

6.035

Fall 2006

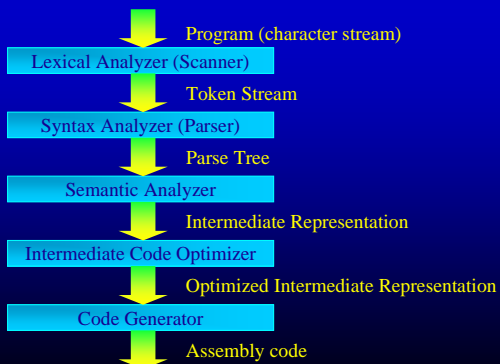
Lecture 7: Unoptimized Code Generation

From the intermediate representation to the machine code

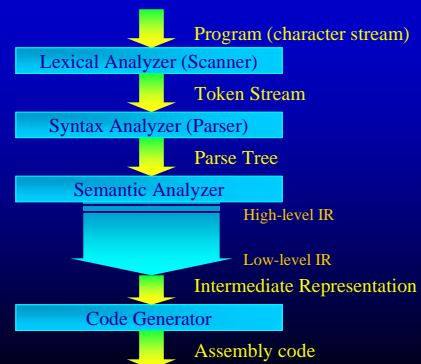
Segment IV Roadmap

- Design and Checkpoint
 - Due Thursday 10/19
 - Checkpoint
 - 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
- Implementation and Report
 - Due on 10/26

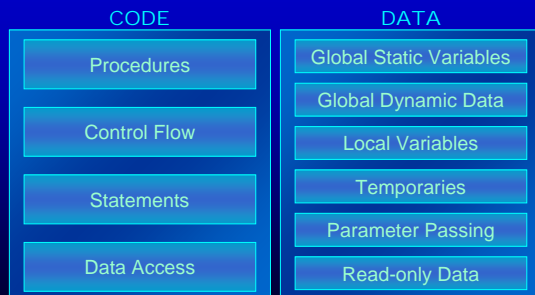
Anatomy of a compiler



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

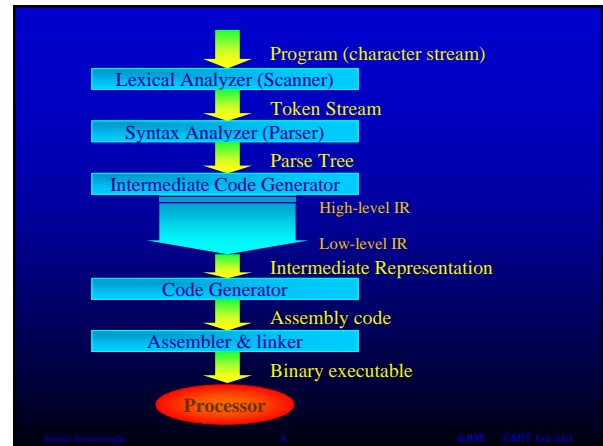
Machines understand...

LOCATION	DATA	ASSEMBLY INSTRUCTION
0046	8B45FC	movl -4(%rbp), %eax
0049	4863F0	movslq %eax,%rsi
004c	8B45FC	movl -4(%rbp), %eax
004E	4863D0	movslq %eax,%rdx
0052	8B45FC	movl -4(%rbp), %eax
0055	4998	cltq
0057	8B048500	movl B(%rax,4), %eax
	000000	
005e	8B149500	movl A(%rdx,4), %edx
	000000	
0065	01C2	addl %eax, %edx
0067	8B45FC	movl -4(%rbp), %eax
006a	4998	cltq
006e	8907	movl %edx, %edi
006e	033C8500	addl C(%rax,4), %edi
	000000	
0075	8B45FC	movl -4(%rbp), %eax
0078	4863C8	movslq %eax,%rcx
007b	8B45F8	movl -8(%rbp), %eax
007e	4998	cltq
0080	8B148500	movl B(%rax,4), %edx

Saman Aravamudan

7

6.035 ©MIT Fall 2001



Saman Aravamudan

8

6.035 ©MIT Fall 2001

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

Saman Aravamudan

9

6.035 ©MIT Fall 2001

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

Saman Aravamudan

10

6.035 ©MIT Fall 2001

Assembly example

```

.LC0:      .section      .rodata
0000 6572726f7200      .string "error"
          .text
          .globl fact
          fact:
0000 55                pushq %rbp
0001 4889e5            movq %rsp, %rbp
0004 4883e10           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 e800000000       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)
002e eb13             jmp .L3
          .L4:
0031 8b7dfc            movl -4(%rbp), %edi
0034 f7c2            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
  
```

Saman Aravamudan

11

6.035 ©MIT Fall 2001

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"
.LC0:      .section      .rodata
          .string "error %d"
          .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 .LCB1-.LCB1
          .long 0x0
          .byte 0x1
          .string ""
          .uleb128 0x1
  
```

Saman Aravamudan

12

6.035 ©MIT Fall 2001

Overview of a modern processor

- ALU
- Control
- Memory
- Registers



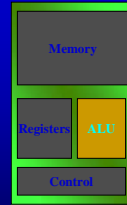
Suman Amarasinghe

13

6.035 ©MIT Fall 2001

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)



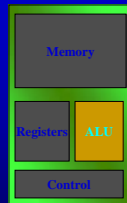
Suman Amarasinghe

14

6.035 ©MIT Fall 2001

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)



Suman Amarasinghe

15

6.035 ©MIT Fall 2001

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



Suman Amarasinghe

16

6.035 ©MIT Fall 2001

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



Suman Amarasinghe

17

6.035 ©MIT Fall 2001

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




Suman Amarasinghe

18

6.035 ©MIT Fall 2001

Control

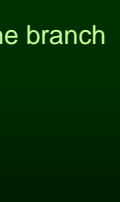
- Control transfer in special (rare) cases
 - traps and exceptions
 - Mechanism
 - Save the next(or current) instruction location
 - find the address to jump to (from an exception vector)
 - jump to that location



Suman Amarasinghe 19 6.035 ©MIT Fall 2001

When to use what?


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



Suman Amarasinghe 20 6.035 ©MIT Fall 2001

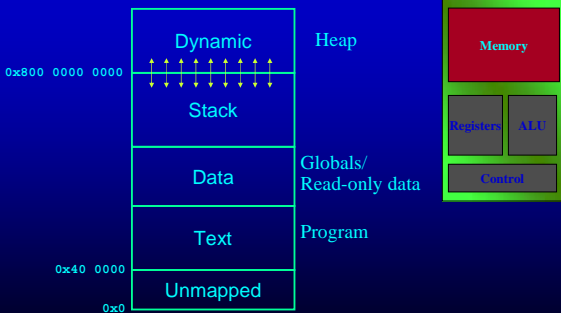

Memory

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



Suman Amarasinghe 21 6.035 ©MIT Fall 2001


Memory

Suman Amarasinghe 22 6.035 ©MIT Fall 2001

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



Suman Amarasinghe 23 6.035 ©MIT Fall 2001

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



Suman Amarasinghe 24 6.035 ©MIT Fall 2001

Memory Layout

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

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

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

Samuel Adams@mit.edu 25 6.035 ©MIT Fall 2001

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"
  
```

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

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

Samuel Adams@mit.edu 26 6.035 ©MIT Fall 2001

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

```

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

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

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

Samuel Adams@mit.edu 27 6.035 ©MIT Fall 2001

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)

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

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

Samuel Adams@mit.edu 28 6.035 ©MIT Fall 2001

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

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

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

Samuel Adams@mit.edu 29 6.035 ©MIT Fall 2001

Parameter passing disciplines

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

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

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

Samuel Adams@mit.edu 30 6.035 ©MIT Fall 2001

Parameter Passing Disciplines

25

```
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 ???

Suman Amarasinghe

31

6.035 ©MIT Fall 2001

Parameter Passing Disciplines

25

```
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 ?

Suman Amarasinghe

32

6.035 ©MIT Fall 2001

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

Suman Amarasinghe

33

6.035 ©MIT Fall 2001

Registers

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

Suman Amarasinghe

34

6.035 ©MIT Fall 2001

Question:

30

- What are the advantages/disadvantages of:
 - Callee saving of registers?
 - Caller saving of registers?
- What registers should be used at the caller and callee if half is caller-saved and the other half is callee-saved?

Suman Amarasinghe

35

6.035 ©MIT Fall 2001

Registers

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

Suman Amarasinghe

36

6.035 ©MIT Fall 2001

Homes for Variables?

- A Simplistic model
 - Allocate a data area for each distinct scope
 - One data area per "sheaf" in scoped table
- What about recursion?
 - Need a data area per invocation (or activation) of a scope
 - We call this the scope's activation record
 - The compiler can also store control information there !
- More complex scheme
 - One activation record (AR) per procedure instance
 - All the procedure's scopes share a single AR
 - Use a stack to keep the activation records

Suman Amarasinghe 37 6.035 ©MIT Fall 2001

The Stack

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

Suman Amarasinghe 38 6.035 ©MIT Fall 2001

Question:

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

Suman Amarasinghe 39 6.035 ©MIT Fall 2001

Procedure Linkages

Standard procedure linkage

- Procedure has
 - standard prolog
 - standard epilog
- Each call involves a
 - pre-call sequence
 - post-return sequence

Suman Amarasinghe 40 6.035 ©MIT Fall 2001

Stack

- 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
  
```

Suman Amarasinghe 41 6.035 ©MIT Fall 2001

Stack

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

```

foo:
push   %rbp
enter  $48, $0
sub    $18, %rbp
mov    %rbx, -8(%rbp)
  
```

Suman Amarasinghe 42 6.035 ©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)
mov     -48(%rbp), %r9
mov     -40(%rbp), %r8
mov     -32(%rbp), %r7
mov     -24(%rbp), %r6
mov     -16(%rbp), %r5
mov     -8(%rbp), %r4
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

```

CODE	
Control Flow	Global Static Variables
Procedures	Global Dynamic Data
Statements	Local Variables
Data Access	Temporaries
	Read-only Data

DATA	
Global Static Variables	Global Dynamic Data
Local Variables	Temporaries
Parameter Passing	Read-only Data

Srinivas Aravamudan 6.035 ©MIT Fall 2001 43

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)

```

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

DATA	
Global Static Variables	Global Dynamic Data
Local Variables	Temporaries
Parameter Passing	Read-only Data

Srinivas Aravamudan 6.035 ©MIT Fall 2001 44

Stack

- Returning Callee
 - 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
leave
ret

```

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

DATA	
Global Static Variables	Global Dynamic Data
Local Variables	Temporaries
Parameter Passing	Read-only Data

Srinivas Aravamudan 6.035 ©MIT Fall 2001 45

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)
...

```

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

DATA	
Global Static Variables	Global Dynamic Data
Local Variables	Temporaries
Parameter Passing	Read-only Data

Srinivas Aravamudan 6.035 ©MIT Fall 2001 46

Question:

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

Srinivas Aravamudan 6.035 ©MIT Fall 2001 47

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

Srinivas Aravamudan 6.035 ©MIT Fall 2001 48

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

Suman Amarasinghe

49

6.035 ©MIT Fall 2001

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 what can be done at...
 - Compile time in the compiler
 - Runtime using generated code

Suman Amarasinghe

50

6.035 ©MIT Fall 2001

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

Suman Amarasinghe

51

6.035 ©MIT Fall 2001