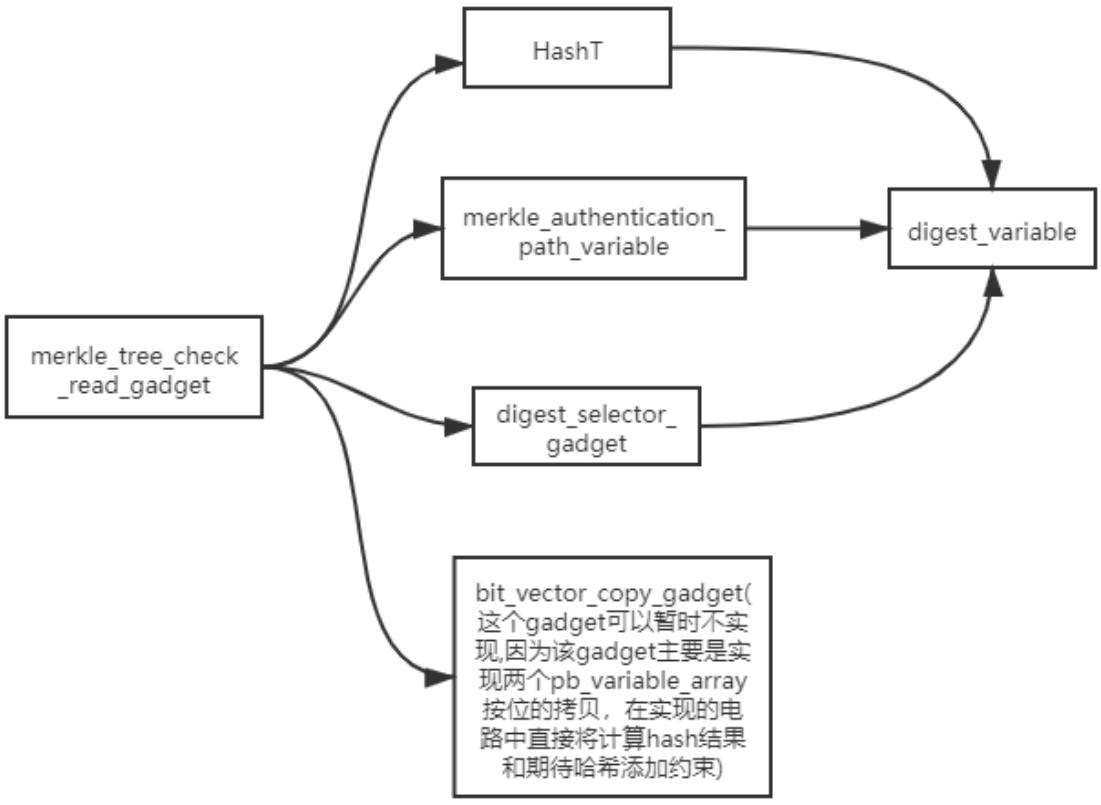
# merkletree Gadget



# merkletree Gadget依赖图




# merkle_authentication_path_variable_gadget

通过address的二进制位将path上的节点安放在left_digests和right_digests上。然后计算的中间结果由digest_selector_gadget安放在左右节点的路径上。

## 约束

分别对每层高度上的左右节点值求约束。

left_digest $lc * (1 - lc) = 0$ 一共digest_size位bit，tree_depth的深度。

right_digest $lc * (1 - lc) = 0$ 一共digest_size位bit，tree_depth的深度。

**一共有 $2 * digest\_size * tree\_depth$ 个约束。**

# merkle_tree_check_read_gadget

验证merkletree的一条路径以及其叶子节点求出的root值是否与期望的根节点的哈希值相同。

## 约束

1. tree_depth个hash约束。hash的具体gadget是通过实例化模板形成。**这一部分hash的约束暂时不是很明白**
2. tree_depth个digest_selector_gadget的约束，该约束主要是根据address_bits的bit位来将internal_output节点安放在求解根节点路径上面的左右。

   $is\_right * (right.bits[i] - left.bits[i]) = (input.bits[i] - left.bits[i])$ digest_size次该条约束，一共循环tree_depth 。 约束数量: $digest\_size * tree\_depth$
3. bit_vector_copy_gadget 将源variable数组按位拷贝给目的variable数组。约束为：
   $1 * (source[i] - target[i]) = 0$ 一共有`source.size()`个约束。

   对于source_bits和以及target_bits的每一位都要为0或者1，因此对于每个bit上需要添加约束。

# hash gadget 约束

这一部分的约束直接调用bellman sha256 gadget接口，暂时没有管sha256 gadget部分的约束分析。

# withness(private input和public input) 变量

为了方便理解merkletree所有约束的汇总，在写merkletree电路的时候需要需要明确withness的输入变量有哪些。如何确定这些变量?

我的理解是通过分析libsnark中的merkle_tree_check_read_gadget的构造函数的输入值，以及generate_r1cs_withness的输入变量有哪些，根据两者的输入变量确定merkletree的withness。因为bellman里面constraint和withness步骤是混在一起写，因此需要在初始化电路的时候传入withness的值。

分析的merkletree的withness输入主要有

```
    tree_depth: u64,   // 树的深度(实际树的高度为tree_depth+1)
    digest_size: u64,  // 哈希摘要值
    address_bits: Vec<Option<E::Fr>>, // address_bits值，主要用于将path以及
internal_output以及leaf的哈希值赋值给left_digests和right_digests中
    leaf_digest: Vec<Option<E::Fr>>, // 需要验证的叶子节点的哈希值
    root_digest: Vec<Option<E::Fr>>, // 期待的根哈希值
    path: Vec<Vec<Option<E::Fr>>>,   // 对于叶子节点的验证路径
```

# merkletree约束总结

为了方便理解bellman中的电路，因此这里将merkletree所依赖gadget的所有约束进行汇总。

1. `left_digests: Vec<Vec<Option<E::Fr>>>` 和 `right_digests: Vec<Vec<Option<E::Fr>>>` 计算根节点左右哈希路径哈希值的约束。哈希上的每一位都要满足约束$lc * (1 - lc) = 0$。
2. tree_depth个digest_selector_gadget的约束，该约束主要是根据address_bits的bit位来将internal_output节点安放在求解根节点路径上面的左右。

   $is\_right * (right.bits[i] - left.bits[i]) = (input.bits[i] - left.bits[i])$ digest_size次该条约束，一共循环tree_depth 。 约束数量: $digest\_size * tree\_depth$
3. 计算的哈希结果和期望的哈希结果直接的约束。$root\_digest[i] * 1 = computed\_root[i]$ 一共 `digest_size` 个约束。

# 测试部分的逻辑

对于测试部分主要是如何构造上面的withness的变量值。

1. 随机生成一个pre_hash(leaf_digest)的哈希值。
2. 随机生成一个computed_is_right bool值，以及一个other(path[i])的哈希值，根据生成的computed_is_right的值来判断pre_hash哈希在右节点(true)或者是other哈希在右节点(false)，并且将computed_is_right的值加入到address_bits的数值中。该过程循环tree_depth次。
3. 最后将pre_hash的值赋值给root_digest。

构造上面的所有withness值结束。

# 注意的问题

1. bellman里面的constraint和withness的过程是写在一起的，但是在生成pk个vk的过程中需要用到构造的电路，因此在生成pk和vk的过程中需要初始化一个空值的电路，用该电路来生成pk和vk。bellman里面的生成pk和vk过程中空值电路(使用Option来包裹具体的值，空值为None)的处理要使用 `ok_or(SynthesisError::AssignmentMissing)` 对传进来的空值进行处理。
2. 电路的初始化生成pk和vk过程只是依赖于电路中构造的约束，并不依赖于withness的传值过程。
3. rust中Vector的append操作会导致被append的Vector被回执为空。
4. 注意匿名函数的返回值应该不加上分号。

# 完整代码加注释分析

见merkletree.rs文件