

# CSE P 501 – Compilers

x86 Architecture  
Hal Perkins  
Autumn 2005

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-1

## Agenda

- Learn/review x86 architecture
  - Core 32-bit part only
    - Ignore crufty, backward-compatible things
  - Suggested target language for MiniJava
    - (But if you want to do something different – x86-64, mips, PPC, ... – that should be fine – talk to us)
- After we've reviewed the x86 we'll look at how to map language constructs to code

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-2

## x86 Selected History

Processor	Intro Year	Intro Clock	Transistors	Features
8086	1978	8 MHz	29 K	16-bit regs., segments
286	1982	12.5 MHz	134 K	Protected mode
386	1985	20 MHz	275 K	32-bit regs., paging
486	1989	25 MHz	1.2 M	On-board FPU
Pentium	1993	60 MHz	3.1 M	MMX on late models
Pentium Pro	1995	200 MHz	5.5 M	P6 core, bigger caches
Pentium II	1997	266 MHz	7 M	P6 w/MMX
Pentium III	1999	700 MHz	28 M	SSE (Streaming SIMD)
Pentium 4	2000	1.5 GHz	42 M	NetBurst core, SSE2
Xeon	2001	2.2 GHz	55 M	Hyper-Threading
Pentium M	2003	1.6 GHz	77 M	Shorter pipelines vs P4

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-3

## And It's Backward-Compatible!

- Current Pentium/Xeon processors will run code written for the 8086(!)
  - (You can get VisiCalc 1.0 & others on the web!)
- ∴ Much of the Intel descriptions of the architecture are loaded down with modes and flags that obscure the modern, fairly simple 32-bit processor model
  - Links to the Intel manuals on the course web
- These slides try to cover the core x86 32-bit instructions and (MS) assembly language

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-4

## MASM – Microsoft Assembler

- MiniJava compiler project output will be an assembler source program
  - Let the assembler handle the translation to binary encodings, address resolutions, etc.
- Examples here use MASM – included in Visual Studio.NET
  - Can be needed to write hand-optimized code for MMX, SSE, and other special applications
- Other x86 assemblers: nasm, gas (GNU)
  - OK to use if you wish; you'll need to use the appropriate syntax, but instructions are the same

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-5

## MASM 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)

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-6

## x86 Memory Model

- 8-bit bytes, byte addressable
- 16-, 32-, 64-bit words, doublewords, and quadwords
  - Usually data should 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

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-7

## 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

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-8

## Processor Fetch-Execute Cycle

- Basic cycle

```
while (running) {
    fetch instruction beginning at eip address
    eip <- eip + instruction length
    execute instruction
}
```
- Execution continues sequentially unless a jump is executed, which stores a new “next instruction” address in `eip`

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-9

## Instruction Format

- Typical data manipulation instruction

```
opcode dst,src
```
- Meaning is

```
dst <- dst op src
```

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-10

## Instruction Operands

- Normally, one operand is a register, the other is a register, memory location, or integer constant
  - In particular, can't have both operands in memory – not enough bits to encode this
- Typical use is fairly “risc-like”
  - Modern processor cores optimized to execute this efficiently
  - Exotic instructions mostly for backward compatibility and normally not as efficient as equivalent code using simple instructions

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-11

## 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
  - Should always be doubleword aligned
    - It will start out this way, and will stay aligned unless your code does something bad
  - Stack grows down

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-12

## 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

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-13

## 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`
  - (Aside: this can be optimized to use `esp`-relative addresses instead. Frees up `ebp`, but needs additional bookkeeping at compile time)

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-14

## 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 an offset

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-15

## 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 1 in `A[i]`  
`mov [ecx+eax*4],1`

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-16

## dword ptr

- 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

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-17

## Basic Data Movement and Arithmetic Instructions

- |   |  |
|---|--|
| <code>mov dst,src</code> <ul style="list-style-type: none"><li>▪ <code>dst &lt;- src</code></li></ul>       | <code>inc dst</code> <ul style="list-style-type: none"><li>▪ <code>dst &lt;- dst + 1</code></li></ul>  |
| <code>add dst,src</code> <ul style="list-style-type: none"><li>▪ <code>dst &lt;- dst + src</code></li></ul> | <code>dec dst</code> <ul style="list-style-type: none"><li>▪ <code>dst &lt;- dst - 1</code></li></ul>  |
| <code>sub dst,src</code> <ul style="list-style-type: none"><li>▪ <code>dst &lt;- dst - src</code></li></ul> | <code>neg dst</code> <ul style="list-style-type: none"><li>▪ <code>dst &lt;- - dst</code><br/>(2's complement arithmetic negation)</li></ul> |

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-18

## 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 subscripts (but address modes can do simple scaling)

`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`

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-19

## 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)

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-20

## 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

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-21

## 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
- There are additional instructions that shift and rotate double words, use a calculated shift amount instead of a constant, etc.

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-22

## 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.

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-23

## Control Flow - GOTO

- At this level, all we have is `goto` and conditional `goto`
- Loops and conditional statements are synthesized from these
- A jump (`goto`) stores the destination address in `eip`, the register that points to the next instruction to be fetched
- Optimization note: jumps play havoc with pipeline efficiency; much work is done in modern compilers and processors to minimize this impact

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-24

## Unconditional Jumps

### jmp dst

- eip <- address of dst
- Assembly language note: dst will be a label. Execution continues at first machine instruction in the code following that label
- Can have multiple labels on separate lines in front of an instruction

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-25

## Conditional Jumps

- Most arithmetic instructions set 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
```

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-26

## 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 individual instructions
- If you use these, it will probably be the result of an optimization

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-27

## Compare and Jump Conditionally

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

```
jmpcond op1,op2,label
```

but can't, because 3-address instructions aren't included in the architecture

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-28

## 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

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-29

## 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 of the actual machine instructions

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-30

## 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

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-31

## call and ret Instructions

call label

- Push address of next instruction and jump
- $esp \leftarrow esp - 4$ ;  $memory[esp] \leftarrow eip$   
 $eip \leftarrow \text{address of label}$

ret

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

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-32

## Win 32 C Function Call Conventions

- Wintel compilers obey 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 handle the "this" pointer
- We'll use these conventions in our code

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-33

## 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) however it wants, without having to save/restore them
- A 32-bit function result is expected to be in eax when the function returns

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-34

## 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  $4 * (\# \text{ arguments})$  to esp after the return, since everything is either a 32-bit variable (int, bool), or a reference (pointer)

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-35

## Call Example

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

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-36

## 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

11/1/2005 © 2002-05 Hal Perkins & UW CSE J-37

## 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
```

11/1/2005 © 2002-05 Hal Perkins & UW CSE J-38

## 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
```

11/1/2005 © 2002-05 Hal Perkins & UW CSE J-39

## Example Function

- Source code
 

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

11/1/2005 © 2002-05 Hal Perkins & UW CSE J-40

## Stack Frame for sumOf

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

11/1/2005 © 2002-05 Hal Perkins & UW CSE J-41

## 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
;; }
```

11/1/2005 © 2002-05 Hal Perkins & UW CSE J-42



## 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 later, an optimization sampler

11/1/2005

© 2002-05 Hal Perkins & UW CSE

J-43