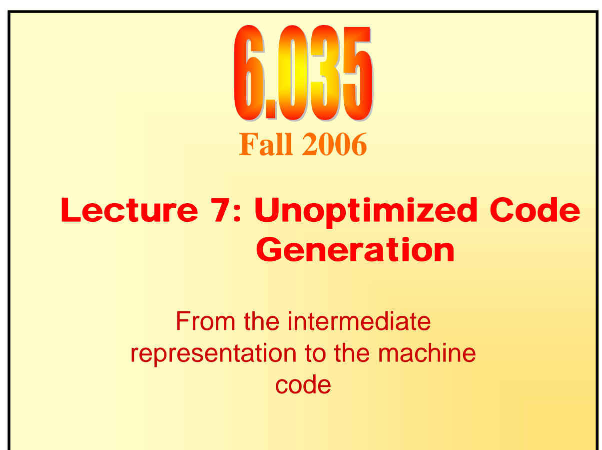
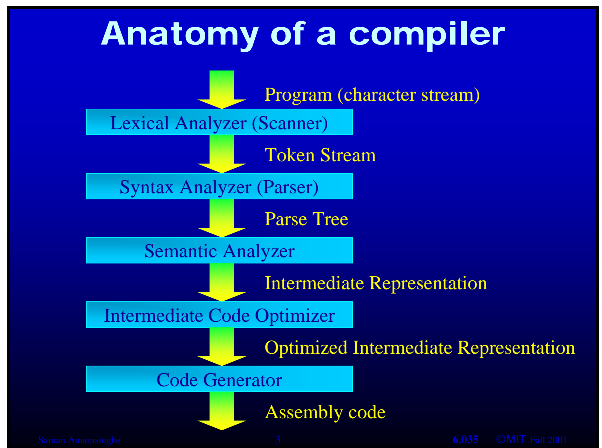
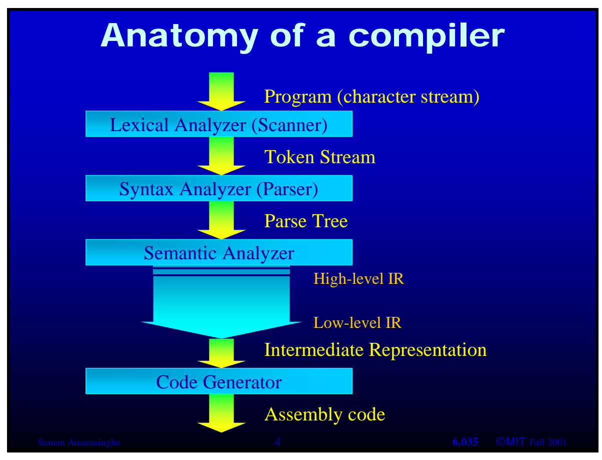
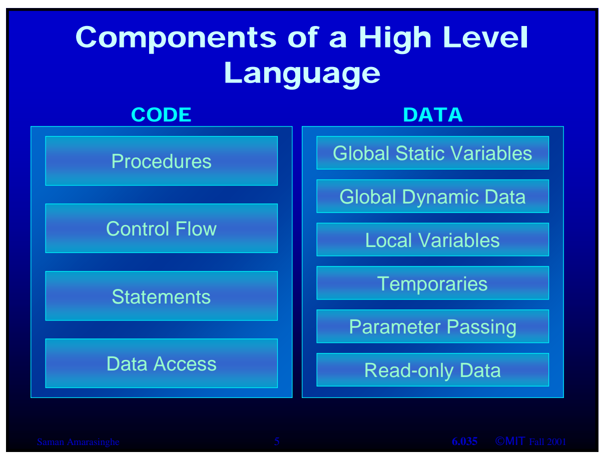


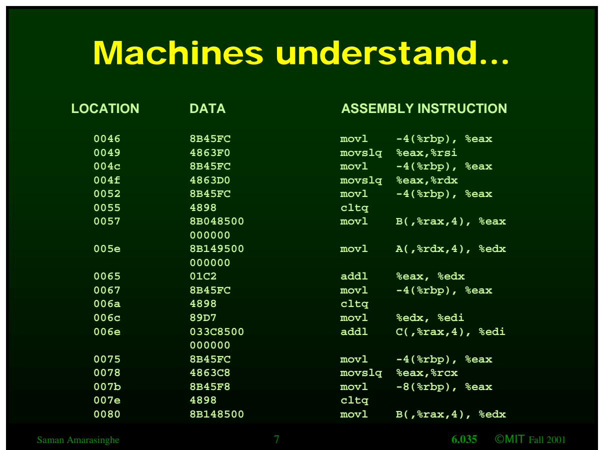


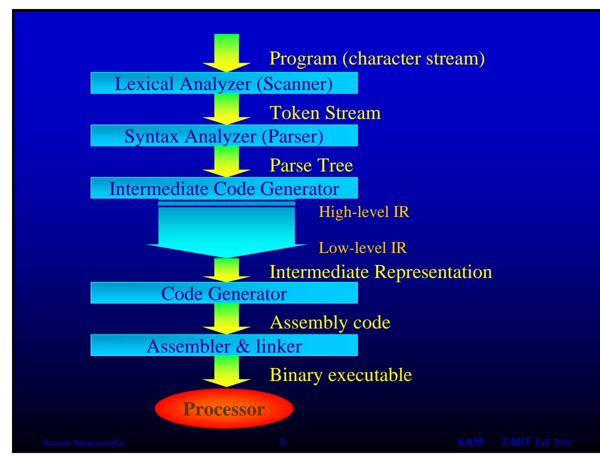




Machine Code Generator Should...

- 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



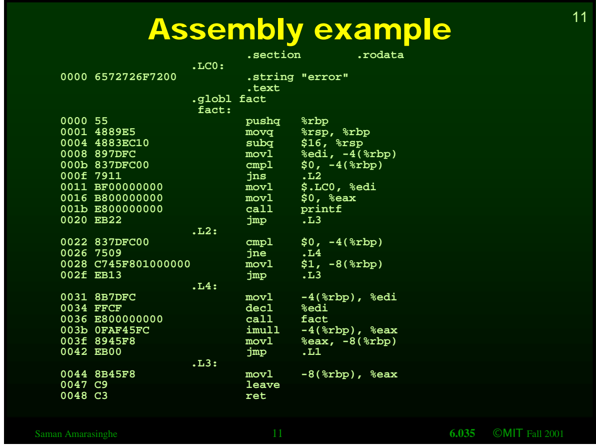


Assembly language

- 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 language

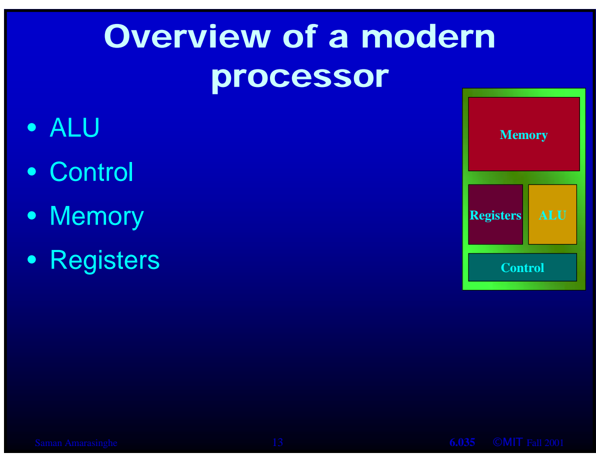
- Relocatable machine language (object modules)
  - all locations(addresses) represented by symbols
  - Mapped to memory addresses at link and load time
  - Flexibility of separate compilation
- Absolute machine language
  - addresses are hard-coded
  - simple and straightforward implementation
  - inflexible -- hard to reload generated code
  - Used in interrupt handlers and device drivers

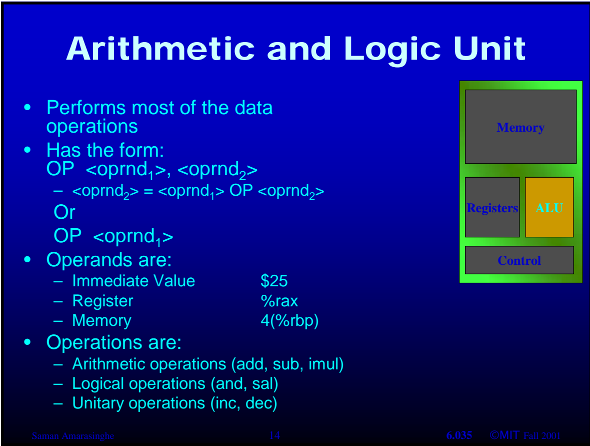


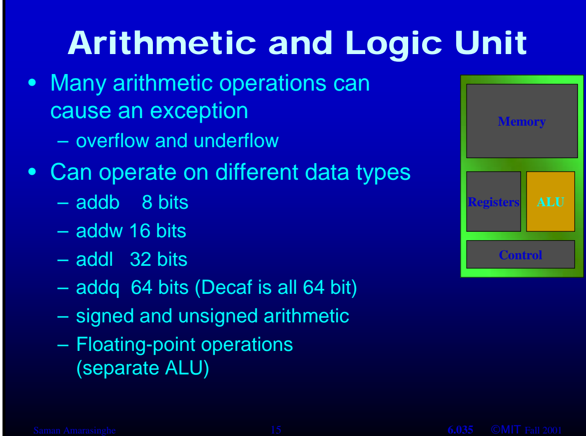
Composition of an Object File

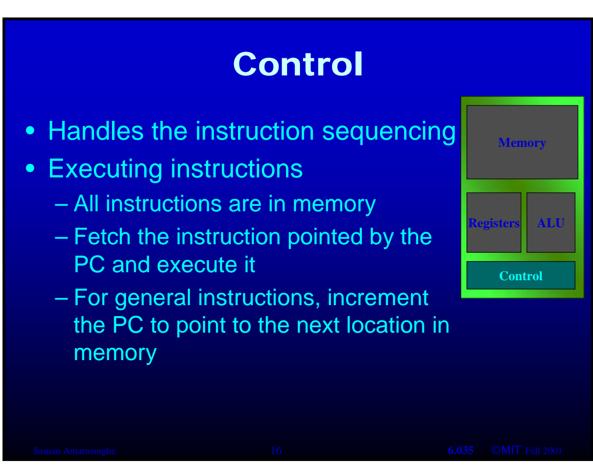
- We use the ELF file format
- The object file has:
  - Multiple Segments
  - Symbol Information
  - Relocation Information
- Segments
  - Global Offset Table
  - Procedure Linkage Table
  - Text (code)
  - Data
  - Read Only Data

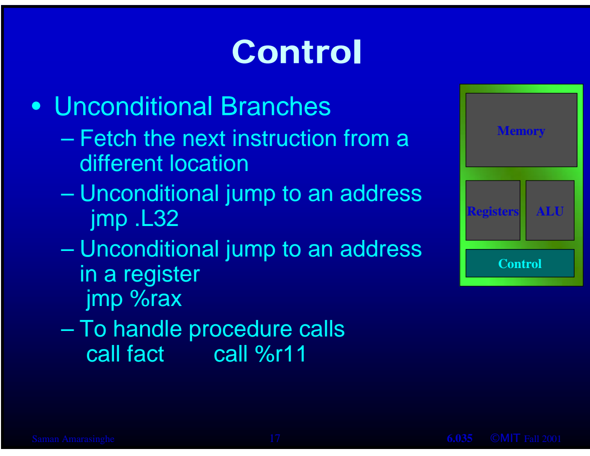
```
 .file   "test2.c"
        .section        .rodata
.LC0:
        .string "error %d"

        .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
```

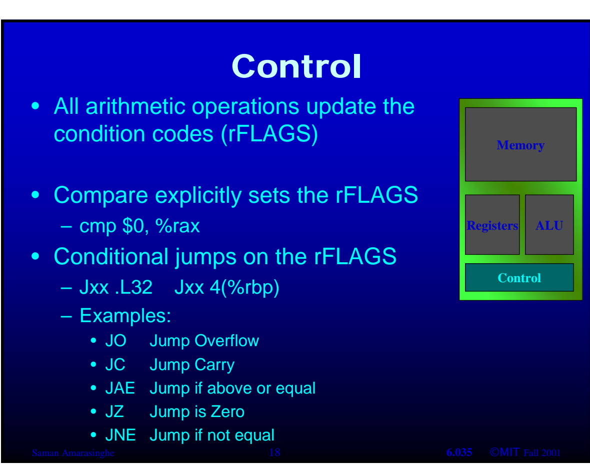

Overview of a modern processor
• ALU
• Control
• Memory
• Registers
Memory
Registers
ALU
Control

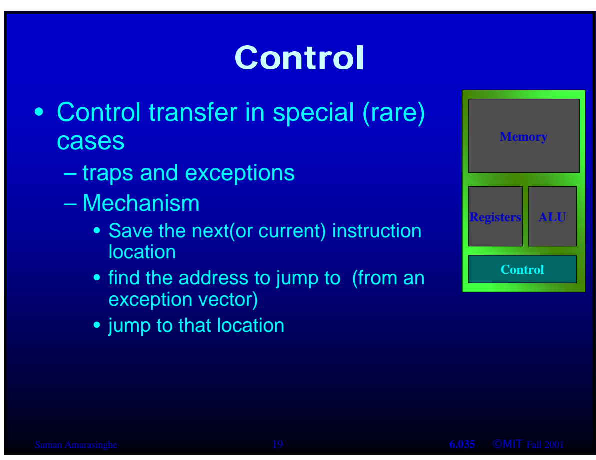

Arithmetic and Logic Unit
• Performs most of the data operations
• Has the form:
OP <oprnd$_1$>, <oprnd$_2$>
– <oprnd$_2$> = <oprnd$_1$> OP <oprnd$_2$>
Or
OP <oprnd$_1$>
• Operands are:
– Immediate Value $25
– Register %rax
– Memory 4(%rbp)
• Operations are:
– Arithmetic operations (add, sub, imul)
– Logical operations (and, sal)
– Unitary operations (inc, dec)
Memory
Registers
ALU
Control


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)
Memory
Registers
ALU
Control

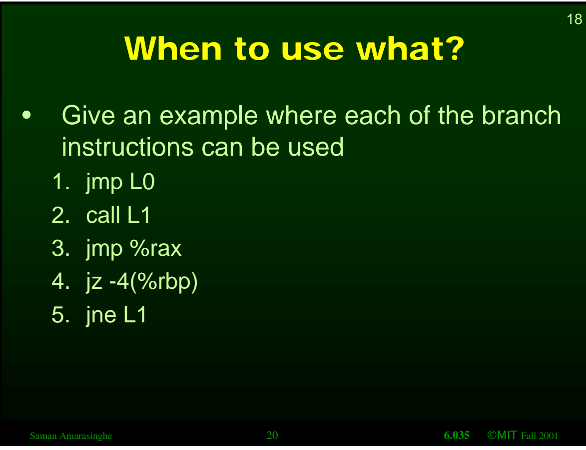

Control
• Handles the instruction sequencing
• Executing instructions
– All instructions are in memory
– Fetch the instruction pointed by the PC and execute it
– For general instructions, increment the PC to point to the next location in memory
Memory
Registers
ALU
Control


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
Memory
Registers
ALU
Control

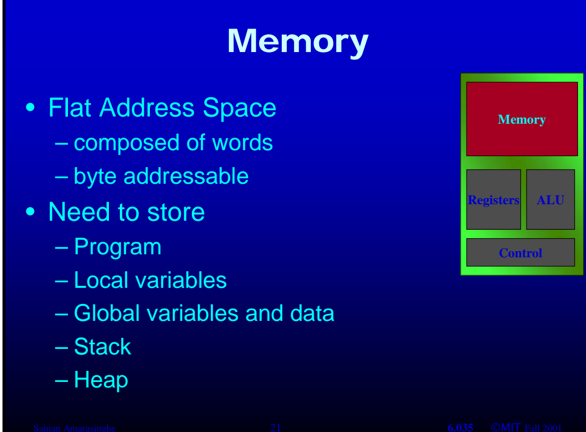

Control
• All arithmetic operations update the condition codes (rFLAGS)
• Compare explicitly sets the rFLAGS
– cmp $0, %rax
• Conditional jumps on the rFLAGS
– Jxx .L32 Jxx 4(%rbp)
– Examples:
• JO Jump Overflow
• JC Jump Carry
• JAE Jump if above or equal
• JZ Jump is Zero
• JNE Jump if not equal
Memory
Registers
ALU
Control



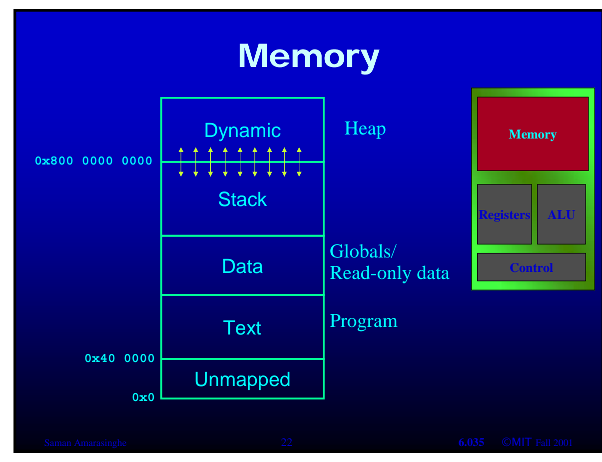
Control
Control transfer in special (rare) cases
– traps and exceptions
– Mechanism
Save the next(or current) instruction location
find the address to jump to (from an exception vector)
jump to that location
Memory
Registers
ALU
Control



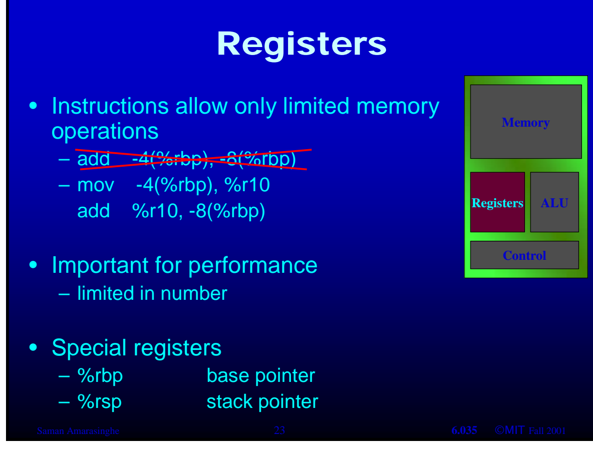
18
When to use what?
Give an example where each of the branch instructions can be used
1. jmp L0
2. call L1
3. jmp %rax
4. jz -4(%rbp)
5. jne L1



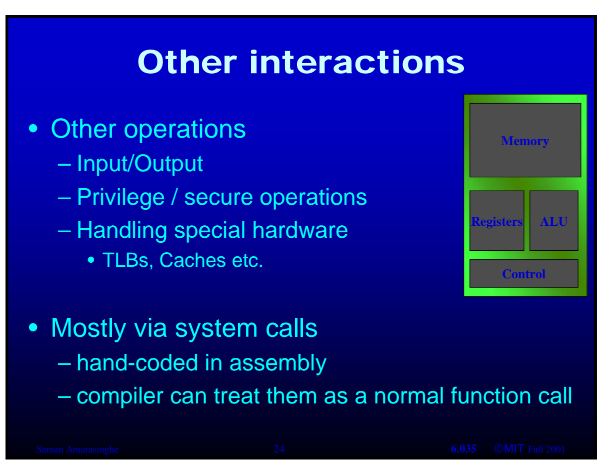
Memory
Flat Address Space
– composed of words
– byte addressable
Need to store
– Program
– Local variables
– Global variables and data
– Stack
– Heap
Memory
Registers
ALU
Control



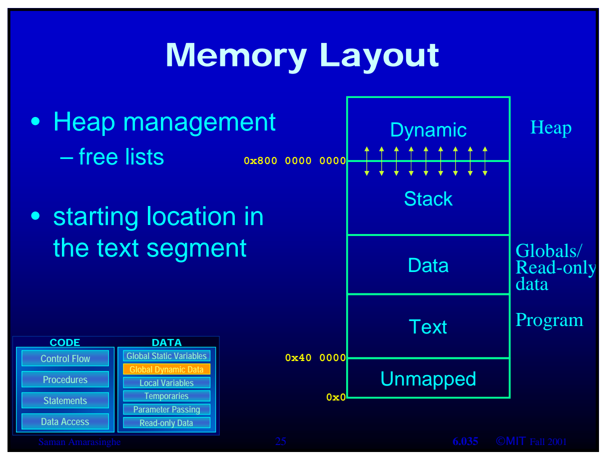
Memory
Dynamic
Heap
0x800 0000 0000
Stack
Data
Globals/
Read-only data
Text
Program
0x40 0000
Unmapped
0x0
Memory
Registers
ALU
Control

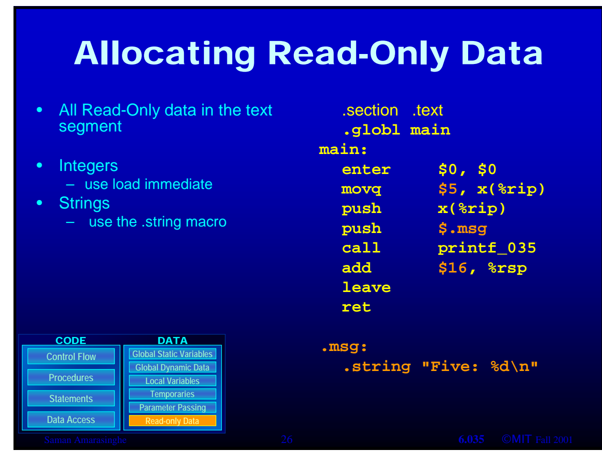


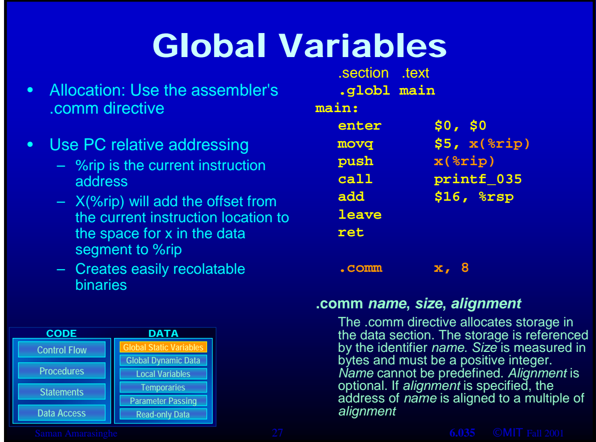
Registers
Instructions allow only limited memory operations
– add -4(%rbp), -8(%rbp)
– mov -4(%rbp), %r10
add %r10, -8(%rbp)
Important for performance
– limited in number
Special registers
– %rbp base pointer
– %rsp stack pointer
Memory
Registers
ALU
Control

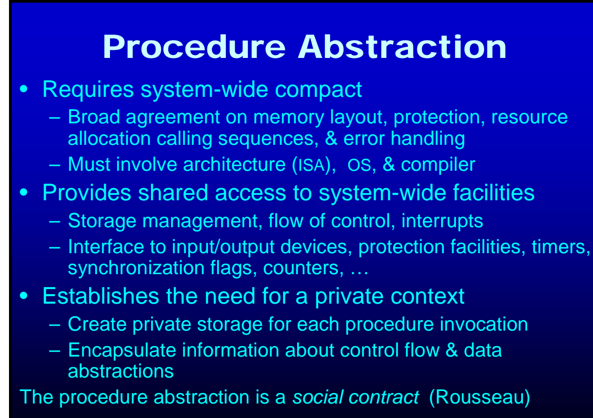


Other interactions
Other operations
– Input/Output
– Privilege / secure operations
– Handling special hardware
TLBs, Caches etc.
Mostly via system calls
– hand-coded in assembly
– compiler can treat them as a normal function call
Memory
Registers
ALU
Control



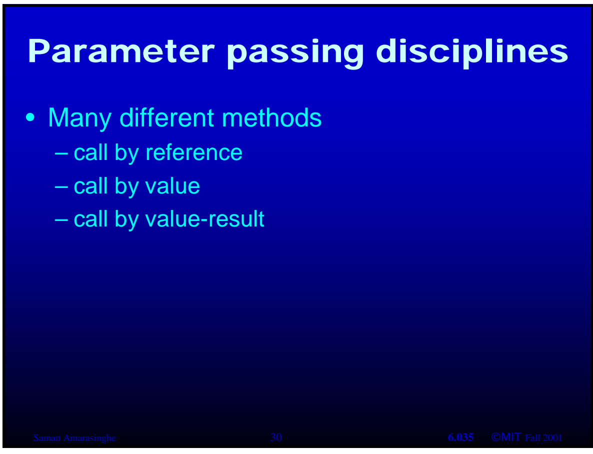






Procedure Abstraction

- 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
      B = B + A
   }
   Main() {
      A = 10;
      foo(A);
   }
}
```

- Call by value A is ???
- Call by reference A is ???
- Call by value-result A is ???

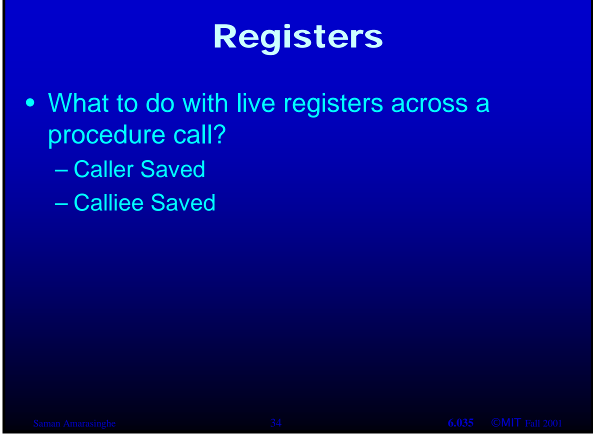
Parameter Passing Disciplines

```
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 ?

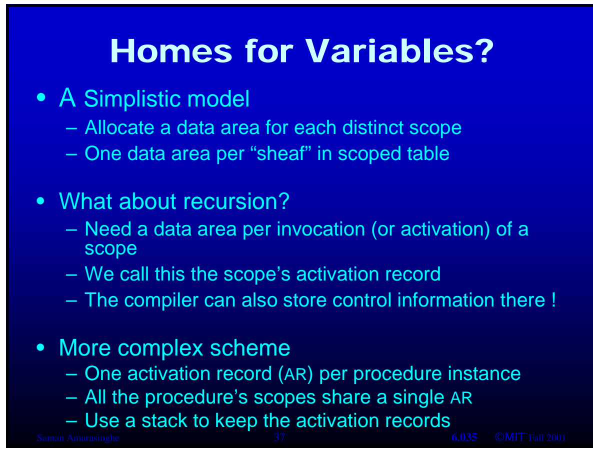
Parameter passing disciplines

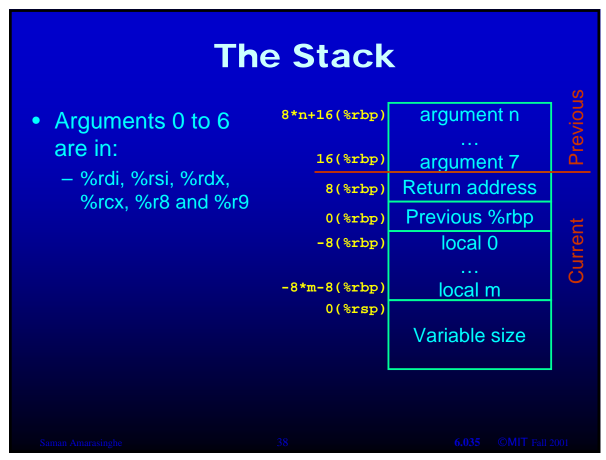
- Many different methods
  - call by reference
  - call by value
  - call by value-result
- How do you pass the parameters?
  - via. the stack
  - via. the registers
  - or a combination
- In the Decaf calling convention, all parameters are passed via the stack



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?



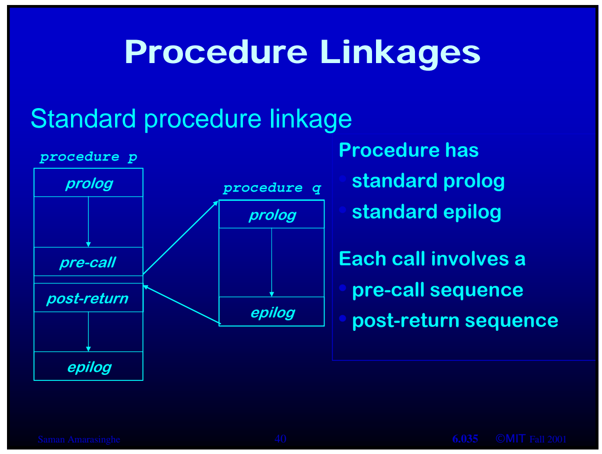


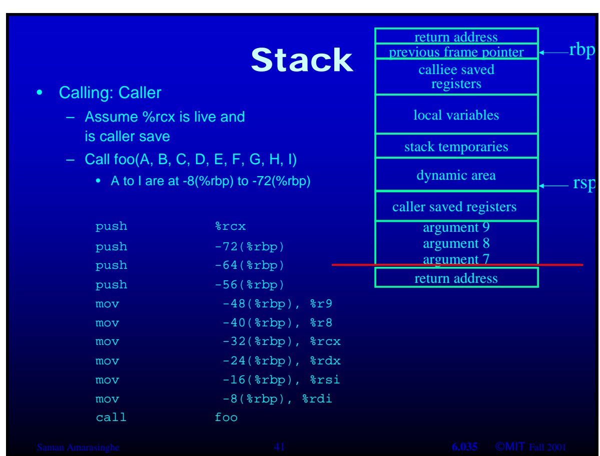
33

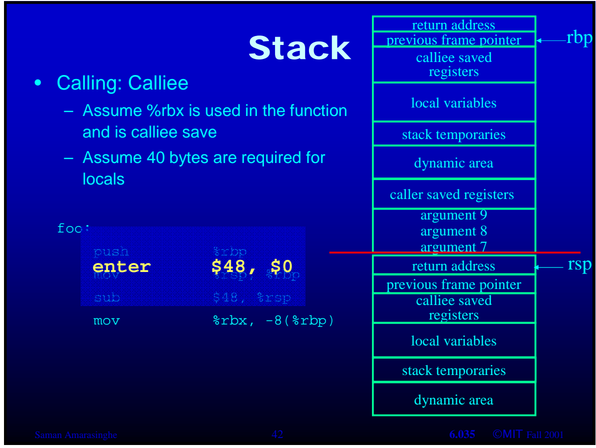
Question:

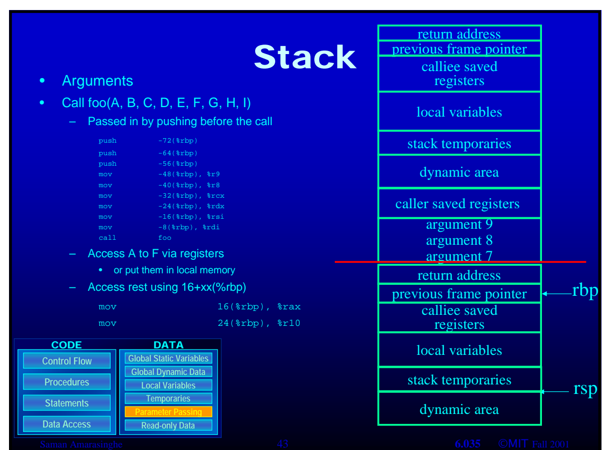
- Why use a stack? Why not use the heap or pre-allocated in the data segment?

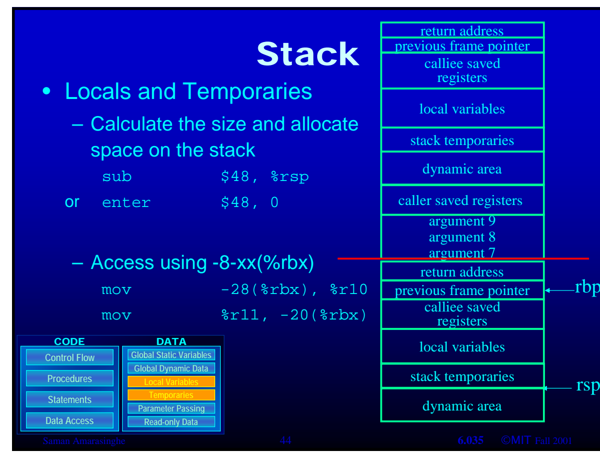
Saman Amarasinghe 39 6.035 ©MIT Fall 2001

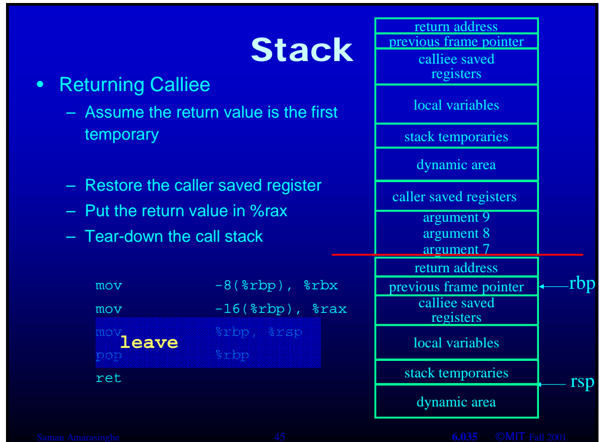


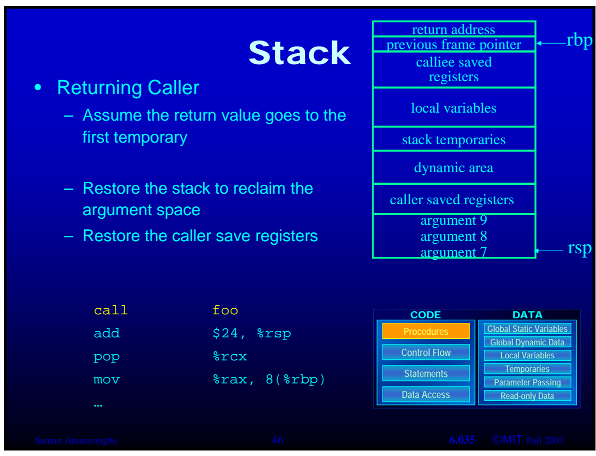


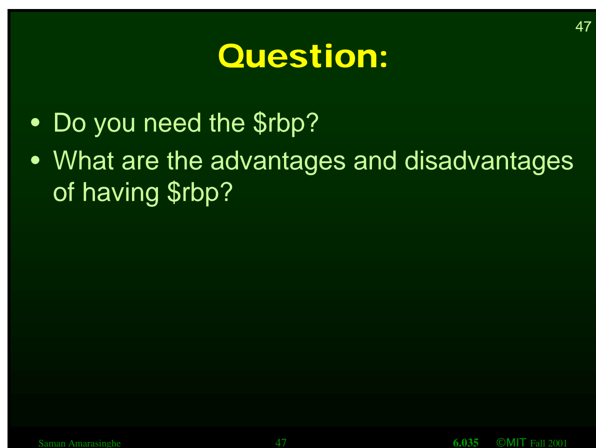


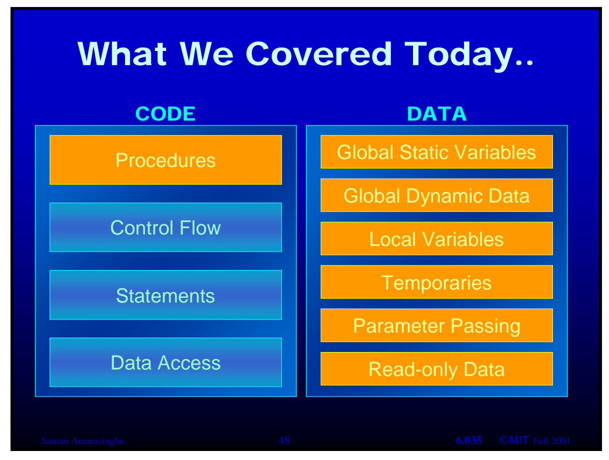

Stack
Arguments
Call foo(A, B, C, D, E, F, G, H, I)
Passed in by pushing before the call
push -72(%rbp)
push -64(%rbp)
push -56(%rbp)
mov -48(%rbp), %r9
mov -40(%rbp), %r8
mov -32(%rbp), %rcx
mov -24(%rbp), %rdx
mov -16(%rbp), %rsi
mov -8(%rbp), %rdi
call foo
Access A to F via registers
or put them in local memory
Access rest using 16+xx(%rbp)
mov 16(%rbp), %rax
mov 24(%rbp), %r10
return address
previous frame pointer
calliee saved registers
local variables
stack temporaries
dynamic area
caller saved registers
argument 9
argument 8
argument 7
return address
previous frame pointer
calliee saved registers
local variables
stack temporaries
dynamic area
rbp
rsp
CODE
Control Flow
Procedures
Statements
Data Access
DATA
Global Static Variables
Global Dynamic Data
Local Variables
Temporaries
Parameter Passing
Read-only Data
Saman Amarasinghe
43
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
calliee saved registers
local variables
stack temporaries
dynamic area
caller saved registers
argument 9
argument 8
argument 7
return address
previous frame pointer
calliee saved registers
local variables
stack temporaries
dynamic area
rbp
rsp
CODE
Control Flow
Procedures
Statements
Data Access
DATA
Global Static Variables
Global Dynamic Data
Local Variables
Temporaries
Parameter Passing
Read-only Data
Saman Amarasinghe
44
6.035 ©MIT Fall 2001


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
return address
previous frame pointer
calliee saved registers
local variables
stack temporaries
dynamic area
caller saved registers
argument 9
argument 8
argument 7
return address
previous frame pointer
calliee saved registers
local variables
stack temporaries
dynamic area
rbp
rsp
Saman Amarasinghe
45
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
call foo
add $24, %rsp
pop %rcx
mov %rax, 8(%rbp)
…
return address
previous frame pointer
calliee saved registers
local variables
stack temporaries
dynamic area
caller saved registers
argument 9
argument 8
argument 7
rbp
rsp
CODE
Procedures
Control Flow
Statements
Data Access
DATA
Global Static Variables
Global Dynamic Data
Local Variables
Temporaries
Parameter Passing
Read-only Data
Saman Amarasinghe
46
6.035 ©MIT Fall 2001


47
Question:
Do you need the $rbp?
What are the advantages and disadvantages of having $rbp?
Saman Amarasinghe
47
6.035 ©MIT Fall 2001


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
Saman Amarasinghe
48
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

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