

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



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)

/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 Porkins & HW CSE



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



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



Instruction Format

- Typical data manipulation instruction opcode dst,src
- Meaning is dst <- dst op src

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



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

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



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</pre> (e.g., push src onto the stack)

pop dst

- dst <- memory[esp]; esp <- esp + 4</p> (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

© 2002-05 Hal Porkins & HW CSE



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 àddresses instead. Frees up ebp, but needs additional bookkeeping at compile time)

© 2002-05 Hal Perkins & LIW CSF



Operand Address Modes (1)

These should cover most of what we'll need

mov eax,17 ; store 17 in eax ; copy ecx to eax mov eax.ecx 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

11/1/2009 © 2002-05 Hal Perkins & UW CSE



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



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

© 2002-05 Hal Perkins & UW CSE



Basic Data Movement and Arithmetic Instructions

mov dst,src

dst <- src</p>

add dst,src

■ dst <- dst - src

dst <- dst + src</p>

sub dst,src

inc dst

dst <- dst + 1</p>

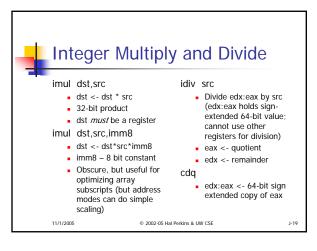
dec dst

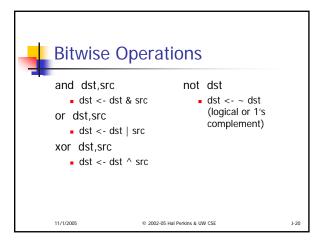
dst <- dst - 1</p>

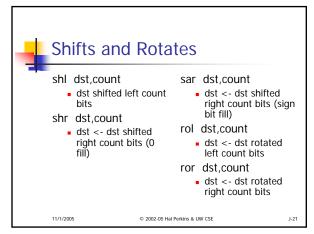
neg dst

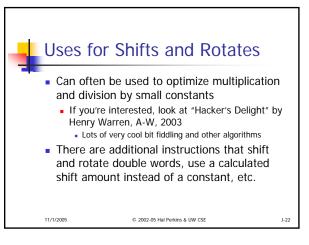
 dst <- - dst (2's complement arithmetic negation)

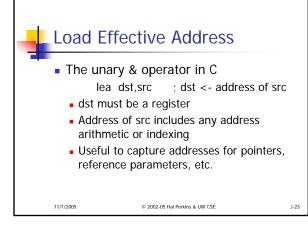
11/1/2005 © 2002-05 Hal Perkins & UW CSE















Unconditional Jumps

imp 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



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

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

Conditional Jumps Following Arithmetic Operations

```
; jump if result == 0
; jump if result != 0
          label
         label
inz
                               ; jump if result > 0
          label
                               ; jump if result <= 0
         label
                               jump if result >= 0
                              ; jump if result < 0
; jump if result < 0
         label
         label
jnl
         label
                               ; jump if result >= 0
         label
                              ; jump if result <= 0
                               ; jump if result > 0
jnle
         label
```

- 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



Compare and Jump Conditionally

- Very common pattern: 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 aren't included in the architecture

J-28

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



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



Conditional Jumps Following Arithmetic Operations

```
; jump if op1 == op2
; jump if op1 != op2
         label
         label
ine
         label
                           jump if op1 > op2
jng
         label
                           jump if op1 <= op2
         label
                           jump if op1 >= op2
jge
jnge
        label
                           jump if op1 < op2
                           ; jump if op1 < op2
        label
jnl
        label
                           ; jump if op1 >= op2
                           jump if op1 <= op2
        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



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 & LIW CSE

4

call and ret Instructions

call label

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

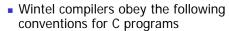
ret

- 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

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



Win 32 C Function Call Conventions



- 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



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



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 32bit variable (int, bool), or a reference (pointer)

11/1/2005

© 2002-05 Hal Perkins & UW CSE



Call Example

n = sumOf(17,42)

push 42

push 17

call sumOf

; push addr

; push args

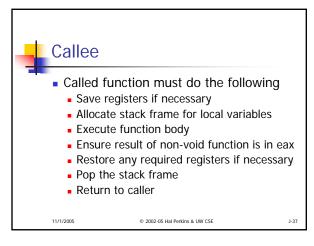
add esp,8

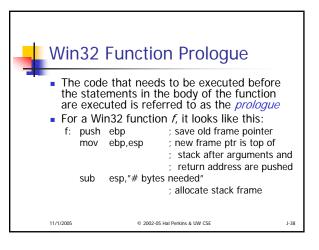
; pop args ; store result

; jump &

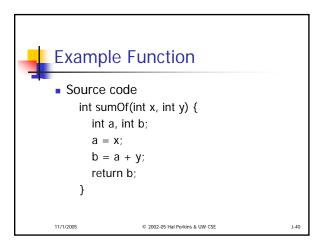
mov [ebp+offset_n],eax

11/1/2005 © 2002-05 Hal Perkins & UW CSE J-3





```
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:
              eax,"function result"
       mov
                              ; put result in eax if not already
                                 there (if non-void function)
                              ; restore esp to old value
        mov
               esp,ebp
                                 before stack frame allocated
                              ; restore ebp to caller's value
        pop
               ebp
                              · return to caller
        ret
11/1/2005
                      © 2002-05 Hal Perkins & UW CSE
```



```
Stack Frame for sumOf su
```

```
Assembly Language Version
;; int sumOf(int x, int y) {
;; int a, int b;
                             mov eax,[ebp-4]
sumOf:
                             add eax,[ebp+12]
  push ebp
              ; prologue
                             mov [ebp-8],eax
  mov ebp,esp
  sub
        esp, 8
                          :: return b:
                             mov eax,[ebp-8]
                             mov esp,ebp
;; a = x;
  mov eax,[ebp+8]
                             pop ebp
  mov [ebp-4],eax
                             ret
                          ;; }
                  © 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

1.4