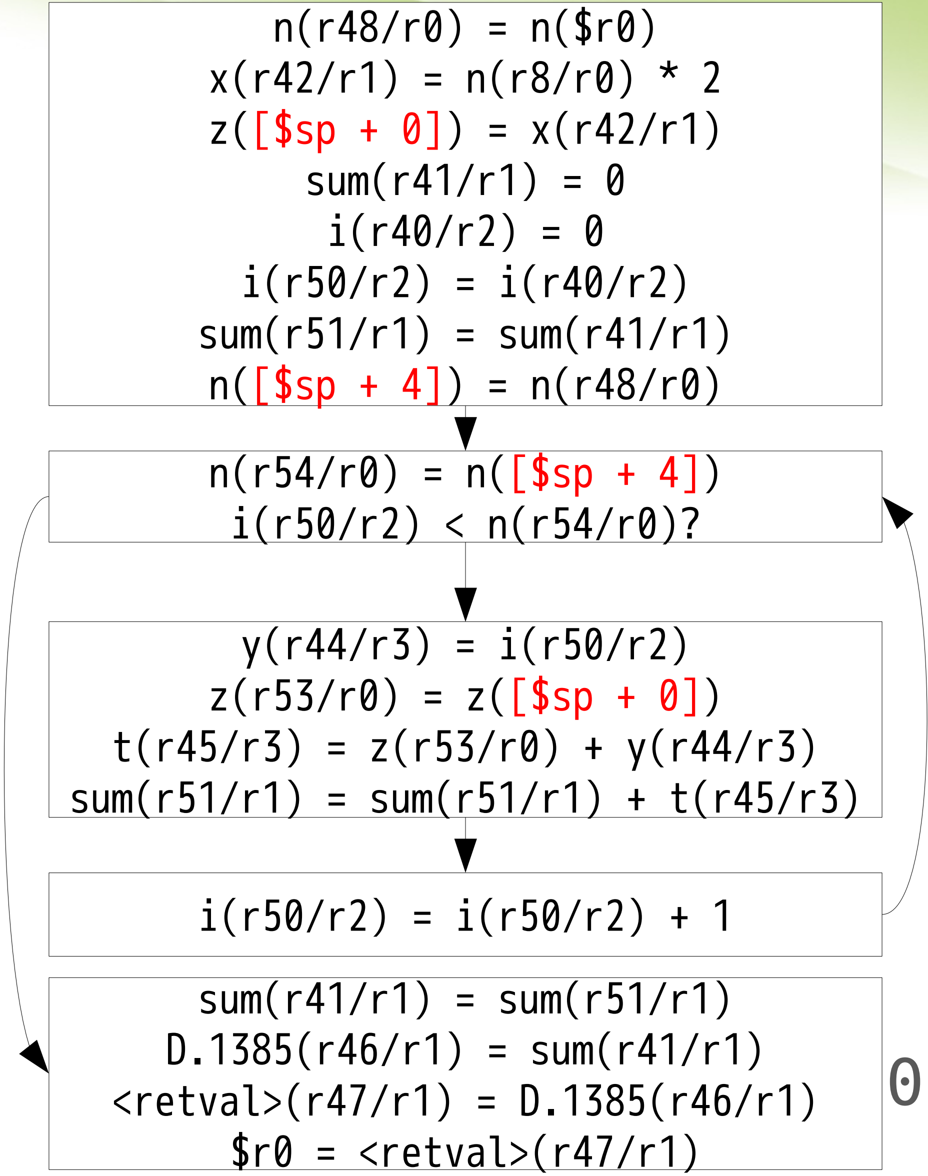
New pseudos or insns

Hard reg substitution, devirtualization and restoring scratches

Finish

LRA Memory Substitution

將所有 Spill 的 Pseudo Register 替換成 Memory!



```
n(r48/r0) = n($r0)
x(r42/r1) = n(r8/r0) * 2
z([$sp + 0]) = x(r42/r1)
sum(r41/r1) = 0
i(r40/r2) = 0
i(r50/r2) = i(r40/r2)
sum(r51/r1) = sum(r41/r1)
n([$sp + 4]) = n(r48/r0)
```

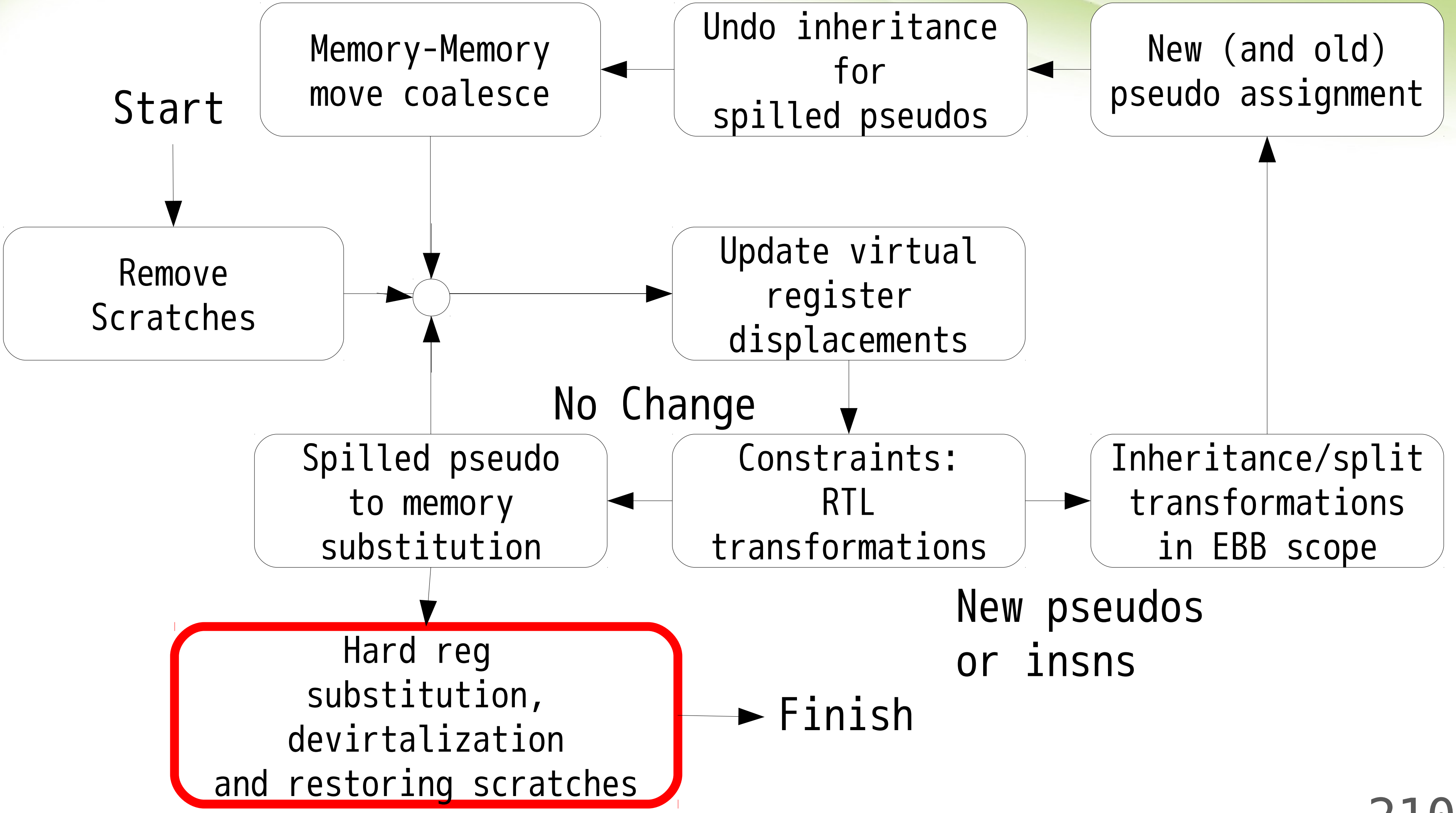
```
n(r54/r0) = n([$sp + 4])
i(r50/r2) < n(r54/r0)?
```

```
y(r44/r3) = i(r50/r2)
z(r53/r0) = z([$sp + 0])
t(r45/r3) = z(r53/r0) + y(r44/r3)
sum(r51/r1) = sum(r51/r1) + t(r45/r3)
```

```
i(r50/r2) = i(r50/r2) + 1
```

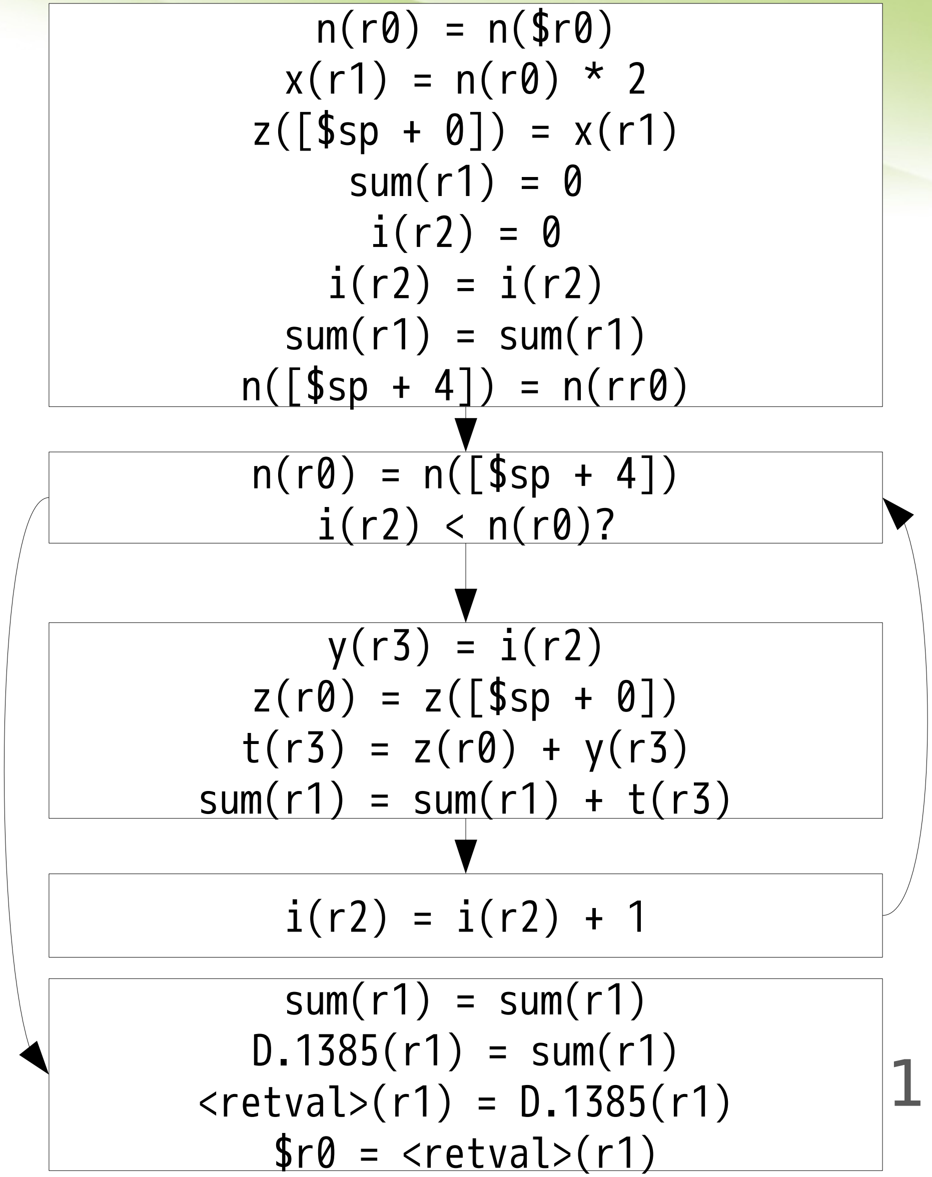
```
sum(r41/r1) = sum(r51/r1)
D.1385(r46/r1) = sum(r41/r1)
<retval>(r47/r1) = D.1385(r46/r1)
$r0 = <retval>(r47/r1)
```

LRA


Start
Memory-Memory move coalesce
Undo inheritance for spilled pseudos
New (and old) pseudo assignment
Remove Scratches
Update virtual register displacements
No Change
Spilled pseudo to memory substitution
Constraints: RTL transformations
Inheritance/split transformations in EBB scope
New pseudos or insns
Hard reg substitution, devirtualization and restoring scratches
Finish

LRA Register Substitution

將所有 Pseudo Register
替換成 Hard Register!



```
n(r0) = n($r0)
x(r1) = n(r0) * 2
z([$sp + 0]) = x(r1)
sum(r1) = 0
i(r2) = 0
i(r2) = i(r2)
sum(r1) = sum(r1)
n([$sp + 4]) = n(rr0)
```

```
n(r0) = n([$sp + 4])
i(r2) < n(r0)?
```

```
y(r3) = i(r2)
z(r0) = z([$sp + 0])
t(r3) = z(r0) + y(r3)
sum(r1) = sum(r1) + t(r3)
```

```
i(r2) = i(r2) + 1
```

```
sum(r1) = sum(r1)
D.1385(r1) = sum(r1)
<retval>(r1) = D.1385(r1)
$r0 = <retval>(r1)
```

reload_completed != 0

在 gcc 中，當 reload_complete 不等於 0 時
代表 Register Allocation 完畢，
並且所有 RTL 都必須是 Strict RTL
隨時都可以輸出成 Assembly Language!

總結

- IRA/LRA 雖然龐大， 但循著理論走會容易理解很多

- RA 在整個編譯階段末段， 但 RA 結果的優劣對於效能與程式大小有相當大的影響