CMPSC 160
Translation of Programming Languages

Lecture 15: Code Generation: Stack Machine Code

Recap on Last Lecture

```
if (x < y)
   x = 5*y + 5*y/3;
else
   y = 5;
x = x+y;
```

```
     load x
     load y
     iflt L1
     push 0
     goto L2
L1:  push 1
L2:  ifne L3:
     goto L4
L3:  push 5
     load y
     multiply
     push 5
     load y
     multiply
     push 3
     divide
     add
     store x
     goto L5
L4:  push 5
     store y
L5:  load x
     load y
     add
     store x
```

pushes the value at the location x to the stack

pops the top two elements and compares them

pops the top element and compares it to 0

pops the top two elements, multiplies them, and pushes the result back to the stack

stores the value at the top of the stack to the location x

Tips for Code Generation:

1. get an idea of your input and out programs
2. figure out their mapping

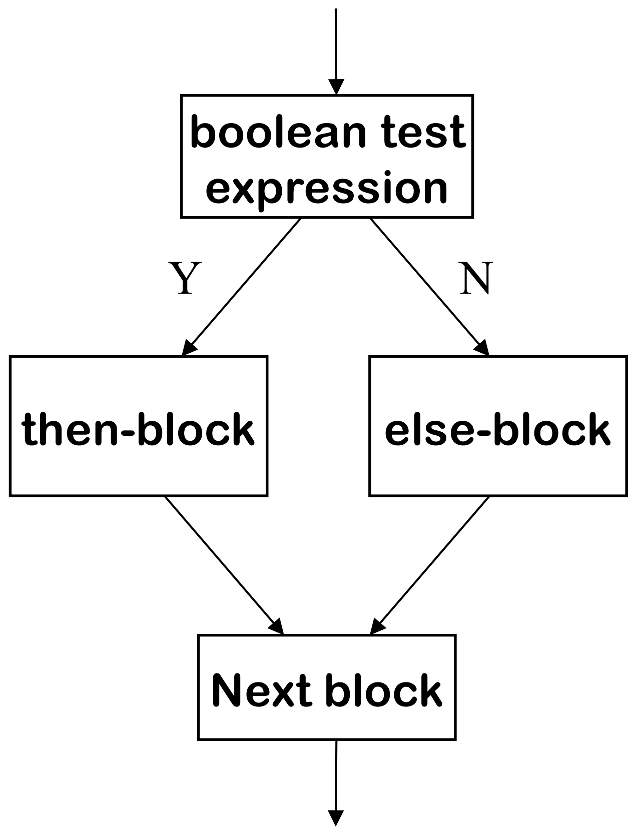
Last lecture:

1. Arithmetic Expression
2. Boolean Expression: numerical evaluation

Flow-of-Control Statements

If-then-else

- Branch based on the result of boolean test expression



Flow-of-Control Statements: Code Structure

- We have to decide on the code layout for the code for flow-of-control, as on hardware, we can only have straight-line code + jumps

$$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$$

	E.code
	if E is false, goto L1
	S_1*.code*
	goto L2:
L1:	S_2*.code*
L2:	⋮

This will be the label of the instruction that comes after this one

Flow-of-Control Statements, Stack-Based Code, Assuming Numeric Representation for Boolean Expressions

Productions	Semantic Rules
$S \rightarrow$ if E then S_1 else S_2	`L1` $\leftarrow$ newlabel(); `L2` $\leftarrow$ newlabel(); $S.code \leftarrow E.code$ || gen(`` `ifeq L1' ``) || $S_1.code$ || gen(`'goto L2'`) || gen(`'L1 :'`) || $S_2.code$ || gen(`'L2 :'`);

	E.code
	if E is false, goto L1
	$S_1.code$
	goto L2:
L1:	$S_2.code$
L2:	⋮

`ifeq <label>`

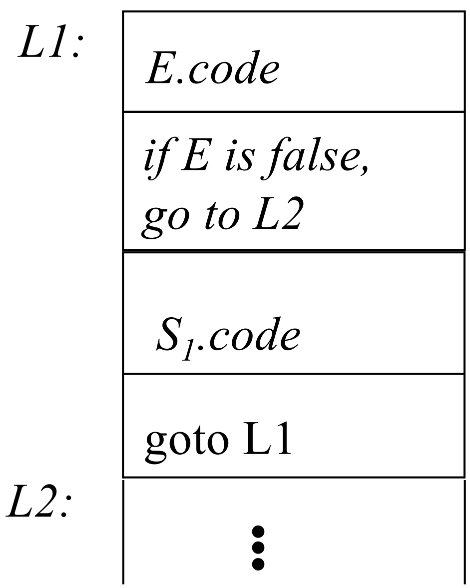
pop the top element from the operand stack and jump to the instruction with the given label if it is equal to 0

ASSUMPTION: the code generated for boolean expression *E* will leave a numeric value (0 or 1) for the expression at the top of the stack

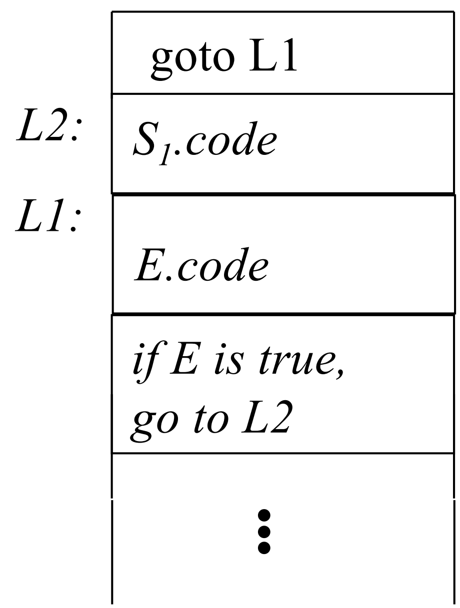
Flow-of-Control Statements: Code Structure

$S \rightarrow$ while E do S_1

- Layout 1:



- Layout 2:



Two different layouts for while statements:

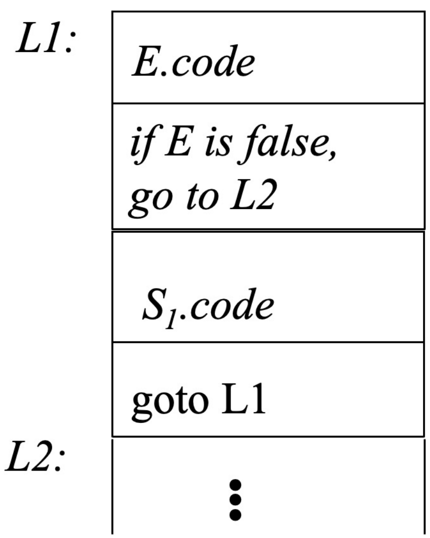
Flow-of-Control Statements, Stack-Based Code, Assuming Numeric Representation for Boolean Expressions

Attributes :	*S.code*: sequence of instructions that are generated for *S*

Productions

$S \rightarrow$ while E do S_1

- Layout 1:



Semantic Rules

Layout 1:

`L1` ← newlabel();
`L2` ← newlabel();
S.code ← gen(‘`L1 :`’) || *E.code* || gen(‘**`ifeq`** `L2`’) || S_1*.code*
|| gen(‘`goto L1`’) || gen(‘`L2 :`’) ;

Flow-of-Control Statements, Stack-Based Code, Assuming Numeric Representation for Boolean Expressions

Attributes :	*S.code*: sequence of instructions that are generated for *S*

Productions

$S \rightarrow$ while E do S_1

- Layout 2:

	goto L1
L2:	S_1*.code*
L1:	*E.code*
	if E is true, go to L2
	⋮

Semantic Rules

Layout 2:

`L1` ← newlabel();
`L2` ← newlabel();
S.code ← gen('`goto L1`') || gen('`L2 :`') || S_1*.code* ||
gen('`L1 :`') || *E.code* || gen('**`ifne`** `L2`') || gen('`goto L2`');

The code after E is the next statement, which shall be executed only when E evaluates to 0. That is why we use instruction **ifne <label>** to branch out to the code S1.

Example

Input code fragment:

```
while (a < b)  {
    if  (c < d)
        x = y + z;
    else
        x = y - z
}
```

```
L1:     load a
        load b
        if_cmplt L2
        push 0
        goto L3
L2:     push 1
L3:     ifeq LNext
        load c
        load d
        if_cmlt L4
        push 0
        goto L5
L4:     push 1
L5:     ifeq L6
        load y
        load z
        add
        store x
        goto L7
L6:     load y
        load z
        subtract
        store x
L7:     goto L1
LNext: ...
```

Optimizing the Stack Machine

- The "**add**" instruction does 3 memory operations
  - Two reads and one write to the stack
  - The top of the stack is frequently accessed
  - Idea: keep the top of the stack in a register (called accumulator) Register accesses are faster
  - The "add" instruction is now acc ← acc + top_of_stack
  - Only one memory operation!

- Key: now we have arithmetic instructions to support operands both in register and on stack. Previously, the operands must be on the stack.

Example

- Consider the expression e1 + e2.
- At a high level, the stack machine code will be:

```
cgen(e1)
push acc on the stack
cgen(e2)
add top stack element and acc, store in acc
pop one elements off the stack
```

We assume that cgen(e1) will keep the stack **invariant** before and after the execution of its code, and with the register acc keeping its evaluation results.

A Bigger Example: 3 + (7 + 5)

Code	Acc	Stack
acc ← 3	3	<init>
push acc	3	3, <init>
acc ← 7	7	3, <init>
push acc	7	7, 3, <init>
acc ← 5	5	7, 3, <init>
acc ← acc + top_of_stack	12	7, 3, <init>
pop	12	3, <init>
acc ← acc + top_of_stack	15	3, <init>
pop	15	<init>

The stack is kept **invariant** before and after the evaluation of an expression

- before and after evaluating individual numbers, e.g., 3, 5, 7
- before and after evaluating (7+5)
- before and after evaluating 3+(7+5)

From Stack Machines to Real Register-based Machines

- The compiler generates code for a stack machine with accumulator
- But mainstream processors are register-based processor（MIPS + x86）
- We simulate **stack machine instructions** using **MIPS/x86 instructions** and registers
  - This process can be rather straightforward.

Both MIPS（RISC）and x86（CISC）are register-based Architectures

- Different number of registers
  - MIPs: 32, 32-bit registers
  - x86: More registers and registers of different lengths

- Different names of registers
  - MIPS: $a0, $s0, $t0, $at, $sp
  - x86: EAX, EBX, CS, ES, ESI, EDI

- X86 allows partial reads/writes
  - MIPS: Can only write all 32 bits
  - x86: Can write to 8-, 16-, or 32-bit portions of the registers

Both MIPS and x86 use the same basic instruction types

- Instruction types: Arithmetic logic Operations, Data Movement (load and store), and Control

- Different memory access model.
  - MIPS must use load and store instructions to use operands and results in memory
  - x86 can directly perform arithmetic logic operation on main memory, for example, **x86 supports (register) ← (register + memory) format.**
  - x86 also has special pop and push instructions for stack operations.

- Different instruction lengths
  - MIPS: all instructions are 32 bits
  - x86: instructions range from 8 bits to over 32 bits

Simulating a Stack Machine on MIPS…

Stack

- The stack is kept in memory
- The stack grows towards lower addresses

Register

- The accumulator is kept in MIPS register $a0
- The address of the next location on the stack is kept in MIPS register $sp
- The top of the stack is at address $sp + 4

A Sample of MIPS Instructions

- li reg imm
  - reg ← imm
- w reg1 offset(reg2)
  - Load 32-bit word from address reg2 + offset into reg1
- sw reg1 offset(reg2)
  - Store 32-bit word in reg1 at address reg2 + offset

- add reg1 reg2 reg3
  - reg1 ← reg2 + reg3
- addiu reg1 reg2 imm
  - reg1 ← reg2 + imm
  - “u” means overflow is not checked

- Jump <label>
- beq reg1 reg2 <label>

Expression Code for MIPS

- The stack-machine code for 7+5 in MIPS:

acc <-- 7	: li $a0, 7
push acc	: sw $a0, 0(**$sp**) addiu $sp, $sp, -4
acc <-- 5	: li $a0, 5
acc < acc + top_of_stack	: lw $t1, 4($sp) add $a0, $a0, $t1
pop	: addiu $sp, $sp, 4

Good news: mostly 1-1 mapping with simple translation rules.

A Sample of x86 Instructions

- movl reg1/(memaddr1)/imm, reg2/(memaddr2)
  - Move 32-bit word from register reg1 (or address memaddr1 or the immediate value itself) into reg2 or to memory address memaddr2
  - More powerful than RISC, e.g., MIPS cannot move immediate value directly to memory

- push reg/(memaddr)/imm
  - esp <-- reg/(memaddr)/imm; **esp** <-- **esp** – 4
- pop reg/(memaddr)/imm
  - reg/(memaddr)/imm <-- **esp**; **esp** <-- **esp**+4
- push/pop are "higher-level" opcodes: enables faster execution paths for these common operations

- add reg1/(memaddr1)/imm, reg2/(memaddr2)
  - reg2/(memaddr2) <-- reg1/(memaddr1)/imm + %reg2/(memaddr2)

Simulating a Stack Machine on x86…

Stack

- The stack is kept in memory
- The stack grows towards lower addresses

Register

- The accumulator is kept in x86 register **eax**
- The address of the next location on the stack is kept in x86 register **esp**

Expression Code for x86

- The stack-machine code for 7+5 in x86:

acc <-- 7	: movl 7,eax
push acc	: pushl eax
acc <-- 5	: movl 5, eax
acc < acc + top_of_stack	: addl (%esp), eax
pop	: pop %ecx

#just pop the top of the stack to an unused register $ecx

Again: mostly 1-1 mapping with simple translation rules.

How about Functions?

- Program for computing the Fibonacci numbers:

```
def fib(x) = if x = 1 then 0 else
                            if x = 2 then 1 else
                                          fib(x - 1) + fib(x – 2)
```

Extend our CFG:

S → D
D → def id(ARGS) = E;
ARGS → id, ARGS | id
E → if E1 then E2 else E3
E → id(E1,…,En)

Our Simple Language

Step1: Language only with Arithmetic Expressions

$S \rightarrow \text{id} := E$
$S \rightarrow S_1 \; ; \; S_2$
$E \rightarrow E_1 \; + \; E_2$
$E \rightarrow E_1 \; * \; E_2$
$E \rightarrow (\, E_1 \,)$
$E \rightarrow - \, E_1$
$E \rightarrow \text{id}$
$E \rightarrow \text{num}$

Step2: Extended with control statements and boolean expressions

$S \rightarrow \text{if } \; \text{E then } S_1 \text{ else } S_2$
$S \rightarrow \text{while } E \text{ do } S_1$
$E \rightarrow E_1 \; \text{relop} \; E_2$
$E \rightarrow E_1 \; \text{and } E_2$
$E \rightarrow E_1 \; \text{or } E_2$
$E \rightarrow \text{not } E_1$
$E \rightarrow \text{true}$
$E \rightarrow \text{false}$

Step3: Extended with functions

$\text{S} \rightarrow \text{D}$
$\text{D} \rightarrow \text{def id(ARGS) = E;}$
$\text{ARGS} \rightarrow \text{id, ARGS} \mid \text{id}$
$\text{E} \rightarrow \text{if E1 then E2 else E3}$
$\text{E} \rightarrow \text{id(E1,\ldots,En)}$

The Activation Record (Lecture 12)

- For this language, an AR with the actual parameters, the return address suffices.
  - Actual parameters are the only variables in this language
  - For f(x1,…,xn), push xn,…,x1 on the stack
  - We need the return address, which points to the caller's next instruction (**code**) to be executed
  - The computation result is always in the accumulator
  - No need to store the result in the AR
  - We do not need to keep the control link as we can keep $sp the same on function exit as it was on function entry (special **invariant** property of stack machine code).
    - Note that control link is pointing to the top of caller's activation record (**data** and related information) on the stack.

Actual parameters	Yes
Returned values	Yes
Control link	No
Access link	No
Saved machine status	No
Local data	No
Temporaries	

Yes, and it is automatic as part of the stack computation

Dedicated Registers (Targeting MIPS)

- Note: We have three **dedicated** registers $pc, $fp, $sp
  - $fp: frame pointer
  - $sp: stack pointer
  - $pc: next instruction to execute

- They are used to support function implementation, in addition to the accumulator register $a0. They makes the generate code much more efficient.
  - Reason for these two pointers will be clear shortly with examples.

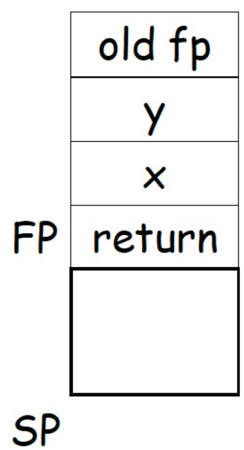
- Note that for stack machine code, we use registers for dedicated usage. There is no need for *register allocation* optimization in a register-based machine code generation.

Why have $fp pointed to the "return Address"?

- Because the stack grows when intermediate results are saved, the variables are not at a fixed offset from $sp

- $fp makes code generation for local variables much easier.
- Let xi be the ith (i = 1,…,n) formal parameter of the function for which code is being generated
  - cgen(xi) = lw $a0 z($fp)   (z = 4*i)

- Example: For a function def f(x,y) = e, the activation and frame pointer are set up as follows:



- X is at fp + 4
- Y is at fp + 8

Code Generation for Function Call

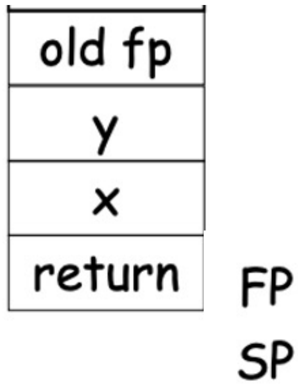
cgen(f(e_1,…,e_n)) =

This is a special implementation of pre-call (lecture 12)

```
sw $fp 0($sp)
addiu $sp $sp -4

cgen(e_n)
sw $a0 0($sp)
addiu $sp $sp -4
…
cgen(e_1)
sw $a0 0($sp)
addiu $sp $sp -4

sw $pc 0($sp)

mv $fp, $sp
addiu $sp $sp-4
jump f_entry
```

- The caller saves its value of the frame pointer $fp on stack
- Then it saves each of its actual parameters on the stack
- The caller saves the return address in register $pc
- The AR so far is 4*n+ 8 bytes long
- f_entry points to the code for the definition of function f.

Activation Record



Code Generation for Function Call

cgen(def f(x1,…,xn) = e)
=

The core part of this is a special implementation of epilogue (lecture 12)

```
cgen(e)

lw $pc ($fp)

addiu $sp $sp z

 lw $fp 0($sp)

jump $pc
```

- We first restore the return address to $pc.
  - This is important as e may included a function call in its body.
- We then popping out the return address, the actual arguments and the saved value of the frame pointer.
  - sp (caller) = sp (callee) + z

    where z = 4*n + 8
- We restore the old $fp, which is stored on bottom of callee's stack
- Return the control to the caller

Summary

- Code generation can be done by ad-hoc syntax directed translation
  – recursive traversal of the AST
- Stack machine code is easy to generate.
  – When dealing with functions, the activation record must be designed together with the code generator
- Stack machine code can also be simulated on register-based machines.