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 →
Intermediate Representation →
Code Generator →
Optimized Intermediate Representation →
Assembly code
**Anatomy of a compiler**

1. **Program (character stream)**
2. **Lexical Analyzer (Scanner)**
3. **Token Stream**
4. **Syntax Analyzer (Parser)**
5. **Parse Tree**
6. **Semantic Analyzer**
   - **High-level IR**
   - **Low-level IR**
   - **Intermediate Representation**
7. **Code Generator**
8. **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...

<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><code>movl -4(%rbp), %eax</code></td>
</tr>
<tr>
<td>0049</td>
<td>4863F0</td>
<td></td>
</tr>
</tbody>
</table>
| 004c     | 8B45FC | `movslq %eax, %rsi`  
|          |       | `-4(%rbp), %eax`       |
| 004f     | 4863D0 |                           |
| 0052     | 8B45FC | `movl -4(%rbp), %eax`  
|          |       | `movl -4(%rbp), %eax`  |
| 0055     | 4898  |                           |
| 0057     | 8B048500 | `movl B(,%rax,4), %eax` |
|          | 000000 |                                   |
| 005e     | 8B149500 | `movl A(,%rdx,4), %edx` |
|          | 000000 |                                   |
| 0065     | 01C2  | `addl %eax, %edx` |
| 0067     | 8B45FC | `addl -4(%rbp), %eax` |
| 006a     | 4898  |                           |
| 006c     | 89D7  | `addl %eax, %edi` |
| 006e     | 033C8500 | `addl C(,%rax,4), %edi` |
|          | 000000 |                                   |
| 0075     | 8B45FC | `addl -4(%rbp), %eax` |
| 0078     | 4863C8 |                           |
| 007b     | 8B45F8 | `movl %eax, %rcx`  
|          |       | `movl 8(%rbp), %eax` |
| 007e     | 4898  |                           |
| 0080     | 8B148500 | `addl B(,%rax,4), %edx` |

**Notes:**
- `movl` and `movslq` are machine instructions.
- `%eax`, `%rsi`, `%rdx` are registers.
-`B(,%rax,4), %eax`, `A(,%rdx,4), %edx`, `C(,%rax,4), %edi` are operations involving registers and memory locations.
Program (character stream)

Lexical Analyzer (Scanner)

Token Stream

Syntax Analyzer (Parser)

Parse Tree

Intermediate Code Generator

High-level IR
Low-level IR
Intermediate Representation

Code Generator

Assembly code
Program (character stream) → Lexical Analyzer (Scanner) → Token Stream → Syntax Analyzer (Parser) → Parse Tree → Intermediate Code Generator → High-level IR, Low-level IR → 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

```
section .rodata
.LC0:
0000 6572726F7200 .string "error"
.text
.globl fact
fact:
0000 55          pushq  %rbp
0001 4889E5      %rsp, %rbp
0004 4883EC10    $16, %rsp
0008 897DFC      %edi, -4(%rbp)
000b 837DFC00
000f 7911
0011 BF00000000 movl $0,-16(%rbp)
0016 B800000000
001b E800000000
0020 EB22
0022 837DFC00
0026 7509
0028 C745F801000000 jne .L3 .L4
002f EB13
0031 8B7DFC
0034 FFCF
0036 E800000000 movl l1 fact
003b 0FAF45FC
003f 8945F8
0042 EB00
0044 8B45F8
0047 C9
0048 C3
```
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

```assembly
.file "test2ioM"
.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 .long .LECIE1-.LSCIE1
.long 0x0h_frame,"a",@progbits
.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:
  \[ \text{OP } \langle \text{oprnd}_1 \rangle, \langle \text{oprnd}_2 \rangle \]
  - \[ \langle \text{oprnd}_2 \rangle = \langle \text{oprnd}_1 \rangle \text{ OP } \langle \text{oprnd}_2 \rangle \]
  Or
  \[ \text{OP } \langle \text{oprnd}_1 \rangle \]
- 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
-Globals/Read-only data
-Program
- Heap
- Unmapped

0x800 0000 0000
0x40 0000
0x0

Memory
Registers
ALU
Control
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

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

- DATA
  - 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, the first 6 parameters are passed in registers.
  – The rest are passed in 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

  \[ 8n + 16(\%rbp) \]

  \[ 16(\%rbp) \]

  \[ 8(\%rbp) \]

  \[ 0(\%rbp) \]

  \[ -8(\%rbp) \]

  \[ -8m - 8(\%rbp) \]

  \[ 0(\%rsp) \]

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

- Calling: Caller
  - Assume %rcx is live and is caller save
  - Call foo(A, B, C, D, E, F, G, H, I)
    - A to l 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

```plaintext
foo:

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

- 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
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

```asm
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\textsubscript{cons}istent
  – 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
6.035 Computer Language Engineering
Spring 2010

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.