Assembly Language: Part 2
Goals of this Lecture

Help you learn:
- Intermediate aspects of IA-32 assembly language…
- Control flow with signed integers
- Control flow with unsigned integers
- Arrays
- Structures
Agenda

Flattened C code
Control flow with signed integers
Control flow with unsigned integers
Arrays
Structures
Flattened C Code

Problem

• Translating from C to assembly language is difficult when the C code contains nested statements

Solution

• Flatten the C code to eliminate all nesting
### Flattened C Code

<table>
<thead>
<tr>
<th>C</th>
<th>Flattened C</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (expr)</td>
<td>if (! expr) goto endif1;</td>
</tr>
<tr>
<td>{</td>
<td>statement1;</td>
</tr>
<tr>
<td>statement1;</td>
<td>...</td>
</tr>
<tr>
<td>statementN;</td>
<td>statementN;</td>
</tr>
<tr>
<td>}</td>
<td>endif1:</td>
</tr>
<tr>
<td></td>
<td>if (! expr) goto else1;</td>
</tr>
<tr>
<td>elseif</td>
<td>statement1;</td>
</tr>
<tr>
<td>{</td>
<td>...</td>
</tr>
<tr>
<td>statementT1;</td>
<td>statementT1;</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>statementTN;</td>
<td>statementTN;</td>
</tr>
<tr>
<td>}</td>
<td>goto endif1;</td>
</tr>
<tr>
<td>else</td>
<td>else1:</td>
</tr>
<tr>
<td>{</td>
<td>statementF1;</td>
</tr>
<tr>
<td>statementF1;</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>statementFN;</td>
<td>statementFN;</td>
</tr>
<tr>
<td>}</td>
<td>endif1:</td>
</tr>
</tbody>
</table>
while (expr)
{
  statement1;
  ...  
  statementN;
}

for (expr1; expr2; expr3)
{
  statement1;
  ...  
  statementN;
}

See Bryant & O’Hallaron book for faster patterns
Agenda

- Flattened C code
- Control flow with signed integers
- Control flow with unsigned integers
- Arrays
- Structures
if Example

C

```c
int i;
...
if (i < 0)
  i = -i;
```

Flattened C

```c
int i;
...
if (i >= 0) goto endif1;
  i = -i;
endif1:
```
if Example

Flattened C

```c
int i;
...
  if (i >= 0) goto endif1;
  i = -i;
endif1:
```

Assem Lang

```asm
.section "bss"
i: .skip 4
...
.section "text"
...
cmpl $0, i
jge endif1
negl i
endif1:
```

Note:

- `cmp` instruction (counterintuitive operand order)
  - Sets CC bits in EFLAGS register
- `jge` instruction (conditional jump)
  - Examines CC bits in EFLAGS register
if...else Example

C

```c
int i;
int j;
int smaller;
...
if (i < j)
    smaller = i;
else
    smaller = j;
```

Flattened C

```c
int i;
int j;
int smaller;
...
if (i >= j) goto else1;
smaller = i;
goto endif1;
else1:
    smaller = j;
endif1:
```
if...else Example

Flattened C

```c
int i;
int j;
int smaller;
...
if (i >= j) goto else1;
    smaller = i;
    goto endif1;
else1:
    smaller = j;
endif1:
```

Assem Lang

```asm
.section ".bss"
i:       .skip 4
j:       .skip 4
smaller: .skip 4
...
.section ".text"
...
movl i, %eax
cmpl j, %eax
jge else1
movl i, %eax
movl %eax, smaller
jmp endif1
else1:
movl j, %eax
movl %eax, smaller
endif1:
```

Note:

`jmp` instruction
(unconditional jump)
while Example

C

```c
int fact;
int n;
...
fact = 1;
while (n > 1)
    { fact *= n;
      n--;
    }
```

Flattened C

```c
int fact;
int n;
...
fact = 1;
loop1:
    if (n <= 1) goto endloop1;
    fact *= n;
    n--;
    goto loop1;
endloop1:
```
while Example

Flattened C

```c
int fact;
int n;
...
    fact = 1;
loop1:
    if (n <= 1) goto endloop1;
    fact *= n;
    n--;
    goto loop1;
endloop1:
```

Note:
- `jle` instruction (conditional jump)
- `imul` instruction

Assem Lang

```assembly
.section ".bss"
fact: .skip 4
n:  .skip 4
...
.section ".text"
...
movl $1, fact
loop1:
    cmpl $1, n
    jle endloop1
    movl fact, %eax
    imull n
    movl %eax, fact
    decl n
    jmp loop1
endloop1:
```
for Example

C

```c
int power = 1;
int base;
int exp;
int i;
...
for (i = 0; i < exp; i++)
    power *= base;
```

Flattened C

```c
int power = 1;
int base;
int exp;
int i;
...
    i = 0;
loop1:
    if (i >= exp) goto endloop1;
    power *= base;
    i++;
    goto loop1;
endloop1:
```
for Example

Flattened C

```c
int power = 1;
int base;
int exp;
int i;
...
i = 0;
loop1:
    if (i >= exp) goto endloop1;
    power *= base;
i++;
goto loop1;
enloop1:
```

Assem Lang

```assembly
section ".data"
power: .long 1
    .section ".bss"
base: .skip 4
exp: .skip 4
i: .skip 4
...
    .section ".text"
...
    movl $0, i
    loop1:
        movl i, %eax
cmpl exp, %eax
jge endloop1
    movl power, %eax
    imull base
    movl %eax, power
    incl i
    jmp loop1
endloop1:
```
Control Flow with Signed Integers

Comparing signed integers

```plaintext
cmp{1,w,b} srcIRM, destRM  Compare dest with src
```

- Sets CC bits in the EFLAGS register
- Beware: operands are in counterintuitive order
- Beware: many other instructions set CC bits
  - Conditional jump should `immediately` follow `cmp`
## Control Flow with Signed Integers

### Unconditional jump

<table>
<thead>
<tr>
<th>mnemonic</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>jmp label</code></td>
<td>Jump to label</td>
</tr>
</tbody>
</table>

### Conditional jumps after comparing signed integers

<table>
<thead>
<tr>
<th>mnemonic</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>je label</code></td>
<td>Jump to label if equal</td>
</tr>
<tr>
<td><code>jne label</code></td>
<td>Jump to label if not equal</td>
</tr>
<tr>
<td><code>jl label</code></td>
<td>Jump to label if less</td>
</tr>
<tr>
<td><code>jle label</code></td>
<td>Jump to label if less or equal</td>
</tr>
<tr>
<td><code>jg label</code></td>
<td>Jump to label if greater</td>
</tr>
<tr>
<td><code>jge label</code></td>
<td>Jump to label if greater or equal</td>
</tr>
</tbody>
</table>

- Examine CC bits in EFLAGS register
Agenda

- Flattened C
- Control flow with signed integers
- Control flow with unsigned integers
- Arrays
- Structures
Signed vs. Unsigned Integers

In C
- Integers are signed or unsigned
- Compiler generates assembly language instructions accordingly

In assembly language
- Integers are neither signed nor unsigned
- Distinction is in the instructions used to manipulate them

Distinction matters for
- Multiplication and division
- Control flow
Handling Unsigned Integers

Multiplication and division

- Signed integers: `imul, idiv`
- Unsigned integers: `mul, div`

Control flow

- Signed integers: `cmp + {je, jne, jl, jle, jg, jge}`
- Unsigned integers: “unsigned `cmp`” + `{je, jne, jl, jle, jg, jge}`
- Unsigned integers: `cmp + {je, jne, jb, jbe, ja, jae}`
while Example

**C**

```c
unsigned int fact;
unsigned int n;
...
fact = 1;
while (n > 1)
{ fact *= n;
  n--;
}
```

**Flattened C**

```c
unsigned int fact;
unsigned int n;
...
    fact = 1;
loop1:
    if (n <= 1) goto endloop1;
    fact *= n;
    n--;
    goto loop1;
endloop1:
```
while Example

Flattened C

```c
unsigned int fact;
unsigned int n;
...
  fact = 1;
loop1:
  if (n <= 1) goto endloop1;
  fact *= n;
  n--;
  goto loop1;
endloop1:
```

Assem Lang

```assembly
.section " .bss"
fact: .skip 4
n: .skip 4
...
.section " .text"
...
    movl $1, fact
loop1:
    cmpl $1, n
    jbe endloop1
    movl fact, %eax
    mull n
    movl %eax, fact
    decl n
    jmp loop1
endloop1:
```

Note:
- **jbe** instruction (instead of **jle**)
- **mull** instruction (instead of **imull**).
for Example

C

```c
unsigned int power = 1;
unsigned int base;
unsigned int exp;
unsigned int i;
...
for (i = 0; i < exp; i++)
    power *= base;
```

Flattened C

```c
unsigned int power = 1;
unsigned int base;
unsigned int exp;
unsigned int i;
...
i = 0;
loop1:
    if (i >= exp) goto endloop1;
    power *= base;
i++;
goto loop1;
endloop1:
```
for Example

Flattened C

```c
unsigned int power = 1;
unsigned int base;
unsigned int exp;
unsigned int i;

i = 0;
loop1:
  if (i >= exp) goto endloop1;
  power *= base;
  i++;
  goto loop1;
endloop1:
```

Note:
- `jae` instruction (instead of `jge`)
- `mull` instruction (instead of `imull`)

Assem Lang

```assembly
.section ".data"
power: .long 1

.section ".bss"
base: .skip 4
exp: .skip 4
i: .skip 4

... 

.section ".text"

... 

movl $0, i
loop1:
  movl i, %eax
cmpl exp, %eax
jae endloop1
  movl power, %eax
  mull base
  movl %eax, power
  incl i
  jmp loop1
endloop1:
```
Control Flow with Unsigned Integers

Comparing unsigned integers

- Same as comparing signed integers

Conditional jumps after comparing unsigned integers

```
je label     Jump to label if equal
jne label    Jump to label if not equal
jb label     Jump to label if below
jbe label    Jump to label if below or equal
ja label     Jump to label if above
jae label    Jump to label if above or equal
```

- Examine CC bits in EFLAGS register
Agenda

Flattened C
Control flow with signed integers
Control flow with unsigned integers

Arrays
Structures
Arrays: Indirect Addressing

C

```c
int a[100];
int i;
int n;
...
i = 3;
...
n = a[i]
```

Assem Lang

```assembly
.section "bss"
a: .skip 400
i: .skip 4
n: .skip 4
...
.section "text"
...
movl $3, i
...
movl i, %eax
sall $2, %eax
addl $a, %eax
movl (%eax), %ecx
movl %ecx, n
...
```

One step at a time…
Arrays: Indirect Addressing

Assem Lang

```assembly
.section ".bss"
a: .skip 400
i: .skip 4
n: .skip 4
...
.section ".text"
...
movl $3, i
...
movl i, %eax
sal $2, %eax
addl $a, %eax
movl (%eax), %ecx
movl %ecx, n
...
```

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>0 1000</td>
</tr>
<tr>
<td>EBX</td>
<td>1 1004</td>
</tr>
<tr>
<td>ECX</td>
<td>2 1008</td>
</tr>
<tr>
<td></td>
<td>3 123  1012</td>
</tr>
<tr>
<td></td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>100 1396</td>
</tr>
<tr>
<td></td>
<td>i 3 1400</td>
</tr>
<tr>
<td></td>
<td>n 1404</td>
</tr>
</tbody>
</table>

28
Arrays: Indirect Addressing

Assem Lang

```
.section "bss"
a: .skip 400
i: .skip 4
n: .skip 4
...
.section "text"
...
movl $3, i
...
movl i, %eax
sall $2, %eax
addl $a, %eax
movl (%eax), %ecx
movl %ecx, n
...
```

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>i</td>
</tr>
<tr>
<td></td>
<td>n</td>
</tr>
</tbody>
</table>

Registers EAX, EBX, ECX are shown with their contents.
### Arrays: Indirect Addressing

#### Assem Lang

```assembly
.section "bss"
a: .skip 400
i: .skip 4
n: .skip 4

...  
.section "text"
...
movl $3, i
...
movl i, %eax
sall $2, %eax
addl $a, %eax
movl (%eax), %ecx
movl %ecx, n
...
```

#### Registers

<table>
<thead>
<tr>
<th></th>
<th>EAX</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EBX</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ECX</td>
<td></td>
</tr>
</tbody>
</table>

#### Memory

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1004</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1008</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>123</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>123</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1396</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>1400</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1404</td>
</tr>
</tbody>
</table>
Arrays: Indirect Addressing

Assem Lang

```assembly
.section ".bss"
a: .skip 400
i: .skip 4
n: .skip 4

,...
.section ".text"
...
movl $3, i
...
movl i, %eax
sall $2, %eax
addl $a, %eax
movl (%eax), %ecx
movl %ecx, n
...```

## Registers

<table>
<thead>
<tr>
<th>Memory</th>
<th>EAX</th>
<th>EBX</th>
<th>ECX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Memory

<table>
<thead>
<tr>
<th>Memory</th>
<th>EAX</th>
<th>EBX</th>
<th>ECX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>123</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
<th>EAX</th>
<th>EBX</th>
<th>ECX</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
<th>EAX</th>
<th>EBX</th>
<th>ECX</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

31
Arrays: Indirect Addressing

Assem Lang

```assembly
.section "\.bss"
a: .skip 400
i: .skip 4
n: .skip 4

... .section "\.text"
... movl $3, i
... movl i, %eax
sal $2, %eax
addl $a, %eax
movl (%eax), %ecx
movl %ecx, n
...```

Registers

<table>
<thead>
<tr>
<th>EAX</th>
<th>1012</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBX</td>
<td></td>
</tr>
<tr>
<td>ECX</td>
<td>123</td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>0</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1004</td>
</tr>
<tr>
<td>2</td>
<td>1008</td>
</tr>
<tr>
<td>3</td>
<td>1012</td>
</tr>
</tbody>
</table>

```
movl $3, i
movl i, %eax
sal $2, %eax
addl $a, %eax
movl (%eax), %ecx
movl %ecx, n
...```

Note:

Indirect addressing
Arrays: Indirect Addressing

Assem Lang

```
  .section " .bss"
  a: .skip 400
  i: .skip 4
  n: .skip 4
  ...
  .section " .text"
  ...
  movl $3, i
  ...
  movl i, %eax
  sall $2, %eax
  addl $a, %eax
  movl (%eax), %ecx
  movl %ecx, n
  ...
```

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX 1012</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1000</td>
</tr>
<tr>
<td>EBX</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1004</td>
</tr>
<tr>
<td>ECX 123</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1008</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>1012</td>
</tr>
<tr>
<td></td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>1396</td>
</tr>
<tr>
<td></td>
<td>i</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>1400</td>
</tr>
<tr>
<td></td>
<td>n</td>
</tr>
<tr>
<td></td>
<td>123</td>
</tr>
<tr>
<td></td>
<td>1404</td>
</tr>
</tbody>
</table>
Arrays: Base+Disp Addressing

C

int a[100];
int i;
int n;
...
i = 3;
...
n = a[i]
...

Assem Lang

.section ".bss"
.a: .skip 400
i: .skip 4
n: .skip 4
...
.section ".text"
...
.movl $3, i
...
.movl i, %eax
.sall $2, %eax
.movl a(%eax), %ecx
.movl %ecx, n
...

One step at a time…
Arrays: Base+Disp Addressing

Assem Lang

```
.section "bss"
a: .skip 400
i: .skip 4
n: .skip 4
...
.section "text"
...
movl $3, i
...
movl i, %eax
sal $2, %eax
movl a(%eax), %ecx
movl %ecx, n
...
```

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>0</td>
</tr>
<tr>
<td>EAX</td>
<td>1</td>
</tr>
<tr>
<td>EAX</td>
<td>2</td>
</tr>
<tr>
<td>EAX</td>
<td>3</td>
</tr>
<tr>
<td>ECX</td>
<td>...</td>
</tr>
<tr>
<td>ECX</td>
<td>100</td>
</tr>
<tr>
<td>ECX</td>
<td>i</td>
</tr>
<tr>
<td>ECX</td>
<td>n</td>
</tr>
</tbody>
</table>

Memory addresses:
- Base address: 0x1000
- Displacement: 0x1000, 0x1004, 0x1008, 0x1012
- Array index: i = 3
Assem Lang

```assembly
.section "bss"
a: .skip 400
i: .skip 4
n: .skip 4
...
.section "text"
...
movl $3, i
...
movl i, %eax
sall $2, %eax
movl a(%eax), %ecx
movl %ecx, n
...
```

### Registers

- **EAX**: 3
- **EBX**: 
- **ECX**: 

### Memory

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>1004</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>1008</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>123</td>
<td></td>
<td>1012</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
<td>1396</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>3</td>
<td></td>
<td>1400</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
<td></td>
<td>1404</td>
<td></td>
</tr>
</tbody>
</table>
Arrays: Base+Disp Addressing

Assem Lang

```
.section "\".bss\"
.a: .skip 400
.i: .skip 4
.n: .skip 4
...
.section "\".text\"
...
movl $3, i
...
movl i, %eax
sall $2, %eax
movl a(%eax), %ecx
movl %ecx, n
...
```
Arrays: Base+Disp Addressing

Assem Lang

```
.section "".bss"
a: .skip 400
i: .skip 4
n: .skip 4
...
.section "".text"
...
movl $3, i
...
movl i, %eax
sal $2, %eax
movl a(%eax), %ecx
movl %ecx, n
...
```

Note:

Base+displacement addressing
Arrays: Base+Disp Addressing

Assem Lang

```assembly
.section ".bss"
a: .skip 400
i: .skip 4
n: .skip 4
...
.section ".text"
...
movl $3, i
...
movl i, %eax
sal $2, %eax
movl a(%eax), %ecx
movl %ecx, n
...
```

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX 12</td>
<td>0 1000</td>
</tr>
<tr>
<td>EBX</td>
<td>1 1004</td>
</tr>
<tr>
<td>ECX 123</td>
<td>2 1008</td>
</tr>
<tr>
<td></td>
<td>3 1012</td>
</tr>
<tr>
<td></td>
<td>a 123</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>100 1396</td>
</tr>
<tr>
<td>i</td>
<td>3 1400</td>
</tr>
<tr>
<td>n 123</td>
<td>1404</td>
</tr>
</tbody>
</table>
Arrays: Scaled Indexed Addressing

C

```c
int a[100];
int i;
int n;
...
i = 3;
...
n = a[i]
...
```

Assem Lang

```assembly
.section ".bss"
a: .skip 400
i: .skip 4
n: .skip 4
...
.section ".text"
...
.movl $3, i
...
.movl i, %eax
.movl a(%eax,4), %ecx
.movl %ecx, n
...
```

One step at a time…
Arrays: Scaled Indexed Addressing

Assem Lang

```assembly
.section " .bss"
a: .skip 400
i: .skip 4
n: .skip 4
...
.section " .text"
...
movl $3, i
...
movl i, %eax
movl a(,%eax,4), %ecx
movl %ecx, n
...
```

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>0 1000</td>
</tr>
<tr>
<td>EBX</td>
<td>1 1004</td>
</tr>
<tr>
<td>ECX</td>
<td>2 1008</td>
</tr>
<tr>
<td></td>
<td>3 123 1012</td>
</tr>
<tr>
<td></td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>3 123 1012</td>
</tr>
<tr>
<td></td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>100 1396</td>
</tr>
<tr>
<td></td>
<td>i 3 1400</td>
</tr>
<tr>
<td></td>
<td>n 1404</td>
</tr>
</tbody>
</table>
Arrays: Scaled Indexed Addressing

Assem Lang

```assembly
.section ".bss"
a: .skip 400
i: .skip 4
n: .skip 4

... .section ".text"
... movl $3, i
... movl i, %eax
... movl a(,%eax,4), %ecx
movl %ecx, n
```

Registers

- EAX: 3
- EBX: 
- ECX: 

Memory

<table>
<thead>
<tr>
<th>i</th>
<th>n</th>
<th>a</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>1000</td>
<td>1004</td>
<td>1008</td>
<td>1012</td>
</tr>
<tr>
<td>i</td>
<td>n</td>
<td>a</td>
<td>100</td>
<td>3</td>
<td></td>
<td>123</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1396</td>
<td>1400</td>
<td>1404</td>
<td></td>
</tr>
</tbody>
</table>
Arrays: Scaled Indexed Addressing

Assem Lang

```assembly
section ".bss"
a: .skip 400
i: .skip 4
n: .skip 4
...
section ".text"
...
movl $3, i
...
movl i, %eax
movl a(,%eax,4), %ecx
movl %ecx, n
...
```

**Registers**
- EAX: 3
- EBX: 
- ECX: 123

**Memory**
- 0 1000
- 1 1004
- 2 1008
- 3 123 1012
- i 3 1400
- n 1404

**Note:**
- Scaled indexed addressing
Arrays: Scaled Indexed Addressing

Assem Lang

```
.section "".bss"
a: .skip 400
i: .skip 4
n: .skip 4
...
.section "".text"
...
movl $3, i
...
movl i, %eax
movl a(%eax,4), %ecx
movl %ecx, n
...
```

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX 12</td>
<td>0 1000</td>
</tr>
<tr>
<td>EBX</td>
<td>1 1004</td>
</tr>
<tr>
<td>ECX 123</td>
<td>2 1008</td>
</tr>
<tr>
<td></td>
<td>3 1012</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>100 1396</td>
</tr>
<tr>
<td>i 3</td>
<td>1400</td>
</tr>
<tr>
<td>n 123</td>
<td>1404</td>
</tr>
</tbody>
</table>

Memory addresses:
- 0: 1000
- 1: 1004
- 2: 1008
- 3: 1012
- 100: 1396
- i: 1400
- n: 1404
Generalization: Memory Operands

Full form of memory operands:

\[ \text{displacement}(\text{base}, \text{index}, \text{scale}) \]

- \textit{displacement} is an integer or a label (default = 0)
- \textit{base} is a register
- \textit{index} is a register
- \textit{scale} is 1, 2, 4, or 8 (default = 1)

Meaning

- Compute the sum
  \[ (\text{displacement}) + (\text{contents of base}) + ((\text{contents of index}) \times (\text{scale})) \]
- Consider the sum to be an address
- Load from (or store to) that address

Note:

- All other forms are subsets of the full form…
Valid subsets:

- **Direct addressing**
  - displacement

- **Indirect addressing**
  - (base)

- **Base+displacement addressing**
  - displacement(base)

- **Indexed addressing**
  - (base, index)
  - displacement(base,index)

- **Scaled indexed addressing**
  - (,index, scale)
  - displacement(,index,scale)
  - (base,index,scale)
  - displacement(base,index,scale)
Operand Examples

Immediate operands
- $5 => use the number 5 (i.e. the number that is available immediately within the instruction)
- $i => use the address denoted by i (i.e. the address that is available immediately within the instruction)

Register operands
- %eax => read from (or write to) register EAX

Memory operands: direct addressing
- 5 => load from (or store to) memory at address 5 (silly; seg fault)
- i => load from (or store to) memory at the address denoted by i

Memory operands: indirect addressing
- (%eax) => consider the contents of EAX to be an address; load from (or store to) that address
Operand Examples

Memory operands: **base+displacement addressing**

- \( 5(\%\text{eax}) \) => compute the sum (5) + (contents of EAX); consider the sum to be an address; load from (or store to) that address
- \( i(\%\text{eax}) \) => compute the sum (address denoted by i) + (contents of EAX); consider the sum to be an address; load from (or store to) that address

Memory operands: **indexed addressing**

- \( 5(\%\text{eax},\%\text{ecx}) \) => compute the sum (5) + (contents of EAX) + (contents of ECX); consider the sum to be an address; load from (or store to) that address
- \( i(\%\text{eax},\%\text{ecx}) \) => compute the sum (address denoted by i) + (contents of EAX) + (contents of ECX); consider the sum to be an address; load from (or store to) that address
Operand Examples

Memory operands: **scaled indexed addressing**

- \(5(\%eax,\%ecx,4)\) => compute the sum \((5) + (\text{contents of EAX}) + ((\text{contents of ECX}) \times 4)\); consider the sum to be an address; load from (or store to) that address
- \(i(\%eax,\%ecx,4)\) => compute the sum (address denoted by \(i\)) + (\text{contents of EAX}) + ((\text{contents of ECX}) \times 4)\); consider the sum to be an address; load from (or store to) that address
Aside: The lea Instruction

**lea**: load effective address

- Unique instruction: suppresses memory load/store

**Example**

- `movl 5(%eax), %ecx`
  - Compute the sum (5) + (contents of EAX); consider the sum to be an address; load 4 bytes from that address into ECX

- `leal 5(%eax), %ecx`
  - Compute the sum (5) + (contents of EAX); move that sum to ECX

**Useful for**

- Computing an address, e.g. as a function argument
  - See precept code that calls `scanf()`
- Some quick-and-dirty arithmetic

What is the effect of this?

`leal (%eax,%eax,4),%eax`
Agenda

- Flattened C
- Control flow with signed integers
- Control flow with unsigned integers
- Arrays
- Structures
Structures: Indirect Addressing

C

```c
struct S
{ int i;
   int j;
};
...
struct S myStruct;
...
myStruct.i = 18;
...
myStruct.j = 19;
```

Assem Lang

```
.section ".bss"
myStruct: .skip 8
...
.section ".text"
...
    movl $myStruct, %eax
    movl $18, (%eax)
...
    movl $myStruct, %eax
    addl $4, %eax
    movl $19, (%eax)
```

Note:

Indirect addressing
Structures: Base+Disp Addressing

C

```c
struct S {
    int i;
    int j;
};
...
struct S myStruct;
...
myStruct.i = 18;
...
myStruct.j = 19;
```

Assem Lang

```assembly
.section "bss"
myStruct: .skip 8
...
.section "text"
...
movl $0, %eax
movl $18, myStruct(%eax)
...
movl $4, %eax
movl $19, myStruct(%eax)
```

Note:
Base+displacement addressing
Beware: Compiler sometimes inserts padding after fields
## Structures: Padding

### IA-32/Linux/gcc217 rules

<table>
<thead>
<tr>
<th>Data type</th>
<th>Must begin at address that is evenly divisible by:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(unsigned) char</td>
<td>1</td>
</tr>
<tr>
<td>(unsigned) short</td>
<td>2</td>
</tr>
<tr>
<td>(unsigned) int</td>
<td>4</td>
</tr>
<tr>
<td>(unsigned) long</td>
<td>4</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>4</td>
</tr>
<tr>
<td>long double</td>
<td>4</td>
</tr>
<tr>
<td>any structure</td>
<td>4</td>
</tr>
<tr>
<td>any pointer</td>
<td>4</td>
</tr>
</tbody>
</table>

- Can override using compiler options (e.g. -malign-double)
Summary

Intermediate aspects of IA-32 assembly language…

Flattened C code

Control transfer with signed integers

Control transfer with unsigned integers

Arrays
  • Full form of instruction operands

Structures
  • Padding
Setting and using CC bits in EFLAGS register
Setting Condition Code Bits

Question
  • How does `cmp` set condition code bits in EFLAGS register?

Answer
  • (See following slides)
Condition Code Bits

Condition code bits

• **ZF**: zero flag: set to 1 iff result is **zero**
• **SF**: sign flag: set to 1 iff result is **negative**
• **CF**: carry flag: set to 1 iff **unsigned overflow** occurred
• **OF**: overflow flag: set to 1 iff **signed overflow** occurred
Example: `addl src, dest`

- Compute sum `(dest+src)`
- Assign sum to `dest`
- `ZF`: set to 1 iff `sum == 0`
- `SF`: set to 1 iff `sum < 0`
- `CF`: set to 1 iff unsigned overflow
  - Set to 1 iff `sum < src`
- `OF`: set if signed overflow
  - Set to 1 iff
    - `(src>0 && dest>0 && sum<0) || (src<0 && dest<0 && sum>=0)`
Condition Code Bits

Example: **subl** src, dest
- Compute sum (dest+(-src))
- Assign sum to dest
- ZF: set to 1 iff sum == 0
- SF: set to 1 iff sum < 0
- CF: set to 1 iff unsigned overflow
  - Set to 1 iff dest<src
- OF: set to 1 iff signed overflow
  - Set to 1 iff
    - (dest>0 && src<0 && sum<0) ||
    - (dest<0 && src>0 && sum>=0)

Example: **cmpl** src, dest
- Same as subl
- But does not affect dest
Question
  • How do conditional jump instructions use condition code bits in EFLAGS register?

Answer
  • (See following slides)
Conditional Jumps: Unsigned

After comparing **unsigned** data

<table>
<thead>
<tr>
<th>Jump Instruction</th>
<th>Use of CC Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>je label</td>
<td>ZF</td>
</tr>
<tr>
<td>jne label</td>
<td>~ZF</td>
</tr>
<tr>
<td>jb label</td>
<td>CF</td>
</tr>
<tr>
<td>jae label</td>
<td>~CF</td>
</tr>
<tr>
<td>jbe label</td>
<td>CF</td>
</tr>
<tr>
<td>ja label</td>
<td>~(CF</td>
</tr>
</tbody>
</table>

**Note:**
- If you can understand why **jb** jumps iff CF
- ... then the others follow
Conditional Jumps: Unsigned

Why does jb jump iff CF? Informal explanation:

(1) largenum – smallnum (not below)
   • Correct result
   • => CF=0 => don’t jump

(2) smallnum – largenum (below)
   • Incorrect result
   • => CF=1 => jump
Conditional Jumps: Signed

After comparing **signed** data

<table>
<thead>
<tr>
<th>Jump Instruction</th>
<th>Use of CC Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>je label</td>
<td>ZF</td>
</tr>
<tr>
<td>jne label</td>
<td>~ZF</td>
</tr>
<tr>
<td>jl label</td>
<td>OF ^ SF</td>
</tr>
<tr>
<td>jge label</td>
<td>~(OF ^ SF)</td>
</tr>
<tr>
<td>jle label</td>
<td>(OF ^ SF)</td>
</tr>
<tr>
<td>jg label</td>
<td>~((OF ^ SF)</td>
</tr>
</tbody>
</table>

Note:

- If you can understand why `jl` jumps iff `OF^SF`
- … then the others follow
Conditional Jumps: Signed

Why does jl jump iff OF^SF? Informal explanation:

(1) largeposnum – smallposnum (not less than)
   • Certainly correct result
   • => OF=0, SF=0, OF^SF==0 => don't jump

(2) smallposnum – largeposnum (less than)
   • Certainly correct result
   • => OF=0, SF=1, OF^SF==1 => jump

(3) largenegnum – smallnegnum (less than)
   • Certainly correct result
   • => OF=0, SF=1 => (OF^SF)==1 => jump

(4) smallnegnum – largenegnum (not less than)
   • Certainly correct result
   • => OF=0, SF=0 => (OF^SF)==0 => don't jump
Conditional Jumps: Signed

(5) posnum – negnum (not less than)
   • Suppose correct result
   • => OF=0, SF=0 => (OF^SF)==0 => don't jump

(6) posnum – negnum (not less than)
   • Suppose incorrect result
   • => OF=1, SF=1 => (OF^SF)==0 => don't jump

(7) negnum – posnum (less than)
   • Suppose correct result
   • => OF=0, SF=1 => (OF^SF)==1 => jump

(8) negnum – posnum (less than)
   • Suppose incorrect result
   • => OF=1, SF=0 => (OF^SF)==1 => jump