Assembly Language: Part 2

Flattened C code
- Control flow with signed integers
- Control flow with unsigned integers
- Assembly Language: Defining global data
- 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

See Bryant & O’Hallaron book for faster patterns

Agenda

Flattened C code
- Control flow with signed integers
- Control flow with unsigned integers
- Assembly Language: Defining global data
- Arrays
- Structures
### If Example

**C**

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

**Flattened C**

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

**Assem Lang**

```assembly
.section .bss
  i: .skip 4
...
.section .text
  ...
  cmpl $0, i
  jge  endif1negl
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) goto else1;
smaller = i;
goto endif1;
else1:
smaller = j;
endif1:
```

**Flattened C**

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

**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:
```

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

### 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;while (n > 1){
  fact *= n;
  n--;
}
```

**Assem Lang**

```assembly
.section .bss

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

**Note:**
- `jle` instruction (conditional jump)
- `imul` instruction
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;
...
for (i = 0; i < exp; i++)
    power *= base;
```

for Example

Assem Lang

```c
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

- Sets condition-code bits in the EFLAGS register
- Beware: operands are in counterintuitive order
- Beware: many other instructions set condition-code bits
  - Conditional jump should immediately follow cmp

Control Flow with Signed Integers

Unconditional jump

```c
jmp label Jump to label
```

Conditional jumps after comparing signed integers

```c
je  label  Jump to label if equal
jne label Jump to label if not equal
jl label Jump to label if less
jle label Jump to label if less or equal
jg label Jump to label if greater
jge label Jump to label if greater or equal
```

Agenda

Flattened C

Control flow with signed integers

Assembly Language: Defining global data

Arrays

Structures

Signed vs. Unsigned Integers

In C
- Integers are signed or unsigned
  - Compiler generates assem lang 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

Signed vs. Unsigned Integers

In C
- Integers are signed or unsigned
  - Compiler generates assem lang 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: `cmp` + `{je, jne, jl, jle, jg, jge, jea, jae}`

Example

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

C 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

```assembler
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`)

Assem Lang

```assembler
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:
```

Note:

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

Comparing unsigned integers (Same as comparing signed integers)

Conditional jumps after comparing unsigned integers

- Examine CC bits in EFLAGS register

```assembly
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
```
Agenda

- Flattened C code
- Control flow with signed integers
- Control flow with unsigned integers
- Assembly Language: Defining global data
- Arrays
- Structures

RAM (Random Access Memory)

Control Unit

ALU

Registers

RAM

TEXT

RODATA

DATA

BSS

HEAP

STACK

RAM

CPU

Data bus

Defining Data: DATA Section 1

- static char c = 'a';
- static short s = 12;
- static int i = 345;
- static long l = 6789;

Note:
- .section instruction (to announce DATA section)
- .globl instruction (marks a spot in RAM)
- .byte instruction (1 byte)
- .word instruction (2 bytes)
- .long instruction (4 bytes)
- .quad instruction (8 bytes)

Note:
- Best to avoid "word" (2 byte) data

Defining Data: DATA Section 2

- char c = 'a';
- short s = 12;
- int i = 345;
- long l = 6789;

Note:
- Can place label on same line as next instruction
- .globl instruction

Defining Data: BSS Section

- static char c;
- static short s;
- static int i;
- static long l;

Note:
- .section instruction (to announce BSS section)
- .skip instruction

Defining Data: RODATA Section

- "hello\n";

Note:
- .section instruction (to announce RODATA section)
- .string instruction
Agenda

- Flattened C
- Control flow with signed integers
- Control flow with unsigned integers
- Assembly Language: Defining global data

Arrays

- Structures

Arrays: Indirect Addressing

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

i = 3;
...
n = a[i]
...
```

Assem Lang

```
.text
...
.movl $3, i
...
.movl $3, i
...
.movl $3, i
...
```

Registers

```
RAX 3
R10 a
```

Memory

```
0   1000
1   1004
2   1008
3   1012
```

Arrays: Indirect Addressing

```
.text
...
.movl $3, i
...
.movl $3, i
...
```

Registers

```
RAX 12
R10 a
```

Memory

```
0   1000
1   1004
2   1008
3   1012
```

Arrays: Indirect Addressing

```
.text
...
.movl $3, i
...
.movl $3, i
...
```

Registers

```
RAX 1012
R10 a
```

Memory

```
0   1000
1   1004
2   1008
3   1012
```

One step at a time...
### Arrays: Indirect Addressing

#### Assem Lang
```
.section '.bss'
a: .skip 400
i: .skip 4
n: .skip 4
```

#### Registers
- RAX: 1012
- R10: 123

#### Memory
```
   a
0  1000
1  1004
2  1008
3  1012
```

### Arrays: Indirect Addressing

#### Assem Lang
```
.section '.text'
```

#### Assem Lang
```
movl $3, i
```

#### Assem Lang
```
movslq i, %rax
salq $2, %rax
addq $a, %rax
movl (%rax), %r10d
movl %r10d, n
```

### Arrays: Base+Disp Addressing

#### Assem Lang
```
.section '.bss'
a: .skip 400
i: .skip 4
n: .skip 4
```

#### Assem Lang
```
.section '.text'
```

#### Assem Lang
```
movl $3, i
```

#### Assem Lang
```
movl i, %eax
salq $2, %eax
addq $a, %eax
movl (%eax), %r10d
movl %r10d, n
```

### Arrays: Base+Disp Addressing

#### Assem Lang
```
.section '.bss'
a: .skip 400
i: .skip 4
n: .skip 4
```

#### Assem Lang
```
.section '.text'
```

#### Assem Lang
```
movl $3, i
```

#### Assem Lang
```
movl i, %eax
salq $2, %eax
addq $a, %eax
movl (%eax), %r10d
movl %r10d, n
```

### Arrays: Base+Disp Addressing

#### Assem Lang
```
.section '.bss'
a: .skip 400
i: .skip 4
n: .skip 4
```

#### Assem Lang
```
.section '.text'
```

#### Assem Lang
```
movl $3, i
```

#### Assem Lang
```
movl i, %eax
salq $2, %eax
addq $a, %eax
movl (%eax), %r10d
movl %r10d, n
```

### Note:
- **Indirect** addressing

---

### Arrays: Base+Disp Addressing

#### Assem Lang
```
.section '.bss'
a: .skip 400
i: .skip 4
n: .skip 4
```

#### Assem Lang
```
.section '.text'
```

#### Assem Lang
```
movl $3, i
```

#### Assem Lang
```
movl i, %eax
salq $2, %eax
addq $a, %eax
movl (%eax), %r10d
movl %r10d, n
```

### Arrays: Base+Disp Addressing

#### Assem Lang
```
.section '.bss'
a: .skip 400
i: .skip 4
n: .skip 4
```

#### Assem Lang
```
.section '.text'
```

#### Assem Lang
```
movl $3, i
```

#### Assem Lang
```
movl i, %eax
salq $2, %eax
addq $a, %eax
movl (%eax), %r10d
movl %r10d, n
```

### C Assem Lang
```
int a[100];
i = 3;
n = a[i]
```

### 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
sall $2, %eax
movl a(%eax), %r10d
movl %r10d, n
```

#### Registers
- RAX: 12
- R10: 12

#### Memory
```
0 1000
1 1004
2 1008
3 1012
```

<table>
<thead>
<tr>
<th>a</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>3</td>
<td>1000</td>
<td>1004</td>
<td>1008</td>
</tr>
<tr>
<td>n</td>
<td>1400</td>
<td>1404</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Note:* Base+displacement 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), %r10d
movl %r10d, n
```

#### Registers
- RAX: 3
- R10: 123

#### Memory
```
0 1000
1 1004
2 1008
3 1012
```

<table>
<thead>
<tr>
<th>a</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>3</td>
<td>1000</td>
<td>1004</td>
<td>1008</td>
</tr>
<tr>
<td>n</td>
<td>1400</td>
<td>1404</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Note:* Scaled indexed addressing

### C Assem Lang
```
int a[100];
int i;
int n;
i = 3;
n = a[i]
```

One step at a time...

### 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), %r10d
movl %r10d, n
```

#### Registers
- RAX: 3

#### Memory
```
0 1000
1 1004
2 1008
3 1012
```

<table>
<thead>
<tr>
<th>a</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>3</td>
<td>1000</td>
<td>1004</td>
<td>1008</td>
</tr>
<tr>
<td>n</td>
<td>1400</td>
<td>1404</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Assem Lang

Registers

Memory

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R10</td>
<td>123</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- .section " .bss"
- a: .skip 400
- i: .skip 4
- n: .skip 4

- .section " .text"
- movl $3, i
- movl i, %eax
- movl a(%eax, 4), %r10d
- movl %r10d, n

Generalization: Memory Operands

Full form of memory operands:

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

- displacement is an integer or a label (default = 0)
- base is a 4-byte or 8-byte register
- index is a 4-byte or 8-byte register
- 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...

Operand Examples

Immediate operands

- $5 \Rightarrow \text{use the number 5 (i.e. the number that is available immediately within the instruction)}$
- $i \Rightarrow \text{use the address denoted by } i \text{ (i.e. the address that is available immediately within the instruction)}$

Register operands

- %rax \Rightarrow \text{read from (or write to) register RAX}$

Memory operands: direct addressing

- 5 \Rightarrow \text{load from (or store to) memory at address 5 (silly; seg fault)}$
- i \Rightarrow \text{load from (or store to) memory at the address denoted by } i$

Memory operands: indirect addressing

- (%rax) \Rightarrow \text{consider the contents of RAX to be an address; load from (or store to) that address}$

Memory operands: base+displacement addressing

- 5(%rax) \Rightarrow \text{compute the sum (5) + (contents of RAX); consider the sum to be an address; load from (or store to) that address}$
- i(%rax) \Rightarrow \text{compute the sum (address denoted by } i) + (\text{contents of RAX}; consider the sum to be an address; load from (or store to) that address}$

Memory operands: indexed addressing

- 5(%rax, %r10) \Rightarrow \text{compute the sum (5) + (contents of RAX) + (contents of R10); consider the sum to be an address; load from (or store to) that address}$
- i(%rax, %r10) \Rightarrow \text{compute the sum (address denoted by } i) + (\text{contents of RAX) + (contents of R10); consider the sum to be an address; load from (or store to) that address}$

Memory operands: scaled indexed addressing

- 5(%rax, %r10, 4) \Rightarrow \text{compute the sum (5) + (contents of RAX) + ((contents of R10) \times 4); consider the sum to be an address; load from (or store to) that address}$
- i(%rax, %r10, 4) \Rightarrow \text{compute the sum (address denoted by } i) + (\text{contents of RAX) + ((contents of R10) \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**
- `movq 5(%rax), %r10`
  - Compute the sum (5) + (contents of RAX); consider the sum to be an address; load 8 bytes from that address into R10
- `leaq 5(%rax), %r10`
  - Compute the sum (5) + (contents of RAX); move that sum to R10

**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? `leaq (%rax, %rax, 4), %rax`

---

### Agenda

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

**Assembly Language: Defining global data**

**Arrays**

**Structures**

---

### Structures: Indirect Addressing

**C**

```c
struct S
{
  int i;
  int j;
}

struct S myStruct;

myStruct.i = 18;
myStruct.j = 19;
```

**Assem Lang**

```assembly
section '.text'

movq $myStruct, %rax
addq $4, %rax
movl $19, (%rax)
```

**Note:** Indirect addressing

---

### Structures: Base+Disp Addressing

**C**

```c
struct S
{
  char c;
  int i;
}

struct S myStruct;

myStruct.c = 'A';
myStruct.i = 18;
```

**Assem Lang**

```assembly
section '.text'

movl $18, 4(%rax)
```

---

### Structures: Padding

**C**

```c
struct S
{
  char c;
  int i;
}

struct S myStruct;

myStruct.c = 'A';
myStruct.i = 18;
```

**Assem Lang**

```assembly
section '.text'

movl $18, 4(%rax)
```

**Beware:** Compiler sometimes inserts padding after fields

---

### Structures: Padding

**x86-64/Linux rules**

<table>
<thead>
<tr>
<th>Data type</th>
<th>Within a struct, 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>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>16</td>
</tr>
<tr>
<td>any pointer</td>
<td>8</td>
</tr>
</tbody>
</table>

- Compiler may add padding after last field if struct is within an array
Intermediate aspects of x86-64 assembly language…

Flattened C code
Control transfer with signed integers
Control transfer with unsigned integers
Arrays
  • Full form of instruction operands
Structures
  • Padding

**Appendix**

Setting and using CC bits in EFLAGS register

**Question**
• How does \texttt{cmp(q, l, w, b)} 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:

\begin{verbatim}
addq src, dest
\end{verbatim}

• Compute sum \((\text{dest} + \text{src})\)
• Assign sum to \text{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
    \((\text{src} >= 0 \&\& \text{dest} >= 0 \&\& \text{sum} < 0) \lor\)
    \((\text{src} < 0 \&\& \text{dest} < 0 \&\& \text{sum} >= 0)\)

Example:

\begin{verbatim}
subq src, dest
\end{verbatim}

• Compute sum \((\text{dest} -(\text{src}))\)
• Assign sum to \text{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 \text{dest}<\text{src}
• OF: set to 1 iff signed overflow
  • Set to 1 iff
    \((\text{dest} > 0 \&\& \text{src} < 0 \&\& \text{sum} < 0) \lor\)
    \((\text{dest} < 0 \&\& \text{src} > 0 \&\& \text{sum} >= 0)\)

Example:

\begin{verbatim}
cmpq src, dest
\end{verbatim}

• Same as \texttt{subq}
• But does not affect \text{dest}
Using Condition Code Bits

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: 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) large negnum – small negnum (less than)
• Certainly correct result
⇒ OF=0, SF=1 ⇒ (OF^SF)==1 ⇒ jump
(4) small negnum – large negnum (not less than)
• Certainly correct result
⇒ OF=0, SF=0 ⇒ (OF^SF)==0 ⇒ don’t jump
(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)==1 ⇒ 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
Byte Order

x86-64 is a little endian architecture
- Least significant byte of multi-byte entity is stored at lowest memory address
- "Little end goes first"

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

<table>
<thead>
<tr>
<th>Byte 0: ff</th>
<th>Byte 1: 77</th>
<th>Byte 2: 33</th>
<th>Byte 3: 00</th>
</tr>
</thead>
</table>

Output on a little-endian machine

| Byte 0: 00 | Byte 1: ff | Byte 2: 33 | Byte 3: 77 |

Output on a big-endian machine

Note:
- Flawed code; uses "b" instructions to manipulate a four-byte memory area
- Flawed code; uses "l" instructions to manipulate a one-byte memory area