CMSC 430
Introduction to Compilers
Fall 2014

Code Generation
Code generation is the process of moving from “highest level” IR down to machine code
- Usually takes place after data flow analysis

Three major components
- Instruction selection — Map IR into assembly code
- Instruction scheduling — Reorder operations
  - Hide latencies in pipelined machines, ensure code obeys processor constraints
  - Modern processors do a lot of this already, and they have better information than the compiler...
- Register allocation — Go from unbounded to finite reg set
  - Implies not all variables can always be in registers

These problems are tightly coupled
- But typically done separately in compilers
Code quality

• Compilers need to produce good “quality” code
  ▪ This used to mean: code should match what an expert assembly programmer would write
  ▪ With modern languages it’s much more unclear, but it mostly comes down to **performance**
    - back-end needs to know ins and outs of target machine code
      - What kind of code can the machine run efficiently?
      - When does the machine need extra help from the compiler?
      - Rise of bytecode: fulfills a long-standing idea of splitting front- and back-end of compiler up, and reusing them in many combinations
    - code generation cannot always be optimal
      - Benchmarking (e.g., SPEC) plays big role in code generator design
      - Compiler vendors play lots of games to do well on benchmarks
  - Rule of thumb: expose as much information as possible
Example: boolean operators

• How should these be represented?
  ▪ Depends on the target machine and how they are used

• Example 1: If-then-else, x86, gcc

```c
if (x < y)
  a = b + c;
else
  a = d + e;
```

```assembly
cmp rx, ry   // result in EFLAGS
jge l1
add ra, rb, rc
jmp l2
l1: add ra, rd, re
l2: nop
```
Boolean operators (cont’d)

• Example 2: Standalone, x86, gcc

\[ a = (x < y); \]

```c
cmp rx, ry        // result in EFLAGS
setl %al          // 16-bit instruction
andb $1, %al      // only low bit set
movzbl %al, %eax  // extend to 32-bits
```
Boolean operations (cont’d)

• Example 3: If-then-else, Lua bytecode

```lua
local a, b, c, d, e, x, y;
if (x < y) then
    a = b + c;
else
    a = d + e;
end
```

```
l1: add R0, R3, R4
l2: return
```

```
l0, R5, R6 // skip next instr if R5 < R6 true
jmp l1     // pc += 2
add R0, R1, R2
jmp l2     // pc += 1
```

```
l1: add R0, R3, R4
l2: return
```
Boolean operations (cont’d)

• Example 4: Stand-alone, Lua

```lua
local a, x, y;
a = (x < y)
```

```
l1: loadbool R0, 1, l2 // R0 <- 1, fall through to l2
l2: return
```

```assembly
lt 1, R1, R2 // skip next instr if R1 < R2 true
jmp l1       // pc += 1
loadbool R0, 0, l2 // R0 <- 0, jump to l2
l1: loadbool R0, 1, l2 // R0 <- 1, fall through to l2
l2: return
```
Example: case statements

• Consider compiling a case/switch statement with n guards
  ▪ How expensive is it to decide which arm applies?

• Option 1: Cascaded if-then-else
  ▪ $O(n)$ — linear in the number of cases, and actual cost depends on where matching arm occurs

• Option 2: Binary search
  ▪ $O(\log n)$ — but needs guards that are totally ordered

• Option 3: Jump table
  ▪ $O(1)$ — but best when guards are dense (e.g., ints 0..10)

• No amount of “optimization” will covert one of these forms into another
Instruction selection

- Arithmetic exprs, global vars, if-then-else
  - See codegen*.ml files on web site
Instruction selection — loops

while (b) do s;  do s while (b);  for (init; b; post) s;
Multi-dimensional arrays

- Conceptually

- Row-major order (most languages)

- Column-major order (Fortran)

- Indirection vectors (Java)
Computing an array address

• \( a[i] \)
  - \( a + i \times \text{sizeof}(\ast a) \)
    - Here \( a \) is the base address of the array, and assume array 0-based

• \( a[i][j] \)
  - Row-major order
    - \( a + i \times \text{sizeof}(\ast a) + j \times \text{sizeof}(\ast \ast a) \)
    - Here \( \text{sizeof}(\ast a) \) is the size of a row or column, as appropriate
    - Much more arithmetic needed if array not 0-based
  - Column-major order
    - \( a + j \times \text{sizeof}(\ast a) + i \times \text{sizeof}(\ast \ast a) \)
  - Indirection vectors
    - \( *(a + i \times \text{sizeof}(\text{pointer})) + j \times \text{sizeof}(\ast \ast a) \)
Functions

• (Aka procedure, subroutine, routine, method, ...)
• Fundamental abstraction of computing
  ▪ Reusable grouping of code
  ▪ Usually also introduces a lexical scope/name space

• Calling conventions to interact with system, libraries, or separately compiled code
  ▪ In these cases, don’t have access to other code at compile time
    - Must have standard for passing parameters, return values, invariants maintained across function call, etc
  ▪ Don’t necessarily need to obey these “within” the language
    - But deviating from them reduces utility of system tools
Terminology

• Run time vs. compile time
  ▪ The code that implements the calling convention is executed at *run time*
  ▪ The code is generated at *compile time*

• Caller vs. callee
  ▪ Caller — that function that made the call
  ▪ Callee — the function that was called
Function invoked at call site
- Control returns to call site when function returns
  - Need to save and restore a “return address”

Function calls may be recursive
- Need a stack of return addresses

Need storage for parameters and local variables

Must preserve caller’s state
- Stack needs space for these

Stack consists of activation records
- We’ll see what these look like and how they are set up next
Activation Record Basics

One AR for each invocation of a procedure
Procedure Linkages

Standard procedure linkage

Procedure has
- standard **prolog**
- standard **epilog**

Each call involves a
- **pre-call** sequence
- **post-return** sequence

These are completely predictable from the call site
\(\Rightarrow\) depend on the number &
type of the actual parameters
Pre-call sequence

• Sets up callee’s basic AR
• Helps preserve its own environment

• The Details
  ▪ Allocate space for the callee’s AR
    - except space for local variables
  ▪ Evaluates each parameter & stores value or address
  ▪ Saves return address, caller’s ARP into callee’s AR
  ▪ Save any caller-save registers
    - Save into space in caller’s AR
  ▪ Jump to address of callee’s prolog code
Post-return sequence

• Finish restoring caller’s environment
• Place any value back where it belongs

• The Details
  ▪ Copy return value from callee’s AR, if necessary
  ▪ Free the callee’s AR
  ▪ Restore any caller-save registers
  ▪ Copy back call-by-value/result parameters
  ▪ Continue execution after the call
Prolog code

• Finish setting up callee’s environment
• Preserve parts of caller’s environment that will be disturbed

• The Details
  ■ Preserve any callee-save registers
  ■ Allocate space for local data
    - Easiest scenario is to extend the AR
  ■ Handle any local variable initializations
Epilog code

• Wind up the business of the callee
• Start restoring the caller’s environment

• The Details
  ▪ Store return value?
    - Some implementations do this on the return statement
    - Others have return assign it & epilog store it into caller’s AR
    - Still others (x86) store it in a register
  ▪ Restore callee-save registers
  ▪ Free space for local data, if necessary
  ▪ Load return address from AR
  ▪ Restore caller’s ARP
  ▪ Jump to the return address
Concrete example: x86

• The CPU has a fixed number of registers
  ▪ Think of these as memory that’s really fast to access
  ▪ For a 32-bit machine, each can hold a 32-bit word

• Important x86 registers
  ▪ eax  generic register for computing values
  ▪ esp  pointer to the top of the stack
  ▪ ebp  pointer to start of current stack frame
  ▪ eip  the program counter (points to next instruction in text segment to execute)
x86 activation record

- The stack just after `f` transfers control to `g`

Based on Fig 6-1 in Intel ia-32 manual
x86 calling convention

- To call a function
  - Push parameters for function onto stack
  - Invoke **CALL** instruction to
    - Push current value of **eip** onto stack
      - I.e., save the program counter
      - Start executing code for called function
  - Callee pushes **ebp** onto stack to save it

- When a function returns
  - Put return value in **eax**
  - Invoke **RET** instruction to load return address into **eip**
    - I.e., start executing code where we left off at call
Example

```c
int f(int a, int b) {
    return a + b;
}

int main(void) {
    int x;
    x = f(3, 4);
}
```

```assembly
f:
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    movl 12(%ebp), %eax
    movl 8(%ebp), %ecx
    movl %ecx, -4(%ebp)
    movl %eax, -8(%ebp)
    movl -4(%ebp), %eax
    addl -8(%ebp), %eax
    addl $8, %esp
    popl %ebp
    retl

main:
    ...
    movl $3, %eax
    movl $4, %ecx
    movl $3, (%esp)
    movl $4, 4(%esp)
    movl %eax, -8(%ebp)
    movl %ecx, -12(%ebp)
    calll f
    movl %eax, -4(%ebp)
    ...
```

gcc -m32 -S a.c
Example

```c
int f(int a, int b) {
    return a + b;
}

int main(void) {
    int x;
    x = f(3, 4);
}
```

gcc -m32 -S a.c
Example

```c
int f(int a, int b) {
    return a + b;
}

int main(void) {
    int x;
    x = f(3, 4);
}
```

```
gcc -m32 -S -O3 a.c
```
Lots more details

- There’s a whole lot more to say about calling functions
  - Local variables are allocated on stack by the callee as needed
    - This is usually the first thing a called function does, by incrementing esp
  - Saving registers
    - If the callee is going to use eax itself, you’d better save it to the stack before you call
  - Passing parameters in registers
    - More efficient than pushing/popping from the stack
    - Can be done if caller and callee cooperate
    - (But watch out for extern functions that could be called from anywhere)
  - Etc...
Even more details

• Different languages/OS’s can have different conventions
  ▪ And conventions have changed over time

• System call interface is different application-level interface
  ▪ Need to switch into kernel mode in some way
  ▪ Details depend on OS
  ▪ Typically, syscalls wrapped by standard library
    - E.g., calling open() in C calls into libc, which does some high-level stuff and then does a syscall
    - Syscall code often implemented as inline assembly
Higher-order languages

• If a called function can outlive its caller, need to keep activation record on the heap
  ■ `fun x -> (fun y -> x + y)`
  ■ I.e., we need *closures* for these

• These get allocated basically like we saw in 330
  ■ Try to avoid allocating these if curried functions called with all arguments at once
Memory layout

- Code, static, and global data have known size
  - Can refer to entities by predetermined offsets
    - (Note: ASLR used to prevent attackers from guessing these)
  - Heap and stack both grow and shrink over time
    - Better utilization if stack and heap grow toward each other (Knuth)

- Note this is a *virtual* address space
The really big picture

Compiler's view

OS's view

Hardware's view

Virtual address spaces

Physical address space

Hardware's view

OS's view

Compiler's view
The really small picture

Source: https://en.wikipedia.org/wiki/Page_table
Linking

• Many languages support separate compilation
  ▪ Individual modules or components are compiled by themselves, without needing to recompile the modules or components they depend on
  ▪ Can dramatically reduce time to recompile program when program is changed

• *Linking* combines components together
  ▪ In C and OCaml, linking is an explicit phase
  ▪ In Java, linking is implicit as dependencies are loaded by the JVM

• Linkers often support *shared libraries*
  ▪ Shared lib code appears only once on disk for all apps
  ▪ Shared lib can be updated, apps automatically see new version
    - → linking against shared lib only checks existence (and maybe type) of symbol
  ▪ Shard lib code must be *position independent*
Linking example

Makefile
all: main.o lib.o
    gcc main.o lib.o -o prog

lib.o: lib.c
    gcc -c lib.c

main.o: main.c
    gcc -c main.c

main.c
extern int print_s(const char *);
int main() {
    print_s("Hello, world!");
}

lib.c
#include <stdio.h>
void print_s(const char *s) {
    printf("%s", s);
}
gcc compilation process

Loading

• OS needs to know many things about a program
  ■ Where is the program code
  ■ Where are values for the data segment
  ■ How should the program be started
  ■ What shared libs does the program refer to

• Thus, compilers must create an executable that is in a standard format
  ■ E.g., ELF on Linux, PE32+ on Windows, Mach-O on OS X

• Details of all these can be found on the web, in man pages, and in developer documentation
 ELF

```c
int x = 1010101;
char *s = "Hello, world!\n";

int main() {
    int x=1;
    return x;
}
```

gcc -o elf.c
objcump -D elf.o
Stack SMASHING!

- **Buffer overflow**

  ![Diagram of stack with buffer overflow]

  - `*str` array
  - `ret` return value
  - `exec("/bin/sh")` executed command

- **strcpy()** - what if bounds aren’t checked?

  ![Diagram of stack with strcpy()]

  - `*str` array
  - `ret` return value
  - `fake_ret` fake return value
  - `"/bin/sh"` command to execute

- **Return to libc**

  ![Diagram of stack with return to libc]

  - `*str` array
  - `ret` return value
  - `fake_ret` fake return value
  - `"/bin/sh"` command to execute

  ![Top of stack](image)
Stack SMASHING! (defences)

- Canary values
  - inject random values in between stack frames
  - check those values during function call

- Address Space Layout Randomisation
  - randomise the layout of key data areas (heap, stack, libraries)

```c
int main () {
    register int *ebp asm("ebp");
    printf("%p\n", ebp);
}
```

```
$: ./randomlayout
0x7fff67835036
$: ./randomlayout
0x7fff663e5036
```
Compiling objects and classes

- **Object** = record with data (fields) and code (meths)
  - In a classless OO language, in general case need to treat each object separately

- **Class** = set of objects with same meths
  - ⇒ All insts of a class can share memory used for meth code
    - (But, each inst has its own fields)

- **Virtual method table (vtable)** contains pointers to methods of class
  - Object record points to vtable, and then vtable used to resolve dynamic dispatch
Example

class A { int f; void m1(void) { ... } }

a1 = A.new();
a2 = A.new();

- The vtable includes the class type (for run-time type tests) and a function pointer for each method
  - At `x.m1()`, call `(x->vtable[0])(())`
  - (Note we know the offset of m1 from the type of x)
Single Inheritance

class A { int f; void m1(void) { ... } }
class B extends A { int g; void m2(void) { ... } }
a = A.new();
b = B.new();

- Ensure superclass layouts are prefixes of subclass layouts
  - At x.m1(), still call (x->vtable[0])()
  - At x.m2(), call (x->vtable[1])()
Multiple inheritance

- (Notice that D overrides method m1)
- Much more complicated!
  - Separate compilation, so don’t know full inheritance hierarchy
  - Must support both up- and downcasts
  - Want method lookup to be efficient
- Solutions? Several—see web for details!