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

Yipeng Huang

Rutgers University

November 6, 2026

Announcements

Class session plan

- ~~Today, 11/3: addressing modes (Book chapter 3.4), arithmetic (Book chapter 3.5). Bomblab phase_1.~~
- Friday, 11/6: Control flow (conditionals, if, for, while, do loops, switch statements) in assembly. (Book chapter 3.6). Bomblab phase_2, phase_3.
- Tuesday, 11/10: Function calls in assembly. (Book chapter 3.7). Bomblab phase_4.
- Friday, 11/13: Arrays and data structures in assembly. (Book chapter 3.8). Bomblab phase_5, phase_6.

Quiz 6 and programming assignment 4

Quiz 6

1. Spanning Friday 11/6 - Friday 11/13.
2. 60 minutes.
3. Three tries.
4. Assembly data movement and control flow.

Programming assignment 4

1. Due Tuesday 11/17.
2. Binary bomb lab.

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

- `2_addressing_modes.s`: Understanding source dest operands and memory addressing modes
- `3_leaq.s`: Borrowing memory address calculation to efficiently implement arithmetic
- Comparisons and program control flow
  - What is control flow?
  - Condition codes
  - Comparison and set instructions
- Modifying control flow via conditional branch statements
  - Jump instructions
  - 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

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_1 ( long * ptr ) {
  ptr[1] = 0xFFFFFFFFFFFFFFFF;
}
```

Assembly code

```
displacement_1:
    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
```

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

- `2_addressing_modes.s`: Understanding source dest operands and memory addressing modes
- `3_leaq.s`: Borrowing memory address calculation to efficiently implement arithmetic
- Comparisons and program control flow
  - What is control flow?
  - Condition codes
  - Comparison and set instructions
- Modifying control flow via conditional branch statements
  - Jump instructions
  - 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

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

```c
long * leaq (
    long * ptr, long index
) {
    return &ptr[index+1];
}
```

```c
long mulAdd (
    long base, long index
) {
    return base+index*8+8;
}
```

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 $y + ax + b$.

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

- `2_addressing_modes.s`: Understanding source dest operands and memory addressing modes
- `3_leaq.s`: Borrowing memory address calculation to efficiently implement arithmetic
- Comparisons and program control flow
  - What is control flow?
  - Condition codes
  - Comparison and set instructions
- Modifying control flow via conditional branch statements
  - Jump instructions
  - 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

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

- `2_addressing_modes.s`: Understanding source dest operands and memory addressing modes
- `3_leaq.s`: Borrowing memory address calculation to efficiently implement arithmetic
- Comparisons and program control flow
  - What is control flow?
    - Condition codes
    - Comparison and set instructions
- Modifying control flow via conditional branch statements
  - Jump instructions
  - 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

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:

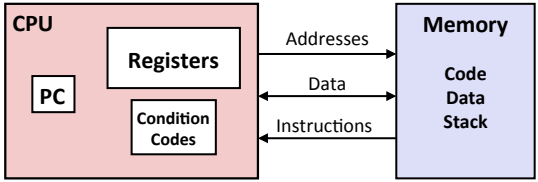
- if-else statements
- do-while loops
- while loops
- for loops
- switch statements

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

- `2_addressing_modes.s`: Understanding source dest operands and memory addressing modes
- `3_leaq.s`: Borrowing memory address calculation to efficiently implement arithmetic
- Comparisons and program control flow
  - What is control flow?
  - Condition codes
  - Comparison and set instructions
- Modifying control flow via conditional branch statements
  - Jump instructions
  - 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

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

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

- `2_addressing_modes.s`: Understanding source dest operands and memory addressing modes
- `3_leaq.s`: Borrowing memory address calculation to efficiently implement arithmetic
- Comparisons and program control flow
  - What is control flow?
  - Condition codes
  - Comparison and set instructions
- Modifying control flow via conditional branch statements
  - Jump instructions
  - 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

Comparison instructions

`cmpq source1, source2`

Performs $\text{source2} - \text{source1}$, 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

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

- `2_addressing_modes.s`: Understanding source dest operands and memory addressing modes
- `3_leaq.s`: Borrowing memory address calculation to efficiently implement arithmetic
- Comparisons and program control flow
  - What is control flow?
  - Condition codes
  - Comparison and set instructions
- Modifying control flow via conditional branch statements
  - Jump instructions
  - 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

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

- `2_addressing_modes.s`: Understanding source dest operands and memory addressing modes
- `3_leaq.s`: Borrowing memory address calculation to efficiently implement arithmetic
- Comparisons and program control flow
  - What is control flow?
  - Condition codes
  - Comparison and set instructions
- Modifying control flow via conditional branch statements
  - Jump instructions
  - 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

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)

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

- `2_addressing_modes.s`: Understanding source dest operands and memory addressing modes
- `3_leaq.s`: Borrowing memory address calculation to efficiently implement arithmetic
- Comparisons and program control flow
  - What is control flow?
  - Condition codes
  - Comparison and set instructions
- Modifying control flow via conditional branch statements
  - Jump instructions
  - 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

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.

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

- `2_addressing_modes.s`: Understanding source dest operands and memory addressing modes
- `3_leaq.s`: Borrowing memory address calculation to efficiently implement arithmetic
- Comparisons and program control flow
  - What is control flow?
  - Condition codes
  - Comparison and set instructions
- Modifying control flow via conditional branch statements
  - Jump instructions
  - 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

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

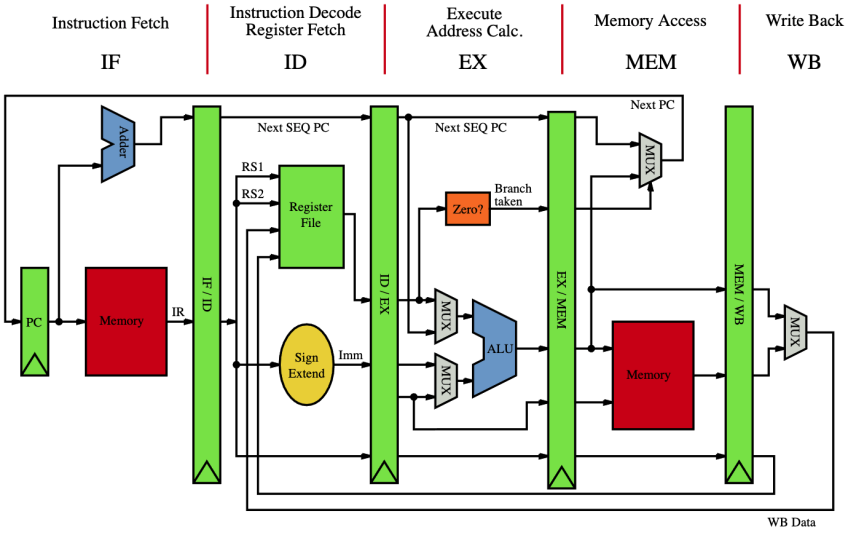


Figure: Pipelined CPU stages. Image credit wikimedia

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

- `2_addressing_modes.s`: Understanding source dest operands and memory addressing modes
- `3_leaq.s`: Borrowing memory address calculation to efficiently implement arithmetic
- Comparisons and program control flow
  - What is control flow?
  - Condition codes
  - Comparison and set instructions
- Modifying control flow via conditional branch statements
  - Jump instructions
  - 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

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

2_addressing_modes.s: Understanding source dest operands and memory addressing modes
3_leaq.s: Borrowing memory address calculation to efficiently implement arithmetic
Comparisons and program control flow
- What is control flow?
- Condition codes
- Comparison and set instructions

Modifying control flow via conditional branch statements
- Jump instructions
- 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

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 }
```

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

2_addressing_modes.s: Understanding source dest operands and memory addressing modes
3_leaq.s: Borrowing memory address calculation to efficiently implement arithmetic
Comparisons and program control flow
- What is control flow?
- Condition codes
- Comparison and set instructions

Modifying control flow via conditional branch statements
- Jump instructions
- 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

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.

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

- `2_addressing_modes.s`: Understanding source dest operands and memory addressing modes
- `3_leaq.s`: Borrowing memory address calculation to efficiently implement arithmetic
- Comparisons and program control flow
  - What is control flow?
  - Condition codes
  - Comparison and set instructions
- Modifying control flow via conditional branch statements
  - Jump instructions
  - 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

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.

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

- `2_addressing_modes.s`: Understanding source dest operands and memory addressing modes
- `3_leaq.s`: Borrowing memory address calculation to efficiently implement arithmetic
- Comparisons and program control flow
  - What is control flow?
  - Condition codes
  - Comparison and set instructions
- Modifying control flow via conditional branch statements
  - Jump instructions
  - 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

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;
```