Machine-level representation of programs: control, comparisons, branching, loops

Yipeng Huang

Rutgers University

March 29, 2022
Table of contents
Announcements
Comparisons and program control flow
Modifying control flow via conditional branch statements
Modifying data flow via conditional move statements
Loop statements
  Compiling for loops to while loops
  Compiling while loops to do-while loops
  Compiling do-while loops to goto statements
  Compiling goto statements to assembly conditional jump instructions
Switch statements
Procedures and function calls: Transferring control
  Special state
  Stack instructions: push and pop
  Procedure call and return: call and ret
  Example in GDB
Procedures and function calls: Transferring data
Announcements

PA4 bomb lab
▶ PA4 bomb lab out and live. Due Tuesday, April 5.

Short quiz this week
Short quiz on assembly basics and control spanning Tuesday 3/29 to Friday 4/1.

Class session plan
▶ Today, Tuesday, 3/29: Loops. Function calls in assembly. (Book chapter 3.7)
▶ Thursday, 3/31: Arrays and data structures in assembly. (Book chapter 3.8)
Table of contents
  Announcements
  Comparisons and program control flow
  Modifying control flow via conditional branch statements
  Modifying data flow via conditional move statements
  Loop statements
    Compiling for loops to while loops
    Compiling while loops to do-while loops
    Compiling do-while loops to goto statements
    Compiling goto statements to assembly conditional jump instructions
  Switch statements
  Procedures and function calls: Transferring control
    Special state
    Stack instructions: push and pop
    Procedure call and return: call and ret
    Example in GDB
  Procedures and function calls: Transferring data
What is control flow?

Control flow is:

▶ Change in the sequential execution of instructions.
▶ Change in the steady incrementation of the program counter / instruction pointer (%rip register).

Control primitives in assembly build up to enable C and Java control statements:

▶ if-else statements
▶ do-while loops
▶ while loops
▶ for loops
▶ switch statements
Condition codes

Assembly/Machine Code View

Programmer-Visible State

- PC: Program counter
  - Address of next instruction
  - Called “RIP” (x86-64)
- Register file
  - Heavily used program data
- Condition codes
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching

Memory

- Byte addressable array
- Code and user data
- Stack to support procedures

Carnegie Mellon

Table of contents

Announcements

Comparisons and program control flow
Modifying control flow via conditional branch statements
Modifying data flow via conditional move statements

Loop statements
  Compiling for loops to while loops
  Compiling while loops to do-while loops
  Compiling do-while loops to goto statements
  Compiling goto statements to assembly conditional jump instructions

Switch statements

Procedures and function calls: Transferring control
  Special state
  Stack instructions: push and pop
  Procedure call and return: call and ret
  Example in GDB

Procedures and function calls: Transferring data
Jump instructions

Jumping

- **jX Instructions**
  - Jump to different part of code depending on condition codes

<table>
<thead>
<tr>
<th>jX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>js</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg</td>
<td>~(SF^OF) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle</td>
<td>(SF^OF)</td>
<td>ZF</td>
</tr>
<tr>
<td>ja</td>
<td>~CF &amp; ~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
Conditional branch statements

---

```c
unsigned long absdiff_ternary ( unsigned long x, unsigned long y ){
    return x<y ? y-x : x-y;
}
```

```c
unsigned long absdiff_if_else ( unsigned long x, unsigned long y ){
    if (x<y) return y-x;
    else return x-y;
}
```

```c
unsigned long absdiff_goto ( unsigned long x, unsigned long y ){
    if (!(x<y)) goto Else;
    return y-x;
    Else:
    return x-y;
}
```

---

All C functions above translate (-fno-if-conversion) to assembly at right.

```assembly
absdiff_if_else:
    cmpq %rsi, %rdi
    jnb .ELSE
    movq %rsi, %rax
    subq %rdi, %rax
    ret

.absdiff_goto:
    cmpq %rsi, %rdi
    jnb .ELSE
    movq %rsi, %rax
    subq %rdi, %rax
    ret

.ELSE:
    movq %rdi, %rax
    subq %rsi, %rax
    ret
```

---

Explanation

- `cmpq %rsi, %rdi`: Calculates \( %\text{rdi} - %\text{rsi} \) \((x - y)\), sets condition codes.
- `jnb .ELSE`: Sets program counter / instruction pointer in \%rip (.ELSE) if CF flag not set indicating no unsigned overflow.
Table of contents

Announcements
Comparisons and program control flow
Modifying control flow via conditional branch statements
Modifying data flow via conditional move statements
Loop statements
  Compiling for loops to while loops
  Compiling while loops to do-while loops
  Compiling do-while loops to goto statements
  Compiling goto statements to assembly conditional jump instructions
Switch statements
Procedures and function calls: Transferring control
  Special state
  Stack instructions: push and pop
  Procedure call and return: call and ret
  Example in GDB
Procedures and function calls: Transferring data
Conditional move statements

```c
unsigned long absdiff_ternary ( unsigned long x, unsigned long y ){
    return x<y ? y-x : x-y;
}
```

```c
unsigned long absdiff_if_else ( unsigned long x, unsigned long y ){
    if (x<y) return y-x;
    else return x-y;
}
```

```c
unsigned long absdiff_goto ( unsigned long x, unsigned long y ){
    if (!(x<y)) goto Else;
    return y-x;
    Else:
    return x-y;
}
```

All C functions above translate (-fif-conversion or -O1) to assembly at right.

```assembly
absdiff_ternary:  
movq %rsi, %rdx // y
subq %rdi, %rdx // y-x
movq %rdi, %rax // x
subq %rsi, %rax // x-y
cmpq %rsi, %rdi
cmovb %rdx, %rax
ret
```

Explanation

- `cmpq %rsi, %rdi`: Calculates %rdi - %rsi (x - y), sets condition codes.
- `jnb .ELSE`: Sets program counter / instruction pointer in %rip (.ELSE) if CF flag not set indicating no unsigned overflow.
Modifying control flow vs. data flow in deep CPU pipelines

Figure: Pipelined CPU stages. Image credit wikimedia
Table of contents

Announcements

Comparisons and program control flow
Modifying control flow via conditional branch statements
Modifying data flow via conditional move statements

Loop statements
  Compiling for loops to while loops
  Compiling while loops to do-while loops
  Compiling do-while loops to goto statements
  Compiling goto statements to assembly conditional jump instructions

Switch statements

Procedures and function calls: Transferring control
  Special state
  Stack instructions: push and pop
  Procedure call and return: call and ret
  Example in GDB

Procedures and function calls: Transferring data
Compiling for loops to while loops

C loop statements such as for loops, while loops, and do-while loops do not exist in assembly. They are instead constructed from conditional jump statements.

```c
unsigned long count_bits_for (unsigned long number) {
    unsigned long tally = 0;
    for (int shift=0; shift<8*sizeof(unsigned long); shift++) {
        tally += 0b1 & number>>shift;
    }
    return tally;
}
```

```c
unsigned long count_bits_while (unsigned long number) {
    unsigned long tally = 0;
    int shift=0;
    while (shift<8*sizeof(unsigned long)) {
        tally += 0b1 & number>>shift;
        shift++; // update
    }
    return tally;
}
```
Compiling while loops to do-while loops

```c
unsigned long count_bits_while (unsigned long number) {
    unsigned long tally = 0;
    int shift = 0; // init
    while (shift < 8*sizeof(unsigned long) // ← test
        // body
        tally += 0b1 & number>>shift;
        shift++; // update
    ) {
        // body
        tally += 0b1 & number>>shift;
        shift++; // update
    }
    return tally;
}
```

```c
unsigned long count_bits_do_while (unsigned long number) {
    unsigned long tally = 0;
    int shift = 0; // init
    do { // body
        tally += 0b1 & number>>shift;
        shift++; // update
    } while (shift < 8*sizeof(unsigned long)); // test
    return tally;
}
```

If initial iteration is guaranteed to run, then do one fewer test.
Compiling do-while loops to goto statements

Loops get compiled into goto statements which are readily translated to assembly.
Compiling goto statements to assembly conditional jump instructions

```c
unsigned long count_bits_goto (  
    unsigned long number  
) {  
    unsigned long tally = 0;  
    int shift=0; // init  
    LOOP:  
        // body  
        tally += 0b1 & number>>shift;  
        shift++; // update  
        if (shift<8*sizeof(unsigned long)) {  
            goto LOOP;  
        }  
    return tally;  
}
```

All C loop statements so far translate to assembly at right.

```assembly
xorl %ecx, %ecx # int shift=0; // init  
xorl %eax, %eax # unsigned long tally = 0;  
LOOP:  
movq %rdi, %rdx # number  
shrq %cl, %rdx # number>>shift  
incl %ecx # shift++; // update  
andl $1, %edx. # 0b1 & number>>shift  
addq %rdx, %rax # tally += 0b1 & number>>shift  
cmpl $64, %ecx # shift<8*sizeof(unsigned long)  
jne .LOOP # goto LOOP;  
ret # return tally;
```
Table of contents
Announcements
Comparisons and program control flow
Modifying control flow via conditional branch statements
Modifying data flow via conditional move statements
Loop statements
  Compiling for loops to while loops
  Compiling while loops to do-while loops
  Compiling do-while loops to goto statements
  Compiling goto statements to assembly conditional jump instructions
Switch statements
Procedures and function calls: Transferring control
  Special state
  Stack instructions: push and pop
  Procedure call and return: call and ret
  Example in GDB
Procedures and function calls: Transferring data
Table of contents
Announcements
Comparisons and program control flow
Modifying control flow via conditional branch statements
Modifying data flow via conditional move statements
Loop statements
  Compiling for loops to while loops
  Compiling while loops to do-while loops
  Compiling do-while loops to goto statements
  Compiling goto statements to assembly conditional jump instructions
Switch statements
Procedures and function calls: Transferring control
  Special state
  Stack instructions: push and pop
  Procedure call and return: call and ret
  Example in GDB
Procedures and function calls: Transferring data
Procedures and function calls

```
P(...) {
    ...
    y = Q(x);
    print(y);
    ...
}
```

```
int Q(int i) {
    int t = 3*i;
    int v[10];
    ...
    return v[t];
}
```

To create the abstraction of functions, need to:

- Transfer control to function and back
- Transfer data to function (parameters)
- Transfer data from function (return type)

Figure: Steps of a C function call. Image credit CS:APP
CPU and memory state in support of procedures and functions

Assembly/Machine Code View

- **CPU**: Program counter (PC), Register file, Condition codes
- **Memory**: Code, Data, Stack

**Programmer-Visible State**
- **PC**: Program counter
  - Address of next instruction
  - Called “RIP” (x86-64)
- **Register file**: Heavily used program data
- **Condition codes**: Store status information about most recent arithmetic or logical operation
  - Used for conditional branching

**Memory**
- Byte addressable array
- Code and user data
- Stack to support procedures

Relevant state in CPU:
- %rip register / instruction pointer / program counter
- %rsp register / stack pointer

Relevant state in Memory:
- Stack
Stack instructions: \textit{push} and \textit{pop}

Initially

<table>
<thead>
<tr>
<th></th>
<th>%rax</th>
<th>0x123</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdx</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>%rsp</td>
<td></td>
<td>0x108</td>
</tr>
</tbody>
</table>

pushq \%rax

<table>
<thead>
<tr>
<th></th>
<th>%rax</th>
<th>0x123</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdx</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>%rsp</td>
<td></td>
<td>0x100</td>
</tr>
</tbody>
</table>

popq \%rdx

<table>
<thead>
<tr>
<th></th>
<th>%rax</th>
<th>0x123</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdx</td>
<td></td>
<td>0x123</td>
</tr>
<tr>
<td>%rsp</td>
<td></td>
<td>0x108</td>
</tr>
</tbody>
</table>

Increasing address

Stack “bottom”

Stack “top”

Stack “bottom”

Stack “top”

Stack “bottom”

Stack “top”

Figure: x86-64 offers dedicated instructions to work with stack in memory. In addition to moving data, the updating of \%rsp is implied. Image credit: CS:APP.
Procedure call and return: \texttt{call} and \texttt{ret}

\begin{figure}
\centering
\begin{tabular}{|c|c|}
\hline
%rip & 0x400563 \\
%rsp & 0x7fffffffe840 \\
\hline
\end{tabular}
\hspace{2cm}
\begin{tabular}{|c|c|}
\hline
%rip & 0x400540 \\
%rsp & 0x7fffffffe838 \\
\hline
\end{tabular}
\hspace{2cm}
\begin{tabular}{|c|c|}
\hline
%rip & 0x400568 \\
%rsp & 0x7fffffffe840 \\
\hline
\end{tabular}
\caption{Effect of \texttt{call} 0x400540 instruction and subsequent return. \texttt{call} and \texttt{ret} instructions update the instruction pointer, the stack pointer, and the stack to create the procedure / function call abstraction. Image credit: CS:APP.}
\end{figure}
Example in GDB

```c
#include <stdio.h>

int return_neg_one() {
    return -1;
}

int main() {
    int num = return_neg_one();
    printf("%d", num);
    return 0;
}
```

Compile, and then run it in GDB:
```
gdb return
```

In GDB, see evolution of `%rip`, `%rsp`, and stack:
- (gdb) layout split
- (gdb) break return_neg_one
- (gdb) print /a $rip
- (gdb) print /a $rsp
- (gdb) x /a $rsp

Step past return instruction, and inspect again:
- (gdb) steppi
Table of contents

Announcements

Comparisons and program control flow

Modifying control flow via conditional branch statements

Modifying data flow via conditional move statements

Loop statements

  Compiling for loops to while loops
  Compiling while loops to do-while loops
  Compiling do-while loops to goto statements
  Compiling goto statements to assembly conditional jump instructions

Switch statements

Procedures and function calls: Transferring control

  Special state
  Stack instructions: push and pop
  Procedure call and return: call and ret
  Example in GDB

Procedures and function calls: Transferring data
Procedures and function calls: Transferring data

For purposes of this class, the Bomb Lab, and the CS:APP textbook, we study the x86-64 Linux Application Binary Interface (ABI). Would be different on ARM or in Windows. So, don’t memorize this, but it is helpful for PA4 Lab.

Passing parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Register / stack</th>
<th>Subset registers</th>
<th>Mnemonic¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>%rdi</td>
<td>%edi, %di</td>
<td>Diane’s</td>
</tr>
<tr>
<td>2nd</td>
<td>%rsi</td>
<td>%esi, %si</td>
<td>silk</td>
</tr>
<tr>
<td>3rd</td>
<td>%rdx</td>
<td>%edx, %dx, %dl</td>
<td>dress</td>
</tr>
<tr>
<td>4th</td>
<td>%rcx</td>
<td>%ecx, %cx, %cl</td>
<td>cost</td>
</tr>
<tr>
<td>5th</td>
<td>%r8</td>
<td>%r8d</td>
<td>$8</td>
</tr>
<tr>
<td>6th</td>
<td>%r9</td>
<td>%r9d</td>
<td>9</td>
</tr>
<tr>
<td>7th and beyond</td>
<td>Stack</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

¹http://csappbook.blogspot.com/2015/08/dianes-silk-dress-costs-89.html
int sscanf (const char *str, // 1st arg, %rdi
        const char *format, // 2nd arg, %rsi
...
Procedures and function calls: Transferring data

Passing function return data

Function return data is passed via:
- the 64-bit `%rax` register
- the 32-bit subset `%eax` register