6.035

Fall 2000

Lecture 8: Unoptimized Code Generation

From the intermediate representation
to the machine code

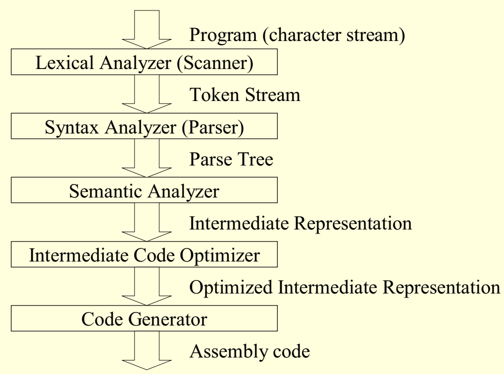
Segment IV Roadmap

- There is a Quiz!
  - On 10/19 in-class
  - But no Homework
  - A sample Quiz will be given shortly
- Checkpoint
  - On 10/26
  - 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

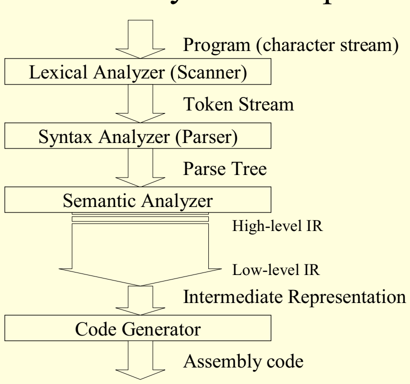
Outline

- Introduction
- Machine Language
- Overview of a modern processor
- Procedure Abstraction
- Procedure Linkage
- Guidelines in Creating a Code Generator

Anatomy of a compiler

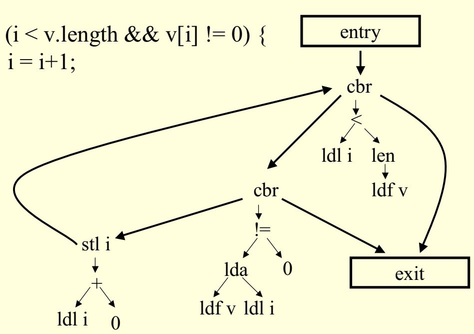


Anatomy of a compiler



Intermediate Format Representation

```
while (i < v.length && v[i] != 0) {
      i = i+1;
}
```



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

Outline

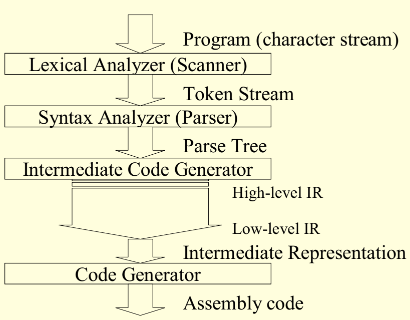
- Introduction
- Machine Language
- Overview of a modern processor
- Procedure Abstraction
- Procedure Linkage
- Guidelines in Creating a Code Generator

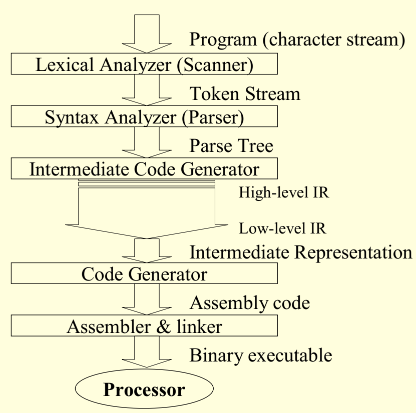
Machines understand...

location	data
0x4009b0:	3c1c0fc0
0x4009b4:	279c7640
0x4009b8:	0399e021
0x4009bc:	8f998044
0x4009c0:	27bdffe0
0x4009c4:	afbf001c
0x4009c8:	afbc0018
0x4009cc:	0320f809
0x4009d0:	2404000a
0x4009d4:	8fbf001c
0x4009d8:	8fbc0018
0x4009dc:	27bd0020
0x4009e0:	03e00008
0x4009e4:	00001025

Machines understand...

```
                  location     data        assembly instruction
main:
  [test.c:   3] 0x4009b0:    3c1c0fc0    lui     gp,0xfc0
  [test.c:   3] 0x4009b4:    279c7640    addiu   gp,gp,30272
  [test.c:   3] 0x4009b8:    0399e021    addu    gp,gp,t9
  [test.c:   3] 0x4009bc:    8f998044    lw      t9,-32700(gp)
  [test.c:   3] 0x4009c0:    27bdffe0    addiu   sp,sp,-32
  [test.c:   3] 0x4009c4:    afbf001c    sw      ra,28(sp)
  [test.c:   3] 0x4009c8:    afbc0018    sw      gp,24(sp)
  [test.c:   3] 0x4009cc:    0320f809    jalr    ra,t9
  [test.c:   3] 0x4009d0:    2404000a    li      a0,10
  [test.c:   3] 0x4009d4:    8fbf001c    lw      ra,28(sp)
  [test.c:   3] 0x4009d8:    8fbc0018    lw      gp,24(sp)
  [test.c:   3] 0x4009dc:    27bd0020    addiu   sp,sp,32
  [test.c:   3] 0x4009e0:    03e00008    jr      ra
  [test.c:   3] 0x4009e4:    00001025    move    v0,zero
```



Assembly language

- Advantages
  - Simplifies code generation due to use of symbolic instructions and symbolic names
  - Logical abstraction layer
  - Architectures can describe by an 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

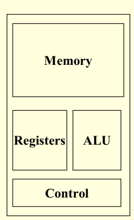
```
  .data
item:
  .word     1
  .text
fib:
  subu      $sp, 40
  sw        $31, 28($sp)
  sw        $4, 40($sp)
  sw        $16, 20($sp)
  .frame    $sp, 40, $31
#   7   if(n == 0) return 0;
  lw        $14, 40($sp)
  bne       $14, 0, $32
  move      $2, $0
  b         lab2
lab1:
  lw        $15, 40($sp)
  bne       $15, 1, $33
  li        $2, 1
  b         lab1
```


Outline

- Introduction
- Machine Language
- Overview of a modern processor
- Procedure Abstraction
- Procedure Linkage
- Guidelines in Creating a Code Generator

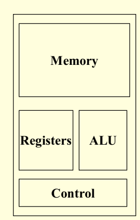
Overview of a modern processor

- ALU
- Control
- Memory
- Registers

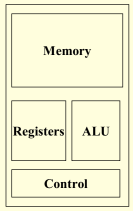


Arithmetic and Logic Unit

- Performs most of the data operations
- Has the form:
  $OP\ R_{dest}, R_{src1}, R_{src2}$
- Operations are:
  - Arithmetic operations (add, sub, mulo)
  - Logical operations (and, sll)
  - Comparison operations (seq, sge, slt)

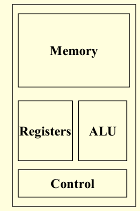


Arithmetic and Logic Unit



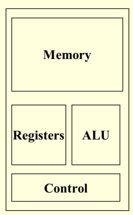
- Many arithmetic operations can cause an exception
  - overflow and underflow
- Can operate on different data types
  - 8, 16, 32 bits
  - signed and unsigned arithmetic
  - Floating-point operations (separate ALU)
  - Instructions to convert between formats (cvt.s.d)

Control



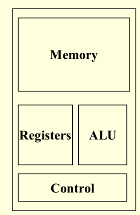
- Handles the instruction sequencing
- Executing instructions
  - All instructions are in memory
  - Fetch the instruction pointed by the PC and execute it
  - For general instructions, increment the PC to point to the next location in memory

Control



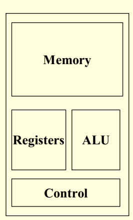
- Unconditional Branches
  - Fetch the next instruction from a different location
  - Unconditional jump to a given address
    j label
  - Unconditional jump to an address in a register
    jr r_{src}
  - To handle procedure calls, do an unconditional jump, but save the next address in the current stream in a register
    jal label jalr r_{src}

Control



- Conditional Branches
  - Perform a test,
    if successful fetch instructions from a new address,
    otherwise fetch the next instruction
  - Instructions are of the form:
    b*relop* R_{src1}, R_{src2}, label
  - relop is of the form:
    eq, ne, gt, ge, lt, le

Control



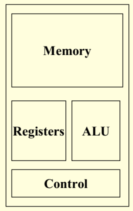
- Control transfer in special (rare) cases
  - traps and exceptions
  - Mechanism
    - Save the next(or current) instruction location
    - find the address to jump to (from an exception vector)
    - jump to that location

When to use what?

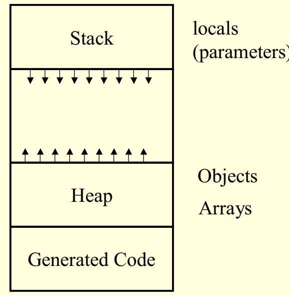
- Give an example where each of the branch instructions can be used
  1. j label
  2. jal label
  3. jr r_{src}
  4. jalr r_{src}
  5. beq R_{src1}, R_{src2}, label

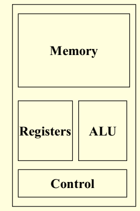
Memory

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



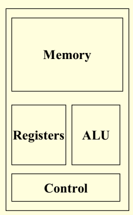
Memory





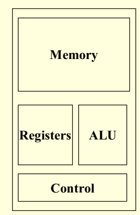
Registers

- Load/store architecture
  - All operations are on register values
  - Need to bring data in-to/out-of registers
- Important for performance
  - limited in number



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



The MIPS ISA and MIPS Processor

- One of the earliest RISC processors
  - Has evolved from 1980's
  - ISA has also evolved
    - Always backward compatible, I.e. add more to the ISA
    - MIPS-I, MIPS-II….MIPS-V
  - Many processor incarnation
    - From a simple 5-stage pipeline to an out-of-order superscalar
    - R2000, R4000, R8000, R10000 …..
- You will be generating code for it

Diversity of Processors

- General Purpose Processors
  - x86, PowerPC, MIPS R4000, HP PA-RISC, Alpha
- Digital Signal Processors (DSP)
  - TI 56000
- Supercomputing Processors
  - Cray
- Embedded Processors
  - StrongARM
- Network Processors

Diversity of Processors

- Diversity in execution
  - VLIW, Superscalar, Vector, Systolic Arrays
- Diversity in the memory system
  - Multiple memories in DSPs
  - register windows in SPARC
- Different/unique ISAs
- Different goals/markets
  - All out performance in supercompuers
  - Maximum energy savings in embedded processors

22

Outline

- Introduction
- Machine Language
- Overview of a modern processor
- Procedure Abstraction
- Procedure Linkage
- Guidelines in Creating a Code Generator

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

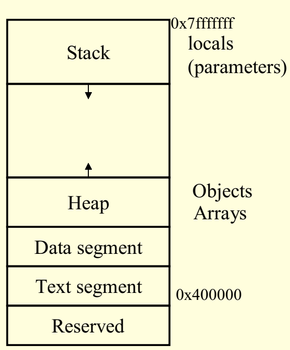
The procedure abstraction is a *social contract* (Rousseau)

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
  - Memory layout
  - Registers
  - Stack

Memory Layout

- Start of the stack
- Heap management
  - free lists
- starting location in the text segment



Parameter passing disciplines

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

Parameter Passing Disciplines

```
A = 10;
Call foo(A)

Subroutine foo(B)
  B = B + 1
  B = B + A
```

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

Parameter Passing Disciplines

```
A = 10;
Call foo(A)

Subroutine foo(B)
  B = B + 1
  B = B + A
```

- Call by value A is 10
- Call by reference A is 22
- Call by value-result A is 21

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

Registers

- Not a register, hard-wired to the constant 0

0	zero	hard-wired to zero

Registers

- Return Address from a call
  - implicitly copied by jal and jalr instructions

0	zero	hard-wired to zero
31	ra	return address

Registers

- Frame pointer
- Stack pointer
- Pointer to global area

0	zero	hard-wired to zero
28	gp	pointer to global area
29	sp	stack pointer
30	fp	frame pointer
31	ra	return address

Registers

- Reserved for assembler to use
  - need storage to handle compound asm instructions

0	zero	hard-wired to zero
1	at	Reserved for asm
28	gp	pointer to global area
29	sp	stack pointer
30	fp	frame pointer
31	ra	return address

Registers

- Returns the results
  - copy the result when ready to return
  - used to evaluate expressions (up to you)

0	zero	hard-wired to zero
1	at	Reserved for asm
2 - 3	v0 - v1	expr. eval and return of results
28	gp	pointer to global area
29	sp	stack pointer
30	fp	frame pointer
31	ra	return address

Registers

- First four arguments to a call
  - Can use it for other purposes when args are dead
  - If more arguments ⇒pass them via the stack

0	zero	hard-wired to zero
1	at	Reserved for asm
2 - 3	v0 - v1	expr. eval and return of results
4 - 7	a0 - a3	arguments 1 to 4
28	gp	pointer to global area
29	sp	stack pointer
30	fp	frame pointer
31	ra	return address

Registers

- Rest are temporaries

0	zero	hard-wired to zero
1	at	Reserved for asm
2 - 3	v0 - v1	expr. eval and return of results
4 - 7	a0 - a3	arguments 1 to 4
8 - 25		keep temporary values
28	gp	pointer to global area
29	sp	stack pointer
30	fp	frame pointer
31	ra	return address

Registers

- Across a procedure call temporaries need to be:
  - Saved by the caller
  - Saved by the calliee
  - Some combination of both

0	zero	hard-wired to zero
1	at	Reserved for asm
2 - 3	v0 - v1	expr. eval and return of results
4 - 7	a0 - a3	arguments 1 to 4
8 - 25		keep temporary values
28	gp	pointer to global area
29	sp	stack pointer
30	fp	frame pointer
31	ra	return address

Registers

- Across a procedure call temporaries need to be:
  - Saved by the caller
  - Saved by the calliee
  - Some combination of both

0	zero	hard-wired to zero
1	at	Reserved for asm
2 - 3	v0 - v1	expr. eval and return of results
4 - 7	a0 - a3	arguments 1 to 4
8-15	t0 - t7	caller saved temporary
16 - 23	s0 - s7	calliee saved temporary
24, 25	t8, t9	caller saved temporary
28	gp	pointer to global area
29	sp	stack pointer
30	fp	frame pointer
31	ra	return address

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?
  - Caller-saved t0 - t9
  - Calliee-saved s0-s7

Where to the Variables Live?

- 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

Question:

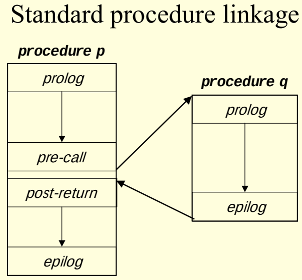
- Why use a stack? Why not use the heap or pre-allocated in the data segment?

Outline

- Introduction
- Machine Language
- Overview of a modern processor
- Procedure Abstraction
- Procedure Linkage
- Guidelines in Creating a Code Generator

Procedure Linkages

Standard procedure linkage



Procedure has
- standard prolog
- standard epilog

Each call involves a
- pre-call sequence
- post-return sequence

Procedure Linkages

- Pre-call Sequence
  - Sets up callee’s basic AR
  - Helps preserve its own environment
- The details
  - Allocate space for the callee’s AR
  - Evaluates each parameter & stores value or address
  - Saves return address, caller’s ARP into callee’s AR
  - Save any caller-save registers
    - Save into space in caller’s AR
  - Jump to address of callee’s prolog code

Procedure Linkages

- Post-return Sequence
  - Finish restoring caller's environment
  - Place any value back where it belongs
- The details
  - Copy return value from callee's AR, if necessary
  - Free the callee's AR
  - Restore any caller-save registers
  - Restore any call-by-reference parameters to registers, if needed
  - Continue execution after the call

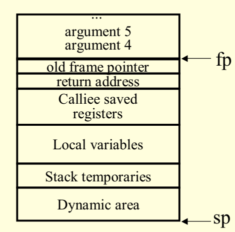
Procedure Linkages

- Prolog Code
  - Finish setting up the callee's environment
  - Preserve parts of the caller's environment that will be disturbed
- The Details
  - Preserve any callee-save registers
  - Allocate space for local data
    - Easiest scenario is to extend the AR
  - Find any static data areas referenced in the callee
  - Handle any local variable initializations

Procedure Linkages

- Eplilog Code
  - Wind up the business of the callee
  - Start restoring the caller's environment
- The Details
  - Restore callee-save registers
  - Free space for local data, if necessary
  - Load return address from AR
  - Restore caller's ARP
  - Jump to the return address

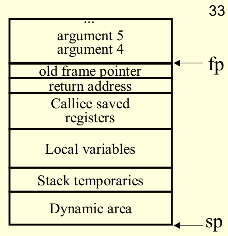
Stack



- Address of the nth argument is -(n-4)*4*$fp
- Local variables are a positive constant off $fp

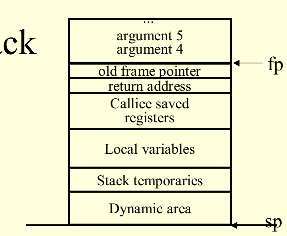
33

Stack



- When calling a new procedure

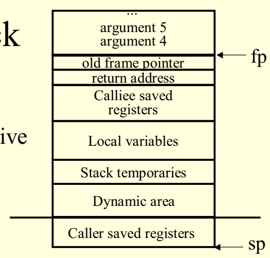
Stack



- When calling a new procedure, caller:

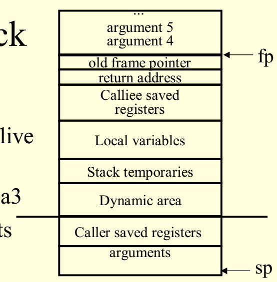
Stack

- When calling a new procedure, caller:
  - push any t0-t9 that has a live value on the stack



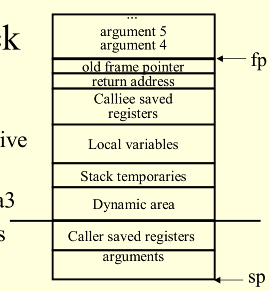
Stack

- When calling a new procedure, caller:
  - push any t0-t9 that has a live value on the stack
  - put arguments 1-4 on a0-a3
  - push rest of the arguments on the stack



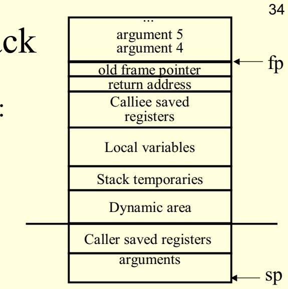
Stack

- When calling a new procedure, caller:
  - push any t0-t9 that has a live value on the stack
  - put arguments 1-4 on a0-a3
  - push rest of the arguments on the stack
  - do a jal or jalr



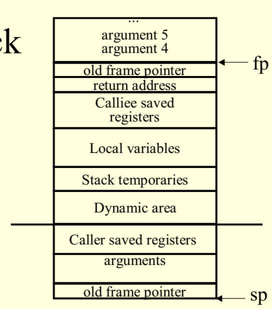
Stack

- In a procedure call, the calliee at the beginning:



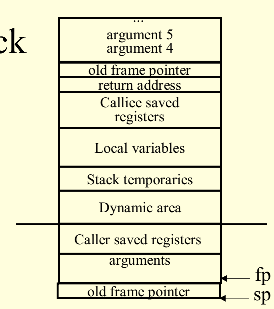
Stack

- In a procedure call, the calliee at the beginning:
  - push $fp on the stack



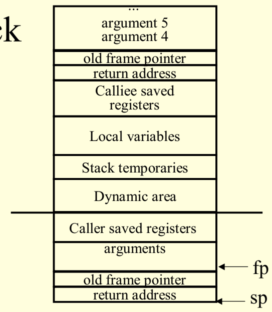
Stack

- In a procedure call, the calliee at the beginning:
  - push $fp on the stack
  - copy $sp+4 to $fp



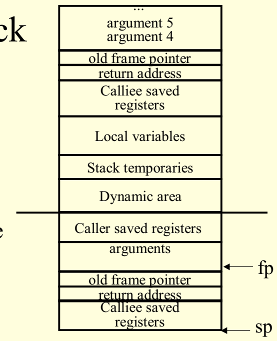
Stack

- In a procedure call, the calliee at the beginning:
  – push $fp on the stack
  – copy $sp+4 to $fp
  – push $ra on the stack



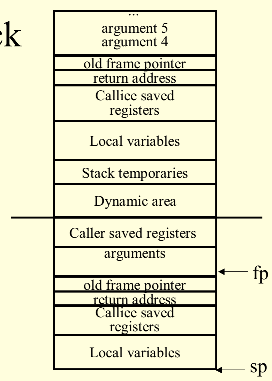
Stack

- In a procedure call, the calliee at the beginning:
  – push $fp on the stack
  – copy $sp+4 to $fp
  – push $ra on the stack
  – if any s0-s7 is used in the procedure save it on the stack



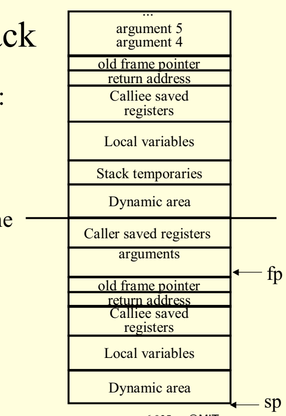
Stack

- In a procedure call, the calliee at the beginning:
  – push $fp on the stack
  – copy $sp+4 to $fp
  – push $ra on the stack
  – if any s0-s7 is used in the procedure save it on the stack
  – create space for local variables on the stack



Stack

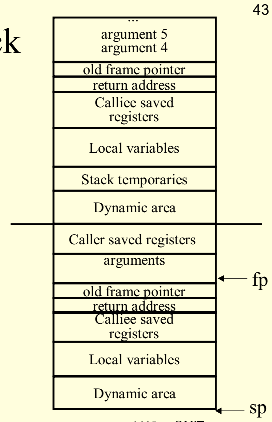
- In a procedure call, the calliee at the beginning:
  – push $fp on the stack
  – copy $sp+4 to $fp
  – push $ra on the stack
  – if any s0-s7 is used in the procedure save it on the stack
  – create space for local variables on the stack
  – execute the calliee...



43

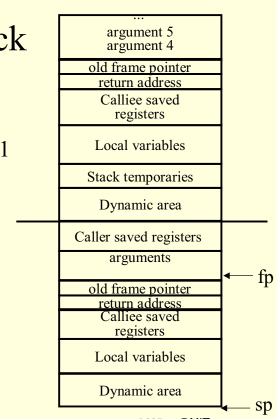
Stack

- In a procedure call, the calliee at the end:



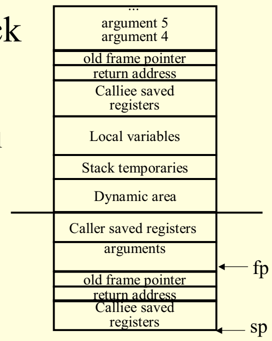
Stack

- In a procedure call, the calliee at the end:
  – put return values on v0,v1



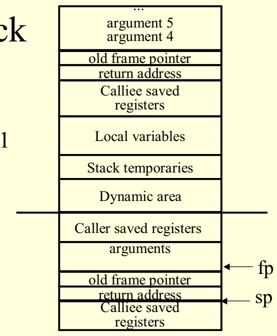
Stack

- In a procedure call, the calliee at the end:
  - put return values on v0,v1
  - update $sp using $fp ($fp+8) + ...



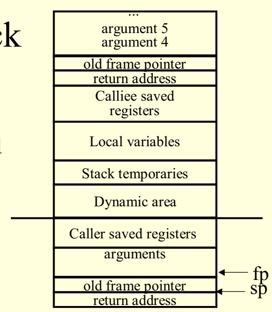
Stack

- In a procedure call, the calliee at the end:
  - put return values on v0,v1
  - update $sp using $fp ($fp+8) + ...
  - Pop the calliee saved registers from stack



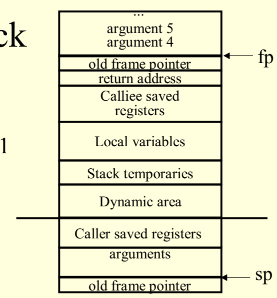
Stack

- In a procedure call, the calliee at the end:
  - put return values on v0,v1
  - update $sp using $fp ($fp+8) + ...
  - Pop the calliee saved registers from stack
  - restore $ra from stack



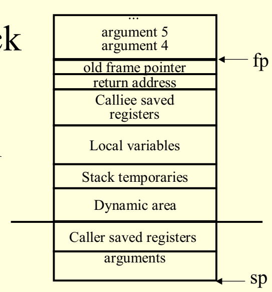
Stack

- In a procedure call, the calliee at the end:
  - put return values on v0,v1
  - update $sp using $fp ($fp+8) + ...
  - Pop the calliee saved registers from stack
  - restore $ra from stack
  - restore $fp from stack



Stack

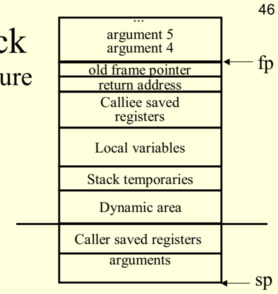
- In a procedure call, the calliee at the end:
  - put return values on v0,v1
  - update $sp using $fp ($fp+8) + ...
  - Pop the calliee saved registers from stack
  - restore $ra from stack
  - restore $fp from stack
  - execute jr ra and return to caller



46

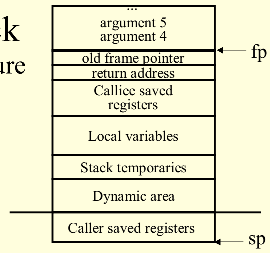
Stack

- On return from a procedure call, the caller:



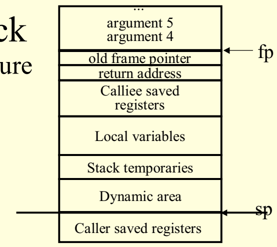
Stack

- On return from a procedure call, the caller:
  - Update $sp to ignore arguments



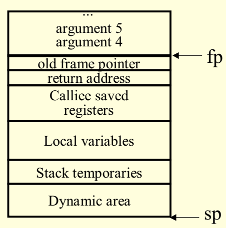
Stack

- On return from a procedure call, the caller:
  - Update $sp to ignore arguments
  - pop the caller saved registers



Stack

- On return from a procedure call, the caller:
  - Update $sp to ignore arguments
  - pop the caller saved registers
  - Continue...



47

Question:

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

Example Program

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

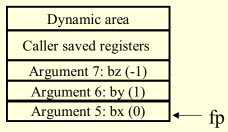

Example Program

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

…
int px, py, pz;
…
auxmath am;
am.sum3d(px, py, pz, 0, 0, 0);
```

Example Program

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

…
int px, py, pz;
px = 10; py = 20; pz = 30;
auxmath am;
am.sum3d(px, py, pz, 0, 1, -1);
```

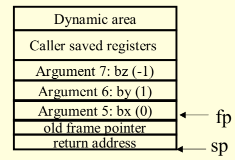
v0	??
v1	??
a0	this
a1	ax (10)
a2	ay (20)
a3	az (30)
v0	??
v1	??
t0	??
t1	??



Example Program

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

…
int px, py, pz;
px = 10; py = 20; pz = 30;
auxmath am;
am.sum3d(px, py, pz, 0, 1, -1);
```

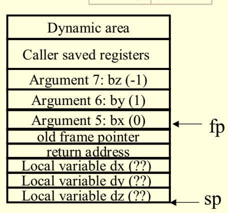
v0	??
v1	??
a0	this
a1	ax (10)
a2	ay (20)
a3	az (30)
v0	??
v1	??
t0	??
t1	??



Example Program

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

…
int px, py, pz;
px = 10; py = 20; pz = 30;
auxmath am;
am.sum3d(px, py, pz, 0, 1, -1);
```

v0	??
v1	??
a0	this
a1	ax (10)
a2	ay (20)
a3	az (30)
v0	??
v1	??
t0	??
t1	??



Outline

- Introduction
- Machine Language
- Overview of a modern processor
- Procedure Abstraction
- Procedure Linkage
- Guidelines in Creating a Code Generator

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
- Make sure you know want can be done at…
  - Compile time in the compiler
  - Runtime in a runtime library
  - Runtime using generated code
- Runtime library is your friend!
  - Don’t try to generate complex code sequences when it can be done in a runtime library assembly hack
  - Example: malloc

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