Java versus C++

Seminar WS 2010 / 2011

Session 2, Wednesday October 27, 2010

(Machine code generation from C / C++)

Prof. Dr. Hannah Bast
Chair for Algorithms and Data Structures
Department of Computer Science
University of Freiburg

UNI FREIBURG

Overview of my talk

UNI FREIBURG

- Machine code generation from C / C++
  - Basic principle
  - Some demos
- Machine code
  - Short history, x86 and RISC
  - x86 general-purpose registers
  - x86 basic instructions
  - x86 memory allocation: heap and stack
  - x86 / AMD64 function calling
  - x86 / AMD64 streaming registers

Machine code generation from C / C++

UNI FREIBURG

- Basic principle of any interpreter / compiler:
  - Take each line from the code
  - And translate it to an equivalent sequence of machine code instructions
  - [Show live example]
  - Quite easy in principle: we could write a compiler for basic C (variables, while, if, functions, I/O) in a week
  - It wouldn't produce very efficient code though …
- The key to producing efficient code:
  - Do not translate the code line by line, but consider appropriate blocks of code together → talk on optimization
  - [Continue live example]

To understand any of this, we need to understand how machine code works … hence this talk

A history of machine code (x86-biased)

- 8-bit architectures
  - 1972: Intel 8008 (the world's first 8-bit microprocessor)
  - 1974: Intel 8080 (added some 16-bit operations)
  - 1976: Zilog Z80 (the most successful one, still in use today)
- 16-bit architectures
  - 1978: Intel 8086 (the first member of the x86 family)
- 32-bit architectures
  - 1985: Intel 80386 aka i386 (backwards-compatible with 8086)
  - 1993: Intel Pentium (again, backwards-compatible)
- 64-bit architectures
  - 2003: AMD 64, Intel 64 (backwards-compatible with x86)

referred to as x86-64 or x64

The vast majority of desktop computers, laptops, and servers today run x86 / x64 machine code

RISC / Load-store architecture

UNI FREIBURG

- RISC = Reduced Instruction Set Computing
  - Basic idea: very small and simple instruction set, enabling faster implementation of hardware
  - In practice: operations load/store for transfer registers ↔ memory; all other operations on registers only
  - Most code is for x86, however, which is not RISC (although some ideas have been picked up over the years, e.g. Pentium)
  - Overproportionally more work went into the x86 optimziation → little performance difference between x86 and RISC today
  - Famous RISC example: the ARM architecture (32-bit)
    - Game Boy, BlackBerry, Palm, iPod, iPhone, iPad, G1, ...
    - So beware that some of the things we find in this seminar may or may not be applicable for these devices

x86 Registers

- Intel 8086 registers (16-bit)
  - AX, BX, CX, DX: general-purpose registers (with special usage as "accumulator", "base", "counter", "data")
  - SI, DI: source index, destination index
  - SP, BP: stack pointer, base pointer
  - CS, DS, SS, ES: segment registers (code, data, stack, extra)
- Intel 80836 registers (32-bit)
  - EAX, EBX, ECX, EDX, etc. [E = extended]
  - additional segment registers FS and GS
  - eight 64-bit streaming registers MMX0, MMX1, ...
- AMD Opteron (64-bit)
  - RAX, RBX, RCX, RDX, etc. [R = ?]
  - additional 64-bit registers R8, R9, ..., R15
  - sixteen 128-bit streaming registers XMM0, XMM1, ...

"segmentation"
→ talk on C++ memory mngment

x86 Basic instructions 1/2

- Assigment
  - mov X, Y : assign the value of X to Y
  - Here, and for many commands, X and Y can be registers, e.g. %rax, or absolute memory locations, e.g. label, or memory locations pointed to by a register, e.g. 4(%rsp).
- Arithmetic and bitwise operations
  - add, sub, mul, div, inc (increment), dec (decrement), ...
  - and, or, xor, sal (shift left), sar (shift right), ...
- Suffixes
  - no suffix = 16 bits, l = 32 bits ("long"), q = 64 bits ("quad")
  - for example: mov, movl, movq, add, addl, addq, ...

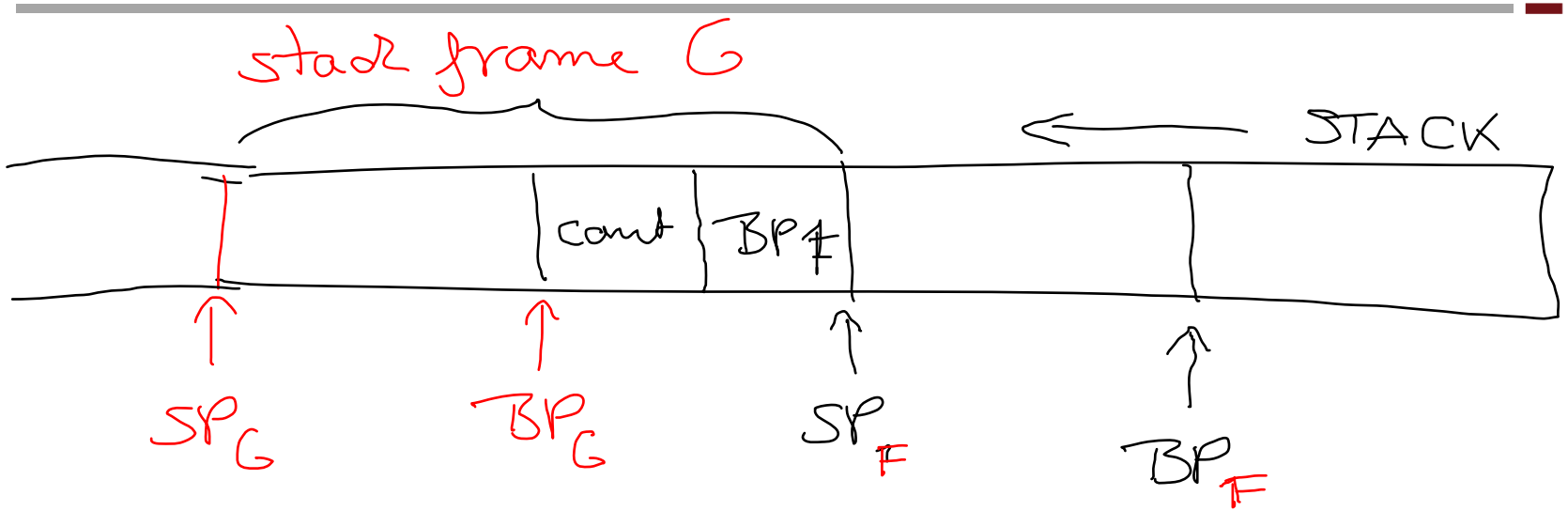
x86 Basic instructions 2/2

- Stack operations
  - push X : push X on stack (decreases SP = stack pointer)
  - pop X : pop X from stack (increases SP = stack pointer)
- Comparisons and jumps
  - cmp X, Y : compare X and Y and remember < oder > or =
  - je X, jne X, jl X : jump to X if equal, not equal, less, ...
  - jmp X : jump unconditionally to X
- Function calling
  - call X : push instruction pointer and jump to X
  - ret : pop instruction pointer and jump to that address
  - enter X : create a new stack frame with room for X bytes
  - leave : restore the old stack frame

Memory allocation: heap and stack

- Heap
  - General-purpose memory allocation
  - At any time we may get a request for any number of bytes
  - At any time we may no longer need any number of bytes
  - The part of memory where this is organized is called the heap, auf Deutsch Haufen → talk on C++ memory allocation
- Stack
  - Memory allocation for global and local variables, function parameters, and function results
  - Has the LIFO property: last object in, first object out
  - Therefore we can organize these objects on a stack (Stapel)

Memory allocation on the stack



function F
{
..
call G
const: ...
}

stack frame of F

to "free" memory on the stack, just increase SP

to "allocate" memory on the stack, just decrease SP

x86 Function calling 1/3

- C-Style calling convention (most common)
  - Stack frame = part of stack that belongs to the function we are currently in, left end = SP < right end = BP
  - Push the function arguments on the stack, from right to left
  - Then call pushes the instruction pointer (IP) on the stack
  - The first action of the called function (the callee) must be
    - to push the BP on the stack, and then
    - set BP = SP, effectively starting a new stack frame
  - Before the callee returns, must pop BP of the stack again
  - Then ret pops the IP from the stack and jumps there
  - Now we are back in the calling function with its stack frame
  - Now calling function must pop the arguments it pushed

x86 Function calling 2/3

- C-Style calling convention example

x86 Function calling 3/3

- Standard call (e.g. the Win32 API uses this)
  - Very similar to C-style call
  - Assumes fixed number of arguments for each function call
  - Again caller pushes arguments on the stack, but callee is now responsible for removing them from the stack again
  - Advantage: less work everytime we call the function
  - Disadvantage: wrong number of arguments is fatal now
- Shortcuts
  - enter 10 = push BP; mov SP→BP; sub 10, SP
  - leave = mov BP→SP; pop BP
  - pusha = push AX, CX, DX, BX, SP, BP, SI, DI
  - popa = pop DI, SI, BP, SP, BX, DX, CX, AX

AMD64 Function calling

- Arguments no longer passed via the stack
  - But via registers (AMD64 has many of them)
  - In particular for system calls
  - This gives significant speedups in practice

SSE

- SSE = Streaming SIMD Extensions
  - Motivation: carry out the same instruction for a number of operators at the same time (SIMD = Single instruction, multiple data)
  - Large (nowadays 128-bit) registers XMM0, XMM1, ...
  - Originally 8 such registers, AMD64 now has 16
  - Example: eight 4-byte integers x1, x2, x3, x4 (stored at address X) and y1, y2, y3, y4 (stored at address Y), then compute x1+y1, x2+y2, x3+y3, x4+y4 (to be stored at address Z) with just three instructions as follows

```
movaps XMM0, X
addps XMM0, Y
movaps Z, XMM0
```

Literature / Links

- x86 and RISC
  - http://en.wikipedia.org/wiki/X86_architecture
  - http://en.wikipedia.org/wiki/RISC
- x86 registers and instruction set
  - http://en.wikipedia.org/wiki/X86#x86_registers
  - http://en.wikipedia.org/wiki/X86_instruction_listings
- x86 Linux assembler tutorial (the basics, very nice)
  - http://www.m-hoeppner.de/projects/asm_ws.pdf
- x86 function calling, C-style vs. Standard
  - http://unixwiz.net/techtips/win32-callconv-asm.html
- SSE = Streaming SIMD Extensions
  - http://en.wikipedia.org/wiki/Streaming_SIMD_Extensions