Lab 4 – Intro to Assembly
Group 8 – Abuke, Gaerlan, Sy, Taruc

(Mishka + Tobey + Ethan) 1.

```
       sall $2, %eax
```

- (shifts value in %eax left by 2 bits) → 2^2 = 4
- shifting is much faster and simpler over multiplication for powers of two
- sall (shift all) shift entire section by a constant (2) number of bits

(Tobey + Mishka + Ethan) 2.

```
       movl  -4(%rbp), %edx    edx=x
       movl  %edx, %eax        eax=x
       sall  $2, %eax          eax = x << 2 = x * 4
       addl  %edx, %eax        eax = 4x + x = 5x
       sall  $3, %eax          eax = 5x << 3 = 5x * 8 = 40x
       addl  %edx, %eax        eax = 40x + x = 41x
```

- the code that's generated is
    - shift %eax by 2 bits (x * (2^2) = 4x) and store it back in %eax
    - add the input x (%edx) to %eax (4x + x= 5x)
    - shift %eax by 3 bits (5x * (2^3) = 40x)
    - add the input x (%edx) to %eax (40x + x= 41x)
    - 41x was decomposed to $((2^2x + x)(2^3) + x)$

(Tobey + Mishka + Ethan) 3.

```
       movl    -4(%rbp), %edx        edx = x
       movl    %edx, %eax            eax = x
       sall    $6, %eax              eax = x << 6 = x * 64 = 64x
       subl    %edx, %eax            eax = 64x - x = 63x
```

- the code that's generated is
    - shift %eax by 6 bits (x * (2^6) = x * 64 = 64x) and store it back in %eax
    - subtract the input x (%edx) from %eax (64x - x = 63x)
    - Essentially, $63x = (2^6)x - x$

(Tobey + Mishka + Alinus + Ethan) 4.

```
       movl    -4(%rbp), %edx        edx = x
       movl    %edx, %eax            eax = x
       sall    $2, %eax              eax = x << 2 = x * 4 = 4x
       addl    %edx, %eax            eax = 4x + x = 5x
       negl    %eax                  eax = 0 - 5x = -5x
```

- the code that's generated is essentially 2 * 5 but negated by two's complement
    - shift %eax by 2 bits (x * (2^2) = 4x) and store it back in %eax
    - add the input x (%edx) to %eax (x + 4x = 5x)
    - negates long (does two''s complement) so the result is 2 * -5

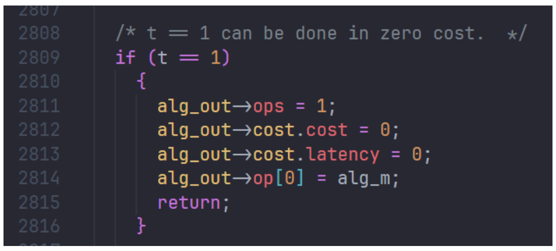
(Tobey + Mishka + Alinus) 5.

```
        imull   $61, %eax, %eax        eax = eax * 61
```

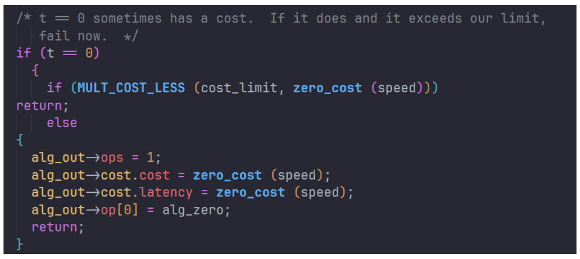
- Compiler just uses imull (integer multiply) directly instead of shifts and adds

The compiler optimizes using the instruction count; apparently, they use heuristics which determine if the cost of shifting/adding is lower than the cost of using imull directly.

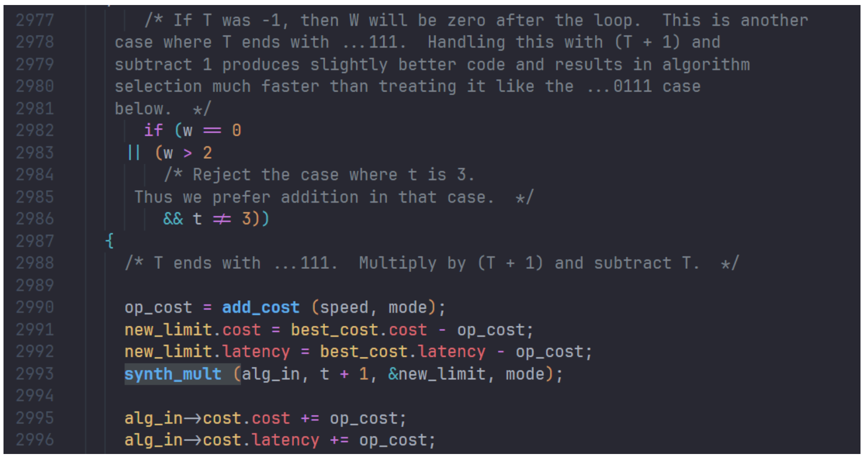
Looking at the algorithm, gcc tries to factor a number into factors of x * 2^n +- 1, and then factors that number recursively until they reach 1 or a number that is handled by a special case. The compiler also has multiple different cases that attempt to account for as many numbers as possible. This is done through the synth_mult function which is under the expmed.cc of the gcc compiler.



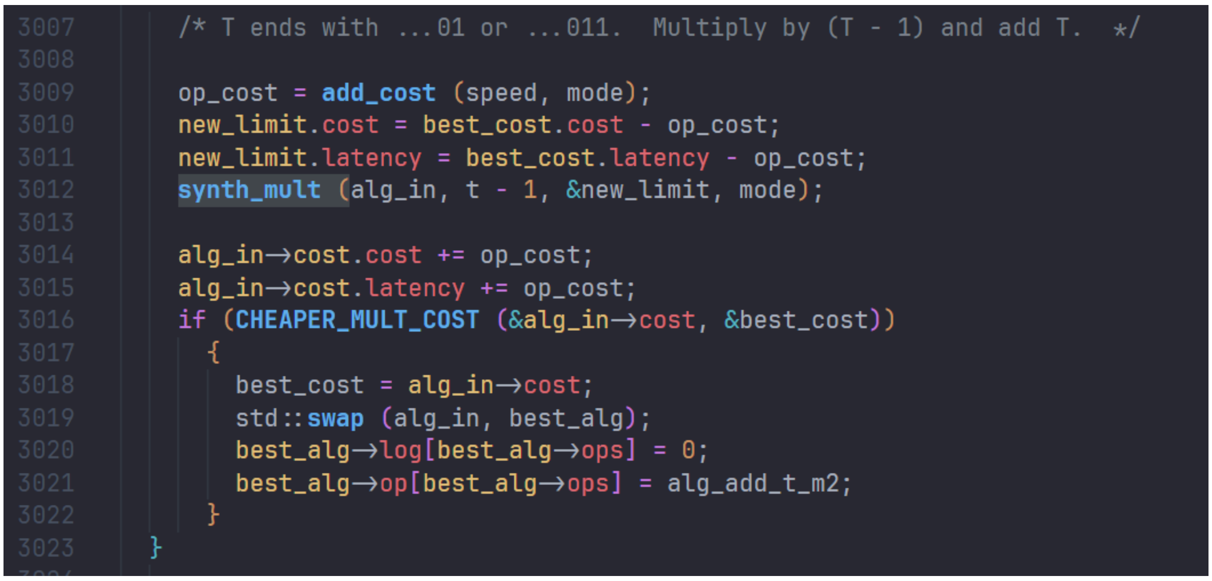
```
  /* t == 1 can be done in zero cost.  */
  if (t == 1)
    {
      alg_out->ops = 1;
      alg_out->cost.cost = 0;
      alg_out->cost.latency = 0;
      alg_out->op[0] = alg_m;
      return;
    }
```



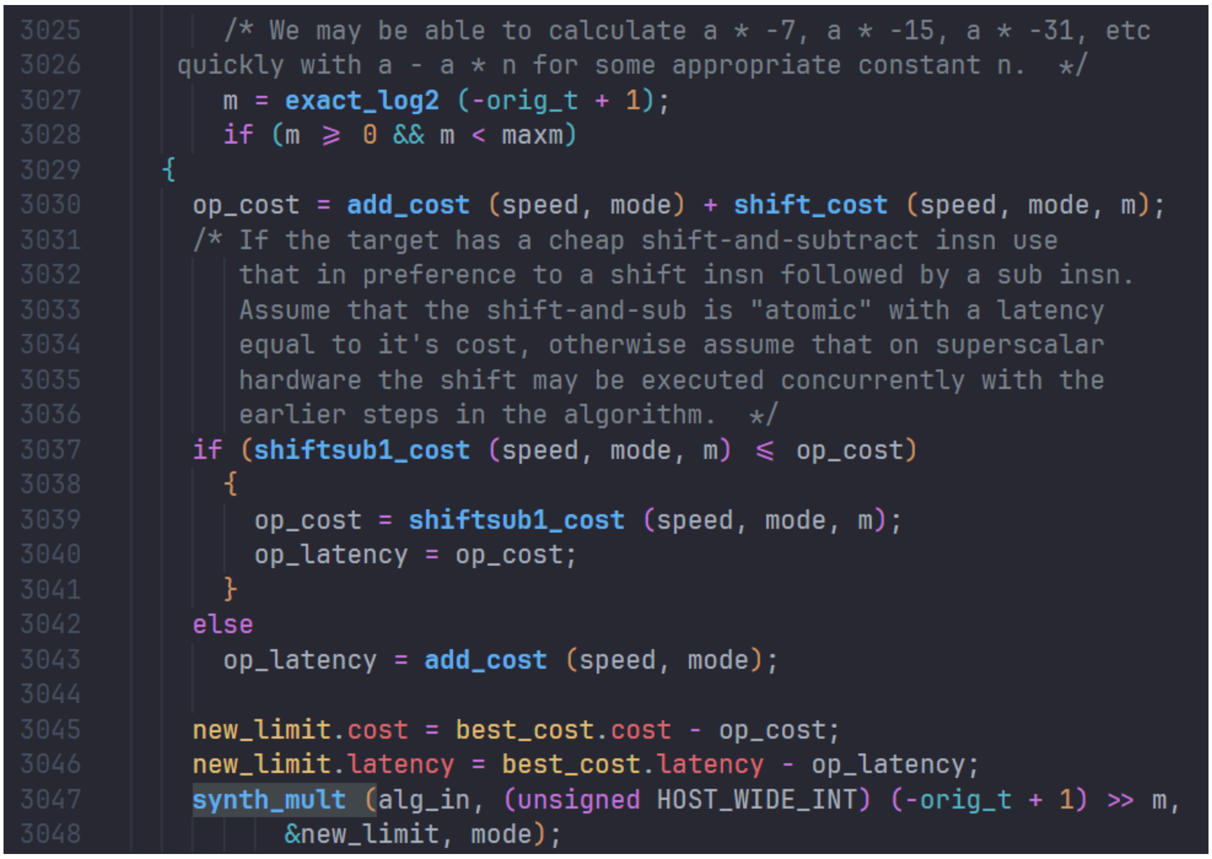
```
/* t == 0 sometimes has a cost.  If it does and it exceeds our limit,
     fail now.  */
if (t == 0)
  {
    if (MULT_COST_LESS (cost_limit, zero_cost (speed)))
return;
    else
{
  alg_out->ops = 1;
  alg_out->cost.cost = zero_cost (speed);
  alg_out->cost.latency = zero_cost (speed);
  alg_out->op[0] = alg_zero;
  return;
}
```



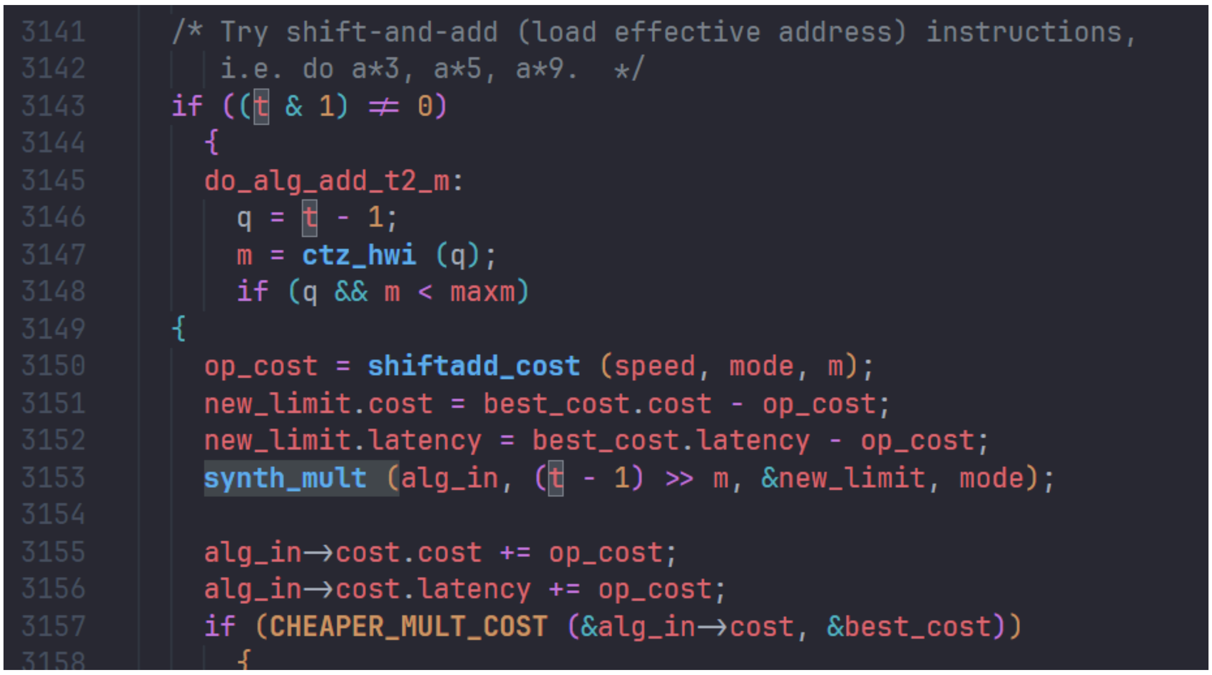
```
        /* If T was -1, then W will be zero after the loop.  This is another
     case where T ends with ...111.  Handling this with (T + 1) and
     subtract 1 produces slightly better code and results in algorithm
     selection much faster than treating it like the ...0111 case
     below.  */
        if (w == 0
      || (w > 2
          /* Reject the case where t is 3.
       Thus we prefer addition in that case.  */
          && t != 3))
    {
      /* T ends with ...111.  Multiply by (T + 1) and subtract T.  */

      op_cost = add_cost (speed, mode);
      new_limit.cost = best_cost.cost - op_cost;
      new_limit.latency = best_cost.latency - op_cost;
      synth_mult (alg_in, t + 1, &new_limit, mode);

      alg_in->cost.cost += op_cost;
      alg_in->cost.latency += op_cost;
```



```c
      /* T ends with ...01 or ...011.  Multiply by (T - 1) and add T.  */

      op_cost = add_cost (speed, mode);
      new_limit.cost = best_cost.cost - op_cost;
      new_limit.latency = best_cost.latency - op_cost;
      synth_mult (alg_in, t - 1, &new_limit, mode);

      alg_in->cost.cost += op_cost;
      alg_in->cost.latency += op_cost;
      if (CHEAPER_MULT_COST (&alg_in->cost, &best_cost))
        {
          best_cost = alg_in->cost;
          std::swap (alg_in, best_alg);
          best_alg->log[best_alg->ops] = 0;
          best_alg->op[best_alg->ops] = alg_add_t_m2;
        }
    }
```

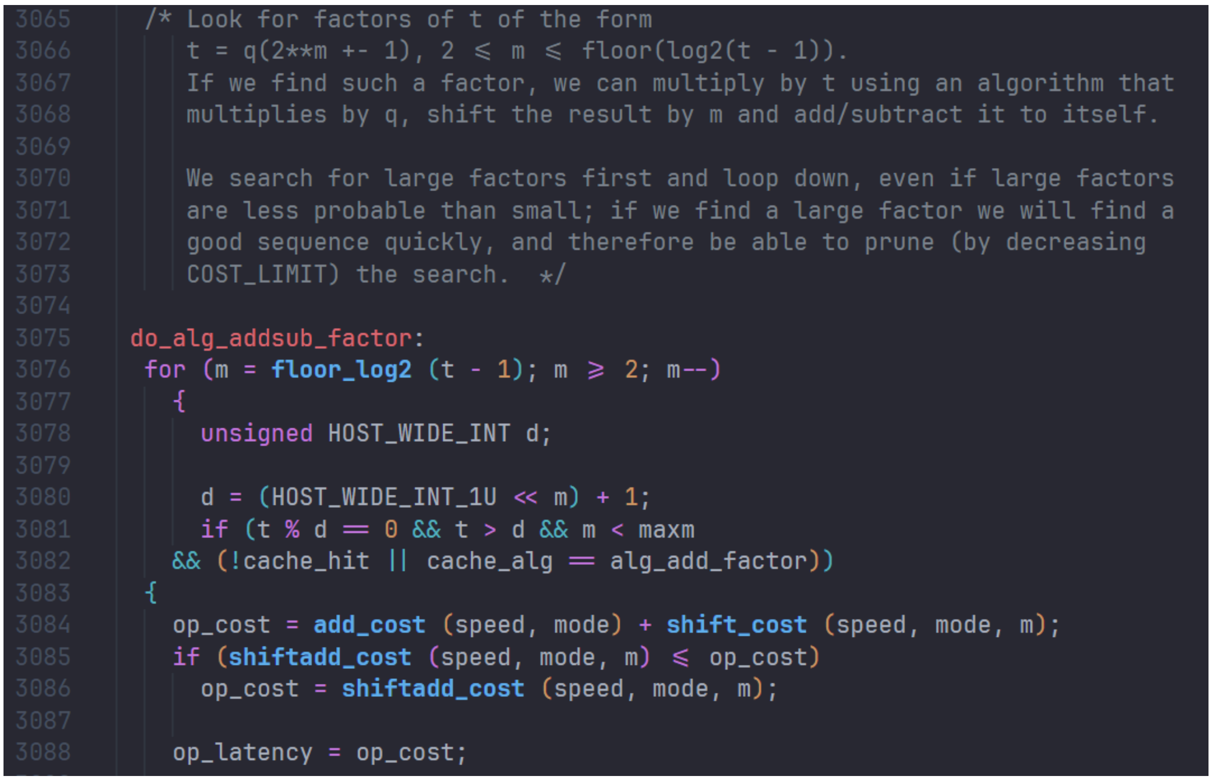


```c
        /* We may be able to calculate a * -7, a * -15, a * -31, etc
     quickly with a - a * n for some appropriate constant n.  */
        m = exact_log2 (-orig_t + 1);
        if (m >= 0 && m < maxm)
    {
      op_cost = add_cost (speed, mode) + shift_cost (speed, mode, m);
      /* If the target has a cheap shift-and-subtract insn use
         that in preference to a shift insn followed by a sub insn.
         Assume that the shift-and-sub is "atomic" with a latency
         equal to it's cost, otherwise assume that on superscalar
         hardware the shift may be executed concurrently with the
         earlier steps in the algorithm.  */
      if (shiftsub1_cost (speed, mode, m) <= op_cost)
        {
          op_cost = shiftsub1_cost (speed, mode, m);
          op_latency = op_cost;
        }
      else
        op_latency = add_cost (speed, mode);

      new_limit.cost = best_cost.cost - op_cost;
      new_limit.latency = best_cost.latency - op_latency;
      synth_mult (alg_in, (unsigned HOST_WIDE_INT) (-orig_t + 1) >> m,
            &new_limit, mode);
```



```
/* Try shift-and-add (load effective address) instructions,
   i.e. do a*3, a*5, a*9.  */
if ((t & 1) != 0)
  {
  do_alg_add_t2_m:
    q = t - 1;
    m = ctz_hwi (q);
    if (q && m < maxm)
  {
    op_cost = shiftadd_cost (speed, mode, m);
    new_limit.cost = best_cost.cost - op_cost;
    new_limit.latency = best_cost.latency - op_cost;
    synth_mult (alg_in, (t - 1) >> m, &new_limit, mode);

    alg_in->cost.cost += op_cost;
    alg_in->cost.latency += op_cost;
    if (CHEAPER_MULT_COST (&alg_in->cost, &best_cost))
      {
```

And then finally, the main algorithm, which is too large to show here, focuses on recursing through possible 2^n +- 1 scenarios



```
/* Look for factors of t of the form
   t = q(2**m +- 1), 2 <= m <= floor(log2(t - 1)).
   If we find such a factor, we can multiply by t using an algorithm that
   multiplies by q, shift the result by m and add/subtract it to itself.

   We search for large factors first and loop down, even if large factors
   are less probable than small; if we find a large factor we will find a
   good sequence quickly, and therefore be able to prune (by decreasing
   COST_LIMIT) the search.  */

do_alg_addsub_factor:
 for (m = floor_log2 (t - 1); m >= 2; m--)
   {
     unsigned HOST_WIDE_INT d;

     d = (HOST_WIDE_INT_1U << m) + 1;
     if (t % d == 0 && t > d && m < maxm
   && (!cache_hit || cache_alg == alg_add_factor))
 {
   op_cost = add_cost (speed, mode) + shift_cost (speed, mode, m);
   if (shiftadd_cost (speed, mode, m) <= op_cost)
     op_cost = shiftadd_cost (speed, mode, m);

   op_latency = op_cost;
```

We can try manually if 61 has good factors of 2^n, 2^n-1, and 2^n+1. We can get 61 with 31 + 15 + 15. Which are three shifts and subtracts added together for a total of 9. Other factors are around or take more operations than this. Since imul takes around 3 cycles. It is faster to just use imul.

(Tobey + Alinus + Ethan) 6.

Lots of using the leal function which seems to follow the format of *leal displacement(base register, offset register, scalar multiplier)*

where

*base register + (offset register * scalar multiplier) + displacement*

Which often compresses and performs the operations of these functions without dereferencing the addresses

Leal is being used as an arithmetic instruction rather than for memory access. It allows the compiler to combine multiple operations like addition and multiplication by constants into a single instruction, instead of using several separate shift and add instructions. leal is a faster way of doing addition + multiplication operations.
Leal works best for shift and add operations (see screenshot) 2^n + 1. So if the synth_mult stumbles upon a 2^n + 1 number, it can use leal for it instead of multiple shifts and adds.

In addition to that, when using the -0 flag, the compiler also removes unnecessary instructions such as extra moves between registers, and redundant computations. Some functions keep the same basic ALU instructions because they are already optimal, while others are rewritten to use leal since it can do the same work in fewer instructions.

The first case is now

```
	leal	0(,%rdi,4), %eax
```

Which roughly translates to 0 + (a * 4) + 0-> eax = a * 4

The second case is now

```
    leal	(%rdi,%rdi,4), %eax
    leal	(%rdi,%rax,8), %eax
```

Which translates to

a + (a * 4) + 0 → eax = a * 5
a + (a * 5 * 8) + 0 → eax = a * 41
(rax is the 64 bit extended eax)

The third case is actually still the same.

The fourth case is now

```
	leal	(%rdi,%rdi,4), %eax
	negl	%eax
```

Which translates to

a + (a * 4) + 0 → eax = a * 5
0 - a * 5 → eax = -a * 5

The fifth case is still using imull

```
	imull	$61, %edi, %eax
```