x86 assembly

CS449 Fall 2016
x86 is a CISC
x86 is a CISC

- CISC (Complex Instruction Set Computer)
  - Hundreds of (complex) instructions
  - Only a handful of registers
x86 is a CISC

• CISC (Complex Instruction Set Computer)
  – Hundreds of (complex) instructions
  – Only a handful of registers

• RISC (Reduced Instruction Set Computer)
  – Relatively a few simple instructions
  – Many registers to store intermediate values
x86 is a CISC

- CISC (Complex Instruction Set Computer)
  - Hundreds of (complex) instructions
  - Only a handful of registers
- RISC (Reduced Instruction Set Computer)
  - Relatively a few simple instructions
  - Many registers to store intermediate values
- Example x86 instruction: \( \text{F2XM1} \) – Compute \( 2^x - 1 \)
  - Computes the exponential value of 2 to the power of the source operand minus 1. The source operand is located in register ST(0) and the result is also stored in ST(0). The value of the source operand must lie in the range -1.0 to +1.0.
  - No you don’t have to learn this instruction. But old x86 assembly programmers did!
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
- **ESP** – Stack Pointer
- **EBP** – Base or Frame Pointer
- **EFLAGS** – Flag register
Register Subfields

EAX

AH

AL

AX
Hello World

.file "asm.c"
.section .rodata.str1.1,"aMS",@progbits,1
.LC0:
.string "hello world!"
.text
.globl main
.type main, @function
main:
pushl %ebp
movl %esp, %ebp
subl $8, %esp
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:
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)
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
  – Memory dereferences are denoted by ( )
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
  - Memory dereferences are denoted by ( )
  - move $.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
  - Dereferences are denoted by []
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
Process’s Address Space

Stack

Data (Heap)

Globals

Text (Code)
Stack
Stack

• Function Call
  – A control transfer to a segment of code that ends with a return to the point in code immediately after where the call was made (the return address)
Stack

• Function Call
  – A control transfer to a segment of code that ends with a return to the point in code immediately after where the call was made (the return address)

• Calling Convention
  – An agreement, usually created by a system's designers, on how function calls should be implemented
Stack

• Function Call
  – A control transfer to a segment of code that ends with a return to the point in code immediately after where the call was made (the return address)

• Calling Convention
  – An agreement, usually created by a system's designers, on how function calls should be implemented

• Stack
  – A 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
Activation Records
Activation Records

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

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

• Also called a **Frame**
  – Structure of frame is part of calling convention
  – Compiler knows exactly how many local variables there are and how many temporaries are needed
    ➔ Can automatically generate code to allocate frame
```c
int main()
{
    foo();
}
void foo()
{
    bar();
}
void bar()
{
    ...
}
```
Stack Memory Management

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

Frame for main

Stack Grows Downwards
Stack Memory Management

```c
int main()
{
    foo();
}

void foo()
{
    bar();
}

void bar()
{
    ...
}
```

Frame for main

Stack Grows Downwards
int main()
{
    foo();
}
void foo()
{
    bar();
}
void bar()
{
    ...
}
Stack Memory Management

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

Stack Grows Downwards
Stack Memory Management

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

Stack Grows Downwards
int main()
{
    foo();
}
void foo()
{
    bar();
}
void bar()
{
    ...
}
int main()
{
    foo();
}
void foo()
{
    bar();
}
void bar()
{
    ...
}
int main()
{
    foo();
}
void foo()
{
    bar();
}
void bar()
{
    ...
}
int main()
{
    foo();
}
void foo()
{
    bar();
}
void bar()
{
    ...
}
Stack Memory Management

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

Frame for main

Stack Grows Downwards
int main()
{
    foo();
}
void foo()
{
    bar();
}
void bar()
{
    ...
}
Register Value Preservation
Register Value Preservation

- Functions have dedicated stack space (the frame)
Register Value Preservation

• Functions have dedicated stack space (the frame)

• But functions share only one set of registers: Calling convention governs how they are shared
Register Value Preservation

• Functions have dedicated stack space (the frame)

• But functions share only one set of registers: Calling convention governs how they are shared

• Caller-Saved Registers
  – A set of registers that must be saved in the stack by a caller function if they need to be preserved across a function call

```c
int main() {
    // Save %caller_saved_reg to stack
    foo(); // May modify %caller_saved_reg
    // Restore %caller_saved_reg from stack
}
```
Register Value Preservation
Register Value Preservation

• Callee-Saved Registers
  – A set of registers that must be saved in the stack by a called function before modifying, and restored to original values before returning

```c
void foo() {
  // Save %callee_saved_reg to stack
  %callee_saved_reg = ...;
  // Restore %callee_saved_reg from stack
  return;
}
```
Register Value Preservation

• Callee-Saved Registers
  – A set of registers that must be saved in the stack by a called function before modifying, and restored to original values before returning
    void foo() {
        // Save %callee_saved_reg to stack
        %callee_saved_reg = ...;
        // Restore %callee_saved_reg from stack
        return;
    }

• Which should caller function prefer: caller or callee saved?
Register Value Preservation

• Callee-Saved Registers
  – A set of registers that must be saved in the stack by a called function before modifying, and restored to original values before returning
    void foo() {
      // Save %callee_saved_reg to stack
      %callee_saved_reg = ...;
      // Restore %callee_saved_reg from stack
      return;
    }

• Which should caller function prefer: caller or callee saved?
  – Callee saved. No need to save/restore.
Register Value Preservation

- **Callee-Saved Registers**
  - A set of registers that must be saved in the stack by a called function before modifying, and restored to original values before returning

```c
void foo() {
    // Save %callee_saved_reg to stack
    %callee_saved_reg = ...;
    // Restore %callee_saved_reg from stack
    return;
}
```

- Which should caller function prefer: caller or callee saved?
  - Callee saved. No need to save/restore.

- Food for thought: what if there was no convention?
Register Value Preservation

• Callee-Saved Registers
  – A set of registers that must be saved in the stack by a called function before modifying, and restored to original values before returning

```c
void foo() {
  // Save %callee_saved_reg to stack
  %callee_saved_reg = ...;
  // Restore %callee_saved_reg from stack
  return;
}
```

• Which should caller function prefer: caller or callee saved?
  – Callee saved. No need to save/restore.

• Food for thought: what if there was no convention?
  – Both caller / callee would have to save *all* registers to be safe
MIPS Calling Convention

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

• Return values $v0-$v1

• $t0-$t9 are caller-saved
• $s0-$s9 are callee-saved
x86 Calling Convention

- Arguments (usually) passed on the stack
- $EAX is the return value
- $EAX, $ECX, and $EDX are generally caller-saved
- $EBP, $EBX, $EDI, and $ESI are generally callee-saved
Hello World

.file  "asm.c"
.section .rodata.str1.1,"aMS",@progbits,1
.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
.section .note.GNU-stack,"",@progbits
.ident "GCC: (GNU) 3.4.6 20060404 (Red Hat 3.4.6-8)"
Hello World Stack

$ESP

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
Hello World Stack

$ESP → Old $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`
Hello World Stack

$ESP \rightarrow \text{Old $EBP} \rightarrow \text{$EBP} \leftarrow $ESP

- 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
Hello World Stack

Push 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
Hello World Stack

Old $EBP

$ESP

$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
Hello World Stack

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
Hello World Stack

- Old $EBP
- $ESP
- Pointer to string

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
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
Hello World Stack

$ESP  \quad \rightarrow \quad$ESP

<table>
<thead>
<tr>
<th>Old $EBP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Pointer to string</td>
</tr>
</tbody>
</table>

- **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?

#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;
}
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