Multiplication by Integer Constants

Preston Briggs
preston@cs.rice.edu

Tim Harvey
harv@cs.rice.edu

July 13, 1994

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

Contents

1 Introduction

Some modern machines have no integer multiply instruction and must rely on expensive software methods to compute integer products. In other cases, the multiply instruction is significantly slower than simple integer addition. When faced with computing $n \times c$, where n is some unknown integer value and c is a known integer constant, we can avoid the need for a general-purpose multiply by rewriting the expression in terms of shifts, adds, and subtracts – typically all one-cycle instructions.

Bernstein gives a detailed discussion of the problem and presents a solution, including Ada code for its implementation [1]. Unfortunately, the code is flawed, at least in part due to typesetting errors. It's also quite difficult to understand.

This document represents an attempt to explain the elements of Bernstein's approach. At the same time, we will develop a complete, working, and hopefully understandable implementation of his approach.

2 Finding Solutions

We don't know of an efficient algorithm that finds optimal sequences of shifts, adds, and subtracts to accomplish multiplication (nor do we know of a proof of the problem's complexity). Bernstein describes two heuristic approaches: the *binary method* and the *factoring method.* Neither finds optimal solutions for every case and neither dominates the other; therefore, Bernstein suggests a combination of the two methods.

2.1 The Binary Method

One way to compute $n \times c$ is to examine the binary representation of the constant c. Each of the 1 bits implies a shift and add (actually, all but one of the bits). For example, to compute $n \times 113$, we observe that $113_{10} = 1110001_2$ giving the sequence

$$
\begin{array}{rcl}
2n & \leftarrow & n \ll 1 \\
3n & \leftarrow & 2n + n \\
6n & \leftarrow & 3n \ll 1 \\
7n & \leftarrow & 6n + n \\
112n & \leftarrow & 7n \ll 4 \\
113n & \leftarrow & 112n + n
\end{array}
$$

Basically, we can simply look at the bits from left to right and dictate the answer.

We can get better solutions by using subtract to handle runs of 1 bits. For example, the previous case could be handled by the sequence

$$
\begin{array}{rcl}
8n & \leftarrow & n \ll 3 \\
7n & \leftarrow & 8n - n \\
112n & \leftarrow & 7n \ll 4 \\
113n & \leftarrow & 112n + n
\end{array}
$$

While it's possible to give code that directly generates solutions using the binary method, we will wait and give a more general method capable of handling the binary method in combination with the factoring method.

2.2 Factoring

Sometimes we can factor the constant multiplier and achieve shorter instruction sequences. Consider the case of $n \times 585$. Since $585_{10} = 1001001001_2$, using the binary method would yield

$$\begin{array}{rcl} 8n & \leftarrow & n \ll 3 \\ 9n & \leftarrow & 8n + n \\ 72n & \leftarrow & 9n \ll 3 \\ 73n & \leftarrow & 72n + n \\ 584n & \leftarrow & 73n \ll 3 \\ 585n & \leftarrow & 584n + n \end{array}$$

However, if we recognize that $585 = 9 \times 65$, we can first generate $9n$, then $65 \times 9n$, requiring only four operations.

$$\begin{array}{rcl} 8n & \leftarrow & n \ll 3 \\ 9n & \leftarrow & 8n + n \\ 576n & \leftarrow & 9n \ll 6 \\ 585n & \leftarrow & 576n + 9n \end{array}$$

Of course, not all factors can be generated quickly. Bernstein suggests searching for factors of the form $2^i \pm 1$ since they can be generated in only two instructions.

3 Searching for Solutions

The choice between the different methods for handling a particular positive constant can resolved by evaluating the formula

$$\begin{aligned} \text{Cost}(1) &= 0 && (1) \\ \text{Cost}(\text{even } c) &= \text{Cost}(\mathit{makeOdd}(c)) + \mathit{shiftCost} && (2) \\ \text{Cost}(\text{odd } c) &= \min(\text{Cost}(c+1) + \mathit{subCost}, && (3) \\ &\qquad \text{Cost}(c-1) + \mathit{addCost}, && (4) \\ &\qquad \text{Cost}(c/(2^i+1)) + \mathit{subCost} + \mathit{shiftCost}, && (5) \\ &\qquad \text{Cost}(c/(2^i-1)) + \mathit{addCost} + \mathit{shiftCost}) && (6) \end{aligned}$$

where we make several assumptions:

- The function $\mathit{makeOdd}(c)$ returns the odd number m such that $m \times 2^i = c$; that is, c is right shifted until it becomes odd.
- We assume that a left shift can shift an arbitrary amount in constant time.
- Line 5 is evaluated for every value of i such that $c/(2^i + 1)$ is an integral value. The same assumption applies to line 6.

Extending the formula to handle both positive and negative constants gives

$$\begin{aligned}
\mathrm{Cost}(1) &= 0 \\
\mathrm{Cost}(-1) &= \mathit{negateCost} \\
\mathrm{Cost}(\text{even } c) &= \mathrm{Cost}(\mathit{makeOdd}(c)) + \mathit{shiftCost} \\
\mathrm{Cost}(\text{odd } c > 1) &= \min(\mathrm{Cost}(c+1) + \mathit{subCost}, \\
&\qquad \mathrm{Cost}(c-1) + \mathit{addCost}, \\
&\qquad \mathrm{Cost}(c/(2^i+1)) + \mathit{subCost} + \mathit{shiftCost}, \\
&\qquad \mathrm{Cost}(c/(2^i-1)) + \mathit{addCost} + \mathit{shiftCost}) \\
\mathrm{Cost}(\text{odd } c < -1) &= \min(\mathrm{Cost}(c+1) + \mathit{subCost}, \\
&\qquad \mathrm{Cost}(1-c) + \mathit{subCost}, \\
&\qquad \mathrm{Cost}(c/(2^i+1)) + \mathit{subCost} + \mathit{shiftCost}, \\
&\qquad \mathrm{Cost}(c/(1-2^i)) + \mathit{addCost} + \mathit{shiftCost})
\end{aligned}$$

Note that we are assuming a two's-complement representation for negative numbers.

Of course, a nice formula is not working code. In the next sections, we develop an efficient implementation of the search implied by the above formula. In the meantime, we give the initial boilerplate for `"multiply.c"`.

`"multiply.c"` 3a ≡
```
⟨Include files 3b⟩
⟨Constant definitions 5a, ... ⟩
⟨Type definitions 7a, ... ⟩
⟨Variable definitions 7b, ... ⟩
⟨Function definitions 3c, ... ⟩
◇
```

We only need one include file for I/O.

⟨Include files 3b⟩ ≡
```
#include <stdio.h>
◇
```

Macro referenced in scrap 3a.

We define a pair of simple helper functions – predicates to test whether a number is odd or even.

⟨Function definitions 3c⟩ ≡
```
#define odd(c) ((c) & 1)
#define even(c) (!odd(c))
◇
```

Macro defined by scraps 3cd, 10d, 13ab, 15bcd, 16c, 17.
Macro referenced in scrap 3a.

We assert that `makeOdd` is always invoked with a non-zero, even argument. The implementation is fairly simple; however, note that `c/2` cannot be replaced by `c>>1` since right shift of negative integers is not guaranteed to preserve the sign bit in ANSI C. This is very sad, but the code can be easily customized for any particular machine/compiler combination with the correct behavior.

⟨Function definitions 3d⟩ ≡
```
static int makeOdd(int c)
{
  do
    c = c / 2;
  while (even(c));
  return c;
}
◇
```

Macro defined by scraps 3cd, 10d, 13ab, 15bcd, 16c, 17.
Macro referenced in scrap 3a.

3.1 Exploring the Search Space

For a first cut, we'll simply give the code that explores the search space for an odd constant. It doesn't actually return any result, so it's not very interesting except to show the planned form of the eventual code. Of course, we recognize this code is terribly slow and wasteful; it'll be refined in later sections.

⟨A first cut 4a⟩ ≡
```
static void explore(int c)
{
  if (c > 1) {
    ⟨Try factors for the positive case 4b⟩
    explore(makeOdd(c - 1));
    explore(makeOdd(c + 1));
  }
  else if (c < -1) {
    ⟨Try factors for the negative case 4c⟩
    explore(makeOdd(1 - c));
    explore(makeOdd(c + 1));
  }
}
◇
```
Macro never referenced.

The code above handles the binary method. To explore different factorings, we include these fragments. By starting at 4 (*i.e.*, 2^2), we initally try factors of 3 and 5, then 7 and 9, *etc.*

⟨Try factors for the positive case 4b⟩ ≡
```
{
  int power = 4;
  int edge = n >> 1;
  while (power < edge) {
    if (c % (power - 1) == 0) explore(c / (power - 1));
    if (c % (power + 1) == 0) explore(c / (power + 1));
    power = power << 1;
  }
}◇
```
Macro referenced in scrap 4a.

The negative case is nearly identical. Note that the expression `-c` will never overflow since `c` is guaranteed to be odd.

⟨Try factors for the negative case 4c⟩ ≡
```
{
  int power = 4;
  int edge = (-c) >> 1;
  while (power < edge) {
    if (c % (1 - power) == 0) explore(c / (1 - power));
    if (c % (power + 1) == 0) explore(c / (power + 1));
    power = power << 1;
  }
}◇
```
Macro referenced in scrap 4a.

3.2 Determining Costs

For a second cut, we'll determine the cost of handling a particular constant. We'll need to define the *machine-dependent* costs for the various desired operations.

⟨Constant definitions 5a⟩ ≡

```
#define ADD_COST 1
#define SUB_COST 1
#define NEG_COST 1
#define SHIFT_COST 1
◇
```

Macro defined by scraps 5a, 10b.
Macro referenced in scrap 3a.

⟨A second cut 5b⟩ ≡

```
static unsigned int find_cost(int c)
{
  if (c == 1)
    return 0;
  else if (c > 1)
    ⟨Explore the positive case 5c, ... ⟩
  else if (c == -1)
    return NEG_COST;
  else
    ⟨Explore the negative case 6c⟩
}
◇
```

Macro defined by scraps 5b, 6a.
Macro never referenced.

Basically, we explore each of the options for generating `n`, comparing costs and keeping the cheapest alternative.

⟨Explore the positive case 5c⟩ ≡

```
{
  unsigned int cost;
  unsigned int best_cost = 10000; /* i.e., a big number! */
  int power = 4;
  int edge = c >> 1;
  while (power < edge) {
    if (c % (power - 1) == 0) {
      cost = find_cost(c / (power - 1)) + SHIFT_COST + ADD_COST;
      if (cost < best_cost) best_cost = cost;
    }
    if (c % (power + 1) == 0) {
      cost = find_cost(c / (power + 1)) + SHIFT_COST + SUB_COST;
      if (cost < best_cost) best_cost = cost;
    }
    power = power << 1;
  }
  cost = find_cost(makeOdd(c - 1)) + SHIFT_COST + ADD_COST;
  if (cost < best_cost) best_cost = cost;
  cost = find_cost(makeOdd(c + 1)) + SHIFT_COST + SUB_COST;
  if (cost < best_cost) best_cost = cost;
  return best_cost;
}◇
```

Macro defined by scraps 5c, 6b.
Macro referenced in scrap 5b.

The negative case would be similar, though we don't give it explicitly.

3.2.1 Introducing a Helper Routine

An alternative approach relies on a small helper procedure called **try** that encapsulates the invocation of **find_cost** and the test for minimal cost.

Of course, we pay some cost in clarity since we now have a pair of mutually recursive routines. Hopefully the incremental presentation will overcome any doubts about the correctness (combined with the obvious waste of the original alternative). In later versions, we expect that **try** will become a more significant routine.

⟨A second cut 6a⟩ ≡

```
static void try(int factor, unsigned int cost, unsigned int *best_cost)
{
  cost += find_cost(factor);
  if (cost < *best_cost)
    *best_cost = cost;
}
◇
```

Macro defined by scraps 5b, 6a.
Macro never referenced.

The new code for exploring the various alternatives is given here.

⟨Explore the positive case 6b⟩ ≡

```
{
  unsigned int best_cost = 10000;
  int power = 4;
  int edge = c >> 1;
  while (power < edge) {
    if (c % (power - 1) == 0)
      try(c / (power - 1), SHIFT_COST + ADD_COST, &best_cost);
    if (c % (power + 1) == 0)
      try(c / (power + 1), SHIFT_COST + SUB_COST, &best_cost);
    power = power << 1;
  }
  try(makeOdd(c - 1), SHIFT_COST + ADD_COST, &best_cost);
  try(makeOdd(c + 1), SHIFT_COST + SUB_COST, &best_cost);
  return best_cost;
}◇
```

Macro defined by scraps 5c, 6b.
Macro referenced in scrap 5b.

The negative case corresponds closely with the positive case.

⟨Explore the negative case 6c⟩ ≡

```
{
  unsigned int best_cost = 10000;
  int power = 4;
  int edge = (-c) >> 1;
  while (power < edge) {
    if (c % (power - 1) == 0)
      try(c / (1 - power), SHIFT_COST + SUB_COST, &best_cost);
    if (c % (power + 1) == 0)
      try(c / (power + 1), SHIFT_COST + SUB_COST, &best_cost);
    power = power << 1;
  }
  try(makeOdd(1 - c), SHIFT_COST + SUB_COST, &best_cost);
  try(makeOdd(c + 1), SHIFT_COST + SUB_COST, &best_cost);
  return best_cost;
}◇
```

Macro referenced in scrap 5b.

3.3 Recording Results

Since we are interested in actually converting mutiplies to simpler instructions, not just finding the cost of such a conversion, we need to record the results of each branch of our search. A tree representation seems natural. First, however, we need names for the alternative approaches at each node.

⟨Type definitions 7a⟩ ≡

```
typedef
  enum {
    IDENTITY,          /* used for n = 1          */
    NEGATE,            /* used for n = -1         */
    SHIFT_ADD,         /* used for makeOdd(n - 1) */
    SHIFT_SUB,         /* used for makeOdd(n + 1) */
    SHIFT_REV,         /* used for makeOdd(1 - n) */
    FACTOR_ADD,        /* used for n/(2^i - 1)    */
    FACTOR_SUB,        /* used for n/(2^i + 1)    */
    FACTOR_REV         /* used for n/(1 - 2^i)    */
  } MulOp;
◇
```

Macro defined by scraps 7ac.
Macro referenced in scrap 3a.

As a notational convenience (and to help avoid typos), we introduce a table holding the costs of each possible **MulOp**.

⟨Variable definitions 7b⟩ ≡

```
static unsigned int costs[] =
  { 0,                         /* for IDENTITY   */
    NEG_COST,                  /* for NEGATE     */
    SHIFT_COST + ADD_COST,     /* for SHIFT_ADD  */
    SHIFT_COST + SUB_COST,     /* for SHIFT_SUB  */
    SHIFT_COST + SUB_COST,     /* for SHIFT_REV  */
    SHIFT_COST + ADD_COST,     /* for FACTOR_ADD */
    SHIFT_COST + SUB_COST,     /* for FACTOR_SUB */
    SHIFT_COST + SUB_COST      /* for FACTOR_REV */
  };
◇
```

Macro defined by scraps 7b, 10c.
Macro referenced in scrap 3a.

The initial definition of a tree node is given here; we'll add a few extra fields later. The `value` field will record the target factor. Initially, `parent` will be `NULL`, indicating that the `cost` field has not been initialized. After one or more alternative derivations have been explored, the `cost` field will indicate the cost of the best (cheapest) alternative to date, where the `opcode` and `parent` fields will indicate the next step in the derivation.

⟨Type definitions 7c⟩ ≡

```
typedef
  struct node {
    struct node *parent;
    int value;
    unsigned int cost;
    MulOp opcode;
    ⟨Other fields in Node 10a⟩
  } Node;
◇
```

Macro defined by scraps 7ac.
Macro referenced in scrap 3a.

We'll modify our search routine to return a pointer into the tree. By tracing back along parent pointers, we'll eventually reach the "one" entry. Reversing the order will give the sequence of desired operations (see Section 4).

⟨A third cut 8a⟩ ≡
```
static Node *find_sequence(int c)
{
  Node *node;
  ⟨Create and initialize node 8b, ... ⟩
  if (c == 1)
    ⟨Handle the identity case 8c⟩
  else if (c > 1)
    ⟨Handle the positive case 9a⟩
  else if (c == -1)
    ⟨Handle the negate case 8d⟩
  else
    ⟨Handle the negative case 9b⟩
  return node;
}
◇
```
Macro defined by scraps 8a, 9c.
Macro never referenced.

⟨Create and initialize node 8b⟩ ≡
```
node = (Node *) malloc(sizeof(Node));
node->value = c;
node->parent = NULL;
◇
```
Macro defined by scraps 8b, 12c.
Macro referenced in scraps 8a, 10e.

We'll give the simplest cases first, just to set the stage.

⟨Handle the identity case 8c⟩ ≡
```
{
  node->cost = 0;
  node->opcode = IDENTITY;
}◇
```
Macro referenced in scrap 8a.

⟨Handle the negate case 8d⟩ ≡
```
{
  node->opcode = NEGATE;
  node->parent = find_sequence(1);
  node->cost = node->parent->cost + NEG_COST;
}◇
```
Macro referenced in scrap 8a.

The other cases change only slightly in that they pass different arguments to **try**.

⟨Handle the positive case 9a⟩ ≡

```
{
  int power = 4;
  int edge = c >> 1;
  while (power < edge) {
    if (c % (power - 1) == 0) try(c / (power - 1), node, FACTOR_SUB);
    if (c % (power + 1) == 0) try(c / (power + 1), node, FACTOR_ADD);
    power = power << 1;
  }
  try(makeOdd(c - 1), node, SHIFT_ADD);
  try(makeOdd(c + 1), node, SHIFT_SUB);
}◇
```

Macro referenced in scraps 8a, 11, 12a.

⟨Handle the negative case 9b⟩ ≡

```
{
  int power = 4;
  int edge = (-c) >> 1;
  while (power < edge) {
    if (c % (1 - power) == 0) try(c / (1 - power), node, FACTOR_REV);
    if (c % (power + 1) == 0) try(c / (power + 1), node, FACTOR_ADD);
    power = power << 1;
  }
  try(makeOdd(1 - c), node, SHIFT_REV);
  try(makeOdd(c + 1), node, SHIFT_SUB);
}◇
```

Macro referenced in scraps 8a, 11, 12a.

We need a slightly fancier version of **try** that will record the best cost and derivation.

⟨A third cut 9c⟩ ≡

```
static void try(int factor, Node *node, MulOp opcode)
{
  unsigned int cost = costs[opcode];
  Node *factor_node = find_sequence(factor);
  if (!node->parent || factor_node->cost + cost < node->cost) {
    node->cost = factor_node->cost + cost;
    node->parent = factor_node;
    node->opcode = opcode;
  }
}
◇
```

Macro defined by scraps 8a, 9c.
Macro never referenced.

3.4 Avoiding Redundant Searches

Up to this point, all our examples have involved enormously wasteful searches. In this section, we rewrite `find_sequence` as a "memo function" that records the results of all its intermediate invocations in a hash table. We store only odd numbers in the table since even numbers are always shifted immediately to become odd. In contrast to Bernstein, we also store negative numbers. This occasionally saves us a cycle. For example, Bernstein's approach to negative numbers would require 3 cycles for -3; ours requires only 2 cycles.

We'll use a form of overflow chaining to resolve collisions; therefore, we need to add a field to our definition of `Node`. The `next` field will always point to the next node in the same hash bucket.

⟨Other fields in `Node` 10a⟩ ≡
```
struct node *next;◇
```
Macro referenced in scrap 7c.

We'll use a simple modulo function to compute our hash. We define the size of the table to be fairly large, though it could probably be much smaller for most cases. Since we keep only odd numbers in the hash table, we must make sure that `HASH_SIZE` is odd; otherwise, half the slots will be wasted.

⟨Constant definitions 10b⟩ ≡
```
#define HASH_SIZE 511
◇
```
Macro defined by scraps 5a, 10b.
Macro referenced in scrap 3a.

Buckets in the table are simply pointers to nodes. Since each `Node` has a slot to hold a hash-collision chain, each pointer is basically to a list of nodes with the same hash value.

⟨Variable definitions 10c⟩ ≡
```
static Node *hash_table[HASH_SIZE];
◇
```
Macro defined by scraps 7b, 10c.
Macro referenced in scrap 3a.

The routine `lookup` looks for a node containing the value `n` in the hash table. If found, the node is returned. If not found, an appropriate node is created, initialized, added to the hash table, and returned.

⟨Function definitions 10d⟩ ≡
```
static Node *lookup(int c)
{
  int hash = abs(c) % HASH_SIZE;
  Node *node = hash_table[hash];
  while (node && node->value != c)
    node = node->next;
  if (!node)
    ⟨Create a new Node and add it to hash_table[hash] 10e⟩
  return node;
}
◇
```
Macro defined by scraps 3cd, 10d, 13ab, 15bcd, 16c, 17.
Macro referenced in scrap 3a.

⟨Create a new `Node` and add it to `hash_table[hash]` 10e⟩ ≡
```
{
  ⟨Create and initialize node 8b, ... ⟩
  node->next = hash_table[hash];
  hash_table[hash] = node;
}◇
```
Macro referenced in scrap 10d.

For the newest version of **find_sequence**, we get the appropriate node from the hash table, checking to see whether it's already been explored. If so, we simply return; otherwise, we explore the alternatives as usual. We'll assume that nodes for 1 and -1 have elready been initialized and added to the hash table.

⟨A fourth cut 11⟩ ≡

```
static Node *find_sequence(int c)
{
  Node *node = lookup(c);
  if (!node->parent) {
    if (c > 0)
      ⟨Handle the positive case 9a⟩
    else
      ⟨Handle the negative case 9b⟩
  }
  return node;
}
◇
```

Macro never referenced.

3.5 Pruning the Search

Sometimes, deep in a search, it becomes clear that we're pursuing a fruitless path. At that point, we'd like to abandon this particular line of exploration and consider other alternatives. By passing a `limit` parameter to each invocation of `find_sequence`, we can avoid searching too deeply in useless directions. The idea is quite similar to using *alpha-beta* cutoff to prune game search trees.

The `limit` parameter will specify how deep we're willing to search, where "depth" is in terms of the cycle-time costs of the target machine's operations. As we consider each alternative to handling a particular factor, we'll pass along the best known result as `limit`. Essentially, `limit` specifies the time to beat.

We'll assert, for a node `n`, that multiplication by `n->value` cannot be accomplished in less than `n->cost` cycles (more precisely, our approach will never find a shorter sequence). We'll further assert that if `n->parent` has been set, then `n->cost` will accurately reflect the cost of multiplying by `n->value`. On the other hand, if `n->parent` is still `NULL`, then we may or may not be able to multiply by `n->value` in `n->cost` cycles.

When we come across a node for the first time, we initialize it with a low cost that we know cannot be beat (for odd numbers greater than one, we can be sure they will cost more than a shift). We then explore the possiblities of generating it within `limit` cycles. If unsuccessful, the node's cost will be set to `limit`. Later searches may further raise the cost of the node, provided they are initiated with larger limits. Eventually, the cost may be raised to the point that a successful search is carried out. At this point, `parent` is set to point to the correct alternative.

Imagine we invoke `find_sequence` with arguments `c = 101` and `limit = 4`. The first time we look at the node for 101, its cost is initialized to 2 cycles, meaning we can't hope to synthesize it in less than 2 cycles. Since its cost is less than `limit` and we haven't proven that its cost is accurate (*i.e.,* `parent` is not set), we explore ways to generate 101. Since a possiblity must beat `limit` to be interesting, we set `cost` to 4. After considering all the alternatives, `cost` will remain equal to 4, since there is no way to generate a multiply by 101 in less than 4 cycles. Thus, later searches with limits of 4 or less will be terminated immediately. Searches with larger limits (at least 7) will eventually discover that a multiply by 101 can be accomplished in 6 cycles.

⟨The final version of find_sequence 12a⟩ ≡

```
static Node *find_sequence(int c, unsigned int limit)
{
  Node *node = lookup(c);
  if (!node->parent && node->cost < limit) {
    node->cost = limit;
    if (c > 0)
      ⟨Handle the positive case 9a⟩
    else
      ⟨Handle the negative case 9b⟩
  }
  return node;
}
◇
```

Macro referenced in scrap 15b.

The routine `try` explores one alternative. If that alternative can be generated cheaply enough, then it is recorded in `node` as the new best case.

⟨The final version of try 12b⟩ ≡

```
static void try(int factor, Node *node, MulOp opcode)
{
  unsigned int cost = costs[opcode];
  unsigned int limit = node->cost - cost;
  Node *factor_node = find_sequence(factor, limit);
  if (factor_node->parent && factor_node->cost < limit) {
    node->parent = factor_node;
    node->opcode = opcode;
    node->cost = factor_node->cost + cost;
  }
}
◇
```

Macro referenced in scrap 15b.

While creating a node, we set `cost` to more than `SHIFT_COST` since no odd value greater than 1 can be handled in a single shift; they all require at least two operations.

⟨Create and initialize node 12c⟩ ≡

```
node->cost = SHIFT_COST + 1;
◇
```

Macro defined by scraps 8b, 12c.
Macro referenced in scraps 8a, 10e.

4 Emitting Code

For testing purposes (and for fun), we define "code" emission routines that write a human-readable code sequence directly to `stdout`. In a real application, these would presumably be replaced by routines that inserted instructions into the control-flow graph. Nevertheless, the current version is still useful as an example of the required structure.

First, we need a routine that will emit a left-shift instruction, shifting `source` left until it equals `target`.

⟨Function definitions 13a⟩ ≡

```
static void emit_shift(int target, int source)
{
  int temp = source;
  unsigned int i = 0;
  do {
    temp <<= 1;
    i++;
  } while (target != temp);
  fprintf(stdout, "%d = %d << %d\n", target, source, i);
}
```

◇

Macro defined by scraps 3cd, 10d, 13ab, 15bcd, 16c, 17.
Macro referenced in scrap 3a.

The main code generation routine is recursive. Invoked on `node`, it switches upon `node->opcode`. The recursion is broken by the discovery of an `IDENTITY` node. Otherwise, `emit_code` is invoked recursively and the correct code is emitted for each case.

⟨Function definitions 13b⟩ ≡

```
static int emit_code(Node *node)
{
  int source;
  unsigned int shift;
  int target = node->value;
  switch (node->opcode) {
    ⟨Opcode cases 13c, ... ⟩
  }
  return target;
}
```

◇

Macro defined by scraps 3cd, 10d, 13ab, 15bcd, 16c, 17.
Macro referenced in scrap 3a.

The `IDENTITY` case ends the recursion.

⟨Opcode cases 13c⟩ ≡

```
case IDENTITY:
  break;
```

◇

Macro defined by scraps 13cd, 14abcde, 15a.
Macro referenced in scrap 13b.

`NEGATE` should only arise for -1; however, we'll handle it in a general fashion.

⟨Opcode cases 13d⟩ ≡

```
case NEGATE:
  source = emit_code(node->parent);
  fprintf(stdout, "%d = 0 - %d\n", target, source);
  break;
```

◇

Macro defined by scraps 13cd, 14abcde, 15a.
Macro referenced in scrap 13b.

The `SHIFT_ADD` case illustrates the general form of the remaining cases. Here, we generate two instructions that accomplish `target = (source << n) + 1`.

⟨Opcode cases 14a⟩ ≡
```
case SHIFT_ADD:
  source = emit_code(node->parent);
  emit_shift(target-1, source);
  fprintf(stdout, "%d = %d + 1\n", target, target-1);
  break;
```
◇

Macro defined by scraps 13cd, 14abcde, 15a.
Macro referenced in scrap 13b.

`SHIFT_SUB` produces `target = (source << n) - 1`.

⟨Opcode cases 14b⟩ ≡
```
case SHIFT_SUB:
  source = emit_code(node->parent);
  emit_shift(target+1, source);
  fprintf(stdout, "%d = %d - 1\n", target, target+1);
  break;
```
◇

Macro defined by scraps 13cd, 14abcde, 15a.
Macro referenced in scrap 13b.

`SHIFT_REV` produces `target = 1 - (source << n)`. It's typically produced when generating negative numbers (avoids the need for an additional negate instruction in many cases).

⟨Opcode cases 14c⟩ ≡
```
case SHIFT_REV:
  source = emit_code(node->parent);
  emit_shift(1-target, source);
  fprintf(stdout, "%d = 1 - %d\n", target, 1-target);
  break;
```
◇

Macro defined by scraps 13cd, 14abcde, 15a.
Macro referenced in scrap 13b.

`FACTOR_ADD` produces `target = (source << n) + source`, effectively multiplying `source` by $2^n + 1$.

⟨Opcode cases 14d⟩ ≡
```
case FACTOR_ADD:
  source = emit_code(node->parent);
  emit_shift(target-source, source);
  fprintf(stdout, "%d = %d + %d\n", target, target-source, source);
  break;
```
◇

Macro defined by scraps 13cd, 14abcde, 15a.
Macro referenced in scrap 13b.

`FACTOR_SUB` produces `target = (source << n) - source`, effectively multiplying `source` by $2^n - 1$.

⟨Opcode cases 14e⟩ ≡
```
case FACTOR_SUB:
  source = emit_code(node->parent);
  emit_shift(target+source, source);
  fprintf(stdout, "%d = %d - %d\n", target, target+source, source);
  break;
```
◇

Macro defined by scraps 13cd, 14abcde, 15a.
Macro referenced in scrap 13b.

FACTOR_REV produces target = source - (source << n), effectively multiplying source by $1 - 2^n$.

⟨Opcode cases 15a⟩ ≡
```
case FACTOR_REV:
  source = emit_code(node->parent);
  emit_shift(source-target, source);
  fprintf(stdout, "%d = %d - %d\n", target, source, source-target);
  break;
```
◇

Macro defined by scraps 13cd, 14abcde, 15a.
Macro referenced in scrap 13b.

5 The Driver Routine

We finally nail down the actual functions, discarding all the early attempts. We also need a prototype for find_sequence so that it can be referenced from try.

⟨Function definitions 15b⟩ ≡
```
static Node *find_sequence(int c, unsigned int limit);
⟨The final version of try 12b⟩
⟨The final version of find_sequence 12a⟩
```
◇

Macro defined by scraps 3cd, 10d, 13ab, 15bcd, 16c, 17.
Macro referenced in scrap 3a.

We need a *machine-dependent* function to estimate the cost of performing an integer multiply using the target machine's multiply instruction (or software routine).

The function given here is for the Motorola 601. It should return 5 cycles for 16-bit numbers and 9 cycles for larger numbers (*my interpretation of the manual may be wrong – it should be verified*). We might allow an extra cycle or two in the larger case to allow for the cost of loading the number into a register.

⟨Function definitions 15c⟩ ≡
```
static unsigned int estimate_cost(int c)
{
  if (c >= 0)
    if (c >= 65536) return 9;
    else return 5;
  else if (c <= -65536) return 9;
  else return 5;
}
```
◇

Macro defined by scraps 3cd, 10d, 13ab, 15bcd, 16c, 17.
Macro referenced in scrap 3a.

Finally, in best bottom-up fashion, we define the multiply routine. Since this routine will eventually be incorporated into our constant propagator, we can assert that multiply is never called with target equal to 0.

⟨Function definitions 15d⟩ ≡
```
void multiply(int target)
{
  unsigned int multiply_cost = estimate_cost(target);
  if (odd(target))
    ⟨Handle the (straightforward) odd case 16a⟩
  else
    ⟨Handle the (relatively complex) even case 16b⟩
}
```
◇

Macro defined by scraps 3cd, 10d, 13ab, 15bcd, 16c, 17.
Macro referenced in scrap 3a.

We call **find_sequence** and check to make sure the resulting sequence is cheap enough to beat a multiply instruction.

⟨Handle the (straightforward) odd case 16a⟩ ≡
```
{
  Node *result = find_sequence(target, multiply_cost);
  if (result->parent && result->cost < multiply_cost)
    (void) emit_code(result);
  else
    fprintf(stdout, "use multiply instruction\n");
}◇
```
Macro referenced in scrap 15d.

To handle even numbers, we immediately shift them until they are odd, then find the sequence computing the odd multiplication. Therefore, we need to account for the cost of the final shift.

⟨Handle the (relatively complex) even case 16b⟩ ≡
```
{
  Node *result = find_sequence(makeOdd(target), multiply_cost - SHIFT_COST);
  if (result->parent && result->cost + SHIFT_COST < multiply_cost) {
    int source = emit_code(result);
    emit_shift(target, source);
  }
  else
    fprintf(stdout, "use multiply instruction\n");
}◇
```
Macro referenced in scrap 15d.

6 Initialization

Before invoking **multiply**, the hash table must be created and initialized.

⟨Function definitions 16c⟩ ≡
```
void init_multiply(void)
{
  Node *node, *node1;
  unsigned int i;
  for (i=0; i<HASH_SIZE; i++) hash_table[i] = NULL;

  node1 = lookup(1);
  node1->parent = node1;        /* must not be NULL */
  node1->opcode = IDENTITY;
  node1->cost = 0;

  node = lookup(-1);
  node->parent = node1;
  node->opcode = NEGATE;
  node->cost = NEG_COST;
}
◇
```
Macro defined by scraps 3cd, 10d, 13ab, 15bcd, 16c, 17.
Macro referenced in scrap 3a.

7 A Dummy Main Routine

While this whole mess will eventually become part of our constant propagator, we'd like to play with it in the meantime. Here's a small `main` routine to support experimentation.

⟨Function definitions 17⟩ ≡
```
void main()
{
  int i;
  init_multiply();
  while (EOF != scanf("%d", &i))
    if (i) multiply(i);
  exit(0);
}
◇
```
Macro defined by scraps 3cd, 10d, 13ab, 15bcd, 16c, 17.
Macro referenced in scrap 3a.

8 Indices

We've defined indices for the scraps and certain useful identifiers.

8.1 Scraps

8.2 Identifiers

Underlined page numbers refer to definitions; other references are to uses.

References

[1] Robert Bernstein. Multiplication by integer constants. *Software – Practice and Experience*, 16(7):641–652, July 1986.