Assembly Code Analysis: Compiler Optimization for Multiplication

Kalpesh Parmar M.Eng, Cybersecurity
University of Maryland, College Park
kalpesh@umd.edu//UMD Directory ID – kalpesh
Course and section - ENPM691 0101

***Abstract*—This report investigates how compiler optimization levels affect the assembly generated for integer multiplication by constants. By analyzing GCC outputs for `-O0` vs. `-O2`, we show when the compiler replaces multiplications with shift/add sequences (and `lea`) and when it keeps `imul`. Screenshots from GDB verify register states and instruction flow.**

I. Purpose

Analyze the assembly code produced by GCC for constant multiplications and observe the impact of `-O0` vs. `-O2`.

II. Methodology

A C program `mul.c` multiplies an integer by constants 2, 12, 33, 53, 77, 95, and 112 with separate functions. Compilations:

- `-O0` (no optimization)
- `-O2` (aggressive optimization)

Flags: `-m32 -g -fno-stack-protector -no-pie -fno-pic`. Disassembly and stepping were done in GDB.

III. Results

Table I summarizes the *observed* (not theoretical) instructions from your runs.

TABLE I
Observed Assembly Instructions for `-O0` and `-O2`

Function	-O0 (Non-Optimized)	-O2 (Optimized)
`mulBy2`	`add %eax,%eax` (repeated)	`add %eax,%eax`
`mulBy12`	`add %eax,%eax; add %edx,%eax; shl $0x2,%eax`	`lea (%eax,%eax,2),%eax; shl $0x2,%eax`
`mulBy33`	`shl $0x5,%eax; add %edx,%eax`	`shl $0x5,%eax; add %edx,%eax`
`mulBy53`	`imul $0x35,%eax,%eax`	`imul $0x35,0x4(%esp),%eax`
`mulBy77`	`imul $0x4d,%eax,%eax`	`imul $0x4d,0x4(%esp),%eax`
`mulBy95`	`imul $0x5f,%eax,%eax`	`imul $0x5f,0x4(%esp),%eax`
`mulBy112`	`add %eax,%eax; add %edx,%eax; shl $0x2,%eax`	`imul $0x70,0x4(%esp),%eax`

IV. Analysis and Discussion

A. mulBy2

Obs. `-O0`: repeated `add %eax,%eax`; `-O2`: single add.
Reason. Multiplying by 2 is a left shift by one or self-add; `add`/`shl` are single-cycle and cheaper than `imul`. `-O2` removes redundant steps.

B. mulBy12

Obs. `-O0`: adds+shift; `-O2`: `lea` ($3x$) then `shl` by 2 to form $12x = (3x) \ll 2$.
Reason. `lea` computes $x + 2x$ efficiently without touching flags; followed by a shift it outperforms a general multiply.

C. mulBy33

Obs. Both levels use $(x \ll 5) + x$.
Reason. Already optimal: one shift + one add, so no gain from further transformation.

D. mulBy53, mulBy77, mulBy95

Obs. Both levels keep `imul`; `-O2` changes operand source to memory.
Reason. For irregular large constants, multiple shifts/adds would exceed the latency/throughput of a single `imul` on modern x86, and increase register pressure.

E. mulBy112

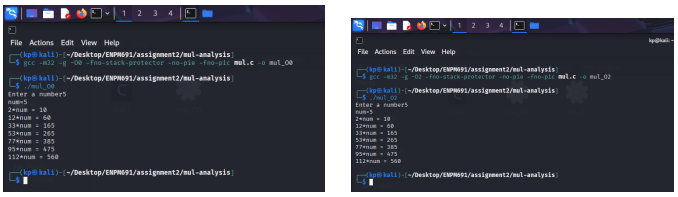
Obs. `-O0`: adds+shift; `-O2`: single `imul`.
Reason. Though $112x$ can be formed from several shifts/adds, the sequence needs at least three operations; a single `imul` is cheaper to schedule and simpler for the allocator.

F. General Insights

- Small constants/powers of two → `add`/`shl`.
- Mid-size constants (e.g., 12, 33) → `lea`+shift.
- Large/irregular constants (53, 77, 95, 112) → `imul`.
- `-O2` reduces instruction count and improves scheduling vs. verbose `-O0`.

V. Debugger Output and Register State

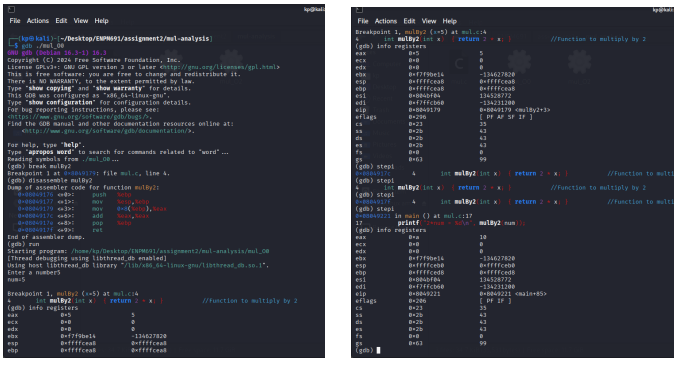
A. Executed Programs



(a) `-O0` (b) `-O2`

Fig. 1. Program output under `-O0` and `-O2`.

B. mulBy2



(a) O0 Step 1 (b) O0 Step 2

(c) O2 Step 1 (d) O2 Step 2

Fig. 2. `mulBy2` debugger/register output.

C. mulBy12

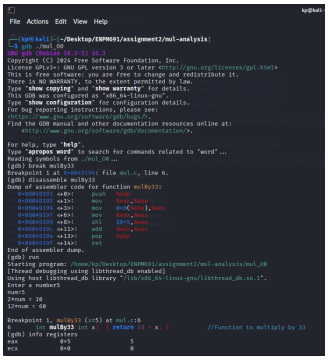
(a) O0 Step 1 (b) O0 Step 2

(c) O2 Step 1 (d) O2 Step 2

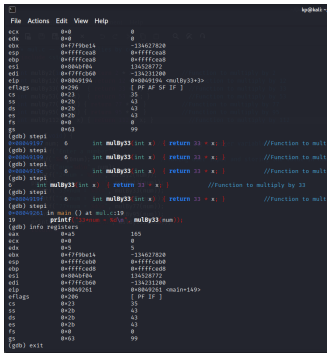
Fig. 3. `mulBy12` debugger/register output.

D. mulBy33

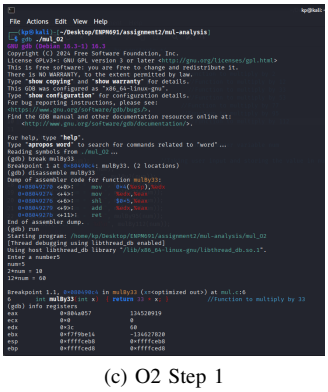
E. mulBy53



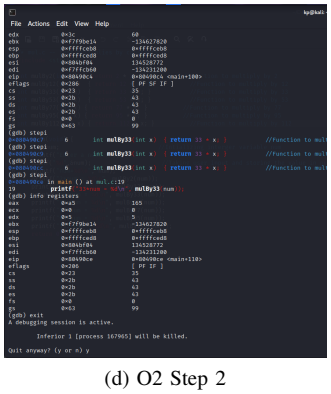
(a) O0 Step 1



(b) O0 Step 2

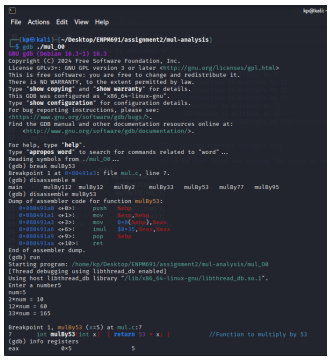


(c) O2 Step 1

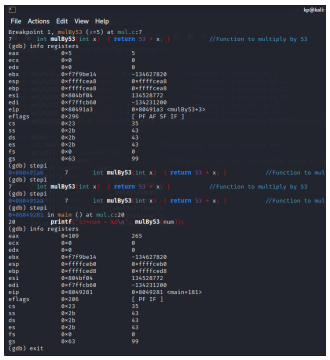


(d) O2 Step 2

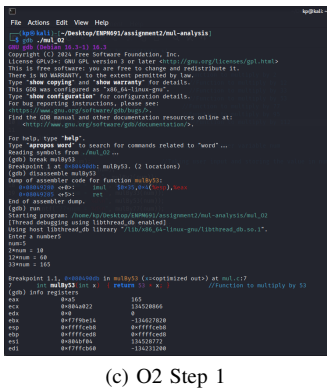
Fig. 4. `mulBy33` debugger/register output.



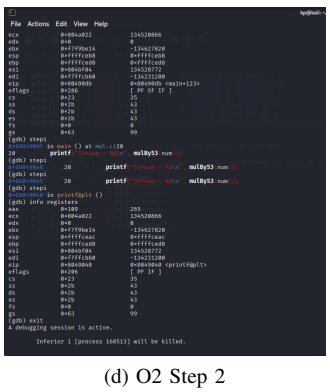
(a) O0 Step 1



(b) O0 Step 2



(c) O2 Step 1

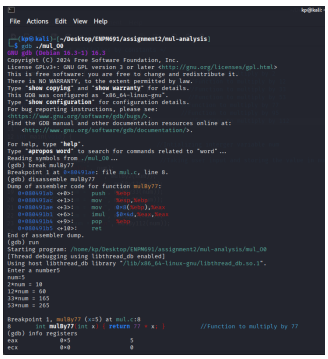


(d) O2 Step 2

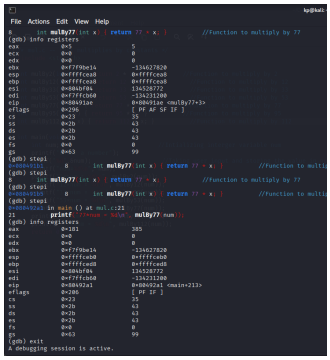
Fig. 5. `mulBy53` debugger/register output.

F. mulBy77

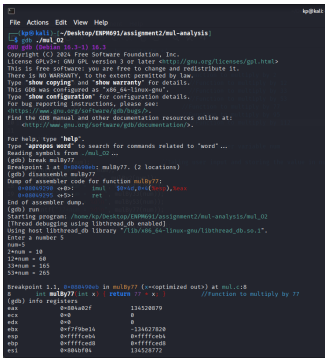
G. mulBy95



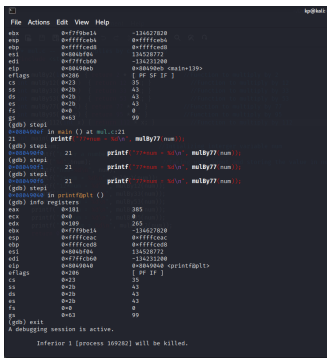
(a) O0 Step 1



(b) O0 Step 2



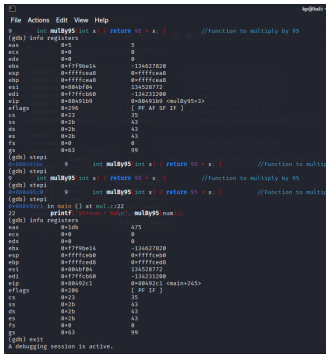
(c) O2 Step 1



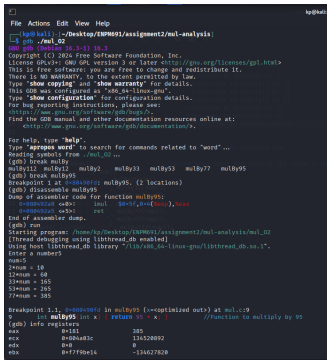
(d) O2 Step 2

Fig. 6. `mulBy77` debugger/register output.

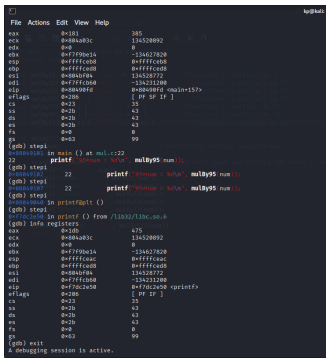
(a) O0 Step 1



(b) O0 Step 2



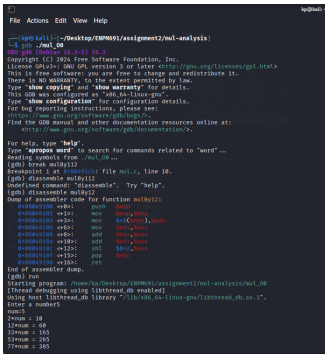
(c) O2 Step 1



(d) O2 Step 2

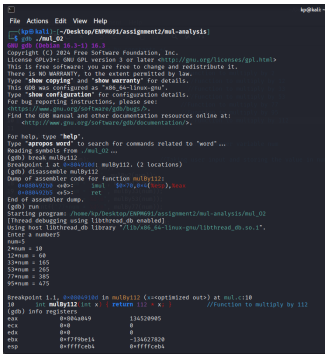
Fig. 7. `mulBy95` debugger/register output.

H. mulBy112



(a) O0 Step 1

(b) O0 Step 2



(c) O2 Step 1

(d) O2 Step 2

Fig. 8. `mulBy112` debugger/register output.

VI. Conclusion

Optimization replaces simple constant multiplies with `add`/`shl`/`lea` and keeps `imul` for irregular large constants. The observed sequences match expectations and illustrate the effect of `-O2` on instruction count and scheduling.

Appendix

A. Source Code (`mul.c`)

```c
/* mul.c -- test multiplies by constants */
#include <stdio.h>

int mulBy2(int x)   { return 2 * x; }
                //Function to multiply by 2
int mulBy12(int x)   { return 12 * x; }
                //Function to multiply by 12
int mulBy33(int x)   { return 33 * x; }
                //Function to multiply by 33
int mulBy53(int x)   { return 53 * x; }
                //Function to multiply by 53
int mulBy77(int x)  { return 77 * x; }
                //Function to multiply by 77
int mulBy95(int x)  { return 95 * x; }
                //Function to multiply by 95
int mulBy112(int x) { return 112 * x; }
                //Function to multiply by 112

int main(void) {
    int num;                                //
        Intializing interger variable num
    printf("Enter a number");
    scanf("%d",&num);                       //
        Taking user input and storing the
        value in num integer
    printf("num=%d\n", num);
    printf("2*num = %d\n", mulBy2(num));
    printf("12*num = %d\n", mulBy12(num));
    printf("33*num = %d\n", mulBy33(num));
    printf("53*num = %d\n", mulBy53(num));
    printf("77*num = %d\n", mulBy77(num));
    printf("95*num = %d\n", mulBy95(num));
    printf("112*num = %d\n", mulBy112(num));
    return 0;
}
```

B. System Configuration

- OS: Linux 64-bit
- Compiler: GCC

C. Compilation Commands

```
gcc -m32 -g -O0 -fno-stack-protector -no-pie -fno-
    pic mul.c -o mul_O0
gcc -m32 -g -O2 -fno-stack-protector -no-pie -fno-
    pic mul.c -o mul_O2
```

References

[1] R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 3rd ed., Pearson, 2016.

[2] A. S. Tanenbaum and T. Austin, *Structured Computer Organization*, 6th ed., Pearson, 2012.

[3] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed., Morgan Kaufmann, 2019.

[4] GCC Documentation, "Using the GNU Compiler Collection (GCC)," Free Software Foundation, 2025. Available: https://gcc.gnu.org/onlinedocs/