Microarchitecture, Performance Measurement, Microoptimizations

Monitor U

Ben Serebrin

3/24/11

Topics

- Performance measurement
  - RDTSC tricks, issues
  - Performance counters
  - Latency
  - Switch statements
- Microoptimizations

What's the cost of foo?

```
start = RDTSC()
for (i = 0; i < iters; i++) {
    foo();
}
end = RDTSC();
fooCost = (end - start) / iters;
```

- `iters` must be "large enough," at least several Reorder buffers worth of instructions in the inner loop, to avoid `end = RDTSC()` being executed too early, and to amortize the cost of the loop.
- Better: pick `iters` so we run at least a few million cycles, but are significantly less than a 1 ms context switch time.

Better: What's the cost of foo?

```
start = RDTSC()
for (i = 0; i < iters; i++) {
}
end = RDTSC();
nopCost = (end – start) / iters;

start = RDTSC()
for (i = 0; i < iters; i++) {
    foo();
}
end = RDTSC();
fooCost = (end – start) / iters - nopCost;
```

- Better still: run multiple iterations and statistically discard outliers.

Timing: Branch prediction introduces difficulty

- If foo() contains branches, we're unfairly heating up the branch predictor. We could unroll the loop (but not much more than the L1 or L2 cache size) and run it twice, discarding the first result.

RDTSC Out of orderness

- RDTSC has no input dependencies; the processor is free to execute it earlier than older dependency chains.
  - RDPMC has only a dependence on ECX, similar issues apply
- RDTSCP is almost the right answer: defined as waiting for all older instructions to execute, but newer instructions can execute before RDTSCP.
  - Good enough for long-running (more than a few ROB-fuls) operations
  - Else, serialize after start=RDTSCP() with MFENCE.

Did you measure the right thing? Repeat rate, latency, parallelism

- Measuring a repeated set of serially-dependent instructions will provide the repeat rate and latency, but will not show if the pipe is fully occupied.
- Other work might be able to fit in the pipe.
- Costs aren't necessarily additive.

`UNROLL100(asm("div %eax, %eax"))` has the same runtime as

`UNROLL100(asm("add %ebx, %ebx; div %eax, %eax"))`

How to measure a piecewise event

- Looking at a larger scale, there are events that require piecewise processing over time.
  - Eg. Guest IPI (interrupt) handling has at least 3 parts:
    1. Guest writes ICRLO to send IPI; VMEXIT; Monitor decode work; send monitor action IPI
    2. Other core VMEXITs, consumes monitor action, injects interrupt, VMRUN
    3. Other core writes EOI, VMEXIT, decode work, APIC Emulate, VMRUN
- We need to start and stop the stopwatch for each of these three epochs, and avoid including external costs. Monitor action consumption could work on other actions; need to subtract.

Microoptimizations

- We can optimize for some subset of:
  - Code space
  - Speed
  - Branches vs. execution bandwidth
  - Register pressure
  - Memory loads and cache misses
  - Use of interesting x86 functions
  - ISA generational compatibility
  - Expressiveness to compiler
  - Legibility
  - Fun
- Always consider if a microoptimization is worthwhile.

cmov vs. if

- x86 has a predicated move instruction, `cmov`.
  - It's limited and non-orthogonal, of course. `cmov` can only move from register/memory to register, but control can be any of the 16 conditions `jcc` uses. (Move from immediate would be very useful!)

  `cmovcc %eax, %ebx` is equivalent to

  ```
  %ebx = cc ? %eax : %ebx;
  ```

  - Another useful but nonorthogonal instruction, `setcc`, sets any byte (register or memory) based on condition codes

  `setcc %al` is equivalent to

  ```
  %al = cc ? 1 : 0;
  ```

- When is `cmov` better than `if`? If branch predictions are warm, `if` that produces branches is better. But don't test your code in a loop because the branch predictor will warm up!
- Test a macrobenchmark and find out…
- How do you induce gcc to produce cmov without using inline assembly? Sometimes ?: will do it but not always.

UNLIKELY() and LIKELY()

- Macros around gcc __builtin_expect() that serve as hints to the compiler for conditionals.
  - Syntactically transparent.

```
if (LIKELY(j == 3)) { blah(); } else { foo(); }
```

- The not-likely code is usually moved out of line, to the end of the function
  - Increases Instruction Cache density of hot code
  - Most forward branches (i.e. the jcc to the out-of-line code) are predicted not taken, so the code falls through to the likely path
- LIKELY, UNLIKELY can be useful in moderation. Be sure your static prediction is actually correct!

Switch statements

- Switch statements with many cases often are converted into indirect jumps through a lookup table. This may not be a good thing for lukewarm-to-cool code.
  - The lookup table is data access that can't always be prefetched
  - I saved ~100 cycles converting a switch(APIC interrupt type) into an if-chain with LIKELY() for the 3 most common indices.

```
if (LIKELY(type == A)) …
else if (LIKELY(type == B)) …
else if (LIKELY(type == C)) …
else if (UNLIKELY(type == D) …
```

  - Callstack profiling showed a lot of hits on the jmp indirect of the original code, but it's not very obvious that the lookup table cache miss was at fault.

My favorite instructions: bit manipulations

`(LOCK) bts, btr, btc`: Bit test and {set, reset, complement}

{Atomically} set/reset/toggle bit N (specified in 1-byte immediate or register) from target register or memory location, and set the Carry Flag to the original bit's value.

The memory-target, register source flavor can take a signed bit offset of any size and give you free byte and bit indexing arithmetic.

Eg: `mov $0x1042, %ecx; bts %ecx, (0x1000)` sets the 0x1042th bit after byte address 0x1000.

When to use BTS

- Set a variable-bit offset (set bit `N` of var `foo`: `foo |= 1 << N;`)

  gcc emits 9 bytes: (`%eax` contains `foo`, `%cl` contains `N`)

  ```
  mov $0x1, %edx
  shl %cl, %edx
  or %eax, %edx
  ```

- SetBit32() emits 3 bytes:

  ```
  bts %cl, %eax
  ```

  See SetBit32(), SetBit64(), ClearBit{32,64}() in vm_basic_asm.h.

BTS vs. constant OR

- foo |= 0x1000 is 3 bytes: `or $0x10, %ah`
- foo |= 0x10000 is 5 bytes: `or $0x10000, %eax`
- Use BTS/SetBit32.

- More than 1 bit:
- foo |= 0x3000 is 3 bytes: `or $0x30, %ah`
- foo |= 0x30000 is 5 bytes: `or $0x30000, %eax`
- Don't use BTS.

- But look at objdump to see if gcc emits crummy code (like extraneous mov %eax, %edi); sometimes the intrinsics force it to be cleaner.

Testing index of highest/lowest set bit

`bsr, bsf:` Bit scan reverse/forward

Provide index of the most/least significant set bit.

See mssb{32,64}_0 and lssb{32,64}_0 in vm_basic_asm.h.

`bsr(n)` is $\log_2(n)$.

Be careful using these in inline assembly: they don't write their target register if the input is zero; they set ZF to (input == 0). Gcc doesn't use this to advantage. The vm_basic_asm.h functions return -1 for input == 0.

Going a bit too far: Copy a bit across variables

Copy bit `srcIdx` from `src` to bit `dstIdx` of `dst`. (Why? Moving guest arithmetic flags around, …)

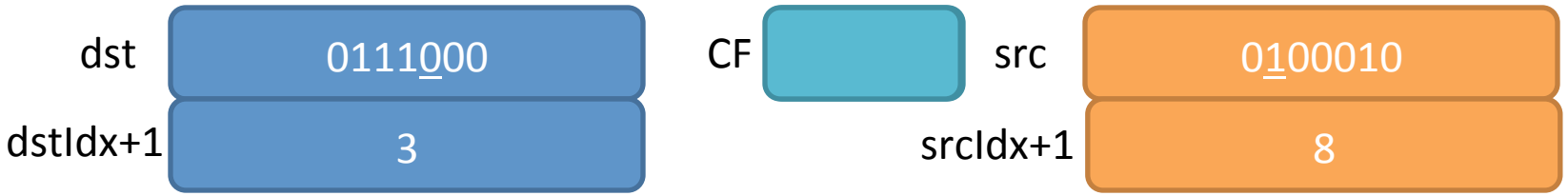
srcIdx: 7	src:	0100010
dstIdx: 2	dst:	0111000
Result:		0111100

```
dst &= ~(1 << dstIdx);
dst |= (src & (1 << srcIdx)) ? (1 << dstIdx) : 0;
```

Use rotate through carry instructions: `rcr, rcl`

Copy a bit across variables (and inline assembly tutorial)

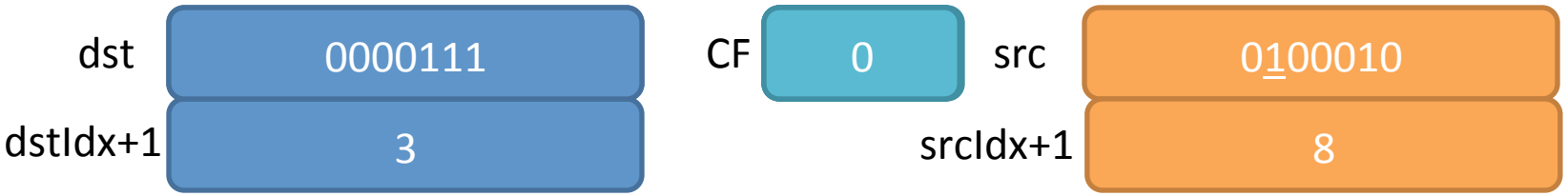
```
uint32 CopyBit(uint32 dst, uint8 dstIdx, uint32 src,
   uint32 srcIdx)
{
    uint64 tmp = dst;
    __asm__ (
        "rcr %[dstIdxPlus1], %[dst]          \n"
        "bt %[srcIdx], %[src]                \n"
        "rcl %[dstIdxPlus1], %[dst]"
        : [dst] "+rm" (tmp)
        : [dstIdxPlus1] "Ic" ((uint8)(dstIdx + 1)),
          [src] "rm" (src),
          [srcIdx] "Ir" (srcIdx));
    return tmp;
}
```



Copy a bit across variables (and inline assembly tutorial)

Rotate dst right by dstIdx+1: put bit (dstIdx) into the carry flag.

```
uint                t32 dst, uint8 dstIdx, uint32 src,
  u
{
  u                 t;
    __asm__ (
        "rcr %[dstIdxPlus1], %[dst]         \n"
        "bt %[srcIdx], %[src]               \n"
        "rcl %[dstIdxPlus1], %[dst]"
        : [dst] "+rm" (tmp)
        : [dstIdxPlus1] "Ic" ((uint8)(dstIdx + 1)),
          [src] "rm" (src),
          [srcIdx] "Ir" (srcIdx));
    return tmp;
}
```

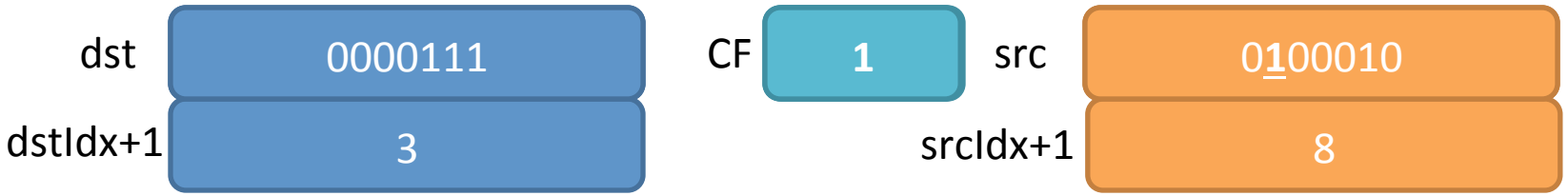


Copy a bit across variables (and inline assembly tutorial)

```
uint                                    stIdx, uint32 src,
    u
{
    u                                   t;
    __asm__ (
        "rcr %[dstIdxPlus1], %[dst]          \n"
        "bt %[srcIdx], %[src]                \n"
        "rcl %[dstIdxPlus1], %[dst]"
        : [dst] "+rm" (tmp)
        : [dstIdxPlus1] "Ic" ((uint8)(dstIdx + 1)),
          [src] "rm" (src),
          [srcIdx] "Ir" (srcIdx));
    return tmp;
}
```

Rotate dst right by dstIdx+1: put bit (dstIdx) into the carry flag.

Copy bit (srcIdx) into carry flag.



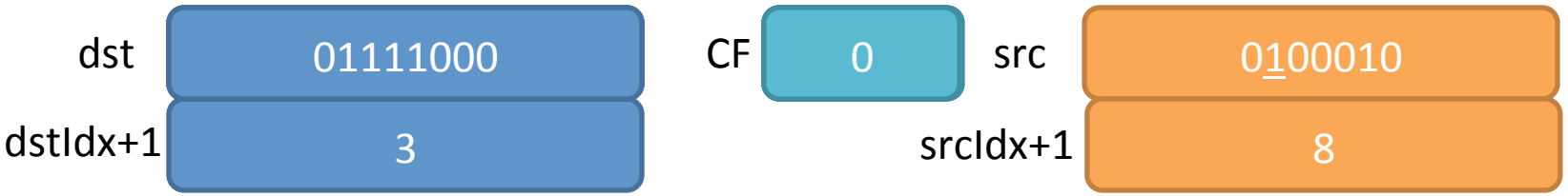
Copy a bit across variables

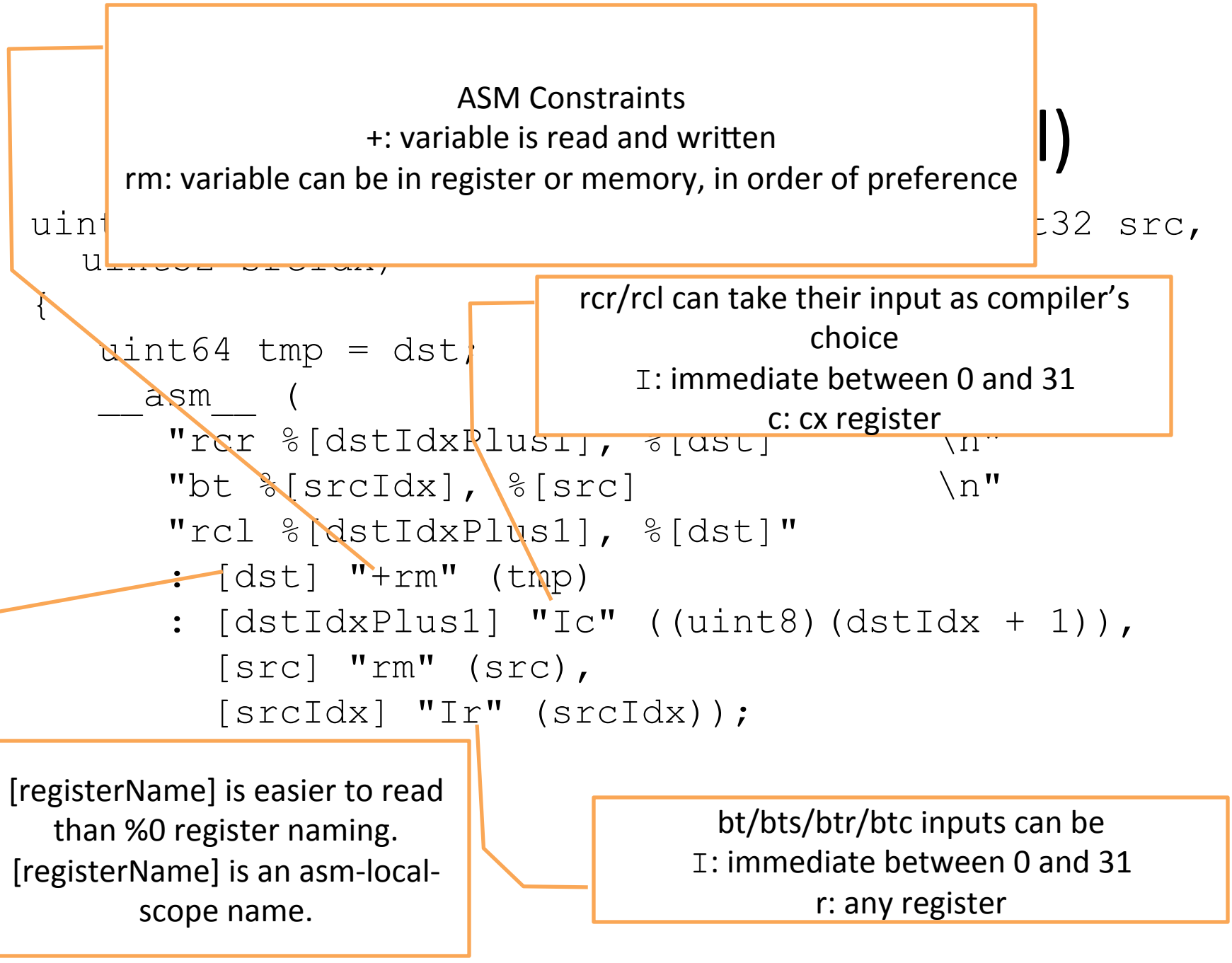
Rotate dst right by dstIdx+1: put bit (dstIdx) into the carry flag.

Copy bit (srcIdx) into carry flag.

Rotate dst back into place, with the newly-modified carry flag moving into position (dstIdx).

```
uint
  u
{
  u
    __asm__ (
        "rcr %[dstIdxPlus1], %[dst]          \n"
        "bt %[srcIdx], %[src]                \n"
        "rcl %[dstIdxPlus1], %[dst]"
        : [dst] "+rm" (tmp)
        : [dstIdxPlus1] "Ic" ((uint8)(dstIdx + 1)),
          [src] "rm" (src),
          [srcIdx] "Ir" (srcIdx));
    return tmp;
}
```




ASM Constraints
+: variable is read and written
rm: variable can be in register or memory, in order of preference
rcr/rcl can take their input as compiler's choice
I: immediate between 0 and 31
c: cx register
[registerName] is easier to read than %0 register naming.
[registerName] is an asm-local-scope name.
bt/bts/btr/btc inputs can be
I: immediate between 0 and 31
r: any register

CopyBit32 quirks

- x86 shifts and rotates' input counts can go up to (operand size-1); even though a 64-bit rotate-through-carry (65 total bits in rotation) would make sense.
- CopyBit32 uses 64-bit temp registers to avoid headaches accessing bit 31.
- CopyBit64 would need a special case for dstIdx=63 (rcl then rcr by 1).

Inline assembly pitfalls

- gcc can't propagate constants through inline assembly. (And it doesn't know much about arithmetic flags.)
- To get around some bad code generation, I used __builtin_constant_p(), which is used to choose code paths at compile time based on compiler information about the input value.
  - If the input is constant, gcc propagates all the way through __builtin_ffs. Otherwise, it generates bsf.

```
static INLINE int
lssb32_0(uint32 value)
{
#if defined(USE_ARCH_X86_CUSTOM)
   if (!__builtin_constant_p(value)) {
      if (UNLIKELY(value == 0)) {
         return -1;
      } else {
         int pos;
         __asm__ ("bsfl %1, %0\n" : "=r" (pos) : "rm" (value) : "cc");
         return pos;
      }
   }
#endif
   return __builtin_ffs(value) - 1;
}
```

Arithmetic flag (ab)uses

- Flags are set (partially or totally) by most instructions:
  - OF (arithmetic signed overflow)
  - SF (sign)
  - ZF (zero)
  - AF (auxiliary carry---for BCD math!) super bonus points for using
  - PF (parity on low 8 bits) bonus points for finding a use
  - CF (carry/borrow flag) also indicates odd conditions

Eg. ZF is usually set for the result. But `bts` sets ZF if the input is zero. In 3 instructions:

```
static INLINE int lssb32_0(uint32 value) {
__asm__ ("bsfl %1, %0\n"
         "cmovz %2, %0" : "=r" (pos) : "rm" (value),
                          "r"(-1) : "cc");
         return pos;
}}}
```

- C often neglects the flags that come out of arithmetic and re-`TEST`s or `CMP`s results.

Tricky arithmetic assortment

- `x & (x-1)` clears the least-significant 1-bit
  x: 010110 → result: 010100
  x-1: 010101
- `x & (-x)` isolates the least-significant 1-bit
  x: 010110 → result: 000010
  -x: 101010

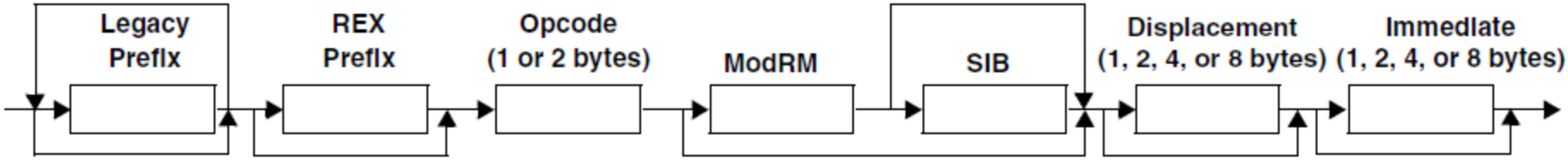
- Alternate between two values val1, val2 in variable foo with XOR:
  foo starts with either val1 or val2.
  foo ^= (val1 ^ val2);

  See Hacker's Delight for more.

- Round up to next power of 2 if x > 1:

```
int roundup(x) {
   return 2 << mssb32_0(x-1);
}
```

Code size considerations: x86 instruction encoding



- x86 instruction length is 1-15 bytes. gcc often picks the shortest encodings. (But sometimes puts in wasteful re-loads or moves)
- Some rules of thumb:
  - 32-bit sized register access is usually default in 64-bit code.
    - All 32-bit register stores zero out the upper 32-bits of the entire 64-bit register.
    - Nearly all instructions that operate on 32-bits are 1 byte shorter than their 64-bit equivalent, which is the same instruction prefixed with REX.

```
add %ecx, %edx: 01 ca
add %rcx, %rdx: 48 01 ca
```

    - But REX prefix is required for r8 and above (REX bytes encode 4 bits: 64-bit size and 3 register name extension bits)

```
add %r9d, %edx: 44 01 ca
add %r9,  %rdx: 4c 01 ca
```

- Instructions are not always orthogonal: Each family of instruction can access some subset of register, memory, immediate, hard-coded register.
- Reading the manual: "/3" means the modr/m byte contains a constant in the r/m field; "/r" means there's a fully programmable modr/m byte. Else, no modr/m byte. SIB is present if the modr/m byte says so.

(Image from Fig 1-1 of AMD Programmer Manual Vol 3)

gcc code quality

- Gcc doesn't always make the best code.

```
void addFoo(void)
{
   volatile unsigned foo = 0;
   UNROLL100({
      foo += (foo % 2 == 0) ? 3 : 5;
   });
}
```

One inner loop becomes:

```
mov (foo), %eax
mov (foo), %edx
and $0x1, %eax        # Test even
cmp $0x1, %eax        # CF = ~bit 0 of %eax.  and;cmp could be bt $1, %eax; cmc
sbb %eax, %eax        # %eax = %eax - %eax – CF == (foo is odd) ? 0 : -1
and $0xfffffffe, %eax        # %eax = (foo is odd) ? 0 : -2
lea 0x5(%rax,%rdx,1), %eax  # %eax = 0x5 + %eax + %edx = 0x5 + ((0 or -2) + foo
mov %eax, (foo)
```

gcc code quality

- Compare with hand-assembled code that uses cmov: 5 instructions/20 bytes/40.9 cycles vs. 8 instructions/30 bytes/47.3 cycles with gcc –O3 on Nehalem.

```
void addFoo(void)
{
   volatile unsigned foo = 0;
   UNROLL100({
      foo += (foo % 2 == 0) ? 3 : 5;
   });
}

void addFooCmov(void)
{
   register int foo asm("%rax") = 0;
   register int five asm("%rcx");
   register int tmp asm("%rdx");
   UNROLL100({
      asm ("mov $3, %[tmp]\n"
           "mov $5, %[five]\n"
           "test $1, %[foo]\n"
           "cmovz %[five], %[tmp]\n"
           "add  %[tmp], %[foo]\n" : [foo] "+r" (foo)
                                   : [five] "r" (five), [tmp] "r" (tmp));
   });
}
```

Other awesome x86 instructions

- `cbw` and relatives (opcode 0x98) do sign extension of various sizes (beware gnu disassembler's different opcode names)
- `cmc` inverts the carry flag (`stc`, `clc` set and clear)
- `prefetch` gets memory contents early, if you know the address far enough in the future…but be careful to avoid defeating the hardware prefetchers or polluting the cache
- `lea` adds 2 registers (or reg + immediate) nondestructively with a scale factor and doesn't modify arithmetic flags
- `popcnt` counts the number of 1-bits in a reg or mem (Barcelona and newer; Nehalem and newer)

References

- Agner Fog's very detailed reverse-engineering of all major x86 generations: http://www.agner.org/optimize/ (see microarchitecture.pdf)
- Arithmetic tricks: http://www.hackersdelight.org/, especially http://www.hackersdelight.org/basics.pdf
- Gcc builtin intrinsics: (eg. _builtin_ffs) http://gcc.gnu.org/onlinedocs/gcc-4.5.2/gcc/Other-Builtins.html
- Inline-assembly reference: http://www.ibm.com/developerworks/linux/library/l-ia.html http://gcc.gnu.org/onlinedocs/gcc-4.5.2/gcc/Constraints.html#Constraints
- "Demystifying Branch Predictors" http://www.ece.wisc.edu/~wddd/2002/final/milenkovic.pdf
- Optimization guides and programmer manuals: http://www.intel.com/products/processor/manuals/ http://developer.amd.com/documentation/guides/pages/default.aspx