CSE P 501 – Compilers

x86 Lite for Compiler Writers

Hal Perkins

Autumn 2011

Agenda

- Learn/review x86 architecture
  - Core 32-bit part only for now
    - Ignore crufty, backward-compatible things
    - Look at x86-64 extensions later
  - Suggest either 32- or 64-bit x86 as compiler target for project (tradeoffs either way)
    - If you want to try something else (ARM, MIPS, ?), let's talk
- After we've reviewed the x86 we'll look at how to map language constructs to code

x86 Selected History

- 30 Years of x86
  - 1978: 8086 – 16-bit processor, segmentation
  - 1982: 80286 – protected mode, floating point
  - 1985: 80386 – 32-bit architecture, "general-purpose" register set, VM
  - 1993: Pentium – mmx
  - 1999: Pentium III – SSE
  - 2000: Pentium IV – SSE2, SSE3, HT, virtualization
  - 2006: Core Duo, Core 2 – Multicore, SSE4+, x86-64
  - 2008: Atom, i7, …
- Many internal implementation changes, pipelining, concurrency, &c

And It's Backward-Compatible!

- Current processors can run 8086 code (!)
  - (You can get VisiCalc 1.0 on the web!)
- ∴ The Intel descriptions of the architecture are loaded down with modes and flags that obscure the modern, fairly simple 32-bit and 64-bit processor models
- Modern processors have a RISC-like core
  - Simple, register-register & load/store architecture
  - Simple x86 instructions preferred; complex CICS instructions supported
    - We'll focus on the basic 32-bit core instructions for now

x86 Assembler

- The nice thing about standards…
- Two main assembler languages for x86:
  - Intel/Microsoft– what's in the documentation
  - GNU / AT&T syntax (Linux, OS X)
    - Use gcc –S to generate examples from C/C++ code
  - You can use either for your project
- Slides use Intel descriptions
- Brief information later on differences
  - Main changes: dst,src reversed, data types in gnu opcodes, various syntactic annoyances

Intel ASM Statements

- Format is

  optLabel: opcode operands ; comment

  - optLabel is an optional label
  - opcode and operands make up the assembly language instruction
  - Anything following a ';' is a comment
- Language is very free-form
  - Comments and labels may appear on separate lines by themselves (we'll take advantage of this)

x86 Memory Model

- 8-bit bytes, byte addressable
- 16-, 32-, 64-bit words, doublewords, and quadwords
  - Data should almost always be aligned on “natural” boundaries; huge performance penalty on modern processors if it isn’t
- Little-endian – address of a 4-byte integer is address of low-order byte

x86 Processor Registers

- 8 32-bit, mostly general purpose registers
  - eax, ebx, ecx, edx, esi, edi, ebp (base pointer), esp (stack pointer)
- Other registers, not directly addressable
  - 32-bit eflags register
    - Holds condition codes, processor state, etc.
  - 32-bit "instruction pointer" eip
    - Holds address of first byte of next instruction to execute

Processor Fetch-Execute Cycle

- Basic cycle (same as every processor you’ve ever seen)

```
while (running) {
   fetch instruction beginning at eip address
   eip <- eip + instruction length
   execute instruction
}
```

- Sequential execution unless a jump stores a new “next instruction” address in eip

Instruction Format

- Typical data manipulation instruction

  opcode dst,src

- Meaning is

  dst <- dst op src

- Normally, one operand is a register, the other is a register, memory location, or integer constant
  - Can't have both operands in memory – can't encode two separate memory addresses

x86 Memory Stack

- Register esp points to the “top” of stack
  - Dedicated for this use; don’t use otherwise
  - Points to the last 32-bit doubleword pushed onto the stack (not next “free” one)
  - Should always be doubleword aligned
    - It will start out this way, and will stay aligned unless your code does something bad
  - Stack grows down

Stack Instructions

push src

- esp <- esp – 4; memory[esp] <- src
  (e.g., push src onto the stack)

pop dst

- dst <- memory[esp]; esp <- esp + 4
  (e.g., pop top of stack into dst and logically remove it from the stack)

- These are highly optimized and heavily used
  - 32-bit function call protocol is stack-based
  - 32-bit x86 doesn’t have enough registers, so the stack is frequently used for temporary space

Stack Frames

- When a method is called, a *stack frame* is traditionally allocated on the top of the stack to hold its local variables
- Frame is popped on method return
- By convention, ebp (base pointer) points to a known offset into the stack frame
  - Local variables referenced relative to ebp
  - (This is often optimized to use esp-relative references instead. Frees up ebp, needs additional bookkeeping at compile time.)

Operand Address Modes (1)

- These should cover most of what we'll need

```
mov  eax,17            ; store 17 in eax
mov  eax,ecx           ; copy ecx to eax
mov  eax,[ebp-12]      ; copy memory to eax
mov  [ebp+8],eax       ; copy eax to memory
```

- References to object fields work similarly – put the object's memory address in a register and use that address plus field offset

Operand Address Modes (2)

- In full generality, a memory address can combine the contents of two registers (with one being scaled) plus a constant displacement:

  [basereg + index*scale + constant]
  - Scale can be 2, 4, 8
- Main use is for array subscripting
- Example: suppose
  - Array of 4-byte ints, address of the array A is in ecx, subscript i is in eax
  - Code to store edx in A[i]

```
mov   [ecx+eax*4],edx     ;; and we didn't even use the offset!
```

dword ptr – Intel assembler

- Obscure, but sometimes necessary…
- If the assembler can't figure out the size of the operands to move, you can explicitly tell it to move 32 bits with the qualifier "dword ptr"

```
        mov   dword ptr  [eax+16],[ebp-8]
```

  - Use this if the assembler complains; otherwise ignore
  - Not an issue in GNU asembler – different opcode mnemonics for different operand sizes

Basic Data Movement and Arithmetic Instructions

mov dst,src

- dst <- src

add dst,src

- dst <- dst + src

sub dst,src

- dst <- dst – src

inc dst

- dst <- dst + 1

dec dst

- dst <- dst - 1

neg dst

- dst <- - dst
  (2's complement arithmetic negation)

Integer Multiply and Divide

imul dst,src

- dst <- dst * src
- 32-bit product
- dst *must* be a register

imul dst,src,imm8

- dst <- dst*src*imm8
- imm8 – 8 bit constant
- Obscure, but useful for optimizing array refs
- There are other mul instructions – see docs

idiv src

- Divide edx:eax by src (edx:eax holds sign-extended 64-bit value; cannot use other registers for division)
- eax <- quotient
- edx <- remainder

cdq

- edx:eax <- 64-bit sign extended copy of eax

Bitwise Operations

and dst,src

- dst <- dst & src

or dst,src

- dst <- dst | src

xor dst,src

- dst <- dst ^ src

not dst

- dst <- ~ dst (logical or 1's complement)

Shifts and Rotates

shl dst,count

  - dst shifted left count bits

shr dst,count

  - dst <- dst shifted right count bits (0 fill)

sar dst,count

  - dst <- dst shifted right count bits (sign bit fill)

rol dst,count

  - dst <- dst rotated left count bits

ror dst,count

  - dst <- dst rotated right count bits

Uses for Shifts and Rotates

- Can often be used to optimize multiplication and division by small constants
  - If you're interested, look at "Hacker's Delight" by Henry Warren, A-W, 2003
    - Lots of very cool bit fiddling and other algorithms
  - But be careful – be sure semantics are OK
- There are additional instructions that shift and rotate double words, use a calculated shift amount instead of a constant, etc.

Load Effective Address

- The unary & operator in C

  ```
  lea  dst,src      ; dst <- address of src
  ```

  - dst must be a register
  - Address of src includes any address arithmetic or indexing
  - Useful to capture addresses for pointers, reference parameters, etc.
  - Also useful for computing arithmetic expressions that match r1+scale*r2+const

Control Flow - GOTO

- At this level, all we have is goto and conditional goto
- Loops and conditional statements are synthesized from these
- Optimization note: random jumps play havoc with pipeline efficiency; much work is done in modern compilers and processors to minimize this impact

Unconditional Jumps

jmp dst

- eip <- address of dst
- Assembly language notes:
  - dst will be a label
  - Can have multiple labels on separate lines preceding an instruction
    - Convenient in compiler-generated asm lang.

Conditional Jumps

- Most arithmetic instructions set “condition code” bits in eflags to record information about the result (zero, non-zero, positive, etc.)
  - True of add, sub, and, or; but *not* imul or idiv
- Other instructions that set eflags

```
cmp dst,src        ; compare dst to src
test dst,src       ; calculate dst & src (logical
                   ; and); doesn’t change either
```

Conditional Jumps Following Arithmetic Operations

```
jz      label           ; jump if result == 0
jnz     label           ; jump if result != 0
jg      label           ; jump if result > 0
jng     label           ; jump if result <= 0
jge     label           ; jump if result >= 0
jnge    label           ; jump if result < 0
jl      label           ; jump if result < 0
jnl     label           ; jump if result >= 0
jle     label           ; jump if result <= 0
jnle    label           ; jump if result > 0
```

- Obviously, the assembler is providing multiple opcode mnemonics for several of these instructions

Compare and Jump Conditionally

- Want: compare two operands and jump if a relationship holds between them
- Would like to do this

  jmp_{cond} op1,op2,label

  but can’t, because 3-address instructions can’t be encoded in x86

cmp and jcc

- Instead, use a 2-instruction sequence

```
cmp    op1,op2
jcc    label
```

where jcc is a conditional jump that is taken if the result of the comparison matches the condition cc

Conditional Jumps Following Arithmetic Operations

```
je      label           ; jump if op1 == op2
jne     label           ; jump if op1 != op2
jg      label           ; jump if op1 > op2
jng     label           ; jump if op1 <= op2
jge     label           ; jump if op1 >= op2
jnge    label           ; jump if op1 < op2
jl      label           ; jump if op1 < op2
jnl     label           ; jump if op1 >= op2
jle     label           ; jump if op1 <= op2
jnle    label           ; jump if op1 > op2
```

- Again, the assembler is mapping more than one mnemonic to some machine instructions

Function Call and Return

- The x86 instruction set itself only provides for transfer of control (jump) and return
- Stack is used to capture return address and recover it
- Everything else – parameter passing, stack frame organization, register usage – is a matter of convention and not defined by the hardware

call and ret Instructions

call label

- Push address of next instruction and jump
- esp <- esp – 4; memory[esp] <- eip
  eip <- address of label

ret

- Pop address from top of stack and jump
- eip <- memory[esp]; esp <- esp + 4
- **WARNING!** The word on the top of the stack had better be an address, not some leftover data

enter and leave

- Complex instructions for languages with nested procedures
  - enter can be slow on current CPUs – best avoided
    - i.e., don’t use it in your project
  - leave is equivalent to

    ```
    mov esp,ebp
    pop ebp
    ```

    and is generated by many compilers. Fits in 1 byte, saves space. Not clear if it’s any faster.

Win 32 C Function Call Conventions

- Wintel code obeys the following conventions for C programs
  - Note: calling conventions normally designed very early in the instruction set/ basic software design. Hard (e.g., basically impossible) to change later.
- C++ augments these conventions to include the “this” pointer
- We’ll use these conventions in our code

Win32 C Register Conventions

- These registers must be restored to their original values before a function returns, if they are altered during execution
  esp, ebp, ebx, esi, edi
  - Traditional: push/pop from stack to save/restore
- A function may use the other registers (eax, ecx, edx) without having to save/restore
- A 32-bit function result is expected to be in eax when the function returns

Call Site

- Caller is responsible for
  - Pushing arguments on the stack from right to left (allows implementation of varargs)
  - Execute call instruction
  - Pop arguments from stack after return
    - For us, this means add 4*(# arguments) to esp after the return, since everything is either a 32-bit variable (int, bool), or a reference (pointer)

Call Example

```
n = sumOf(17,42)
        push  42                      ; push args
        push  17
        call  sumOf                   ; jump &
                                      ;   push addr
        add   esp,8                   ; pop args
        mov   [ebp+offset_n],eax      ; store result
```

Callee

- Called function must do the following
  - Save registers if necessary
  - Allocate stack frame for local variables
  - Execute function body
  - Ensure result of non-void function is in eax
  - Restore any required registers if necessary
  - Pop the stack frame
  - Return to caller

Win32 Function Prologue

- The code that needs to be executed before the statements in the body of the function are executed is referred to as the *prologue*
- For a Win32 function *f*, it looks like this:

```
f:  push   ebp              ; save old frame pointer
    mov    ebp,esp          ; new frame ptr is top of
                            ;  stack after arguments and
                            ;  return address are pushed
    sub    esp,”# bytes needed”
                            ; allocate stack frame (size
                            ; must be multiple of 4)
```

Win32 Function Epilogue

- The *epilogue* is the code that is executed for a return statement (or if execution "falls off" the bottom of a void function)
- For a Win32 function, it looks like this:

```
mov     eax,"function result"
                              ; put result in eax if not already
                              ;     there (if non-void function)
mov     esp,ebp               ; restore esp to old value
                              ;     before stack frame allocated
pop     ebp                   ; restore ebp to caller's value
ret                           ; return to caller
```

Example Function

- Source code

```
int sumOf(int x, int y) {
    int a, int b;
    a = x;
    b = a + y;
    return b;
}
```

Stack Frame for sumOf

```
int sumOf(int x, int y) {
   int a, int b;
   a = x;
   b = a + y;
   return b;
}
```

Assembly Language Version

```
;; int sumOf(int x, int y) {
;;  int a, int b;
sumOf:
    push  ebp       ; prologue
    mov   ebp,esp
    sub   esp, 8

;; a = x;
    mov  eax,[ebp+8]
    mov  [ebp-4],eax
```

```
;; b = a + y;
    mov  eax,[ebp-4]
    add  eax,[ebp+12]
    mov  [ebp-8],eax

;; return b;
    mov  eax,[ebp-8]
    mov  esp,ebp
    pop  ebp
    ret
;; }
```

Coming Attractions

- Now that we've got a basic idea of the x86 instruction set, we need to map language constructs to x86
  - Code Shape
- Then on to basic code generation and execution
  - And later, optimizations