Machine Programming Basics

CSC 235 - Computer Organization

References

- Slides adapted from CMU

Outline

- History of Intel processors and architectures
- Assembly basics: registers, operands, move
- Arithmetic and logical operations
- C, assembly and machine code

Intel x86 Processors

- Dominate laptop/desktop/server market
- Evolutionary design
  - Backwards compatible up until 8086, introduced in 1978
  - Added more features as time goes on
- Complex instruction set computer (CISC)
  - Many different instructions with many different formats
  - Difficult to match performance of Reduced Instruction Set Computers (RISC)
  - But, Intel has done just that in terms of speed, less so for low power

Intel x86 Evolution: Milestones

Name	Date	Transistors	MHz	Notes
8086	1978	29K	5-10	16-bit
386	1985	275K	16-33	32-bit
Pentium 4E	2004	125M	2800-3800	64-bit
Core 2	2006	291M	1060-3333	multi-core
Core i7	2008	731M	1600-4400	four cores

x86 Clones: Advanced Micro Devices (AMD)

- Historically
  - AMD has followed just behind Intel
  - A little bit slower, a lot cheaper
- Then
  - Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
  - Built Opteron: tough competitor to Pentium 4
  - Developed x86-64, their own extension to 64 bits
- Recent years
  - Intel leads the world in semiconductor technology
  - AMD has fallen behind

Intel’s 64 bit History

- 2001: Intel attempts radical shift from IA32 to IA64
  - Totally different architecture (Itanium)
  - Performance disappointing
- 2003: AMD steps in with evolutional solution
  - x86-64 (now called “AMD64”)
- 2004: Intel Announces EM64T extension to IA32
  - Extended Memory 64 bit Technology
  - Almost identical to x86-64
- All but low-end x86 processors support x86-64
  - but, lots of code still runs in 32 bit mode

Definitions

- Architecture: the parts of a processor design that one needs to understand for writing correct machine/assembly code
  - Machine code: the byte level programs that a processor executes
  - Assembly code: a text representation of machine code
- Microarchitecture: implementation of the architecture
- Example Instruction Set Architectures (ISA)
  - Intel: x86, IA32, Itanium, x86-64
  - ARM: Used in almost all mobile phones
  - RISC V: new open source ISA

Assembly/Machine Code View

- Programmer Visible State
    - PC: Program counter
        - Address of next instruction
    - Register file
    - Condition codes
        - store status information about most recent arithmetic or logical operation
- Memory
    - Byte addressable array
    - Code and user data
    - Stack to support procedures

Assembly Characteristics

- “Integer” data of 1, 2, 4, or 8 bytes
  - data values
  - addresses (untyped pointers)
- Floating point data of 4, 8, or 10 bytes
- SIMD vector data types of 8, 16, 32, or 64 bytes
- Code: byte sequences encoding series of instructions
- No aggregate types such as arrays or structures

x86-64 Integer Registers

8-byte register	bytes 0-3	bytes 0-1	byte 0
%rax	%eax	%ax	%al
%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rbx	%ebx	%bx	%bl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil
%rsp	%esp	%sp	%spl
%rbp	%ebp	%bp	%bpl

x86-64 Integer Registers (continued)

8-byte register	bytes 0-3	bytes 0-1	byte 0
%r8	%r8d	%r8w	%r8b
%r9	%r9d	%r9w	%r9b
%r10	%r10d	%r10w	%r10b
%r11	%r11d	%r11w	%r11b
%r12	%r12d	%r12w	%r12b
%r13	%r13d	%r13w	%r13b
%r14	%r14d	%r14w	%r14b
%r15	%r15d	%r15w	%r15b

x86-64 Integer Registers (continued)

- Some assembly instructions include a suffix that indicates what portion of the register is accessed:
  - **q**: “quadword” 8 bytes
  - **l**: “double word” lower 4 bytes
  - **w**: “word” lower 2 bytes
  - **b**: “byte” lowest byte

Assembly Characteristics: Operations

- Transfer data between memory and register
  - Load data from memory into register
  - Store register data into memory
- Perform arithmetic function on register or memory data
- Transfer control
  - Unconditional jumps to/from procedures
  - Conditional branches
  - Indirect branches

Moving Data

- Instruction:
  - `movq` *source* (Src), *destination* (Dest)
- Operand types
  - Immediate (Imm): constant integer data
  - Register (Reg): one of 16 integer registers
  - Memory (Mem): 8 consecutive bytes of memory at address given by register

`movq` Operand Combinations

Source	Destination	Example	C Analog
Imm	Reg	`movq $0x4, %rax`	`temp = 0x04;`
Imm	Mem	`movq $-147, (%rax)`	`*p = -147;`
Reg	Reg	`movq %rax, %rdx`	`temp2 = temp1;`
Reg	Mem	`movq %rax, (%rdx)`	`*p = temp;`
Mem	Reg	`movq (%rax), %rdx`	`temp = *p;`

Memory Addressing Modes

- Immediate
  - `$val`
  - val: constant integer value
  - example: `movq $7, %rax`
- Normal
  - (R) Mem[Reg[R]]
  - R: register R specifies memory address
  - `movq (%rcx), %rax`

Memory Addressing Modes (continued)

- Displacement
    - D(R) Mem[Reg[R] + D]
    - R: register specifies start of memory region
    - D: constant displacement D specifies offset
    - example: `movq 8(%rdi), %rdx`

Memory Addressing Modes (continued)

- Indexed
  - D(Rb, Ri, S) Mem[Reg[Rb] + S*Reg[Ri]+D]
  - D: constant displacement 1, 2, or 4 bytes
  - Rb: base register
  - Ri: index register: any except %esp
  - S: scale: 1, 2, 4, or 8
  - example: `movq 0x100(%rcx, %rax, 4), %rdx`

Addressing Modes Example

- Example C code

```
void swap (long *xp, long *yp) {
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

Addressing Modes Example

- x86 assembly version

```
    # %rdi = xp
    # %rsi = yp
swap:
    movq    (%rdi), %rax # t0 = *xp
    movq    (%rsi), %rdx # t1 = *yp
    movq    %rdx, (%rdi) # *xp = t1
    movq    %rax, (%rsi) # *yp = t0
    ret
```

Address Computation Examples

- `rdx` contains `0xf000`
- `rcx` contains `0x0100`

Expression	Address Computation	Address
`0x8 (%rdx)`	`0xf000 + 0x8`	`0xf008`
`(%rdx, %rcx)`	`0xf000 + 0x100`	`0xf100`
`(%rdx, %rcx, 4)`	`0xf000 + 4*0x100`	`0xf400`
`0x80(,%rdx,2)`	`2*0xf000 + 0x80`	`0x1e080`

Address Computation Instruction

- `leaq` Src, Dest
  - Load effective address of source into destination
- Uses
  - Computing addresses without a memory reference
  - Computing arithmetic expressions of the form `x + k * y`
- Example

```
long m12(long x) {
    return x*12;
}

leaq (%rdi, %rdi, 2), %rax # t = x+2*x
salq $2, %rax
```

Some Arithmetic Operations

- Binary operators

`addq`	Src, Dest	Dest = Dest + Src
`subq`	Src, Dest	Dest = Dest - Src
`imulq`	Src, Dest	Dest = Dest * Src
`salq`	Src, Dest	Dest = Dest << Src
`sarq`	Src, Dest	Dest = Dest >> Src (arithmetic)
`shrq`	Src, Dest	Dest = Dest >> Src (logical)
`xorq`	Src, Dest	Dest = Dest ^ Src
`andq`	Src, Dest	Dest = Dest & Src
`orq`	Src, Dest	Dest = Dest | Src

- Be careful of the argument order

Some Arithmetic Operations

- Unary operators

`incq`	Dest	Dest = Dest + 1
`decq`	Dest	Dest = Dest - 1
`negq`	Dest	Dest = - Dest
`notq`	Dest	Dest = ~ Dest

Arithmetic Expression Example

- C code

```
long arith (long x, long y, long z) {
    long t1 = x+y;
    long t2 = z+t1;
    long t3 = x+4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 + t5;
    return rval;
}
```

Arithmetic Expression Example

- Assembly code

```
    # %rdi = x
    # %rsi = y
    # %rdx = z
arith:
    leaq (%rdi, %rsi), %rax       # t1
    addq %rdx, %rax               # t2
    leaq (%rsi, %rsi, 2), %rdx
    salq $4, %rdx                 # t4
    leaq 4(%rdi, %rdx), %rcx      # t5
    imulq %rcx, %rax              # rval
    ret
```

Turning C into Object Code

- Code in files `p1.c` and `p2.c`
- Compile with command: `gcc -Og p1.c p2.c -o p`
  - use basic optimizations (`-Og`)
  - put resulting binary in file `p`
- The above `gcc` command runs the following programs:
  - source text $\rightarrow$ cpp $\rightarrow$ compiler $\rightarrow$ assembler $\rightarrow$ linker

Assembly

- Compiling C to assembly: `gcc -Og -S <file>`
  - produces an assembly file `<file>.s`
- Disassembling Code: `objdump -d <file>`
  - useful tool for examing object code
  - analyzes bit pattern of series of instructions
  - produces approximate rendition of assembly code

Summary

- History of Intel processors and architectures
- C, assembly, machine code
  - new forms of visible state: program counter, registers, ...
  - Compiler must transform language constructs into low level instruction sequences
- Assembly basics: registers, operands, move
- Arithmetic