6.035

Fall 2002

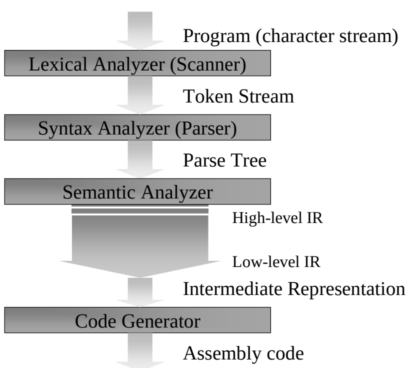
Lecture 9: Unoptimized Code Generation

From the intermediate representation to the machine code

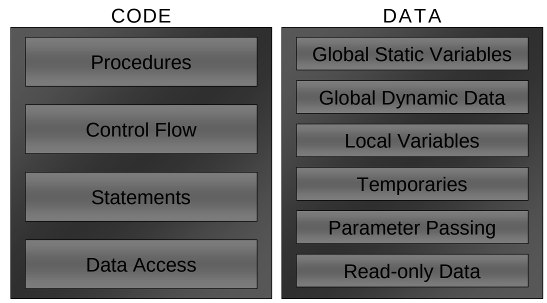
Segment IV Roadmap

- 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



Components of a High Level Language

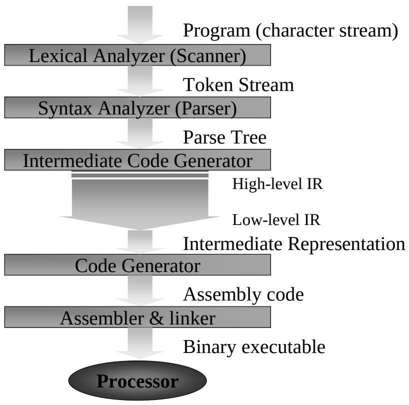


Machine Code Generator Should...

- 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...

LOCATION	DATA	ASSEMBLY INSTRUCTION
0046	8B45FC	movl -4(%rbp), %eax
0049	4863F0	movslq %eax,%rsi
004c	8B45FC	movl -4(%rbp), %eax
004f	4863D0	movslq %eax,%rdx
0052	8B45FC	movl -4(%rbp), %eax
0055	4898	cltq
0057	8B048500 000000	movl B(,%rax,4), %eax
005e	8B149500 000000	movl A(,%rdx,4), %edx
0065	01C2	addl %eax, %edx
0067	8B45FC	movl -4(%rbp), %eax
006a	4898	cltq
006c	89D7	movl %edx, %edi
006e	033C8500 000000	addl C(,%rax,4), %edi
0075	8B45FC	movl -4(%rbp), %eax
0078	4863C8	movslq %eax,%rcx
007b	8B45F8	movl -8(%rbp), %eax
007e	4898	cltq
0080	8B148500	movl B(,%rax,4), %edx



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
    $\Rightarrow$ 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                     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 837DFC00                   cmpl    $0, -4(%rbp)
0026 7509                       jne     .L4
0028 C745F801000000             movl    $1, -8(%rbp)
002f EB13                       jmp     .L3
                 .L4:
0031 8B7DFC                     movl    -4(%rbp), %edi
0034 FFCF                       decl    %edi
0036 E800000000                 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
```

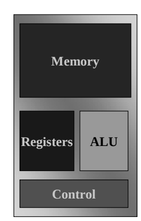

Composition of an Object File

- 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

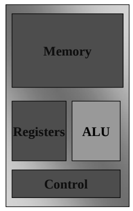
```
 .file   "test2.c"
        .section        .rodata
.LC0:
        .string "error %d"

        .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
```


Overview of a modern processor

- ALU
- Control
- Memory
- Registers



Arithmetic and Logic Unit

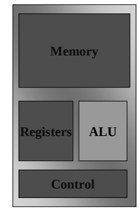


- Performs most of the data operations
- Has the form:
  OP <oprnd$_1$>, <oprnd$_2$>
  - <oprnd$_2$> = <oprnd$_1$> OP <oprnd$_2$>

  Or

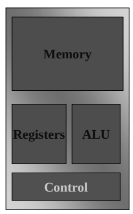
  OP <oprnd$_1$>
- Operands are:
  - Immediate Value $25
  - Register %rax
  - Memory 4(%rbp)
- Operations are:
  - Arithmetic operations (add, sub, imul)
  - Logical operations (and, sal)
  - Unitary operations (inc, dec)

Arithmetic and Logic Unit



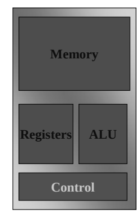
- Many arithmetic operations can cause an exception
  - overflow and underflow
- Can operate on different data types
  - addb 8 bits
  - addw 16 bits
  - addl 32 bits
  - addq 64 bits (Decaf is all 64 bit)
  - signed and unsigned arithmetic
  - Floating-point operations (separate ALU)

Control



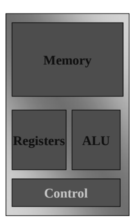
- Handles the instruction sequencing
- Executing instructions
  - All instructions are in memory
  - Fetch the instruction pointed by the PC and execute it
  - For general instructions, increment the PC to point to the next location in memory

Control



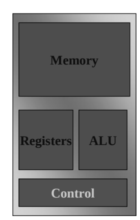
- Unconditional Branches
  - Fetch the next instruction from a different location
  - Unconditional jump to an address
    jmp .L32
  - Unconditional jump to an address in a register
    jmp %rax
  - To handle procedure calls
    call fact call %r11

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

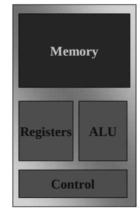
18

When to use what?

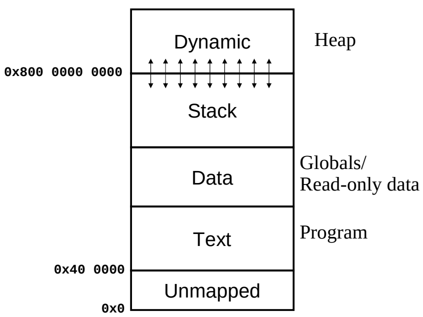
- Give an example where each of the branch instructions can be used
  1. jmp L0
  2. call L1
  3. jmp %rax
  4. jz -4(%rbp)
  5. jne L1

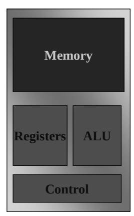
Memory

- Flat Address Space
  - composed of words
  - byte addressable
- Need to store
  - Program
  - Local variables
  - Global variables and data
  - Stack
  - Heap



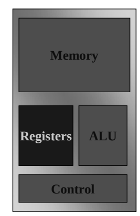
Memory





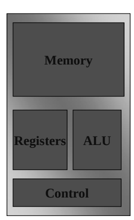
Registers

- 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



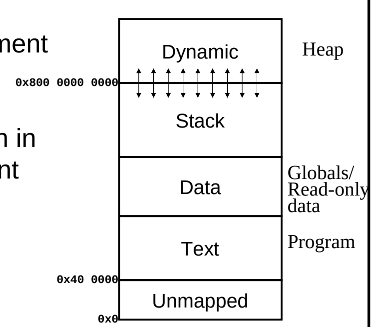
Other interactions

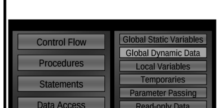
- Other operations
  - Input/Output
  - Privilege / secure operations
  - Handling special hardware
    - TLBs, Caches etc.
- Mostly via system calls
  - hand-coded in assembly
  - compiler can treat them as a normal function call



Memory Layout

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

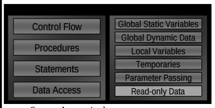




Allocating Read-Only Data

- 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"
```



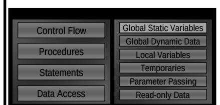
Global Variables

- 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*



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

25

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

- 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

30

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?

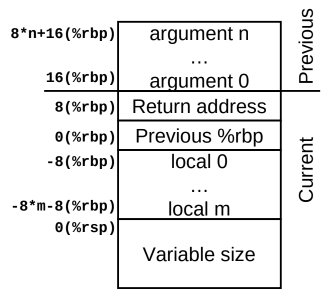
Registers

- 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



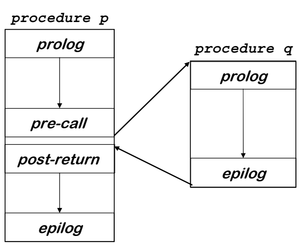
33

Question:

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

Procedure Linkages

Standard procedure linkage



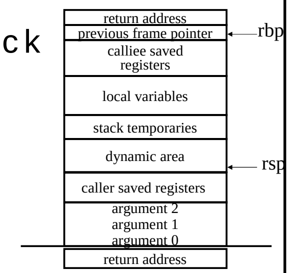
Procedure has
- **standard prolog**
- **standard epilog**

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)

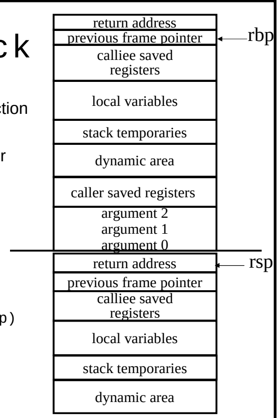
```
push    %rcx
push    -24(%rbp)
push    -16(%rbp)
push    -8(%rbp)
call    foo
```



Stack

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

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



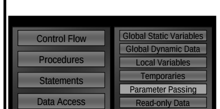
Stack

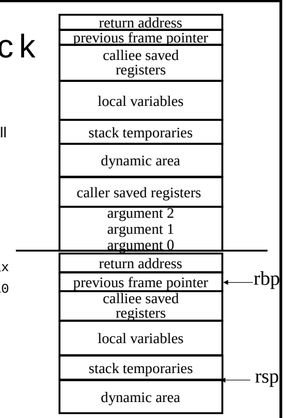
- Arguments
- Call foo(A, B, C)
  - Passed in by pushing before the call

```
push    -24(%rbp)
push    -16(%rbp)
push    -8(%rbp)
call    foo
```

  - Accessed using 16+xx(%rbp)

```
mov     16(%rbp), %rax
mov     24(%rbp), %r10
```





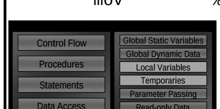
Stack

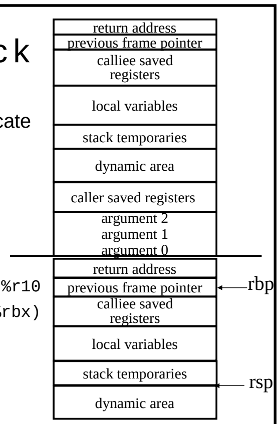
- Locals and Temporaries
  - Calculate the size and allocate space on the stack

```
    sub     $48, %rsp
or  enter   $48, 0
```

  - Access using -8-xx(%rbx)

```
    mov     -28(%rbx), %r10
    mov     %r11, -20(%rbx)
```

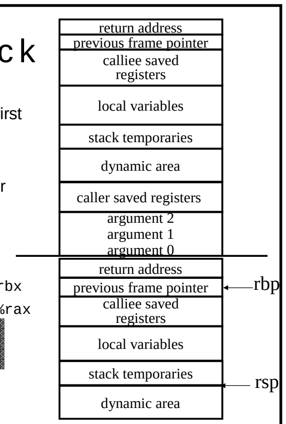




Stack

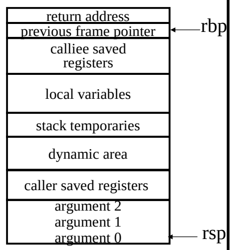
- Returning Calliee
  - Assume the return value is the first temporary
  - Restore the caller saved register
  - Put the return value in %rax
  - Tear-down the call stack

```
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



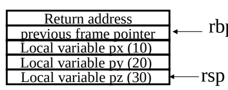

```
call        foo
add         $24, %rsp
pop         %rcx
mov         %rax, 8(%rbp)
…
```


Question:

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

Example Program

```
program {
  int sum3d(int ax, int ay, int az)
  {
     int dx, dy, dz;
     if(ax > ay)
            dx = ax - bx;
     else
            dx = bx - ax;
    …
     retrun dx + dy + dz;
  }

   main() {
     int px, py, pz;
     px = 10; py = 20; pz = 30;
     sum3d(px, py, pz);
}
```



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