DoNotOptimize forms an important building block of any benchmarking library.

With functions like DoNotOptimize, We are asking the compiler to avoid optimizing the parameter of DoNotOptimize.

Eg;

```
r := fibonacci(i)
DoNotOptimize(r);
```

} benchmark loop

Google benchmax implements DoNotOptimize by using inline assembly using `asm volatile` which is honored by gcc & clang.

Format of asm volatile:

```
asm volatile (
        "assembly instruction" :
        "output operands" :
        "input operands" :
); "clobbers"
```

① assembly instruction can be empty,

② Output operands are the variables that are written to by assembly,

③ Input operands are inputs to assembly.

④ Clobbers like 'memory' tell the compiler that the assembly block may access other memory locations, not just the input operands, so the compiler should not reorder instructions around

asm block.

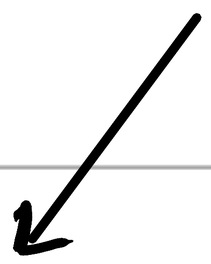
☆ Each operand uses a constraint string to tell the compiler what kind of resources (register/memory) are used to access the value. So compiler can generate appropriate instructions

Eg:

```
asm volatile("" : : "r"(value),
                    "memory");
```

assembly code & output operands are empty.

Input operand: "+r(value)"

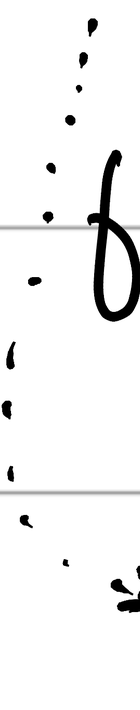


This has 3 pieces:

+: value may be read & written by

r: value is accessed from register

value: input



asm block

```cpp
template <class Tp>
void DoNotOptimize1(Tp& value) {
  asm volatile("" : "+r"(value) : : "memory");
}

int main() {
  int32_t x = 123;
  DoNotOptimize1(x);
  return x;  // Force the compiler to use x in some way
}
```

→ asm volatile

with

"+r(value)"

Generated assembly

```
        ret
void DoNotOptimize1<int>(int&):
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-8], rdi
        mov     rax, QWORD PTR [rbp-8]
        mov     eax, DWORD PTR [rax]
        mov     rdx, QWORD PTR [rbp-8]
        mov     DWORD PTR [rdx], eax
        nop
        pop     rbp
        ret
```

→ DoNotOptimize1

1) mov QWORD PTR [rbp-8] rdi

load first argument from rdi register to local stack, in this case stack frame contains the address of the input.

2) mov rax, QWORD PTR [rbp-8]

↑

load the address of argument in rax

↑ DoNOTOptimize

it taking a reference

3) mov eax, DWORD PTR [rax]

↑

load the actual value in eax.

from the address which was stored in rax.

4) mov &rdx, QWORD PTR [rbp-8]

load the address of argument in rdx

(Again... ↓ check 5)

5) mov DWORD PTR [rdx], eax

Write the value of eax into memory location identified by rdx.

(mov destination, source)

"+r(value)"

tells the compiler that the value needs to be made available in a register. Our asm volatile has an empty assembly instruction, the compiler will not parse that assembly instruction, it will just consider "+" in our "+r(value)" & consider that the assembly block may modify the value. Thus,

Compiler will not optimize value.

In the generated assembly, the register value is written back to the memory because the compiler is made to believe that the asm block may modify the value in register.

Hence, the compiler generates a mov instruction to write to memory.

```cpp
template <class Tp>
void DoNotOptimize(Tp& value) {
  asm volatile("" : "+m"(value) : : "memory");
}

int main() {
  int32_t x = 123;
  DoNotOptimize(x);
  return x;  // Force the compiler to use x in some way
}
```

→ using +m

```
        ret
void DoNotOptimize<int>(int&):
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-8], rdi
        mov     rax, QWORD PTR [rbp-8]
        mov     rdx, QWORD PTR [rbp-8]
        nop
        pop     rbp
        ret
```

→ Generated assembly.

+m means the value is accessed directly from memory, it need not be made available in register. + means asm block may modify the value in memory, hence compile should not optimize value.

Generated assembly.

① mov QWORD PTR [rbp-8], rdi

Move pointer argument in local stack frame.

② mov rax, QWORD PTR [rbp-8]

load the pointer (address of value) in rax.

③ Instruction 2 is regenerated with

a different destination register.

Actual value is not loaded, only pointer is; the compiler considers with "+m...", asm block can read/write the memory address. Compiler only needs to generate the code to provide the address of variable in memory.

+r : for small, copyable types

+m : for large types.

```
template <class Tp>
inline BENCHMARK_ALWAYS_INLINE
    typename std::enable_if<std::is_trivially_copyable<Tp>::value &&
                            (sizeof(Tp) <= sizeof(Tp*))>::type
    DoNotOptimize(Tp& value) {
  asm volatile("" : "+m,r"(value) : : "memory");
}

template <class Tp>
inline BENCHMARK_ALWAYS_INLINE
    typename std::enable_if<!std::is_trivially_copyable<Tp>::value ||
                            (sizeof(Tp) > sizeof(Tp*))>::type
    DoNotOptimize(Tp& value) {
  asm volatile("" : "+m"(value) : : "memory");
}
```

"+m,r" for copyable types

+m for non-copyable types

"+m,r" : let compiler decide, use register to pass value or use memory if needed.