Run-time Systems

Run-time Systems

- We have completed the entire front-end of a compiler:
  - Scanning
  - Parsing
  - Semantic analysis

- These stages depend only on the properties of the source language.

- They are completely independent of :
  - The target (machine or assembly) language
  - The properties of the target machine
  - Operating system

Run-time Systems

- The front-end:
  - Enforces the language definition
  - Builds data structures that are needed to do code generation

- If the front-end has not generated any error:
  - We have a valid program in the source language that we are compiling
  - We are ready to produce code which is a valid translation of the source code and that can executed on a given target architecture

Run-time Systems

- The Back-end:
    - Optimization
    - Code generation
- Runtime Systems:
    - What the target program looks like and how is it organized. Why?
        - We have to know what we need to generate before knowing how we generate it and how such a generation strategy makes sense.
-

Run-time Support

- The target program interacts with system resources.
- There is a need to manage memory when a program is running
  - This memory management must connect to the data objects of programs
  - Programs request for memory blocks and release memory blocks
  - Passing parameters to functions needs attention
- Other resources such as printers, file systems, etc., also need to be accessed

Runtime Support

- Execution of a program is initially under the control of the

  operating system

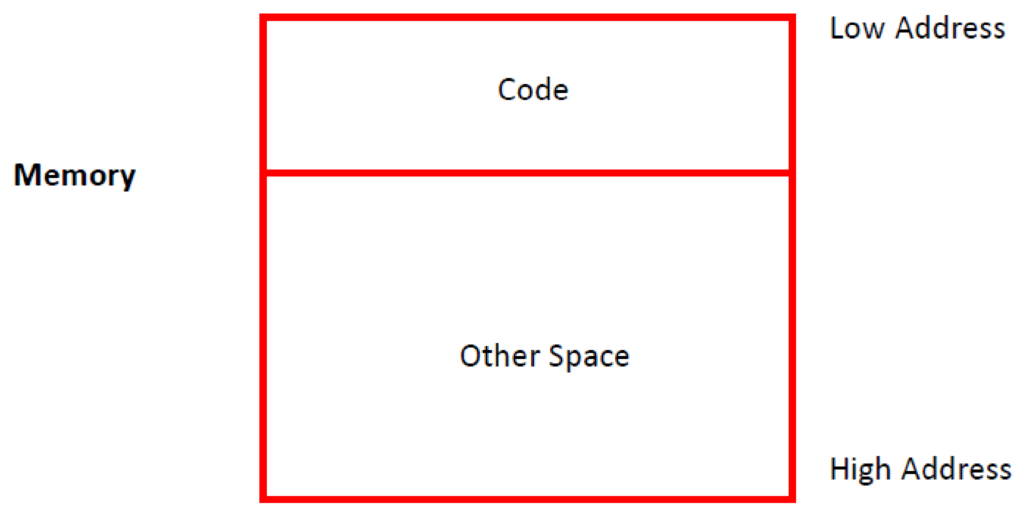
- When a program is invoked:
  - The OS allocates space for the program
  - The code is loaded into part of the space
  - The OS jumps to the entry point (i.e., “main”)

Management of Run-time Resources

- The compiler is not only responsible for generating code but also handling the associated data

- Compiler needs to decide what the layout of data is going to be and then generate code that correctly manipulates the data
  - References data within the code
  - Code and layout of data needs to be designed together

- Storage Organization



Procedure Activations

- Two goals in code generation:
  - Correctness
  - Speed

- Fast as well as correct – Difficult

- Two assumptions of Activation:
  - Execution is sequential; control moves from one point in a program to another in a well-defined order
  - When a procedure is called, control always returns to the point immediately after the call

Procedure Activations

- An invocation of procedure **P** is an *activation of **P***

- The *lifetime of an activation of **P*** is
  - All the steps to execute **P**
  - Including all the steps in procedures **P** calls

- The *lifetime of a variable x* is the portion of execution in which x is defined

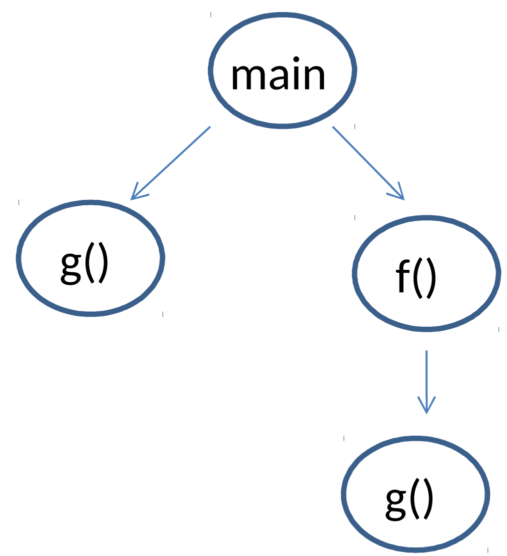
- Note that
  - Lifetime is a dynamic (run-time) concept
  - Scope is a static concept

Procedure Activations

- Observation
  - When **P** calls **Q**, then **Q** returns before **P** returns

- Lifetimes of procedure activations are properly nested
- Activation lifetimes can be depicted as a tree

```
int main {
      int g() { return 1; };
      int f()  { return g(); };
      int main() {{ g(); f(); }};
}
```



Procedure Activations

```
class main {
    int g() : { return1; };
    int f(int x) { if (x == 0) then return g(); else
                                      return 1 + f(x - 1); }
    int main ( )  { f(3); };
}
```

Nesting of activations

- The activation tree may be different for every program input
- Since activations are properly nested, a stack can track currently active procedures

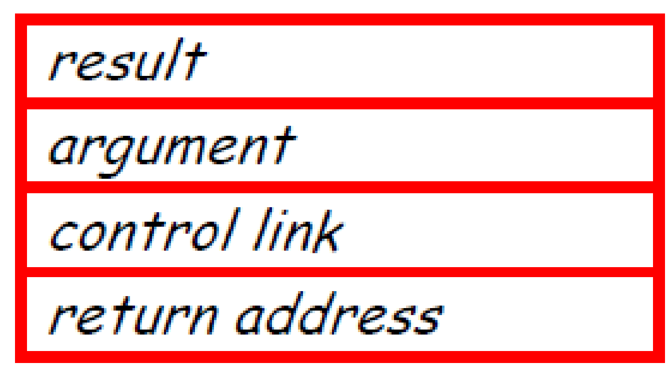
Activation Records

- Information needed to manage one procedure activation is called an *activation record (AR) or frame*
- If procedure **F** calls **G**, then **G**'s activation record contains a mix of info about **F** and **G**.
- **F** is "suspended" until **G** completes, at which point **F** resumes
- **G**'s AR contains information needed to
  - Complete execution of **G**
  - Resume execution of **F**

Activation Records

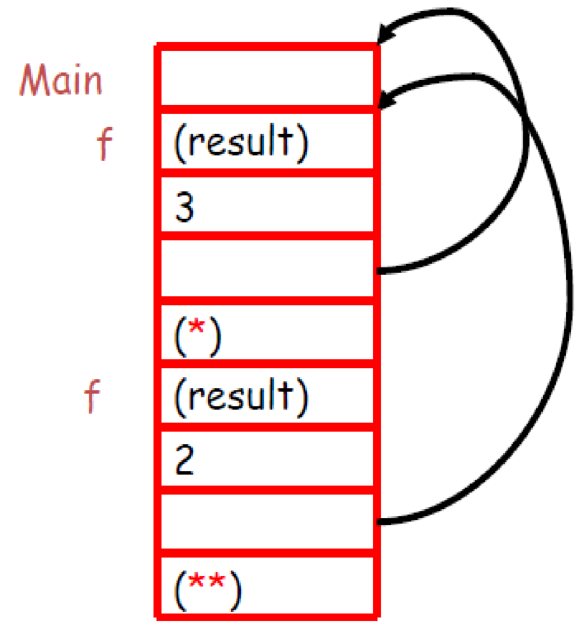
- Space for **G**'s return value
- Actual parameters
- Pointer to the previous activation record
  - The *control link; points to AR of caller of* ***G***
- Machine status prior to calling **G**
  - Contents of registers & program counter
  - Local variables
- Other temporary values

AR



Activation Records

```
class main {
      int g() : { return1; };
      int f(int x) { if (x == 0) then return g();
                     else return 1+ f(x - 1);(**) };
      int main()  {f(3); (*)};
}
```



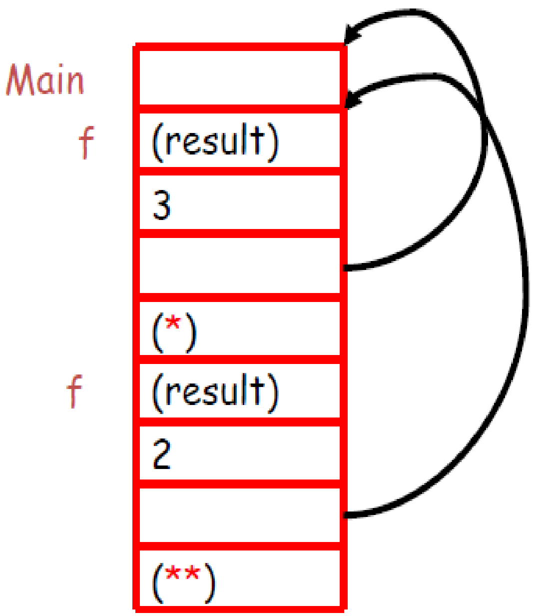
- AR:

result
argument
control link
return address

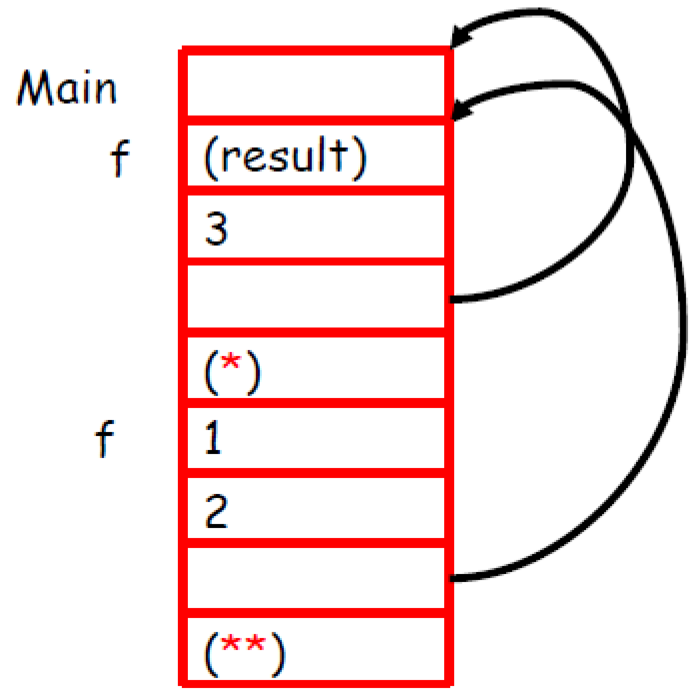
- **main()** has no argument or local variables and its result is never used; its AR is uninteresting
- (*) and (**) are return addresses of the invocations of **f**

Activation Records

- The advantage of placing the return value 1st in a frame is that the caller can find it at a fixed offset from its own frame



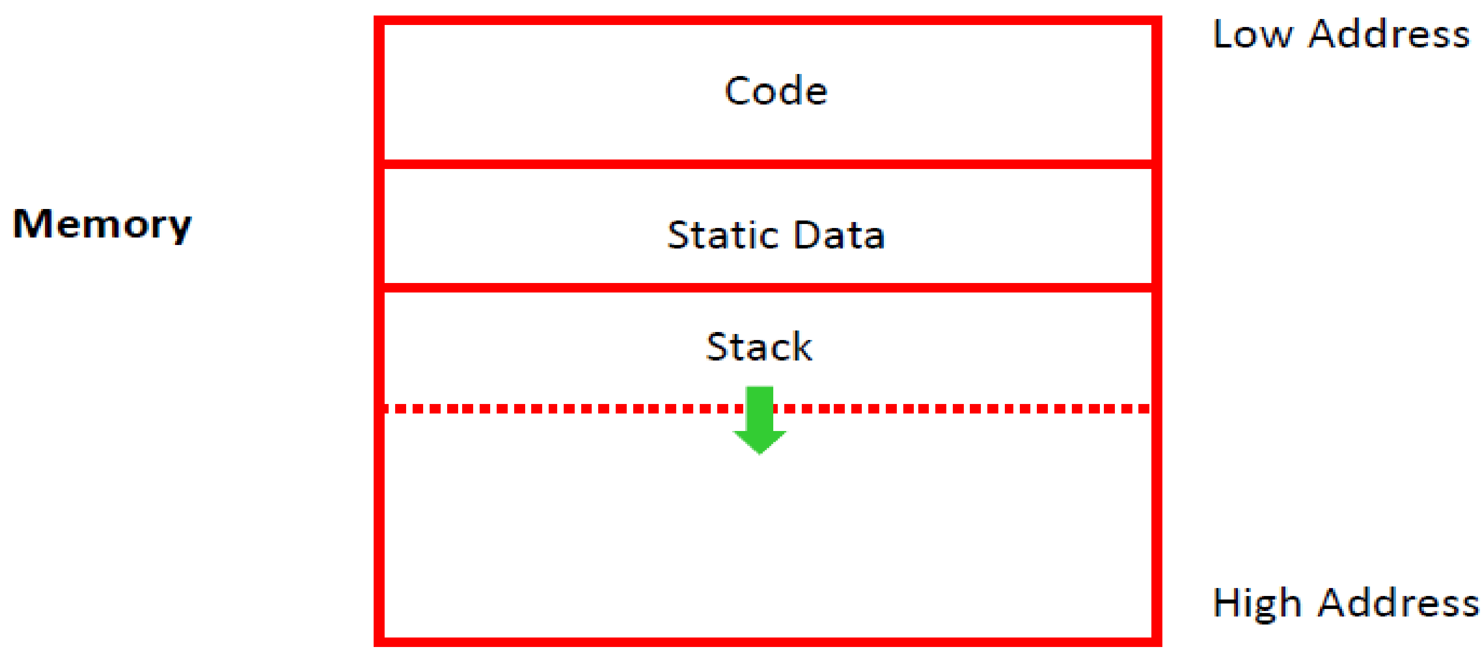
Stack state after the call to the 2nd invocation of **f** returns



Global Data

- All references to a global variable point to the same object
  - Can’t store a global in an activation record

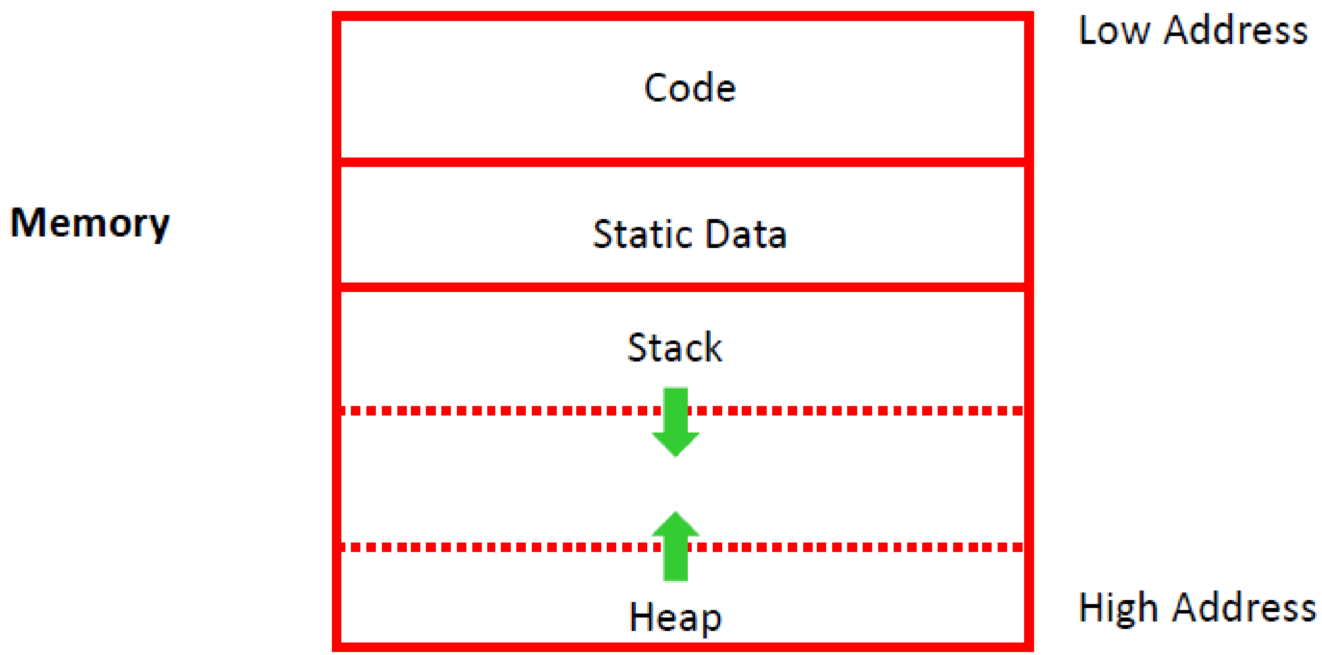
- Globals are assigned a fixed address statically



Heap

- A value that outlives the procedure that creates it cannot be kept in AR
    - method foo() { new Bar }
    - The Bar value must survive deallocation of foo's AR

- A *heap* is generally used to store dynamically allocated data



Alignment

- Most modern machines are 32 or 64 bit
- Machines are either byte or word addressable
- Data is *word aligned if it begins at a word boundary*
- Machines generally have alignment restrictions
  - Accessing mis-aligned data incurs significant overhead – say 5x slower

- *Padding* is used to word align next data object in memory
  - Most frequently used with strings
  - Say we have the string "Hello" – requires 2 "padding" characters

Code Generation

Stack Machines

- Only storage is a stack

- An instruction $r = F(a_1,\ldots a_n)$:
  - Pops **n** operands from the stack
  - Computes the operation **F** using the operands
  - Pushes the result **r** on the stack

- Consider two instructions:
  - push i - push integer i on the stack
  - add - add two integers

Stack Machines

- A program:
  - push 7
  - push 5
  - Add

- Stack machines provide a simple machine model
  - Simple compiler
  - Inefficient

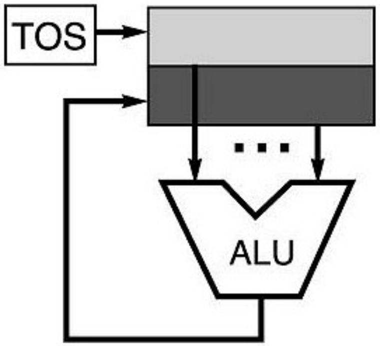
- Location of the operands/result is not explicitly stated
  - Always the top of the stack

Stack Architectures

- Instruction set:

  add, sub, mult, div, . . .

  push A, pop A



- Example: A*B - (A+C*B)

  push A

  push B

  mul

  push A

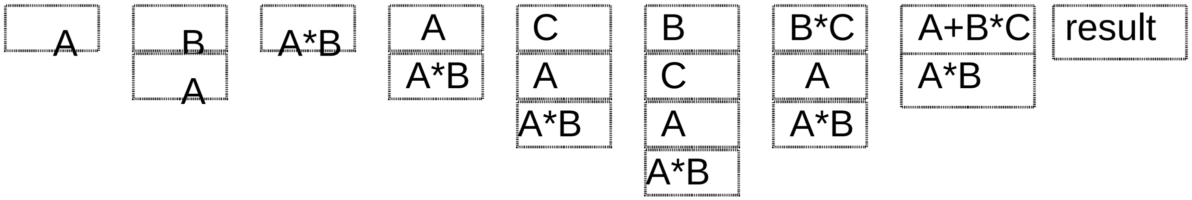
  push C

  push B

  mul

  add

  sub



Stack Machines

- Stack machine Vs. register machine
  - **add** instead of **add r_1, r_2, r_3**
  - More compact programs
- There is an intermediate point between a pure stack machine and a pure register machine
  - An *n-register stack machine*
  - Conceptually, keep the top *n locations* of the pure stack machine's stack in registers
- 1-register stack machine
  - The register is called the *accumulator*

Stack Machines

- In a pure stack machine
  - An **add** does **3** memory operations: Two reads and one write
- In a 1-register stack machine the **add** does
  - acc $\leftarrow$ acc + top_of_stack
- In general, for an operation $\mathbf{op(e_1,\ldots,e_n)}$
  - $\mathbf{e_1,\ldots,e_n}$ are subexpressions
- For each $\mathbf{e_i}$ $\mathbf{(0 < i < n)}$
  - Compute $\mathbf{e_i}$
  - Push result on the stack
- Pop **n-1** values from the stack, compute **op**
- Store result in the accumulator

Stack Machines

Operations for the stack machine with accumulator: $3 + (7 + 5)$

Code	Accumulator	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>

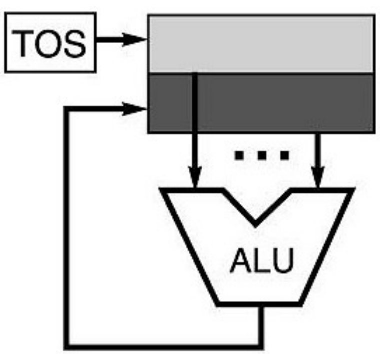
Stacks: Pros and Cons

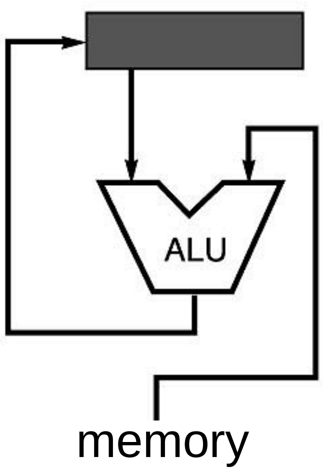
- Pros
  - Good code density (implicit top of stack)
  - Low hardware requirements
  - Easy to write a simpler compiler for stack architectures

- Cons
  - Stack becomes the bottleneck
  - Little ability for parallelism or pipelining
  - Data is not always at the top of stack when need, so additional instructions like TOP and SWAP are needed
  - Difficult to write an optimizing compiler for stack architectures

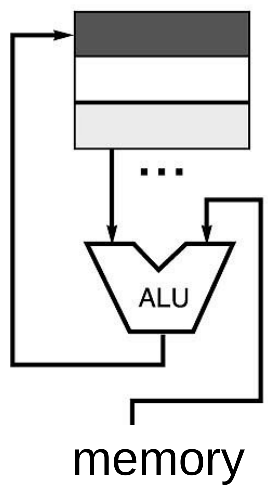
Code Sequence C = A + B for Four Instruction Sets

Stack	Accumulator (Stack with 1 reg.)	Register (register-memory)	Register (load-store)
Push A **Push B** **Add** **Pop (to) C**	**PUSH A** **Add B** **Pop (to) C**	**Load R1, A** **Add R1, B** **Store C, R1**	**Load R1,A** **Load R2, B** **Add R3, R1, R2** **Store C, R3**

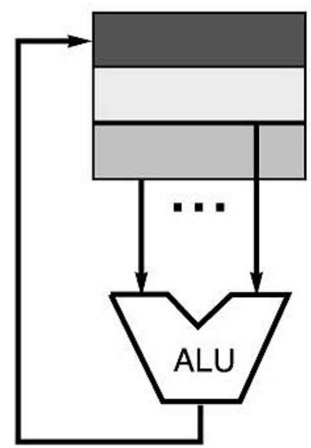




acc = acc + mem[C]



R1 = R1 + mem[C]



R3 = R1 + R2

Code Generation

… to continue

Code Generation

- Code that can be executed on a real machine
  - The MIPS processor
- We will simulate a stack machine model using MIPS instructions and registers
- The accumulator is kept in MIPS register $a0

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

- Address of the next location on stack is kept in register $sp
  - Top of the stack is at address $sp + 4

- MIPS uses RISC processor model
- 32 general purpose registers (32 bits each)
- We use $sp (stack pointer), $a0 (accumulator) and $t1 (a temporary register)

MIPS (originally an acronym for Microprocessor without Interlocked Pipeline Stages) is a reduced instruction set computer (RISC) instruction set architecture (ISA) developed by MIPS Technologies

MIPS Instructions

- lw reg1 offset(reg2) /*load word*/
  - Load 32-bit word from address reg_2 + offset into reg_1

- add reg_1 reg_2 reg_3
  - $reg_1 \leftarrow reg_2 + reg_3$

- sw reg1 offset(reg_2) /*store word*/
  - Store 32-bit word in reg_1 at address reg_2 + offset

- addiu reg_1 reg_2 imm (#) /*add immediate unsigned*/
  - $reg_1 \leftarrow reg_2 + imm$ where # □ a number
  - "u" means overflow is not checked

- li reg imm (#) /*load immediate*/
  - $reg \leftarrow imm$

Example

- 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`

Code Generation

- A language with integers and integer operations

$P \rightarrow D; P \mid D$

$D \rightarrow$ def id(ARGS) = E;

$ARGS \rightarrow$ id, ARGS | id

$E \rightarrow$ int | id | if $E_1 = E_2$ then E_3 else E_4

| $E_1 + E_2$ | $E_1 - E_2$ | id(E_1, …, E_n)

- A program consists of a list of declarations

- A declaration is a function definition.
- The function takes a list of identifiers as arguments.
- The function body is an expression.

- Expressions are integers, identifiers, if-then-else with a predicate which allows the equality test, sums and differences of expressions and function calls.

- The first function definition in the list is the entry point, that is the *main* routine.

Code Generation

- This language may be used to define the fibonacci function:

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

- To generate code for this language, we generate MIPS code for each expression **e** that:
    - Computes the value of **e** in **$a0**
    - Preserves **$sp** and the contents of the stack
- We define a code generation function **cgen(e)** whose result is the code generated for **e**

Code Generation

- **cgen(e)** is going to work by cases.

- The code to evaluate a constant simply copies it into the accumulator:
  - cgen(i) = li $a0 i

- *This preserves the stack, as required*

- cgen(e_1 + e_2) =

```
 cgen(e1)
 sw $a0 0($sp)
 addiu $sp $sp - 4
 cgen(e2)
 lw $t1 4($sp)
 add $a0 $t1 $a0
 addiu $sp $sp 4
```

- The code for + is a template with "holes" for code for evaluating e_1 and e_2
- Stack machine code generation is recursive
- Code generation for expressions can be done as a recursive-descent of the AST

Code Generation

- MIPS instruction: `sub` reg_1 reg_2 reg_3
  - Implements $reg_1 \leftarrow reg_2 - reg_3$

```
cgen(e1 - e2) =
          cgen(e1)
          sw $a0 0($sp)
          addiu $sp $sp - 4
          cgen(e2)
          lw $t1 4($sp)
          sub $a0 $t1 $a0
          addiu $sp $sp 4
```

Code Generation

- Write MIPS assembly code for the given expressions following the 1-register stack machine model:
    - 1 + (2 - 3)
    - (5 - 4) + 3

Code Generation

… to continue

Condition Checking

- MIPS instruction: `beq` reg_1 reg_2 `label`
    - Branch to label if $reg_1 = reg_2$

- MIPS instruction: `b label`
    - Unconditional branch to label

cgen(if e_1 = e_2 then e_3 else e_4) =

```
      cgen(e1)
      sw $a0 0($sp)
      addiu $sp $sp - 4
      cgen(e2)
      lw $t1 4($sp)
      addiu $sp $sp 4
      beq $a0 $t1 True_branch
```

```
False_branch:
   cgen(e4)
   b end_if
True_branch:
   cgen(e3)
End_if:
```

Code Generation

- A language with integers and integer operations

$$P \rightarrow D; P \mid D$$

$$D \rightarrow \text{def } id(ARGS) = E;$$

$$ARGS \rightarrow id, ARGS \mid id$$

$$E \rightarrow int \mid id \mid \text{if } E_1 = E_2 \text{ then } E_3 \text{ else } E_4$$

$$\mid E_1 + E_2 \mid E_1 - E_2 \mid id(E_1, \ldots, E_n)$$

Function Calls

- Code for function calls and function definitions depends on the layout of the AR

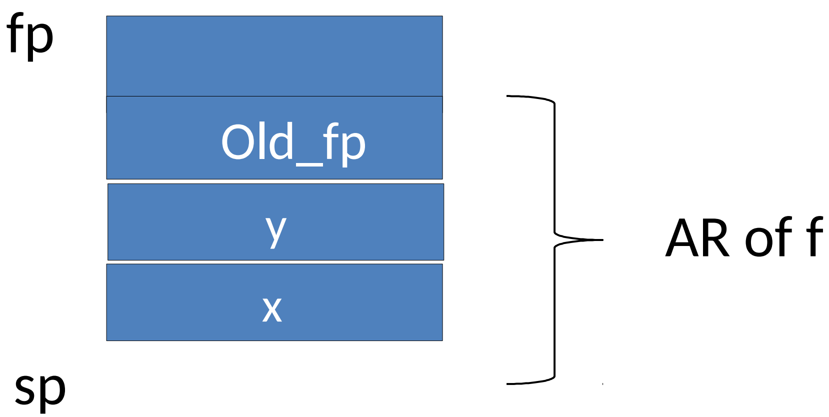
- A very simple AR suffices for this language:
  - The result is always in the accumulator
  - No need to store the result in the AR
  - The activation record holds actual parameters
    - For $f(x_1,\ldots,x_n)$ push $x_n,\ldots,x_1$ on the stack
    - These are the only variables (no other local or global variables) in this language (constraint)
  - The stack discipline guarantees that on function exit $sp is the same as it was on function entry (preservation of the SP)
    - No need for a control link

Function Calls

- A pointer to the current activation is useful
  - This pointer lives in register $fp (frame pointer)

- So, for this language, an AR with the caller's frame pointer, the actual parameters, and the return address suffices

- Consider a call to f(x,y), the AR is:



Function Calls

- The calling sequence is the instructions (of both caller and callee) to set up a function invocation

- New MIPS instruction: jal label /*jump and link*/
  - Jump to label, save address of next instruction in $ra
  - To be used in Caller

- New MIPS instruction: jr reg
  - Jump to address in register reg
  - To be used in Callee

Function Calls

Code in Caller

```
cgen(f(e1,…,en)) =
      sw $fp 0($sp)
      addiu $sp $sp - 4
      cgen(en)
      sw $a0 0($sp)
      addiu $sp $sp - 4
      …
      cgen(e1)
      sw $a0 0($sp)
      addiu $sp $sp - 4
      jal f_entry
```

Code in the caller side

- The caller saves its fp to stack
- Then it saves the actual parameters in reverse order
- Execute jump and link to call
- Finally the caller saves the return address in register $ra
- The AR so far is 4*n+4 bytes long

Code Generation – Function Calls

Code in Callee

```
cgen(def f(x1,…,xn)= e) =

F_entry: move $fp $sp
     sw $ra 0($sp)
     addiu $sp $sp - 4
     cgen(e)
         lw $ra 4($sp)
     addiu $sp $sp z
     lw $fp 0($sp)
     jr $ra
```

- Store the $sp as the $fp of the current function.
- Store the return address ($ra) to stack
- Execute expression e
- Pop return address
- Store the current fp to $fp
- Jump to $ra to return calling function

- The frame pointer points to the top, not the bottom
- The callee pops the return address, the actual arguments and the saved value of the frame pointer
- z = 4*n + 8

Function Calls

- The "variables" of a function are just its parameters
  - They are all in the AR
  - Pushed by the caller

- Problem: Because the stack grows when intermediate results are saved, the variables are not at a fixed offset from $**sp**
- Solution: use a frame pointer
  - Always points to the return address on the stack

- Let x_i be the i^{th} ($i = 1,\ldots,n$) formal parameter of the function for which code is being generated
  - $cgen(x_i)$ = lw $a0 z($fp) (z = 4*i)

Handling Temporaries

Code Generation

- In production compilers:
  - Emphasis is on keeping values in registers
    - Especially the current stack frame
  - Intermediate results are laid out in the AR, not pushed and popped from the stack

Code Generation – Handling Temporaries

- Let $NT(e)$ = Number of temporaries needed to evaluate e
- $NT(e_1 + e_2)$
    - Needs at least as many temporaries as $NT(e_1)$
    - Needs at least as many temporaries as $NT(e_2) + 1$
- Space used for temporaries in e_1 can be reused for temporaries in e_2

- $NT(e_1 + e_2) = max(NT(e_1), 1 + NT(e_2))$

- $NT(\text{if } e_1 = e_2 \text{ then } e_3 \text{ else } e_4) = max(NT(e_1), 1 + NT(e_2), NT(e_3), NT(e_4))$

- $NT(id(e_1,\ldots,e_n)) = max(NT(e_1),\ldots,NT(e_n))$

- $NT(int / id) = 0$

Code Generation

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

2 Temporary variables required

- For a function definition $f(x_1,\ldots, x_n) = e$ the AR has 2 + n + NT(e) elements
  - Return address (1)
  - Frame pointer (1)
  - n arguments (n)
  - NT(e) locations for intermediate results (NT(e))

Old_fp
X_n
…
X_1
Return Address
Temp NT(e)
…
Temp 1

Code Generation

- Code generation must know how many temporaries are in use at each point
- Add a new argument to code generation
  - The position of the next available temporary
- The temporary area is used like a small, fixed-size stack

```
cgen(e1 + e2) =
    cgen(e1)
    sw $a0 0($sp)
    addiu $sp $sp - 4
    cgen(e2)
    lw $t1 4($sp)
    add $a0 $t1 $a0
    addiu $sp $sp 4
```

```
cgen(e1 + e2, nt) =
     cgen(e1, nt)
   sw $a0 nt($fp)
   cgen(e2, nt + 4)
   lw $t1 nt($fp)
   add $a0 $t1 $a0
```

Code Generation Example

```
def sumto(x) = if x = 0
                   then 0
               else
                   x + sumto(x – 1)
```

Code Optimization

Code Optimization

- Most complexity in modern compilers is in the optimizer
  - Also by far the largest phase

- Optimizations can be performed on
  - AST / DAG
    - Machine independent optimization but too high level
  - Assembly code
    - Target (machine) dependent optimization
  - On an intermediate language

Basic Blocks

- A basic block is a maximal sequence of instructions with:
  - no labels (except at the first instruction), and
  - no jumps (except in the last instruction)

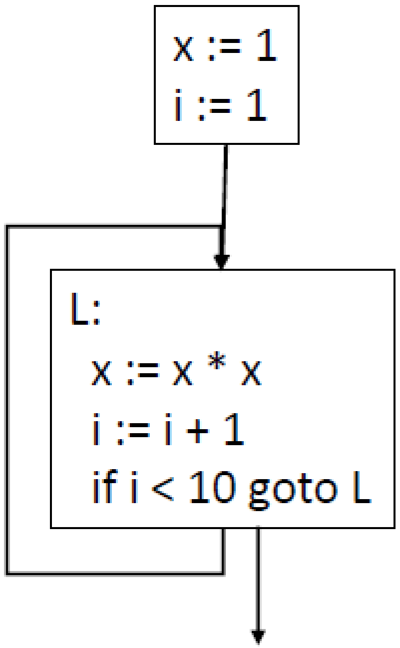
- Idea:
  - Cannot jump into a basic block (except at beginning)
  - Cannot jump out of a basic block (except at end)
  - A basic block is a single-entry, single-exit, straight-line code segment

- The property of sequential control flow can be useful for many optimizations.

Control Flow Graphs

- Control-Flow Graph (CFG)
  - Models the way that the code transfers control between blocks in the procedure.

    - Node: a single basic block
    - Edge: transfer of control between basic blocks.
    - All return nodes are terminal



Code Optimization

- Optimization seeks to improve a program's resource utilization
  - Execution time (most often)
  - Code size
  - Network messages sent
  - Memory Usages
  - Disk Accesses
  - Power
- Optimization should not alter what the program computes
  - The answer before and after optimization must remain same

Code Optimization

- There are three granularities of optimizations
  - Local optimizations
    - Apply to a basic block in isolation
  - Global optimizations
    - Apply to a control-flow graph (method body) in isolation
  - Inter-procedural optimizations
    - Apply across method boundaries

- Most compilers do local and global optimizations but not the third

- Often a conscious decision is made not to implement the fanciest optimization known
  - Goal: Maximum benefit for minimum cost

Local Optimization

- Algebraic simplification and Reassociation
  - Simplifications use algebraic properties or particular operator-operand combinations to simplify expressions.
  - Reassociation refers to using properties such as associativity, commutativity and distributivity to rearrange an expression to enable other optimizations

Local Optimization

- Algebraic simplification and Reassociation
  - Simplification Examples:

- x+0 = x   0+x = x   x*1 = x   1*x = x   0/x = 0   x-0 = x
- b && true = b   b && false = false
- b || true = true   b || false = b

Example: Re-arrangement + constant folding

```
b = 5 + a + 10 ;
```

```
t0 = 5 ;
t1 = t0 + a ;
t2 = t1 + 10 ;
b = t2 ;
```

```
t0 = 15 ;
t1 = a + t0 ;
b = t1 ;
```

Local Optimization

- Constant Folding
    - Evaluation at compile-time of expressions whose operands are known to be constant
  - Example

a = 10 * 5 + 6 - b;

t0 = 10 ;
t1 = 5 ;
t2 = t0 * t1 ;
t3 = 6 ;
t4 = t2 + t3 ;
t5 = t4 – b;
a = t5 ;

t0 = 56 ;
t1 = t0 – b ;
a = t1 ;

Code Optimization

…. to continue

Local Optimization

- Constant Propagation
  - If a variable is assigned a constant value, then subsequent uses of that variable can be replaced by the constant as long as no intervening assignment has changed the value of the variable.
  - Example:

```
t0 = 12 ;
t1 = arr + t0 ;
t2 = *(t1) ;
```

```
li $t0, 12
lw $t1, 8($fp)
add $t2, $t1, $t0
lw $t3, 0($t2)
```

Constant propagation + rearrangement cuts no. of regs. and insns. from 4 to 2

```
t0 = *(arr + 12) ;
```

```
lw $t0, -8($fp)
lw $t1, 12($t0)
```

Local Optimization

- Operator Strength Reduction
  - Replaces an operator by a "less expensive" one
  - Often performed as part of *loop-induction variable elimination*

- Example:

```
while (i < 100)
{
  arr[i] = 0;
  i = i + 1;
}
```

```
t1 = i;
L0:
If  t1>100 Goto L1 ;
t2 = 4 * t1 ;
t3 = arr + t2 ;
*(t3) = 0 ;
t1 = t1 + 1 ;
Jump L0
L1:
```

```
t1 = arr ;
L0: If  i > 100  Goto L1 ;
* t1 = 0;
t1 = t1 + 4;
i = i + 1 ;
L1:
```

Local Optimization

- Copy Propagation
  - Similar to constant propagation, but generalized to non-constant values
  - For a = b, we can replace later occurrences of a with b (assuming there are no changes to either variable in-between)
  - Example

```
t2 = t1 ;
t3 = t2 * t1;
t4 = t3 ;
t5 = t3 * t2 ;
c = t5 + t4 ;
```

```
t3 = t1 * t1 ;
t5 = t3 * t1 ;
c = t5 + t3 ;
```