Introduction to x86 ISA and Compilers
High-level Structure of Compiler

There may be different assembly languages for the same ISA. Example: AT&T (used by gcc) and Intel (used by icc) formats for x86 ISA.
x-86 instruction set

• x-86 ISA is very complex
  – CISC instruction set
  – Evolved over time:
    • 16 bit → 32 bit → 64 bit
    • MMX vector instructions
  – Assembly format: AT&T format and Intel format

• We will focus on x86-32 bit ISA since it is easier to understand

• Once you figure this out, x86-64 bit ISA is not hard
Useful website

• https://godbolt.org/
X86-32 Quick Overview

- **Registers:**
  - General purpose 32bit: eax, ebx, ecx, edx, esi, edi
  - Also 16-bit: ax, bx, etc., and 8-bit: al, ah, bl, bh, etc.
  - Special registers:
    - esp: stack pointer
    - ebp: frame base pointer
Note on register names

Registers are general-purpose: can be used for anything programmer wants

Historically, the registers were intended to be used as shown below, hence their odd names:

- AX/EAX/RAX: accumulator
- BX/EBX/RBX: base
- CX/ECX/RCX: counter
- DX/EDX/RDX: data/general
- SI/ESI/RSI: "source index" for string operations.
- DI/EDI/RDI: "destination index" for string operations.
- SP/ESP/RSP: stack pointer for top address of the stack.
- BP/EBP/RBP: stack base pointer for holding the address of the current stack frame.
- IP/EIP/RIP: instruction pointer. Holds the current instruction address.
Memory Layout

- Code
- Static area
- Heap
- Stack

Globals, Static data
Object fields, arrays
Locals, parameters

Byte addressable
x86 Quick Overview

• Instructions:
  – Arithmetic: add, sub, inc, mod, idiv, imul, etc.
  – Logic: and, or, not, xor
  – Comparison: cmp, test
  – Control flow: jmp, jcc, jecz
  – Function calls: call, ret
  – Data movement: mov (many variants)
  – Stack manipulations: push, pop
  – Other: lea
Instruction set

• x86 instruction set: two-address instruction set
  – Op a, b
    • a,b specify the two operands
    • result of operation is stored in b
      – warning: AT&T and Intel formats are different: see last slide
      – we will assume AT&T format in slides
    • a,b: registers or memory address
    • at most one operand can be in memory
    • memory addresses can be specified as offset from ebp (or other registers)
      – pushl 8(%ebp)
      – more generally, address can be specified as \texttt{disp(base,offset,\text{scale})}
  – Examples:
    • addl $3, %eax //add constant 3 to register eax
    • movl %eax, %ebx //move contents of register eax to register ebx
    • movl 8(%ebp), %eax //move contents at memory address (8 + contents(ebp)) to register eax
    • movl %eax, 8(%ebx,%ecx,4) //effective address is 8 + contents(%ebx) + 4*contents(%ecx)
Little-endian

x86 instruction set can address bytes and supports data of different sizes, so you have to be aware of the representation of data.

How are 32-bit quantities stored in memory?

Storing value 0x0A0B0C0D in memory

x86 is “little-endian”
Condition code register

- **Condition code register**
  - Bits in this register are set implicitly when instructions are executed
  - (eg) ZF bit is the zero flag and is set if the result of the operation is zero
  - (eg) SF bit is the sign flag and is set if the result of the operation is negative
  - ....

- **Branch instructions can test one or more flags and branch conditionally on the outcome**
  - (eg) je/jz is “jump if equal”: jumps if ZF is set
  - (eg) jne/jnz is “jump if not equal”
  - Many other conditional branch operations
gcc/icc stack frame

- arguments are pushed right to left
  \[ f(\text{arg1, arg2, \ldots, argN}) \]
- registers are saved by caller and callee
  gcc convention
  - caller save: eax, ecx, edx
  - callee save: ebp, ebx, esi, edi
- ebp (FBR) is one of callee save registers
- eax is used to return a value from function
- on x64, registers are used to pass arguments
Accessing Stack Variables

- To access stack variables: use offsets from ebp

- Example:
  8(%ebp) = parameter 1
  12(%ebp) = parameter 2
  -4(%ebp) = local 1
Accessing Stack Variables

- Translate accesses to variables:
  - For parameters, compute offset from %ebp using:
    - Parameter number
    - Sizes of other parameters
  - For local variables, look at data layout and assign offsets from frame pointer to each local

- Example:
  - a: local, offset-4
  - p: parameter, offset+16, q: parameter, offset+8
  - Assignment $a = p + q$ becomes equivalent to:
    $$-4(%ebp) = 16(%ebp) + 8(%ebp)$$
  - How to write this in assembly?
Arithmetic

• How to translate: p+q?
  – Assume p and q are locals or parameters
  – Determine offsets for p and q
  – Perform the arithmetic operation

• Problem: the ADD instruction in x86 cannot take both operands from memory; notation for possible operands:
  – mem32: register or memory 32 bit (similar for r/m8, r/m16)
  – reg32: register 32 bit (similar for reg8, reg16)
  – imm32: immediate 32 bit (similar for imm8, imm16)
  – At most one operand can be mem!

• Translation requires using an extra register
  – Place p into a register (e.g. %ecx): mov 16(%ebp), %ecx
  – Perform addition of q and %ecx: add 8(%ebp), %ecx
Data Movement

- **Translate** $a = p+q$:
  - **Load** memory location ($p$) into register (%ecx) using a move instr.
  - Perform the addition
  - **Store** result from register into memory location ($a$):
    
    ```
    mov 16(%ebp), %ecx (load)
    add 8(%ebp), %ecx (arithmetic)
    mov %ecx, -8(%ebp) (store)
    ```

- **Move instructions cannot have two memory operands**
  Therefore, copy instructions must be translated using an extra register:
  
  ```
  a = p \Rightarrow \quad \text{mov } 16(%ebp), %ecx
  \quad \text{mov } %ecx, -8(%ebp)
  ```

- **However, loading constants doesn’t require extra registers:**
  
  ```
  a = 12 \Rightarrow \quad \text{mov } \$12, -8(%ebp)
  ```
int plus3 (int x) { int res = x + 3; return res; }
int doit (int x) { return plus3 (x); }
int main (void) { return doit (8); }

1. _plus3:
   pushl %ebp       // save ebp
   movl %esp, %ebp  //ebp points to current frame
   pushl %esi       //save register esi
   movl 8(%ebp), %esi //x → esi
   addl $3, %esi    //esi + 3 → esi
   movl %esi, %eax  //eax now has return value
   popl %esi        //restore esi
   movl %ebp, %esp  //pop local variables
   popl %ebp        //restore ebp

2. _doit:
   pushl %ebp       // save ebp
   movl %esp, %ebp  //ebp points to current frame
   pushl 8(%ebp)    //local variable 3
   call _plus3      //call plus3
   movl %ebp, %esp  //local variable 2
   popl %ebp        //local variable 1
   pushl %ebp       //saved EBP
   pushl %ebp       //local variable 1
   pushl %ebp       //local variable 2
   pushl %ebp       //local variable 3
   pushl %ebp       //return address

3. _main:
   pushl %ebp       // save ebp
   movl %esp, %ebp  //ebp points to current frame
   pushl $8         //parameter 1
   call _doit       //call doit
   movl %ebp, %esp  //parameter 2
   popl %ebp        //parameter 3
   ret
Accessing Global Variables

- Global (static) variables and constants not stack allocated
- Have fixed addresses throughout the execution of the program
  - Compile-time known addresses (relative to the base address where program is loaded)
  - Hence, can directly refer to these addresses using symbolic names in the generated assembly code

- Example: string constants
  
  ```
  str: .string "Hello world!"
  ```
  - The string will be allocated in the static area of the program
  - Here, "str" is a label representing the address of the string
  - Can use $str as a constant in other instructions:
    ```
    push $str
    ```
Control-Flow

- **Label instructions**
  - Simply translated as labels in the assembly code
  - E.g., label2: mov $2, %ebx

- **Unconditional jumps:**
  - Use jump instruction, with a label argument
  - E.g., jmp label2

- **Conditional jumps:**
  - Translate conditional jumps using test/cmp instructions:
  - E.g., tjump b L → cmp %ecx, $0
    jnz L
    where %ecx hold the value of b, and we assume booleans are represented as
    0=false, 1=true
Array accesses in language with dynamic array size

- access \( a[i] \) requires:
  - Compute address of element: \( a + i \times \text{size} \)
  - Access memory at that address
- Can use indexed memory accesses to compute addresses
- Example: assume size of array elements is 4 bytes, and local variables \( a, i \) (offsets –4, -8)

\[
a[i] = 1 \\
mov -4(%ebp), %ebx \quad \text{(load a)} \\
mov -8(%ebp), %ecx \quad \text{(load i)} \\
mov $1, (%ebx,%ecx,4) \quad \text{(store into the heap)}
\]
Data structures: multi-dimensional arrays (I)

- **Multi-dimensional arrays**
  - Elements of array are stored sequentially in memory in some order

- **Two important orders**
  - Row-major order: elements of each row are contiguous in memory and rows are stored one after another starting from the first row (all languages other than FORTRAN)
  - Column-major order: similar to row-major but columns are stored contiguously, not rows (FORTRAN)

- **Array allocated on heap (using malloc or new)**
  - Pointer to array (address of A[0,0]) is stored on stack
Data structures: multi-dimensional arrays (II)

- **Address arithmetic:**
  - Assume array A: MxN of ints/floats/whatever (assume each element requires “size” bytes)
  - Array allocated on heap in row major order
  - Starting address of A is stored at -4(%ebp) for example
  - What is address of A[i,j]?

- **Address(A[r,c]) = -4(%ebp) + (r*N+c)*size**
Data structures: multi-dimensional arrays (III)

- Usually array elements are accessed within loops
- Optimizing compilers will optimize the address arithmetic for array access using loop invariant removal and strength reduction (see later)
- Sequential accesses to row elements
  - Register points into array
  - Incremented by “size” after each access to get to the next element
Data structures: objects

- Objects can be stack- or heap-allocated
- Example: Point type
  - Fields: x, y
  - Methods: getx, gety
- Stack allocation:
  (C++) Point p;
- Heap:
  (C++) Point *p = new Point;
  (Java) Point p = new Point();
Run-time Checks

• Run-time checks:
  – Check if array/object references are non-null
  – Check if array index is within bounds

• Example: array bounds checks:
  – if v holds the address of an array, insert array bounds checking code for v before each load (...=v[i]) or store (v[i] = ...)
  – Assume array length is stored just before array elements:

```assembly
    cmp $0, -12(%ebp)  ; (compare i to 0)
    jl ArrayBoundsError ; (test lower bound)
    mov -8(%ebp), %ecx   ; (load v into %ecx)
    mov -4(%ecx), %ecx   ; (load array length into %ecx)
    cmp -12(%ebp), %ecx  ; (compare i to array length)
    jle ArrayBoundsError ; (test upper bound)
```
...
# X86 Assembly Syntax

- Two different notations for assembly syntax:
  - AT&T syntax and Intel syntax
  - In the examples: AT&T (gcc) syntax

- Summary of differences:

<table>
<thead>
<tr>
<th></th>
<th>AT&amp;T</th>
<th>Intel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Order of operands</td>
<td>op a, b : b is destination</td>
<td>op a, b : a is destination</td>
</tr>
<tr>
<td>Memory addressing</td>
<td>disp(base,offset,scale)</td>
<td>[base + offset*scale + disp]</td>
</tr>
<tr>
<td>Size of memory operands</td>
<td>instruction suffixes (b,w,l) (e.g., movb, movw, movl)</td>
<td>operand prefixes (byte ptr, word ptr, dword ptr)</td>
</tr>
<tr>
<td>Registers</td>
<td>%eax, %ebx, etc.</td>
<td>eax, ebx, etc.</td>
</tr>
<tr>
<td>Constants</td>
<td>$4, $foo, etc</td>
<td>4, foo, etc</td>
</tr>
</tbody>
</table>
Tutorial

• This website has a simple example with comments

https://eli.thegreenplace.net/2011/02/04/where-the-top-of-the-stack-is-on-x86/
Introduction to Compilers
Optimizing compiler structure

Source code
if (b == 0) a = b;

Understand source code

Intermediate code

Optimize

Intermediate code

Generate assembly code

Assembly code
cmp $0, ecx
cmovz edx, ecx

Front end (machine-independent)

Optimizer

Back end (machine-dependent)
Syntax analysis is also known as parsing.
What Next?

• At this point we could generate assembly code

• Better:
  – Optimize the program first
  – Then generate code

• If optimization performed at the IR level, then they apply to all target machines
Optimizations

Source code
(character stream)
if (b == 0) a = b;

Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation

Errors

Correct program
In High IR (usually trees)

Optimize

Program
In Low IR (closer to assembly)

Optimize
When to Apply Optimizations

**High IR**
- Function inlining
- Function cloning
- Constant folding
- Constant propagation
- Value numbering
- Dead code elimination

**Low IR**
- Loop-invariant code motion
- Common sub-expression elimination
- Strength reduction
- Constant folding & propagation
- Branch prediction/optimization
- Loop unrolling

**Assembly**
- Register allocation
- Cache optimization
What are Optimizations?

• **Optimizations** = code transformations that *improve* the program

• **Different kinds**
  – space optimizations: improve (reduce) memory use
  – time optimizations: improve (reduce) execution time

• **Code transformations must be safe!**
  – They must preserve the meaning of the program
Why Optimize?

- Programmers don’t always write optimal code – can recognize ways to improve code (e.g., avoid recomputing same expression)

- High-level language may make some optimizations inconvenient or impossible to express

\[
a[i][j] = a[i][j] + 1;
\]

- High-level unoptimized code may be more readable: cleaner, modular

```c
int square(x) { return x*x; }
```
Where to Optimize?

• Usual goal: improve time performance
• Problem: many optimizations trade off space versus time
• Example: loop unrolling
  – Increases code space, speeds up one loop
  – Frequently executed code with long loops: space/time tradeoff is generally a win
  – Infrequently executed code: may want to optimize code space at expense of time
• Want to optimize program hot spots
Many Possible Optimizations

• Many ways to optimize a program
• Some of the most common optimizations:
  
  Function Inlining
  Function Cloning
  Constant folding
  Constant propagation
  Dead code elimination
  Loop-invariant code motion
  Common sub-expression elimination
  Strength reduction
  Branch prediction/optimization
  Loop unrolling
Constant Propagation

- If value of variable is known to be a constant, replace use of variable with constant
- Example:
  
  \[
  n = 10 \\
  c = 2 \\
  \text{for (i=0; i<n; i++) \{} s = s + i*c; \}\]

- Replace n, c:
  
  \[
  \text{for (i=0; i<10; i++) \{} s = s + i*2; \}\]

- Each variable must be replaced only when it has known constant value:
  - Forward from a constant assignment
  - Until next assignment of the variable
Constant Folding

• Evaluate an expression if operands are known at compile time (i.e., they are constants)

• Example:
  \[ x = 1.1 \times 2; \quad \Rightarrow \quad x = 2.2; \]

• Performed at every stage of compilation
  – Constants created by translations or optimizations

  \[
  \text{int } x = a[2] \quad \Rightarrow \quad t1 = 2 \times 4 \\
  t2 = a + t1 \\
  x = *t2
  \]
Algebraic Simplification

• More general form of constant folding: take advantage of usual simplification rules

\[
\begin{align*}
\text{a} \times \text{1} & \Rightarrow \text{a} & \text{a} \times \text{0} & \Rightarrow \text{0} \\
\text{a} / \text{1} & \Rightarrow \text{a} & \text{a} + \text{0} & \Rightarrow \text{a} \\
\text{b} \land \land \text{false} & \Rightarrow \text{b} & \text{b} \& \& \text{true} & \Rightarrow \text{b}
\end{align*}
\]

• Repeatedly apply the above rules

\[
(y\times1+0)/1 \Rightarrow y\times1+0 \Rightarrow y\times1 \Rightarrow y
\]

• Must be careful with floating point!
Copy Propagation

• After assignment \( x = y \), replace uses of \( x \) with \( y \)
• Replace until \( x \) is assigned again

\[
\begin{align*}
x &= y; \\
\text{if } (x > 1) \\
s &= x \times f(x - 1); \\
\Rightarrow \\
\text{if } (y > 1) \\
s &= y \times f(y - 1);
\end{align*}
\]

• What if there was an assignment \( y = z \) before?
  – Transitively apply replacements
Common Subexpression Elimination

• If program computes same expression multiple time, can reuse the computed value

• Example:

\[
\begin{align*}
a & = b+c; \\
c & = b+c; \\
d & = b+c;
\end{align*}
\]

⇒

\[
\begin{align*}
a & = b+c; \\
c & = a; \\
d & = b+c;
\end{align*}
\]

• Common subexpressions also occur in low-level code in address calculations for array accesses:

\[
a[i] = b[i] + 1;
\]
Unreachable Code Elimination

- Eliminate code that is never executed
- Example:
  ```
  #define debug false
  s = 1;
  if (debug)
    print(“state = ”, s);
  ⇒ s = 1;
  ```

- Unreachable code may not be obvious in low IR (or in high-level languages with unstructured “goto” statements)
Unreachable Code Elimination

- Unreachable code in while/if statements when:
  - Loop condition is always false (loop never executed)
  - Condition of an if statement is always true or always false (only one branch executed)

  \[
  \text{if (false) } S \Rightarrow ; \\
  \text{if (true) } S \text{ else } S' \Rightarrow S \\
  \text{if (false) } S \text{ else } S' \Rightarrow S' \\
  \text{while (false) } S \Rightarrow ; \\
  \text{while (2>3) } S \Rightarrow ;
  \]
Dead Code Elimination

• If effect of a statement is never observed, eliminate the statement

\[ x = y + 1; \]
\[ y = 1; \]
\[ x = 2z; \]

\[ \Rightarrow \quad y = 1; \]
\[ x = 2z; \]

• Variable is *dead* if value is never used after definition

• Eliminate assignments to dead variables

• Other optimizations may create dead code
Loop Optimizations

• Program hot spots are usually loops (exceptions: OS kernels, compilers)
• Most execution time in most programs is spent in loops: 90/10 is typical
• Loop optimizations are important, effective, and numerous
Loop-Invariant Code Motion

• If result of a statement or expression does not change during loop, and it has no externally-visible side-effect (!), can hoist its computation out of the loop

• Often useful for array element addressing computations – invariant code not visible at source level

• Requires analysis to identify loop-invariant expressions
Code Motion Example

• Identify invariant expression:

\[
\begin{align*}
\text{for}(i=0; i<n; i++) \\
a[i] &= a[i] + (x \times x)/(y \times y);
\end{align*}
\]

• Hoist the expression out of the loop:

\[
\begin{align*}
\text{c} &= (x \times x)/(y \times y); \\
\text{for}(i=0; i<n; i++) \\
a[i] &= a[i] + c;
\end{align*}
\]
Another Example

• Can also hoist statements out of loops
• Assume x not updated in the loop body:

```plaintext
... 
while (…) {
    y = x*x;     \(\Rightarrow\)  
    ... 
}
... 

\(\Rightarrow\)

... 
while (…) {
    y = x*x;     
    ...
}
... 
```

• ... Is it safe?
**Strength Reduction**

- Replaces expensive operations (multiplies, divides) by cheap ones (adds, subtrahes)
- Strength reduction more effective in loops and useful for address arithmetic
- **Induction variable** = loop variable whose value is depends linearly on the iteration number
- Apply strength reduction to induction variables

```plaintext
s = 0;
for (i = 0; i < n; i++) {
    v = 4*i;
    s = s + v;
}
```

```plaintext
s = 0; v = -4;
for (i = 0; i < n; i++) {
    v = v + 4;
    s = s + v;
}
```

⇒

```plaintext
s = 0;
for (i = 0; i < n; i++) {
    v = 4*i;
    s = s + v;
}
```
Strength Reduction

- Can apply strength reduction to computation other than induction variables:

  \[ x \times 2 \Rightarrow x + x \]
  \[ i \times 2^c \Rightarrow i \ll c \]
  \[ i / 2^c \Rightarrow i \gg c \]
Induction Variable Elimination

• If there are multiple induction variables in a loop, can
  eliminate the ones that are used only in the test condition
• Need to rewrite test using the other induction variables
• Usually applied after strength reduction

\[ s = 0; v = -4; \]
\[ \text{for}\ (i = 0; i < n; i++)\ \{\]
  \[ v = v + 4; \]
  \[ s = s + v; \]
\[ \}\]
\[ \Rightarrow \]
\[ s = 0; v = -4; \]
\[ \text{for}\ (v < (4*n-4);)\ \{\]
  \[ v = v + 4; \]
  \[ s = s + v; \]
\[ \}\]
Loop Unrolling

• Execute loop body multiple times at each iteration

• Example:
  
  ```
  for (i = 0; i < n; i++) { S }
  ```

• Unroll loop four times:
  
  ```
  for (i = 0; i < n-3; i+=4) { S; S; S; S; }
  for (; i < n; i++) S;
  ```

• Gets rid of \( \frac{3}{4} \) of conditional branches!

• Space-time tradeoff: program size increases
Function Inlining

• Replace a function call with the body of the function:

  int g(int x) { return f(x)-1; }
  int f(int n) { int b=1; while (n--) { b = 2*b }; return b; }

  int g(int x) { int r;
    int n = x;
    { int b =1; while (n--) { b = 2*b }; r = b }
    return r – 1; }

• Can inline methods, but more difficult
• ... how about recursive procedures?
Function Cloning

• Create specialized versions of functions that are called from different call sites with different arguments

```c
void f(int x[], int n, int m) {
    for(int i=0; i<n; i++) { x[i] = x[i] + i*m; }
}
```

• For a call `f(a, 10, 1)`, create a specialized version of `f`:

```c
void f1(int x[]) {
    for(int i=0; i<10; i++) { x[i] = x[i] + i; }
}
```

• For another call `f(b, p, 0)`, create another version `f2(…)`
When to Apply Optimizations

High IR
- Function inlining
- Function cloning
- Constant folding
- Constant propagation
- Value numbering
- Dead code elimination
- Loop-invariant code motion
- Common sub-expression elimination
- Strength reduction
- Constant folding & propagation
- Branch prediction/optimization
- Loop unrolling
- Register allocation
- Cache optimization

Low IR

Assembly
Summary

• Many useful optimizations that can transform code to make it faster

• Whole is greater than sum of parts: optimizations should be applied together, sometimes more than once, at different levels