8 静态单赋值

徐辉，xuh@fudan.edu.cn

本章学习目标：

- ⋆⋆ 了解静态单赋值形式
- ⋆⋆⋆ 掌握基于混沌迭代的数据流分析方法
- ⋆⋆⋆ 掌握静态单赋值形式的构造方法

8.1 静态单赋值

静态单赋值（SSA, Static Single Assignment） [1] 是一种特殊的线性中间表示（IR）。其核心目标是更清晰地刻画变量的定义—使用（def-use）关系，从而便于后续的程序分析与代码优化。SSA 通常满足以下要求：

- **单次定义：**每个标识符对应一个虚拟寄存器，且只能被定义（def）或赋值一次；若变量值发生变化，则必须重新定义新的标识符以表示更新后的值。
- **Phi 指令：**当控制流汇合导致某变量在同一使用点（use）可能对应多个不同来源的定义时，需要使用 phi 指令统一表示这些可能的取值来源。
- **最优化：**应尽可能减少 phi 指令的数量，以简化数据流关系并降低 IR 的冗余。

上一章中使用的 LLVM IR 已经满足“标识符仅定义一次”的要求，但尚未引入 phi 指令。在不同控制流路径产生不同变量值时，我们采用的是 `store-load` 机制进行处理，而非使用 phi 指令。接下来，我们将讨论如何将上一章 LLVM IR 中基于 `load/store` 的表示方式逐步转换为基于 phi 指令的表示，并最终构造出最优的 SSA 形式。

8.2 基于冗余消除的 SSA 构造方法

8.2.1 消除 IR 中冗余的 load/store

在 AST 翻译为 IR 的过程中，为了降低 def-use 关系分析的复杂度，我们要求变量在使用前必须先执行 load，并在变量值更新后立即执行 store。这种策略虽然简化了前端生成逻辑，但也会引入大量冗余的 load/store 指令。本节将采用基于混沌迭代（Chaotic Iteration）的数据流分析方法，对 IR 中冗余的 load/store 操作进行消除，从而为后续构造 SSA 形式奠定基础。

```
define i32 @fac(i32 %0) {
bb0:
    %n = alloca i32
    %r = alloca i32
    store i32 %0, i32* %n
    store i32 1, i32* %r
    br label %bb1
bb1:
    %t1 = load i32, i32* %n
    %t2 = icmp sgt i32 %t1, 0
    br i1 %t2, label %bb2, label %bb3
bb2:
    %t3 = load i32, i32* %r
    %t4 = load i32, i32* %n
    %t5 = mul i32 %t3, %t4
    store i32 %t5, i32* %r
    %t6 = load i32, i32* %n
    %t7 = sub i32 %t6, 1
    store i32 %t7, i32* %n
    br label %bb1
bb3:
    %t8 = load i32, i32* %r
    ret i32 %t8
}
```

代码 8.1: IR 代码示例：阶乘函数

消除冗余 load

代码 ?? 展示了一段 IR。由于变量 n 的值在代码块 bb1 中已经被加载到虚拟寄存器 %t1，且在到达代码块 bb2 的过程中没有发生对 n 的更新，因此 bb2 中再次将 n 加载到 %t4 和 %t6 的操作是冗余的，可以直接复用 %t1。其基本规律是：对于同一变量，若两次 load 之间不存在对应的 store 操作，则说明该变量的值未发生变化，因此后一次 load 为冗余操作，可以直接使用前一次 load 得到的虚拟寄存器；反之，若两次 load 之间存在 store，则说明变量值已经被更新，之前的 load 结果失效，必须重新加载。

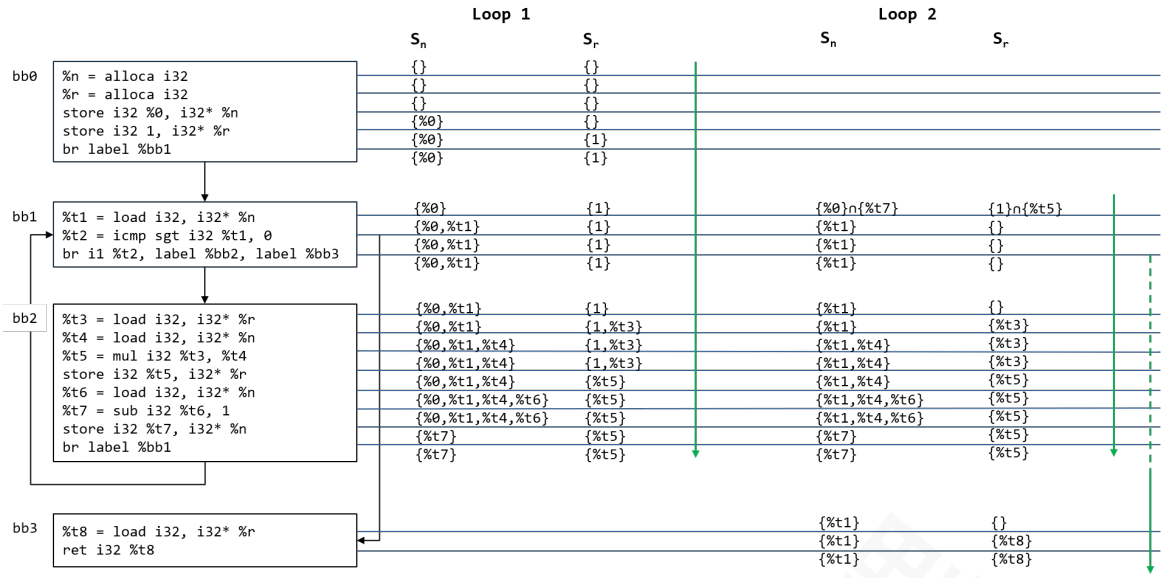


图 8.1: 冗余 load 指令分析

基于上述分析，我们可以总结出与该优化相关的指令及其影响，即表 8.1 中定义的 transfer 函数。对于线性的指令序列，可以直接将这些 transfer 函数依次作用于代码块中的各条指令；然而，当程序中存在控制流分支或循环时，还需要额外的数据流分析机制进行处理。

表 8.1: Transfer 函数定义：可用 load 指令分析

IR 指令	举例	Transfer 函数
`load`	`%t = load i32, i32* %x`	$S_x = S_x \cup \{t\}$
`store`	`store i32 %t, i32* %x`	$S_x = \{t\}$

混沌迭代（Chaotic Iteration）算法是一种处理控制流数据流分析的经典框架，可以根据具体分析任务设计不同的数据流状态与 transfer 操作。如算法 1 所示，该算法对每条指令 `i` 进行分析并计算其对应的 `OUT[i]`。若某程序点存在多个前驱节点，则需要对各前驱的分析结果进行合并。由于本节进行的是可用表达式类型的 Must Analysis，因此只有当某个变量对应的虚拟寄存器在所有前驱路径上均一致可用时，才能认为其在当前程序点可用，因此这里采用交集操作进行合并。将该算法应用于图 8.1 后，便可以求得每个程序节点处各变量对应的可用虚拟寄存器，从而为后续消除冗余 `load` 指令提供依据。

算法 1 混沌迭代算法：可用 load 指令分析

Require: IR and variables of a target function

for each $i \in irs$ **do**
  $IN[i] \leftarrow \{S_v = \emptyset \mid v \in Var\}$;
  $OUT[i] \leftarrow \{S_v = \emptyset \mid v \in Var\}$;
end for
repeat
  for each $i \in irs$ **do**
    for each $p \in \text{Predecessor}(i)$ **do**
      $IN[i] \leftarrow IN[i] \cap OUT[p]$;
    end for
    $OUT[i] \leftarrow \text{Transfer}(i)$;
  end for
until $IN[i]$ and $OUT[i]$ stop changing for all i

消除冗余 store

如果同一变量的两条 `store` 指令之间不存在对应的 `load` 操作，则前一条 `store` 的结果不会被读取，因此该 `store` 属于冗余操作，可以直接删除。与冗余 `load` 分析不同，该分析仅需维护一个“已经被 `store` 但尚未被 `load`”的变量集合，而无需为每个变量分别维护对应的可用虚拟寄存器集合。表 8.2 总结了不同指令对应的 transfer 函数。

表 8.2: Transfer 函数定义：可用 store 分析

IR 指令	举例	Transfer 函数
`store`	`store i32 %t, i32* %x`	$S = S \cup \{x\}$
`load`	`%t = load i32, i32* %x`	$S = S \setminus \{x\}$
`alloca`	`%x = alloca i32`	$S = S \setminus \{x\}$

算法 2 混沌迭代算法：可用 store 分析

Require: IR and variables of a target function

1: **for each** $i \in irs$ **do**
2: $IN[i] \leftarrow \emptyset$;
3: $OUT[i] \leftarrow \emptyset$;
4: **end for**
5: **repeat**
6: **for each** $i \in irs$ **do**
7: **for each** $s \in$Successor(i) **do**
8: $OUT[i] \leftarrow OUT[i] \cap IN[s]$;
9: **end for**
10: $IN[i] \leftarrow$ Transfer(i);
11: **end for**
12: **until** $IN[i]$ and $OUT[i]$ stop changing for all i

根据算法 2 对 IR 控制流图进行逆向遍历，即可识别所有满足条件的冗余 `store` 操作。由于图 8.2 中不存在“store 后未被读取便再次 store”的情况，因此其中不包含冗余 `store` 指令。

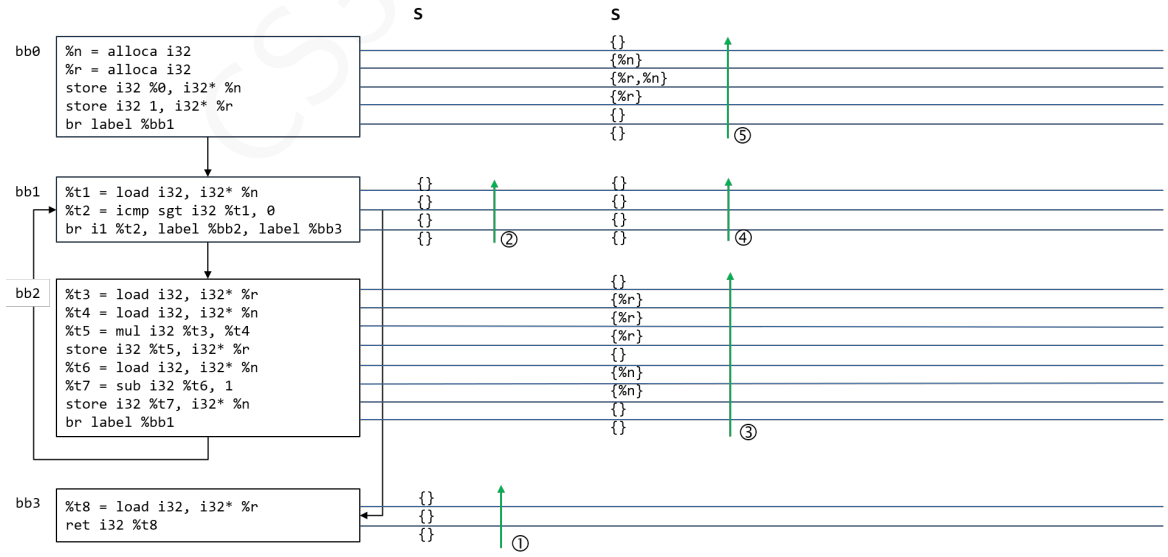


图 8.2: 冗余 load 指令分析

8.2.2 转换为静态单赋值形式

这一步的目标是消除 IR 中所有针对局部变量的 `store/load` 指令，使变量值完全通过虚拟寄存器传递，从而构造纯 SSA 形式的 IR。其中的关键问题在于：某些 `load` 指令可能对应来自不同控制流路径的多个定义，此时无法直接使用单一虚拟寄存器替换，而需要引入 phi 指令对不同来源的值进行统一表示。

以图 8.3 为例，代码块 `bb1` 中的 `load`（`%n`）可能对应两种不同的定义来源：

- 若控制流路径为 `bb0` → `bb1`，则其值来源于 `bb0` 中的 `%0`；
- 若控制流路径为 `bb2` → `bb1`，则其值来源于 `bb2` 中的 `%t7`。

由于该 `load` 的取值依赖于控制流路径，因此需要使用 phi 指令对这些可能的定义进行合并。下面介绍从 LLVM IR 转换到 SSA 形式的方法，其主要分为两个步骤：

1) 值流（value-flow）分析：识别每个 `load` 指令在不同控制流路径下可能对应的变量定义来源；

2) 使用 phi 指令替换 `store/load`。

值流分析

对于值流分析，我们同样采用混沌迭代（Chaotic Iteration）方法分析 `store` 指令对 def-use 关系的影响。具体而言，通过正向遍历控制流图，在遇到 `store` 指令时应用表 8.3 中定义的 transfer 函数；当遇到控制流合并节点时，则对来自不同前驱路径的分析结果取并集。经过迭代直至达到不动点后，便可以得到各程序点处变量可能对应的定义来源。图 8.3 展示了该分析的结果。

表 8.3: Transfer 函数定义：值流分析

IR 指令	举例	Transfer 函数
`store`	`store i32 %t, i32* %x`	$S_x = \{t\}$

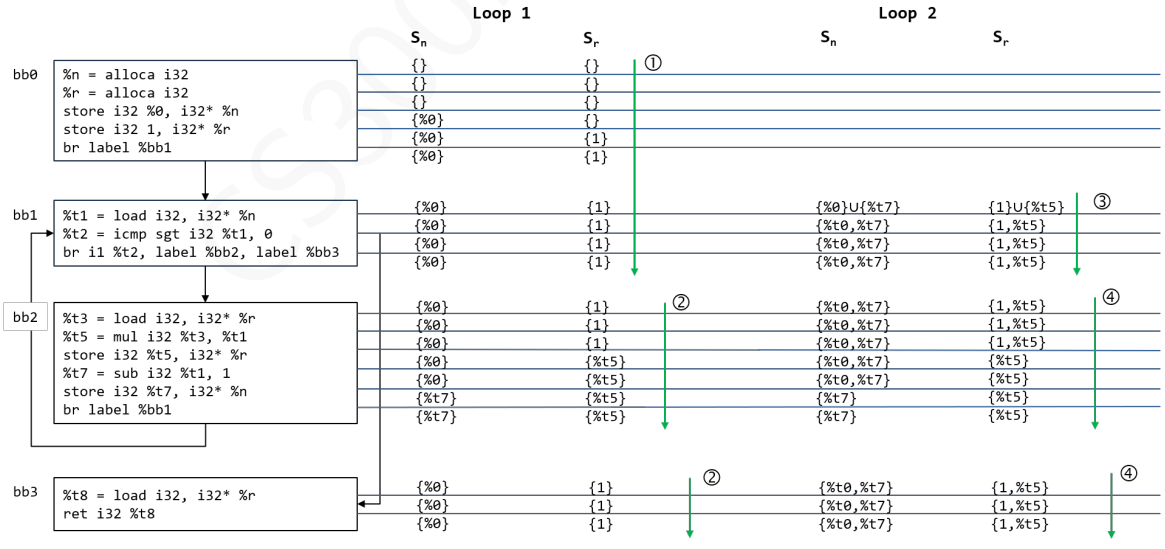


图 8.3: 数值流分析

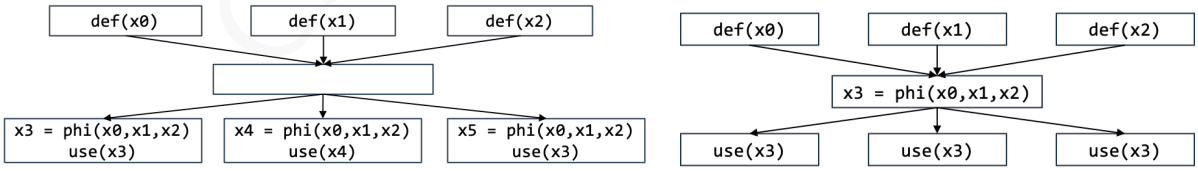
使用 phi 指令替换 store-load

确定了每个程序节点可能对应的变量数值定义后，只需在存在多个定义来源的程序点插入 phi 指令，即可使用虚拟寄存器替换原有的 `store/load` 操作。对于图 8.3 而言，变量 `n` 在代码块 `bb1` 中同时可能来源于 `bb0` 和 `bb2`，因此必须在 `bb1` 中插入 `phi(n)`。变量 `r` 在代码块 `bb1` 中也存在不同来源，但在 `bb1` 中并未被使用，因此其对应的 phi 指令既可以在 `bb1` 中插入，也可以延迟到 `bb2`, `bb3` 再插入。显然，在 `bb1` 中同时插入 `phi(n)` 和 `phi(r)` 更为理想。这种方式能够统一循环入口处的变量定义关系，使 IR 结构更加规整。最终得到的 SSA 形式如代码 8.2 所示。

```
define i32 @fac(i32 %0) {
bb0:
    br label %bb1
bb1:
    %n0 = phi i32 [%0 %bb0], [%t7:%bb2];
    %r0 = phi i32 [1 %bb0], [%t5:%bb2];
    %t2 = icmp sgt i32 %n0, 0
    br i1 %t2, label %bb2, label %bb3
bb2:
    %t5 = mul i32 %r0, %n0
    store i32 %t5, i32* %r
    %t7 = sub i32 %n0, 1
    store i32 %t7, i32* %n
    br label %bb1
bb3:
    ret i32 %r0
}
```

代码 8.2: 代码 8.1的静态单赋值形式

值得注意的是，虽然纯寄存器形式的 IR 能够将变量的 def-use 关系显式表示出来，但这并不意味着其 def-use 关系一定足够简洁。若 phi 指令插入位置不合理，仍可能导致 def-use 关系数量快速增长。为了获得更优的 phi 指令组织形式，应尽量在靠近控制流汇合起点的位置提前插入 phi 指令。以图 8.4a 为例，在直接使用寄存器表示后，其 def-use 关系数量达到 3×3，并且会随着控制流深度增加呈指数级增长。若将 phi 指令前移，如图 8.4b 所示，则 def-use 关系数量可降低为 3+3，从而有效避免 def-use 关系的指数爆炸问题。



(a) 非最优 phi 指令放置　　(b) 最优 phi 指令放置

图 8.4: Phi 指令放置位置与 def-use 关系的优化举例

上述基于冗余消除的 SSA 构造方法虽然可行，但在实际编译器中较少直接使用。其主要原因在于，该方法难以自然地得到较优的 phi 指令布局，尤其是在复杂控制流下，phi 指令的数量与位置往往不够理想，容易导致冗余的 def-use 关系和额外的优化开销。因此，当函数控制流较为复杂时，通常还需要进一步设计专门的 phi 优化机制，以获得更加紧凑和高效的 SSA 形式。

8.3 基于支配边界的 SSA 构造方法

对于 phi 指令放置位置的确定，实际编译器中更常采用基于支配边界（Dominance Frontier）的 SSA 构造方法。与前述“先进行冗余消除、再逐步引入 phi 指令”的思路不同，该方法首先根据控制流图的支配关系确定 phi 指令的插入位置，再通过数据流分析与变量重命名过程更新 IR 中的虚拟寄存器使用关系，从而直接构造出较为规范且高效的 SSA 形式。

8.3.1 支配边界

定义 1（支配）**.** 给定有向图 $G(V, E)$ 与起始节点 v_0，若从 v_0 到节点 v_j 的所有路径都必须经过节点 v_i，则称 v_i 支配（dominate）v_j，记作 $v_i \in Dom(v_j)$。若进一步满足 $v_i \neq v_j$，则称 v_i 严格支配 v_j，记作 $v_i \in IDom(v_j)$。

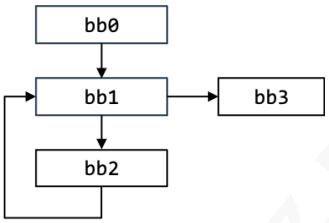


图 8.5: 控制流图举例

支配关系可以通过基于混沌迭代的数据流分析方法计算得到。具体而言，通过正向遍历控制流图，并维护从起始节点到当前节点路径上所有必经的代码块集合；当遇到具有多个前驱的控制流合并节点时，对各前驱路径的结果取交集。

以图 8.5 为例，各节点对应的支配节点集合如下：

$$Dom(bb_0) = \{bb_0\}$$

$$Dom(bb_1) = \{bb_0, bb_1\}$$

$$Dom(bb_2) = \{bb_0, bb_1, bb_2\}$$

$$Dom(bb_3) = \{bb_0, bb_1, bb_3\}$$

定义 2（支配边界）**.** 节点 v_i 的支配边界（Dominance Frontier）定义为所有满足以下条件的节点 v_j 的集合：

- v_i 支配 v_j 的某个前驱节点；
- v_i 不严格支配 v_j。

设节点 $v_j \in V$ 的前驱节点集合为 P_j，支配节点集合为 $Dom(v_j)$，则支配边界可通过集合关系直接计算得到。即对于任意 $v_p \in P_j$，以及任意：

$$v_i \in Dom(v_p) \backslash IDom(v_j)$$

都有：

$$v_j \in DF(v_i)$$

图 8.5 中各节点对应的支配边界如下：

$$DF(bb_0) = \emptyset$$

$$DF(bb_1) = \{bb_1\}$$

$$DF(bb_2) = \{bb_1\}$$

$$DF(bb_3) = \emptyset$$

在 SSA 构造中，若某节点对变量 x 进行了赋值，则需要在该节点的支配边界上插入 $\phi(x)$。以代码 8.1 为例，由于 bb_2 的支配边界为 bb_1，且 bb_2 中对变量 r 和 n 进行了赋值，因此应在 bb_1 中插入 $\phi(r)$ 与 $\phi(n)$。

8.3.2 更新 def-use 关系

在确定了各程序点需要插入的 phi 指令之后，需要进一步更新 IR 中的 def-use 关系，从而完成 SSA 形式的构造。该过程可以视为一种基于控制流图的值传播与重命名过程：沿控制流图进行遍历，为每个变量维护其当前对应的虚拟寄存器。

具体过程如下：

- 进入基本块时，若该块包含 phi 指令（位于控制流合并处），则根据各前驱路径上的变量取值，为 phi 指令补充对应的参数；
- 在基本块内部，当遇到新的定义（包括普通指令和 phi 指令）时，更新该变量当前对应的虚拟寄存器；在该过程中，每个程序点上每个变量至多对应一个虚拟寄存器，从而保证单赋值性质；
- 在使用变量时，直接引用当前记录的虚拟寄存器；

最终，通过上述过程可以完全消除原有的 `load/store` 指令，并建立起基于虚拟寄存器的显式 def-use 关系，从而得到规范的 SSA 形式。

参考文献

[1] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. “An efficient method of computing static single assignment form.” *In ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1989.

练习

1) 分析图 8.6 中各基本块的支配边界。

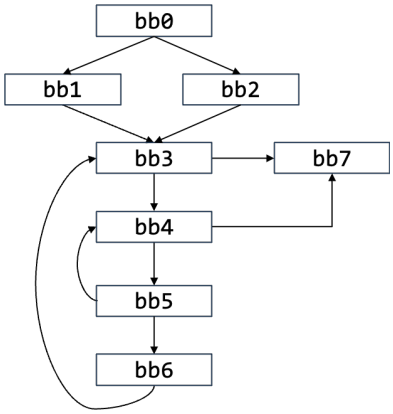


图 8.6: 控制流图

2) 代码 8.7 给出了 Eratosthenes 质数筛选算法的 IR。请按照以下步骤将其转换为 SSA 形式：

   1) 计算各基本块的支配边界；

   2) 在合适位置插入 phi 指令；

   3) 更新虚拟寄存器的 def-use 关系。

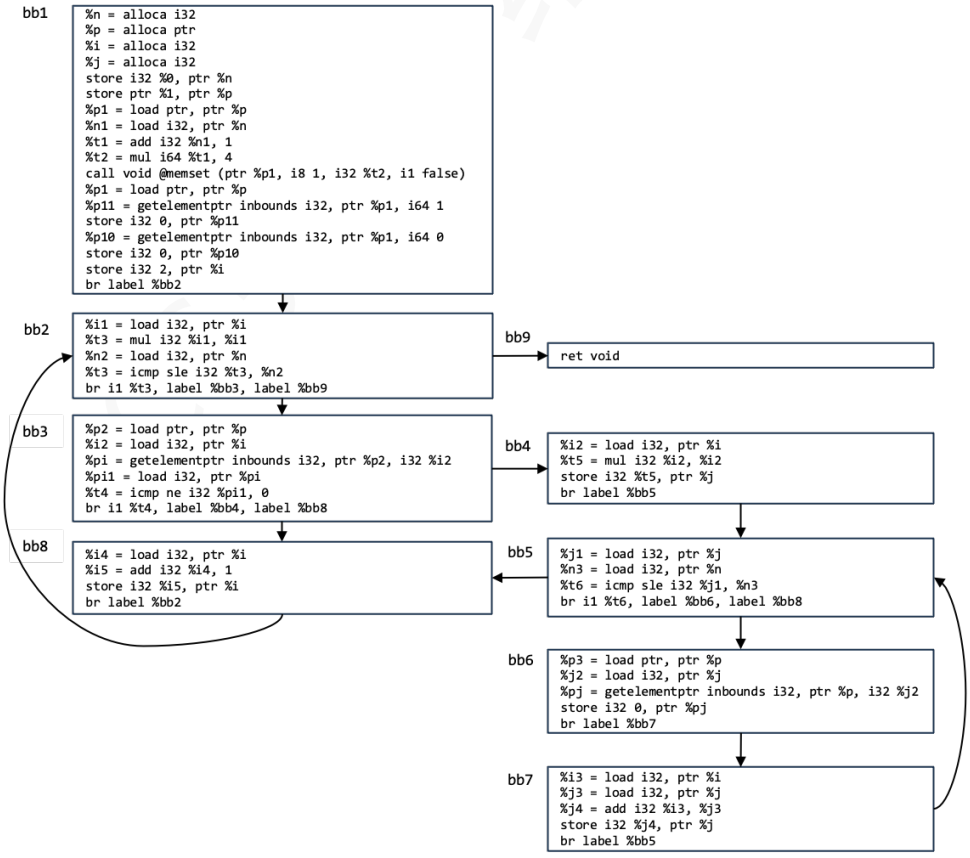


图 8.7: 质数筛选算法对应的 IR