Machine-Level Representation of Programs: Addressing modes, arithmetic, control flow

Yipeng Huang

Rutgers University

November 4, 2025

Table of contents

Announcements

Class session plan

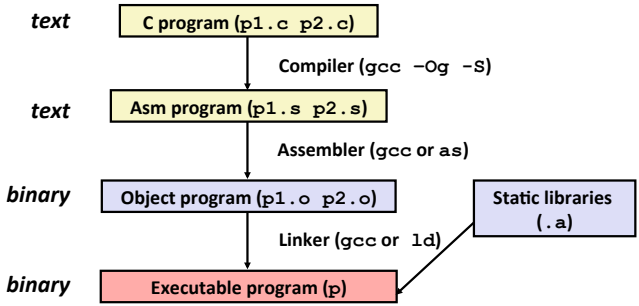
- ~~Today, 10/30: addressing modes (Book chapter 3.4), arithmetic (Book chapter 3.5). Bomblab phase_1.~~
- Today, 11/4: Control flow (conditionals, if, for, while, do loops, switch statements) in assembly. (Book chapter 3.6). Bomblab phase_2, phase_3.
- Thursday, 11/6: Function calls in assembly. (Book 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

Unraveling the compilation chain

Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc –Og p1.c p2.c -o p`
  - Use basic optimizations (`–Og`) [New to recent versions of GCC]
  - Put resulting binary in file `p`



- gcc -0g -S swap.c
- objdump -d swap

Let's go to CS:APP textbook lecture slides (05-machine-basics.pdf) slide 28

Data movement instructions

Does unsigned / signed matter?

1. `void swap_uc ( unsigned char*a, unsigned char*b );`
2. `void swap_sc ( signed char*a, signed char*b );`

Swapping different data sizes

1. `void swap_c ( char*a, char*b );`
2. `void swap_s ( short*a, short*b );`
3. `void swap_i ( int*a, int*b );`
4. `void swap_l ( long*a, long*b );`

Table of contents

Data size and x86 / x86-64 registers

Assembly syntax

Instruction Source, Dest

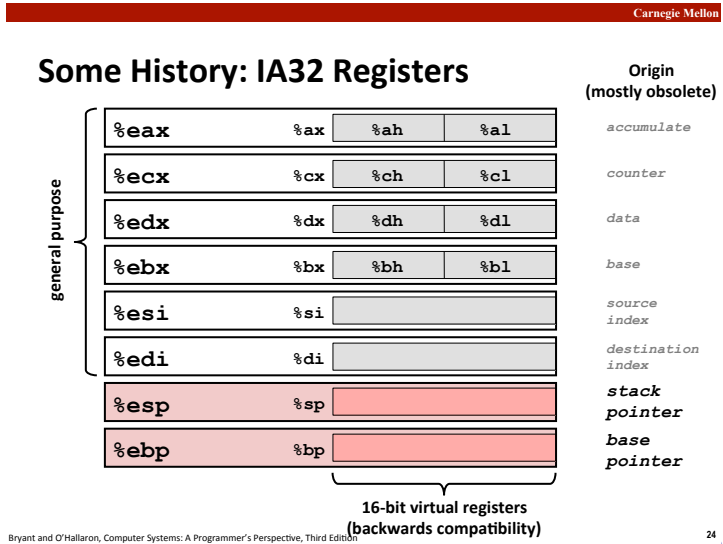
```
swap_l:
 movq (%rsi), %rax
 movq (%rdi), %rdx
 movq %rdx, (%rsi)
 movq %rax, (%rdi)
 ret
```

swap	data type	mov operation	registers
swap_uc	unsigned char	movb (move byte)	%al, %dl
swap_sc	signed char	movb (move byte)	%al, %dl
swap_c	char	movb (move byte)	%al, %dl
swap_s	short	movw (move word)	%ax, %dx
swap_i	int	movl	%eax, %edx
swap_l	long	movq	%rax, %rdx

Data size and IA32, x86, and x86-64 registers

data type	registers
char	%al, %dl
short	%ax, %dx
int	%eax, %edx
long	%rax, %rdx

Note the backward compatibility.



Data size and IA32, x86, and x86-64 registers

data type	registers
char	%al, %dl
short	%ax, %dx
int	%eax, %edx
long	%rax, %rdx

Note the backward compatibility.

x86-64 Integer Registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)

Data size and IA32, x86, and x86-64 registers

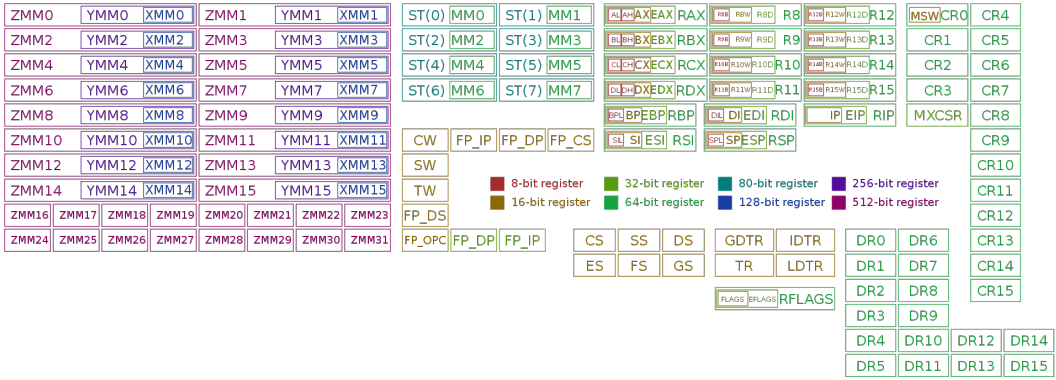


Figure: x86-64 with SIMD extensions registers. Image credit: `https://commons.wikimedia.org/wiki/File:Table_of_x86_Registers_svg.svg`

Table of contents

Sign extension due to unsigned and signed data types

Converting to a data type with more bits

```
1 unsigned short uc_to_us (
2     unsigned char input
3 ) {
4     return input;
5 }
```

```
1 signed short sc_to_ss (
2     signed char input
3 ) {
4     return input;
5 }
```

$$\begin{aligned} 255 &= 1111\_1111_2 \\ &= 0000\_0000\_1111\_1111_2 \\ &= 255 \end{aligned}$$

$$\begin{aligned} 127 &= 0111\_1111_2 \\ &= 0000\_0000\_0111\_1111_2 \\ &= 127 \end{aligned}$$

$$\begin{aligned} -128 &= 1000\_0000_2 \\ &= 1111\_1111\_1000\_0000_2 \\ &= -128 \end{aligned}$$

Sign extension due to unsigned and signed data types

Converting to a data type with more bits

```
1 unsigned short uc_to_us (
2     unsigned char input
3 ) {
4     return input;
5 }
```

```
1 signed short sc_to_ss (
2     signed char input
3 ) {
4     return input;
5 }
```

function signature	assembly code
unsigned short uc_to_us (unsigned char input);	movzbl %dil, %eax
signed short uc_to_ss (unsigned char input);	movzbl %dil, %eax
unsigned short sc_to_us (signed char input);	movsbw %dil, %ax
signed short sc_to_ss (signed char input);	movsbw %dil, %ax

- movz: zero extension in the MSBs
- movs: signed extension in the MSBs

Table of contents

Left shift operation

```c
unsigned long sl_ul (
    unsigned long in0,
    unsigned long in1
) {
    return in0<<in1;
}
```

```c
signed long sl_sl (
    signed long in0,
    signed long in1
) {
    return in0<<in1;
}
```

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

```
sl_ul:
sl_sl:
    movq %rdi, %rax
    movb %sil, %cl
    salq %cl, %rax
    ret
```

Explanation

- `movq`: in0 → %rdi → %rax
- `movb`: in1 → %sil → %cl
- `salq src,dest`: (dest << src) → dest
- Why only use movb for in1?

Right shift operation

Right shift of unsigned types yields logical (zero-filled) right shift

```
1 unsigned long sr_ul (
2     unsigned long in0,
3     unsigned long in1
4 ) {
5     return in0>>in1;
6 }
```

```
sr_ul:
    movq %rdi, %rax
    movb %sil, %cl
    shrq %cl, %rax
    ret
```

Right shift of signed types yields arithmetic (sign-extended) right shift

```
1 signed long sr_sl (
2     signed long in0,
3     signed long in1
4 ) {
5     return in0>>in1;
6 }
```

```
sr_sl:
    movq %rdi, %rax
    movb %sil, %cl
    sarq %cl, %rax
    ret
```

Bitwise operations

Assembly instruction	Instruction effect
`notq dest`	$\sim$ dest $\rightarrow$ dest
`andq src,dest`	src&dest $\rightarrow$ dest
`orq src,dest`	src|dest $\rightarrow$ dest
`xorq src,dest`	src $\wedge$ dest $\rightarrow$ dest

Integer arithmetic operations

Assembly instruction	Instruction effect
`incq dest`	$\text{dest} + 1 \rightarrow \text{dest}$
`decq dest`	$\text{dest} - 1 \rightarrow \text{dest}$
`negq dest`	$-\text{dest} \rightarrow \text{dest}$
`addq src,dest`	$\text{src} + \text{dest} \rightarrow \text{dest}$
`subq src,dest`	$\text{src} - \text{dest} \rightarrow \text{dest}$
`imulq src,dest`	$\text{src} \times \text{dest} \rightarrow \text{dest}$

Table of contents

Immediate, register, and memory

Immediate

Constant integer values. Example: 2_addressing_modes.c immediate()

Register

One of the registers of appropriate size for data type. Example: 1_swap.c

Memory

Access to memory at calculated

movq Operand Combinations

	Source	Dest	Src,Dest	C Analog
movq	Imm	Reg	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	Reg	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`
	Mem	Reg	`movq (%rax),%rdx`	`temp = *p;`

Cannot do memory-memory transfer with a single instruction

Addressing modes

Simple Memory Addressing Modes

- **Normal (R) Mem[Reg[R]]**
  - Register R specifies memory address
  - Aha! Pointer dereferencing in C

```
movq (%rcx),%rax
```

- **Displacement D(R) Mem[Reg[R]+D]**
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

```
movq 8(%rbp),%rdx
```


Normal

Simple pointers. Example: 2_addressing_modes.c immediate()

Displacement

Array access with constant index. Example: 2_addressing_modes.c displacement()

Addressing modes

Complete Memory Addressing Modes

- **Most General Form**

  D(Rb,Ri,S) **Mem[Reg[Rb]+S*Reg[Ri]+ D]**

  - D: Constant "displacement" 1, 2, or 4 bytes
  - Rb: Base register: Any of 16 integer registers
  - Ri: Index register: Any, except for `%rsp`
  - S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

- **Special Cases**

  (Rb,Ri) **Mem[Reg[Rb]+Reg[Ri]]**

  D(Rb,Ri) **Mem[Reg[Rb]+Reg[Ri]+D]**

  (Rb,Ri,S) **Mem[Reg[Rb]+S*Reg[Ri]]**

Indexed

Array access with variable index.
Example: 2_addressing_modes.c index()

Addressing modes

Address Computation Examples

`%rdx`	`0xf000`
`%rcx`	`0x0100`

Expression	Address Computation	Address
`0x8(%rdx)`	`0xf000 + 0x8`	`0xf008`
`(%rdx,%rcx)`	`0xf000 + 0x100`	`0xf100`
`(%rdx,%rcx,4)`	`0xf000 + 4*0x100`	`0xf400`
`0x80(,%rdx,2)`	`2*0xf000 + 0x80`	`0x1e080`

2_addressing_modes.c: Imm→Mem

C code

```
void immediate ( long * ptr ) {
    *ptr = 0xFFFFFFFFFFFFFFFF;
}
```

Assembly code

```
immediate:
    movq $-1, (%rdi)
    ret
```

- $ indicates the immediate value; corresponds to literals in C
- (%rdi) indicates memory location at address stored in %rdi register

2_addressing_modes.c: Imm→Mem (with displacement)

C code

```
void displacement_l ( long * ptr ) {
  ptr[1] = 0xFFFFFFFFFFFFFFFF;
}
```

Assembly code

```
displacement_l:
    movq $-1, 8(%rdi)
    ret
```

- 8(%rdi) indicates memory location at address stored in %rdi register + 8

2_addressing_modes.c: Imm→Mem (with displacement)

function signature	assembly code
void displacement_c (char * ptr);	movb $-1, 1(%rdi)
void displacement_s (short * ptr);	movw $-1, 2(%rdi)
void displacement_i (int * ptr);	movl $-1, 4(%rdi)
void displacement_l (long * ptr);	movq $-1, 8(%rdi)

2_addressing_modes.c: Imm→Mem (with index)

C code

```
void index_l ( long * ptr, long index ) {
  ptr[index] = 0xFFFFFFFFFFFFFFFF;
}
```

Assembly code

```
index_l:
    movq $-1, (%rdi,%rsi,8)
    ret
```

- (%rdi,%rsi,8) indicates memory location at address stored in %rdi register + 8 × value stored in %rsi register

2_addressing_modes.c: Imm→Mem (with index)

function signature	assembly code
void index_c (char * ptr, long index);	movb $-1, (%rdi,%rsi)
void index_s (short * ptr, long index);	movw $-1, (%rdi,%rsi,2)
void index_i (int * ptr, long index);	movl $-1, (%rdi,%rsi,4)
void index_l (long * ptr, long index);	movq $-1, (%rdi,%rsi,8)

2_addressing_modes.c: Imm→Mem (with displacement and index)

C code

```
void displacement_and_index ( long * ptr, long index ) {
  ptr[index+1] = 0xFFFFFFFFFFFFFFFF;
}
```

- 8(%rdi,%rsi,8) indicates memory location at address stored in %rdi register + 8 × value stored in %rsi register + 8

Assembly code

```
displacement_and_index:
    movq $-1, 8(%rdi,%rsi,8)
    ret
```

Table of contents

3_leaq.s: Borrowing memory address calculation to efficiently implement arithmetic

Address Computation Instruction

- **leaq Src, Dst**
  - Src is address mode expression
  - Set Dst to address denoted by expression

- **Uses**
  - Computing addresses without a memory reference
    - E.g., translation of `p = &x[i];`
  - Computing arithmetic expressions of the form x + k*y
    - k = 1, 2, 4, or 8

- **Example**

```
long m12(long x)
{
  return x*12;
}
```

Converted to ASM by compiler:

```
leaq (%rdi,%rdi,2), %rax # t <- x+x*2
salq $2, %rax            # return t<<2
```

Example: 3_leaq.c

Load effective address

```
1 long * leaq (
2     long * ptr, long index
3 ) {
4     return &ptr[index+1];
5 }
```

```
1 long mulAdd (
2     long base, long index
3 ) {
4     return base+index*8+8;
5 }
```

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

```
leaq:
mulAdd:
    leaq 8(%rdi,%rsi,8), %rax
    ret
```

Explanation

- `leaq src,dest` takes the effective address of the memory (index, displacement) expression of src and puts it in dest.
- `leaq` has shorter latency (takes fewer CPU cycles) than `imulq`, so GCC will use leaq whenever it can to calculate expressions like $y + ax + b$.

Table of contents

What is control flow?

Control flow is:

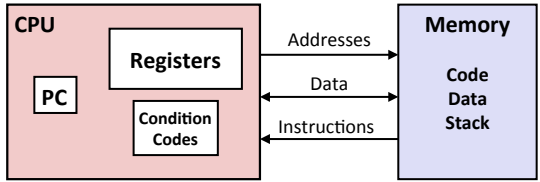
- Change in the sequential execution of instructions.
- Change in the steady incrementation of the program counter / instruction pointer (%rip register).

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

```
1 short equal_sl (
2     long x,
3     long y
4 ) {
5     return x==y;
6 }
```

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

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

```
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

```
1 unsigned long absdiff_ternary (
2     unsigned long x, unsigned long y ){
3         return x<y ? y-x : x-y;
4 }
```

```
1 unsigned long absdiff_if_else (
2     unsigned long x, unsigned long y ){
3         if (x<y) return y-x;
4         else return x-y;
5 }
```

```
1 unsigned long absdiff_goto (
2     unsigned long x, unsigned long y ){
3         if (!(x<y)) goto Else;
4         return y-x;
5     Else:
6         return x-y;
7 }
```

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

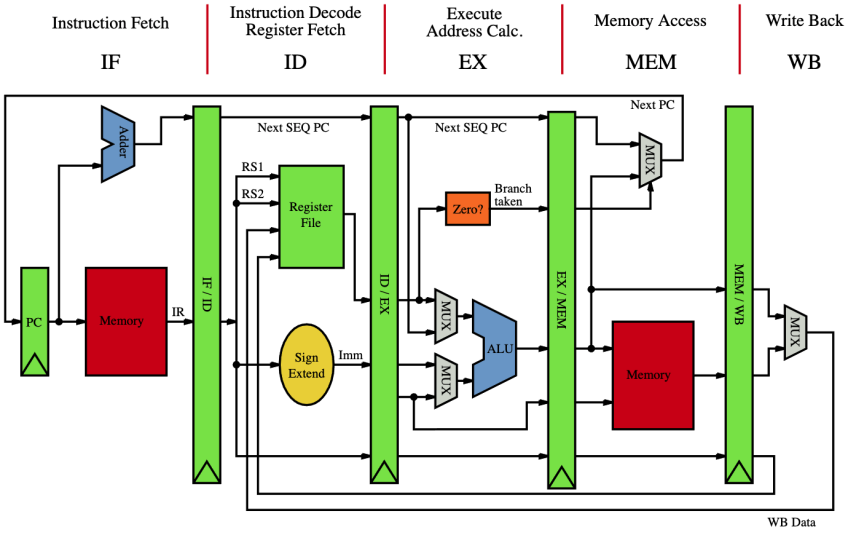


Figure: Pipelined CPU stages. Image credit wikimedia

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.

```
1  unsigned long count_bits_for (
2    unsigned long number
3  ) {
4    unsigned long tally = 0;
5    for (
6      int shift=0; // init
7      shift<8*sizeof(unsigned long); // ↩
           test
8      shift++ // update
9    ) {
10     // body
11     tally += 0b1 & number>>shift;
12   }
13   return tally;
14 }
```

```
1  unsigned long count_bits_while (
2    unsigned long number
3  ) {
4    unsigned long tally = 0;
5    int shift=0; // init
6    while (
7      shift<8*sizeof(unsigned long) // ↩
           test
8    ) {
9      // body
10     tally += 0b1 & number>>shift;
11     shift++; // update
12   }
13   return tally;
14 }
```

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

```
1  unsigned long count_bits_do_while (
2    unsigned long number
3  ) {
4    unsigned long tally = 0;
5    int shift=0; // init
6    do {
7      // body
8      tally += 0b1 & number>>shift;
9      shift++; // update
10   } while (shift<8*sizeof(unsigned long↩
         )); // test
11   return tally;
12 }
```

```
1  unsigned long count_bits_goto (
2    unsigned long number
3  ) {
4    unsigned long tally = 0;
5    int shift=0; // init
6  LOOP:
7    // body
8    tally += 0b1 & number>>shift;
9    shift++; // update
10   if (shift<8*sizeof(unsigned long)) { ↩
         // test
11     goto LOOP;
12   }
13   return tally;
14 }
```

Loops get compiled into goto statements which are readily translated to assembly.

Compiling goto statements to assembly conditional jump instructions

```
1  unsigned long count_bits_goto (
2    unsigned long number
3  ) {
4    unsigned long tally = 0;
5    int shift=0; // init
6  LOOP:
7    // body
8    tally += 0b1 & number>>shift;
9    shift++; // update
10   if (shift<8*sizeof(unsigned long)) { ↩
         // test
11     goto LOOP;
12   }
13   return tally;
14 }
```

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;
```