Writing better code with help from the compiler

Thiago Macieira

Qt Developer Days & LinuxCon Europe – October/2014

Who am I?

Example scenario

Interview question

You have 2 MB of data and you want to calculate how many bits are set, how would you do it?
Memory usage is not a constraint (within reason).

Approach 1: count the number of bits in each byte

```c
static unsigned char data[2*1024*1024];

int bitcount()
{
    int result = 0;
    for (int i = 0; i < sizeof(data); ++i) {
        unsigned char x = data[i];
        result += !!(x & 1);
        result += !!(x & 2);
        result += !!(x & 4);
        result += !!(x & 8);
        result += !!(x & 16);
        result += !!(x & 32);
        result += !!(x & 64);
        result += !!(x & 128);
    }
    return result;
}
```

```c
static unsigned char data[2*1024*1024];

int bitcount()
{
    int result = 0;
    for (int i = 0; i < sizeof(data); ++i) {
        unsigned char x = data[i];
        for ( ; x; ++result)
            x &= x - 1;
    }
    return result;
}
```

Approach 2: use a lookup table

```
static unsigned char data[2*1024*1024];
extern const ushort bitcount_table[65536];

int bitcount()
{
    int result = 0;
    for (int i = 0; i < sizeof(data); i += 2)
        result += bitcount_table[*(ushort*)(data + i)];
    return result;
}
```

My answer

- **Use the POPCNT instruction**
  - Added with the first Intel Core-i7 generation, Nehalem (SSE4.2, but separate CPUID bit)

How do you use the POPCNT instruction?

- **Write assembly**
- **Use the GCC intrinsic: `__builtin_popcount()`**
- **Use the Intel intrinsic: `_mm_popcnt_u32()`**

When can I use the instruction?

- **Use unconditionally!**
- **Check CPUID**
- **Ask the linker for help**
- **Check if surrounding code already requires a CPU that supports the feature anyway**

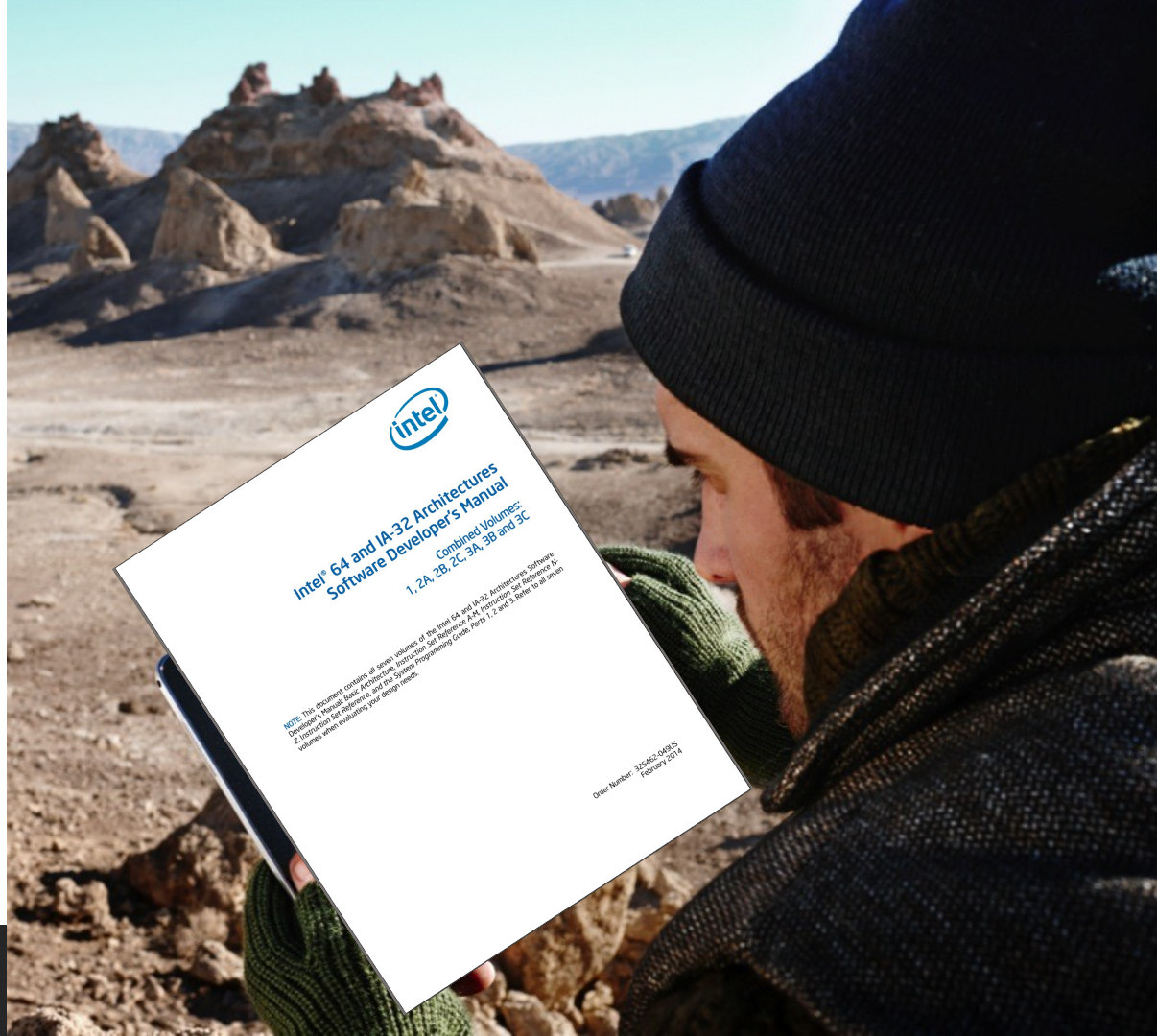
Choosing the solution

- **What affects the choice:**
  - CPUs it will run on
  - Compilers / toolchains it will be compiled with
  - Libraries you're using

Other architectures

- **Intrinsics exist for ARM and PowerPC too (Neon and Altivec)**
- **Not all compiler features work on those architectures yet**
- **But not discussed on this presentation**

Using intrinsics



Finding out which intrinsic to use

- **Use the SDM, Luke!**

POPCNT — Return the Count of Number of Bits Set to 1

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 0F B8 /r	POPCNT *r16, r/m16*	RM	Valid	Valid	POPCNT on *r/m16*
F3 0F B8 /r	POPCNT *r32, r/m32*	RM	Valid	Valid	POPCNT on *r/m32*
F3 REX.W 0F B8 /r	POPCNT *r64, r/m64*	RM	Valid	N.E.	POPCNT on *r/m64*

Intel C/C++ Compiler Intrinsic Equivalent

POPCNT: int _mm_popcnt_u32(unsigned int a);

POPCNT: int64_t _mm_popcnt_u64(unsigned __int64 a);

Examples using intrinsics

- **The population count**

```
static unsigned char data[2*1024*1024];
int bitcount()
{
    int result = 0;
    for (int i = 0; i < sizeof(data); i += 4)
        result +=
          __builtin_popcount(*(unsigned int*)(data + i));
    return result;
}
```

- **Calculating CRC32**

```
static unsigned char data[2*1024*1024];
int crc32()
{
    int h = 0;
    for (int i = 0; i < sizeof(data); i += 4)
        h = _mm_crc32_u32(h, *(unsigned int*)(data + i));
    return h;
}
```

Where are intrinsics allowed?

For all compilers: recent enough (e.g., GCC 4.7 for AVX2, 4.9 for AVX512F, etc.)

Compiler	Permitted usage
Microsoft Visual Studio	Anywhere, no special build options required
Intel C++ Compiler	
Clang	Anywhere, as long as code generation is enabled
GCC 4.8 or earlier	(e.g., `-mavx` / `-mavx2` / `-march=core-avx-i` / etc. active)
GCC 4.9	Code generation enabled; **or** functions decorated with `__attribute__((target("avx")))` (etc.)

How I solved this for Qt 5.4

- **Macro for testing with #if**
- **Macro that expands to `__attribute__((target(xxx))` (or empty)**

```
#if QT_COMPILER_SUPPORTS_HERE(SSE4_2)
QT_FUNCTION_TARGET(SSE4_2)
static uint crc32(const char *ptr, size_t len, uint h)
{
    // Implementation using _mm_crc32_u64 / u32 / u16 / u8 goes here
}
#else
static uint crc32(...)
{
    Q_UNREACHABLE();
    return 0;
}
#endif
```

Runtime dispatching

Runtime dispatching basics

1) **Detect CPU**

2) **Determine best implementation**

3) **Run it**

```
void function_sse2();
void function_plain();
void function()
{
    if (/* CPU supports SSE2 */)
        function_sse2();
    else
        function_plain();
}
```

With GCC 4.8:
(doesn't work with
Clang, ICC or MSVC)

```
void function_sse2();
void function_plain();
void function()
{
    if (__builtin_cpu_supports("sse2"))
        function_sse2();
    else
        function_plain();
}
```

Identifying the CPU

- **Running CPUID left as an exercise to the reader**
- **Just remember: cache the result**

```
extern int qt_cpu_features;
extern void qDetectCpuFeatures(void);

static inline int qCpuFeatures()
{
    int features = qt_cpu_features;
    if (Q_UNLIKELY(features == 0)) {
        qDetectCpuFeatures();
        features = qt_cpu_features;
    }
    return features;
}
```

CPUID goes here

Checking surrounding code

thiago : zsh – Konsole

```
thiago@tjmaciei-mobl4 ~ $ cd /tmp
thiago@tjmaciei-mobl4 /tmp $ cd -
~
thiago@tjmaciei-mobl4 ~ $ comm -13 <(gcc-4.9 -m32 -dM -E -xc /dev/null | sort) <(gcc-4.9 -m32 -dM -E -m
arch=haswell -xc /dev/null | sort)
#define __AES__ 1
#define __AVX__ 1
#define __AVX2__ 1
#define __BIGGEST_ALIGNMENT__ 32
#define __BMI__ 1
#define __BMI2__ 1
#define __core_avx2 1
#define __core_avx2__ 1
#define __F16C__ 1
#define __FMA__ 1
#define __FSGSBASE__ 1
#define __FXSR__ 1
#define __haswell 1
#define __haswell__ 1
#define __LZCNT__ 1
#define __MMX__ 1
#define __PCLMUL__ 1
#define __POPCNT__ 1
#define __RDRND__ 1
#define __SSE__ 1
#define __SSE2__ 1
#define __SSE3__ 1
#define __SSE4_1__ 1
#define __SSE4_2__ 1
#define __SSSE3__ 1
#define __tune_core_avx2__ 1
#define __tune_haswell__ 1
#define __XSAVE__ 1
#define __XSAVEOPT__ 1
thiago@tjmaciei-mobl4 ~ $
```

Putting it together

- **Result on 64-bit: unconditional call to the SSE2 version**

```
void function_sse2();
void function_plain();
void function()
{
    if (qCpuHasFeature(SSE2))
        function_sse2();
    else
        function_plain();
}
```

Asking the linker and dynamic linker for help

- **Requires:**
  - Glibc 2.11.1, Binutils 2.20.1, GCC 4.8 / ICC 14.0
  - Not supported with Clang or on Android (due to Bionic)

Magic goes here

```
void *memcpy(void *, const void *, size_t)
    __attribute__((ifunc("resolve_memcpy")));

void *memcpy_avx(void *, const void *, size_t);
void *memcpy_sse2(void *, const void *, size_t);
static void *(*resolve_memcpy(void))(void *, const void *, size_t)
{
    return qCpuHasFeature(AVX) ? memcpy_avx : memcpy_sse2;
}
```

GCC 4.9 auto-dispatcher (a.k.a. “Function Multi Versioning”)

- **C++ only!**

```
__attribute__((target("popcnt")))
int bitcount()
{
    int result = 0;
    for (int i = 0; i < sizeof(data); i += 4)
        result += __builtin_popcount(*(uint*)(data + i));
    return result;
}

__attribute__((target("default")))
int bitcount()
{
    int result = 0;
    for (int i = 0; i < sizeof(data); i += 2)
        result += bitcount_table[*(ushort*)(data + i)];
    return result;
}
```

Finding better answers to interview questions

- **“How would you write a function that returns a 32-bit random number?”**
- **“How would you zero-extend a block of data from 8- to 16-bit?”**
- **“How do you calculate the next power of 2 for a given non-zero integer?”**

```
uint32 nextPowerOfTwo(uint32 v)
{
    v--;
    v |= v >> 1;
    v |= v >> 2;
    v |= v >> 4;
    v |= v >> 8;
    v |= v >> 16;
    ++v;
    return v;
}
```

Better answer

```
uint32 nextPowerOfTwo_x86(uint32 v)
{
    int idx = _bit_scan_reverse(v);
    return 2U << idx;
}
```

Summary

- **Learn from the SDM: use intrinsics**
- **Check the CPU at compile time, run time and dispatch**
- **Use library, compiler and linker tools**

Zero-extending from 8- to 16-bit

- **Highly parallelisable**
  - No inter-element dependencies
- **Used in Latin-1 to UTF-16 conversion**

```
while (size--)
    *dst++ = (uchar)*str++;
```

Left to the whims of the compiler (-O3)

```
GCC 4.8
    2d8c:     movdqu    (%rsi,%rax,1),%xmm1
    2d91:     add       $0x1,%r10
    2d95:     movdqa    %xmm1,%xmm3
    2d99:     punpckhbw %xmm0,%xmm1
    2d9d:     punpcklbw %xmm0,%xmm3
    2da1:     movdqu   %xmm1,0x10(%rdi,%rax,2)
    2da7:     movdqu    %xmm3,(%rdi,%rax,2)
    2dac:     add       $0x10,%rax
    2db0:     cmp       %r9,%r10
    2db3:     jb        2d8c
```

```
ICC 14
     7d3:     movq      (%r8,%rsi,1),%xmm1
     7d9:     punpcklbw %xmm0,%xmm1
     7dd:     movdqa    %xmm1,(%rdi,%r8,2)
     7e3:     add       $0x8,%r8
     7e7:     cmp       %rax,%r8
     7ea:     jb        7d3
```

```
Clang 3.4
    2150:     movq      (%rsi,%rcx,1),%xmm1
    2155:     punpcklbw %xmm0,%xmm1
    2159:     movq      0x8(%rsi,%rcx,1),%xmm2
    215f:     punpcklbw %xmm0,%xmm2
    2163:     pand      %xmm0,%xmm1
    2167:     pand      %xmm0,%xmm2
    216b:     movdqu    %xmm1,(%rdi,%rcx,2)
    2170:     movdqu    %xmm2,0x10(%rdi,%rcx,2)
    2176:     add       $0x10,%rcx
    217a:     cmp       %rcx,%r9
    217d:     jne       2150
```

Left to the whims of the compiler (-O3 -mavx2)

```
GCC 4.9
    2bb6:     vmovdqu    (%rsi,%rdi,1),%ymm0
    2bbb:     add        $0x1,%r11
    2bbf:     vpmovzxbw  %xmm0,%ymm1
    2bc4:     vextracti128 $0x1,%ymm0,%xmm0
    2bca:     vpmovzxbw  %xmm0,%ymm0
    2bcf:     vmovdqa    %ymm1,(%rbx,%rdi,2)
    2bd4:     vmovdqa    %ymm0,0x20(%rbx,%rdi,2)
    2bda:     add        $0x20,%rdi
    2bde:     cmp        %r11,%rax
    2be1:     ja         2bb6
```

```
ICC 14
     7dc:     vpmovzxbw  (%r8,%rsi,1),%ymm0
     7e2:     vmovdqu    %ymm0,(%rdi,%r8,2)
     7e8:     add        $0x10,%r8
     7ec:     cmp        %rax,%r8
     7ef:     jb         7dc
```

```
Clang 3.4
    21a0:     vmovdqu    -0x20(%rsi),%xmm0
    21a5:     vmovdqu    -0x10(%rsi),%xmm1
    21aa:     vmovdqu    (%rsi),%xmm2
    21ae:     vpmovzxbw  %xmm0,%ymm0
    21b3:     vpmovzxbw  %xmm1,%ymm1
    21b8:     vpmovzxbw  %xmm2,%ymm2
    21bd:     vmovdqu    %ymm0,-0x40(%rdi)
    21c2:     vmovdqu    %ymm1,-0x20(%rdi)
    21c7:     vmovdqu    %ymm2,(%rdi)
    21cb:     add        $0x60,%rdi
    21cf:     add        $0x30,%rsi
    21d3:     add     $0xffffffffffffffd0,%rcx
    21d7:     cmp        %rcx,%r8
    21da:     jne        21a0
```

Helping out the compiler

- **GCC's implementation was the best with SSE2**
  - ICC produces better code for AVX2
- **Let's rewrite using intrinsics**

```
const char *e = str + size;
qptrdiff offset = 0;
const __m128i nullMask = _mm_set1_epi32(0);
// we're going to read str[offset..offset+15] (16 bytes)
for ( ; str + offset + 15 < e; offset += 16) {
    const __m128i chunk = _mm_loadu_si128((__m128i*)(str + offset)); // load 16 bytes

    // unpack the first 8 bytes, padding with zeros
    const __m128i firstHalf = _mm_unpacklo_epi8(chunk, nullMask);
    _mm_storeu_si128((__m128i*)(dst + offset), firstHalf); // store 16 bytes

    // unpack the last 8 bytes, padding with zeros
    const __m128i secondHalf = _mm_unpackhi_epi8(chunk, nullMask);
    _mm_storeu_si128((__m128i*)(dst + offset + 8), secondHalf); // store next 16 bytes
}
```

Code generated with the intrinsics

```
Before
    2d8c:     movdqu    (%rsi,%rax,1),%xmm1
    2d91:     add       $0x1,%r10
    2d95:     movdqa    %xmm1,%xmm3
    2d99:     punpckhbw %xmm0,%xmm1
    2d9d:     punpcklbw %xmm0,%xmm3
    2da1:     movdqu   %xmm1,0x10(%rdi,%rax,2)
    2da7:     movdqu    %xmm3,(%rdi,%rax,2)
    2dac:     add       $0x10,%rax
    2db0:     cmp       %r9,%r10
    2db3:     jb        2d8c
```

```
After
    2d70:     movdqu    (%rsi,%rcx,1),%xmm0
    2d75:     add       $0x10,%rax
    2d79:     add       %rcx,%rcx
    2d7c:     cmp       %r8,%rax
    2d7f:     movdqa    %xmm0,%xmm2
    2d83:     punpckhbw %xmm1,%xmm0
    2d87:     punpcklbw %xmm1,%xmm2
    2d8b:     movdqu   %xmm0,0x10(%rdi,%rcx,1)
    2d91:     movdqu    %xmm2,(%rdi,%rcx,1)
    2d96:     mov       %rax,%rcx
    2d99:     jne       2d70
```

Better or worse?

Extending to AVX2 support

```
    const char *e = str + size;
    qptrdiff offset = 0;
    // we're going to read str[offset..offset+15] (16 bytes)
    for ( ; str + offset + 15 < e; offset += 16) {
        const __m128i chunk = _mm_loadu_si128((__m128i*)(str + offset)); // load 16 bytes
#ifdef __AVX2__
        // zero extend to an YMM register
        const __m256i extended = _mm256_cvtepu8_epi16(chunk);
        // store 32 bytes
        _mm256_storeu_si256((__m256i*)(dst + offset), extended);
#else
        const __m128i nullMask = _mm_set1_epi32(0);

        // unpack the first 8 bytes, padding with zeros
        const __m128i firstHalf = _mm_unpacklo_epi8(chunk, nullMask);
        _mm_storeu_si128((__m128i*)(dst + offset), firstHalf); // store 16 bytes

        // unpack the last 8 bytes, padding with zeros
        const __m128i secondHalf = _mm_unpackhi_epi8 (chunk, nullMask);
        _mm_storeu_si128((__m128i*)(dst + offset + 8), secondHalf); // store next 16 bytes
#endif
    }
```

Thiago Macieira
thiago.macieira@intel.com
http://google.com/+ThiagoMacieira