Process Layout and Function Calls

CS 161 – Spring 2017
Process Layout in Memory

▶ **Stack**
  ▶ grows towards *decreasing* addresses.
  ▶ is initialized at *run-time*.

▶ **Heap**
  ▶ grow towards *increasing* addresses.
  ▶ is initialized at *run-time*.

▶ **BSS section**
  ▶ size fixed at *compile-time*.
  ▶ is initialized at *run-time*.
  ▶ was grouped into **Data** in CS61C.

▶ **Data section**
  ▶ is initialized at *compile-time*.

▶ **Text section**
  ▶ holds the program instructions (read-only).
Process Layout in Memory

- **Stack**
  - grows towards *decreasing* addresses.
  - is initialized at *run-time*.

- **Heap**
  - grow towards *increasing* addresses.
  - is initialized at *run-time*.

- **BSS section**
  - size fixed at *compile-time*.
  - is initialized at *run-time*.
  - was grouped into Data in CS61C.

- **Data section**
  - is initialized at *compile-time*.

- **Text section**
  - holds the program instructions (read-only).
## Key Differences Between AT&T Syntax and Intel Syntax

<table>
<thead>
<tr>
<th>Parameter</th>
<th>AT&amp;T</th>
<th>Intel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter Order</td>
<td>src before dst</td>
<td>dst before src</td>
</tr>
<tr>
<td></td>
<td>movl $4, %eax</td>
<td>mov eax, 5</td>
</tr>
<tr>
<td>Parameter Size</td>
<td>Mnemonics suffixed with a letter indicating size of operands: q for qword, l for long (dword), w for word, and b for byte</td>
<td>Derived from name of register that is used (e.g. rax, eax, ax, al imply q, l, w, b, respectively)</td>
</tr>
<tr>
<td></td>
<td>addl $4, %esp</td>
<td>add esp, 4</td>
</tr>
<tr>
<td>Sigils</td>
<td>Immediate values prefixed with a $, registers prefixed with a %</td>
<td>Assembler automatically detects type of symbols; i.e., whether they are registers, constants or something else</td>
</tr>
</tbody>
</table>

void foo(int a, int b, int c)
{
    int bar[2];
    char qux[3];
    bar[0] = 'A';
    qux[0] = 0x42;
}

int main(void)
{
    int i = 1;
    foo(1, 2, 3);
    return 0;
}
int main(void) {
    int i = 1;
    foo(1, 2, 3);
    return 0;
}

main:
    pushl %ebp
    movl %esp,%ebp
    subl $4,%esp
    movl $1,-4(%ebp)
    pushl $3
    pushl $2
    pushl $1
    call foo
    addl $12,%esp
    xorl %eax,%eax
    leave
    ret
Function Calls in Assembler

int main(void)
{
    int i = 1;
    foo(1, 2, 3);
    return 0;
}

main:
    pushl %ebp
    movl %esp,%ebp
    subl $4,%esp
    movl $1,-4(%ebp)
    pushl $3
    pushl $2
    pushl $1
    call foo
    addl $12,%esp
    xorl %eax,%eax
    leave
    ret
Function Calls in Assembler

```c
int main(void)
{
    int i = 1;
    foo(1, 2, 3);
    return 0;
}
```

```assembly
main:
    pushl %ebp
    movl %esp,%ebp
    subl $4,%esp
    movl $1,-4(%ebp)
    pushl $3
    pushl $2
    pushl $1
    call foo
    addl $12,%esp
    xorl %eax,%eax
    leave
    ret
```
Function Calls in Assembler

int main(void)
{
    int i = 1;
    foo(1, 2, 3);
    return 0;
}

main:
pushl %ebp
movl %esp,%ebp
subl $4,%esp
movl $1,-4(%ebp)
pushl $3
pushl $2
pushl $1
call foo
addl $12,%esp
xorl %eax,%eax
leave
ret
int main(void)
{
    int i = 1;
    foo(1, 2, 3);
    return 0;
}

main:
pushl %ebp
movl %esp,%ebp
subl $4,%esp
movl $1,-4(%ebp)
pushl $3
pushl $2
pushl $1
call foo
addl $12,%esp
xorl %eax,%eax
leave
ret
Function Calls in Assembler

```c
int main(void)
{
    int i = 1;
    foo(1, 2, 3);
    return 0;
}

main:
pushl %ebp
movl %esp,%ebp
subl $4,%esp
movl $1,-4(%ebp)
pushl $3
pushl $2
pushl $1
call foo
addl $12,%esp
xorl %eax,%eax
leave
ret
```
Function Calls in Assembler

```c
int main(void) {
    int i = 1;
    foo(1, 2, 3);
    return 0;
}
```

```assembly
main:
pushl %ebp
movl %esp,%ebp
subl $4,%esp
movl $1,-4(%ebp)
pushl $3
pushl $2
pushl $1
call foo
addl $12,%esp
xorl %eax,%eax
leave
ret
```
void foo(int a, int b, int c)
{
    int bar[2];
    char qux[3];
    bar[0] = 'A';
    qux[0] = 0x42;
}

foo:
    pushl %ebp
    movl %esp,%ebp
    subl $12,%esp
    movl $65,-8(%ebp)
    movb $66,-12(%ebp)
    leave
    ret
void foo(int a, int b, int c)
{
    int bar[2];
    char qux[3];
    bar[0] = 'A';
    qux[0] = 0x42;
}

foo:
    pushl %ebp
    movl %esp,%ebp
    subl $12,%esp
    movl $65,-8(%ebp)
    movb $66,-12(%ebp)
    leave
    ret
void foo(int a, int b, int c)
{
    int bar[2];
    char qux[3];
    bar[0] = ’A’;
    qux[0] = 0x42;
}

foo:
    pushl %ebp
    movl %esp,%ebp
    subl $12,%esp
    movl $65,-8(%ebp)
    movb $66,-12(%ebp)
    leave
    ret
void foo(int a, int b, int c)
{
    int bar[2];
    char qux[3];
    bar[0] = 'A';
    qux[0] = 0x42;
}

foo:
    pushl %ebp
    movl %esp,%ebp
    subl $12,%esp
    movl $65,-8(%ebp)
    movb $66,-12(%ebp)
    leave
    ret

Larger Memory Addresses

sfp
ofp
ofp (m)
esbp
esp
Larger Memory Addresses
void foo(int a, int b, int c)
{
    int bar[2];
    char qux[3];
    bar[0] = 'A';
    qux[0] = 0x42;
}

foo:
    pushl %ebp
    movl %esp,%ebp
    subl $12,%esp
    movl $65,-8(%ebp)
    movb $66,-12(%ebp)
    leave
    ret
Function Calls in Assembler

```c
void foo(int a, int b, int c)
{
    int bar[2];
    char qux[3];
    bar[0] = 'A';
    qux[0] = 0x42;
}

foo:
    pushl %ebp
    movl %esp,%ebp
    subl $12,%esp
    movl $65,-8(%ebp)
    movb $66,-12(%ebp)
    leave
    ret
```
void foo(int a, int b, int c)
{
    int bar[2];
    char qux[3];
    bar[0] = 'A';
    qux[0] = 0x42;
}

foo:
    pushl %ebp
    movl %esp,%ebp
    subl $12,%esp
    movl $65,-8(%ebp)
    movb $66,-12(%ebp)
    leave
    ret

<table>
<thead>
<tr>
<th>rip</th>
<th>sfp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>esp + ebp</td>
<td></td>
</tr>
</tbody>
</table>

Function Calls
Function Calls in Assembler

void foo(int a, int b, int c) {
    int bar[2];
    char qux[3];
    bar[0] = 'A';
    qux[0] = 0x42;
}

foo:
    pushl %ebp
    movl %esp,%ebp
    subl $12,%esp
    movl $65,-8(%ebp)
    movb $66,-12(%ebp)
    leave
    ret
Function Calls in Assembler

```c
void foo(int a, int b, int c)
{
    int bar[2];
    char qux[3];
    bar[0] = 'A';
    qux[0] = 0x42;
}

foo:
    pushl %ebp
    movl %esp,%ebp
    subl $12,%esp
    movl $65,-8(%ebp)
    movb $66,-12(%ebp)
    leave
    ret
```

Larger Memory Addresses

```
00 00 00 41
42
```

Function Calls 5 / 8
int main(void)
{
    int i = 1;
    foo(1, 2, 3);
    return 0;
}

main:
pushl %ebp
movl %esp,%ebp
subl $4,%esp
movl $1,-4(%ebp)
pushl $3
pushl $2
pushl $1
call foo
addl $12,%esp
xorl %eax,%eax
leave
ret
int main(void)
{
    int i = 1;
    foo(1, 2, 3);
    return 0;
}

main:
    pushl %ebp
    movl %esp,%ebp
    subl $4,%esp
    movl $1,-4(%ebp)
    pushl $3
    pushl $2
    pushl $1
    call foo
    addl $12,%esp
    xorl %eax,%eax
    leave
    ret
int main(void)
{
    int i = 1;
    foo(1, 2, 3);
    return 0;
}

main:
pushl %ebp
movl %esp,%ebp
subl $4,%esp
movl $1,-4(%ebp)
pushl $3
pushl $2
pushl $1
call foo
addl $12,%esp
xorl %eax,%eax
leave
ret
int main(void)
{
    int i = 1;
    foo(1, 2, 3);
    return 0;
}

main:
pushl %ebp
movl %esp,%ebp
subl $4,%esp
movl $1,-4(%ebp)
pushl $3
pushl $2
pushl $1
call foo
addl $12,%esp
xorl %eax,%eax
leave
ret
Function Calls in Assembler

```c
int main(void)
{
    int i = 1;
    foo(1, 2, 3);
    return 0;
}
```

main:

```assembly
pushl %ebp
movl %esp,%ebp
subl $4,%esp
movl $1,-4(%ebp)
pushl $3
pushl $2
pushl $1
call foo
addl $12,%esp
xorl %eax,%eax
leave
ret
```
```c
int main(void)
{
    int i = 1;
    foo(1, 2, 3);
    return 0;
}

main:
    pushl %ebp
    movl %esp,%ebp
    subl $4,%esp
    movl $1,-4(%ebp)
    pushl $3
    pushl $2
    pushl $1
    call foo
    addl $12,%esp
    xorl %eax,%eax
    leave
    ret
```
MIPS → IA-32 [Reference]

- **RISC vs CISC**
  - IA-32 has many more instructions
  - IA-32 instructions are variable length
  - IA-32 instructions can have implicit arguments and side effects

- **Limited Number of Registers**
  - MIPS has 18 general purpose registers ($s0-$s7, $t0-$t9)
  - IA-32 has 6 (%eax, %edx, %ecx, %ebx, %esi, %edi)
    - This means lots of stack operations!

- **Operand Directions**
  - MIPS: mov dst src
  - IA-32: mov src dst

- **Memory operations**
  - Very common to see push/pop/mov in IA-32
    - We’ll see more of this later

- **The list goes on!**
## MIPS → IA-32 [Reference]

### Registers

<table>
<thead>
<tr>
<th>Use</th>
<th>MIPS</th>
<th>IA32</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Counter</td>
<td>PC</td>
<td>%eip</td>
<td>Can not be referenced directly</td>
</tr>
<tr>
<td>Stack Pointer</td>
<td>$sp</td>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>Frame Pointer</td>
<td>$fp</td>
<td>%ebp</td>
<td></td>
</tr>
<tr>
<td>Return Address</td>
<td>$ra</td>
<td>-</td>
<td>RA kept on stack in IA-32</td>
</tr>
<tr>
<td>Return Value (32 bit)</td>
<td>$v0</td>
<td>%eax</td>
<td>%eax not used solely for RV</td>
</tr>
<tr>
<td>Argument Registers</td>
<td>$a0-$a3</td>
<td>-</td>
<td>Passed on stack in IA-32</td>
</tr>
<tr>
<td>Zero</td>
<td>$0</td>
<td>-</td>
<td>Use immediate value on IA-32</td>
</tr>
</tbody>
</table>

### Register Terminology

- **SFP** saved frame pointer: saved %ebp on the stack
- **OFP** old frame pointer: old %ebp from the previous stack frame
- **RIP** return instruction pointer: return address on the stack
# IA-32 [Reference]

## IA32 Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl Src, Dest</td>
<td>Dest = Src</td>
</tr>
<tr>
<td>addl Src, Dest</td>
<td>Dest = Dest + Src</td>
</tr>
<tr>
<td>subl Src, Dest</td>
<td>Dest = Dest - Src</td>
</tr>
<tr>
<td>imull Src, Dest</td>
<td>Dest = Dest * Src</td>
</tr>
<tr>
<td>sall Src, Dest</td>
<td>Dest = Dest &lt;&lt; Src</td>
</tr>
<tr>
<td>sarl Src, Dest</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>shr1 Src, Dest</td>
<td>Dest = Dest &gt;&gt; Src</td>
</tr>
<tr>
<td>xorl Src, Dest</td>
<td>Dest = Dest ^ Src</td>
</tr>
<tr>
<td>andl Src, Dest</td>
<td>Dest = Dest &amp; Src</td>
</tr>
<tr>
<td>orl Src, Dest</td>
<td>Dest = Dest</td>
</tr>
<tr>
<td>incl Dest</td>
<td>Dest = Dest + 1</td>
</tr>
<tr>
<td>decl Dest</td>
<td>Dest = Dest - 1</td>
</tr>
<tr>
<td>negl Dest</td>
<td>Dest = - Dest</td>
</tr>
<tr>
<td>notl Dest</td>
<td>Dest = ~ Dest</td>
</tr>
<tr>
<td>leal Src, Dest</td>
<td>Dest = address of Src</td>
</tr>
<tr>
<td>cmpl Src2, Src1</td>
<td>Sets CCs Src1 - Src2</td>
</tr>
<tr>
<td>testl Src2, Src1</td>
<td>Sets CCs Src1 &amp; Src2</td>
</tr>
<tr>
<td>jmp label</td>
<td>jump</td>
</tr>
<tr>
<td>je label</td>
<td>jump equal</td>
</tr>
<tr>
<td>jne label</td>
<td>jump not equal</td>
</tr>
<tr>
<td>js label</td>
<td>jump negative</td>
</tr>
<tr>
<td>jns label</td>
<td>jump non-negative</td>
</tr>
<tr>
<td>jg label</td>
<td>jump greater (signed)</td>
</tr>
<tr>
<td>jge label</td>
<td>jump greater or equal (signed)</td>
</tr>
<tr>
<td>jl label</td>
<td>jump less (signed)</td>
</tr>
<tr>
<td>jle label</td>
<td>jump less or equal (signed)</td>
</tr>
<tr>
<td>ja label</td>
<td>jump above (unsigned)</td>
</tr>
<tr>
<td>jb label</td>
<td>jump below (unsigned)</td>
</tr>
</tbody>
</table>

## Addressing Modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>$val$, Val</td>
</tr>
<tr>
<td>Normal</td>
<td>(R), Mem[Reg[R]]</td>
</tr>
<tr>
<td>Displacement</td>
<td>D(R), Mem[Reg[R]+D]</td>
</tr>
<tr>
<td>Indexed</td>
<td>D(Rb,Ri,S), Mem[Reg[Rb]+S*Reg[Ri]+D]</td>
</tr>
</tbody>
</table>

- Register R specifies memory address
- Constant displacement D specifies offset

## Condition Codes

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CF</td>
<td>Carry Flag</td>
</tr>
<tr>
<td>ZF</td>
<td>Zero Flag</td>
</tr>
<tr>
<td>SF</td>
<td>Sign Flag</td>
</tr>
<tr>
<td>OF</td>
<td>Overflow Flag</td>
</tr>
</tbody>
</table>