Compiler Development (CMPSC 401)
Code Generation, ARM, x86

Janyl Jumadinova

April 11, 2019
Review

What we did last time

http://www.bravegnu.org/gnu-eprog/hello-arm.html
http://clang.llvm.org/get_started.html
LLVM is a compiler infrastructure designed as a set of reusable libraries with well-defined interfaces.

- Implemented in C++
- Several front-ends
- Several back-ends
- Lots of tools to compile and optimize code
- Open source
- http://llvm.org
Build a set of modular compiler components:
- Reduces the time and cost to construct a particular compiler
- Components are shared across different compilers
- Allows choice of the right component for the job

Build compilers out of these components
The front-end that parses C into bytecodes

Machine independent optimizations, such as constant propagation

Machine dependent optimizations, such as register allocation

clang

file.c

LLVM

opt

file.bc

file.bc

llc

file.s
LLVM

- LLVM represents programs, internally, via its own instruction set.
- Bytecode is a form of instruction set designed for efficient execution by a software interpreter.
- The tool lli directly executes programs in LLVM bitcode format.

https://clang.llvm.org/docs/ClangCommandLineReference.html#actions
https://llvm.org/docs/CommandGuide/llc.html

clang -c -emit-llvm example.c -o example.bc
lli example.bc
LLVM IR

- RISC instruction set, with usual opcodes
  - add, mul, or, shiR, branch, load, store, etc.
- Typed representation.
- Static Single Assignment format
  - Each variable noun has only one definition in the program code.
- Can program directly on the IR.
Generating Machine Code

- Once the intermediate program is optimized, it can be translated to machine code.
- In LLVM, use the `llc` tool to perform this translation. This tool is able to target many different architectures.

`llc --version`
Generating Machine Code

with conversion to bytecode and optimization (to x86)

```
clang -c -emit-llvm example.c -o example.bc
opt -mem2reg example.bc -o example.opt.bc
llc -march=x86 example.opt.bc -o example.x86
```

or, without converting to bytecode and without optimization (to ARM)

```
clang -S -emit-llvm example.c -o example.ll
llc -march=aarch64 example.ll -o example.S
```
LLVM Summary

LLVM implements the entire compilation flow
- Frontend, e.g., clang and clang++
- Middleend, e.g., analyses and optimizations
- Backend, e.g., different computer architectures

LLVM has a highlevel intermediate representation (types, explicit control flow)

```
C → clang → file.bc → llc → x86
```

```
opt
```

Janyl Jumadinova
Compiler Development (CMPSC 401)
April 11, 2019
ARM
Given an expression tree and a machine architecture, generate a set of instructions that evaluate the tree

- Initially, consider only trees (no common subexpressions)
- Interested in the quality of the program
- Interested in the running time of the algorithm
Basic Code Generation Strategy

- Walk the IR (for us, a syntax tree or an AST), outputting code for each construct encountered
- Handling of node’s children is dependent on type of node
Walk the IR (for us, a syntax tree or an AST), outputting code for each construct encountered

Handling of node’s children is dependent on type of node
E.g., for binary operation like +:
  - Generate code to compute operand 1 (and store result)
  - Generate code to compute operand 2 (and store result)
  - Generate code to load operand results and add them together
The following slides will walk through how this is done for many common language constructs.

- Examples show code snippets in isolation
  - Much the way we’ll generate code for different parts of the AST in our compiler

- Register eax used as a generic example
  - Rename as needed for more complex code using multiple registers
Code Generation for Constants

- Source: 17
- ARM: mov eax #17
Assignment Statement

- Source: `var = exp;`
- ARM: `mov eax exp`
Source: exp1 + exp2

ARM:
<code evaluating exp1 into eax>
<code evaluating exp2 into ebx>
add eax, ebx
If exp2 is a simple variable or constant, don’t need to load it into another register first. Instead:
add eax, exp2 ;
Change exp1 + (−exp2) into exp1 − exp2
**Binary Minus**

- **Source**: \( \text{exp1} - \text{exp2} \)
- **ARM**: Similar to Plus
  
  Use `SUB`
Binary Multiplication

- **Source**: \( \text{exp1} \times \text{exp2} \)
- **ARM**: Similar to Plus and Minus
  Use **MUL**
Integer Division

- Source: \( \text{exp1} / \text{exp2} \)
- ARM: We can use mov and shifting LSR and ASR for this
While

Source: while (cond) stmt

ARM:

start_while:
<code evaluating cond>
    Branch_Cond end_while
<code for stmt>
    Branch start_while

end_while:
Do While

- Source: `do stmt while(cond)`
- ARM:
  ```
  loop:
    <code for stmt>
  <code evaluating cond>
  Branch_Cond loop
  ```
If

- **Source**: if (cond) stmt
- **ARM**:
  
  <code evaluating cond>
  Branch_Cond  skip
  <code for stmt>

  skip:
If-Else

- Source: `if (cond) stmt1 else stmt2`
- ARM:
  
  ```
  <code evaluating cond>
  Branch_Cond else
    <code for stmt1>
    Branch jump
  else: <code for stmt2>
  jump:
  ```
Code for exp1 > exp2

- Generated code depends on context
- What is the branch target?
- Branch if the condition is true or if false?

```c
Example: evaluate exp1 > exp2, branch on false
<evaluate exp1 to eax
<evaluate exp2 to edx
cmp eax,edx
BGT ...
Code for $\text{exp1} > \text{exp2}$

- Generated code depends on context
- What is the branch target?
- Branch if the condition is true or if false?
- Example: evaluate $\text{exp1} > \text{exp2}$, branch on false
  
  <evaluate $\text{exp1}$ to eax
  <evaluate $\text{exp2}$ to edx
  cmp eax,edx
  BGT ...
Code for exp1 == exp2

- Evaluate exp1 == exp2, branch on false
  - evaluate exp1 to eax
  - evaluate exp2 to edx
  - CMP eax, edx
  - BNE ...
Parameter Passing

- Parameters passed using registers/stack
- Parameter passing does not need to be done using only registers or only stack
- Some inputs could come in registers and some on stack and outputs could be returned in registers and some on the stack
Parameter Passing

- Parameters passed using registers/stack
- Parameter passing does not need to be done using only registers or only stack
- Some inputs could come in registers and some on stack and outputs could be returned in registers and some on the stack

**Calling Convention**
- Application Binary Interface (ABI)
- ARM: use registers for first 4 parameters, use stack beyond, return using R0
Parameter Passing

- ABI standard for all ARM architectures
- Use registers R0, R1, R2, and R3 to pass the first four input parameters (in order) into any function, C or assembly.
- We place the return parameter in Register R0.
- Functions can freely modify registers R0-R3 and R12.
Parameter Passing

- ABI standard for all ARM architectures
- Use registers R0, R1, R2, and R3 to pass the first four input parameters (in order) into any function, C or assembly.
- We place the return parameter in Register R0.
- Functions can freely modify registers R0-R3 and R12.
- If a function needs to use R4 through R11, it is necessary to:
  - to push the current register value onto the stack,
  - use the register, and
  - then pop the old value off the stack before returning.
Focus on the Pentium instruction set

- x86 is the dominant chip in today’s computers (Mac, Windows, Linux)
- over 100 million chips sold per year
- over $5 billion annual development budget
Intel x86 Architecture

- 8, 16, or 32-bit architecture
- 2-address instruction set
- CISC (not RISC, load/store)
- uses condition codes for control instructions
Intel x86 Architecture

CISC: Complex Instruction Set Computer

- instructions of different complexity and length (1-15 bytes)
- some very complex: vector operations on floats
- complexities increasingly addressed with more hardware (Xeon E7 processors have 2.6 billion transistors)
Intel x86 Architecture

- 4 general purpose registers: AX, BX, CX, DX
- Stack pointer: SP
- Base pointer: BP
- Address registers: SI, DI
- 8 bit registers: AH/AL, CH/CL, DH/DL, BH/BL
- 32 bit registers: prefix with “E”, e.g., EAX
- 64 bit registers: prefix with “R”, e.g., RAX8
- Additional registers added (R8-R15)
- Additional floating point registers: ST(0)-ST(7)
add EAX, EBX ; add two registers
add EAX, 42  ; add value 42 to register value
add EAX, [ff02] ; add value from memory location ff02 to register
add [ff02], EAX ; as above, store result in memory
add [ff02], 20 ; add 20 to value stored in memory location ff02
Addressing Modes

mov [ff02], EAX ; load from address ff02
mov [ESP], EAX ; load from address specified in register ESP
mov [ESP+40], EAX ; address is register value + 40
mov [ESP+EBX], EAX ; address is sum of register values
## Data Types

<table>
<thead>
<tr>
<th>C</th>
<th>Intel type</th>
<th>Assembly suffix</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>byte</td>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>word</td>
<td>w</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>double word</td>
<td>l</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>quad word</td>
<td>q</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>single precision</td>
<td>s</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>double precision</td>
<td>d</td>
<td>8</td>
</tr>
</tbody>
</table>