

# 6.035

Fall 2002

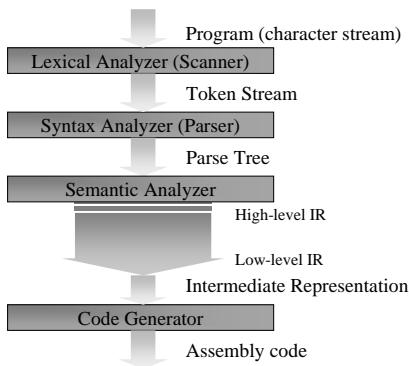
## Lecture 9: Unoptimized Code Generation

From the intermediate representation to the machine code

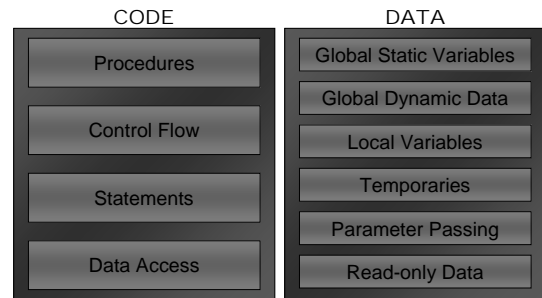
## Segment IV Roadmap

- 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 (17<sup>th</sup>)
  - Prof: Rinard                                Next Friday (21<sup>st</sup>)
  - Paper will be available on the web site

## Anatomy of a compiler



## Components of a High Level Language

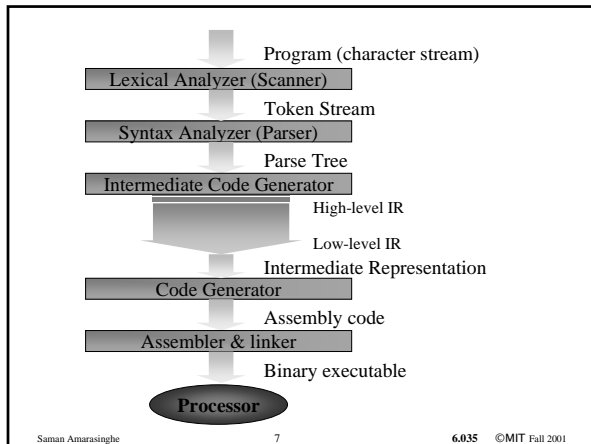


## Machine Code Generator Should...

- 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	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	4898	cltq
006c	8B07	movl %edx, %edi
006e	032C8500	addl C(%rax,4), %edi
	000000	
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
    - ⇒ can modify the implementation
      - macro assembly instructions
- Disadvantages
  - Additional process of assembling and linking
  - Assembler adds overhead

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

6.035 ©MIT Fall 2001

## Assembly example

```

    .section      .rodata
    .LC0:        .string "error"
    .text
    .globl fact
    fact:
    0000 55          pushq %rbp
    0001 4889E5     movq  %rbp, %rbp
    0004 4883BC10   subq  $16, %rbp
    0008 897DFC    movl  %edi, -4(%rbp)
    000b 837DFC00   cmpl  $0, -4(%rbp)
    000f 7911       jns   .L2
    0011 B800000000 movl  $LC0, %edi
    0016 B800000000 movl  $0, %eax
    001b E800000000 call  printf
    0020 B822       jmp   .L3
    0022 837DFC00   cmpl  $0, -4(%rbp)
    0026 7509       jne   .L4
    0028 C745F801000000 movl  $1, -8(%rbp)
    002f EB13       jmp   .L3
    0031 8B7DFC     movl  -4(%rbp), %edi
    0034 FFCF     decl  %edi
    0036 B800000000 call  fact
    003b 0FAF45FC   imull -4(%rbp), %eax
    003f 8945F8    movl  %eax, -8(%rbp)
    0042 EB00       jmp   .L1
    0044 8B45F8    movl  -8(%rbp), %eax
    0047 C9       leave
    0048 C3       ret
  
```

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"
    .section      .rodata
    .LC0:        .string "error %d"
    .section      .text
    .globl fact
    fact:
    pushq %rbp
    movq  %rbp, %rbp
    subq  $16, %rbp
    movl  -8(%rbp), %eax
    leave
    ret
    .comm bar,4,4
    .comm a,1,1
    .comm b,1,1
    .section      .eh_frame,"a",@progbits
    .long .LCIE1-.LSCIE1
    .long 0x0
    .byte 0x1
    .string ""
    .uleb128 0x1
  
```

6.035 ©MIT Fall 2001

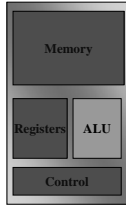
## Overview of a modern processor

- ALU
- Control
- Memory
- Registers

6.035 ©MIT Fall 2001

## Arithmetic and Logic Unit

- Performs most of the data operations
- Has the form:
  - OP <oprnd<sub>1</sub>>, <oprnd<sub>2</sub>>
  - <oprnd<sub>2</sub>> = <oprnd<sub>1</sub>> OP <oprnd<sub>2</sub>>
  - Or
  - OP <oprnd<sub>1</sub>>
- 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)



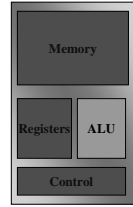
Saman Amarasinghe

13

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)



Saman Amarasinghe

14

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



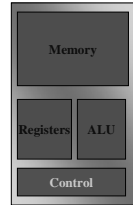
Saman Amarasinghe

15

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



Saman Amarasinghe

16

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



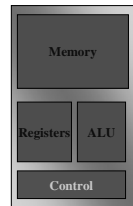
Saman Amarasinghe

17

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



Saman Amarasinghe

18

6.035 ©MIT Fall 2001

## When to use what?


18

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

Saman Amarasinghe 19 6.035 ©MIT Fall 2001

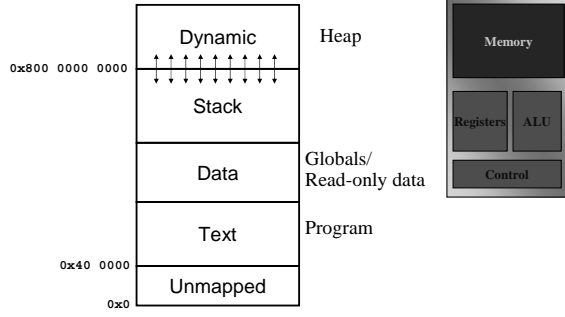
## Memory

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



Saman Amarasinghe 20 6.035 ©MIT Fall 2001

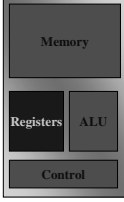
## Memory



Saman Amarasinghe 21 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



Saman Amarasinghe 22 6.035 ©MIT Fall 2001

## Other interactions

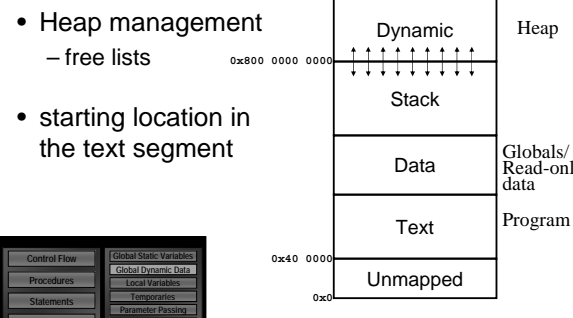
- Other operations
  - Input/Output
  - Privilege / secure operations
  - Handling special hardware
    - TLBs, Caches etc.
- Mostly via system calls
  - hand-coded in assembly
  - compiler can treat them as a normal function call



Saman Amarasinghe 23 6.035 ©MIT Fall 2001

## Memory Layout

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



Saman Amarasinghe 24 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"
```



Saman Amarasinghe

25

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, %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 redefined. `Alignment` is optional. If `alignment` is specified, the address of `name` is aligned to a multiple of `alignment`



Saman Amarasinghe

26

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)

Saman Amarasinghe

27

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

Saman Amarasinghe

28

6.035 ©MIT Fall 2001

## Parameter passing disciplines

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

Saman Amarasinghe

29

6.035 ©MIT Fall 2001

## 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 ???

Saman Amarasinghe

30

6.035 ©MIT Fall 2001

25

## 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
  - Callee Saved

## Question:

30

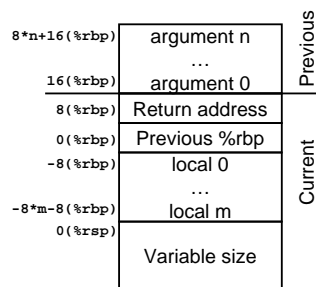
- What are the advantages/disadvantages of:
  - Callee saving of registers?
  - Caller saving of registers?
- What registers should be used at the caller and callee if half is caller-saved and the other half is callee-saved?

## 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
  - Should not be live across procedure calls
  - Will start keeping data in the registers for performance in Segment V

```
mov -4(%rbp), %r10
mov -8(%rbp), %r11
add %r10, %r11
mov %r11, -8(%rbp)
```

## The Stack



## Question:

33

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

## Procedure Linkages

### Standard procedure linkage

*procedure p*

```

prolog
↓
pre-call
↓
post-return
↓
epilog
            
```

*procedure q*

```

prolog
↓
epilog
            
```

**Procedure has**

- standard prolog
- standard epilog

**Each call involves a**

- pre-call sequence
- post-return sequence

Saman Amarasinghe 37 6.035 ©MIT Fall 2001

## 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
            
```

return address	
previous frame pointer	← rbp
callee saved registers	
local variables	
stack temporaries	
dynamic area	
caller saved registers	
argument 2	
argument 1	
argument 0	
return address	

← rsp

Saman Amarasinghe 38 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:
enter   $48, $0
mov     %rbx, -8(%rbp)
            
```

return address	
previous frame pointer	← rbp
callee saved registers	
local variables	
stack temporaries	
dynamic area	
caller saved registers	
argument 2	
argument 1	
argument 0	
return address	
previous frame pointer	← rsp
callee saved registers	
local variables	
stack temporaries	
dynamic area	

Saman Amarasinghe 39 6.035 ©MIT Fall 2001

## 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
                
```

return address	
previous frame pointer	
callee saved registers	
local variables	
stack temporaries	
dynamic area	
caller saved registers	
argument 2	
argument 1	
argument 0	
return address	
previous frame pointer	← rbp
callee saved registers	
local variables	
stack temporaries	
dynamic area	

← rsp

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

Saman Amarasinghe 40 6.035 ©MIT Fall 2001

## 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)
                    
```

return address	
previous frame pointer	
callee saved registers	
local variables	
stack temporaries	
dynamic area	
caller saved registers	
argument 2	
argument 1	
argument 0	
return address	
previous frame pointer	← rbp
callee saved registers	
local variables	
stack temporaries	
dynamic area	

← rsp

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

Saman Amarasinghe 41 6.035 ©MIT Fall 2001

## 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
leave
ret
            
```

return address	
previous frame pointer	← rbp
callee saved registers	
local variables	
stack temporaries	
dynamic area	
caller saved registers	
argument 2	
argument 1	
argument 0	
return address	
previous frame pointer	← rbp
callee saved registers	
local variables	
stack temporaries	
dynamic area	

← rsp

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

Saman Amarasinghe 42 6.035 ©MIT Fall 2001

## 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

return address	← rbp
previous frame pointer	
callee saved registers	
local variables	
stack temporaries	
dynamic area	
caller saved registers	
argument 2	
argument 1	
argument 0	← rsp

```

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

```

Saman Amarasinghe 43 6.035 ©MIT Fall 2001

## Question:

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

Saman Amarasinghe 44 6.035 ©MIT Fall 2001

## 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;
    ...
    return dx + dy + dz;
  }

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

```

Return address	← rbp
previous frame pointer	
Local variable px (10)	
Local variable py (20)	
Local variable pz (30)	← rsp

Saman Amarasinghe 45 6.035 ©MIT Fall 2001

## 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

Saman Amarasinghe 46 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

Saman Amarasinghe 47 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

Saman Amarasinghe 48 6.035 ©MIT Fall 2001