CSC 2400: Computer Systems

Assembly: Accessing Memory

1

Review:

Instructions to Recognize

2

Arithmetic Instructions (1)

- Two Operand Instructions

`addl`	*Src, Dest*	*Dest = Dest + Src*	
`subl`	*Src, Dest*	*Dest = Dest - Src*	
`imull`	*Src, Dest*	*Dest = Dest * Src*	
`sall`	*Src, Dest*	*Dest = Dest << Src*	
`sarl`	*Src, Dest*	*Dest = Dest >> Src*	**Arithmetic**
`shrl`	*Src, Dest*	*Dest = Dest >> Src*	**Logical**
`xorl`	*Src, Dest*	*Dest = Dest ^ Src*	
`andl`	*Src, Dest*	*Dest = Dest & Src*	
`orl`	*Src, Dest*	*Dest = Dest | Src*	

3

Arithmetic Instructions (2)

- One Operand Instructions

`incl`	*Dest*	*Dest = Dest + 1*
`decl`	*Dest*	*Dest = Dest - 1*
`negl`	*Dest*	*Dest = -Dest*
`notl`	*Dest*	*Dest = ~Dest*

4

Compare and Test Instructions

`cmpl` *Src, Dest*

Compute Dest-Src without setting Dest

`testl` *Src, Dest*

Compute Dest&Src without setting Dest

5

Jump Instructions

- Jump depending on the result of the previous arithmetic instruction:

Jump	Description
`jmp`	Unconditional
`je`	Equal / Zero
`jne`	Not Equal / Not Zero
`js`	Negative
`jns`	Nonnegative
`jg`	Greater (Signed)
`jge`	Greater or Equal (Signed)
`jl`	Less (Signed)
`jle`	Less or Equal (Signed)
`ja`	Above (unsigned)
`jb`	Below (unsigned)

6

Accessing Memory

7

Lecture Goals

- Help you learn…
  - To manipulate data of various sizes
  - To leverage more sophisticated addressing modes
- Focusing on the assembly-language code
  - Rather than the layout of memory for storing data
- Why?
  - Understand the relationship to data types and common programming constructs in higher-level languages

8

Variable Sizes in High-Level Language

- C data types vary in size
  - Character: 1 byte
  - Short, int, and long: varies, depending on the computer
  - Float and double: varies, depending on the computer
  - Pointers: typically 4 bytes
- Programmer-created types
  - Struct: arbitrary size, depending on the fields
- Arrays
  - Multiple consecutive elements of some fixed size
  - Where each element could be a struct

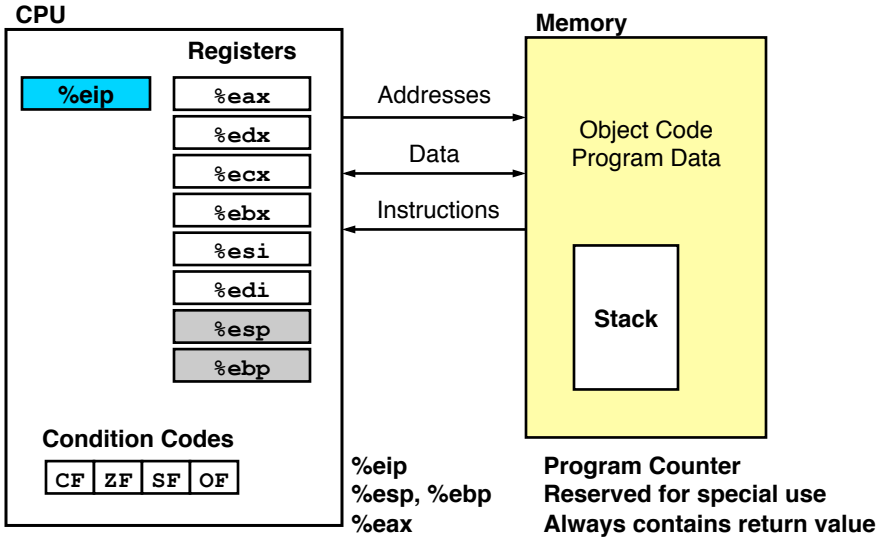
9

Supporting Different Sizes in IA-32

- Three main data sizes
  - Byte (b): 1 byte
  - Word (w): 2 bytes
  - Long (l): 4 bytes
- Separate assembly-language instructions
  - E.g., addb, addw, and addl
- Separate ways to access (parts of) a register
  - E.g., %ah or %al, %ax, and %eax
- Larger sizes (e.g., struct)
  - Manipulated in smaller byte, word, or long units

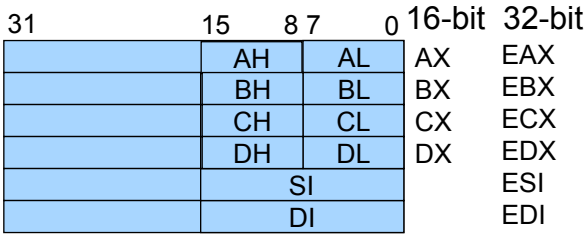
10

IA-32 Architecture



11

IA-32 General Purpose Registers



General-purpose registers

12

Byte Order in Multi-Byte Entities

- Intel is a little endian architecture
  - Least significant byte of multi-byte entity is stored at lowest memory address
  - "Little end goes first"

The int 5 at address 1000:

1000	00000101
1001	00000000
1002	00000000
1003	00000000

- Some other systems use big endian
  - Most significant byte of multi-byte entity is stored at lowest memory address
  - "Big end goes first"

The int 5 at address 1000:

1000	00000000
1001	00000000
1002	00000000
1003	00000101

13

Little Endian Example

```
int main(void) {
  int i=0x003377ff, j;
  unsigned char *p = (unsigned char *) &i;
  for (j=0; j<4; j++)
    printf("Byte %d: %x\n", j, p[j]);
}
```

Output on a little-endian machine

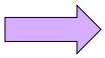
Byte 0: ff
Byte 1: 77
Byte 2: 33
Byte 3: 0

14

C Example: One-Byte Data

Global *char* variable i is in *%al*,
the *lower byte* of the "A" register.

```
char i;
…
if (i > 5) {
  i++;
else
  i--;
}
```



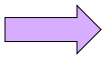
```
      cmpb $5, %al
      jle else
      incb %al
      jmp endif
else:
      decb %al
endif:
```

15

C Example: Four-Byte Data

Global *int* variable i is in *%eax*,
the *full 32 bits* of the "A" register.

```
int i;
…
if (i > 5) {
  i++;
else
  i--;
}
```



```
      cmpl $5, %eax
      jle else
      incl %eax
      jmp endif
else:
      decl %eax
endif:
```

16

Loading and Storing Data

- Processors have many ways to access data
  - Known as “addressing modes”
  - Two simple ways seen in previous examples
- Immediate addressing
  - Example: movl **$0**, %ecx
  - Data (e.g., number “0”) embedded in the instruction
  - Initialize register ECX with zero
- Register addressing
  - Example: movl **%edx**, **%ecx**
  - Choice of register(s) embedded in the instruction
  - Copy value in register EDX into register ECX

17

Direct Addressing

- Load or store from a particular memory location
  - Memory address is embedded in the instruction
  - Instruction reads from or writes to that address
- IA-32 example: movl **2000**, %ecx
  - Four-byte variable located at address 2000
  - Read four bytes starting at address 2000
  - Load the value into the ECX register
- Useful when the address is known in advance
  - Global variables in the Data or BSS sections
- Can use a label for (human) readability
  - E.g., “i” to allow “movl i, %eax”

19

Indirect Addressing

- Load or store from a previously-computed address
  - Register with the address is embedded in the instruction
  - Instruction reads from or writes to that address
- IA-32 example: movl **(%eax)**, %ecx
  - EAX register stores a 32-bit address (e.g., 2000)
  - Read long-word variable stored at that address
  - Load the value into the ECX register
- Useful when address is not known in advance
  - Dynamically allocated data referenced by a pointer
  - The "(%eax)" essentially dereferences a pointer

20

Base Pointer Addressing

- Load or store with an offset from a base address
  - Register storing the base address
  - Fixed offset also embedded in the instruction
  - Instruction computes the address and does access
- IA-32 example: movl 8(%eax), %ecx
  - EAX register stores a 32-bit base address (e.g., 2000)
  - Offset of 8 is added to compute address (e.g., 2008)
  - Read long-word variable stored at that address
  - Load the value into the ECX register
- Useful when accessing part of a larger variable
  - Specific field within a "struct"
  - E.g., if "age" starts at the 8th byte of "student" record

21

Indexed Addressing

- Load or store with an offset and multiplier
  - Fixed based address embedded in the instruction
  - Offset computed by multiplying register with constant
  - Instruction computes the address and does access
- IA-32 example: movl 2000(,%eax,4), %ecx
  - Index register EAX (say, with value of 10)
  - Multiplied by a multiplier of 1, 2, 4, or 8 (say, 4)
  - Added to a fixed base of 2000 (say, to get 2040)
- Useful to iterate through an array (e.g., a[i])
  - Base is the start of the array (i.e., "a")
  - Register is the index (i.e., "i")
  - Multiplier is the size of the element (e.g., 4 for "int")

22

Indexed Addressing Example

```
int a[20];
...
int i, sum=0;
for (i=0; i<20; i++)
    sum += a[i];
```

global variable

```
    movl $0, %eax
    movl $0, %ebx
sumloop:
    movl a(,%eax,4), %ecx
    addl %ecx, %ebx
    incl %eax
    cmpl $19, %eax
    jle sumloop
```

EAX: i
EBX: sum
ECX: temporary

Indexed Address: D(Base,Index,Scale)

Address = Base + Index * Scale + D

Examples:

- Displacement — `movl foo, %ebx`
- Base — `movl (%eax), %ebx`
- Base + disp — `movl foo(%eax), %ebx` / `movl 1(%eax), %ebx`
- (Index * scale) + disp — `movl (,%eax,4), %ebx`
- Base + (index * scale) + disp — `movl foo(%edx,%eax,4),%ebx`

24

Address Computation Examples

%edx	0xf000
%ecx	0x100

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

25

Address Computation Instruction

- **`leal`** *Src*,*Dest*
  - *Src* is address mode expression
  - Set *Dest* to address denoted by expression
- Uses
  - Computing address without doing memory reference
    - E.g., translation of `p = &x[i];`
  - Computing arithmetic expressions of the form x + k*y + z
    - k = 1, 2, 4, or 8.
    - z is an 8-bit signed constant

Using `leal` for Arithmetic Expressions

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

```
arith:
    pushl %ebp
    movl %esp,%ebp

    movl 8(%ebp),%eax
    movl 12(%ebp),%edx
    leal (%edx,%eax),%ecx
    leal (%edx,%edx,2),%edx
    sall $4,%edx
    addl 16(%ebp),%ecx
    leal 4(%edx,%eax),%eax
    imull %ecx,%eax

    movl %ebp,%esp
    popl %ebp
    ret
```

Set Up

Body

Finish

Understanding `arith`

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

```
x is at address ebp+8
y is at address ebp+12
x is at address ebp+16

    To be explained in next lecture
```

```
      movl 8(%ebp),%eax         # eax =
      movl 12(%ebp),%edx        # edx =
      leal (%edx,%eax),%ecx     # ecx =
      leal (%edx,%edx,2),%edx   # edx =
      sall $4,%edx              # edx =
      addl 16(%ebp),%ecx        # ecx =
      leal 4(%edx,%eax),%eax    # eax =
      imull %ecx,%eax           # eax =
```

28

Understanding `arith`

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

```
x is at address ebp+8
y is at address ebp+12
x is at address ebp+16

    To be explained in next lecture
```

```
      movl 8(%ebp),%eax         # eax = x
      movl 12(%ebp),%edx        # edx = y
      leal (%edx,%eax),%ecx     # ecx = x+y   (t1)
      leal (%edx,%edx,2),%edx   # edx = 3*y
      sall $4,%edx              # edx = 48*y (t4)
      addl 16(%ebp),%ecx        # ecx = z+t1 (t2)
      leal 4(%edx,%eax),%eax    # eax = 4+t4+x (t5)
      imull %ecx,%eax           # eax = t5*t2 (return value)
```

29

Data Access Methods: Summary

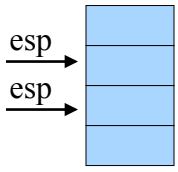
- Immediate addressing: data stored in the instruction itself
  - **movl $10, %ecx**
- Register addressing: data stored in a register
  - **movl %eax, %ecx**
- Direct addressing: address stored in instruction
  - **movl foo, %ecx**
- Indirect addressing: address stored in a register
  - **movl (%eax), %ecx**
- Base pointer addressing: includes an offset as well
  - **movl 4(%eax), %ecx**
- Indexed addressing: instruction contains base address, and specifies an index register and a multiplier (1, 2, 4, or 8)
  - **movl 2000(,%eax,1), %ecx**

30

Data Transfer Instructions

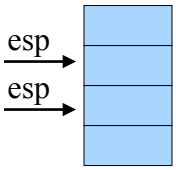
- **`mov{b,w,l} source, dest`**
  - General move instruction
- **`push{w,l} source`**

```
pushl %ebx    # equivalent instructions
              subl $4, %esp
              movl %ebx, (%esp)
```



- **`pop{w,l} dest`**

```
popl %ebx     # equivalent instructions
              movl (%esp), %ebx
              addl  $4, %esp
```



31

Conclusions

- Accessing data
  - Byte, word, and long-word data types
  - Wide variety of addressing modes

- Next time
  - Calling functions, using the stack