U N I V E R S I T Y

CS 50011: Introduction to Systems II

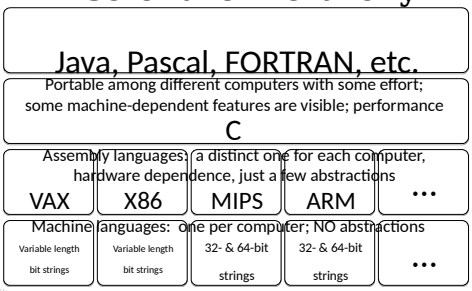
Lecture 4: Introduction to Assembly

Prof. Jeff Turkstra

Lecture 03

- History
- Background
- x86
  - Syntax
  - Operands
  - Addressing modes
  - Data types
  - Instructions

A software hierarchy



Assembly language

- All (somewhat) different
- Many assembly languages share the same fundamental structure
  - Why?
- Typical assembly language statement syntax and corresponding machine code in hex…

```
label: op result, operand1, operand2
0x004005F9 0x23CC803C
```

- Label is symbolic (an abstraction) for a memory address
- "op" is a mnemonic for the operation

Assembly is two-pass

- Initial pass of assembler resolves memory addresses for all labels
  - Even (especially) forward references
  - Symbol table
- Second pass emits machine code bitstrings
  - Translates mnemonics, register names, etc
  - Uses symbol table to fill in offset bit field
    - Offset = branch_target - current_addr

Why?

- Many languages are one-pass
  - C, for example
    - Have to prototype functions, declare/define
- Would have to manually determine instruction addresses and branch targets
- Changing the code often changes all of the offsets and addresses
- Impractical

Opcodes

- Set of opcode-field bit strings defines what the processor circuit can do
- Different processors have different sets of opcodes
- Assembly language defines a memorable symbolic name of a few characters for each opcode, a mnemonic
- No agreement on opcode mnemonics across assembly languages

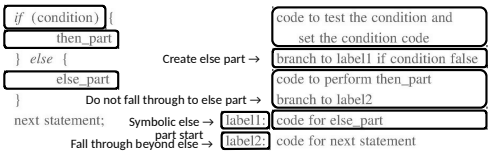
Readability

- Assembly is easy to write but hard to follow
- Comments are essential
  - Block comment – explain the purpose of a section of code, detail the use of registers and memory
  - Line comment – explains each instruction
- Comment usually starts with a delimiter, runs to end of line
- Best strategy: comment every line

Example

```
#####################################################
# Search linked list of free memory blocks to find  #
# a block of size N bytes or greater. Pointer must  #
# be in r3 and N in r4. Code destroys contents of   #
# r5, which is used to walk the list.               #
#####################################################
        ld   r5,r3   # load address of list into r5
loop_1: cmp  r5,0    # test to see if at list end
        bz   notfnd  # if reached end go to notfnd
```


Coding IF-THEN-ELSE in assembly



"Fall through" means to fetch at the default next instruction location; must code two exceptions for if-then-else

Figure 9.2 (a) An *if-then-else* statement used in a high-level language, and (b) the equivalent assembly language code.

Subroutine call in assembly

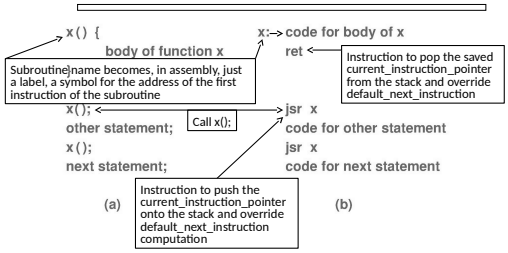


Figure 9.5 (a) A declaration for procedure *x* and two invocations in a high-level language, and (b) the assembly language equivalent.

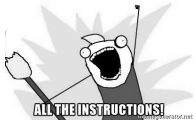
Language specifics

- Documentation
  - Operand order
  - Register naming
  - Syntax
    - Immediate values, register values, memory, etc
- Assembly language does not provide any program control structures, nor enforce any coding style

Intel documentation

- Volume 1: Basic Architecture
  - 482 pages
  - 19 Chapters
  - Includes basic execution environment as well as summary of instructions
    - Groups instructions for programming
      - MMX, SIMD, SSE, etc

- Volume 2: Instruction Set Reference A-Z
  - 2234 pages
  - "Only" 6 chapters
  - Instruction format
  - All of the instructions
  - Safer Mode Extensions



- Volume 3: System Programming Guide
  - 1660 pages
  - 43 Chapters
  - Everything the hardware does to support an OS and how to use it

CPUs have errata

- Ever hear of the original Pentium floating point bug?
  - Could have been errata, but the press picked it up
- Ever find a compiler error?
- Imagine finding a hardware error
  - Probably involves premature baldness
    - Possibly temporary

x86 Assembly

- Unfortunately, x86 is arguably the most complex assembly language around
  - MOV is even Turing complete
- Exposure to most common instructions
  - Focus on ability to read assembled C programs
  - Maybe a little writing
- Differences with x86_64

The Intel Legacy

- Started with 4004
  - 4-bit processor
- 8086, first x86 CPU
  - 16-bits
  - June 8, 1978
  - 5MHz, 8MHz, and 10MHz
- 80186, 80286
- 80386 (SX/DX), 80486 (SX/DX/DX2/etc)

Pentium

- MMX
- SSE, SSE2, SSE3
- X86-64
- AMD-V
- Intel VT-x
- etc
- …and it's all backwards compatible

Fortunately

- Some analyses claim only 14 instructions account for 90% of compiled code

Assembly is symbolic

- label: mnemonic arg1, arg2, arg3
  - Zero to three args
  - Right is source, left is destination
- Mnemonic may represent different (multiple) opcodes

Remember

Figure 5.1 The general instruction format that many processors use. The opcode at the beginning of an instruction determines exactly which operands follow.

64-bit prefix ordering

mov rcx,0x4004e0
48 c7 c1 e0 04 40 00

48: REX.W prefix: 64-bit operand
c7: MOV
c1: ecx (but really rcx)
e0044000: 004004e0

REX prefix

Wat?

Syntax

- Intel
  - [base + index*scale + disp]

```
call DWORD PTR [rbx+rsi*4-0xe8]
mov rax, DWORD PTR [rbp+0x8]
lea rax, [rbx-0xe8]
```

- AT&T
  - disp(base, index, scale)

```
call *-0xe8(%rbx,%rsi,4)
mov 0x8(%rbp), %rax
lea -0xe8(%rbx), %rax
```


Intel vs. AT&T syntax

- Intel
  - Destination comes first

```
mov rbp, rsp
add rax, 0x14
```

- AT&T
  - Reverse

```
mov %rsp, %rbp
add $0x14, %rsp
```

  - Registers prefixed with %, immediate $

Registers

- EIP/RIP
- (E|R)[ABCD]X
  - A: Accumulator
  - B: Base
  - C: Counter
  - D: Data
- ESI, EDI: source and destination pointers for string operations
  - Based off DS in compatibility mode
- ESP, EBP
  - SS segment

EFLAGS/RFLAGS

Figure 3-8. EFLAGS Register

Operand Addressing

- Data for a source operand can be found in…
  - The instruction itself (immediate)
  - A register
  - A memory location
  - An I/O port
- A destination operand can be:
  - A register
  - A memory location
  - An I/O port

Immediate operands

- Example: ADD EAX, 14
- All arithmetic instructions permit an immediate source operand.
- Max value varies, never larger than an unsigned doubleword integer (2^{32})

Register operands

- 64-bit general-purpose registers:
  - RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP, R8-R15
- 32-bit general-purpose registers:
  - EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP, R8D-R15D
- 16-bit general-purpose registers
- 8-bit general-purpose registers
- Segment registers

- RFLAGS
- FPU registers
- MMX, XMM, Control, Debug, and MSR registers
- RDX:RAX register pair (128-bit operand)

Memory operands

- Segment selector and offset
- 64-bit mode segmentation is generally disabled (flat 64-bit linear address space)
  - CS, DS, ES, SS are 0
  - FS and GS can be used as additional base registers

Memory offset

- Displacement: 8, 16, or 32-bits
  - Direct, static value
- Base and Index
  - Values from general-purpose registers
- Scale factor
  - 2, 4, or 8
  - Multiplies Index
- RIP + Displacement
- Result is called an effective address

64-bit prefix ordering

SIB?

- Scale
- Index
- Base

Effective address computation

Data types

LEA

- LEA, the only instruction that performs memory addressing calculations but doesn't actually address memory. LEA accepts a standard memory addressing operand, but does nothing more than store the calculated memory offset in the specified register, which may be any general purpose register.
- What does that give us? Two things that ADD doesn't provide:
  - the ability to perform addition with either two or three operands, and
  - the ability to store the result in any register; not just one of the source operands.

What about 32-bits

- Many systems now are x86_64
- BUT, they can run a lot of 32-bit software
  - "Compatibility mode"
  - Segment registers actually matter
  - Relies on 32-bit registers/addresses/etc
- x86_64 CPUs can switch in and out of compatibility mode with ease
  - Consider system calls for a 64-bit kernel running a 32-bit program

Instruction set

- Data transfer instructions
- Binary arithmetic
- Decimal arithmetic
- Logical
- Shift and rotate
- Bit and byte
- Control
- String

- Flag control ([ER]FLAG)
- Segment registers
- Miscellaneous

Data transfer instructions

- Move data between memory and registers
  - Can be conditional
  - Includes stack access
- CMOV and friends
- XCHG
- BSWAP
- PUSH, PUSHA
- POP, POPA

MOV

- Register to register
- Memory to register
- Register to Memory
- Never memory to memory
  - Remember DMA?

Binary arithmetic instructions

- Basic binary integer computations
- ADD
- SUB
- IMUL, IDIV
- MUL, DIV
- INC, DEC, NEG
- CMP

Decimal arithmetic instructions

- Manipulate BCD data
- Invalid in 64-bit mode

Logical, shift and rotate instructions

- AND, OR, XOR, NOT
- SAR, SHR, SAL, SHL
- ROR, ROL, RCR, RCL

Bit and byte instructions

- BT
- BTS, BTR
  - Semaphores
- SETE, SETZ and friends
- TEST
- CRC32, POPCNT

Control transfer instructions

- JMP
- JE, JZ, JNE, JNZ
- CALL, RET
- INT, IRET
- ENTER, LEAVE

String instructions

- MOVS, MOVSB
  - B/W/D: byte, word, doubleword
- CMPS, CMPSB

Flag control instructions

- STC, CLC
- STD, CLD
- LAHF, SAHF
- PUSHF, PUSHFD
- POPF, POPFD
- STI
- CLI

Questions?