Value Numbering

L. Taylor Simpson
lts@cs.rice.edu

September 6, 1996

[0]This work has been supported by ARPA, through ONR grant N00014-91-J-1989.

Contents

List of Figures

Chapter 1

Introduction

Value numbering is a program analysis method that allows redundant computations to be removed. This document presents an implementation of several types of value numbering within the Massively Scalar Compiler Project. Although the name was originally applied to a method for improving single basic blocks, it is now used to describe a collection of optimizations that vary in power and scope. The primary objective of value numbering is to assign an identifying number (a *value number*) to each expression in a particular way. Two expressions must have the same number if the compiler can prove they are equal for all possible program inputs. These numbers can then be used to find redundant computations and remove them. There are two other objectives accomplished by certain forms of value numbering:

1. To recognize certain algebraic identities, like $i = i + 0$ and $j = j \times 1$, and to use them to simplify the code and to expand the set of expressions known to be equal.
2. To evaluate expressions whose operands are constants and to propagate their values through the code.

There are several ways to accomplish each of these goals, and the methods can be applied across different scopes. The reader should be aware of the partitioning methods, but they will not be implemented here [2, 5, 16]. This document implements a collection of techniques built around hashing. Our approach hashes an operator and the value numbers of its operands to produce a value number for the resulting expression. Hashing provides an efficient representation of the expressions known at any point during the analysis. The efficiency relies on the constant expected-time behavior of hashing. This approach can easily be extended to propagate constants and simplify algebraic identities.

We perform value numbering over the static single assignment (SSA) form of the routine [10, 6]. The basic idea used in constructing SSA form is to give unique names to the targets of all assignments in the routine, and then to overwrite uses of the assignments with the new names. Special assignments, called ϕ-functions, are inserted to select the appropriate definition where more that one definition site (each with a unique SSA name) reaches a point in the routine. Each ϕ-function defines a new name for the original item as a function of all of the SSA names which are current at the end of the join point's predecessors. Any uses of the original name after the ϕ-function are replaced by the ϕ-function's result. The ϕ-function selects the value of the input that corresponds to the block from which control is transferred and assigns this value to the result. The critical property of SSA for this work is the naming discipline that it imposes on the code. Each SSA name is assigned a value by exactly one operation in a routine; therefore, no name is ever reassigned, and no expression ever becomes inaccessible.

The primary step in hash-based value numbering is applied to each operation in the region being optimized. For an operation $x \leftarrow y \ op \ z$, we look up both y and z to obtain their value numbers, vn_y and vn_z. Next, we look up $\langle vn_y \ op \ vn_z \rangle$ in the hash table to obtain a value number for the whole expression. Two possibilities exist:

1. $\langle vn_y \ op \ vn_z \rangle$ is not in the table. The compiler enters it in the table with a new value number.
2. $\langle vn_y \ op \ vn_z \rangle$ is already in the table. The compiler has already seen a computation of $vn_y \ op \ vn_z$ and given it a value number. This recomputation can be replaced by a reference to the earlier value.

In either case, the compiler assigns the value number for $\langle vn_y \;\; op \;\; vn_z\rangle$ to x.

The difference between the various hash-based methods lies in the framework that surrounds the primary step. The framework determines both the set of instructions processed and the order in which they are processed. For example, the classic local algorithm applies the primary step in sequential order to each instruction in a single basic block [9]. That algorithm has been extended to operate over larger scopes by carefully manipulating the hash table and ordering the applications of the primary step [5, 15].

The primary step can be applied across the following scopes:

Single basic blocks We apply the primary step to each instruction (from top to bottom) in a single basic block.

Extended basic blocks An extended basic block is a sequence of blocks B_1, B_2, ..., B_n where B_i is the only predecessor of B_{i+1}, for $1 \leq i < n$, and B_1 does not have a unique predecessor. To apply value numbering to a single extended basic block, we apply the single block algorithm to each block in the sequence, in order, and use the results from B_{i-1} to initialize the tables for B_i. This works because each B_i has a single predecessor, $1 < i \leq n$.

Dominator-based value numbering technique (DVNT) We can apply the primary step to each instruction in a preorder walk of the dominator tree. A value can enter a block B in one of two ways: either it is defined by a ϕ-function at the start of B, or it flows through B's parent in the dominator tree. Therefore, we process each block by initializing the hash table with the information resulting from value numbering its parent. To accomplish this, we use a scoped hash table.

Value numbering with a unified hash table We walk the dominator tree using a unified table (*i.e.,* a single hash table for the entire routine). Using a unified hash-table provides the uniform naming discipline required for AVAIL-based removal (see Section 4) or partial redundancy elimination [14]. However, it has one important algorithmic consequence. Replacements cannot be performed on-line because the table no longer reflects availability. In previous algorithms, the existence of an expression in the hash table meant that the expression was computed earlier in the program. No such relationship between expressions exists under this approach. Thus, we cannot immediately remove expressions found in the table. We rely on either AVAIL-based removal or partial redundancy elimination to remove the redundancies.

SCC-based value numbering This is a global algorithm centered around the strongly connected components (SCCs) of the SSA graph. It differs from the other techniques in two ways. First, it uses Tarjan's depth-first search algorithm [17] (operating over the SSA graph) to order the application of the primary step for value numbering. Second, it uses a more complex form of the primary step that allows it to make optimistic assumptions about values that propagate along back edges and to prove or disprove those assumptions later.

1.1 Optimism vs. Pessimism

Program analysis techniques are often described as optimistic or pessimistic. Wegman and Zadeck introduced the notion of "optimism" to describe a property of their constant propagation algorithm [18, 19]. Their algorithm "optimistically" assumes that values are constant and then proves or disproves the assumption. Click formalized optimism and pessimism in a particularly clear way [8]. In his model, an optimistic analysis constructs a set of facts top-down. It begins with a large set of facts as its initial condition and systematically applies a set of monotonic rules to determine which facts are supported by the rules. The process must reach a fixed point; when it does, all remaining facts are true and no larger set of true facts exists for the rules. In contrast, a pessimistic analysis begins with the empty set and repeatedly applies a set of monotonic rules to build up to a fixed point. Given a set of rules, an optimistic analysis will always produce a set at least as large as the set produced by a pessimistic analysis.

Because its intermediate states contain facts that may yet be disproven, an optimistic analysis must run to completion. In contrast, the intermediate results from a pessimistic analysis contain only facts already proven true; thus, the analysis can be stopped prematurely. As a consequence, optimistic techniques cannot, in general, update the code as they analyze it. Pessimistic techniques can sometimes perform online updates.

SCC-based value numbering is optimistic, and all other forms ov value numbering implemented here are pessimistic.

⟨Macros 3⟩ ≡

```
#define VALID                        0
#define OPTIMISTIC                   1
```
◇

Macro defined by scraps 3, 7ab, 12b, 36d.
Macro referenced in scrap 4a.

1.2 Usage

valnum can be invoked from the command line as follows:

`valnum` [`-dvpamfstc`] [*filename*]

valnum reads `ILOC` input from *filename* (or from `stdin` if no *filename* is specified) and prints its output to `stdout`. The command line flags may be concatenated as in `-vvvvfsmp`.

The following options are available:

`-d` Increases the debug level. The default level is zero – meaning no debugging output. Each additional `d` in the flags list increases the amount of debugging information printed. The following levels are supported:

1. Indicate when the major phases of the program begin
2. Indicate when the minor phases of the program begin
3. Show the routine in SSA form

All debugging output is to `stderr`.

`-v` Increase the value numbering level. The default level is zero – meaning value number single basic blocks. Each additional `v` in the flags list increases the level. The following levels are supported:

1. Value number single basic blocks
2. Value number extended basic blocks
3. Value number over the dominator tree
4. Value number using the unified hash table
5. Perform SCC-based value numbering

`-p` Prepare the output for partial redundancy elimination.

`-a` Remove operations based on AVAIL. This option requires either the unified hash table or SCC-based value numbering.

`-m` Trace values through memory.

`-f` Perform constant folding.

`-s` Perform algebraic simplification.

`-t` Print timing information for the major phases.

`-c` Perform value-driven code motion. This option requires either the unified hash table or SCC-based value numbering.

1.3 Overall Structure

This is a simple boilerplate that specifies how the major components fo the C program will fit together. We need include files for the shared library code, for the routines to handle SSA form, and for the mathematical functions performed during constant folding.

```
"valnum.c" 4a ≡
      #include <Shared.h>
      #include <SSA.h>
      #include <limits.h>
      #include <math.h>

      ⟨Type Declarations 34a, ... ⟩
      ⟨Macros 3, ... ⟩
      ⟨Global Variables 4b, ... ⟩
      ⟨Prototypes 8d, ... ⟩
      ⟨Functions 9a, ... ⟩
      ⟨The main routine 5⟩
      ◇
```

1.4 The Main Routine

```
⟨Global Variables 4b⟩ ≡
      static Arena valnum_arena;
      ◇
```

Macro defined by scraps 4b, 6b, 7d, 12a, 13b, 17b, 33a, 34b, 44a, 61c.
Macro referenced in scrap 4a.

The **main** routine shows the major steps of the program. The steps involved in reading the **ILOC** file, converting to and from SSA form, and printing the optimized program will be explained in this section. The steps that are directly involved in value numbering will be given detailed explanations in the remaining chapters of this document. The printing of timing information is controlled by the **time_print** flag which will be turned on during parsing of the command line if the user specified the **-t** option. The variable **time** keeps track of the time spent along the way.

⟨The `main` routine 5⟩ ≡
```
Void main(Unsigned_Int argc, Char **argv)
{
    Timer time = Time_Start();
    Char time_message[128];

    ⟨Parse the command line 6a⟩
    ⟨Read the ILOC file 8b⟩
    ⟨Convert to SSA form 8c⟩

    valnum_arena = Arena_Create();
    ⟨Perform value numbering 11⟩
    ⟨Optionally report the number of hash table entries 61d⟩

    ⟨Optionally invent new value numbers 43⟩
    ⟨Renumber the φ-nodes and operations in the routine 46a⟩

    ⟨Optionally perform value-driven code motion 9c⟩
    Arena_Destroy(valnum_arena);

    ⟨Optionally perform AVAIL-based removal 51a⟩

    ⟨Convert out of SSA form 10a⟩
    ⟨Write the ILOC file 10b⟩
    free(time);
    if (time_print) Arena_DumpStats();

    exit(0);
}
◇
```
Macro referenced in scrap 4a.

Option	Variable
-d	debug
-v	valnum_level
-p	prepare_for_partial
-a	do_avail
-m	trace_memory
-f	fold_constants
-s	simplify
-t	time_print (defined in the shared library)
-c	do_vdcm

Table 1.1: Global variables to support command line options

We parse the command line by looping over the entries in argv. The variable i is the current index into argv, and c_ptr is a pointer into argv[i]. We assume that any input beginning with the character '-' is a list of flags, and all others are file names. If more than one file name is entered, we assume the user entered something wrong. In this case, we print out the usage message and halt.

⟨Parse the command line 6a⟩ ≡
```
{
    Unsigned_Int i;
    for (i = 1; i < argc; i++)
    {
        Char *c_ptr = argv[i];

        if (*c_ptr++ == '-')
            ⟨Parse a list of flags 7c⟩
        else if (!file_name)
            file_name = argv[i];
        else
            ⟨Print the usage message and halt 8a⟩
    }
}
◇
```
Macro referenced in scrap 5.

We'll need some global variables to handle the command-line options. Table 1.1 shows the variable that corresponds to each option.

⟨Global Variables 6b⟩ ≡
```
Unsigned_Int debug = 0;
static Unsigned_Int valnum_level = VALNUM_BASIC_BLOCKS;
static Boolean prepare_for_partial = FALSE;
static Boolean do_avail = FALSE;
static Boolean trace_memory = FALSE;
static Boolean fold_constants = FALSE;
static Boolean simplify = FALSE;
static Boolean do_vdcm = FALSE;
◇
```
Macro defined by scraps 4b, 6b, 7d, 12a, 13b, 17b, 33a, 34b, 44a, 61c.
Macro referenced in scrap 4a.

The debug variable has three valid values.

⟨Macros 7a⟩ ≡
```
#define MAJOR_PHASES                    1
#define MINOR_PHASES                    2
#define SHOW_SSA                        3
```
◇

Macro defined by scraps 3, 7ab, 12b, 36d.
Macro referenced in scrap 4a.

The valnum_level variable has five valid values.

WARNING: The code in this document assumes a specific order for these values. Do not rearrange them!

⟨Macros 7b⟩ ≡
```
#define VALNUM_BASIC_BLOCKS             1
#define VALNUM_EXTENDED_BASIC_BLOCKS    2
#define VALNUM_DOMINATOR_TREE           3
#define VALNUM_UNIFIED                  4
#define VALNUM_SCC                      5
```
◇

Macro defined by scraps 3, 7ab, 12b, 36d.
Macro referenced in scrap 4a.

We examine a list of flags one character at a time. For each valid flag, there is a corresponding global variable. If we encounter a character which is not in the set of recognizable flags, we print out the usage message and halt.

⟨Parse a list of flags 7c⟩ ≡
```
while (*c_ptr)
    switch (*c_ptr++)
    {
        case 'd':   debug++;                        break;
        case 'v':   valnum_level++;                 break;
        case 'p':   prepare_for_partial = TRUE;     break;
        case 'a':   do_avail = TRUE;                break;
        case 'm':   trace_memory = TRUE;            break;
        case 'f':   fold_constants = TRUE;          break;
        case 's':   simplify = TRUE;                break;
        case 't':   time_print = TRUE;              break;
        case 'c':   do_vdcm = TRUE;                 break;
        default:
            ⟨Print the usage message and halt 8a⟩
    }
```
◇

Macro referenced in scrap 6a.

The string file_name is initially set to NULL, indicating that we want to use the default source file (stdin). When we find a filename, we point file_name at it. If we find more than one, we complain and halt.

⟨Global Variables 7d⟩ ≡
```
static Char *file_name = NULL;
```
◇

Macro defined by scraps 4b, 6b, 7d, 12a, 13b, 17b, 33a, 34b, 44a, 61c.
Macro referenced in scrap 4a.

If we encounter an error parsing the command line, we print the usage string and halt the program. We use the ABORT macro, which is set up to dump core if we are running in debug mode and exit with an error code otherwise.

⟨Print the usage message and halt 8a⟩ ≡
```
{
    fprintf(stderr, "Usage: %s [-dvpamfstc] [filename]\n", argv[0]);
    ABORT;
}
◇
```
Macro referenced in scraps 6a, 7c.

Once we have parsed the command line, we are ready to read in the ILOC file and convert it to SSA form. The Block_Init routine provided by the shared library will read the ILOC file and build a control-flow graph (CFG) [4].

⟨Read the ILOC file 8b⟩ ≡
```
Block_Init(file_name);
sprintf(time_message, "%s: Parsing required %%1.2fseconds\n",
        file_name ? file_name : "stdin");
Time_Dump(time, time_message);
◇
```
Macro referenced in scrap 5.

The SSA_Build routine converts the routine to SSA form. This conversion renames all the items in the routine so that there is a unique definition point for each item in the routine. If we are performing SCC-based value numbering, we also construct a table containing the definition point for each name.

⟨Convert to SSA form 8c⟩ ≡
```
if (debug >= MAJOR_PHASES)
    fprintf(stderr, "Convert the routine to SSA form\n");

if (valnum_level == VALNUM_SCC)
    SSA_Build(SSA_PRUNED | SSA_FOLD_COPIES | SSA_BUILD_USE_DEF_CHAINS);
else
    SSA_Build(SSA_PRUNED | SSA_FOLD_COPIES);
sprintf(time_message, "%s: Building SSA form required %%1.2f seconds\n",
        file_name ? file_name : "stdin");
Time_Dump(time, time_message);
if (debug >= SHOW_SSA)
{
    fprintf(stderr, "Before value numbering\n");
    Block_Dump_All(phi_node_printer, 1);
}
◇
```
Macro referenced in scrap 5.

The phi_node_printer routine is invoked by the Block_Dump_All routine. It will print out the ϕ-nodes for the block. Notice that the printing is different for registers and tags.

⟨Prototypes 8d⟩ ≡
```
static Void phi_node_printer(Block *block);
◇
```
Macro defined by scraps 8d, 9b, 13d, 14b, 16b, 17c, 19b, 24a, 29b, 34e, 36b, 38, 64c, 65b, 66b, 67b, 69a, 70b, 72bd, 73b, 75b, 76c, 84a.
Macro referenced in scrap 4a.

⟨Functions 9a⟩ ≡

```
static Void phi_node_printer(Block *block)
{
    Unsigned_Int pred_count = block->pred_count;
    Phi_Node *phi_node;
    Unsigned_Int i;

    Block_ForAllPhiNodes(phi_node, block)
        if (phi_node->old_name < register_count)
        {
            fprintf(stderr, "Phi node for r%d: [",
                    phi_node->old_name);
            for (i = 0; i < pred_count; i++)
                fprintf(stderr, " r%d", phi_node->parms[i]);
            fprintf(stderr, " ] => r%d\n", phi_node->new_name);
        }
        else
        {
            fprintf(stderr, "Phi node for %s: [",
                    Expr_Get_String(Tag_Unmap(phi_node->old_name)));
            for (i = 0; i < pred_count; i++)
                fprintf(stderr, " %d", phi_node->parms[i]);
            fprintf(stderr, " ] => %d\n", phi_node->new_name);
        }
}
◇
```

Macro defined by scraps 9a, 14a, 15, 16c, 18a, 19c, 24b, 29c, 35a, 36c, 39a, 65a, 66a, 67a, 68a, 69b, 71, 72c, 73ac, 75c, 77a, 84b.
Macro referenced in scrap 4a.

The code to perform value-driven code motion is contained in a separate module. We simply need to declare the function and invoke it if requested. It must be invoked before the routine is converted out of SSA form. The value of **next_name** will indicate the size of the bit vectors. The actual value of **next_name** will be computed when we renumber the registers based on value numbers.

⟨Prototypes 9b⟩ ≡

```
extern Void vdcm(Unsigned_Int2 *, Unsigned_Int);
◇
```

Macro defined by scraps 8d, 9b, 13d, 14b, 16b, 17c, 19b, 24a, 29b, 34e, 36b, 38, 64c, 65b, 66b, 67b, 69a, 70b, 72bd, 73b, 75b, 76c, 84a.
Macro referenced in scrap 4a.

⟨Optionally perform value-driven code motion 9c⟩ ≡

```
if (do_vdcm)
{
    if (debug >= MAJOR_PHASES)
        fprintf(stderr, "Perform value-driven code motion\n");
    vdcm(value_num, next_name);
    sprintf(time_message, "%s: Value-driven code motion required %%1.2f seconds\n",
            file_name ? file_name : "stdin");
    Time_Dump(time, time_message);
}
◇
```

Macro referenced in scrap 5.

We convert the routine out of SSA form by calling `SSA_Replace_Phi_Nodes` and `SSA_Restore`.

⟨Convert out of SSA form 10a⟩ ≡

```
if (debug >= MAJOR_PHASES)
    fprintf(stderr, "Convert the routine out of SSA form\n");
if (debug >= MINOR_PHASES)
    fprintf(stderr, "    Replace phi-nodes\n");
SSA_Replace_Phi_Nodes();
if (debug >= MINOR_PHASES)
    fprintf(stderr, "    Restore the numbering\n");
SSA_Restore();
sprintf(time_message, "%s: Converting out of SSA form required %%1.2f seconds\n",
        file_name ? file_name : "stdin");
Time_Dump(time, time_message);
if (debug >= SHOW_SSA)
{
    fprintf(stderr, "After converting out of SSA form\n");
    Block_Dump_All(phi_node_printer, 1);
}
◇
```

Macro referenced in scrap 5.

After all is said and done, we write the results out. We need to set the flag `print_all_operations` to `FALSE` so that operations marked as non-critical are not printed. We use this convention to simplify deletion of redundant operations during the actual value numbering. We also reset the value of the global variable `register_count` so that the output routines will produce the correct code. Finally, we simply call a shared routine to dump the entire control-flow graph to `stdout`.

⟨Write the ILOC file 10b⟩ ≡

```
print_all_operations = FALSE;
if (next_name) register_count = next_name;
Block_Put_All(stdout);
sprintf(time_message, "%s: Printing required %%1.2f seconds\n",
        file_name ? file_name : "stdin");
Time_Dump(time, time_message);
◇
```

Macro referenced in scrap 5.

Chapter 2

Value Numbering

This chapter describes the implementation of several types of value numbering supported by this program. The primary step is the same for each type of value numbering. The differences lie in the order in which the operations and ϕ-nodes are processed and the manipulation of the hash tables. This scrap is responsible for initializing the data structures and selecting the proper type of value numbering to apply.

⟨Perform value numbering 11⟩ ≡

```
if (debug >= MAJOR_PHASES)
    switch (valnum_level)
    {
        case VALNUM_BASIC_BLOCKS:
            fprintf(stderr, "Perform value numbering over single basic blocks\n");
            break;
        case VALNUM_EXTENDED_BASIC_BLOCKS:
            fprintf(stderr, "Perform value numbering over extended basic blocks\n");
            break;
        case VALNUM_DOMINATOR_TREE:
            fprintf(stderr, "Perform value numbering over the dominator tree\n");
            break;
        case VALNUM_UNIFIED:
            fprintf(stderr, "Perform value numbering using the unified table\n");
            break;
        case VALNUM_SCC:
            fprintf(stderr, "Perform SCC-based value numbering\n");
            break;
        default:
            fprintf(stderr, "ERROR: Unknown value numbering level: %d\n",
                    valnum_level);
            ABORT;
    }

⟨Initialize the hash tables 62a, ... ⟩
⟨Initialize the data structures for value numbering 13a, ... ⟩
if (valnum_level <= VALNUM_UNIFIED)
    ⟨Perform dominator-based value numbering 29a⟩
else
    ⟨Perform SCC-based value numbering 33b⟩
sprintf(time_message, "%s: Value numbering required %%1.2f seconds\n",
        file_name ? file_name : "stdin");
Time_Dump(time, time_message);
◇
```

Macro referenced in scrap 5.

Valnum(*oper*, *table*)
  Assume *oper* is of the form "$x \leftarrow y$ *op* z"
  expr $\leftarrow \langle$*value_num*$[y]$, *op*, *value_num*$[z]\rangle$
  Try to simplify *expr* (if requested)
  temp $\leftarrow$ *lookup*(*expr*, *table*, x)
  if *value_num*$[x] \neq$ *temp*
    changed $\leftarrow$ `TRUE`
    value_num$[x] \leftarrow$ *temp*

Figure 2.1: Basic value numbering operation

2.1 Basic Value Numbering Functions

Figure 2.1 shows pseudo-code for value numbering an instruction. The *value_num* array maps SSA names to value numbers, and *table* contains the set of known expressions. The *lookup* function searches *table* for the expression. If the expression is found, *lookup* will return its value number. Otherwise, *lookup* will add the expression to the table with value number x.

Before we get to the actual code, we must set up the supporting data structures. The **value_num** array maps SSA names to their value number. To simplify the implementation of the algorithm, the SSA name of the first occurrence of an expression becomes the expression's value number. In our discussion, we will surround an SSA name that represents a value number with angle brackets (*e.g.,* $\langle x_0 \rangle$).

The **changed** variable is set to **TRUE** whenever an entry in the **value_num** array is changed.

⟨Global Variables 12a⟩ ≡

```
static Unsigned_Int2 *value_num = NULL;
static Boolean changed = FALSE;
◇
```

Macro defined by scraps 4b, 6b, 7d, 12a, 13b, 17b, 33a, 34b, 44a, 61c.
Macro referenced in scrap 4a.

We use ⊤ (**TOP**) as the initial value number for each name; it indicates that this value has yet to be examined, and it is considered equal to everything. We define a value for ⊤ that is guaranteed to be different from any value number we assign. Since **SSA_def_count** is larger than any SSA name, it will be larger than any value number.

⟨Macros 12b⟩ ≡

```
#define TOP                     SSA_def_count
◇
```

Macro defined by scraps 3, 7ab, 12b, 36d.
Macro referenced in scrap 4a.

The first `tag_count + 1` definitions (SSA names `0`...`tag_count`) are dummy definitions to catch uninitialized values. We will simply set their value number to zero (which indicates an undefined value). The other SSA names have their initial value number set to `TOP`.

⟨Initialize the data structures for value numbering 13a⟩ ≡

```
{
    Unsigned_Int i;

    value_num = Arena_GetMem(valnum_arena, SSA_def_count*sizeof(Unsigned_Int2));
    for (i = 0; i <= tag_count; i++)
        value_num[i] = 0;
    for (i = tag_count + 1; i < SSA_def_count; i++)
        value_num[i] = TOP;
}
```

◇

Macro defined by scraps 13ac.
Macro referenced in scrap 11.

If the user requests folding of constants or algebraic simplification, we must be able to determine which values are constants and what their corresponding values are. We will use a bit vector, `constants`, to indicate which value numbers are constants and an array of `Expr`s, called `values`, to indicate their values. Note that only members of `constants` will have valid entries in `values`. We allocate `constants` with `SSA_def_count+1` entries so we can test that $\top$ is a member.

⟨Global Variables 13b⟩ ≡

```
static VectorSet constants;
static Expr *values;
```

◇

Macro defined by scraps 4b, 6b, 7d, 12a, 13b, 17b, 33a, 34b, 44a, 61c.
Macro referenced in scrap 4a.

⟨Initialize the data structures for value numbering 13c⟩ ≡

```
if (fold_constants || simplify)
{
    constants = VectorSet_Create(valnum_arena, SSA_def_count+1);
    values = Arena_GetMem(valnum_arena, SSA_def_count*sizeof(Expr));
}
```

◇

Macro defined by scraps 13ac.
Macro referenced in scrap 11.

The `valnum_phi_node` routine is the primary value numbering step for ϕ-nodes. It first checks if all the parameters of the ϕ-node have the same value number, v. If so, the value number of the ϕ-node is set to v. Otherwise, we look up the ϕ-node in the hash table to see if it is redundant.

⟨Prototypes 13d⟩ ≡

```
static Void valnum_phi_node(Phi_Node *phi_node, Block *block,
                            Unsigned_Int which_table);
```

◇

Macro defined by scraps 8d, 9b, 13d, 14b, 16b, 17c, 19b, 24a, 29b, 34e, 36b, 38, 64c, 65b, 66b, 67b, 69a, 70b, 72bd, 73b, 75b, 76c, 84a.
Macro referenced in scrap 4a.

⟨Functions 14a⟩ ≡

```
static Void valnum_phi_node(Phi_Node *phi_node, Block *block,
                            Unsigned_Int which_table)
{
    Unsigned_Int SSA_name = phi_node->new_name;
    Int result = equal_parms(phi_node);
    if (result > 0)
    {
        if (value_num[SSA_name] != result)
        {
            changed = TRUE;
            value_num[SSA_name] = result;
        }
    }
    else
        ⟨Look for phi_node in table 16a⟩
}
◇
```

Macro defined by scraps 9a, 14a, 15, 16c, 18a, 19c, 24b, 29c, 35a, 36c, 39a, 65a, 66a, 67a, 68a, 69b, 71, 72c, 73ac, 75c, 77a, 84b.
Macro referenced in scrap 4a.

The **equal_parms** function will return the value of the significant parameters of **phi_node** if they are all equal and -1 otherwise. Which parameters are significant depends on the type of value numbering. For dominator-based value numbering, we ignore any uninitialized values (value number 0). For SCC-based value numbering, we ignore any unvisited values (value number $\top$).

⟨Prototypes 14b⟩ ≡

```
static Int equal_parms(Phi_Node *phi_node); ◇
```

Macro defined by scraps 8d, 9b, 13d, 14b, 16b, 17c, 19b, 24a, 29b, 34e, 36b, 38, 64c, 65b, 66b, 67b, 69a, 70b, 72bd, 73b, 75b, 76c, 84a.
Macro referenced in scrap 4a.

⟨Functions 15⟩ ≡

```
static Int equal_parms(Phi_Node *phi_node)
{
    Boolean need_val = TRUE;
    Unsigned_Int last_val;
    Unsigned_Int2 *reg_ptr;

    Phi_Node_ForAllParms(reg_ptr, phi_node)
    {
        Unsigned_Int value = value_num[*reg_ptr];
        if ((valnum_level <= VALNUM_UNIFIED && value) ||
            (valnum_level >= VALNUM_SCC && value != TOP))
            if (need_val)
            {
                last_val = value;
                need_val = FALSE;
            }
            else if (last_val != value)
                return -1;
    }

    if (need_val)
    {
        fprintf(stderr, "Internal error: phi-node with no parameters\n");
        ABORT;
    }
    return last_val;
}
```

◇

Macro defined by scraps 9a, 14a, 15, 16c, 18a, 19c, 24b, 29c, 35a, 36c, 39a, 65a, 66a, 67a, 68a, 69b, 71, 72c, 73ac, 75c, 77a, 84b.
Macro referenced in scrap 4a.

If `phi_node` cannot be simplified, we determine if it is redundant. We look for it in the hash table by calling `phi_search`. If the return value is non-negative, it represents the value number of the ϕ-node, and we assign this value number to `SSA_name`. Otherwise, we use `phi_node`'s SSA name as its value number and add an entry to the hash table by calling `phi_add`.

⟨Look for phi_node in table 16a⟩ ≡

```
{
    Int result = phi_search(which_table, phi_node, block);
    if (result >= 0)
    {
        if (value_num[SSA_name] != result)
        {
            changed = TRUE;
            value_num[SSA_name] = result;
        }
    }
    else
    {
        if (value_num[SSA_name] != SSA_name)
        {
            changed = TRUE;
            value_num[SSA_name] = SSA_name;
        }
        phi_add(which_table, phi_node, block);
    }
}
◇
```

Macro referenced in scrap 14a.

The `valnum_oper` routine performs the primary value numbering step for operations. The action performed depends on the type of operation being processed. This is determined by the `details` flags for the opcode.

⟨Prototypes 16b⟩ ≡

```
static Void valnum_oper(Operation *oper, Unsigned_Int which_table);
◇
```

Macro defined by scraps 8d, 9b, 13d, 14b, 16b, 17c, 19b, 24a, 29b, 34e, 36b, 38, 64c, 65b, 66b, 67b, 69a, 70b, 72bd, 73b, 75b, 76c, 84a.
Macro referenced in scrap 4a.

⟨Functions 16c⟩ ≡

```
static Void valnum_oper(Operation *oper, Unsigned_Int which_table)
{
    Unsigned_Int details = opcode_specs[oper->opcode].details;

    oper->critical = TRUE;
    if (details & EXPR)
        ⟨Value number an expression operation 17a⟩
    else if (details & LOAD)
        ⟨Value number a LOAD operation 23a⟩
    else if (details & STORE)
        ⟨Value number a STORE operation 25⟩
    else
        ⟨Give a unique value number to oper's defined registers and tags 26b⟩
}
◇
```

Macro defined by scraps 9a, 14a, 15, 16c, 18a, 19c, 24b, 29c, 35a, 36c, 39a, 65a, 66a, 67a, 68a, 69b, 71, 72c, 73ac, 75c, 77a, 84b.
Macro referenced in scrap 4a.

2.1.1 Value Numbering Expressions

Expressions are the most common type of operation for value numbering, so they get the most attention. The first thing we do is pull the pieces of the expression out of the `ILOC` operation. Next, we try to simplify the expression using commutativity, constant folding (if `-f` was specified), and algebraic simplification (if `-s` was specified). Since constant folding and algebraic simplification are highly dependent on the intermediate representation, we describe them in Appendices B and C. The next step is to search the hash table for the simplified expression to determine if it is redundant. Finally, we record whether or not the result is a constant. If so, we also record its value.

⟨Value number an expression operation 17a⟩ ≡
```
{
    unpack_expr(oper);
    ⟨Sort the operands of commutative operators 18b⟩
    ⟨Optionally try to simplify the expression 74a⟩
    ⟨Value number the expression 20a⟩
    ⟨Decide if oper computes a constant value 20b⟩
}
◇
```
Macro referenced in scrap 16c.

The pieces of the expression are stored in these global variables. They are global because they might be changed when the expression is simplified.

⟨Global Variables 17b⟩ ≡
```
static Opcode_Names opcode_name;
static Unsigned_Int dest_reg, value0, value1;
◇
```
Macro defined by scraps 4b, 6b, 7d, 12a, 13b, 17b, 33a, 34b, 44a, 61c.
Macro referenced in scrap 4a.

The `unpack_expr` routine pulls the pieces of an expression out of the `ILOC` operation and stores them in the above global variables. Generally, expressions have only 1 or 2 operands. There is one exception, the `xCONor` operations, which have a pair of extra parameters that can be ignored. The `constants` field of the operation determines which operands are constants and which are registers. We record the value number for register operands and the `Expr` for constant operands.

⟨Prototypes 17c⟩ ≡
```
static Void unpack_expr(Operation *oper);
◇
```
Macro defined by scraps 8d, 9b, 13d, 14b, 16b, 17c, 19b, 24a, 29b, 34e, 36b, 38, 64c, 65b, 66b, 67b, 69a, 70b, 72bd, 73b, 75b, 76c, 84a.
Macro referenced in scrap 4a.

⟨Functions 18a⟩ ≡

```
static Void unpack_expr(Operation *oper)
{
    Unsigned_Int operand;

    opcode_name = oper->opcode;
    switch (oper->referenced)
    {
        case 1:
            operand = oper->arguments[0];
            value0 = oper->constants ? operand : value_num[operand];
            value1 = 0;
            dest_reg = oper->arguments[1];
            break;
        case 2:
            operand = oper->arguments[0];
            value0 = oper->constants ? operand : value_num[operand];
            operand = oper->arguments[1];
            value1 = oper->constants > 1 ? operand : value_num[operand];
            dest_reg = oper->arguments[2];
            break;
        case 4:
            value0 = oper->arguments[2];
            value1 = value_num[oper->arguments[3]];
            dest_reg = oper->arguments[4];
            break;
        default:
            fprintf(stderr,
                    "Internal error: Unknown value for oper->referenced: %d\n",
                    oper->referenced);
            ABORT;
    }
}
◇
```

Macro defined by scraps 9a, 14a, 15, 16c, 18a, 19c, 24b, 29c, 35a, 36c, 39a, 65a, 66a, 67a, 68a, 69b, 71, 72c, 73ac, 75c, 77a, 84b.
Macro referenced in scrap 4a.

To be sure we notice the identity of expressions like `a + b` and `b + a`, we sort the operands of commutative operations. We always put the operand with the smaller value number first.

⟨Sort the operands of commutative operators 18b⟩ ≡

```
if (details & COMMUTE)
{
    if (value1 < value0)
    {
        Unsigned_Int temp = value0;
        value0 = value1;
        value1 = temp;
    }
}
else
    ⟨Check for xCMPle, xCMPge, ... 19a⟩
◇
```

Macro referenced in scrap 17a.

We also handle compares with inequalities. They are not really commutative *per se*, but we would like to recognize the equality of operations like `a < b` and `b > a`. Again, we put the operand with the smaller value number first.

⟨Check for `xCMPle`, `xCMPge`, ... 19a⟩ ≡
```
switch (opcode_name)
{
    case iCMPle:    fix_inequality(oper, iCMPge);    break;
    case fCMPle:    fix_inequality(oper, fCMPge);    break;
    case dCMPle:    fix_inequality(oper, dCMPge);    break;
    case iCMPge:    fix_inequality(oper, iCMPle);    break;
    case fCMPge:    fix_inequality(oper, fCMPle);    break;
    case dCMPge:    fix_inequality(oper, dCMPle);    break;
    case iCMPlt:    fix_inequality(oper, iCMPgt);    break;
    case fCMPlt:    fix_inequality(oper, fCMPgt);    break;
    case dCMPlt:    fix_inequality(oper, dCMPgt);    break;
    case iCMPgt:    fix_inequality(oper, iCMPlt);    break;
    case fCMPgt:    fix_inequality(oper, fCMPlt);    break;
    case dCMPgt:    fix_inequality(oper, dCMPlt);    break;
    default:        /* do nothing */                 break;
}
```
◇

Macro referenced in scrap 18b.

The `fix_inequality` routine tries to reverse the order of operands of an comparison based on inequality. Notice that we also overwrite the operation. This is because we are relying on the convention that `opcode_name` being different from `oper->opcode` indicates that we have simplified the operation.

⟨Prototypes 19b⟩ ≡
```
static Void fix_inequality(Operation *oper, Opcode_Names new_opcode);
```
◇

Macro defined by scraps 8d, 9b, 13d, 14b, 16b, 17c, 19b, 24a, 29b, 34e, 36b, 38, 64c, 65b, 66b, 67b, 69a, 70b, 72bd, 73b, 75b, 76c, 84a.
Macro referenced in scrap 4a.

⟨Functions 19c⟩ ≡
```
static Void fix_inequality(Operation *oper, Opcode_Names new_opcode)
{
    if (value0 < value1)
    {
        Unsigned_Int temp = value0;
        value0 = value1;
        value1 = temp;
        opcode_name = new_opcode;
        temp = oper->arguments[0];
        oper->arguments[0] = oper->arguments[1];
        oper->arguments[1] = temp;
        oper->opcode = new_opcode;
    }
}
```
◇

Macro defined by scraps 9a, 14a, 15, 16c, 18a, 19c, 24b, 29c, 35a, 36c, 39a, 65a, 66a, 67a, 68a, 69b, 71, 72c, 73ac, 75c, 77a, 84b.
Macro referenced in scrap 4a.

Once we have finished simplifying the expression, we are ready to decide if it is redundant. We look up the expression in the hash table by calling `expr_search`. If the return value is non-negative, it represents the value number of the expression, and we assign this value number to `dest_reg`. If we are value numbering over single basic blocks, extended basic blocks, or the dominator tree, we can remove the operation immediately (see Section 1.1). Otherwise, we rely on a later pass to decide if it is redundant.

If the return value from expr_search is negative, the expression was not found in the table. Therefore, we assign it the value number ⟨dest_reg⟩ and add it to the table.

⟨Value number the expression 20a⟩ ≡
```
if (⟨We have not already simplified oper 81b⟩)
{
    Int result = expr_search(which_table, opcode_name, value0, value1);
    if (result >= 0)
    {
        if (value_num[dest_reg] != result)
        {
            changed = TRUE;
            value_num[dest_reg] = result;
        }
        if (valnum_level < VALNUM_UNIFIED)
            oper->critical = FALSE;
    }
    else
    {
        if (value_num[dest_reg] != dest_reg)
        {
            changed = TRUE;
            value_num[dest_reg] = dest_reg;
        }
        expr_add(which_table, opcode_name, value0, value1, dest_reg);
    }
}
◇
```
Macro referenced in scrap 17a.

A register contains a constant value if it is the result of a xLDI operation. If the simplification process discovered that the result is a constant, the value of opcode_name will be changed to the appropriate xLDI opcode. Notice that we remove dest_reg from constants if the result is not a constant. This is because it may have previously been optimistically assumed to be constant (see Section 1.1).

⟨Decide if oper computes a constant value 20b⟩ ≡
```
if (fold_constants || simplify)
    if (⟨This is a xLDI operation 20c⟩)
    {
        VectorSet_Insert(constants, dest_reg);
        values[dest_reg] = value0;
    }
    else
        VectorSet_Delete(constants, dest_reg);
◇
```
Macro referenced in scrap 17a.

ILOC has several xLDI opcodes – one for each data type supported.

WARNING: This scrap is dependent on the order that the opcodes are enumerated in the file Shared.h.

⟨This is a xLDI operation 20c⟩ ≡
```
(iLDI <= opcode_name && opcode_name <= qLDI)
◇
```
Macro referenced in scrap 20b.

$r_0 \leftarrow$ LOAD	$[x_0]$	$r_0 \leftarrow$ LOAD	$[x_0]$
$r_1 \leftarrow$ LOAD	$[x_0]$		
$\vdots$		$\vdots$	
$r_2 \leftarrow \ldots$		$r_2 \leftarrow \ldots$	
STORE r_2	$[x_0][x_1]$	STORE r_2	$[x_0][x_1]$
$r_3 \leftarrow$ LOAD	$[x_1]$		
$\vdots$		$\vdots$	
STORE r_3	$[x_1][x_2]$		
Before		**After**	

Figure 2.2: Example program containing LOADs and STOREs

2.1.2 Value Numbering Load and Store Operations

Another improvement we can make to this technique is to trace values into and out of memory. To accomplish this, the front end annotates operations that can affect memory (*e.g.,* LOADs, STOREs, and CALLs) with lists of symbolic names (called *tags*) for distinct memory locations. For LOAD and STORE operations, the first tag in each list is the one that appears in the source code, and the others are possible aliases. During the conversion to SSA form, we insert ϕ-functions for tags and renumber them in the same manner as symbolic registers. Memory operations are hashed using another table similar to the ones used for expressions and ϕ-nodes.

We value number a LOAD operation by searching the hash table for a register that already contains the result. This can happen in two ways:

1. We could have previously loaded the value into a register.
2. We could have previously stored into this location from a register (without overwriting the location before we get to this LOAD).

To accommodate both cases, we must consistently hash LOAD and STORE operations. When we encounter a LOAD operation, we convert the opcode into the corresponding STORE opcode before hashing.[1] We search for the resulting expression in the hash table. If the search succeeds, then we remove the LOAD and use the expression's value number (SSA name) in its place. If the search fails, we assign a new, unique value number to the result and insert the expression into the table. Intuitively, we update the hash table as if the LOAD were followed by a STORE of the same value to the same location.

When we encounter a STORE operation, we want to determine whether the value already stored at that location is the same as the value being written. This is often the case for the STOREs inserted by the front end just before subroutine calls. We look up the value stored in the memory location and compare it to the value being stored. If the two values are equal, the STORE operation can be deleted. Otherwise, we add an entry to the hash table indicating the current value number contained in this location. In the presence of aliasing, the STORE will be annotated with a list of possible aliases for the location. Because aliasing is a MAY relationship[3], we must assign a new, unique value number to each alias.

The example in Figure 2.2 illustrates how this process works. Notice that the LOADs into r_0 and r_1 will produce the same value. When the first LOAD is processed, we search the hash table. The expression will not be found, so an entry is added to the table with value number $\langle r_0 \rangle$. When the next LOAD is processed, we will find the expression in the table and thus prove that $r_0 = r_1$. Next, the program stores a new value into location x. The STORE operation is marked with a list of both referenced and defined locations. The referenced list contains the "before" name of each location, and the defined list contains the "after" name

[1] In our compiler, LOAD and STORE operations are distinguished by the type of the value and the addressing mode.

of each location. If the value number of the "before" name is the same as the value number being written to this memory location, then the STORE is redundant and can be removed. In this case the value numbers ($\langle r_0 \rangle$ and $\langle r_2 \rangle$) are not equal, so we add the expression "STORE x_1" with the value number $\langle r_2 \rangle$ to the table. This entry is then used to prove that $r_2 = r_3$. Finally, the STORE into x from r_3 can be deleted because the expression "STORE x_1" and r_3 have the same value number, namely $\langle r_2 \rangle$.

For simplicity, we have ignored the address calculation used to access the variable in our example. For scalar variables it is irrelevant. On the other hand, the address must be considered when processing array variables. In reality, we consider the address to be another operand of the STORE expression (*e.g.,* STORE $\langle addr\ \ a_0 \rangle$ r_1). By doing this, we can sometimes distinguish between accesses to different elements of an array. Conservatively, each STORE to an array kills all other elements of the array. A subsequent LOAD from the array will be redundant only if it accesses the same SSA name for the memory location, and the address expressions have the same value number. Assume that the variable x in Figure 2.2 is an array rather than a scalar. If each access is to the same element and if value numbering can determine that the addresses for each access are equal, then the redundancies will be eliminated as before. However, if each access is to a different element, the address operand of the expressions will not match, and the program will remain unchanged.

Here is the actual code for value numbering a LOAD operation. There are two types of addressing modes supported by **ILOC**. The "offset+register" mode forms the address by adding a constant offset to the contents of a register. The "register+register" mode adds the contents of two registers. Therefore, the first operand could be either a register or a constant; for a register, we look up the value number, and for a constant, we use its **Expr** value.

If the return value from tag_search is non-negative, it represents the value number of the expression, and we assign this value number to dest_reg. If we are value numbering over single basic blocks, extended basic blocks, or the dominator tree, we can remove the operation immediately. Otherwise, we rely on a later pass to decide if it is redundant.

⟨Value number a LOAD operation 23a⟩ ≡

```
{
    Opcode_Names opcode = store_opcode(oper->opcode);
    Unsigned_Int offset_or_index =
        oper->constants > 2 ? oper->arguments[2] :
                              value_num[oper->arguments[2]];
    Unsigned_Int base_reg = value_num[oper->arguments[3]];
    Unsigned_Int dest_reg = oper->arguments[4];
    Unsigned_Int tag_value = value_num[oper->arguments[5]];
    Int result;

    ⟨Sort the operands of a commutative memory operation 23b⟩

    result = tag_search(which_table, opcode, offset_or_index, base_reg,
                        tag_value);
    if (result >= 0)
    {
        if (value_num[dest_reg] != result)
        {
            changed = TRUE;
            value_num[dest_reg] = result;
        }
        if (valnum_level < VALNUM_UNIFIED)
            oper->critical = FALSE;
    }
    else
    {
        if (value_num[dest_reg] != dest_reg)
        {
            changed = TRUE;
            value_num[dest_reg] = dest_reg;
        }
        tag_add(which_table, opcode, offset_or_index, base_reg, tag_value, dest_reg);
    }
}
◇
```

Macro referenced in scrap 16c.

The "register+register" addressing mode is commutative. We always put the operand with the smaller value number first.

⟨Sort the operands of a commutative memory operation 23b⟩ ≡

```
{
    Unsigned_Int details = opcode_specs[opcode].details;
    if (details & COMMUTE)
        if (base_reg < offset_or_index)
        {
            Unsigned_Int temp = offset_or_index;
            offset_or_index = base_reg;
            base_reg = temp;
        }
}
◇
```

Macro referenced in scraps 23a, 25.

In order to consistently hash LOAD and STORE operations, we will convert all LOAD opcodes into the corresponding STORE opcodes before hashing. The `store_opcode` function handles the conversion. In `ILOC`, there are *scalar* and *array* LOAD operations. These operations are also distinguished by the type and the addressing mode.

⟨Prototypes 24a⟩ ≡
```
static Opcode_Names store_opcode(Opcode_Names load_opcode);
◇
```
Macro defined by scraps 8d, 9b, 13d, 14b, 16b, 17c, 19b, 24a, 29b, 34e, 36b, 38, 64c, 65b, 66b, 67b, 69a, 70b, 72bd, 73b, 75b, 76c, 84a.
Macro referenced in scrap 4a.

⟨Functions 24b⟩ ≡
```
static Opcode_Names store_opcode(Opcode_Names load_opcode)
{
    switch (load_opcode)
    {
        case bSLDor:    return bSSTor;            case bSLDrr:    return bSSTrr;
        case wSLDor:    return wSSTor;            case wSLDrr:    return wSSTrr;
        case iSLDor:    return iSSTor;            case iSLDrr:    return iSSTrr;
        case fSLDor:    return fSSTor;            case fSLDrr:    return fSSTrr;
        case dSLDor:    return dSSTor;            case dSLDrr:    return dSSTrr;
        case cSLDor:    return cSSTor;            case cSLDrr:    return cSSTrr;
        case qSLDor:    return qSSTor;            case qSLDrr:    return qSSTrr;

        case bLDor:     return bSTor;             case bLDrr:     return bSTrr;
        case wLDor:     return wSTor;             case wLDrr:     return wSTrr;
        case iLDor:     return iSTor;             case iLDrr:     return iSTrr;
        case fLDor:     return fSTor;             case fLDrr:     return fSTrr;
        case dLDor:     return dSTor;             case dLDrr:     return dSTrr;
        case cLDor:     return cSTor;             case cLDrr:     return cSTrr;
        case qLDor:     return qSTor;             case qLDrr:     return qSTrr;

        default:
            fprintf(stderr,
                    "Internal error: Unknown load opcode - %s.\n",
                    opcode_specs[load_opcode].opcode);
            ABORT;
    }
}
◇
```
Macro defined by scraps 9a, 14a, 15, 16c, 18a, 19c, 24b, 29c, 35a, 36c, 39a, 65a, 66a, 67a, 68a, 69b, 71, 72c, 73ac, 75c, 77a, 84b.
Macro referenced in scrap 4a.

When we encounter a STORE operation, we want to determine if it is writing the same value that is currently stored in that location. This is often the case for the STOREs inserted just before subroutine calls. Just as we did for LOAD operations, we must decide if the first operand is a constant or a register. We use the value number of a register and the `Expr` for a constant.

We look up the value stored in the memory location and compare it to the value being stored. If the two values are equal, the STORE operation can be deleted. We must also assign unique value numbers to any possible aliases.

⟨Value number a STORE operation 25⟩ ≡

```
if (trace_memory)
{
    Opcode_Names opcode = oper->opcode;
    Unsigned_Int offset_or_index =
        oper->constants > 2 ? oper->arguments[2] :
                              value_num[oper->arguments[2]];
    Unsigned_Int base_reg = value_num[oper->arguments[3]];
    Unsigned_Int value = value_num[oper->arguments[4]];
    Unsigned_Int ref_tag = value_num[oper->arguments[5]];
    Unsigned_Int def_tag = *Operation_Second_List_Start(oper);
    Int result;

    ⟨Sort the operands of a commutative memory operation 23b⟩

    result = tag_search(which_table, opcode, offset_or_index, base_reg, ref_tag);
    if (result == value)
    {
        value_num[def_tag] = ref_tag;
        oper->critical = FALSE;
        ⟨Assign a unique value number to all the possible aliases 26a⟩
    }
    else
    {
        value_num[def_tag] = def_tag;
        ⟨Assign a unique value number to all the possible aliases 26a⟩
        tag_add(which_table, opcode, offset_or_index, base_reg, def_tag, value);
    }
}
else
{
    Unsigned_Int def_tag = *Operation_Second_List_Start(oper);
    value_num[def_tag] = def_tag;
    ⟨Assign a unique value number to all the possible aliases 26a⟩
}
◇
```

Macro referenced in scrap 16c.

Each STORE operation contains a list of defined tags. The first tag is the memory location that is definitely written by the operation (*i.e.,* it appears in the source code). The remaining tags are possible aliases. Because aliasing is a MAY relationship, each alias must be given a unique value number.

⟨Assign a unique value number to all the possible aliases 26a⟩ ≡

```
{
    Boolean first = TRUE;
    Unsigned_Int2 *tag_ptr;
    Operation_ForAllDefTags(tag_ptr, oper)
        if (first)
            first = FALSE;
        else
        {
            Unsigned_Int tag = *tag_ptr;
            value_num[tag] = tag;
        }
}
◇
```

Macro referenced in scrap 25.

2.1.3 Value Numbering Other Operations

Some operations (*e.g.,* **FRAME**, or **xJSRx**) define unknown values. For these operations, we simply assign a unique value number to all the defined registers and tags.

⟨Give a unique value number to oper's defined registers and tags 26b⟩ ≡

```
{
    Unsigned_Int2 *item_ptr;
    Operation_ForAllDefs(item_ptr, oper)
    {
        Unsigned_Int item = *item_ptr;
        if (value_num[item] != item)
        {
            changed = TRUE;
            value_num[item] = item;
        }
    }
    Operation_ForAllDefTags(item_ptr, oper)
    {
        Unsigned_Int item = *item_ptr;
        if (value_num[item] != item)
        {
            changed = TRUE;
            value_num[item] = item;
        }
    }
}
◇
```

Macro referenced in scrap 16c.

2.2 Dominator-Based Value Numbering

Given the primary steps to value number operations and ϕ-nodes, we are ready to piece them together to form the various types of value numbering. We implement several types of value numbering using variations of a preorder walk over the dominator tree: value numbering over single basic blocks, value numbering over extended basic blocks, dominator-based value numbering, and value numbering using the unified hash table. In a flow graph, if node X appears on every path from the start node to node Y, then X *dominates* Y ($X \geqslant\!\!\!\geqslant Y$)[13]. If $X \geqslant\!\!\!\geqslant Y$ and $X \neq Y$, then X *strictly dominates* Y ($X \gg Y$). The *immediate dominator* of

```
procedure DVNT(Block B)
    Mark the beginning of a new scope
    for each φ-function p of the form "n ← φ(...)" in B
        if p is meaningless or redundant
            Put the value number for p into VN[n]
            Remove p
        else
            VN[n] ← n
            Add p to the hash table
    for each assignment a of the form "x ← y op z" in B
        Overwrite y with VN[y] and z with VN[z]
        expr ← ⟨y op z⟩
        if expr can be simplified to expr′
            Replace a with "x ← expr′"
            expr ← expr′
        if expr is found in the hash table with value number v
            VN[x] ← v
            Remove a
        else
            VN[x] ← x
            Add expr to the hash table with value number x
    for each successor s of B
        Adjust the φ-function inputs in s
    for each child c of B in the dominator tree (in reverse postorder)
        DVNT(c)
    Clean up the hash table after leaving this scope
```

Figure 2.3: Dominator-based value numbering technique

Y (idom(Y)) is the closest strict dominator of Y[11]. In the routine's *dominator tree*, the parent of each node is its immediate dominator. When a routine is in SSA form, a value can enter a block B in one of two ways: either it is defined by a ϕ-function at the start of B or it flows through B's parent in the dominator tree. This observation led us to an algorithm that extends value numbering to larger regions by using the dominator tree.

Dominator-based value numbering processes each block by initializing the hash table with the information resulting from value numbering its parent in the dominator tree. To accomplish this, we use a scoped hash table. The value numbering proceeds by recursively walking the dominator tree. Figure 2.3 shows high-level pseudo-code for the algorithm.

When a redundant computation of an expression is found, the compiler removes the operation and replaces all uses of the defined SSA name with the expression's value number. The compiler can use this replacement scheme over a limited region of code – in blocks dominated by the operation and in parameters to ϕ-functions in the *dominance frontier* of the operation.[2] In both cases, we know that control must flow through the block where the first evaluation occurred (defining the SSA name's value).

The ϕ-functions require special treatment. Before the compiler can analyze the ϕ-functions in a block, it must previously have assigned value numbers to all of the inputs. This is not possible in all cases; specifically, any ϕ-function input whose value flows along a back edge (with respect to the dominator tree) cannot have a value number. If any of the parameters of a ϕ-function have not been assigned a value number, then the compiler cannot analyze the ϕ-function, and it must assign a new, unique value number to the result. The following two conditions guarantee that all ϕ-function parameters in a block have been assigned value

[2] The *dominance frontier* of node X is the set of nodes Y such that X dominates a predecessor of Y, but X does not strictly dominate Y (*i.e.*, $\mathrm{DF}(X) = \{Y \mid \exists P \in \mathrm{Pred}(Y), X \geq\!\!\!\geq P \text{ and } X \not\gg Y\}$).

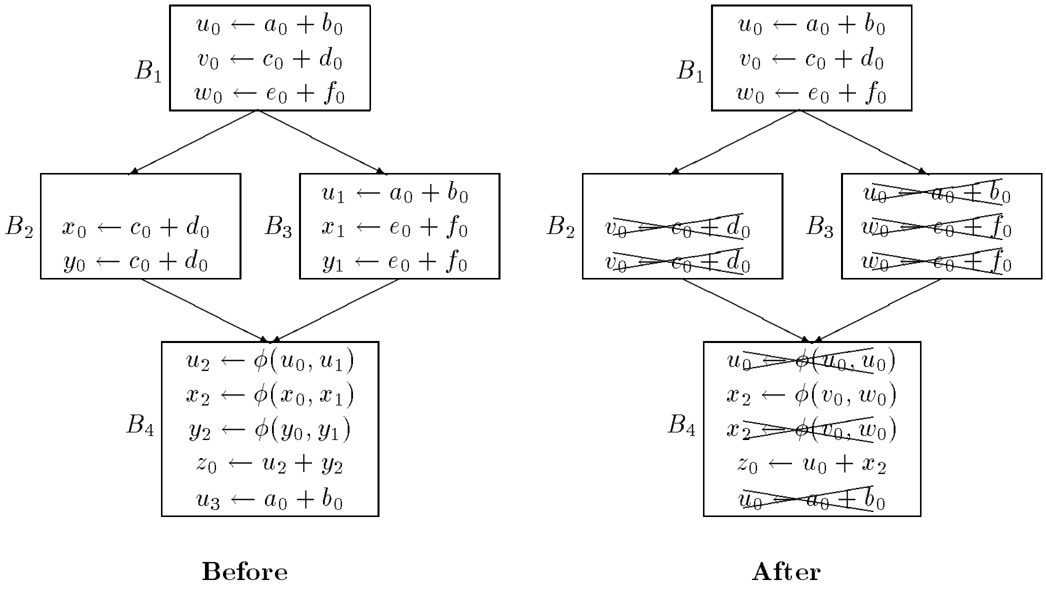


Figure 2.4: Dominator-tree value-numbering example

numbers:

1. When procedure DVNT (see Figure 2.3) is called recursively for the children of block B in the dominator tree, the children must be processed in reverse postorder. This ensures that all of a block's predecessors are processed before the block itself, unless the predecessor is connected by a back edge relative to the DFS tree.

2. The block must have no incoming back edges.

If the above conditions are met, we can analyze the ϕ-functions in a block and decide if they can be eliminated. A ϕ-function can be eliminated if it is meaningless or redundant. A ϕ-function is *meaningless* if all its inputs have the same value number. A meaningless ϕ-function can be removed if the references to its result are replaced with the value number of its input parameters. A ϕ-function is *redundant* if it computes the same value as another ϕ-function in the same block. The compiler can identify redundant ϕ-functions using a hashing scheme analogous to the one used for expressions. Without additional information about the conditions controlling the execution of different blocks, the compiler cannot compare ϕ-functions in different blocks.

After value numbering the ϕ-functions and the operations in a block, the algorithm visits each successor block and updates any ϕ-function inputs that come from the current block. This involves determining which ϕ-function parameter corresponds to input from the current block and overwriting the parameter with its value number. Notice the resemblance between this step and the corresponding step in the SSA construction algorithm. This step must be performed before value numbering any of the block's children in the dominator tree, if the compiler is going to analyze ϕ-functions.

To illustrate how the algorithm works, we will apply it to the code fragment in Figure 2.4. The first block processed will be B_1. Since none of the expressions have been seen, the names u_0, v_0, and w_0 will be assigned their SSA name as their value number.

The next block processed will be B_2. Since the expression $c_0 + d_0$ was defined in block B_1 (which dominates B_2), we can delete the two assignments in this block by assigning the value number for both x_0 and y_0 to be $\langle v_0 \rangle$. Before we finish processing block B_2, we must fill in the ϕ-function parameters in its successor block, B_4. The first argument of the ϕ-functions in B_4 corresponds to input from block B_2, so we replace u_0, x_0, and y_0 with $\langle u_0 \rangle$, $\langle v_0 \rangle$, and $\langle v_0 \rangle$, respectively.

Block B_3 will be visited next. Since every right-hand-side expression has been seen, we assign the value numbers for u_1, x_1, y_1 to be $\langle u_0 \rangle$, $\langle w_0 \rangle$, and $\langle w_0 \rangle$, respectively, and remove the assignments. To finish processing B_3, we fill in the second parameter of the ϕ-functions in B_4 with $\langle u_0 \rangle$, $\langle w_0 \rangle$, and $\langle w_0 \rangle$, respectively.

The final block processed will be B_4. The first step is to examine the ϕ-functions. Notice that we are able to examine the ϕ-functions only because we processed B_1's children in the dominator tree (B_2, B_3, and B_4) in reverse postorder and because there are no back edges flowing into B_4. The ϕ-function defining u_2 is meaningless because all its parameters are equal (They have the same value number – $\langle u_0 \rangle$). Therefore, we eliminate the ϕ-function by assigning u_2 the value number $\langle u_0 \rangle$. It is interesting to note that this ϕ-function was made meaningless by eliminating the only assignment to u in a block with B_4 in its dominance frontier. In other words, when we eliminate the assignment to u in block B_3, we eliminate the reason that the ϕ-function for u was inserted during the construction of SSA form. The second ϕ-function combines the values v_0 and w_0. Since this is the first appearance of a ϕ-function with these parameters, x_2 is assigned its SSA name as its value number. The ϕ-function defining y_2 is redundant because it is equal to x_2. Therefore, we eliminate this ϕ-function by assigning y_2 the value number $\langle x_2 \rangle$. When processing the assignments in the block, we replace each operand by its value number. This results in the expression $\langle u_0 \rangle + \langle x_2 \rangle$ in the assignment to z_0. The assignment to u_3 is eliminated by giving u_3 the value number $\langle u_0 \rangle$.

Notice that if we applied single-basic-block value numbering to this example, the only redundancies we could remove are the assignments to y_0 and y_1. If we applied extended-basic-block value numbering, we could also remove the assignments to x_0, u_1, and x_1. Only dominator-based value numbering can remove the assignments to u_2, y_2, and u_3.

To implement value numbering over single or extended basic blocks, we rely on the `visited` field of each `Block` to determine if it has been visited. Therefore, we clear all the flags before beginning.

⟨Perform dominator-based value numbering 29a⟩ ≡
```
{
    Block_Clear_Visited_Flags();
    DVNT(start_block);
}
◇
```
Macro referenced in scrap 11.

The `DVNT` routine is responsible for the walk over the dominator tree. It also creates and destroys a new scope within the hash table. Notice that some of the children in the dominator tree are visited after closing the current scope. This allows us to implement single and extended basic block value numbering.

⟨Prototypes 29b⟩ ≡
```
static Void DVNT(Block *block);
◇
```
Macro defined by scraps 8d, 9b, 13d, 14b, 16b, 17c, 19b, 24a, 29b, 34e, 36b, 38, 64c, 65b, 66b, 67b, 69a, 70b, 72bd, 73b, 75b, 76c, 84a.
Macro referenced in scrap 4a.

⟨Functions 29c⟩ ≡
```
static Void DVNT(Block *block)
{
    block->visited = TRUE;
    ⟨Mark the beginning of a new scope 62c⟩
    ⟨Value number the φ-nodes in block 30a⟩
    ⟨Value number the operations in block 31b⟩
    ⟨Visit children in the dominator tree based on valnum_level 31c⟩
    if (valnum_level < VALNUM_UNIFIED)
        ⟨Clean up after leaving a scope 63c⟩
    ⟨Value number any unvisited children in the dominator tree 32b⟩
}
◇
```
Macro defined by scraps 9a, 14a, 15, 16c, 18a, 19c, 24b, 29c, 35a, 36c, 39a, 65a, 66a, 67a, 68a, 69b, 71, 72c, 73ac, 75c, 77a, 84b.
Macro referenced in scrap 4a.

The simplest (and least useful) way to value number ϕ-nodes is to assign a new value number to each one. If we are performing dominator-based value numbering or value numbering using the unified hash table, we will perform more aggressive value numbering of ϕ-nodes. We rely on the fact that the dominator package orders the child list in the dominator tree in reverse postorder. This guarantees that any predecessors except those through back edges have already been processed, and thus all ϕ-node operands have been given a value number. Therefore, we can process the ϕ-nodes in `block` provided none of the incoming edges are back edges.

Notice that we do not handle values flowing in through back edges. We require SCC-based value numbering (see Section 2.3) to handle these types of congruences. Notice also that we only consider ϕ-nodes in the same block. We would like to look for equivalent ϕ-nodes in other blocks, but this would require additional knowledge about which conditionals control the function of ϕ-nodes in different blocks.

⟨Value number the ϕ-nodes in `block` 30a⟩ ≡

```
{
    Boolean processing_ok = valnum_level >= VALNUM_DOMINATOR_TREE;
    Boolean found_back_edge = FALSE;

    if (processing_ok)
        ⟨Decide if any incoming edges are back edges 30b⟩
    if (processing_ok && !found_back_edge)
        ⟨Process the φ-nodes in block 30c⟩
    else
        ⟨Assign a unique value number to each φ-node in block 31a⟩
}
◇
```

Macro referenced in scrap 29c.

Each edge in the CFG contains a flag indicating if it is a back edge. This flag was set during parsing of the `ILOC` file.

⟨Decide if any incoming edges are back edges 30b⟩ ≡

```
{
    Edge *edge;
    Block_ForAllPreds(edge, block)
        if (edge->back_edge)
        {
            found_back_edge = TRUE;
            break;
        }
}
◇
```

Macro referenced in scrap 30a.

If we determine that there are no incoming back edges, we pass each ϕ-node to the `valnum_phi_node` routine for processing.

⟨Process the ϕ-nodes in `block` 30c⟩ ≡

```
{
    Phi_Node *phi_node;
    Block_ForAllPhiNodes(phi_node, block)
        valnum_phi_node(phi_node, block, VALID);
}
◇
```

Macro referenced in scrap 30a.

If there are incoming back edges, we cannot analyze the ϕ-nodes in `block`. In this case, we will assign a unique value number to each one.

⟨Assign a unique value number to each ϕ-node in `block` 31a⟩ ≡

```
{
    Phi_Node *phi_node;
    Block_ForAllPhiNodes(phi_node, block)
    {
        Unsigned_Int name = phi_node->new_name;
        value_num[name] = name;
    }
}
◇
```

Macro referenced in scrap 30a.

After processing the ϕ-nodes, we pass each operation (in order) to **valnum_oper** for processing.

⟨Value number the operations in `block` 31b⟩ ≡

```
{
    Inst *inst;
    Block_ForAllInsts(inst, block)
    {
        Operation **oper_ptr;
        Inst_ForAllOperations(oper_ptr, inst)
            valnum_oper(*oper_ptr, VALID);
    }
}
◇
```

Macro referenced in scrap 29c.

If we are value numbering over extended basic blocks or over the entire dominator tree, we will visit some of this block's children before we clean up the hash tables.

⟨Visit children in the dominator tree based on `valnum_level` 31c⟩ ≡

```
if (valnum_level == VALNUM_EXTENDED_BASIC_BLOCKS)
    ⟨Recursively visit any extended basic blocks 31d⟩
else if (valnum_level >= VALNUM_DOMINATOR_TREE)
    ⟨Recursively visit children in the dominator tree 32a⟩
◇
```

Macro referenced in scrap 29c.

An extended basic block will be a child in the dominator tree with **block** as its only predecessor. We have a macro that iterates over all the children, so all we do is check the **pred_count** of each child.

⟨Recursively visit any extended basic blocks 31d⟩ ≡

```
{
    Block_List_Node *child;
    Dominator_ForChildren(child, block->dom_node)
    {
        Block *child_block = child->block;
        if (child_block->pred_count == 1)
            DVNT(child_block);
    }
}
◇
```

Macro referenced in scrap 31c.

To visit all the children, we use the same macro.

⟨Recursively visit children in the dominator tree 32a⟩ ≡
```
{
    Block_List_Node *child;
    Dominator_ForChildren(child, block->dom_node)
        DVNT(child->block);
}
◇
```
Macro referenced in scrap 31c.

After we have cleaned up the hash tables, we visit any unvisited children in the dominator tree. If we are value numbering the entire dominator tree, there will not be any unvisited children.

⟨Value number any unvisited children in the dominator tree 32b⟩ ≡
```
if (valnum_level < VALNUM_DOMINATOR_TREE)
{
    Block_List_Node *child;
    Dominator_ForChildren(child, block->dom_node)
    {
        Block *child_block = child->block;
        if (!child_block->visited)
            DVNT(child_block);
    }
}
◇
```
Macro referenced in scrap 29c.

2.3 SCC-Based Value Numbering

SCC-based value numbering is an algorithm that combines the ability to fold constants, recognize algebraic identities, and use commutativity with the ability to make optimistic assumptions and later prove or disprove them. It is centered around the strongly connected components (SCCs) of the SSA graph. It differs from dominator-based value numbering in two ways. First, it uses Tarjan's depth-first search (DFS) algorithm (operating over the SSA graph) to order the application of the primary step for value numbering [17]. Second, it uses a more complex form of the primary step that allows it to make optimistic assumptions about values that propagate along back edges and to prove or disprove those assumptions later.

To perform global value numbering, we need to apply the primary step to each instruction in the procedure. If the SSA-graph is acyclic, we can simply process the nodes in a single reverse postorder walk. This guarantees that all operands of an expression are visited before the expression itself is processed. Thus, a single pass will suffice. If cycles are present in the SSA-graph, then no such ordering exists. To handle this, we can convert the cyclic SSA-graph into an acyclic analog by finding the cycles, analyzing them to determine their effects, and replacing them in the graph with a representative node.

To organize the algorithm, we use Tarjan's technique for finding SCCs in a graph. An SCC is a maximal collection of nodes with the property that a path exists between any two nodes in the SCC. Every cycle in the SSA-graph must be part of some SCC. Thus, we can find the SCCs, analyze and collapse them, and walk the resulting graph in reverse postorder. We can avoid the need to actually construct the derived graph and walk it in a separate step by embedding the analysis into the SCC-finder because it already processes the nodes in the order that we require.

2.3.1 Finding Strongly Connected Components

Figure 2.5 shows the SCC-finding algorithm. It uses a stack to accumulate the nodes for each SCC. All the nodes in a given SCC are popped from the stack together. The order in which it pops the SCCs is critical. When an SCC is popped, any operand referenced in an operation inside the SCC is either defined inside the

```
DFS(node)
    node.DFSnum ← nextDFSnum++
    node.visited ← TRUE
    node.low ← node.DFSnum
    PUSH(node)
    for each o ∈ {operands of node}
        if not o.visited
            DFS(o)
            node.low ← MIN(node.low, o.low)
        if o.DFSnum < node.DFSnum and o ∈ stack
            node.low ← MIN(o.DFSnum, node.low)
    if node.low = node.DFSnum
        SCC ← ∅
        do
            x ← POP()
            SCC ← SCC ∪ {x}
        while x ≠ node
        process_scc(SCC)
```

Figure 2.5: Tarjan's SCC-finding algorithm

SCC itself or else it is defined in an SCC that has already been popped from the stack. This order precisely fits our need; as each SCC is popped from the stack, it is passed to `process_scc()`.

We will need a separate arena to support Tarjan's algorithm.

⟨Global Variables 33a⟩ ≡

```
static Arena dfs_arena = NULL;
```

◇

Macro defined by scraps 4b, 6b, 7d, 12a, 13b, 17b, 33a, 34b, 44a, 61c.
Macro referenced in scrap 4a.

The DFS algorithm operates over the SSA graph. Unlike DFS applied to the CFG, we can not simply start the process at a single node. There is no single node that will guarantee that we reach all other nodes, so we repeatedly start DFS on any unvisited node. We start with the larger SSA names because they generally are functions of smaller names, so each search will hopefully reach more nodes in the SSA graph.

⟨Perform SCC-based value numbering 33b⟩ ≡

```
{
    dfs_arena = Arena_Create();
    ⟨Mark the beginning of a new scope 62c⟩
    ⟨Initialize the date structures for depth-first search 34c⟩
    ⟨Repeatedly call DFS at the largest unvisited SSA name 34d⟩
    Arena_Destroy(dfs_arena);
}
```

◇

Macro referenced in scrap 11.

For each SSA name, we keep the following information to support DFS.

⟨Type Declarations 34a⟩ ≡

```
typedef struct
{
    Boolean visited;          /* if the node has been visited yet */
    Boolean in_stack;         /* if the node is currenty in the SCC stack */
    Unsigned_Int2 dfs_num;    /* the DFS number */
    Unsigned_Int2 low;        /* the node with the smallest DFS number that
                                 is reachable from this node */
    Unsigned_Int2 next;       /* the next node in the SCC stack */
} DFS_INFO;
```

◇

Macro defined by scraps 34a, 51b, 58, 59ab, 60abcd, 61ab.
Macro referenced in scrap 4a.

We will need some global variables to support DFS.

⟨Global Variables 34b⟩ ≡

```
static DFS_INFO *dfs_info;
static Unsigned_Int dfs_stack;
static Unsigned_Int next_dfs_num = 0;
```

◇

Macro defined by scraps 4b, 6b, 7d, 12a, 13b, 17b, 33a, 34b, 44a, 61c.
Macro referenced in scrap 4a.

We allocate `dfs_info` with `Arena_GetMemClear` so that all of the `visited` and `in_stack` flags are initially `FALSE`.

⟨Initialize the date structures for depth-first search 34c⟩ ≡

```
dfs_info = Arena_GetMemClear(dfs_arena, SSA_def_count*sizeof(DFS_INFO));
```

◇

Macro referenced in scrap 33b.

Since we asked `SSA_Build` to fold copies, `SSA_def_count` is slightly too high. Therefore, before calling `DFS` we check the `SSA_use_def_chains` entry to see if we have a valid name.

⟨Repeatedly call `DFS` at the largest unvisited SSA name 34d⟩ ≡

```
{
    Unsigned_Int i;

    for (i = SSA_def_count - 1; i > 0; i--)
        if (SSA_use_def_chains[i].op_pointer.operation && !dfs_info[i].visited)
            DFS(i);
}
```

◇

Macro referenced in scrap 33b.

The `DFS` routine is responsible for finding SCCs in the value graph. As it does, it assigns value numbers.

⟨Prototypes 34e⟩ ≡

```
static Void DFS(Unsigned_Int SSA_name);
```

◇

Macro defined by scraps 8d, 9b, 13d, 14b, 16b, 17c, 19b, 24a, 29b, 34e, 36b, 38, 64c, 65b, 66b, 67b, 69a, 70b, 72bd, 73b, 75b, 76c, 84a.
Macro referenced in scrap 4a.

⟨Functions 35a⟩ ≡

```
static Void DFS(Unsigned_Int SSA_name)
{
    dfs_info[SSA_name].visited = TRUE;
    dfs_info[SSA_name].dfs_num = dfs_info[SSA_name].low = next_dfs_num++;
    ⟨Push SSA_name onto the DFS stack 35b⟩

    if (SSA_use_def_chains[SSA_name].is_phi_node)
        ⟨Visit the arguments of a φ-node 35c⟩
    else
        ⟨Visit the registers and tags referenced by an operation 36a⟩

    ⟨Decide if we are ready to assign a value number to SSA_name 37⟩
}
◇
```

Macro defined by scraps 9a, 14a, 15, 16c, 18a, 19c, 24b, 29c, 35a, 36c, 39a, 65a, 66a, 67a, 68a, 69b, 71, 72c, 73ac, 75c, 77a, 84b.
Macro referenced in scrap 4a.

Pushing a name onto the stack is a matter of putting it at the front of the list and setting the **in_stack** field.

⟨Push **SSA_name** onto the DFS stack 35b⟩ ≡

```
dfs_info[SSA_name].in_stack = TRUE;
dfs_info[SSA_name].next = dfs_stack;
dfs_stack = SSA_name;
◇
```

Macro referenced in scrap 35a.

Since the mechanisms for iterating over the arguments of ϕ-nodes and operations are different, we use the **visit_argument** function for the actual processing. Notice that we ignore any arguments of a ϕ-node that are zero. These represent uninitialized values.

⟨Visit the arguments of a ϕ-node 35c⟩ ≡

```
{
    Phi_Node *phi_node = SSA_use_def_chains[SSA_name].op_pointer.phi_node;
    Unsigned_Int2 *arg_ptr;

    Phi_Node_ForAllParms(arg_ptr, phi_node)
    {
        Unsigned_Int arg = *arg_ptr;
        if (arg) visit_argument(SSA_name, arg);
    }
}
◇
```

Macro referenced in scrap 35a.

Operations have two types of operands – registers and tags. Normally, we simply visit each operand of the operation. The one exception is when `SSA_name` represents a possible alias of the tag defined by a STORE operation. Each STORE operation contains a list of defined tags. The first tag is the memory location that is definitely written by the operation (*i.e.,* it appears in the source code). The shared routine `Operation_Second_List_Start` returns a pointer to the first tag defined by the operation. The remaining tags are possible aliases. We treat them as if they have no operands.

⟨Visit the registers and tags referenced by an operation 36a⟩ ≡

```
{
    Operation *oper = SSA_use_def_chains[SSA_name].op_pointer.operation;
    Unsigned_Int details = opcode_specs[oper->opcode].details;

    if (!(details & STORE) || SSA_name == *Operation_Second_List_Start(oper))
    {
        Unsigned_Int2 *name;

        Operation_ForAllUses(name, oper)
            visit_argument(SSA_name, *name);

        Operation_ForAllRefTags(name, oper)
            visit_argument(SSA_name, *name);
    }
}
◇
```

Macro referenced in scrap 35a.

The `visit_argument` routine handles the recursive calls to `DFS` for operands of ϕ-nodes and operations. It also updates the `low` field for the node based on the results of the recursive call.

⟨Prototypes 36b⟩ ≡

```
static void visit_argument(Unsigned_Int name, Unsigned_Int arg);
◇
```

Macro defined by scraps 8d, 9b, 13d, 14b, 16b, 17c, 19b, 24a, 29b, 34e, 36b, 38, 64c, 65b, 66b, 67b, 69a, 70b, 72bd, 73b, 75b, 76c, 84a.
Macro referenced in scrap 4a.

⟨Functions 36c⟩ ≡

```
static void visit_argument(Unsigned_Int name, Unsigned_Int arg)
{
    if (!dfs_info[arg].visited)
    {
        DFS(arg);
        dfs_info[name].low =
            MIN(dfs_info[name].low, dfs_info[arg].low);
    }

    if (dfs_info[arg].dfs_num < dfs_info[name].dfs_num && dfs_info[arg].in_stack)
        dfs_info[name].low = MIN(dfs_info[arg].dfs_num, dfs_info[name].low);
}
◇
```

Macro defined by scraps 9a, 14a, 15, 16c, 18a, 19c, 24b, 29c, 35a, 36c, 39a, 65a, 66a, 67a, 68a, 69b, 71, 72c, 73ac, 75c, 77a, 84b.
Macro referenced in scrap 4a.

⟨Macros 36d⟩ ≡

```
#define MIN(x, y)              (((x) < (y)) ? (x) : (y))
◇
```

Macro defined by scraps 3, 7ab, 12b, 36d.
Macro referenced in scrap 4a.

When the low and the dfs_num fields are equal, we have identified an SCC, and we are ready to assign value numbers. We pop names from the DFS stack until we reach SSA_name. Popping the DFS stack involves removing the item at the front of the list and clearing the in_stack flag for that name.

⟨Decide if we are ready to assign a value number to SSA_name 37⟩ ≡

```
if (dfs_info[SSA_name].low == dfs_info[SSA_name].dfs_num)
{
    Unsigned_Int scc = 0;
    Unsigned_Int name;
    do
    {
        name = dfs_stack;
        dfs_stack = dfs_info[name].next;
        dfs_info[name].in_stack = FALSE;
        dfs_info[name].next = scc;
        scc = name;
    } while (name != SSA_name);
    process_scc(scc);
}
◇
```

Macro referenced in scrap 35a.

2.3.2 Processing Strongly Connected Components

The process_scc routine handles SCCs as they are identified by Tarjan's algorithm. Two cases arise, based on the size of the SCC.

Single node In the SSA-graph, the only type of node that can be involved in a self-cycle is a ϕ-node. If the node is involved in a cycle, the primary ϕ-node processing will suffice. If the node is not involved in any cycle, its operands were already processed. In either case, process_scc passes it directly to the primary value numbering step.

Multiple nodes The SCC contains a cycle in the SSA-graph. Recall that Tarjan's algorithm has the property that when an SCC is popped, any operands from outside the SCC have already been assigned value numbers. Assigning value numbers to the nodes inside the SCC requires special handling. This is the key to making the algorithm both global and optimistic.

Introducing Optimism

Simply inserting the primary step of hash-based value numbering into the SCC framework produces a pessimistic algorithm. To capture the optimistic properties of the partitioning algorithm, we must process the multiple-node SCCs in a special way.

At the heart of optimistic analysis is the ability to assume that some fact is true, to continue analyzing the code, and, later, to discover that the fact was false. The challenge is to accommodate this sort of speculation in a hash-based system. To do this, we rely on two key features of SCC-based value numbering:

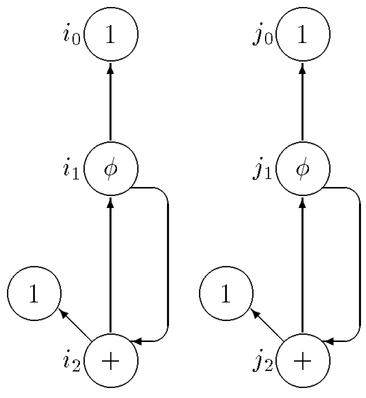
1. We iterate through the members of a multiple-node SCC in reverse postorder, with respect to the control-flow graph. At each node, process_scc invokes the primary step. Iteration continues until it reaches a fixed point for the value numbers in the SCC.

2. We use two hash tables: a *valid table* and an *optimistic table*.[3] The valid table contains only facts that are known to be true. The optimistic table contains facts that may later be disproven. Single node SCCs are always processed with the valid table, since all of their operands have already been assigned a valid value number. Multiple-node SCCs are processed iteratively using the optimistic table. The iteration converges to a fixed point in at most $2D(\text{SSA})+2$ passes, where $D(\text{SSA})$ is the *loop connectedness* (the maximum number of back edges on any acyclic path) of the SSA graph. After the iteration stabilizes, we know that the value numbers for the SCC are valid. We record the valid entries by processing the SCC one final time using the valid table. The effect of this step is to copy the valid entries from the

[3] To save memory, the two tables are implemented as a single table whose entries can have a valid value number and an optmistic value number (see Chapter A.

$i_0 \leftarrow 1$
$j_0 \leftarrow 1$
while $(\ldots)$
  $i_1 \leftarrow \phi(i_0, i_2)$
  $j_1 \leftarrow \phi(j_0, j_2)$
  $i_2 \leftarrow i_1 + 1$
  $j_2 \leftarrow j_1 + 1$

SSA Form



SSA Graph

Figure 2.6: Example with equal induction variables

optimistic table into the valid table. Thus, any entries not copied represent assumptions that have been disproven. There is no explicit step to disprove facts; they are disproven implicitly.

To further clarify the algorithm, consider how it would proceed if given the example in Figure 2.6. The values i_0 and j_0 are not contained in any cycle, so they will be assigned value numbers before either of the SCCs. Assume they are each given the value number $\langle i_0 \rangle$ and that the SCC containing i_1 and i_2 is processed next. During the first pass over the SCC, the ϕ-node $\phi(\langle i_0 \rangle, \top)$ will be simplified (optimistically) to $\langle i_0 \rangle$, and i_1 will be given value number $\langle i_0 \rangle$. Then, the expression defining i_2, $\langle i_0 \rangle + 1$, can be simplified to 2.[4] An entry mapping the constant 2 to value number $\langle i_2 \rangle$ will be added to the *optimistic* table. During the second pass, the expression $\phi(\langle i_0 \rangle, \langle i_2 \rangle)$ cannot be simplified, so an entry mapping the expression to $\langle i_1 \rangle$ is added to the *optimistic* table. Next, an entry mapping $\langle i_1 \rangle + 1$ to $\langle i_2 \rangle$ will be added to the *optimistic* table. At this point the value numbers have stabilized, so we make a final pass using the *valid* table. During this pass, we add entries mapping $\phi(\langle i_0 \rangle, \langle i_2 \rangle)$ to $\langle i_1 \rangle$ and mapping $\langle i_1 \rangle + 1$ to $\langle i_2 \rangle$ to the *valid* table. Notice that the optimistic entry mapping 2 to $\langle i_2 \rangle$ is not added to the *valid* table - this assumption has been disproven implicitly.

The next step is to process the SCC containing j_1 and j_2. During the first pass, the expression $\phi(\langle i_0 \rangle, \top)$ will be simplified to $\langle i_0 \rangle$, and j_1 will be given the value number $\langle i_0 \rangle$. Next, the expression $\langle i_0 \rangle + 1$ can be simplified to 2; it will be found in the *optimistic* table with value number $\langle i_2 \rangle$. During the second pass, the expression $\phi(\langle i_0 \rangle, \langle i_2 \rangle)$ will be found in the *optimistic* table with value number $\langle i_1 \rangle$, and $\langle i_1 \rangle + 1$ will be found with value number $\langle i_2 \rangle$. At this point the value numbers have stabilized, so we process the SCC using the *valid* table. Since entries already exist mapping $\phi(\langle i_0 \rangle, \langle i_2 \rangle)$ to $\langle i_1 \rangle$ and $\langle i_1 \rangle + 1$ to $\langle i_2 \rangle$, no new entries will be added to the *valid* table. Thus, the algorithm has determined that i_1 is equal to j_1 and i_2 is equal to j_2.

The `process_scc` routine assigns value numbers to the members of an SCC. The members of the SCC are linked together using the `next` field of the `dfs_info` array. Therefore, we can determine if the SCC is a single node by checking for a second item in the list.

$\langle$Prototypes 38$\rangle \equiv$
```
static Void process_scc(Unsigned_Int scc);
```
$\diamond$

Macro defined by scraps 8d, 9b, 13d, 14b, 16b, 17c, 19b, 24a, 29b, 34e, 36b, 38, 64c, 65b, 66b, 67b, 69a, 70b, 72bd, 73b, 75b, 76c, 84a.
Macro referenced in scrap 4a.

[4] Remember that expressions are formed from an operator and the value numbers of the operands, not the operands themselves.

⟨Functions 39a⟩ ≡

```
static Void process_scc(Unsigned_Int scc)
{
    if (!dfs_info[scc].next)
        ⟨Value number a single node scc using the valid table 39b⟩
    else
        ⟨Value number a multiple node scc 39c⟩
}
◇
```

Macro defined by scraps 9a, 14a, 15, 16c, 18a, 19c, 24b, 29c, 35a, 36c, 39a, 65a, 66a, 67a, 68a, 69b, 71, 72c, 73ac, 75c, 77a, 84b.
Macro referenced in scrap 4a.

Single names are processed using the valid table. Normally, we pass the node to the appropriate primary step. The one exception is when `scc` represents a possible alias of the tag defined by a STORE operation. Each STORE operation contains a list of defined tags. The first tag is the memory location that is definitely written by the operation (*i.e.,* it appears in the source code). The shared routine `Operation_Second_List_Start` returns a pointer to the first tag defined by the operation. The remaining tags are possible aliases. These are assigned a unique value number without calling `valnum_oper`.

⟨Value number a single node scc using the valid table 39b⟩ ≡

```
if (SSA_use_def_chains[scc].is_phi_node)
    valnum_phi_node(SSA_use_def_chains[scc].op_pointer.phi_node,
                    SSA_block_map[scc], VALID);
else
{
    Operation *oper = SSA_use_def_chains[scc].op_pointer.operation;
    Unsigned_Int details = opcode_specs[oper->opcode].details;
    if ((details & STORE) && scc != *Operation_Second_List_Start(oper))
        value_num[scc] = scc;
    else
        valnum_oper(oper, VALID);
}
◇
```

Macro referenced in scrap 39a.

Multiple node SCCs require special handling. First we must sort the members in reverse postorder with respect to the CFG. Next, we iteratively value number the members using the optimistic table until a fixed point is reached. Finally, we process the SCC one more time using the valid tables.

⟨Value number a multiple node scc 39c⟩ ≡

```
{
    ⟨Sort the members of scc in reverse postorder 40⟩
    ⟨Iteratively value number the members of scc using the optimistic table 42a⟩
    ⟨Value number the members of scc using the valid table 42b⟩
}
◇
```

Macro referenced in scrap 39a.

We use a bucket-sorting algorithm for the nodes in the SCC. Each bucket will contain a list of nodes defined in a particular block. First, we place each member of `scc` into the bucket for its defining block. Next, the contents of each bucket are put together to form a linked list. To make chaining the buckets together easier, we will insert names into each bucket in reverse order. They will be removed from these lists and placed into the final list in reverse order.

⟨Sort the members of `scc` in reverse postorder 40⟩ ≡

```
{
    Unsigned_Int2 *blocks;
    Unsigned_Int name = scc;
    Unsigned_Int i;

    Arena_Mark(dfs_arena);
    blocks = Arena_GetMemClear(dfs_arena, (block_count+1)*sizeof(Unsigned_Int2));

    while (name)
    {
        Unsigned_Int block_index = SSA_block_map[name]->postorder_index;
        Unsigned_Int next = dfs_info[name].next;

        ⟨Insert name into the list at blocks[block_index] 41a⟩
        name = next;
    }

    ⟨Create a linked list from the buckets 41b⟩

    Arena_Release(dfs_arena);
}
◇
```

Macro referenced in scrap 39c.

We must keep the list of names in each bucket sorted. Remember that we are sorting in reverse order. We search the list for the appropriate location and insert the name.

⟨Insert `name` into the list at `blocks[block_index]` 41a⟩ ≡

```
{
    Unsigned_Int name_index;
    Unsigned_Int pred = 0;

    for (name_index = blocks[block_index];
         name_index;
         name_index = dfs_info[name_index].next)
    {
        if (name_index < name)
            break;
        pred = name_index;
    }

    if (pred)
    {
        dfs_info[name].next = dfs_info[pred].next;
        dfs_info[pred].next = name;
    }
    else
    {
        dfs_info[name].next = blocks[block_index];
        blocks[block_index] = name;
    }
}
◇
```

Macro referenced in scrap 40.

Since each bucket contains a list of names in reverse order, we can insert each name at the front of the list – effectively reversing the order of the list.

⟨Create a linked list from the buckets 41b⟩ ≡

```
scc = 0;
for (i = 0; i <= block_count; i++)
{
    Unsigned_Int name_index = blocks[i];
    while (name_index)
    {
        Unsigned_Int next_index = dfs_info[name_index].next;
        dfs_info[name_index].next = scc;
        scc = name_index;
        name_index = next_index;
    }
}
◇
```

Macro referenced in scrap 40.

During the iterative processing, we use the optimistic tables. Normally, we call the appropriate primary value numbering step for each node in the SCC. The one exception is when `name` represents a possible alias of the tag defined by a STORE operation. Each STORE operation contains a list of defined tags. The first tag is the memory location that is definitely written by the operation (*i.e.*, it appears in the source code). The shared routine `Operation_Second_List_Start` returns a pointer to the first tag defined by the operation. The remaining tags are possible aliases. These are assigned a unique value number without calling `valnum_oper`.

The global variable `changed` will be set if any value numbers change.

⟨Iteratively value number the members of scc using the optimistic table 42a⟩ ≡

```
do
{
    Unsigned_Int name;

    changed = FALSE;
    for (name = scc; name; name = dfs_info[name].next)
        if (SSA_use_def_chains[name].is_phi_node)
            valnum_phi_node(SSA_use_def_chains[name].op_pointer.phi_node,
                            SSA_block_map[name], OPTIMISTIC);
        else
        {
            Operation *oper = SSA_use_def_chains[name].op_pointer.operation;
            Unsigned_Int details = opcode_specs[oper->opcode].details;
            if ((details & STORE) && name != *Operation_Second_List_Start(oper))
                value_num[name] = name;
            else
                valnum_oper(oper, OPTIMISTIC);
        }
} while (changed);
◇
```

Macro referenced in scrap 39c.

After the iteration reaches a fixed point, we know that all the value numbers are valid, so we add the entries to the valid table. Again, we call the appropriate primary step.

⟨Value number the members of scc using the valid table 42b⟩ ≡

```
{
    Unsigned_Int name;

    for (name = scc; name; name = dfs_info[name].next)
        if (SSA_use_def_chains[name].is_phi_node)
            valnum_phi_node(SSA_use_def_chains[name].op_pointer.phi_node,
                            SSA_block_map[name], VALID);
        else
        {
            Operation *oper = SSA_use_def_chains[name].op_pointer.operation;
            Unsigned_Int details = opcode_specs[oper->opcode].details;
            if ((details & STORE) && name != *Operation_Second_List_Start(oper))
                value_num[name] = name;
            else
                valnum_oper(oper, VALID);
        }
}
◇
```

Macro referenced in scrap 39c.

Chapter 3

Renumbering

Once we have assigned a value number to each SSA name, we are ready to move into the next phase of value numbering. We rewrite the ϕ-nodes and operations in the routine based on the value numbers. Since we have the value number of every SSA name stored in the `value_num` array, we can do this in a single pass over the routine. However, if the user wants to prepare the output for partial redundancy elimination or value-driven code motion, the register numbers must obey certain conventions. Since the value numbers are more or less arbitrary, we will create a mapping from value numbers to register numbers that obey the naming rules. We will create this mapping by overwriting the `value_num` array. This will allow us to perform renumbering as a separate step that is independent of this mapping.

3.1 Preparing for Partial Redundancy Elimination

Our implementations of partial redundancy elimination and value-driven code motion require the registers to be numbered in such a way that the register number of an expression register will be higher than the register numbers of its operands [7]. To accomplish this, we walk the CFG in reverse postorder to guarantee that the operands of an expression are visited before the expression. During this walk, we rename value numbers in the order of appearance. We will use an array called `name_map` to hold the mapping of value numbers to names in the new numbering scheme. We use the convention that a zero entry in `name_map` indicates an invalid entry. The only exception will be the 0^{th} entry which represents an undefined value.

⟨Optionally invent new value numbers 43⟩ ≡

```
if (prepare_for_partial || do_vdcm)
{
    Arena partial_arena = Arena_Create();
    Unsigned_Int2 *name_map =
        Arena_GetMemClear(partial_arena, SSA_def_count*sizeof(Unsigned_Int2));

    if (debug >= MAJOR_PHASES)
        fprintf(stderr, "Invent new register names\n");

    ⟨Set up the mapping 44b⟩
    ⟨Overwrite the value_num array 45b⟩
    Arena_Destroy(partial_arena);
}
◇
```

Macro referenced in scrap 5.

We use `next_name` to keep track of the next available name. This value will be used in several other places, so we use a global variable.

⟨Global Variables 44a⟩ ≡
```
static Unsigned_Int next_name = 0;
◇
```
Macro defined by scraps 4b, 6b, 7d, 12a, 13b, 17b, 33a, 34b, 44a, 61c.
Macro referenced in scrap 4a.

⟨Set up the mapping 44b⟩ ≡
```
{
    Block *block;

    ⟨Add a mapping for zero 44c⟩
    ForAllBlocks_rPostorder(block)
    {
        ⟨Create a new value number for any φ-nodes 44d⟩
        ⟨Create a new value number for any operations 45a⟩
    }
}
◇
```
Macro referenced in scrap 43.

The value number zero signifies an undefined value. We preserve this convention by forcing zero to always map to itself.

⟨Add a mapping for zero 44c⟩ ≡
```
name_map[0] = next_name++;
◇
```
Macro referenced in scrap 44b.

Any ϕ-node whose value number has not been assigned a new value number will be given one. Notice that we only renumber the tags in the routine if we are performing value-driven code motion.

⟨Create a new value number for any ϕ-nodes 44d⟩ ≡
```
{
    Phi_Node *phi_node;
    Block_ForAllPhiNodes(phi_node, block)
        if ((do_vdcm || phi_node->old_name < register_count) &&
            equal_parms(phi_node) == -1)
        {
            Unsigned_Int vn = value_num[phi_node->new_name];
            if (vn && !name_map[vn])
                name_map[vn] = next_name++;
        }
}
◇
```
Macro referenced in scrap 44b.

We iterate through all the operations in the block and assign a new register number to any defined item whose value number does not have an entry in `name_map`. Notice that we only renumber the tags in the routine if we are performing value-driven code motion.

⟨Create a new value number for any operations 45a⟩ ≡

```
{
    Inst *inst;
    Block_ForAllInsts(inst, block)
    {
        Operation **oper_ptr;
        Inst_ForAllOperations(oper_ptr, inst)
        {
            Operation *oper = *oper_ptr;
            Unsigned_Int2 *def_ptr;
            Operation_ForAllDefs(def_ptr, oper)
            {
                Unsigned_Int vn = value_num[*def_ptr];
                if (vn && !name_map[vn])
                    name_map[vn] = next_name++;
            }
            if (do_vdcm)
                Operation_ForAllDefTags(def_ptr, oper)
                {
                    Unsigned_Int vn = value_num[*def_ptr];
                    if (vn && !name_map[vn])
                        name_map[vn] = next_name++;
                }
        }
    }
}
◇
```

Macro referenced in scrap 44b.

Once we have built `name_map`, we must store these values in the `value_num` array. Overwriting the array allows us to renumber the routine without knowing that this mapping is being used.

⟨Overwrite the `value_num` array 45b⟩ ≡

```
{
    Unsigned_Int i;
    for (i = 0; i < SSA_def_count; i++)
    {
        Unsigned_Int vn = value_num[i];
        if (vn != TOP && name_map[vn])
            value_num[i] = name_map[vn];
    }
}
◇
```

Macro referenced in scrap 43.

3.2 Renumbering the ϕ-nodes and Registers

Once the value numbering is completed and the register mapping (if requested) is set up, we are ready to renumber the ϕ-nodes and registers in the routine according to their value numbers. We accomplish this by overwriting every name in the routine by the corresponding value in the `value_num` array.

⟨Renumber the ϕ-nodes and operations in the routine 46a⟩ ≡

```
{
    Block *block;
    if (debug >= MAJOR_PHASES)
        fprintf(stderr, "Renumber the phi-nodes and operations in the routine\n");
    ForAllBlocks(block)
    {
        ⟨Renumber the φ-nodes that define registers 46b⟩
        ⟨Renumber the operations 47b⟩
    }
    sprintf(time_message, "%s: Renumbering required %1.2f seconds\n",
            file_name ? file_name : "stdin");
    Time_Dump(time, time_message);
}
◇
```

Macro referenced in scrap 5.

We iterate through the ϕ-nodes that define registers and renumber them according to their value number. Notice that we do not renumber ϕ-nodes that define tags. Before renumbering a ϕ-node, we determine if it is redundant. If so, we remove it by setting all its parameters and its result to zero.

⟨Renumber the ϕ-nodes that define registers 46b⟩ ≡

```
{
    Phi_Node *phi_node;
    Block_ForAllPhiNodes(phi_node, block)
        if (phi_node->old_name < register_count)
            if (equal_parms(phi_node) == value_num[phi_node->new_name])
                ⟨Remove phi_node from block 47a⟩
            else
            {
                Unsigned_Int2 *arg_ptr;
                Phi_Node_ForAllParms(arg_ptr, phi_node)
                    *arg_ptr = value_num[*arg_ptr];

                phi_node->new_name = value_num[phi_node->new_name];
            }
}
◇
```

Macro referenced in scrap 46a.

We remove a ϕ-node by setting all of its parameters to zero. This prevents `SSA_Replace_Phi_Nodes` from inserting any copies for it. We also set its `new_name` to zero, so it does not look like it defines anything useful.

⟨Remove `phi_node` from `block` 47a⟩ ≡

```
{
    Unsigned_Int pred_count = block->pred_count;
    Unsigned_Int i;

    for (i = 0; i < pred_count; i++)
        phi_node->parms[i] = 0;

    phi_node->new_name = 0;
}
```

◇

Macro referenced in scraps 46b, 56b.

We iterate through the operations in the block. Before we renumber the registers, we determine if the operation has been modified by constant folding or algebraic simplification. If so, we overwrite the entire operation. The operation can be eliminated if it is a redundant STORE and the user has requested that we trace values through memory.

⟨Renumber the operations 47b⟩ ≡

```
{
    Inst *inst;
    Block_ForAllInsts(inst, block)
    {
        Operation **oper_ptr;
        Inst_ForAllOperations(oper_ptr, inst)
        {
            Operation *oper = *oper_ptr;
            Unsigned_Int details = opcode_specs[oper->opcode].details;

            if (valnum_level >= VALNUM_SCC) oper->critical = TRUE;
            if ((fold_constants || simplify) && (details & EXPR) &&
                VectorSet_Member(constants, oper->arguments[oper->referenced]))
                ⟨Change oper to a load immediate 48a⟩
            else if (trace_memory && (details & STORE))
            {
                Unsigned_Int ref_tag = oper->arguments[5];
                Unsigned_Int def_tag = *Operation_Second_List_Start(oper);
                if (value_num[ref_tag] == value_num[def_tag])
                    oper->critical = FALSE;
                else
                    ⟨Renumber the registers in oper 48b⟩
            }
            else
            {
                ⟨Renumber the registers in oper 48b⟩
                ⟨Decide if oper computes an identity 49⟩
            }
        }
    }
}
```

◇

Macro referenced in scrap 46a.

If we determine that the operation defines a constant value, we overwrite it with a load immediate. We accomplish this by manipulating the fields of `oper`.

⟨Change `oper` to a load immediate 48a⟩ ≡

```
{
    Unsigned_Int dest_reg = oper->arguments[oper->referenced];
    Unsigned_Int def_type = Operation_Def_Type(oper, dest_reg);

    switch (def_type)
    {
        case INT_DEF:        oper->opcode = iLDI;    break;
        case FLOAT_DEF:      oper->opcode = fLDI;    break;
        case DOUBLE_DEF:     oper->opcode = dLDI;    break;
        case COMPLEX_DEF:    oper->opcode = cLDI;    break;
        case DCOMPLEX_DEF:   oper->opcode = qLDI;    break;
        default:
            fprintf(stderr, "Internal error: Unknown type - %d.\n",
                    def_type);
            ABORT;
    }
    oper->arguments[0] = values[dest_reg];
    oper->arguments[1] = value_num[dest_reg];
    oper->constants = 1;
    oper->referenced = 1;
    oper->defined = 2;
}
```

◇

Macro referenced in scrap 47b.

We renumber the used and defined registers of the operation by overwriting them. We often use the **diff** utility for debugging and comparing various forms of value numbering. In order to make using the **diff** program easier, we reorder commutative operations so that the smaller operand appears first.

⟨Renumber the registers in `oper` 48b⟩ ≡

```
{
    Unsigned_Int details = opcode_specs[oper->opcode].details;
    Unsigned_Int2 *item_ptr;
    Operation_ForAllUses(item_ptr, oper)
        *item_ptr = value_num[*item_ptr];
    Operation_ForAllDefs(item_ptr, oper)
        *item_ptr = value_num[*item_ptr];

    if ((details & COMMUTE) &&
        oper->arguments[oper->referenced-1] < oper->arguments[oper->referenced-2])
    {
        Unsigned_Int temp = oper->arguments[oper->referenced-2];
        oper->arguments[oper->referenced-2] = oper->arguments[oper->referenced-1];
        oper->arguments[oper->referenced-1] = temp;
    }
}
```

◇

Macro referenced in scrap 47b.

We decide if an operation computes an identity by checking for the same register used as one of the operands and the result. After renumbering, the register names represent value numbers; all occurrences of the same name must be equal. For example, if we encounter `iADD ⟨r4⟩ ⟨r5⟩ => ⟨r5⟩`, value numbering must have determined that the result is equal to the second operand based on an algebraic identity; in this case, ⟨`r4`⟩ must be zero. These operations need to be removed so that **partial** and value-driven code motion do not think the defined register is an expression.

⟨Decide if `oper` computes an identity 49⟩ ≡

```
if (details & EXPR)
{
    Unsigned_Int dest_reg = oper->arguments[oper->referenced];
    Unsigned_Int2 *reg_ptr;
    Operation_ForAllUses(reg_ptr, oper)
        if (*reg_ptr == dest_reg)
        {
            oper->critical = FALSE;
            break;
        }
}
```

◇

Macro referenced in scrap 47b.

Chapter 4

AVAIL-Based Removal

Chapters 2 and 3 explain how to assign value numbers and how to renumber the registers and ϕ-nodes so that equal values are given the same number. However, these two steps alone will not improve the running time of the routine; we must also remove redundant computations. This chapter describes how to remove computations that are in the set of available expressions (AVAIL) at the point where they appear in the routine [1]. This approach uses data-flow analysis to determine the set of expressions available along all paths from the start of the routine.

To perform AVAIL-based removal, we must compute the set of available expressions at the beginning of each block. An expression is available if it is computed along all paths from `start_block`. If an operation computes a value already in the set of available expressions then it can be removed. First, we compute the AVAIL set for each block, then we remove any operations whose result is in the set.

Because of the properties of SSA form, the data-flow problem solved is slightly different from the traditional AVAIL problem. We do not consider the killed set for a block because no SSA name is overwritten. Consider the following code fragment:

$$\begin{array}{l} Z \leftarrow X + Y \\ X \leftarrow \ldots \\ Z \leftarrow X + Y \end{array}$$

Under the traditional data-flow framework, the assignment to X would kill the Z expression. However, since value numbering has proven that the two assignments to Z are equal, the second one is redundant and can be removed. One way this can happen is if the assignment to X is equal to the definition of X that reaches the first assignment to Z. Note that if the assignment to X caused the two assignments to Z to have different values, then they would not be equal to each other, and they would be assigned different value numbers. The data-flow equations we use are shown in Figure 4.1.

$$\begin{array}{rcl} \text{AVAIL\_in}_i & = & \begin{cases} \emptyset, & \text{if } i = \texttt{start\_block} \\ \displaystyle\bigcap_{j \in pred(i)} \text{AVAIL\_out}_j, & \text{otherwise} \end{cases} \\ \text{AVAIL\_out}_i & = & \text{AVAIL\_in}_i \cup \texttt{defined}_i \end{array}$$

Figure 4.1: Data-flow equations for AVAIL

⟨Optionally perform AVAIL-based removal 51a⟩ ≡
```
if (do_avail)
{
    Arena avail_arena = Arena_Create();

    if (debug >= MAJOR_PHASES)
        fprintf(stderr, "Perform AVAIL-based removal\n");

    ⟨Compute the AVAIL set for each block 51c⟩
    ⟨Remove operations whose result is in AVAIL 56a⟩

    Arena_Destroy(avail_arena);
    sprintf(time_message, "%s: AVAIL-based removal required %%1.2f seconds\n",
            file_name ? file_name : "stdin");
    Time_Dump(time, time_message);
}
◇
```
Macro referenced in scrap 5.

We will store information for each block in the `block_extension` field contained in the `Block` structure.

⟨Type Declarations 51b⟩ ≡
```
struct block_extension
{
    VectorSet defined;    /*  The set of values defined inside this block  */
    VectorSet AVAIL_in;   /*  The set of available expressions at the beginning
                              of this block  */
    VectorSet AVAIL_out;  /*  The set of available expressions at the end
                              of this block  */
    Boolean dirty;        /*  Indicates if AVAIL_in should be recomputed */
};
◇
```
Macro defined by scraps 34a, 51b, 58, 59ab, 60abcd, 61ab.
Macro referenced in scrap 4a.

4.1 Finding Available Expressions

We compute the AVAIL sets using iterative data-flow analysis. The `defined` set can be determined locally for each block. The analysis starts with an initial approximation to the solution and iteratively refines the solution until it stabilizes. All the bit vectors will have `SSA_def_count` elements.

⟨Compute the AVAIL set for each block 51c⟩ ≡
```
{
    Unsigned_Int set_size = SSA_def_count;

    if (debug >= MINOR_PHASES)
        fprintf(stderr, "    Compute the AVAIL set for each block\n");

    ⟨Find the defined set for each block 52a⟩
    ⟨Initialize the AVAIL data-flow problem 53a⟩
    ⟨Find the AVAIL set for all blocks 54a⟩
}
◇
```
Macro referenced in scrap 51a.

The **defined** set will contain all the registers defined in a block.

⟨Find the `defined` set for each block 52a⟩ ≡

```
{
    Block *block;
    ForAllBlocks(block)
    {
        Block_Extension *extension = block->block_extension =
            Arena_GetMem(avail_arena, sizeof(Block_Extension));
        VectorSet defined = extension->defined =
            VectorSet_Create(avail_arena, set_size);
        ⟨Add the φ-nodes to defined 52b⟩
        ⟨Add the operations to defined 52c⟩
    }
}
◇
```

Macro referenced in scrap 51c.

AVAIL-based removal is only concerned with registers, so we ignore any ϕ-nodes that represents a tag.

⟨Add the ϕ-nodes to `defined` 52b⟩ ≡

```
{
    Phi_Node *phi_node;
    Block_ForAllPhiNodes(phi_node, block)
        if (phi_node->old_name < register_count)
            VectorSet_Insert(defined, phi_node->new_name);
}
◇
```

Macro referenced in scrap 52a.

⟨Add the operations to `defined` 52c⟩ ≡

```
{
    Inst *inst;
    Block_ForAllInsts(inst, block)
    {
        Operation **oper_ptr;
        Inst_ForAllOperations(oper_ptr, inst)
        {
            Operation *oper = *oper_ptr;
            Unsigned_Int2 *reg_ptr;

            if (oper->critical)
                Operation_ForAllDefs(reg_ptr, oper)
                    VectorSet_Insert(defined, *reg_ptr);
        }
    }
}
◇
```

Macro referenced in scrap 52a.

⟨Initialize the AVAIL data-flow problem 53a⟩ ≡

```
{
    Block *block;
    ForAllBlocks(block)
        if (block == start_block)
            ⟨Initialize AVAIL for the start_block 53b⟩
        else
            ⟨Initialize AVAIL for the other blocks 53c⟩
}
◇
```

Macro referenced in scrap 51c.

The initial value of AVAIL for the `start_block` is the empty set. In fact, we do not even need to allocate a set for it. We know that the final value of AVAIL for the `start_block` will be the empty set, so we set its `dirty` flag to `FALSE`.

⟨Initialize AVAIL for the `start_block` 53b⟩ ≡

```
{
    Block_Extension *extension = start_block->block_extension;

    extension->AVAIL_out = VectorSet_Create(avail_arena, set_size);
    extension->AVAIL_in = NULL;
    extension->dirty = FALSE;
}
◇
```

Macro referenced in scrap 53a.

The initial value of AVAIL for blocks other than the `start_block` is the set of all registers. Since this is merely an initial guess for the AVAIL set, we set the `dirty` flag to `TRUE`.

⟨Initialize AVAIL for the other blocks 53c⟩ ≡

```
{
    Block_Extension *extension = block->block_extension;
    VectorSet AVAIL_in = extension->AVAIL_in =
        VectorSet_Create(avail_arena, set_size);
    VectorSet AVAIL_out = extension->AVAIL_out =
        VectorSet_Create(avail_arena, set_size);

    VectorSet_Complement(AVAIL_in, AVAIL_in);
    VectorSet_Complement(AVAIL_out, AVAIL_out);
    extension->dirty = TRUE;
}
◇
```

Macro referenced in scrap 53a.

We perform iterative data-flow analysis to find AVAIL for all blocks. We step through the blocks in reverse postorder because it makes the analysis terminate rapidly [12]. Notice that we only compute the information for a block if it is marked `dirty`. The algorithm terminates when an iteration completes with no changes to the AVAIL sets.

⟨Find the AVAIL set for all blocks 54a⟩ ≡
```
{
    Boolean changed;
    VectorSet temp = VectorSet_Create(avail_arena, set_size);

    do
    {
        Block *block;

        changed = FALSE;
        ForAllBlocks_rPostorder(block)
        {
            Block_Extension *extension = block->block_extension;

            if (extension->dirty)
            {
                ⟨Start with AVAIL_out from the first predecessor 54b⟩
                ⟨Look at AVAIL_out for the other predecessors 55a⟩
                ⟨If temp ≠ AVAIL_in, set changed and recompute AVAIL_out 55b⟩
            }
        }
    } while (changed);
}
◇
```
Macro referenced in scrap 51c.

The predecessors of a block are stored in a linked list pointed to by the `pred` field of the `Block` structure. We will use the `AVAIL_out` set from the first predecessor as our initial value for the `AVAIL_in` set. The set will be computed in `temp` so we can see if we have changed anything.

⟨Start with `AVAIL_out` from the first predecessor 54b⟩ ≡
```
{
    Edge *edge = block->pred;
    extension->dirty = FALSE;
    VectorSet_Copy(temp, edge->pred->block_extension->AVAIL_out);
}
◇
```
Macro referenced in scrap 54a.

After we have looked at the **AVAIL_out** set from the first predecessor, we iterate through the other predecessors by starting with **block->pred->next_pred** (the second predecessor) and moving through the **next_pred** pointers until we reach a **NULL** pointer. For each of these predecessors, we update the intersection of the **AVAIL_out** sets. This intersection will be the **AVAIL_in** set for the block.

⟨Look at AVAIL_out for the other predecessors 55a⟩ ≡

```
{
    Edge *edge;
    for (edge = block->pred->next_pred; edge; edge = edge->next_pred)
    {
        Block_Extension *extension = edge->pred->block_extension;
        VectorSet_Intersect(temp, temp, extension->AVAIL_out);
    }
}
◇
```

Macro referenced in scrap 54a.

If the **AVAIL_in** set for this block has changed, we must recompute the **AVAIL_out** set and mark all the successor blocks **dirty**.

⟨If temp $\neq$ AVAIL_in, set changed and recompute AVAIL_out 55b⟩ ≡

```
if (!VectorSet_Equal(extension->AVAIL_in, temp))
{
    changed = TRUE;
    VectorSet_Copy(extension->AVAIL_in, temp);
    VectorSet_Union(extension->AVAIL_out,
                    extension->AVAIL_in, extension->defined);
    ⟨Mark all the successors dirty 55c⟩
}
◇
```

Macro referenced in scrap 54a.

⟨Mark all the successors dirty 55c⟩ ≡

```
{
    Edge *edge;
    Block_ForAllSuccs(edge, block)
        edge->succ->block_extension->dirty = TRUE;
}
◇
```

Macro referenced in scrap 55b.

4.2 Removing Operations

Once we have computed AVAIL for each block, we are ready to start removing operations from the routine. We step through the blocks and use the **AVAIL_in** set as a guide for removing operations.

⟨Remove operations whose result is in AVAIL 56a⟩ ≡
```
{
    Block *block;

    if (debug >= MINOR_PHASES)
        fprintf(stderr, "    Remove operations whose result is in AVAIL\n");

    ForAllBlocks(block)
    {
        VectorSet AVAIL_in = block->block_extension->AVAIL_in;
        ⟨Remove any φ-nodes in AVAIL 56b⟩
        ⟨Remove any operations in AVAIL 56c⟩
    }
}
◇
```
Macro referenced in scrap 51a.

AVAIL-based removal is only concerned with registers, so we ignore any ϕ-nodes that represents a tag.

⟨Remove any ϕ-nodes in AVAIL 56b⟩ ≡
```
{
    Phi_Node *phi_node;
    Block_ForAllPhiNodes(phi_node, block)
        if (phi_node->old_name < register_count)
            if (VectorSet_Member(AVAIL_in, phi_node->new_name))
                ⟨Remove phi_node from block 47a⟩
            else
                VectorSet_Insert(AVAIL_in, phi_node->new_name);
}
◇
```
Macro referenced in scrap 56a.

⟨Remove any operations in AVAIL 56c⟩ ≡
```
{
    Inst *inst;
    Block_ForAllInsts(inst, block)
    {
        Operation **oper_ptr;
        Inst_ForAllOperations(oper_ptr, inst)
        {
            Operation *oper = *oper_ptr;
            ⟨If oper is in AVAIL, remove it 57a⟩
        }
    }
}
◇
```
Macro referenced in scrap 56a.

We can remove any **LOAD** or **EXPR** operation whose result register is in the **AVAIL_in** set. If the operation is not removed, we add its defined registers to the **AVAIL_in** set because they are available to operations later in the block. Notice that we can safely add these before we finish processing all operations in the instruction[1]. This is because if two operations in the same instruction compute the same value, we can eliminate either one of them. We choose to eliminate the second.

⟨If `oper` is in AVAIL, remove it 57a⟩ ≡

```
{
    Unsigned_Int details = opcode_specs[oper->opcode].details;
    Unsigned_Int reg = oper->arguments[oper->referenced];

    if ((details & (LOAD | EXPR)) && VectorSet_Member(AVAIL_in, reg))
        oper->critical = FALSE;
    else
        ⟨Add all the defined registers to AVAIL_in 57b⟩
}
◇
```

Macro referenced in scrap 56c.

⟨Add all the defined registers to `AVAIL_in` 57b⟩ ≡

```
{
    Unsigned_Int2 *reg_ptr;
    Operation_ForAllDefs(reg_ptr, oper)
        VectorSet_Insert(AVAIL_in, *reg_ptr);
}
◇
```

Macro referenced in scrap 57a.

[1] All operations in an `ILOC` instruction are considered to execute in parallel.

Appendix A

Maintaining the Hash Tables

We maintain our hash tables using "overflow chaining". The optimistic and valid tables used by SCC-based value numbering are implemented as a single table where each entry has both a valid and an optimistic value number. This saves space by avoiding the redundancy that exists between the tables. A parameter to each routine indicates which value number the caller needs. Further, since we value number during a walk of the dominator tree, we need a *scoped* hash table, similar to the style sometimes used as the symbol table during compilation of a lexically-scoped language. At any point during the processing, the scoped table contains a sequence of nested scopes, one for each block that is an ancestor of the current block.

- As new blocks are processed, new scopes are created in the table. Any entries added to the current scope will supersede entries with the same name in any enclosing scope. Searches are performed starting with the innermost scope and proceeding outward until a matching entry is found.
- As the algorithm returns upward from a block, it must undo the effects of processing that block. Using a scoped table, this corresponds to deleting the block's scope from the table. To facilitate this process, we link together all the entries in the same scope.

Figure A.1 shows the data structure for the hash tables.

A.1 Data Structures

To support value numbering, we need three different hash tables: the expression table, the ϕ-node table, and the tag table. Each table operates the same way, but they hold different types of data. Here are the fields needed in every entry in the expression table. Notice that each entry can have a valid and an optimistic value number.

⟨Type Declarations 58⟩ ≡

```
typedef struct expr_entry
{
    struct expr_entry *next_chain;    /* next entry in this hash bucket */
    struct expr_entry *next_scope;    /* next entry in this scope */
    Unsigned_Int2 hash_value;         /* which hash bucket */
    Unsigned_Int2 valid_value;        /* the valid value number */
    Unsigned_Int2 optimistic_value;   /* the optimistic value number */
    Unsigned_Int2 operand[2];         /* the expression's 2 operands */
    Opcode_Names opcode;              /* the expression's opcode */
} EXPR_ENTRY;
```

◇

Macro defined by scraps 34a, 51b, 58, 59ab, 60abcd, 61ab.
Macro referenced in scrap 4a.

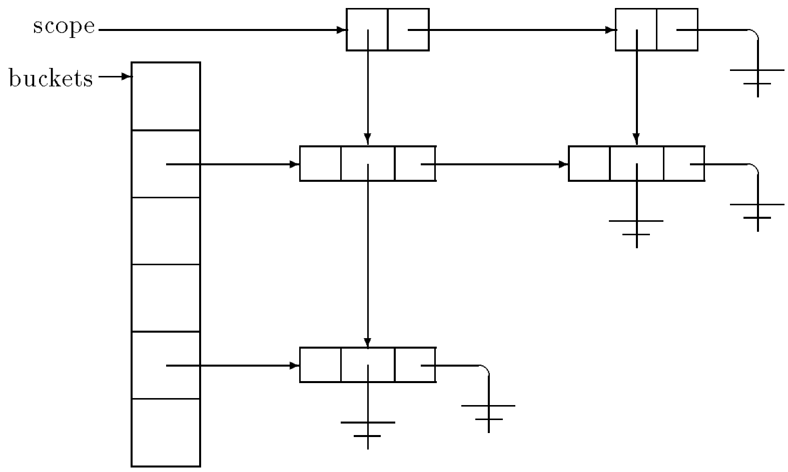


Figure A.1: Data structure for scoped hash table

We implement the nested scopes using a stack in which each item is a linked list of entries in the table.

⟨Type Declarations 59a⟩ ≡
```
typedef struct expr_scope
{
    EXPR_ENTRY *scope;
    struct expr_scope *next;
} EXPR_SCOPE;
```
◇

Macro defined by scraps 34a, 51b, 58, 59ab, 60abcd, 61ab.
Macro referenced in scrap 4a.

The expression table is composed of an array of `buckets` and a stack of nested scopes.

⟨Type Declarations 59b⟩ ≡
```
typedef struct
{
    EXPR_ENTRY **buckets;
    EXPR_SCOPE *scope;
} EXPR_TABLE;
```
◇

Macro defined by scraps 34a, 51b, 58, 59ab, 60abcd, 61ab.
Macro referenced in scrap 4a.

We also maintain a hash table containing ϕ-nodes. It operates in the same way that the expression table does. Each ϕ-node is identified by the `preorder_index` of the block in which it is contained and the value numbers of its parameters. Notice that the `parms` array is declared with size 1. Because it is the last field in the structure, we can change its size when we allocate the structure.

⟨Type Declarations 60a⟩ ≡

```
typedef struct phi_entry
{
    struct phi_entry *next_chain;    /* next entry in this hash bucket */
    struct phi_entry *next_scope;    /* next entry in this scope */
    Unsigned_Int2 hash_value;        /* which hash bucket */
    Unsigned_Int2 preorder_index;    /* indicates block containing phi node */
    Unsigned_Int2 valid_value;       /* the valid value number */
    Unsigned_Int2 optimistic_value;  /* the optimistic value number */
    Unsigned_Int2 parms[1];          /* the phi node's parameters */
} PHI_ENTRY;
```

◇

Macro defined by scraps 34a, 51b, 58, 59ab, 60abcd, 61ab.
Macro referenced in scrap 4a.

⟨Type Declarations 60b⟩ ≡

```
typedef struct phi_scope
{
    PHI_ENTRY *scope;
    struct phi_scope *next;
} PHI_SCOPE;
```

◇

Macro defined by scraps 34a, 51b, 58, 59ab, 60abcd, 61ab.
Macro referenced in scrap 4a.

⟨Type Declarations 60c⟩ ≡

```
typedef struct
{
    PHI_ENTRY **buckets;
    PHI_SCOPE *scope;
} PHI_TABLE;
```

◇

Macro defined by scraps 34a, 51b, 58, 59ab, 60abcd, 61ab.
Macro referenced in scrap 4a.

Finally, we maintain a table of tags used to trace values through memory (if requested). Each entry has two operands (representing the address of the memory location) and a tag value (representing the SSA name for this memory location).

⟨Type Declarations 60d⟩ ≡

```
typedef struct tag_entry
{
    struct tag_entry *next_chain;    /* next entry in this hash bucket */
    struct tag_entry *next_scope;    /* next entry in this scope */
    Unsigned_Int2 hash_value;        /* which hash bucket */
    Unsigned_Int2 valid_value;       /* the valid value number */
    Unsigned_Int2 optimistic_value;  /* the optimistic value number */
    Unsigned_Int2 operand[2];        /* the operands of the memory operation */
    Unsigned_Int2 tag;               /* the SSA name of the tag */
    Opcode_Names opcode;             /* the memory operation's opcode */
} TAG_ENTRY;
```

◇

Macro defined by scraps 34a, 51b, 58, 59ab, 60abcd, 61ab.
Macro referenced in scrap 4a.

⟨Type Declarations 61a⟩ ≡
```
typedef struct tag_scope
{
    TAG_ENTRY *scope;
    struct tag_scope *next;
} TAG_SCOPE;
```
◇

Macro defined by scraps 34a, 51b, 58, 59ab, 60abcd, 61ab.
Macro referenced in scrap 4a.

⟨Type Declarations 61b⟩ ≡
```
typedef struct
{
    TAG_ENTRY **buckets;
    TAG_SCOPE *scope;
} TAG_TABLE;
```
◇

Macro defined by scraps 34a, 51b, 58, 59ab, 60abcd, 61ab.
Macro referenced in scrap 4a.

We require some global variables.

⟨Global Variables 61c⟩ ≡
```
static Arena hash_arena;
static Unsigned_Int hash_size;
static EXPR_TABLE expressions;
static PHI_TABLE phi_nodes;
static TAG_TABLE tags;
static Unsigned_Int num_expressions = 0;
static Unsigned_Int num_phi_nodes = 0;
static Unsigned_Int num_tags = 0;
```
◇

Macro defined by scraps 4b, 6b, 7d, 12a, 13b, 17b, 33a, 34b, 44a, 61c.
Macro referenced in scrap 4a.

As we go along, we will count the number of entries in each table. This scrap will report this information.

⟨Optionally report the number of hash table entries 61d⟩ ≡
```
if (time_print)
{
    fprintf(stderr,"%s: Value numbering required %d hash table entries\n",
            file_name ? file_name : "stdin",
            num_expressions + num_phi_nodes + num_tags);
    fprintf(stderr, "%s: Value numbering tables used %d bytes\n",
            file_name ? file_name : "stdin",
            num_expressions*sizeof(EXPR_ENTRY) +
            num_phi_nodes*sizeof(PHI_ENTRY) +
            num_tags*sizeof(TAG_ENTRY));
}
```
◇

Macro referenced in scrap 5.

We set `hash_size` to be the first Mersenne "prime" greater than `SSA_def_count`, where `SSA_def_count` serves as a nice approximation to the number of expressions in the routine.

⟨Initialize the hash tables 62a⟩ ≡
```
for (hash_size = 4; hash_size < SSA_def_count; hash_size *= 2)
    /* do nothing */;
hash_size--;
```
◇

Macro defined by scraps 62ab.
Macro referenced in scrap 11.

Once we have determined the size of the tables, we are ready to allocate memory for them.

⟨Initialize the hash tables 62b⟩ ≡
```
hash_arena = Arena_Create();
expressions.buckets = Arena_GetMemClear(hash_arena, hash_size*sizeof(EXPR_ENTRY *));
expressions.scope = NULL;
phi_nodes.buckets = Arena_GetMemClear(hash_arena, hash_size*sizeof(PHI_ENTRY *));
phi_nodes.scope = NULL;
tags.buckets = Arena_GetMemClear(hash_arena, hash_size*sizeof(TAG_ENTRY *));
tags.scope = NULL;
```
◇

Macro defined by scraps 62ab.
Macro referenced in scrap 11.

A.2 Scoping

When we begin value numbering a new basic block, we create a new scope in each hash table. Notice that we use `Arena_Mark` so that we can reclaim the memory when the scope is removed.

⟨Mark the beginning of a new scope 62c⟩ ≡
```
{
    Arena_Mark(hash_arena);
    ⟨Create a new scope in expressions 62d⟩
    ⟨Create a new scope in phi_nodes 63a⟩
    ⟨Create a new scope in tags 63b⟩
}
```
◇

Macro referenced in scraps 29c, 33b.

We create a new scope by pushing a node onto the `scope` stack.

⟨Create a new scope in `expressions` 62d⟩ ≡
```
{
    EXPR_SCOPE *expr_scope = Arena_GetMem(hash_arena, sizeof(EXPR_SCOPE));
    expr_scope->scope = NULL;
    expr_scope->next = expressions.scope;
    expressions.scope = expr_scope;
}
```
◇

Macro referenced in scrap 62c.

⟨Create a new scope in phi_nodes 63a⟩ ≡

```
{
    PHI_SCOPE *phi_scope = Arena_GetMem(hash_arena, sizeof(PHI_SCOPE));
    phi_scope->scope = NULL;
    phi_scope->next = phi_nodes.scope;
    phi_nodes.scope = phi_scope;
}
◇
```

Macro referenced in scrap 62c.

⟨Create a new scope in tags 63b⟩ ≡

```
{
    TAG_SCOPE *tag_scope = Arena_GetMem(hash_arena, sizeof(TAG_SCOPE));
    tag_scope->scope = NULL;
    tag_scope->next = tags.scope;
    tags.scope = tag_scope;
}
◇
```

Macro referenced in scrap 62c.

When we are done with a block (and all its descendents), we need to remove all the block's entries from the hash tables and free up all their storage.

⟨Clean up after leaving a scope 63c⟩ ≡

```
{
    ⟨Clean up expressions after leaving a scope 63d⟩
    ⟨Clean up phi_nodes after leaving a scope 64a⟩
    ⟨Clean up tags after leaving a scope 64b⟩
    Arena_Release(hash_arena);
}
◇
```

Macro referenced in scrap 29c.

To clean up a table, we remove the entries in the list associated with this scope.

⟨Clean up expressions after leaving a scope 63d⟩ ≡

```
{
    EXPR_ENTRY *expr_scope = expressions.scope->scope;
    expressions.scope = expressions.scope->next;
    while (expr_scope)
    {
        Unsigned_Int i = expr_scope->hash_value;
        expressions.buckets[i] = expr_scope->next_chain;
        expr_scope = expr_scope->next_scope;
    }
}
◇
```

Macro referenced in scrap 63c.

⟨Clean up phi_nodes after leaving a scope 64a⟩ ≡

```
{
    PHI_ENTRY *phi_scope = phi_nodes.scope->scope;
    phi_nodes.scope = phi_nodes.scope->next;
    while (phi_scope)
    {
        Unsigned_Int i = phi_scope->hash_value;
        phi_nodes.buckets[i] = phi_scope->next_chain;
        phi_scope = phi_scope->next_scope;
    }
}
◇
```

Macro referenced in scrap 63c.

⟨Clean up tags after leaving a scope 64b⟩ ≡

```
{
    TAG_ENTRY *tag_scope = tags.scope->scope;
    tags.scope = tags.scope->next;
    while (tag_scope)
    {
        Unsigned_Int i = tag_scope->hash_value;
        tags.buckets[i] = tag_scope->next_chain;
        tag_scope = tag_scope->next_scope;
    }
}
◇
```

Macro referenced in scrap 63c.

A.3 Searching

There are two basic entry points for accessing each table. For the expression table, they are **expr_search** and **expr_add**. The **expr_search** routine takes an expression and looks it up in the hash table. If found, it returns the value number of the expression; otherwise, it returns `-1`.

To find an expression in the table, we hash the expression to an integer index. Then we look in the **buckets** array at that index and walk down the chain, checking for the correct expression. Since our hash table is larger than the number of possible expressions, we expect that the length of a chain will be quite short (on average, < 1). The **which_table** parameter tells which value number (valid or optimistic) the caller wants.

⟨Prototypes 64c⟩ ≡

```
static Int expr_search(Unsigned_Int which_table, Opcode_Names opcode,
                       Unsigned_Int operand0, Unsigned_Int operand1);
◇
```

Macro defined by scraps 8d, 9b, 13d, 14b, 16b, 17c, 19b, 24a, 29b, 34e, 36b, 38, 64c, 65b, 66b, 67b, 69a, 70b, 72bd, 73b, 75b, 76c, 84a.
Macro referenced in scrap 4a.

⟨Functions 65a⟩ ≡

```
static Int expr_search(Unsigned_Int which_table, Opcode_Names opcode,
                       Unsigned_Int operand0, Unsigned_Int operand1)
{
    Unsigned_Int index = expr_hash(opcode, operand0, operand1);
    EXPR_ENTRY *chain = expressions.buckets[index];

    while (chain)
    {
        if (chain->opcode == opcode &&
            chain->operand[0] == operand0 &&
            chain->operand[1] == operand1)
            return which_table == VALID ?
                       chain->valid_value :
                       chain->optimistic_value;
        chain = chain->next_chain;
    }
    return -1;
}
```

◇

Macro defined by scraps 9a, 14a, 15, 16c, 18a, 19c, 24b, 29c, 35a, 36c, 39a, 65a, 66a, 67a, 68a, 69b, 71, 72c, 73ac, 75c, 77a, 84b.
Macro referenced in scrap 4a.

The **phi_search** routine takes a ϕ-node and looks it up in the hash table. If found, it returns the value number of the ϕ-node; otherwise, it returns `-1`. The `which_table` parameter tells which value number (valid or optimistic) the caller wants.

⟨Prototypes 65b⟩ ≡

```
static Int phi_search(Unsigned_Int which_table, Phi_Node *phi_node, Block *block);
```

◇

Macro defined by scraps 8d, 9b, 13d, 14b, 16b, 17c, 19b, 24a, 29b, 34e, 36b, 38, 64c, 65b, 66b, 67b, 69a, 70b, 72bd, 73b, 75b, 76c, 84a.
Macro referenced in scrap 4a.

⟨Functions 66a⟩ ≡

```
static Int phi_search(Unsigned_Int which_table, Phi_Node *phi_node, Block *block)
{
    Unsigned_Int pred_count = block->pred_count;
    Unsigned_Int index = phi_hash(phi_node, pred_count);
    PHI_ENTRY *chain = phi_nodes.buckets[index];

    while (chain)
    {
        if (chain->preorder_index == block->preorder_index)
        {
            Unsigned_Int2 *test_parms = chain->parms;
            Boolean equal = TRUE;
            Unsigned_Int i;
            for (i = 0; i < pred_count; i++)
                if (value_num[phi_node->parms[i]] != test_parms[i])
                {
                    equal = FALSE;
                    break;
                }

            if (equal)
                return which_table == VALID ?
                           chain->valid_value :
                           chain->optimistic_value;
        }

        chain = chain->next_chain;
    }
    return -1;
}
◇
```

Macro defined by scraps 9a, 14a, 15, 16c, 18a, 19c, 24b, 29c, 35a, 36c, 39a, 65a, 66a, 67a, 68a, 69b, 71, 72c, 73ac, 75c, 77a, 84b.
Macro referenced in scrap 4a.

The `tag_search` routine takes a memory operation and looks it up in the hash table. If found, it returns the value number of the operation; otherwise, it returns `-1`. The `which_table` parameter tells which value number (valid or optimistic) the caller wants.

⟨Prototypes 66b⟩ ≡

```
static Int tag_search(Unsigned_Int which_table, Opcode_Names opcode,
                      Unsigned_Int operand0, Unsigned_Int operand1,
                      Unsigned_Int tag);
◇
```

Macro defined by scraps 8d, 9b, 13d, 14b, 16b, 17c, 19b, 24a, 29b, 34e, 36b, 38, 64c, 65b, 66b, 67b, 69a, 70b, 72bd, 73b, 75b, 76c, 84a.
Macro referenced in scrap 4a.

⟨Functions 67a⟩ ≡

```
static Int tag_search(Unsigned_Int which_table, Opcode_Names opcode,
                      Unsigned_Int operand0, Unsigned_Int operand1,
                      Unsigned_Int tag)
{
    Unsigned_Int index = tag_hash(opcode, operand0, operand1, tag);
    TAG_ENTRY *chain = tags.buckets[index];

    while (chain)
    {
        if (chain->opcode == opcode &&
            chain->operand[0] == operand0 && chain->operand[1] == operand1 &&
            chain->tag == tag)
            return which_table == VALID ?
                       chain->valid_value :
                       chain->optimistic_value;
        chain = chain->next_chain;
    }
    return -1;
}
```

◇

Macro defined by scraps 9a, 14a, 15, 16c, 18a, 19c, 24b, 29c, 35a, 36c, 39a, 65a, 66a, 67a, 68a, 69b, 71, 72c, 73ac, 75c, 77a, 84b.
Macro referenced in scrap 4a.

A.4 Adding

The **expr_add** routine takes an expression and its value number and adds them to the **expressions** table. The **which_name** indicates which value number the caller is adding.

Before creating a new entry in the table, we first check for an existing entry for the same expression. Recall that we are implementing both the optimistic and valid tables in a single table, so the expression may have been inserted in the other "table". If an entry for this expression is found, we store the name there. Otherwise, we create a new entry for the expression. Notice that we set the other value number to **-1** so that calls to **expr_search** will return "not found".

⟨Prototypes 67b⟩ ≡

```
static Void expr_add(Unsigned_Int which_table, Opcode_Names opcode,
                     Unsigned_Int operand0, Unsigned_Int operand1,
                     Unsigned_Int dest_reg);
```

◇

Macro defined by scraps 8d, 9b, 13d, 14b, 16b, 17c, 19b, 24a, 29b, 34e, 36b, 38, 64c, 65b, 66b, 67b, 69a, 70b, 72bd, 73b, 75b, 76c, 84a.
Macro referenced in scrap 4a.

⟨Functions 68a⟩ ≡

```
static Void expr_add(Unsigned_Int which_table, Opcode_Names opcode,
                     Unsigned_Int operand0, Unsigned_Int operand1,
                     Unsigned_Int dest_reg)
{
    Unsigned_Int index = expr_hash(opcode, operand0, operand1);
    EXPR_ENTRY *new_entry;

    ⟨Decide if an entry for the expression already exists 68b⟩

    new_entry = Arena_GetMem(hash_arena, sizeof(EXPR_ENTRY));
    num_expressions++;

    new_entry->hash_value = index;
    if (which_table == VALID)
    {
        new_entry->valid_value = dest_reg;
        new_entry->optimistic_value = -1;
    }
    else
    {
        new_entry->valid_value = -1;
        new_entry->optimistic_value = dest_reg;
    }
    new_entry->opcode = opcode;
    new_entry->operand[0] = operand0;
    new_entry->operand[1] = operand1;
    new_entry->next_chain = expressions.buckets[index];
    new_entry->next_scope = expressions.scope->scope;
    expressions.scope->scope = new_entry;
    expressions.buckets[index] = new_entry;
}
◇
```

Macro defined by scraps 9a, 14a, 15, 16c, 18a, 19c, 24b, 29c, 35a, 36c, 39a, 65a, 66a, 67a, 68a, 69b, 71, 72c, 73ac, 75c, 77a, 84b.
Macro referenced in scrap 4a.

Before adding a new entry to the table, we must decide if one exists. If so, we record the value in that entry and return.

⟨Decide if an entry for the expression already exists 68b⟩ ≡

```
{
    EXPR_ENTRY *chain = expressions.buckets[index];
    while (chain)
    {
        if (chain->opcode == opcode &&
            chain->operand[0] == operand0 &&
            chain->operand[1] == operand1)
        {
            if (which_table == VALID)
                chain->valid_value = dest_reg;
            else
                chain->optimistic_value = dest_reg;
            return;
        }
        chain = chain->next_chain;
    }
}
◇
```

Macro referenced in scrap 68a.

The **phi_add** routine takes a ϕ-node and its value number and adds them to the **phi_nodes** table. The **which_name** indicates which value number the caller is adding.

⟨Prototypes 69a⟩ ≡

```
static Void phi_add(Unsigned_Int which_table, Phi_Node *phi_node, Block *block);
```

◇

Macro defined by scraps 8d, 9b, 13d, 14b, 16b, 17c, 19b, 24a, 29b, 34e, 36b, 38, 64c, 65b, 66b, 67b, 69a, 70b, 72bd, 73b, 75b, 76c, 84a.
Macro referenced in scrap 4a.

⟨Functions 69b⟩ ≡

```
static Void phi_add(Unsigned_Int which_table, Phi_Node *phi_node, Block *block)
{
    Unsigned_Int pred_count = block->pred_count;
    Unsigned_Int index = phi_hash(phi_node, pred_count);
    PHI_ENTRY *new_entry;
    Unsigned_Int i;

    ⟨Decide if an entry for phi_node already exists 70a⟩

    new_entry =
        Arena_GetMem(hash_arena,
                     sizeof(PHI_ENTRY)+(pred_count-1)*sizeof(Unsigned_Int2));
    num_phi_nodes++;
    new_entry->hash_value = index;
    if (which_table == VALID)
    {
        new_entry->valid_value = phi_node->new_name;
        new_entry->optimistic_value = -1;
    }
    else
    {
        new_entry->valid_value = -1;
        new_entry->optimistic_value = phi_node->new_name;
    }
    new_entry->preorder_index = block->preorder_index;
    for (i = 0; i < pred_count; i++)
        new_entry->parms[i] = value_num[phi_node->parms[i]];
    new_entry->next_chain = phi_nodes.buckets[index];
    new_entry->next_scope = phi_nodes.scope->scope;
    phi_nodes.scope->scope = new_entry;
    phi_nodes.buckets[index] = new_entry;
}
```

◇

Macro defined by scraps 9a, 14a, 15, 16c, 18a, 19c, 24b, 29c, 35a, 36c, 39a, 65a, 66a, 67a, 68a, 69b, 71, 72c, 73ac, 75c, 77a, 84b.
Macro referenced in scrap 4a.

Before adding a new entry to the table, we must decide if one exists. If so, we record the value in that entry and return.

⟨Decide if an entry for phi_node already exists 70a⟩ ≡
```
{

    PHI_ENTRY *chain = phi_nodes.buckets[index];
    while (chain)
    {
        if (chain->preorder_index == block->preorder_index)
        {
            Unsigned_Int2 *test_parms = chain->parms;
            Boolean equal = TRUE;
            Unsigned_Int i;
            for (i = 0; i < pred_count; i++)
                if (value_num[phi_node->parms[i]] != test_parms[i])
                {
                    equal = FALSE;
                    break;
                }
            if (equal)
            {
                if (which_table == VALID)
                    chain->valid_value = phi_node->new_name;
                else
                    chain->optimistic_value = phi_node->new_name;
                return;
            }
        }
        chain = chain->next_chain;
    }
}
```
◇

Macro referenced in scrap 69b.

The `tag_add` routine takes a memory operation and its value number and adds them to the `tags` table. The `which_name` indicates which value number the caller is adding.

⟨Prototypes 70b⟩ ≡
```
static Void tag_add(Unsigned_Int which_table, Opcode_Names opcode,
                    Unsigned_Int operand0, Unsigned_Int operand1,
                    Unsigned_Int tag, Unsigned_Int dest_reg);
```
◇

Macro defined by scraps 8d, 9b, 13d, 14b, 16b, 17c, 19b, 24a, 29b, 34e, 36b, 38, 64c, 65b, 66b, 67b, 69a, 70b, 72bd, 73b, 75b, 76c, 84a.
Macro referenced in scrap 4a.

⟨Functions 71⟩ ≡

```
static Void tag_add(Unsigned_Int which_table, Opcode_Names opcode,
                    Unsigned_Int operand0, Unsigned_Int operand1,
                    Unsigned_Int tag, Unsigned_Int dest_reg)
{
    Unsigned_Int index = tag_hash(opcode, operand0, operand1, tag);
    TAG_ENTRY *new_entry;

    ⟨Decide if an entry for this tag already exists 72a⟩

    new_entry = Arena_GetMem(hash_arena, sizeof(TAG_ENTRY));
    num_tags++;
    new_entry->hash_value = index;
    if (which_table == VALID)
    {
        new_entry->valid_value = dest_reg;
        new_entry->optimistic_value = -1;
    }
    else
    {
        new_entry->valid_value = -1;
        new_entry->optimistic_value = dest_reg;
    }
    new_entry->opcode = opcode;
    new_entry->operand[0] = operand0;
    new_entry->operand[1] = operand1;
    new_entry->tag = tag;
    new_entry->next_chain = tags.buckets[index];
    new_entry->next_scope = tags.scope->scope;
    tags.scope->scope = new_entry;
    tags.buckets[index] = new_entry;
}
◇
```

Macro defined by scraps 9a, 14a, 15, 16c, 18a, 19c, 24b, 29c, 35a, 36c, 39a, 65a, 66a, 67a, 68a, 69b, 71, 72c, 73ac, 75c, 77a, 84b.
Macro referenced in scrap 4a.

Before adding a new entry to the table, we must decide if one exists. If so, we record the value in that entry and return.

⟨Decide if an entry for this tag already exists 72a⟩ ≡
```
{
    TAG_ENTRY *chain = tags.buckets[index];
    while (chain)
    {
        if (chain->opcode == opcode &&
            chain->operand[0] == operand0 && chain->operand[1] == operand1 &&
            chain->tag == tag)
        {
            if (which_table == VALID)
                chain->valid_value = dest_reg;
            else
                chain->optimistic_value = dest_reg;
            return;
        }

        chain = chain->next_chain;
    }
}
```
◇

Macro referenced in scrap 71.

A.5 Hash Functions

The **expr_hash** function takes an expression and returns an integer in the range `0 <= i < hash_size`. To perform the hash, we add shifted versions of the parameters together and find the modulo **hash_size**. We shift by 8, since there are approximately 256 possible opcodes.

⟨Prototypes 72b⟩ ≡
```
static Unsigned_Int expr_hash(Opcode_Names opcode,
                              Unsigned_Int operand0, Unsigned_Int operand1);
```
◇

Macro defined by scraps 8d, 9b, 13d, 14b, 16b, 17c, 19b, 24a, 29b, 34e, 36b, 38, 64c, 65b, 66b, 67b, 69a, 70b, 72bd, 73b, 75b, 76c, 84a.
Macro referenced in scrap 4a.

⟨Functions 72c⟩ ≡
```
static Unsigned_Int expr_hash(Opcode_Names opcode,
                              Unsigned_Int operand0, Unsigned_Int operand1)
{
    Unsigned_Int sum = opcode + (operand0 << 8) + (operand1 << 16);
    return sum % hash_size;
}
```
◇

Macro defined by scraps 9a, 14a, 15, 16c, 18a, 19c, 24b, 29c, 35a, 36c, 39a, 65a, 66a, 67a, 68a, 69b, 71, 72c, 73ac, 75c, 77a, 84b.
Macro referenced in scrap 4a.

The **phi_hash** function takes a ϕ-node and returns an integer in the range `0 <= i < hash_size`. To perform the hash, we add the value numbers of the parameters of the ϕ-node and find the modulo **hash_size**.

⟨Prototypes 72d⟩ ≡
```
static Unsigned_Int phi_hash(Phi_Node *phi_node, Unsigned_Int pred_count);
```
◇

Macro defined by scraps 8d, 9b, 13d, 14b, 16b, 17c, 19b, 24a, 29b, 34e, 36b, 38, 64c, 65b, 66b, 67b, 69a, 70b, 72bd, 73b, 75b, 76c, 84a.
Macro referenced in scrap 4a.

⟨Functions 73a⟩ ≡

```
static Unsigned_Int phi_hash(Phi_Node *phi_node, Unsigned_Int pred_count)
{
    Unsigned_Int sum = 0;
    Unsigned_Int i;
    for (i = 0; i < pred_count; i++)
        sum += value_num[phi_node->parms[i]];

    return sum % hash_size;
}
◇
```

Macro defined by scraps 9a, 14a, 15, 16c, 18a, 19c, 24b, 29c, 35a, 36c, 39a, 65a, 66a, 67a, 68a, 69b, 71, 72c, 73ac, 75c, 77a, 84b.
Macro referenced in scrap 4a.

The `tag_hash` function takes a memory operation and returns an integer in the range `0 <= i < hash_size`. To perform the hash, we add shifted versions of the operands and find the modulo `hash_size`.

⟨Prototypes 73b⟩ ≡

```
static Unsigned_Int tag_hash(Opcode_Names opcode,
                             Unsigned_Int operand0, Unsigned_Int operand1,
                             Unsigned_Int tag);
◇
```

Macro defined by scraps 8d, 9b, 13d, 14b, 16b, 17c, 19b, 24a, 29b, 34e, 36b, 38, 64c, 65b, 66b, 67b, 69a, 70b, 72bd, 73b, 75b, 76c, 84a.
Macro referenced in scrap 4a.

⟨Functions 73c⟩ ≡

```
static Unsigned_Int tag_hash(Opcode_Names opcode,
                             Unsigned_Int operand0, Unsigned_Int operand1,
                             Unsigned_Int tag)
{
    Unsigned_Int sum = opcode + (operand0 << 8) + (operand1 << 16) + (tag << 24);
    return sum % hash_size;
}
◇
```

Macro defined by scraps 9a, 14a, 15, 16c, 18a, 19c, 24b, 29c, 35a, 36c, 39a, 65a, 66a, 67a, 68a, 69b, 71, 72c, 73ac, 75c, 77a, 84b.
Macro referenced in scrap 4a.

Appendix B

Algebraic Simplification

This appendix describes how we perform algebraic simplification. During value numbering, we will try to simplify each expression after we look up the value numbers of the operands and before we search the hash table. If the expression can be simplified, we will search for the new expression.

There are three main ways that an expression can be simplified: (1) if the operands are equal, (2) if both operands are constant, and (3) if one of the operands is constant. Typically, a simplification involves replacing the expression either with a constant (*e.g.*, $x - x = 0$) or with a copy from one of the operands (*e.g.*, $x + 0 = x$).

⟨Optionally try to simplify the expression 74a⟩ ≡

```
{
    if (simplify)
        ⟨Check for an expression whose operands are equal 74b⟩
    if (fold_constants)
        ⟨Check for an expression whose operands are constants 81a⟩
    if (simplify)
        ⟨Check for an expression with one constant operand 78⟩
}
◇
```

Macro referenced in scrap 17a.

Certain operations can be simplified if we know that the operands are equal. For example, we know the result of $x - x$ is always zero. Note that we must *not* fold x/x because of the possibility that $x = 0$. A similar argument applies to the remainder operations.

⟨Check for an expression whose operands are equal 74b⟩ ≡

```
if (value0 == value1)
    switch (opcode_name)
    {
        ⟨Check for an expression that behaves as a copy if the operands are equal 75a⟩
        ⟨Check for an expression that tests equality of equal values 76a⟩
        ⟨Check for an expression that tests inequality of equal values 76b⟩
        default:
            break;
    }
◇
```

Macro referenced in scrap 74a.

Certain operations behave as a copy when their operands are equal. For example, $a \wedge a$ is always a. We turn these operations into copies by setting the value number of the result equal to the value number of the first operand.

⟨Check for an expression that behaves as a copy if the operands are equal 75a⟩ ≡
```
case lAND: case lOR:
case iSIGN: case fSIGN: case dSIGN:
case iMAX: case fMAX: case dMAX:
case iMIN: case fMIN: case dMIN:
    simulate_copy(oper, value0);
    break;
```
◇

Macro referenced in scrap 74b.

The **simulate_copy** routine is called when we have discovered that an expression is equal to one of its operands. It updates the global variables representing the expression so that it will be treated as a copy.

⟨Prototypes 75b⟩ ≡
```
static Void simulate_copy(Operation *oper, Unsigned_Int value);
```
◇

Macro defined by scraps 8d, 9b, 13d, 14b, 16b, 17c, 19b, 24a, 29b, 34e, 36b, 38, 64c, 65b, 66b, 67b, 69a, 70b, 72bd, 73b, 75b, 76c, 84a.
Macro referenced in scrap 4a.

⟨Functions 75c⟩ ≡
```
static Void simulate_copy(Operation *oper, Unsigned_Int value)
{
    Unsigned_Int def_type = Operation_Def_Type(oper, dest_reg);

    switch (def_type)
    {
        case INT_DEF:         opcode_name = i2i;    break;
        case FLOAT_DEF:       opcode_name = f2f;    break;
        case DOUBLE_DEF:      opcode_name = d2d;    break;
        case COMPLEX_DEF:     opcode_name = c2c;    break;
        case DCOMPLEX_DEF:    opcode_name = q2q;    break;
        default:
            fprintf(stderr, "Internal error: Unknown type - %d.\n",
                    def_type);
            ABORT;
    }

    oper->critical = FALSE;
    if (value_num[dest_reg] != value)
    {
        changed = TRUE;
        value_num[dest_reg] = value;
    }
    value0 = value;
    value1 = 0;
}
```
◇

Macro defined by scraps 9a, 14a, 15, 16c, 18a, 19c, 24b, 29c, 35a, 36c, 39a, 65a, 66a, 67a, 68a, 69b, 71, 72c, 73ac, 75c, 77a, 84b.
Macro referenced in scrap 4a.

If we know the operands are equal, any test involving equality will always be **TRUE**. In this case, we convert the operation to a load immediate. Note that it will then be available for value numbering later.

⟨Check for an expression that tests equality of equal values 76a⟩ ≡

```
case lEQV:
case iCMPeq: case fCMPeq: case dCMPeq: case cCMPeq: case qCMPeq:
case iCMPle: case fCMPle: case dCMPle:
case iCMPge: case fCMPge: case dCMPge:
    simulate_load_immediate(oper, Expr_Install_Int(TRUE), which_table);
    break;
```

◇

Macro referenced in scrap 74b.

Similarly, any test involving inequality will always be **FALSE**. This scrap also handles subtraction of equal values.

⟨Check for an expression that tests inequality of equal values 76b⟩ ≡

```
case lXOR:
case iSUB: case fSUB: case dSUB: case cSUB: case qSUB:
case iDIM: case fDIM: case dDIM:
case iCMP: case fCMP: case dCMP: case cCMP: case qCMP:
case iCMPne: case fCMPne: case dCMPne: case cCMPne: case qCMPne:
case iCMPlt: case fCMPlt: case dCMPlt:
case iCMPgt: case fCMPgt: case dCMPgt:
    simulate_load_immediate(oper, Expr_Install_Int(0), which_table);
    break;
```

◇

Macro referenced in scrap 74b.

The **simulate_load_immediate** routine is called when we have discovered that an expression computes a constant value. It updates the global variables representing the expression so that it will be treated as a load immediate. Notice that we also look up the new expression in the expression table. This is because the operation has been changed and the calling routine will not perform the value numbering.

⟨Prototypes 76c⟩ ≡

```
static Void simulate_load_immediate(Operation *oper, Expr expr,
                                    Unsigned_Int which_table);
```

◇

Macro defined by scraps 8d, 9b, 13d, 14b, 16b, 17c, 19b, 24a, 29b, 34e, 36b, 38, 64c, 65b, 66b, 67b, 69a, 70b, 72bd, 73b, 75b, 76c, 84a.
Macro referenced in scrap 4a.

⟨Functions 77a⟩ ≡

```
static Void simulate_load_immediate(Operation *oper, Expr expr,
                                    Unsigned_Int which_table)
{
    Unsigned_Int def_type = Operation_Def_Type(oper, dest_reg);

    switch (def_type)
    {
        case INT_DEF:          opcode_name = iLDI;    break;
        case FLOAT_DEF:        opcode_name = fLDI;    break;
        case DOUBLE_DEF:       opcode_name = dLDI;    break;
        case COMPLEX_DEF:      opcode_name = cLDI;    break;
        case DCOMPLEX_DEF:     opcode_name = qLDI;    break;
        default:
            fprintf(stderr, "Internal error: Unknown type - %d.\n",
                    def_type);
            ABORT;
    }

    value0 = expr;
    value1 = 0;

    ⟨Value number the constant expression 77b⟩
}
◇
```

Macro defined by scraps 9a, 14a, 15, 16c, 18a, 19c, 24b, 29c, 35a, 36c, 39a, 65a, 66a, 67a, 68a, 69b, 71, 72c, 73ac, 75c, 77a, 84b.
Macro referenced in scrap 4a.

Since we have simplified the expression, we need to make sure there is an entry for it in the table. If a previous entry exists, we will use its value number. Otherwise, we add the expression to the table.

⟨Value number the constant expression 77b⟩ ≡

```
{
    Int result = expr_search(which_table, opcode_name, expr, 0);
    if (result >= 0)
    {
        if (value_num[dest_reg] != result)
        {
            changed = TRUE;
            value_num[dest_reg] = result;
        }
        if (valnum_level < VALNUM_UNIFIED)
            oper->critical = FALSE;
    }
    else
    {
        if (value_num[dest_reg] != dest_reg)
        {
            changed = TRUE;
            value_num[dest_reg] = dest_reg;
        }
        expr_add(which_table, opcode_name, expr, 0, dest_reg);
    }
}
◇
```

Macro referenced in scrap 77a.

If only one operand of an expression is constant, we may still be able to simplify it. For example, $x + 0 = x$. We represent constants with `Expr`s. An `Expr` can be either an integer or a string representing a name that must be resolved by the linker. For each `Expr`, we decide if it is an integer and get its value. Then we try to simplify the expression.

⟨Check for an expression with one constant operand 78⟩ ≡

```
if (simplify && ⟨We have not already simplified oper 81b⟩ &&
    ⟨This is not a xLDI operation 81c⟩)
{
    Boolean is_constant0, is_constant1;

    is_constant0 = oper->constants > 0 || VectorSet_Member(constants, value0);
    is_constant1 = value1 && VectorSet_Member(constants, value1);
    if (is_constant0)
    {
        Expr expr = oper->constants ? value0 : values[value0];
        if (Expr_Is_Integer(expr))
        {
            Int value = Expr_Get_Integer(expr);
            if (!value)
                ⟨Simplify an expression with left value zero 79a⟩
            else if (value == 1)
                ⟨Simplify an expression with left value one 79b⟩
        }
    }
    else if (is_constant1)
    {
        Expr expr = values[value1];
        if (Expr_Is_Integer(expr))
        {
            Int value = Expr_Get_Integer(expr);
            if (!value)
                ⟨Simplify an expression with right value zero 80a⟩
            else if (value == 1)
                ⟨Simplify an expression with right value one 80b⟩
        }
    }
}
◇
```

Macro referenced in scrap 74a.

If the left value of an expression is zero, we can simplify both addition and multiplication operations.

⟨Simplify an expression with left value zero 79a⟩ ≡

```
switch (opcode_name)
{
    case iADD: case fADD: case dADD: case cADD: case qADD:
    case iADDI:
    case lOR: case lXOR: case lSHIFT:
        simulate_copy(oper, value1);
        break;
    case iMUL: case fMUL: case dMUL: case cMUL: case qMUL:
    case iDIV: case fDIV: case dDIV: case cDIV: case qDIV:
    case lAND:
        simulate_load_immediate(oper, Expr_Install_Int(0), which_table);
        break;
    default:
        /*  can not simplify  */
        break;
}
◇
```

Macro referenced in scrap 78.

Multiplication by one can be replaced by a copy.

⟨Simplify an expression with left value one 79b⟩ ≡

```
switch (opcode_name)
{
    case iMUL: case fMUL: case dMUL: case cMUL: case qMUL:
        simulate_copy(oper, value1);
        break;
    default:
        /*  can not simplify  */
        break;
}
◇
```

Macro referenced in scrap 78.

If we identify a divide by zero, we will warn the user about the problem. However, if we are performing SCC-based value numbering, then this may be an optimistic assumption. Therefore, we only print the message if we are performing pessimistic value numbering.

⟨Simplify an expression with right value zero 80a⟩ ≡
```
switch (opcode_name)
{
    case iADD: case fADD: case dADD: case cADD: case qADD:
    case iSUB: case fSUB: case dSUB: case cSUB: case qSUB:
    case lOR: case lXOR:
        simulate_copy(oper, value0);
        break;
    case iMUL: case fMUL: case dMUL: case cMUL: case qMUL:
    case lAND: case lSHIFT:
        simulate_load_immediate(oper, Expr_Install_Int(0), which_table);
        break;
    case iDIV: case fDIV: case dDIV: case cDIV: case qDIV:
        if (valnum_level <= VALNUM_UNIFIED)
            fprintf(stderr, "WARNING: Divide by zero in routine.\n");
        break;
    default:
        /*  can not simplify  */
        break;
}
◇
```
Macro referenced in scrap 78.

Multiplication or division by one can be replaced by a copy. Remainder operations always result in zero when the right value is one.

⟨Simplify an expression with right value one 80b⟩ ≡
```
switch (opcode_name)
{
    case iMUL: case fMUL: case dMUL: case cMUL: case qMUL:
    case iDIV: case fDIV: case dDIV: case cDIV: case qDIV:
        simulate_copy(oper, value0);
        break;
    case iMOD: case fMOD: case dMOD:
        simulate_load_immediate(oper, Expr_Install_Int(0), which_table);
        break;
    default:
        /*  can not simplify  */
        break;
}
◇
```
Macro referenced in scrap 78.

Appendix C

Folding constants

This chapter shows how we fold expressions where both operands are constant. The first step is to determine if constant folding is actually needed. We check that (1) the user requested constant folding, (2) we have not already simplified the operation, (3) the operation is not already a load-immediate, and (4) that both operands are constants.

```
⟨Check for an expression whose operands are constants 81a⟩ ≡
      if (fold_constants && ⟨We have not already simplified oper 81b⟩ &&
          ⟨This is not a xLDI operation 81c⟩ &&
          ⟨The first operand is a constant 81d⟩ &&
          ⟨The second operand is a constant 82a⟩)
          ⟨Fold an expression that computes a constant value 82b⟩
      ◇
```

Macro referenced in scrap 74a.

We can decide if we have simplified the current operation by comparing the current value of `opcode_name` to `oper->opcode`.

```
⟨We have not already simplified oper 81b⟩ ≡
      opcode_name == oper->opcode
      ◇
```

Macro referenced in scraps 20a, 78, 81a.

There are several load immediate opcodes in `ILOC` – one for each type of operand.

WARNING: This scrap is dependent on the order that the opcodes are enumerated in the file `Shared.h`.

```
⟨This is not a xLDI operation 81c⟩ ≡
      (iLDI > opcode_name || opcode_name > qLDI)
      ◇
```

Macro referenced in scraps 78, 81a.

In `ILOC`, the first operand of an expression can be either a constant or a register. If `oper->constants > 0`, the first operand is a constant. Otherwise, the operand is a register, and we decide if it has a constant value.

```
⟨The first operand is a constant 81d⟩ ≡
      (oper->constants > 0 || VectorSet_Member(constants, value0))
      ◇
```

Macro referenced in scrap 81a.

Not all **ILOC** expressions have a second operand. If so, we have set **value1** to zero. Otherwise, the operand is a register, and we decide if it has a constant value.

⟨The second operand is a constant 82a⟩ ≡

```
(!value1 || VectorSet_Member(constants, value1))
```

◇

Macro referenced in scrap 81a.

We represent constants with **Expr**'s which can be either strings (symbolic values passed to the linker) or integers. We can fold integers by doing the math ourselves, but if one of the constants is a string, we must fold the constants symbollically by creating a new string representing the expression. The **fold_ok** variable will be set to **FALSE** if there are problems (*e.g.* division by zero) encountered during folding.

⟨Fold an expression that computes a constant value 82b⟩ ≡

```
{
    Boolean use_strings = FALSE;
    Expr expr0 = oper->constants ? value0 : values[value0];
    Expr expr1 = value1 ? values[value1] : Expr_Install_Int(0);
    Int constant0, constant1;
    Char *string0 = Expr_Get_String(expr0);
    Char *string1 = Expr_Get_String(expr1);
    Boolean fold_ok = TRUE;
    Expr result_expr;

    if (Expr_Is_Integer(expr0))
        constant0 = Expr_Get_Integer(expr0);
    else
        use_strings = TRUE;

    if (Expr_Is_Integer(expr1))
        constant1 = Expr_Get_Integer(expr1);
    else
        use_strings = TRUE;

    ⟨Find the result_expr of a constant expression 83a⟩
    if (fold_ok)
        simulate_load_immediate(oper, result_expr, which_table);
}
```

◇

Macro referenced in scrap 81a.

We will divide the opcodes into several logical groups. The first group is the load-immediate opcodes – these expressions are already constants.

⟨Find the `result_expr` of a constant expression 83a⟩ ≡
```
switch (opcode_name)
{
    case iLDI: case fLDI: case dLDI: case cLDI: case qLDI:
    case bCONor: case iCONor: case fCONor: dCONor: cCONor: qCONor:
        fold_ok = FALSE;
        break;
    ⟨Fold an addition expression 83b⟩
    ⟨Fold a subtraction expression 84c⟩
    ⟨Fold a multiplication expression 85⟩
    ⟨Fold a division expression 86⟩
    ⟨Fold a mod expression 87a⟩
    ⟨Fold a comparison expression 87b, ... ⟩
    ⟨Fold a condition code expression 89a⟩
    ⟨Fold a negation expression 89b⟩
    ⟨Fold an absolute value expression 90a⟩
    ⟨Fold a shift expression 90b, ... ⟩
    ⟨Fold a logical expression 92, ... ⟩
    ⟨Fold a conversion expression 93b⟩
    ⟨Fold a square root expression 94a⟩
    ⟨Fold an exponentiation expression 94b⟩
    ⟨Fold a trigonometric expression 95⟩
    default:
        fprintf(stderr, "Internal error: Unknown opcode - %s.\n",
                opcode_specs[opcode_name].opcode);
        ABORT;
}
◇
```
Macro referenced in scrap 82b.

If one of the operands of addition is a string, we may still be able to simplify the expression if the other operand is 0.

⟨Fold an addition expression 83b⟩ ≡
```
case iADD: case fADD: case dADD: case cADD: case qADD:
case iADDI:
    if (use_strings)
        if (Expr_Is_Integer(expr0) && !constant0)
            result_expr = expr1;
        else if (Expr_Is_Integer(expr1) && !constant1)
            result_expr = expr0;
        else
            result_expr = fold_strings(string0, string1, '+');
    else
        result_expr = Expr_Install_Int(constant0 + constant1);
    break;
◇
```
Macro referenced in scrap 83a.

When we are folding strings, we must monkey around with the underscore character. The **fold_strings** function handles this. Hopefully, this will be a temporary fix that can someday be removed.

⟨Prototypes 84a⟩ ≡

```
static Expr fold_strings(Char *string0, Char *string1, Char op);
```

◇

Macro defined by scraps 8d, 9b, 13d, 14b, 16b, 17c, 19b, 24a, 29b, 34e, 36b, 38, 64c, 65b, 66b, 67b, 69a, 70b, 72bd, 73b, 75b, 76c, 84a.
Macro referenced in scrap 4a.

⟨Functions 84b⟩ ≡

```
static Expr fold_strings(Char *string0, Char *string1, Char op)
{
    Char result_string[200];
    Char ch0 = string0[strlen(string0) - 1];
    Boolean need_under0 = ch0 != ')' && !isdigit(ch0);
    Char ch1 = string1[strlen(string1) - 1];
    Boolean need_under1 = ch1 != ')' && !isdigit(ch1);

    if (need_under0 && need_under1)
        sprintf(result_string, "(%s_)%c(%s_)", string0, op, string1);
    else if (need_under0)
        sprintf(result_string, "(%s_)%c(%s)", string0, op, string1);
    else if (need_under1)
        sprintf(result_string, "(%s)%c(%s_)", string0, op, string1);
    else
        sprintf(result_string, "(%s)%c(%s)", string0, op, string1);

    return Expr_Install_String(result_string);
}
```

◇

Macro defined by scraps 9a, 14a, 15, 16c, 18a, 19c, 24b, 29c, 35a, 36c, 39a, 65a, 66a, 67a, 68a, 69b, 71, 72c, 73ac, 75c, 77a, 84b.
Macro referenced in scrap 4a.

⟨Fold a subtraction expression 84c⟩ ≡

```
case iSUB: case fSUB: case dSUB: case cSUB: case qSUB:
case iSUBI:
    if (use_strings)
        if (Expr_Is_Integer(expr1) && !constant1)
            result_expr = expr0;
        else
            result_expr = fold_strings(string0, string1, '-');
    else
        result_expr = Expr_Install_Int(constant0 - constant1);
    break;
```

◇

Macro referenced in scrap 83a.

If one of the operands of multiplication is a string, we may still be able to simplify the expression if the other operand is 0 or 1.

⟨Fold a multiplication expression 85⟩ ≡

```
case iMUL: case fMUL: case dMUL: case cMUL: case qMUL:
    if (use_strings)
        if (Expr_Is_Integer(expr0) && !constant0)
            result_expr = Expr_Install_Int(0);
        else if (Expr_Is_Integer(expr0) && constant0 == 1)
            result_expr = expr1;
        else if (Expr_Is_Integer(expr1) && !constant1)
            result_expr = Expr_Install_Int(0);
        else if (Expr_Is_Integer(expr1) && constant1 == 1)
            result_expr = expr0;
        else
            result_expr = fold_strings(string0, string1, '*');
    else
        result_expr = Expr_Install_Int(constant0 * constant1);
    break;
```

◇

Macro referenced in scrap 83a.

We can only represent integers as **Expr**'s, so we must check any floating point operations to make sure the result is still an integer. Also, notice that we only report division by zero if we are performing pessimistic value numbering.

⟨Fold a division expression 86⟩ ≡

```
case iDIV:
    if (use_strings)
        if (Expr_Is_Integer(expr0) && !constant0)
            result_expr = Expr_Install_Int(0);
        else if (Expr_Is_Integer(expr1) && constant1 == 1)
            result_expr = expr0;
        else
            result_expr = fold_strings(string0, string1, '/');
    else
    {
        if (!constant1)
        {
            if (valnum_level <= VALNUM_UNIFIED)
                fprintf(stderr, "WARNING: Divide by zero in routine.\n");
            fold_ok = FALSE;
        }
        else
            result_expr = Expr_Install_Int(constant0 / constant1);
    }
    break;
case fDIV: case dDIV: case cDIV: qDIV:
    if (use_strings)
        if (Expr_Is_Integer(expr0) && !constant0)
            result_expr = Expr_Install_Int(0);
        else if (Expr_Is_Integer(expr1) && constant1 == 1)
            result_expr = expr0;
        else
            fold_ok = FALSE;
    else
    {
        if (!constant1)
        {
            if (valnum_level <= VALNUM_UNIFIED)
                fprintf(stderr, "WARNING: Divide by zero in routine.\n");
            fold_ok = FALSE;
        }
        else
        {
            Int answer = constant0 / constant1;
            if (answer*constant1 == constant0)
                result_expr = Expr_Install_Int(answer);
            else
                fold_ok = FALSE;
        }
    }
    break;
```
◇

Macro referenced in scrap 83a.

⟨Fold a mod expression 87a⟩ ≡

```
case iMOD: case fMOD: case dMOD:
    if (use_strings)
        if (Expr_Is_Integer(expr1) && constant1 == 1)
            result_expr = Expr_Install_Int(0);
        else
            fold_ok = FALSE;
    else
    {
        if (!constant1)
        {
            fprintf(stderr, "WARNING: Mod by zero in routine.\n");
            fold_ok = FALSE;
        }
        else
            result_expr = Expr_Install_Int(constant0 % constant1);
    }
    break;
```

◇

Macro referenced in scrap 83a.

`ILOC` has a basic compare operation for each type of operand. The comparison returns `-1`, `0`, or `1` indicating that the first operand is less than, equal to, or greater than the second operand.

⟨Fold a comparison expression 87b⟩ ≡

```
case iCMP: fCMP: case dCMP: case cCMP: case qCMP:
    if (use_strings)
        fold_ok = FALSE;
    else
    {
        if (constant0 < constant1)
            result_expr = Expr_Install_Int(-1);
        else if (constant0 == constant1)
            result_expr = Expr_Install_Int(0);
        else
            result_expr = Expr_Install_Int(1);
    }
    break;
```

◇

Macro defined by scraps 87b, 88.
Macro referenced in scrap 83a.

`ILOC` also has compare operations based on a specific relationship (*e.g.*, $=$). These comparisons return `TRUE` or `FALSE`.

⟨Fold a comparison expression 88⟩ ≡

```
case iCMPeq: case fCMPeq: case dCMPeq: case cCMPeq: case qCMPeq:
    if (use_strings)
        fold_ok = FALSE;
    else
        result_expr = Expr_Install_Int(constant0 == constant1 ? TRUE : FALSE);
    break;
case iCMPne: case fCMPne: case dCMPne: case cCMPne: case qCMPne:
    if (use_strings)
        fold_ok = FALSE;
    else
        result_expr = Expr_Install_Int(constant0 != constant1 ? TRUE : FALSE);
    break;
case iCMPle: case fCMPle: case dCMPle:
    if (use_strings)
        fold_ok = FALSE;
    else
        result_expr = Expr_Install_Int(constant0 <= constant1 ? TRUE : FALSE);
    break;
case iCMPge: case fCMPge: case dCMPge:
    if (use_strings)
        fold_ok = FALSE;
    else
        result_expr = Expr_Install_Int(constant0 >= constant1 ? TRUE : FALSE);
    break;
case iCMPlt: case fCMPlt: case dCMPlt:
    if (use_strings)
        fold_ok = FALSE;
    else
        result_expr = Expr_Install_Int(constant0 < constant1 ? TRUE : FALSE);
    break;
case iCMPgt: case fCMPgt: case dCMPgt:
    if (use_strings)
        fold_ok = FALSE;
    else
        result_expr = Expr_Install_Int(constant0 > constant1 ? TRUE : FALSE);
    break;
◇
```

Macro defined by scraps 87b, 88.
Macro referenced in scrap 83a.

The condition code operations take the result of a basic compare operation (*e.g.*, `iCMP`) and return `TRUE` or `FALSE` based on a spedific relationship (*e.g.*, =).

⟨Fold a condition code expression 89a⟩ ≡

```
case EQ:
    if (use_strings)
        fold_ok = FALSE;
    else
        result_expr = Expr_Install_Int(!constant0 ? TRUE : FALSE);
    break;
case NE:
    if (use_strings)
        fold_ok = FALSE;
    else
        result_expr = Expr_Install_Int(constant0 ? TRUE : FALSE);
    break;
case LE:
    if (use_strings)
        fold_ok = FALSE;
    else
        result_expr = Expr_Install_Int(constant0 != 1 ? TRUE : FALSE);
    break;
case GE:
    if (use_strings)
        fold_ok = FALSE;
    else
        result_expr = Expr_Install_Int(constant0 != -1 ? TRUE : FALSE);
    break;
case LT:
    if (use_strings)
        fold_ok = FALSE;
    else
        result_expr = Expr_Install_Int(constant0 == -1 ? TRUE : FALSE);
    break;
case GT:
    if (use_strings)
        fold_ok = FALSE;
    else
        result_expr = Expr_Install_Int(constant0 == 1 ? TRUE : FALSE);
    break;
```

◇

Macro referenced in scrap 83a.

⟨Fold a negation expression 89b⟩ ≡

```
case iNEG: case fNEG: case dNEG: case cNEG: case qNEG:
    if (use_strings)
        result_expr = fold_strings("0", string0, '-');
    else
        result_expr = Expr_Install_Int(-constant0);
    break;
```

◇

Macro referenced in scrap 83a.

⟨Fold an absolute value expression 90a⟩ ≡

```
case iABS: case fABS: case dABS: case cABS: case qABS:
    if (use_strings)
        fold_ok = FALSE;
    else if (constant0 >= 0)
        result_expr = expr0;
    else
        result_expr = Expr_Install_Int(-constant0);
    break;
◇
```

Macro referenced in scrap 83a.

⟨Fold a shift expression 90b⟩ ≡

```
case lSHIFT:
    if (use_strings)
        if (Expr_Is_Integer(expr0) && !constant0)
            result_expr = expr1;
        else if (Expr_Is_Integer(expr1) && !constant1)
            result_expr = Expr_Install_Int(0);
        else
            fold_ok = FALSE;
    else if (constant0  > 0)
        result_expr = Expr_Install_Int(constant1 << constant0);
    else
        result_expr = Expr_Install_Int(constant1 >> -constant0);
    break;
◇
```

Macro defined by scraps 90b, 91.
Macro referenced in scrap 83a.

Notice the special handling for `iSR` and `iSRI`. This is to account for the fact that ANSI C does not specify whether the `>>` operator is an arithmetic shift (vacated bits filled with sign bits) or a logical shift (vacated bits filled in with zero). We want `iSR` and `iSRI` to implement arithmetic shifts. The `WORD_BIT` constant is defined in `<limits.h>`; it represents the number of bits in an integer on the target machine.

⟨Fold a shift expression 91⟩ ≡

```
case lSL: case iSL: case lSLI: case iSLI:
    if (use_strings)
        if (Expr_Is_Integer(expr0) && !constant0)
            result_expr = expr1;
        else if (Expr_Is_Integer(expr1) && !constant1)
            result_expr = Expr_Install_Int(0);
        else
            fold_ok = FALSE;
    else
        result_expr = Expr_Install_Int(constant1 << constant0);
    break;
case lSR: case iSR: case lSRI: case iSRI:
    if (use_strings)
        if (Expr_Is_Integer(expr0) && !constant0)
            result_expr = expr1;
        else if (Expr_Is_Integer(expr1) && !constant1)
            result_expr = Expr_Install_Int(0);
        else
            fold_ok = FALSE;
    else if ((opcode_name == iSR || opcode_name == iSRI) && constant1 < 0)
    {
        Unsigned_Int pad = ~0 << (WORD_BIT - constant0);
        result_expr = Expr_Install_Int((constant1 >> constant0) | pad);
    }
    else
        result_expr = Expr_Install_Int(constant1 >> constant0);
    break;
```

◇

Macro defined by scraps 90b, 91.
Macro referenced in scrap 83a.

⟨Fold a logical expression 92⟩ ≡

```
case lAND:
    if (use_strings)
        if (Expr_Is_Integer(expr0) && !constant0)
            result_expr = Expr_Install_Int(0);
        else if (Expr_Is_Integer(expr1) && !constant1)
            result_expr = Expr_Install_Int(0);
        else
            fold_ok = FALSE;
    else
        result_expr = Expr_Install_Int(constant0 & constant1);
    break;
case lOR:
    if (use_strings)
        if (Expr_Is_Integer(expr0) && !constant0)
            result_expr = expr1;
        else if (Expr_Is_Integer(expr1) && !constant1)
            result_expr = expr0;
        else
            fold_ok = FALSE;
    else
        result_expr = Expr_Install_Int(constant0 | constant1);
    break;
case lNAND:
    if (use_strings)
        fold_ok = FALSE;
    else
        result_expr = Expr_Install_Int(~(constant0 & constant1));
    break;
◇
```

Macro defined by scraps 92, 93a.
Macro referenced in scrap 83a.

⟨Fold a logical expression 93a⟩ ≡

```
case lNOR:
    if (use_strings)
        fold_ok = FALSE;
    else
        result_expr = Expr_Install_Int(~(constant0 | constant1));
    break;
case lEQV:
    if (use_strings)
        fold_ok = FALSE;
    else
        result_expr = Expr_Install_Int(~(constant0 ^ constant1));
    break;
case lXOR:
    if (use_strings)
        if (Expr_Is_Integer(expr0) && !constant0)
            result_expr = expr1;
        else if (Expr_Is_Integer(expr1) && !constant1)
            result_expr = expr0;
        else
            fold_ok = FALSE;
    else
        result_expr = Expr_Install_Int(constant0 ^ constant1);
    break;
case lNOT:
    if (use_strings)
        fold_ok = FALSE;
    else
        result_expr = Expr_Install_Int(~constant0);
    break;
```

◇

Macro defined by scraps 92, 93a.
Macro referenced in scrap 83a.

⟨Fold a conversion expression 93b⟩ ≡

```
case i2f: case i2d: case i2c: case i2q:
case f2i: case f2d: case f2c: case f2q:
case d2i: case d2f: case d2c: case d2q:
case c2i: case c2f: case c2d: case c2q:
case q2i: case q2f: case q2d: case q2c:
    result_expr = expr0;
    break;
```

◇

Macro referenced in scrap 83a.

Square roots and exponentiation are also examples where we must make sure the result of a floating point operation is still an integer. Notice that we only report an error if we are performing pessimistic value numbering.

⟨Fold a square root expression 94a⟩ ≡

```
case fSQRT: case dSQRT: case cSQRT: case qSQRT:
    if (use_strings)
        fold_ok = FALSE;
    else if (constant0 < 0)
    {
        if (valnum_level <= VALNUM_UNIFIED)
            fprintf(stderr, "WARNING: Square root of a negative number taken.\n");
        fold_ok = FALSE;
    }
    else
    {
        Double result = sqrt((Double)constant0);
        Int int_result = (Int) result;
        if (result == int_result)
            result_expr = Expr_Install_Int(int_result);
        else
            fold_ok = FALSE;
    }
    break;
```

◇

Macro referenced in scrap 83a.

⟨Fold an exponentiation expression 94b⟩ ≡

```
case iPOW: case fPOW: case dPOW: case cPOW: case qPOW:
case fPOWi: case dPOWi: case cPOWi: case qPOWi:
    if (use_strings)
        fold_ok = FALSE;
    else
    {
        Double result = pow((Double)constant0, (Double)constant1);
        Int int_result = (Int) result;
        if (result == int_result)
            result_expr = Expr_Install_Int(int_result);
        else
            fold_ok = FALSE;
    }
    break;
```

◇

Macro referenced in scrap 83a.

⟨Fold a trigonometric expression 95⟩ ≡

```
case fCOS: case dCOS:
case cCOS: case qCOS:
case fCOSH: case dCOSH:
    if (use_strings || constant0)
        fold_ok = FALSE;
    else
        result_expr = Expr_Install_Int(1);
    break;
case fTAN: case dTAN:
case fASIN: case dASIN:
case fATAN: case dATAN:
case fSINH: case dSINH:
case fTANH: case dTANH:
    if (use_strings || constant0)
        fold_ok = FALSE;
    else
        result_expr = Expr_Install_Int(0);
    break;
case fACOS: case dACOS:
    if (use_strings || constant0 != 1)
        fold_ok = FALSE;
    else
        result_expr = Expr_Install_Int(0);
    break;
case fATAN2: case dATAN2:
    fold_ok = FALSE;
    break;
```

◇

Macro referenced in scrap 83a.

Appendix D

Indices

D.1 Files

"`valnum.c`" Defined by scrap 4a.

D.2 Scraps

D.3 Identifiers

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, San Diego, California, January 1988.

[3] Jeffrey M. Barth. An interprocedural data flow analysis algorithm. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 119–131, Los Angeles, California, January 1977.

[4] Preston Briggs. The massively scalar compiler project. Technical report, Rice University, July 1994. Preliminary version available via anonymous ftp.

[5] Preston Briggs, Keith D. Cooper, and L. Taylor Simpson. Value numbering. *Software – Practice and Experience*, 1996. (*to appear*).

[6] Preston Briggs, Tim Harvey, and Taylor Simpson. Static single assignment construction. Technical report, Rice University, May 1995. Available via anonymous ftp.

[7] Preston Briggs and Rob Shillner. Elimination of partial redundancies. Technical report, Rice University, March 1993. Available via anonymous ftp.

[8] Cliff Click. *Combining Analyses, Combining Optimizations.* PhD thesis, Rice University, 1995.

[9] John Cocke and Jacob T. Schwartz. Programming languages and their compilers: Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University, 1970.

[10] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[11] Matthew S. Hecht. *Flow Analysis of Computer Programs.* Programming Languages Series. Elsevier North-Holland, Inc., 52 Vanderbilt Avenue, New York, NY 10017, 1977.

[12] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, January 1976.

[13] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[14] Etienne Morel and Claude Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[15] L. Taylor Simpson. *Value-Driven Redundancy Elimination.* PhD thesis, Rice University, May 1996.

[16] Taylor Simpson. Global value numbering. Technical report, Rice University, August 1994. Available via anonymous ftp.

[17] Robert E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, June 1972.

[18] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 291–299, New Orleans, Louisiana, January 1985.

[19] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.