__nop intrinsic

9.123 __nop intrinsic

This intrinsic inserts a `NOP` instruction or an equivalent code sequence into the instruction stream.

Syntax

```
void __nop(void);
```

Usage

The compiler does not optimize away the `NOP` instructions, except for normal unreachable code elimination. One `NOP` instruction is generated for each `__nop` intrinsic in the source.
ARMv6 and previous architectures do not have a `NOP` instruction, so the compiler generates a `MOV r0,r0` instruction instead.
In addition, `__nop` creates a special sequence point that prevents operations with side effects from moving past it under all circumstances. Normal sequence points allow operations with side effects past if they do not affect program behavior. Operations without side effects are not restricted by the intrinsic, and the compiler can move them past the sequence point. The `__schedule_barrier` intrinsic also creates this special sequence point, and at optimization level `-O0` emits two `NOP` instructions. These instructions are removed at other optimization levels.
Section 5.1.2.3 of the C standard defines operations with side effects as those that change the state of the execution environment. These operations:

- Access volatile objects.
- Modify a memory location.
- Modify a file.
- Call a function that does any of the above.

Examples

In the following example, the compiler ensures that the read from the volatile variable `x` is enclosed between two `NOP` instructions.

```
volatile int x;
int z;
int read_variable(int y)
{
    int i;
    int a = 0;
    __nop();
    a = x;
    __nop();
    return z + y;
}
```

If the `__nop` intrinsics are removed, and the compilation is performed at `-O3 -Otime` for `--cpu=Cortex-M3`, for example, then the compiler can schedule the read of the non-volatile variable `z` to be before the read of variable `x`.
In the following example, the compiler ensures that the write to variable `z` is enclosed between two `NOP` instructions.

```
int x;
int z;
int write_variable(int y)
{
    int i;
    for (i = 0; i < 10; i++)
    {
        __nop();
        z = y;
        __nop();
        x += y;
    }
    return z;
}
```

In this case, if the `__nop` intrinsics are removed, then with `-O3 -Otime --cpu=Cortex-A8`, the compiler can fold away the loop.
In the following example, because `pure_func` has no side effects, the compiler can move the call to it to outside of the loop. Still, the compiler ensures that the call to `func` is enclosed between two `NOP` instructions.

```
int func(int x);
int pure_func(int x) __pure;
int read(int x)
{
    int i;
    int a=0;
    for (i=0; i<10; i++)
    {
        __nop();
        a += pure_func(x) + func(x);
        __nop();
    }
    return a;
}
```

Note

- You can use the `__schedule_barrier` intrinsic to insert a scheduling barrier without generating a `NOP` instruction.
- In the examples above, the compiler would treat `__schedule_barrier` in the same way as `__nop`.

Related reference

9.13 __pure
9.133 __schedule_barrier intrinsic
3.4 Generic intrinsics
9.135 __sev intrinsic
9.144 __wfe intrinsic
9.145 __wfi intrinsic
9.146 __yield intrinsic

Related information

NOP