Inline Assembly/Examples

< Inline Assembly

The following is a collection of Inline Assembly functions so common that they should be useful to most OS developers using GCC. Other compilers may have intrinsic alternatives (see references). Notice how these functions are implemented using GNU extensions to the C language and that particular keywords may cause you trouble if you disable GNU extensions. You can still use the disabled keywords such as `asm` if you instead use the alternate keywords in the reserved namespace such as `__asm__`. Be wary of getting inline assembly just right: The compiler doesn't understand the assembly it emits and can potentially cause rare nasty bugs if you lie to the compiler.

WARNING!

Be aware, `asm("");` (no constaints, known as basic `asm` (https://gcc.gnu.org/onlinedocs/gcc/Basic-Asm.html)) and `asm("":);` (empty constraints, a form of extended `asm` (https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html)) are not the same! Among other differences, in basic `asm` you must prefix registers with a single percentage sign '%', in extended `asm` (even if the list of constraints is empty) you must use '%%' as a prefix in the assembly template string. If you get gcc error messages like "Error: bad register name '%%eax'", then you should insert a colon (':') before the closing parenthesis. Extended `asm` is generally preferred, though there are cases where basic `asm` is necessary (https://gcc.gnu.org/onlinedocs/gcc/Basic-Asm.html#Remarks) . Also note that this register prefix only applies to AT&T syntax assembly, gcc's default.

Contents

WRMSR
RDMSR
Memory Fence

Memory access

FAR_PEEKx

Read a 8/16/32-bit value at a given memory location using another segment than the default C data segment. Unfortunately there is no constraint for manipulating segment registers directly, so issuing the `mov <reg>, <segmentreg>` manually is required.

```
static inline uint32_t farpeekl(uint16_t sel, void* off)
{
    uint32_t ret;
    asm ( "push %%fs\n\t"
          "mov  %1, %%fs\n\t"
          "mov  %%fs:(%2), %0\n\t"
          "pop  %%fs"
          : "=r"(ret) : "g"(sel), "r"(off) );
    return ret;
}
```

FAR_POKEx

Write a 8/16/32-bit value to a segment:offset address too. Note that much like in farpeek, this version of farpoke saves and restore the segment register used for the access.

```
static inline void farpokeb(uint16_t sel, void* off, uint8_t v)
{
    asm ( "push %%fs\n\t"
          "mov  %0, %%fs\n\t"
          "movb %2, %%fs:(%1)\n\t"
          "pop %%fs"
          : : "g"(sel), "r"(off), "r"(v) );
    /* TODO: Should "memory" be in the clobber list here? */
}
```

I/O access

OUTx

Sends a 8/16/32-bit value on a I/O location. Traditional names are `outb`, `outw` and `outl` respectively. The `a` modifier enforces val to be placed in the eax register before the asm command is issued and `Nd` allows for one-byte constant values to be assembled as constants, freeing the edx register for other cases.

```
static inline void outb(uint16_t port, uint8_t val)
{
    __asm__ volatile ( "outb %b0, %w1" : : "a"(val), "Nd"(port) : "memory");
    /* There's an outb %al, $imm8 encoding, for compile-time constant port numbers that fit in 8b. (N constraint).
     * Wider immediate constants would be truncated at assemble-time (e.g. "i" constraint).
     * The  outb  %al, %dx  encoding is the only option for all other cases.
     * %1 expands to %dx because  port  is a uint16_t.  %w1 could be used if we had the port number a wider C type
```

```
*/
}
```

INx

Receives a 8/16/32-bit value from an I/O location. Traditional names are `inb`, `inw` and `inl` respectively.

```
static inline uint8_t inb(uint16_t port)
{
    uint8_t ret;
    __asm__ volatile ( "inb %w1, %b0"
                   : "=a"(ret)
                   : "Nd"(port)
                   : "memory");
    return ret;
}
```

IO_WAIT

Wait a very small amount of time (1 to 4 microseconds, generally). Useful for implementing a small delay for PIC remapping on old hardware or generally as a simple but imprecise wait.

You can do an IO operation on any unused port: the Linux kernel by default uses port 0x80, which is often used during POST to log information on the motherboard's hex display but almost always unused after boot.

```
static inline void io_wait(void)
{
    outb(0x80, 0);
}
```

Interrupt-related functions

Are interrupts enabled?

Returns a `true` boolean value if IRQs are enabled for the CPU.

```
static inline bool are_interrupts_enabled()
{
    unsigned long flags;
    asm volatile ( "pushf\n\t"
                   "pop %0"
                   : "=g"(flags) );
    return flags & (1 << 9);
}
```

Push/pop interrupt flag

Sometimes it is helpful to disable interrupts and then only re-enable them if they were disabled before. While the above function can be used for that, the below functions do it the same way regardless of the setting of the state of the IF:

```
static inline unsigned long save_irqdisable(void)
{
```

```
    unsigned long flags;
    asm volatile ("pushf\n\tcli\n\tpop %0" : "=r"(flags) : : "memory");
    return flags;
}

static inline void irqrestore(unsigned long flags)
{
    asm ("push %0\n\tpopf" : : "rm"(flags) : "memory","cc");
}

static void intended_usage(void)
{
    unsigned long f = save_irqdisable();
    do_whatever_without_irqs();
    irqrestore(f);
}
```

Memory clobber to force ordering, cc clobber because all condition codes are overwritten in the second example. No volatile in the second case since it has no outputs and is thus always volatile.

LIDT

Define a new interrupt table.

```
static inline void lidt(void* base, uint16_t size)
{
    // This function works in 32 and 64bit mode
    struct {
        uint16_t length;
        void*    base;
    } __attribute__((packed)) IDTR = { size, base };

    asm ( "lidt %0" : : "m"(IDTR) );  // let the compiler choose an addressing mode
}
```

CPU-related functions

CPUID

Request for CPU identification. See CPUID for more information.

Note: GCC has a `<cpuid.h>` header you should use instead of this.

```
static inline void cpuid(int code, uint32_t* a, uint32_t* d)
{
    asm volatile ( "cpuid" : "=a"(*a), "=d"(*d) : "0"(code) : "ebx", "ecx" );
}
```

RDTSC

Read the current value of the CPU's time-stamp counter and store into EDX:EAX. The time-stamp counter contains the amount of clock ticks that have elapsed since the last CPU reset. The value is stored in a 64-bit MSR and is incremented after each clock cycle.

```
static inline uint64_t rdtsc()
{
    uint64_t ret;
    asm volatile ( "rdtsc" : "=A"(ret) );
```

```
    return ret;
}
```

This can be used to find out how much time it takes to do certain functions, very useful for testing/benchmarking /etc. Note: This is only an approximation.

On x86_64, the "A" constraint expects to write into the "rdx:rax" registers instead of "edx:eax". So GCC can in fact optimize the above code by not setting RDX at all. You instead need to do it manually with bitshifting:

```
uint64_t rdtsc(void)
{
    uint32_t low, high;
    asm volatile("rdtsc":"=a"(low),"=d"(high));
    return ((uint64_t)high << 32) | low;
}
```

Read GCC Inline Assembly Machine Constraints (https://gcc.gnu.org/onlinedocs/gcc/Machine-Constraints.html#Machine-Constraints) for more details.

READ_CRx

Read the value in a control register.

```
static inline unsigned long read_cr0(void)
{
    unsigned long val;
    asm volatile ( "mov %%cr0, %0" : "=r"(val) );
    return val;
}
```

INVLPG

Invalidates the TLB (Translation Lookaside Buffer) for one specific virtual address. The next memory reference for the page will be forced to re-read PDE and PTE from main memory. Must be issued every time you update one of those tables. The `m` pointer points to a logical address, not a physical or virtual one: an offset for your ds segment.

```
static inline void invlpg(void* m)
{
    /* Clobber memory to avoid optimizer re-ordering access before invlpg, which may cause nasty bugs. */
    asm volatile ( "invlpg (%0)" : : "b"(m) : "memory" );
}
```

WRMSR

Write a 64-bit value to a MSR. The `A` constraint stands for concatenation of registers EAX and EDX.

```
static inline void wrmsr(uint32_t msr_id, uint64_t msr_value)
{
    asm volatile ( "wrmsr" : : "c" (msr_id), "A" (msr_value) );
}
```

On x86_64, this needs to be:

```
static inline void wrmsr(uint64_t msr, uint64_t value)
{
    uint32_t low = value & 0xFFFFFFFF;
    uint32_t high = value >> 32;
    asm volatile (
        "wrmsr"
        :
        : "c"(msr), "a"(low), "d"(high)
    );
}
```

RDMSR

Read a 64-bit value from a MSR. The `A` constraint stands for concatenation of registers EAX and EDX.

```
static inline uint64_t rdmsr(uint32_t msr_id)
{
    uint64_t msr_value;
    asm volatile ( "rdmsr" : "=A" (msr_value) : "c" (msr_id) );
    return msr_value;
}
```

On x86_64, this needs to be:

```
static inline uint64_t rdmsr(uint64_t msr)
{
    uint32_t low, high;
    asm volatile (
        "rdmsr"
        : "=a"(low), "=d"(high)
        : "c"(msr)
    );
    return ((uint64_t)high << 32) | low;
}
```

Memory Fence

Compiler memory fences can be used to ensure that data is accessed in the specified order by the compiler. A compiler fence is done by specifying memory as a clobber to any inline assembly, telling GCC that memory was supposedly modified, therefore requiring it to stay where it is. This is usually not enough for SMP systems.

```
#define fence() __asm__ volatile ("":::"memory")
```

Remember that `volatile` is **not** a replacement for a compiler memory fence.
