Segment IV Roadmap

- Checkpoint
  - On Tuesday 10/22
  - Hand-in a tarball of what you have
  - If you get codegen to work, no effect
  - If you have problems at end, we will be very harsh if you haven’t done much work by the checkpoint
- Due on 10/31

- Paper discussion
  - Prof: Amarasinghe  Next Monday (17th)
  - Prof: Rinard  Next Friday (21st)

---

**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**

<table>
<thead>
<tr>
<th>CODE</th>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Procedures</td>
<td>Global Static Variables</td>
</tr>
<tr>
<td>Control Flow</td>
<td>Global Dynamic Data</td>
</tr>
<tr>
<td>Statements</td>
<td>Local Variables</td>
</tr>
<tr>
<td>Data Access</td>
<td>Temporaries</td>
</tr>
<tr>
<td>Parameter Passing</td>
<td>Read-only Data</td>
</tr>
</tbody>
</table>

**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

**Machines understand...**

<table>
<thead>
<tr>
<th>LOCATION</th>
<th>DATA</th>
<th>ASSEMBLY INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0044</td>
<td>064120</td>
<td>movl -4(%rbp), %eax</td>
</tr>
<tr>
<td>0045</td>
<td>064120</td>
<td>movl -4(%rbp), %eax</td>
</tr>
<tr>
<td>0046</td>
<td>064120</td>
<td>movl -4(%rbp), %eax</td>
</tr>
<tr>
<td>0047</td>
<td>064120</td>
<td>movl -4(%rbp), %eax</td>
</tr>
<tr>
<td>0048</td>
<td>064120</td>
<td>movl -4(%rbp), %eax</td>
</tr>
<tr>
<td>0049</td>
<td>064120</td>
<td>movl -4(%rbp), %eax</td>
</tr>
<tr>
<td>004A</td>
<td>064120</td>
<td>movl -4(%rbp), %eax</td>
</tr>
<tr>
<td>004B</td>
<td>064120</td>
<td>movl -4(%rbp), %eax</td>
</tr>
<tr>
<td>004C</td>
<td>064120</td>
<td>movl -4(%rbp), %eax</td>
</tr>
<tr>
<td>004D</td>
<td>064120</td>
<td>movl -4(%rbp), %eax</td>
</tr>
<tr>
<td>004E</td>
<td>064120</td>
<td>movl -4(%rbp), %eax</td>
</tr>
<tr>
<td>004F</td>
<td>064120</td>
<td>movl -4(%rbp), %eax</td>
</tr>
<tr>
<td>0050</td>
<td>064120</td>
<td>movl -4(%rbp), %eax</td>
</tr>
<tr>
<td>0051</td>
<td>064120</td>
<td>movl -4(%rbp), %eax</td>
</tr>
<tr>
<td>0052</td>
<td>064120</td>
<td>movl -4(%rbp), %eax</td>
</tr>
<tr>
<td>0053</td>
<td>064120</td>
<td>movl -4(%rbp), %eax</td>
</tr>
<tr>
<td>0054</td>
<td>064120</td>
<td>movl -4(%rbp), %eax</td>
</tr>
<tr>
<td>0055</td>
<td>064120</td>
<td>movl -4(%rbp), %eax</td>
</tr>
<tr>
<td>0056</td>
<td>064120</td>
<td>movl -4(%rbp), %eax</td>
</tr>
<tr>
<td>0057</td>
<td>064120</td>
<td>movl -4(%rbp), %eax</td>
</tr>
<tr>
<td>0058</td>
<td>064120</td>
<td>movl -4(%rbp), %eax</td>
</tr>
<tr>
<td>0059</td>
<td>064120</td>
<td>movl -4(%rbp), %eax</td>
</tr>
<tr>
<td>005A</td>
<td>064120</td>
<td>movl -4(%rbp), %eax</td>
</tr>
<tr>
<td>005B</td>
<td>064120</td>
<td>movl -4(%rbp), %eax</td>
</tr>
<tr>
<td>005C</td>
<td>064120</td>
<td>movl -4(%rbp), %eax</td>
</tr>
<tr>
<td>005D</td>
<td>064120</td>
<td>movl -4(%rbp), %eax</td>
</tr>
<tr>
<td>005E</td>
<td>064120</td>
<td>movl -4(%rbp), %eax</td>
</tr>
<tr>
<td>005F</td>
<td>064120</td>
<td>movl -4(%rbp), %eax</td>
</tr>
</tbody>
</table>
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 example

```
.globl fact
fact:
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl -8(%rbp), %eax
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

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)

- 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
- 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
  - Jx $L32 Jx 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 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 Layout**

- Heap management
  - free lists
- starting location in the text segment

**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

**Registers**

- Instructions allow only limited memory operations
  - add -4(%rbp), -8(%rbp)
  - mov -4(%rbp), %r10
  - add %r10, -8(%rbp)
- Important for performance
  - limited in number
- Special registers
  - %rbp base pointer
  - %rsp stack pointer
Allocating Read-Only Data

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

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 relocatable
    binaries

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
  - 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)

Parameter passing disciplines

- Many different methods
  - call by reference
  - call by value
  - call by value-result

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

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

---

**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?

---

**The Stack**

<table>
<thead>
<tr>
<th>Previous %rbp</th>
<th>Previous address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0(%rbp)</td>
<td>8*n+16(%rbp)</td>
</tr>
<tr>
<td>0(%rbp)</td>
<td>16(%rbp)</td>
</tr>
<tr>
<td>0(%rbp)</td>
<td>8(%rbp)</td>
</tr>
<tr>
<td>0(%rbp)</td>
<td>%r11</td>
</tr>
<tr>
<td>%r10</td>
<td>%r11, -8(%rbp)</td>
</tr>
<tr>
<td>-8(%rbp)</td>
<td>mov -4(%rbp), %r10</td>
</tr>
<tr>
<td>-8(%rbp)</td>
<td>add %r10, %r11</td>
</tr>
</tbody>
</table>

**Question:**

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

---

**Registers**

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

---

**The Stack**

<table>
<thead>
<tr>
<th>Previous %rbp</th>
<th>Previous address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0(%rbp)</td>
<td>8*n+16(%rbp)</td>
</tr>
<tr>
<td>0(%rbp)</td>
<td>16(%rbp)</td>
</tr>
<tr>
<td>0(%rbp)</td>
<td>8(%rbp)</td>
</tr>
<tr>
<td>0(%rbp)</td>
<td>%r11</td>
</tr>
<tr>
<td>%r10</td>
<td>%r11, -8(%rbp)</td>
</tr>
<tr>
<td>-8(%rbp)</td>
<td>mov -4(%rbp), %r10</td>
</tr>
<tr>
<td>-8(%rbp)</td>
<td>add %r10, %r11</td>
</tr>
</tbody>
</table>

**Question:**

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

---

**The Stack**

<table>
<thead>
<tr>
<th>Previous %rbp</th>
<th>Previous address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0(%rbp)</td>
<td>8*n+16(%rbp)</td>
</tr>
<tr>
<td>0(%rbp)</td>
<td>16(%rbp)</td>
</tr>
<tr>
<td>0(%rbp)</td>
<td>8(%rbp)</td>
</tr>
<tr>
<td>0(%rbp)</td>
<td>%r11</td>
</tr>
<tr>
<td>%r10</td>
<td>%r11, -8(%rbp)</td>
</tr>
<tr>
<td>-8(%rbp)</td>
<td>mov -4(%rbp), %r10</td>
</tr>
<tr>
<td>-8(%rbp)</td>
<td>add %r10, %r11</td>
</tr>
</tbody>
</table>

**Question:**

- Why use a stack? Why not use the heap or pre-allocated in the data segment?
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)
    - A is at -8(%rbp)
    - B is at -16(%rbp)
    - C is at -24(%rbp)
  - Arguments
    - Call foo(A, B, C)
      - Passed in by pushing before the call
        - push -24(%rbp)
        - push -16(%rbp)
        - push -8(%rbp)
        - call foo
      - Accessed using 16+xx %rbp
        - mov 16(%rbp), %rax
        - mov 24(%rbp), %r10

- Locals and Temporaries
  - Calculate the size and allocate space on the stack
    - sub $48, %rsp
    - or enter %rsp
    - push -8(%rbp), %rbx
    - mov -28(%rbx), %r10
    - mov %r11, -20(%rbx)

- 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
    - mov -8(%rbp), %rbx
    - mov -16(%rbp), %rax
    - 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

```assembly
call foo
add $24, %rbp
pop %rcx
mov %rax, 8(%rbp)
```

Example Program

```c
program {
  int sum3d(int ax, int ay, int az)
  {
    int dx, dy, dz;
    if(ax > ay)
      dx = ax - bx;
    else
      dx = bx - ax;
    return dx + dy + dz;
  }
  main()
  { int px, py, pz;
    px = 10; py = 20; pz = 30;
    sum3d(px, py, pz);
  }
}
```

Question:

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

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

- 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

- 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