Lecture 7: Unoptimized Code Generation

From the intermediate representation to the machine code
Outline

- Introduction
- Machine Language
- Overview of a modern processor
- Memory Layout
- Procedure Abstraction
- Procedure Linkage
- Guidelines in Creating a Code Generator
Anatomy of a compiler

Program (character stream)

Lexical Analyzer (Scanner)

Token Stream

Syntax Analyzer (Parser)

Parse Tree

Semantic Analyzer

Intermediate Representation

Intermediate Code Optimizer

Optimized Intermediate Representation

Code Generator

Assembly code
Anatomy of a compiler

Program (character stream)

Lexical Analyzer (Scanner)

Token Stream

Syntax Analyzer (Parser)

Parse Tree

Semantic Analyzer

High-level IR

Low-level IR

Intermediate Representation

Code Generator

Assembly code
Components of a High Level Language

**CODE**
- Procedures
- Control Flow
- Statements
- Data Access

**DATA**
- Global Static Variables
- Global Dynamic Data
- Local Variables
- Temporaries
- Parameter Passing
- Read-only Data
Machine Code Generator Should...

• Translate all the instructions in the intermediate representation to assembly language
• Allocate space for the variables, arrays etc.
• Adhere to calling conventions
• Create the necessary symbolic information
Outline

• Introduction
• Machine Language
• Overview of a modern processor
• Memory Layout
• Procedure Abstraction
• Procedure Linkage
• Guidelines in Creating a Code Generator
Machines understand understand...

<table>
<thead>
<tr>
<th>LOCATION</th>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0046</td>
<td>8B45FC</td>
</tr>
<tr>
<td>0049</td>
<td>4863F0</td>
</tr>
<tr>
<td>004c</td>
<td>8B45FC</td>
</tr>
<tr>
<td>004f</td>
<td>4863D0</td>
</tr>
<tr>
<td>0052</td>
<td>8B45FC</td>
</tr>
<tr>
<td>0055</td>
<td>4898</td>
</tr>
<tr>
<td>0057</td>
<td>8B048500</td>
</tr>
<tr>
<td></td>
<td>000000</td>
</tr>
<tr>
<td>005e</td>
<td>8B149500</td>
</tr>
<tr>
<td></td>
<td>000000</td>
</tr>
<tr>
<td>0065</td>
<td>01C2</td>
</tr>
<tr>
<td>0067</td>
<td>8B45FC</td>
</tr>
<tr>
<td>006a</td>
<td>4898</td>
</tr>
<tr>
<td>006c</td>
<td>89D7</td>
</tr>
<tr>
<td>006e</td>
<td>033C8500</td>
</tr>
<tr>
<td></td>
<td>000000</td>
</tr>
<tr>
<td>0075</td>
<td>8B45FC</td>
</tr>
<tr>
<td>0078</td>
<td>4863C8</td>
</tr>
<tr>
<td>007b</td>
<td>8B45F8</td>
</tr>
<tr>
<td>007e</td>
<td>4898</td>
</tr>
<tr>
<td>0080</td>
<td>8B148500</td>
</tr>
<tr>
<td></td>
<td>000000</td>
</tr>
</tbody>
</table>
### Machines understand...

<table>
<thead>
<tr>
<th>LOCATION</th>
<th>DATA</th>
<th>ASSEMBLY INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0046</td>
<td>8B45FC</td>
<td>movl -4(%rbp), %eax</td>
</tr>
<tr>
<td>0049</td>
<td>4863F0</td>
<td>movslq %eax,%rsi</td>
</tr>
<tr>
<td>004c</td>
<td>8B45FC</td>
<td>movl -4(%rbp), %eax</td>
</tr>
<tr>
<td>004f</td>
<td>4863D0</td>
<td>movslq %eax,%rdx</td>
</tr>
<tr>
<td>0052</td>
<td>8B45FC</td>
<td>movl -4(%rbp), %eax</td>
</tr>
<tr>
<td>0055</td>
<td>4898</td>
<td>cltq</td>
</tr>
<tr>
<td>0057</td>
<td>8B048500</td>
<td>movl B(,%rax,4), %eax</td>
</tr>
<tr>
<td></td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>005e</td>
<td>8B149500</td>
<td>movl A(,%rdx,4), %edx</td>
</tr>
<tr>
<td></td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>0065</td>
<td>01C2</td>
<td>addl %eax, %edx</td>
</tr>
<tr>
<td>0067</td>
<td>8B45FC</td>
<td>movl -4(%rbp), %eax</td>
</tr>
<tr>
<td>006a</td>
<td>4898</td>
<td>cltq</td>
</tr>
<tr>
<td>006c</td>
<td>89D7</td>
<td>movl %edx, %edi</td>
</tr>
<tr>
<td>006e</td>
<td>033C8500</td>
<td>addl C(,%rax,4), %edi</td>
</tr>
<tr>
<td></td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>0075</td>
<td>8B45FC</td>
<td>movl -4(%rbp), %eax</td>
</tr>
<tr>
<td>0078</td>
<td>4863C8</td>
<td>movslq %eax,%rcx</td>
</tr>
<tr>
<td>007b</td>
<td>8B45F8</td>
<td>movl -8(%rbp), %eax</td>
</tr>
<tr>
<td>007e</td>
<td>4898</td>
<td>cltq</td>
</tr>
<tr>
<td>0080</td>
<td>8B148500</td>
<td>movl B(,%rax,4), %edx</td>
</tr>
</tbody>
</table>
Program (character stream)
Lexical Analyzer (Scanner)
Token Stream
Syntax Analyzer (Parser)
Parse Tree
Intermediate Code Generator
Intermediate Representation
Code Generator
Assembly code
Assembler & linker
Binary executable
Processor
Assembly language

• Advantages
  – Simplifies code generation due to use of symbolic instructions and symbolic names
  – Logical abstraction layer
  – Multiple Architectures can describe by a single assembly language
  ⇒ can modify the implementation
    • macro assembly instructions

• Disadvantages
  – Additional process of assembling and linking
  – Assembler adds overhead
Assembly language

• Relocatable machine language (object modules)
  – all locations(addresses) represented by symbols
  – Mapped to memory addresses at link and load time
  – Flexibility of separate compilation
• Absolute machine language
  – addresses are hard-coded
  – simple and straightforward implementation
  – inflexible -- hard to reload generated code
  – Used in interrupt handlers and device drivers
Assembly example

```assembly
    .section .rodata
.LC0:
0000 6572726F7200         .string "error"
.text
.globl fact
    .globl fact
fact:
0000 55                    pushq   %rbp
0001 4889E5                movq    %rsp, %rbp
0004 4883EC10              subq    $16, %rsp
0008 897DFC                movl    %edi, -4(%rbp)
000b 837DFC00              cmpl    $0, -4(%rbp)
000f 7911                  jns     .L2
0011 BF00000000            movl    $.LC0, %edi
0016 B800000000            movl    $0, %eax
001b E800000000            call    printf
0020 EB22                  jmp     .L3
.L2:
0022 837DFC                movl    -4(%rbp), %edi
0026 7509                  jne    .L4
0028 C745F8010000000       movl    $1, -8(%rbp)
002f EB13                  jmp     .L3
.L4:
0031 8B7DFC                movl    -4(%rbp), %edi
0034 FFCF                  decl    %edi
0036 E8000000000          call     fact
003b 0FAF45FC              imull   -4(%rbp), %eax
003f 8945F8                movl    %eax, -8(%rbp)
0042 EB00                  jmp     .L1
.L3:
0044 8B45F8                movl    -8(%rbp), %eax
0047 C9                    leave
0048 C3                    ret
```
Composition of an Object File

• We use the ELF file format

• The object file has:
  – Multiple Segments
  – Symbol Information
  – Relocation Information

• Segments
  – Global Offset Table
  – Procedure Linkage Table
  – Text (code)
  – Data
  – Read Only Data

```
.file  "test2.c"
.section .rodata
.LC0:
.string "error %d"
.section .text
.globl fact
fact:
pushq   %rbp
movq    %rsp, %rbp
subq    $16, %rsp
movl    -8(%rbp), %eax
leave
ret

.comm   bar,4,4
.comm   a,1,1
.comm   b,1,1

.section .eh_frame,"a",@progbits
.long   .LECIE1-.LSCIE1
.long   0x0
.byte   0x1
.string ""
.uleb128 0x1
```
Outline

• Introduction
• Machine Language
• Overview of a modern processor
• Memory Layout
• Procedure Abstraction
• Procedure Linkage
• Guidelines in Creating a Code Generator
Overview of a modern processor

- ALU
- Control
- Memory
- Registers
Arithmetic and Logic Unit

• Performs most of the data operations
• Has the form:
  OP <oprnd₁>, <oprnd₂>
  – <oprnd₂> = <oprnd₁> OP <oprnd₂>
  Or
  OP <oprnd₁>
• Operands are:
  – Immediate Value $25
  – Register %rax
  – Memory 4(%rbp)
• Operations are:
  – Arithmetic operations (add, sub, imul)
  – Logical operations (and, sal)
  – Unitary operations (inc, dec)
Arithmetic and Logic Unit

• Many arithmetic operations can cause an exception
  – overflow and underflow
• Can operate on different data types
  – addb 8 bits
  – addw 16 bits
  – addl 32 bits
  – addq 64 bits (Decaf is all 64 bit)
  – signed and unsigned arithmetic
  – Floating-point operations (separate ALU)
Control

• Handles the instruction sequencing
• Executing instructions
  – All instructions are in memory
  – Fetch the instruction pointed by the PC and execute it
  – For general instructions, increment the PC to point to the next location in memory
Control

• Unconditional Branches
  – Fetch the next instruction from a different location
  – Unconditional jump to an address
    jmp .L32
  – Unconditional jump to an address in a register
    jmp %rax
  – To handle procedure calls
    call fact    call %r11
Control

• All arithmetic operations update the condition codes (rFLAGS)

• Compare explicitly sets the rFLAGS
  – cmp $0, %rax

• Conditional jumps on the rFLAGS
  – Jxx .L32 Jxx 4(%rbp)
  – Examples:
    • JO Jump Overflow
    • JC Jump Carry
    • JAE Jump if above or equal
    • JZ Jump is Zero
    • JNE Jump if not equal
Control

• Control transfer in special (rare) cases
  – traps and exceptions
  – Mechanism
    • Save the next(or current) instruction location
    • find the address to jump to (from an exception vector)
    • jump to that location
When to use what?

- Give an example where each of the branch instructions can be used
  1. `jmp L0`
  2. `call L1`
  3. `jmp %rax`
  4. `jz -4(%rbp)`
  5. `jne L1`
Memory

• Flat Address Space
  – composed of words
  – byte addressable

• Need to store
  – Program
  – Local variables
  – Global variables and data
  – Stack
  – Heap
Memory

- Dynamic
- Stack
- Data
- Text
- Unmapped
- Heap
- Globals/Read-only data
- Program

0x800 0000 0000

0x40 0000

0x0

Memory

Registers

ALU

Control

©MIT Fall 2001
Registers

• Instructions allow only limited memory operations
  – add -4(%rbp), -8(%rbp)
  add %r10, -8(%rbp)

• Important for performance
  – limited in number

• Special registers
  – %rbp base pointer
  – %rsp stack pointer
Moving Data

- `mov source dest`
  - Moves data
    - from one register to another
    - from registers to memory
    - from memory to registers

- `push source`
  - Pushes data into the stack

- `pop dest`
  - Pops data from the stack to `dest`
Other interactions

- Other operations
  - Input/Output
  - Privilege / secure operations
  - Handling special hardware
    - TLBs, Caches etc.

- Mostly via system calls
  - hand-coded in assembly
  - compiler can treat them as a normal function call
Outline

• Introduction
• Machine Language
• Overview of a modern processor
• Memory Layout
• Procedure Abstraction
• Procedure Linkage
• Guidelines in Creating a Code Generator
Components of a High Level Language

- Control Flow
- Procedures
- Statements
- Data Access
- Global Static Variables
- Global Dynamic Data
- Local Variables
- Temporaries
- Parameter Passing
- Read-only Data
Memory Layout

• Heap management
  – free lists

• starting location in the text segment
Allocating Read-Only Data

• All Read-Only data in the text segment
• Integers
  – use load immediate
• Strings
  – use the .string macro

```
.section .text
.globl main
main:
  enter $0, $0
  movq $5, x(%rip)
  push x(%rip)
  push $.msg
  call printf_035
  add $16, %rsp
  leave
  ret

/msg:
  .string "Five: %d\n"
```
Global Variables

• Allocation: Use the assembler's .comm directive

• Use PC relative addressing
  – %rip is the current instruction address
  – X(%rip) will add the offset from the current instruction location to the space for x in the data segment to %rip
  – Creates easily recolatable binaries

```
.section .text
.globl main
main:
enter $0, $0
movq $5, x(%rip)
push x(%rip)
call printf_035
add $16, %rsp
leave
ret

.comm x, 8
```

`.comm name, size, alignment`

The .comm directive allocates storage in the data section. The storage is referenced by the identifier name. Size is measured in bytes and must be a positive integer. Name cannot be predefined. Alignment is optional. If alignment is specified, the address of name is aligned to a multiple of alignment.
Outline

• Introduction
• Machine Language
• Overview of a modern processor
• Memory Layout
• **Procedure Abstraction**
• Procedure Linkage
• **Guidelines in Creating a Code Generator**
Procedure Abstraction

- Requires system-wide compact
  - Broad agreement on memory layout, protection, resource allocation calling sequences, & error handling
  - Must involve architecture (ISA), OS, & compiler
- Provides shared access to system-wide facilities
  - Storage management, flow of control, interrupts
  - Interface to input/output devices, protection facilities, timers, synchronization flags, counters, ...
- Establishes the need for a private context
  - Create private storage for each procedure invocation
  - Encapsulate information about control flow & data abstractions

The procedure abstraction is a social contract (Rousseau)
Procedure Abstraction

• In practical terms it leads to...
  – multiple procedures
  – library calls
  – compiled by many compilers, written in different languages, hand-written assembly

• For the project, we need to worry about
  – Parameter passing
  – Registers
  – Stack
  – Calling convention
Parameter passing disciplines

- Many different methods
  - call by reference
  - call by value
  - call by value-result (copy-in/copy-out)
Parameter Passing Disciplines

Program {
    int A;
    foo(int B) {
        B = B + 1
        B = B + A
    }
    Main() {
        A = 10;
        foo(A);
    }
}

• Call by value        A is ???
• Call by reference    A is ???
• Call by value-result A is ???
Parameter Passing Disciplines

Program {
    int A;
    foo(int B) {
        B = B + 1
        B = B + A
    }
    Main() {
        A = 10;
        foo(A);
    }
}

- Call by value   \( A \) is 10
- Call by reference \( A \) is 22
- Call by value-result \( A \) is 21
Parameter passing disciplines

- Many different methods
  - call by reference
  - call by value
  - call by value-result
- How do you pass the parameters?
  - via. the stack
  - via. the registers
  - or a combination
- In the Decaf calling convention, all parameters are passed via the stack
Registers

• What to do with live registers across a procedure call?
  – Caller Saved
  – Calliee Saved
Question:

• What are the advantages/disadvantages of:
  – Calliee saving of registers?
  – Caller saving of registers?

• What registers should be used at the caller and calliee if half is caller-saved and the other half is calliee-saved?
Registers

• What to do with live registers across a procedure call?
  – Caller Saved
  – Calliee Saved

• In this segment, use registers only as short-lived temporaries
  mov  -4(%rbp), %r10
  mov  -8(%rbp), %r11
  add  %r10, %r11
  mov  %r11, -8(%rbp)
  – Should not be live across procedure calls
  – Will start keeping data in the registers for performance in Segment V
The Stack

- Arguments 0 to 6 are in:
  - %rdi, %rsi, %rdx, %rcx, %r8 and %r9

- %rbp
  - 0(%rbp)
  - 8(%rbp)
  - Previous %rbp
  - -8(%rbp)
  - -8*m-8(%rbp)

- Current
  - 16(%rbp)
  - 8(%rbp)
  - Previous %rbp
  - -8(%rbp)
  - -8*m-8(%rbp)

- Variable size

- Argument n
  - ... argument 7
  - Return address
  - Previous %rbp
  - local 0
  - Variable size
  - ... local m

Saman Amarasinghe
Question:

- Why use a stack? Why not use the heap or pre-allocated in the data segment?
Outline

• Introduction
• Machine Language
• Overview of a modern processor
• Memory Layout
• Procedure Abstraction
• Procedure Linkage
• Guidelines in Creating a Code Generator
Procedure Linkages

Standard procedure linkage

Procedure has
- standard prolog
- standard epilog

Each call involves a
- pre-call sequence
- post-return sequence
Calling: Caller
- Assume %rcx is live and is caller save
- Call foo(A, B, C, D, E, F, G, H, I)
  • A to I are at -8(%rbp) to -72(%rbp)

```
push %rcx
push -72(%rbp)
push -64(%rbp)
push -56(%rbp)
mov   -48(%rbp), %r9
mov   -40(%rbp), %r8
mov   -32(%rbp), %rcx
mov   -24(%rbp), %rdx
mov   -16(%rbp), %rsi
mov   -8(%rbp), %rdi
call  foo
```
Stack

- **Calling: Calliee**
  - Assume %rbx is used in the function and is calliee save
  - Assume 40 bytes are required for locals

```assembly
   foo:
   push  %rbp
   mov   %rsp, %rbp
   sub    $48, %rsp
   mov    %rbx, -8(%rbp)
   enter  $48, $0
```

©MIT Fall 2001
Stack

- Arguments
- Call foo(A, B, C, D, E, F, G, H, I)
  - Passed in by pushing before the call
    ```
    push -72(%rbp)
    push -64(%rbp)
    push -56(%rbp)
    mov -48(%rbp), %r9
    mov -40(%rbp), %r8
    mov -32(%rbp), %rcx
    mov -24(%rbp), %rdx
    mov -16(%rbp), %rsi
    mov -8(%rbp), %rdi
    call foo
    ```
  - Access A to F via registers
    - or put them in local memory
  - Access rest using 16+xx(%rbp)
    ```
    mov 16(%rbp), %rax
    mov 24(%rbp), %r10
    ```
Stack

- Locals and Temporaries
  - Calculate the size and allocate space on the stack
    
    sub $48, %rsp
    or  enter $48, 0

  - Access using -8-xx(%rbp)
    
    mov  -28(%rbp), %r10
    mov  %r11, -20(%rbp)
• Returning Calliee
  - Assume the return value is the first temporary
  - Restore the caller saved register
  - Put the return value in %rax
  - Tear-down the call stack

```assembly
mov       -8(%rbp), %rbx
mov       -16(%rbp), %rax
mov       %rbp, %rsp
pop       %rbp
leave
ret
```
Stack

- Returning Caller
  - Assume the return value goes to the first temporary
  - Restore the stack to reclaim the argument space
  - Restore the caller save registers

```c
call foo
add $24, %rsp
pop %rcx
mov %rax, 8(%rbp)
...`
```
Question:

• Do you need the $rbp? 
• What are the advantages and disadvantages of having $rbp?
Outline

• Introduction
• Machine Language
• Overview of a modern processor
• Memory Layout
• Procedure Abstraction
• Procedure Linkage
• Guidelines in Creating a Code Generator
What We Covered Today..

**CODE**
- Procedures
- Control Flow
- Statements
- Data Access

**DATA**
- Global Static Variables
- Global Dynamic Data
- Local Variables
- Temporaries
- Parameter Passing
- Read-only Data
Guidelines for the code generator

• Lower the abstraction level slowly
  – Do many passes, that do few things (or one thing)
    • Easier to break the project down, generate and debug

• Keep the abstraction level consistent
  – IR should have ‘correct’ semantics at all time
    • At least you should know the semantics
  – You may want to run some of the optimizations between the passes.

• Use assertions liberally
  – Use an assertion to check your assumption
Guidelines for the code generator

• Do the simplest but dumb thing
  – it is ok to generate $0 + 1*x + 0*y$
  – Code is painful to look at, but will help optimizations

• Make sure you know want can be done at…
  – Compile time in the compiler
  – Runtime using generated code
Guidelines for the code generator

• Remember that optimizations will come later
  – Let the optimizer do the optimizations
  – Think about what optimizer will need and structure your code accordingly
  – Example: Register allocation, algebraic simplification, constant propagation

• Setup a good testing infrastructure
  – regression tests
    • If a input program creates a bug, use it as a regression test
  – Learn good bug hunting procedures
    • Example: binary search