Machine-Level Representation of Programs: Loops, Procedures

Yipeng Huang

Rutgers University

November 6, 2025

Table of contents

Announcements

Class session plan

- ~~Thursday, 10/30: addressing modes (Book chapter 3.4), arithmetic (Book chapter 3.5). Bomblab phase_1.~~
- ~~Tuesday, 11/4: Control flow (conditionals, if, for, while, do loops, switch statements) in assembly. (Book chapter 3.6). Bomblab phase_2, phase_3.~~
- Thursday, 11/6: Function calls in assembly. (Book chapter 3.7). Bomblab phase_4.
- Tuesday, 11/11: Arrays and data structures in assembly. (Book chapter 3.8). Bomblab phase_5, phase_6.

Table of contents

What is control flow?

Control flow is:

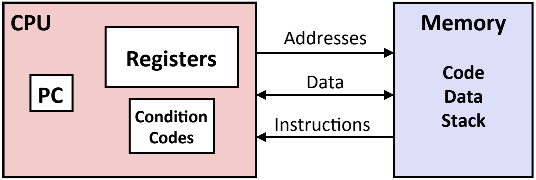
- Change in the sequential execution of instructions.
- Change in the steady incrementation of the program counter / instruction pointer (%rip register).

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

Condition codes

Automatically set by most arithmetic instructions.

Applicable types	Condition code	Name	Use
Signed and unsigned	`ZF`	Zero flag	The most recent operation yielded zero.
Unsigned types	`CF`	Carry flag	The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations
Signed types	`SF`	Sign flag	The most recent operation yielded a negative value.
Signed types	`OF`	Overflow flag	The most recent operation yielded a two's complement positive or negative overflow.

Table: Condition codes important for control flow

Comparison instructions

`cmpq source1, source2`

Performs source2 − source1, and sets the condition codes without setting any destination register.

Test for equality

```c
short equal_sl (
    long x,
    long y
) {
    return x==y;
}
```

C code function above translates to the assembly on the right.

```
equal_sl:
    xorl %eax, %eax
    cmpq %rsi, %rdi
    sete %al
    ret
```

Explanation

- `xorl %eax, %eax`: Zeros the 32-bit register %eax.
- `cmpq %rsi, %rdi`: Calculates %rdi − %rsi ($x - y$), sets condition codes without updating any destination register.
- `sete %al`: Sets the 8-bit %al subset of %eax if op yielded zero.

Test if unsigned x is below unsigned y

```c
short below_ul (
    unsigned long x,
    unsigned long y
) {
    return x<y;
}
```

```c
short nae_ul (
    unsigned long x,
    unsigned long y
) {
    return !(x>=y);
}
```

Both C code functions above translate to the assembly on the right.

```
below_ul:
nae_ul:
    xorl %eax, %eax
    cmpq %rsi, %rdi
    setb %al
    ret
```

Explanation

- `xorl %eax, %eax`: Zeros %eax.
- `cmpq %rsi, %rdi`: Calculates %rdi − %rsi ($x - y$), sets condition codes without updating any destination register.
- `setb %al`: Sets %al if CF flag set indicating unsigned overflow.

Side review: De Morgan's laws

- $\neg A \wedge \neg B \iff \neg(A \vee B)$
- $(\sim A)\&(\sim B) \iff \sim (A|B)$

Set instructions

`cmp source1, source2` performs source2 – source1, sets condition codes.

Applicable types	Set instruction	Logical condition	Intutive condition
Signed and unsigned	`sete` / `setz`	`ZF`	Equal / zero
Signed and unsigned	`setne` / `setnz`	$\sim$ `ZF`	Not equal / not zero
Unsigned	`setb` / `setnae`	`CF`	Below
Unsigned	`setbe` / `setna`	`CF`|`ZF`	Below or equal
Unsigned	`seta` / `setnbe`	$\sim$ `CF` & $\sim$ `ZF`	Above
Unsigned	`setnb` / `setae`	$\sim$ `CF`	Above or equal
Signed	`sets`	`SF`	Negative
Signed	`setns`	$\sim$ `SF`	Nonegative
Signed	`setl` / `setnge`	`SF ^ OF`	Less than
Signed	`setle` / `setng`	(`SF ^ OF`)|`ZF`	Less than or equal
Signed	`setg` / `setnle`	$\sim$ (`SF ^ OF`) & $\sim$ `ZF`	Greater than
Signed	`setge` / `setnl`	$\sim$ (`SF ^ OF`)	Greater than or equal

Table: Set instructions

Table of contents

Jump instructions

Jumping

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

jX	Condition	Description
`jmp`	`1`	Unconditional
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signed)
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF&~ZF`	Above (unsigned)
`jb`	`CF`	Below (unsigned)

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.

```
absdiff_if_else:
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 %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.

Table of contents

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

```
absdiff_ternary:
absdiff_if_else:
absdiff_goto:
    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

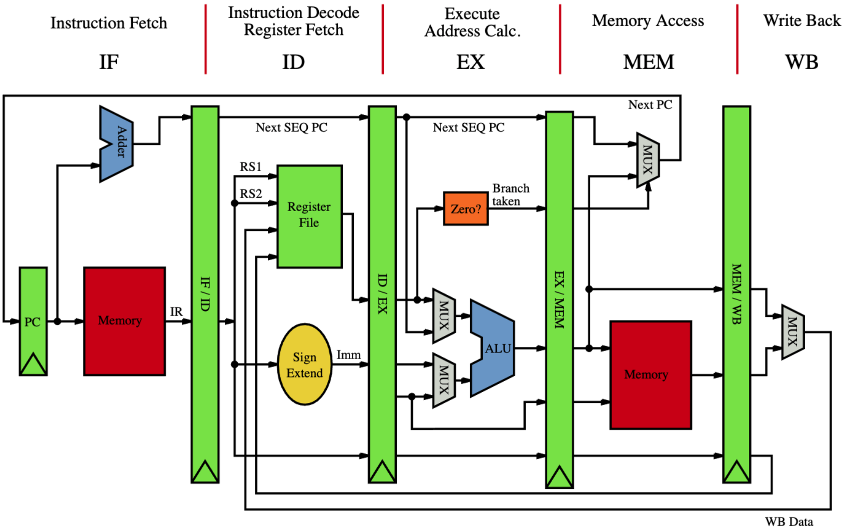


Figure: Pipelined CPU stages. Image credit wikimedia

Table of contents

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

```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
  }
  return tally;
}
```

Compiling while loops to do-while loops

```
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
  }
  return tally;
}
```

```
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

```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;
}
```

```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)) { // test
    goto LOOP;
  }
  return tally;
}
```

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)) { // test
    goto LOOP;
  }
  return tally;
}
```

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

```
count_bits_for:
count_bits_while:
count_bits_do_while:
count_bits_goto:
  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>>shi
  cmpl $64, %ecx  # shift<8*sizeof(unsigned lo
  jne .LOOP       # goto LOOP;
  ret             # return tally;
```

Table of contents

Procedures and function calls

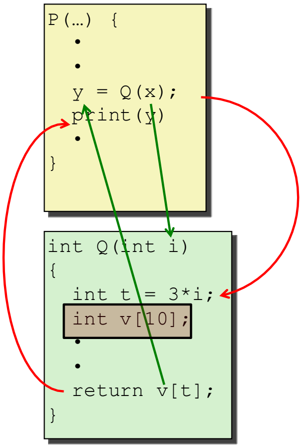
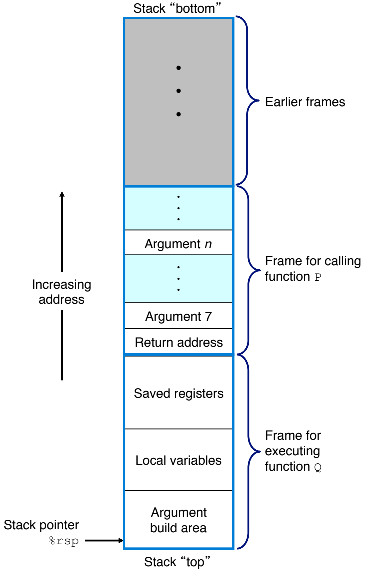


Figure: Steps of a C function call. Image credit CS:APP

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)

Memory stack frames



Structure of stack for currently executing function Q()

- P() calls Q(). P() is the caller function. Q() is the callee function.

Stack instructions: `push src` and `pop dest`

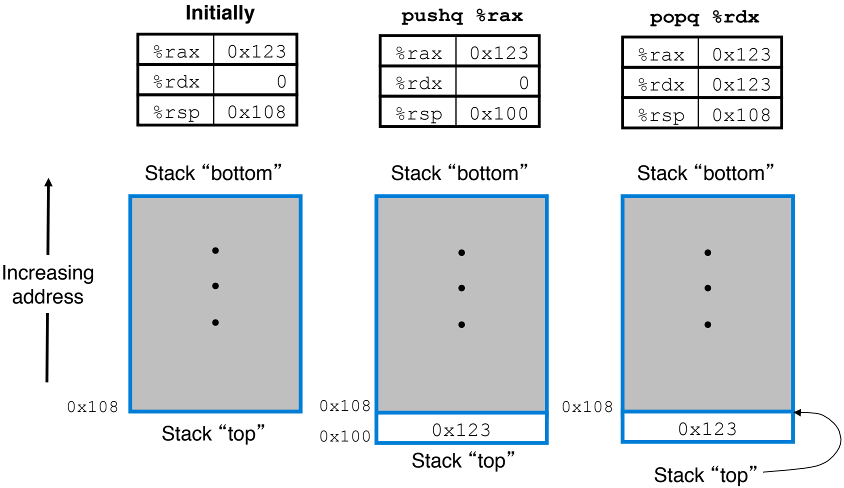
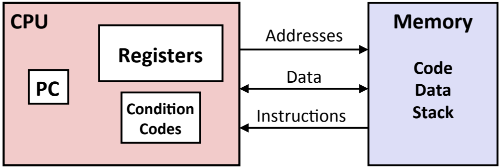


Figure: x86-64 offers dedicated instructions to work with stack in memory. In addition to moving data, the updating of %rsp is implied. Image credit: CS:APP.

Table of contents

CPU and memory state in support of procedures and functions

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

Relevant state in CPU:

- %rip register / instruction pointer / program counter
- %rsp register / stack pointer

Relevant state in Memory:

- Stack

Procedure call and return: `call` and `ret`

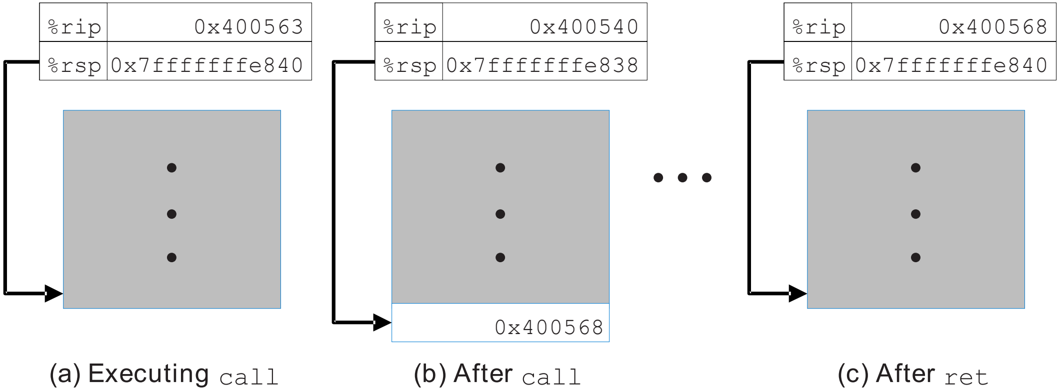


Figure: Effect of `call 0x400540` instruction and subsequent return. `call` and `ret` instructions update the instruction pointer, the stack pointer, and the stack to create the procedure / function call abstraction. Image credit: CS:APP.

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;
}
```

```
return_neg_one:
    movl $-1, %eax
    ret
main:
    subq $8, %rsp
    movl $0, %eax
    call return_neg_one
    movl %eax, %edx
    ...
```

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) info stack`
- `(gdb) print /a $rip`
- `(gdb) print /a $rsp`
- `(gdb) x /a $rsp`

Step past return instruction, and inspect again:

- `(gdb) stepi`
- `(gdb) info stack`

Table of contents

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

Parameter	Register / stack	Subset registers	Mnemonic[1]
1st	%rdi	%edi, %di	Diane's
2nd	%rsi	%esi, %si	silk
3rd	%rdx	%edx, %dx, %dl	dress
4th	%rcx	%ecx, %cx, %cl	cost
5th	%r8	%r8d	$8
6th	%r9	%r9d	9
7th and beyond	Stack		

[1] http://csappbook.blogspot.com/2015/08/dianes-silk-dress-costs-89.html

PA4 Defusing a Binary Bomb: `sscanf();`

```
int sscanf (
    const char *str, // 1st arg, %rdi
    const char *format, // 2nd arg, %rsi
    ...
)
```

Procedures and function calls: Transferring data

Passing function return data

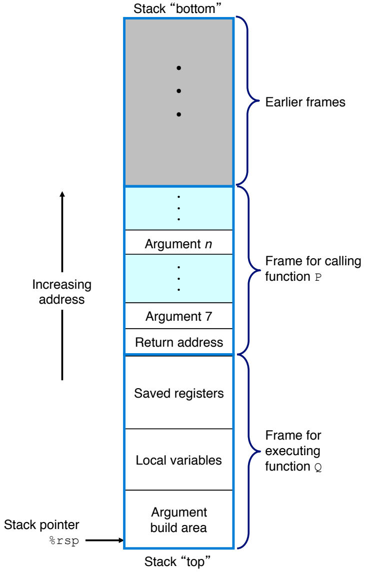
Function return data is passed via:

- the 64-bit %rax register
- the 32-bit subset %eax register

Example from textbook slides on assembly procedures

Slides 33 through 38.

Data transferred via memory



Structure of stack for currently executing function Q()

- P() calls Q(). P() is the caller function. Q() is the callee function.

Example from textbook slides on assembly procedures

Slides 40 through 44.

Table of contents

3_recursion.c: Putting it all together to support recursion

Discussion points

- Use info stack, info args in GDB to see recursion depth
- Difference between compiling with and without -g for debugging information.
- Memory costs of recursion.
- Compilers can recognize tail recursive calls to reduce memory use. Enabled with -foptimize-sibling-calls, -O2, -O3, and -Os.