x86 assembly

CS449 Fall 2017
x86 is a CISC

• CISC (Complex Instruction Set Computer) – e.g. x86
  – Hundreds of (complex) instructions
  – Only a handful of registers
• RISC (Reduced Instruction Set Computer) – e.g. MIPS
  – Relatively a few simple instructions
  – Many registers to store intermediate values
• Example complex x86 instruction: \( F2XM1 \)
  – Computes \( 2^x - 1 \), where value of \( X \) must between -1.0 and +1.0.
  – Fast compared to using a combination of simpler instructions
  – But has few use cases and complicates overall design of CPU
• History has proved RISC to be superior
  – Simple CPU design enabled making the common case fast
Why is x86 Still a CISC?

- Two words: backwards compatibility
  - By the time Intel realized mistake it was too late
  - Too many programs have already been produced

- Internally x86 CPUs are designed like RISC
  - Focused on making simple instructions (+,-,*,/) fast
  - Complex instructions supported only for compatibility
    - Internally translated to simple instructions (called micro-ops)

- Nowadays x86 code looks similar to RISC code
  - Complex instructions (e.g. \( \text{F2XM1} \)) almost never used
  - You just need to learn a handful of simple instructions!
32-Bit General Purpose Registers

- **EAX**: Accumulator
- **EBX**: Base
- **ECX**: Counter
- **EDX**: Data
- **ESI**: String Source
- **EDI**: String Destination
Other 32-Bit Registers

- **EIP**: Instruction Pointer (a.k.a. Program Counter)
  - Keeps track of where in code we are executing
- **ESP**: Stack Pointer
  - Points to top of current frame (top of stack)
- **EBP**: Base or Frame Pointer
  - Points to base of current frame
- **EFLAGS** – Flag register
  - Stores flags describing current state of CPU
  - State can be many things. E.g.
    - Result of last comparison instruction (true or false)
    - Whether last add instruction overflowed
    - And more …
Register Subfields

EAX

AH | AL

AX
.file  "asm.c"
.section  .rodata
.LC0:
.string  "hello world!"
.text
.globl  main
.type  main, @function
main:
pushl  %ebp
movl  %esp, %ebp
subl  $8, %esp
andl  $-16, %esp
movl  $.LC0, (%esp) ;1111 1111 1111 0000
subl  $16, %esp
movl  $.LC0, (%esp)
call  puts
movl  $0, %eax
leave
ret
.size  main, .-main
.section  .note.GNU-stack,"",@progbits
.ident  "GCC: (GNU) 3.4.6 20060404 (Red Hat 3.4.6-8)"
AT&T Syntax

• **gcc and gas** use AT&T syntax:
  – Opcode appended by type
    • b – byte (8-bit)
    • w – word (16-bit)
    • l – long (32-bit)
    • q – quad (64-bit)
  – First operand is source
  – Second operand is destination
  –Operand can be one of …
    • Immediate value: $8, $-16, $.LC0, ...
    • Register: %ebp, %esp, %eax, ...
    • Memory dereference: (%esp), (%eax), ...
  – **movl** $ .LC0, (%esp): Move 32-bit value $ .LC0 to location pointed to by %esp (top of stack)
Intel Syntax

- Microsoft (MASM), Intel, NASM
  - Type sizes are spelled out
    - BYTE – 1 byte
    - WORD – 2 bytes
    - DWORD – 4 bytes (double word)
    - QWORD – 8 bytes (quad word)
  - First operand is destination
  - Second operand is source
  - Operand can be one of ...
    - Immediate value: 8, -16, .LC0, ...
    - Register: $ebp, $esp, $eax, ...
    - Memory dereference: [ $esp ], [ $eax ], ...
Intel Hello World

main:

    push    $ebp
    mov     $ebp, $esp
    sub     $esp, 8
    and     $esp, -16 ;1111 1111 1111 0000
    sub     $esp, 16
    mov     DWORD PTR [$$esp], .LC0
    call    puts
    mov     $eax, 0
    leave
    ret
Stacks, Frames, and Calling Conventions

CS449 Fall 2017
Automatic Variables Live in Stack

- Stack starts at 0 bytes on program launch
- Stack grows / shrinks as automatic variables enter / exit scope
- Stack grows downwards

0xc0000000

brk (program break)

0

Stack (Automatic variables)

Heap (Malloc’ed Memory)

Data Segment (Static variables)

Text Segment (Function Code)

Unused memory space
Stack

- **Function Call**
  - A jump to a section of code that ends with a return to the point immediately following the call (the *return address*)

- **Calling Convention**
  - Standard a compiler follows to convert function calls in source code to machine code (e.g. CDECL, STDCALL, FASTCALL)
  - Two functions compiled using different compilers can still call each other if same calling convention is followed

- **Stack**
  - The portion of memory managed in a last-in, first-out (LIFO) fashion, to store function *activation records*
  - Activation record is pushed on function call
  - Activation record is popped on function return
  - Structure of activation record is part of calling convention
Activation Records

• An object containing all the necessary data for a function call, stored on the stack
  – Function parameters
  – Local (automatic) variables
  – Return value
  – Return address (point immediately following call)
  – Temporaries (intermediate values)

• Also called a **Frame**
  – Compiler knows exactly how many local variables there are and how many temporaries are needed
  ➔ Can automatically generate code to allocate frame
Stack Memory Management

```c
int main()
{
    foo();
}
void foo()
{
    bar();
}
void bar()
{
    ...
}
```
Register Value Preservation

• Functions have dedicated stack space (the frame)
• But functions must share only one set of CPU registers:
  – Register values must be saved / restored across function calls
    int main() {
      // Save all CPU register values to stack
      foo(); // foo() modifies CPU registers
      // Restore all CPU register values from stack
    }
  – main() is called **caller** function, foo() is called **callee** function
• We are faced with a choice…
  – Caller saves / restores all CPU registers
  – Callee saves / restores all CPU registers
  – Caller does half, callee does the other half ➔ best choice
Caller-Saved / Callee-Saved

- **Caller-Saved Registers**
  - Set of registers saved / restored by caller function
- **Callee-Saved Registers**
  - Set of registers saved / restored by callee function
- **Example**

```c
int main() {
    // Save caller-saved registers to stack
    foo();
    // Restore caller-saved registers from stack
}

void foo() {
    // Save callee-saved registers to stack
    ...
    // Restore callee-saved registers from stack
}
```
Caller-Saved / Callee-Saved

• Why is this a good idea?
  – Caller can use callee-saved registers without saving / restoring
  – Callee can use caller-saved registers without saving / restoring
  – Gives opportunity for compiler to minimize saving / restoring

• Best case scenario
  ```c
  int main() {
      // Use only callee-saved registers
      foo();
  }
  
  void foo() {
      // Use only caller-saved registers
  }
  ```

• Compiler can get away with no saving / restoring at all!
Caller-Saved / Callee-Saved

• Calling convention dictates which registers are caller-saved and which registers are callee-saved
  – Both caller and callee must follow the same calling convention

• Space to store registers is also considered to be part of the activation record of the function
MIPS Calling Convention

• First 4 arguments $a0-$a3
  – Remainder put on stack

• Return values in $v0-$v1

• $t0-$t9: caller-saved
• $s0-$s7: callee-saved
x86 Calling Convention

- Arguments (usually) passed on the stack
- Return value in $EAX
- $EAX, $ECX, $EDX: generally caller-saved
- $EBP, $EBX, $EDI, $ESI: generally callee-saved
Hello World

.file "asm.c"
.section .rodata

.LC0:
.string "hello world!"
.text
.globl main
.type main, @function

main:
pushl %ebp
movl %esp, %ebp
subl $8, %esp
andl $-16, %esp
subl $16, %esp
movl $.LC0, (%esp)
call puts
movl $0, %eax
leave
ret

.size main, .-main

.ident "GCC: (GNU) 3.4.6 20060404 (Red Hat 3.4.6-8)"
Hello World Stack

$ESP → Old $EBP → $EBP

- pushl %ebp ;save old ebp
- movl %esp, %ebp ;start frame
- subl $8, %esp ;space for locals
- andl $-16, %esp
- subl $16, %esp ;space for args
- movl $.LC0, (%esp)
- call puts
- movl $0, %eax
- leave

• leave pops current frame.

It translates to two instructions:
- movl %ebp, %esp ;collapse frame
- popl %ebp ;restore old ebp
Remember this from Scoping?

```c
#include <stdio.h>
int* foo() {
    int x = 5;
    return &x;
}
void bar() { int y = 10; }
int main() {
    int *p = foo();
    printf("*p=%d\n", *p);
    bar();
    printf("*p=%d\n", *p);
    return 0;
}
```

• The activation records for foo() and bar() landed on the same stack space