CSE 401 – Compilers

Lecture 16: x86 Lite for Compiler Writers (a quick review)

Michael Ringenburg
Winter 2013

Reminders

• Project Part 2 due 11:59pm tonight
  – In theory – but the closest the dropbox will enforce is 12:01am tomorrow morning. Hopefully you won’t need those extra two minutes, though.
• Midterm on Friday.
  – Review session tomorrow in Sections.
• Homework 1 and 2 solutions are available – pick them up on your way out of class.
  – Please dispose of these after the final.
Quick Shift-Reduce Conflict Example

S ::= AList B
AList ::= A AList | ε
A ::= x x
B ::= x y

Stack  Rest of input
$             x ? ? ...

A lookahead 1 parser only sees the very next character

Problem: We are forced to decide which path to take before we have enough information, because one path requires a shift and the other a reduce.

If second symbol is x, we want to:
shift x, shift x, reduce x x → A

If second symbol is y, we want to:
reduce ε → AList, shift x, shift y, reduce x y → B, reduce AList B → S
What If We Used Left Recursion?

This works better: We don’t have to reduce until we know which path we’re on.

Note: This type of problem can also sometimes occur without empty productions — try walking through an example with \( AList ::= A \) \( AList \) | \( A \)

If second symbol is \( x \), we want to:
- Shift \( x \), shift \( x \), reduce \( x x \rightarrow A \), reduce \( AList A \rightarrow A \)

If second symbol is \( y \), we want to:
- Shift \( x \), shift \( y \), reduce \( x y \rightarrow B \), reduce \( AList B \rightarrow S \)
Agenda for the next couple weeks

• Overview of x86 architecture
  – Core 32-bit part only to start, not old cruft
  – 64-bit x86-64 later for the project
• Then...
  – Mapping source language constructs to x86
  – Code generation for MiniJava project
• Rest of the quarter...
  – Survey of compiler optimizations, more sophisticated code generation techniques

x86 Selected History

• 40 Years of x86
  – Early 70s: 8008 and 8080 – 8 bit processors
  – 1978: 8086 – 16-bit processor, segmentation – first to use x86 instruction set (designed to be easily translatable from 8008 and 8080 assembly)
  – 1982: 80286 – memory protection
  – 1985: 80386 – 32-bit architecture, “general-purpose” register set, better virtual memory support, IA-32 ISA ← Today’s class
  – 1993: Pentium – SIMD support
  – Late 90’s, early 00’s: Improved SIMD support, many other improvements
  – 2006: Core & Core 2 – Multicore, x86-64 64-bit ISA ← A future class
And It’s Backward-Compatible!

- 32-bit mode on current processors will run code written for the 8086
  - And 8008/8080 ISAs can be mechanically translated to run on 8086...
- The Intel descriptions are bloated with modes and flags that obscure the modern, fairly simple 32-bit processor model
- Modern x86 processors have a RISC-like core
  - Simple register-register and register-memory operations
  - Simple x86 instructions preferred; complex CISC instructions supported for compatibility

x86 Assembler

- Two main assembler languages for x86
  - Intel/Microsoft version – what’s in the documentation
  - AT&T/GNU assembler – what we’re generating
    - Use gcc –S to generate examples from C/C++ code
- These slides use Intel descriptions
  - But slides for x86-64 (the target of your project) will use GNU
- Information later on differences
  - Main changes: dst,src reversed, data types in gnu opcodes, various syntactic changes
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

x86 Memory Model

• 8-bit bytes, byte addressable
• 16-bit words, 32-bit doublewords, and 64-bit 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
Processor Registers

- 8 32-bit, mostly general purpose registers (in theory)
  - In reality, 6 or 7 are usable, depending on whether you use ebp
  - 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 conceptual cycle (same as most other processors you’ve 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 memory addresses in 1 instruction

x86 Memory Stack

- Register esp points to the “top” of stack
  - Dedicated for this use; don’t use otherwise – some instructions expect this usage
  - Stack grows down (push decrements esp, pop increments)
  - Points to the last 32-bit doubleword pushed onto the stack (not next “free” doubleword)
  - Should always be doubleword aligned
    - Can assume it will start out this way, and will stay aligned unless your code does something bad
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
  - The 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 addresses instead. Frees up ebp, which can be helpful on a register-starved machine; needs additional bookkeeping at compile time, not too hard)
Operand Address Modes (1)

- These should cover most of what we’ll need
  
  \begin{verbatim}
  mov eax,17 ; store 17 in eax
  mov eax,ecx ; copy ecx to eax
  mov eax,[ebp+8] ; copy memory to eax
  mov [ebp-12],eax ; copy eax to memory
  \end{verbatim}

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

- Remember: can’t have two memory addresses in a single instruction

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:
  
  \[ \text{basereg + indexreg*scale + constant} \]
  
  - Scale can be 2, 4, 8

- Main use for general form 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 ebx in A[i]

  \begin{verbatim}
  mov [ecx+eax*4],ebx
  \end{verbatim}
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
  - Can be useful for some subscript computations
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

\begin{itemize}
\item \textbf{and} \texttt{dst}, \texttt{src}
  \begin{itemize}
  \item \texttt{dst} <- \texttt{dst} \& \texttt{src}
  \end{itemize}
\item \textbf{or} \texttt{dst}, \texttt{src}
  \begin{itemize}
  \item \texttt{dst} <- \texttt{dst} \mid \texttt{src}
  \end{itemize}
\item \textbf{xor} \texttt{dst}, \texttt{src}
  \begin{itemize}
  \item \texttt{dst} <- \texttt{dst} \^ \texttt{src}
  \end{itemize}
\item \textbf{not} \texttt{dst}
  \begin{itemize}
  \item \texttt{dst} <- \texttt{~dst} (logical or 1's complement)
  \end{itemize}
\end{itemize}

Shifts and Rotates

\begin{itemize}
\item \textbf{shl} \texttt{dst}, \texttt{count}
  \begin{itemize}
  \item \texttt{dst} shifted left \texttt{count} \texttt{bits}
  \end{itemize}
\item \textbf{shr} \texttt{dst}, \texttt{count}
  \begin{itemize}
  \item \texttt{dst} shifted right \texttt{count} \texttt{bits} (0 fill)
  \end{itemize}
\item \textbf{sar} \texttt{dst}, \texttt{count}
  \begin{itemize}
  \item \texttt{dst} shifted right \texttt{count} \texttt{bits} (sign bit fill)
  \end{itemize}
\item \textbf{rol} \texttt{dst}, \texttt{count}
  \begin{itemize}
  \item \texttt{dst} rotated left \texttt{count} \texttt{bits}
  \end{itemize}
\item \textbf{ror} \texttt{dst}, \texttt{count}
  \begin{itemize}
  \item \texttt{dst} rotated right \texttt{count} \texttt{bits}
  \end{itemize}
\end{itemize}
Uses for Shifts and Rotates

• Can often be used to optimize multiplication and division by small constants
    • 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/C++
  lea dst,src ; dst <- address of src
  – dst must be a register, src should be memory address computation
  – Computes any address arithmetic or indexing in src, stores resulting address in dst
  – Useful to capture addresses for pointers, reference parameters, etc.
  – Also useful for computing arithmetic expressions that match r1+scale*r2+const
Unconditional Jumps

jmp dst

– eip <- address of dst (label)
– Processor will execute that instruction at dst next

Conditional Jumps

• Most arithmetic instructions set “condition code” bits in eflags to record information about the result (zero, non-zero, >0, etc.)
  – True of add, sub, and, or; but not imul, idiv, lea

• 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

```c
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 the actual instructions

Compare and Jump Conditionally

- Want: compare two operands and jump if a relationship holds between them
- Would like to do this
  ```c
  jmp_{\text{cond}} \, \text{op1,op2,label}
  ```
  but can’t, because 3-operand instructions can’t be encoded in x86
  
  \(\text{(also true of most other machines for that matter)}\)
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`
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

Winter 2013
UW CSE 401 (Michael Ringenburg)

---

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 – i.e., make sure you’ve popped off everything you pushed since the call
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.
  - Note: Mac x86 has more restrictive stack frame alignment requirements
- C++ augments these conventions to handle the “this” pointer
Win32 C Register Calling 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
  - Sometimes called *callee-saved* registers
  - Traditional: push/pop from stack to save/restore (aka *spill/reload*)
- A function may use the other registers (eax, ecx, edx) however it wants, without having to save/restore them
  - Sometimes called *caller-saved*, because the caller must save their values if it wants them later.
- A 32-bit function result is expected to be in eax when the function returns
- Generated code can get away with bending the rules, but watch it when you call external C code (and you will need to do this for your project).
  - i.e., follow the rules when you do make these calls...

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 basic MiniJava, this means add 4*(# arguments) to esp after the return, since everything is either a 32-bit variable (int, bool), or a reference (pointer), and there are no varargs to keep track of
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 (spill) registers if necessary
  – Allocate stack frame for local variables
  – Execute function body
  – Ensure result of non-void function is in eax
  – Restore (reload) any required registers if necessary
  – Pop the stack frame
  – Return to caller
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 for stack frame"
                   ; allocate stack frame
  ```

Win32 Function Epilogue

- The **epilogue** is the code that is executed to obey 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
  ```c
  int sumOf(int x, int y) {
    int a, int b;
    a = x;
    b = a + y;
    return b;
  }
  ```

Example Function

```c
;; 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
```
Example Function

```
;; 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
```

Winter 2013

UW CSE 401 (Michael Ringenburg)
Example Function

```c
;; 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
   }
```

Winter 2013

UW CSE 401 (Michael Ringenburg)
Example Function

```c
;; 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

;; }
```

```
Arg2: y
Arg1: x

Return address

Old frame ptr

a
b

eax = b
```
Example Function

```plaintext
;; 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

;; }
```

Return address

Arg1: x
Arg2: y

eax = b

ebp
esp

Winter 2013
Example Function

```plaintext
;; 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

eax = b

};
```

Caller then pops arguments and stores return value from eax.

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 x86-64, gnu assembler, and MiniJava code generation and execution