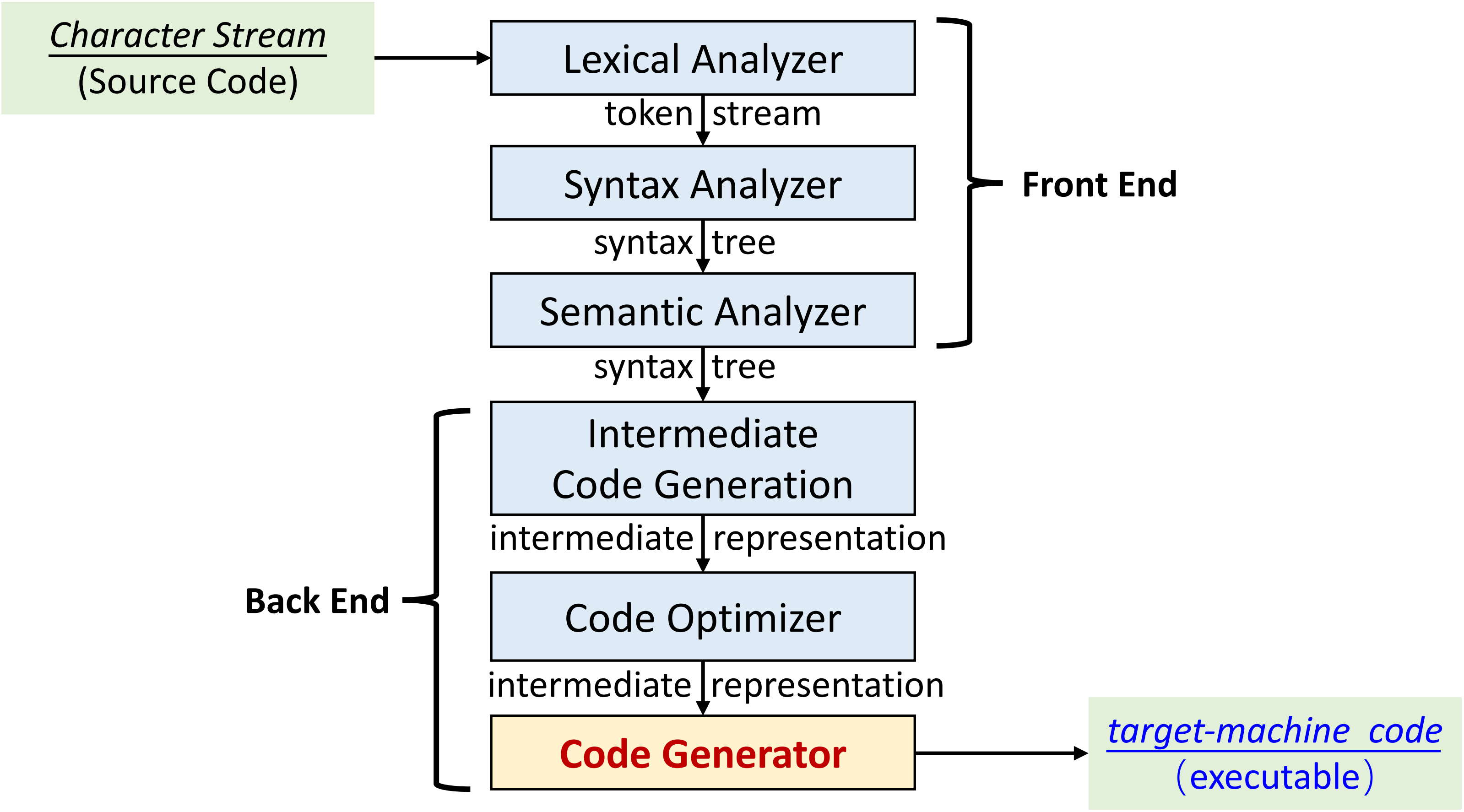
编译原理
Complier Principles

Lecture 9
Target Code Generation

赵帅

计算机学院
中山大学

Compilation Phases[编译阶段]



Target Code Generation[目标代码生成]

- Target code generation
  - Transform the syntactically analyzed or optimized intermediate code into target code.
- The main issues we need to consider:
  - How to make the target code shorter?
  - How to make full use of the registers and reduce the number of accesses to storage units in the target code ?
  - How to make full use of characteristics of the computer's instruction system ?

Primary Tasks [主要任务]

- What we have now ?
  - IR of the source program
  - Symbol table
- Three primary tasks:
  - Instruction selection[指令选取]
    - Choose appropriate target-machine instructions to implement the IR statements
  - Register allocation and assignment[寄存器分配]
    - decide what values to keep in which register
  - Instruction ordering[指令排序]
    - Decide in what order to schedule the execution of instructions

Instruction Selection[指令选取]

- **Instruction selection** is the stage of a compiler backend that transforms intermediate representation (IR) into a low-level IR.
  - contains both instruction scheduling and register allocation.
  - Its output IR may still be subject to peephole optimization[窥孔优化].

IR code (TAC):

```
a = b + c;
d = a + e;
```

Target code:

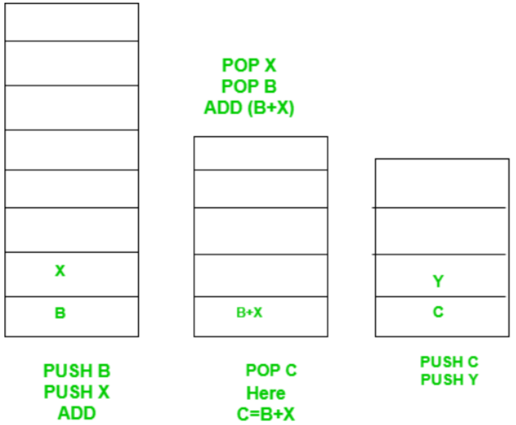
```
LD R0, b          // R0 = b
ADD R0, R0, c     // R0 = R0 + c
ST a, R0          // a = R0
LD R0, a          // R0 = a
ADD R0, R0, e     // R0 = R0 + e
ST d, R0          // d = R0
```

Register Allocation & Inst. Order

- **Register allocation**: the process of assigning local variables and expression results to processor registers[寄存器分配].
  - Registers are the fastest storage unit but are of limited numbers
    - Values not held in registers need to reside in memory
    - Instructions involving register operands are much shorter and faster
  - Many allocations, including register allocation, are NP problems.
- **Instruction order**: the order in which computations are performed can affect the efficiency of the target code[执行顺序]
  - Some computation orders require fewer registers to hold intermediate results than others
  - However, picking a best order in the general case is NP-hard

Stack Machine [栈式计算机]

- A simple evaluation model:
  - No variables or registers
  - A stack of values for intermediate results
- Each instruction:
  - Push operands to the stack[讲操作数压入栈中]
  - Takes its operands from the top of the stack[栈顶取操作数]
  - Removes those operands from the stack[从栈中移除操作数]
  - Computes the required operation on them[计算]
  - Pushes the result on the stack[将计算结果入栈]



Optimize the Stack Machine

- Note that the add instruction does 3 memory operations
  - two reads and one write.
- The top of the stack is frequently accessed.
- Idea: keep the top of the stack in a register (called accumulator) [使用寄存器]
- The “add” instruction is now
  - ◆ acc ← acc + top_of_stack
  - ◆ Only one memory operation

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>

From Stack Machine to MIPS

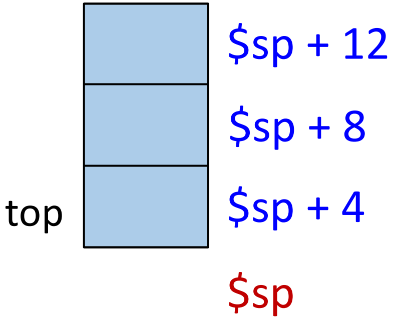
- The compiler generates code for a stack machine with accumulator.

- We want to run the resulting code on the MIPS processor

- We simulate stack machine instructions using MIPS instructions and registers

Simulating a Stack Machine

- The ACC is kept in MIPS register $t0
- The stack is kept in memory
- 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
- The stack grows towards lower addresses
    - Standard convention on the MIPS architecture



MIPS Architecture

- Prototypical Reduced Instruction Set Computer (RISC) architecture
- Arithmetic operations use registers for operands and results
- Must use load and store instructions to use operands and results in memory
  - All other instructions access only registers
- 32 general purpose registers (32 bits each)
  - We will use $t0 (ACC), $sp (address of the next location on the stack), and $t1 (a temporary register)
  - 31 of these are general-purpose that can be used in any of the instructions
  - The last one (zero), is to contain the number zero at all times

A Sample of MIPS Instructions

- lw reg1 offset(reg2)
  - Load 32-bit word from address reg2 + offset into reg1
- add reg1 reg2 reg3
  - reg1 ← reg2 + reg3
- sw reg1 offset(reg2)
  - Store 32-bit word in reg1 at address reg2 + offset
- addiu reg1 reg2 imm
  - reg1 ← reg2 + imm
  - "u" means overflow is not checked
- li reg imm
  - reg ← imm

Code Generation Consideration

- We used to store values in unlimited temporary variables, but registers are limited --> must reuse registers[重复使用寄存器]

- Must save/restore registers when reusing them[保存-恢复]
  - e.g., suppose that we need to store results of expressions in $t0
  - When generating E -> E1 + E2,
    - E1 will first store result into $t0
    - E2 will next store result into $t0, overwriting E1's result
    - Must save $t0 somewhere before generating E2

Code Generation Consideration(cont.)

- Registers are saved on and restored from the stack
- Note: $sp - stack pointer register, pointing to the top of stack
  - Saving a register $t0 on the stack:
    - **sw $t0, 0($sp)** // store word in $t0 on the top of stack
    - **addiu $sp, $sp, -4** // allocate (push) a word on the stack
  - Restoring a value from stack to register $t0:
    - **lw $t0, 4($sp)** // load word from top of stack to $t0
    - **addiu $sp, $sp, 4** // free (pop) word from stack

*MIPS instruction set: https://www.dsi.unive.it/~gasparetto/materials/MIPS_Instruction_Set.pdf

Stack Operations[栈操作]

- To **push** elements onto the stack
  - To move stack pointer $sp down to make room for the new data
  - Store the elements into the stack
- For example, to push registers $t1 and $t2 onto stack
  - add $sp, $sp, -8
  - sw $t1, 4($sp)
  - sw $t2, 0($sp)

MIPS Assembly Example.

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

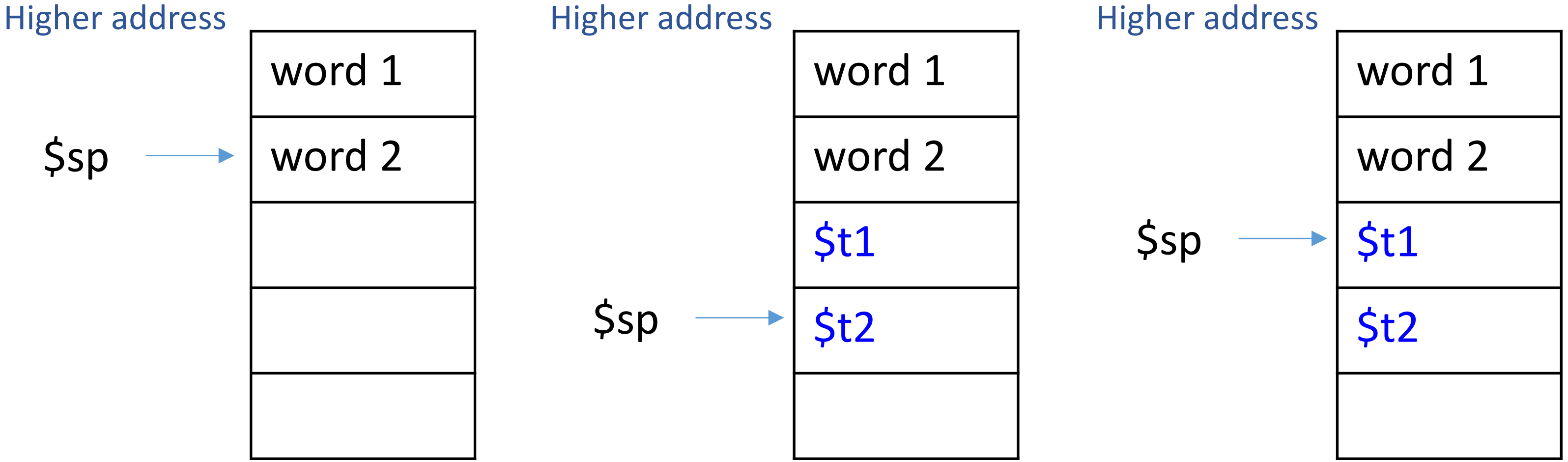
acc ← 7	li $t0 7	// store 7 in $t0
push acc	sw $t0 0($sp)	// store $t0 in the stack
	addiu $sp $sp -4	// decrement sp to make space for the value
acc ← 5	li $t0 5	// store 5 in $t0
acc ← acc + stack_top	lw $t1 4($sp)	// load value from $sp+4 into $t1
	add $t0 $t0 $t1	// add $t0+$t1 = 5 + 7, store result in $t0
pop	addiu $sp $sp 4	// pop constant 7 off stack

Question: What is the value of $t0, $t1 and $sp-4 after the computation?

*MIPS instruction set: https://www.dsi.unive.it/~gasparetto/materials/MIPS_Instruction_Set.pdf

Stack Operations (cont.)

- **Pop** elements simply by adjusting the \$sp upwards
  - Note that the popped data is still present in memory, but data past the stack pointer is considered invalid



Code Generation Strategy

中山大學 SUN YAT-SEN UNIVERSITY

- For each expression e, we generate MIPS code that:
  - Computes the value of e into $t0
  - Preserves $sp and the contents of the stack
- We define a code generation function cgen(e)
  - Its result is the code generated for e

```
cgen(e1 + e2):
    cgen(e1) # stores result of e1 in $t0
    sw $t0 0($sp)
    addiu $sp $sp -4 # pushes $t0 on stack
    cgen(e2) # overwrites result in $t0
    lw $t1 4($sp) # pops value of e1 to $t1
    addiu $sp $sp 4
    add $t0 $t1 $t0 # performs addition
```

```
cgen(e1 + e2):
    cgen(e1) # stores result in $t0
    move $t1 $t0 # copy result of $t0 to $t1
    cgen(e2) # stores result in $t0
    add $t0 $t1 $t0 # performs addition
```

Possible optimization: put the result of e1 directly in register $t1? What if 3 + (7 + 5)?

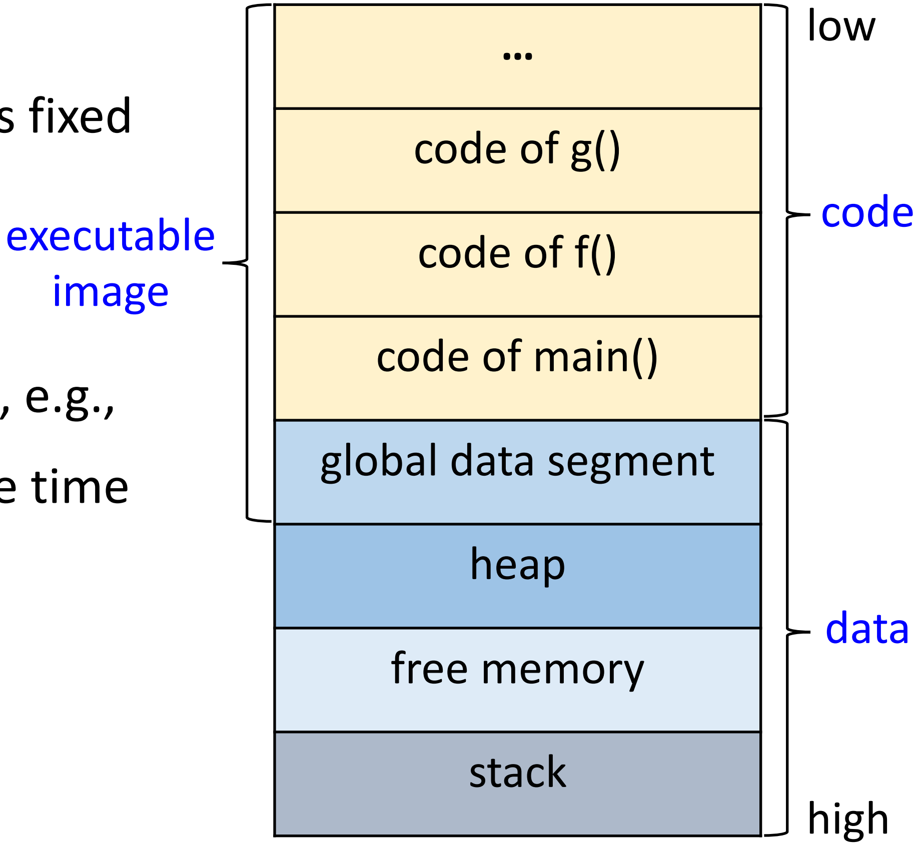
Code Generation for the Conditional

- We need flow control instructions
- New instruction: beq reg1 reg2 label
  - Branch to label if reg1 ==reg2
- New instruction: b label
  - Unconditional jump to label

```
cgen(if e1 == e2 then e3 else e4):
        cgen(e1)
        # pushes $t0 on stack
        sw $t0 0($sp)
        addiu $sp $sp -4
        # overwrites $t0
        cgen(e2)
        # pops value of e1 to $t1
        lw $t1 4($sp)
        addiu $sp $sp 4
        # performs comparison
        beq $t0 $t1 true_branch
    false_branch:
        cgen(e4)
        b end_if
    true_branch:
        cgen(e3)
    end_if:
```

Example Memory Layout

- **Code**
  - ◆ the size of the generated target code is fixed at compile time
- **Global/static**
  - ◆ the size of some program data objects, e.g., global constants, are known at compile time
- **Stack**
  - ◆ store dynamic data structures
- **Heap**
  - ◆ manage long-lived data



Activation[活动]

- Compiler typically allocates memory in the unit of procedure.
- Each execution of a procedure is called as its activation[活动].
  - ◆ starts at the beginning of the procedure body
  - ◆ When completed, returns the control to the point immediately after the place where that procedure is called
- Activation record (AR[活动记录]) is used to manage the information needed by a single execution of a procedure
- Stack is to hold activation records that get generated during procedure calls

ARs in Stack Memory[在栈中管理]

- Manage ARs like a stack in memory[AR栈管理]
  - On function entry: AR instance allocated at top of stack
  - On function return: AR instance removed from top of stack
- Hardware support
  - Stack pointer ($SP) register[栈指针]
    - $SP stores address of top of the stack
    - Allocation/de-allocation can be done by moving $SP
  - Frame pointer ($FP) register[帧指针]
    - $FP stores base address of current frame
    - Frame: another word for AR
    - Variable addresses translated to an offset from
  - $FP and $SP together delineate the bounds of current AR

Contents of ARs

- Example layout of a function AR

Temporaries	临时变量
Local variables	局部变量
Saved Caller/Callee Register Values	保存的寄存器值
Saved Caller's Instruction Pointer ($IP)	保存的调用者指令指针
Saved Caller's AR Frame Pointer ($FP)	保存的调用者AR指针
Parameters	参数
Return Value	返回值

- Registers such as $FP and $IP overwritten by callee → Must be saved to/restored from AR on call/return
  - ◆ Caller's $IP: where to execute next on function return (a.k.a. return address: instruction following function call)
  - ◆ Caller's $FP: where $FP should point to on function return

Caller/Callee Conventions

- Important registers should be saved across function calls
  - Otherwise, values might be overwritten
- But, who should take the responsibility?
  - The caller knows which registers are important to it and should be saved
  - The callee knows exactly which registers it will use and potentially overwrite
  - However, the caller and the callee don’t know anything about each other’s implementation

Caller/Callee Conventions (cont.)

- Potential solutions
  - Solution 1: caller to save any important registers that it needs before calling a func, and to restore them after (but not all will be overwritten)
  - Solution 2: callee saves and restores any registers it might overwrite (but not all are important to caller)
- Caller and callee should cooperate

Caller/Callee Conventions (cont.)

- Caller: save and restore any of the following caller-saved registers that it cares
  - $t0-$t9, $a0-$a3, $v0-$v1
  - The callee can modify these registers, assuming that the caller already saved them
- Callee: save and restore any of the following callee-saved registers that it uses
  - $s0-$s7, $ra
  - The caller assume these registers are not changed by the callee

Symbolic Name	Number	Usage
zero	0	Constant 0.
at	1	Reserved for the assembler.
v0 - v1	2 - 3	Result Registers.
a0 - a3	4 - 7	Argument Registers 1 · · · 4.
t0 - t9	8 - 15, 24 - 25	Temporary Registers 0 · · · 9.
s0 - s7	16 - 23	Saved Registers 0 · · · 7.
k0 - k1	26 - 27	Kernel Registers 0 · · · 1.
gp	28	Global Data Pointer.
sp	29	Stack Pointer.
fp	30	Frame Pointer.
ra	31	Return Address.

Detailed Calling Steps

- The **caller** sets up for the call via these steps[调用者]
  - ◆ Make space on stack for and save any caller-saved registers
  - ◆ Pass arguments by pushing them on the stack, one by one, right to left
  - ◆ Jump to the function (saves the next inst in $ra)
- The **callee** then takes over and does the following[被调用者]
  - ◆ Make space on stack for and save values of $fp and $ra[caller当前的FP地址和返回地址]
  - ◆ Configure frame pointer by setting $fp to base of frame[设置自己的FP]
  - ◆ Allocate space for stack frame (required for all local and temporary variables)
  - ◆ Execute function body, code can access params at positive offset from $fp, locals/temps at negative offsets from $fp

Detailed Calling Steps (cont.)

- When ready to exit, the **callee** does the following[调用退出]
  - Assign the return value (if any) to $v0
  - Pop stack frame off the stack (locals/temps/saved regs)
  - Restore the value of $fp and $ra
  - Jump to the address saved in $ra
- When control returns to the **caller**, it cleans up from the call with the steps[调用返回]
  - Pop the parameters from the stack
  - Restore value of any caller-saved registers

Code Generation for Function Call

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

- New instruction: jal label.
  - Jump to label, after saving address of next instruction in $ra.

- New instruction: jr reg
  - Jump to address in register reg

```
cgen(f(e1, …, en)):
        cgen(en)                  # push arguments in reserve order
        addiu $sp $sp -4
        sw $a0 0($sp)
        …
        addiu $sp $sp -4          # saves FP
        sw $fp 0($sp)
        addiu $sp, $sp, -4        # pushes return address
        sw $ra, 0($sp)
        move $fp, $sp             # begins new AR in stack
        jal f_entry               # jump (update $ra)

cgen(def f(x1,…,xn) = e):
f_entry:
        cgen(e)
        move $sp $fp              # removes AR from stack
        sw $ra 0($sp)             # pops return address
        addiu $sp $sp 4
        lw $fp 0($sp)             # pops old fp
        addiu $sp $sp 4
        jr $ra                    # jumps to return address
```

Code Generation for Variables

- The "variables" of a function are just its 'parameters'
  - ◆ They are all in the AR
  - ◆ Pushed by the caller
- **Problem**: the stack grows when intermediate results are saved, so the variables are not at a fixed offset from $sp
  - ◆ Thus, access to locations in the stack frame cannot use $sp-relative addressing
- **Solution**: use the frame pointer $fp instead
  - ◆ Always points to the return address on the stack
  - ◆ Since it does not move, it can be used to find the variables

Example

- Local variables are referenced from an offset from $fp
  - $fp is pointing to old $ip (return address)
- For a function def f(x,y) = e, the activation and frame pointer are set up as follows:
  - The parameters are pushed right to left by the caller
  - The locals are pushed left to right by the callee

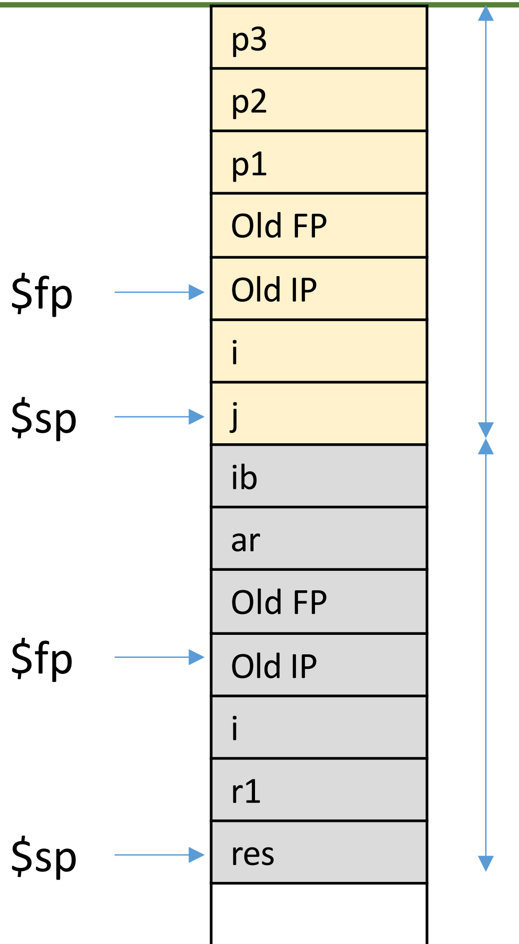
y	
x	
Old FP	
Old IP	← $fp
Local variables	
Temporaries	← $sp
…	

x: +8($fp)
y: +12($fp)
First local variable: -4($fp)

Example

```
double fun1(int p1, double p2, int p3) {
        int i, j;
        res = fun2(p1*p2, j);
        return res;
 }
```

```
double fun2(double ar, int ib) {
        int i, r1;
        double res;
        …
        return res;
}
```



Code Generation for OO

- Objects are like structures in C
  - Objects are laid out in contiguous memory
  - Each member variable is stored at a fixed offset in object

- Unlike structures, objects have member methods

Code Generation for OO(cond.)

- Two types of member methods:
  - **Nonvirtual** member methods: cannot be overridden
    - Parent obj = new Child();
    - obj.nonvirtual(); // Parent::nonvirtual() called
    - Method called depends on (static) reference type
    - Compiler can decide call targets statically
  - **Virtual** member methods: can be overridden by child class
    - Parent obj = new Child();
    - obj.virtual(); // Child::virtual() called
    - Method called depends on (runtime) type of object
    - Need to call different targets depending on runtime type

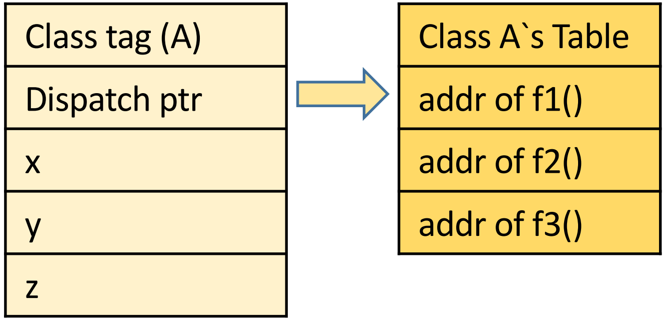
Static and Dynamic Dispatch

- **Dispatch**: to send to a particular place for a purpose
  - i.e., to jump to a (particular) function
- **Static Dispatch**: selects call target at compile time
  - Nonvirtual methods implemented using static dispatch
  - Implication for code generation -- Can hard code function address into binary
- **Dynamic Dispatch**: selects call target at runtime
  - Virtual methods implemented using dynamic dispatch
  - Implication for code generation:
    - Must generate code to select correct call target, but how?
  - At compile time, generate a dispatch table for each class, containing call targets for all virtual methods of that class.
  - At runtime, each object has a pointer to its dispatch table, which is indexed into to find call target for its runtime type.

Typical Object Layout

- **Class tag** is used for dynamic type checking
- **Dispatch ptr** is a pointer to the dispatch table
- Compiler translates member accesses to offset accesses

```
if(...) obj = new Parent();
else  obj = new Child();
obj.x = 10;     // move 10, x_offset(obj)
obj.f2();       // call f2_offset(obj.dispatch_ptr)
```



- Offsets must remain identical
  - ◆ How to layout object and dispatch table to make it so?

Inheritance and Subclasses

- **Invariant**: the offset of a member variable or member method is the same in a class and all of its subclasses

```
class A1 {
  int x;
  virtual void f1() { ... }
  virtual void f2() { ... }
}
class A2 inherits A1 {
  int y;
  virtual void f2() { ... }
}
class A3 inherits A2 {
  int z;
  virtual void f3() { ... }
 }
```

Class tag (A1)
Dispatch ptr
x

A1's Table
f1()
f2()

Class tag (A2)
Dispatch ptr
x
y

A2's Table
f1()
f2()

Class tag (A3)
Dispatch ptr
x
y
z

A3's Table
f1()
f2()
f3()

Note, f2() in A1's and A2's tables are different

Inheritance and Subclasses (cont.)

- Member variable access
  - Generate code using offset for reference type (class)
  - Object may be a child type, but will still have same offset
- Member method call
  - Generate code to load call target from dispatch table using offset for reference type
  - Again, object may be of child type, but still same offset

Machine Optimizations[机器相关优化]

- After performing IR optimizations
  - We need to convert the optimized IR into the target language (e.g., assembly, machine code)
- Specific machine features are taken into account to produce code optimized for the particular architecture[考虑特定的架构特性]
  - e.g., specialized instructions, hardware pipeline abilities, register details
- Typical machine optimizations[典型的优化方案]
  - Instruction selection and scheduling[指令选择与调度]
    - select which instructions to implement the operators in IR
    - Decide in what order the instructions are executed.
  - Register allocation[寄存器分配优化]: map values to registers and manage
  - Peephole optimization[窥管优化]: locally improve the target code

Instruction Selection[指令选取]

- To find an efficient mapping from the IR to a target-specific assembly listing[IR到汇编的映射]
- Instruction selection is particularly important when targeting architectures with CISC (e.g., x86)
  - In these architectures, there are typically several possible implementations of the same IR operation, each with different properties
  - e.g., on x86, an addition of one can be implemented by an add or inc

x = y + z

```
MOV y,R0
ADD z,R0
MOV R0,x
```

a = a + 1

```
MOV a,R0
ADD #1,R0
MOV R0,a
```

→

```
MOV a,R0
INC R0
MOV R0,a
```

Instruction Cost[指令成本]

- Instruction cost = 1 + cost (source-mode) + cost (destination-mode)

Mode	Form	Address	Added Cost
Absolute	**M**	**M**	1
Register	**R**	**R**	0
Indexed	*c*(**R**)	*c*+*contents*(**R**)	1
Indirect register	***R**	*contents*(**R**)	0
Indirect indexed	*****c*(**R**)	*contents*(*c*+*contents*(**R**))	1
Literal	**#***c*	N/A	1

- Examples

Instruction	Operation	Cost
`MOV R0,R1`	Store *content*(**R0**) into register **R1**	1
`MOV R0,M`	Store *content*(**R0**) into memory location **M**	2
`MOV M,R0`	Store *content*(**M**) into register **R0**	2
`MOV 4(R0),M`	Store *contents*(4+*contents*(**R0**)) into **M**	3
`MOV *4(R0),M`	Store *contents*(*contents*(4+*contents*(**R0**))) into **M**	3
`MOV #1,R0`	Store 1 into **R0**	2
`ADD 4(R0),*12(R1)`	Add *contents*(4+*contents*(**R0**)) to *contents*(12+*contents*(**R1**))	

Instruction Cost (cont.)

- Suppose we translate TAC x:=y+z to

  MOV y, R0

  ADD z, R0

  MOV R0, x

Mode	Form	Address	Added Cost
Absolute	M	M	1
Register	R	R	0
Indexed	c(R)	c+contents(R)	1
Indirect register	*R	contents(R)	0
Indirect indexed	*c(R)	contents(c+contents(R))	1
Literal	#c	N/A	1

- a := b + c

```
MOV b, R0
ADD c, R0
MOV R0, a
```

cost = 6

```
MOV b, a
ADD c, a
```

cost = 6

```
MOV *R1, *R0
ADD *R2, *R0
```

cost = 2

Assuming R0, R1 and R2 contain the addresses of a, b, and c

- a := a + 1

```
MOV a, R0
ADD #1, R0
MOV R0, a
```

cost = 6

```
ADD #1, a
```

cost = 3

```
INC a
```

cost = 2

Instruction Scheduling[指令调度]

- Some facts
  - Instructions take clock cycles to execute (latency)
  - Modern machines issue several operations per cycle (Out-of-Order Execution)
  - Execution time is order-dependent
- Goal: reorder the operations to minimize execution time
  - Minimize wasted cycles
  - Avoid spilling registers
  - Improve locality

```
A = x * y;
B = A + 1;
C = y;
```

→

```
A = x * y;
C = y;
B = A + 1;
```

(Now C=y can execute while waiting for A=x*y)

Register Allocation[寄存器分配]

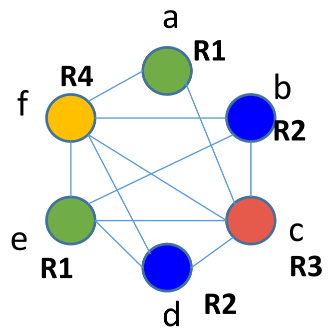
- In TAC, there are an unlimited number of variables
    - On a physical machine there are a small number of registers
- Register allocation is the process of assigning variables to registers and managing data transfer in and out of registers
    - How to assign variables to finitely many registers?
    - What to do when the number of variables outweighs the number registers?
    - How to do so efficiently?
- Using registers intelligently is a critical step in any compiler
    - Accesses to memory are costly, even with caches
    - A good register allocator can generate code orders of magnitude better than a bad register allocator

Register Allocation (cont.)

- Goals of register allocation
  - ◆ Keep frequently accessed variables in registers
  - ◆ Keep variables in registers only as long as they are live
- Local register allocation[局部]
  - ◆ Allocate registers basic block by basic block
  - ◆ Makes decisions on a per-block basis (hence 'local')
- Global register allocation[全局]
  - ◆ Makes global decisions about register allocation such that
    - ▫ Var to reg mappings remain consistent across blocks
    - ▫ Structure of CFG is taken into account on decisions
- Three well-known register allocation algorithms
  - ◆ Graph coloring allocator[图着色]
  - ◆ Linear scan allocator[线性扫描]
  - ◆ ILP (Integer Linear Programming) allocator[整数线性规划]

Graph Coloring[图着色]

- **Register interference graph** (RIG)[相交图]
  - Each node represents a variable
  - An edge between two nodes V_1 and V_2 represents an interference in live ranges[活跃期/生存期]
- Based on RIG
  - Two variables can be allocated in the same register if there is no edge between them[若无边相连, 可使用同一寄存器]
  - Otherwise, they cannot be allocated in the same register
- Problem of register allocation maps to graph coloring
  - Once solved, k colors can be mapped back to k registers
  - If the graph is k-colorable, it's k-register-allocatable



Register Spilling[寄存器溢出]

- Determining whether a graph is k-colorable is NP-complete
  - Therefore, problem of k-register allocation is NP-complete
  - In practice: use heuristic polynomial algorithm that gives sub-optimal allocations in most of the time
  - Chaitin's graph coloring is a popular heuristic algorithm
    - e.g., most backends of GCC use Chaitin's algorithm
- What if k-register allocation does not exist?
  - Spill a variable to memory to reduce RIG and try again
  - Spilled variable stays in memory and is not allocated a register
- Spilling is slow
  - Placed into memory, loaded into register when needed, and written back to memory when no longer used

Peephole Optimization[窥孔优化]

- Optimization ways
  - Usual: produce good code through careful instruction selection and register allocation
  - Alternative: generate naive target code and then improve
- A simple but effective technique for locally improving the target code
  - Done by examining a sliding window of target instructions (called **peephole**)
  - Replace instruction sequences within the peephole by a shorter or faster sequence
  - Can also be applied directly after IR generation to improve IR
- Example transformations
  - Redundant-instruction elimination
  - Flow-of-control optimizations
  - Algebraic simplifications

Jump to jumps

```
if a < b goto L1
…
L1: goto L2
```

⇨

```
if a < b goto L2
…
L1: goto L2
```

Summary

- Code can be optimized at different levels with various techniques
  - Peephole, local, loop, global
  - IR: local, global, CSE, constant folding and propagation, …
  - Target: instruction, register, …
- Interactions between the various optimization techniques
  - Some transformations may expose possibilities for others
  - One optimization may hide or remove possibilities for others
- Affect of compiler optimizations are intertwined and hard to separate
  - Finding optimal optimization combinations is in research
  - Compilers package optimization that typically go together into levels (e.g., -O1, -O2)