中山大學 SUN YAT-SEN UNIVERSITY
计算机学院（软件学院）
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Compilation Principle
编 译 原 理

第25讲：目标代码生成(3)

张献伟

xianweiz.github.io

DCS290, 6/13/2023

Review Questions

- Q1: is add a0, a0, 5(t0) a valid RISC-V instruction?

  NO. RISC-V is load-store architecture, i.e., only load and store can access memory (all others must use registers).

- Q2: what are *fp* and *sp* registers for?

  Frame and stack pointer, indicating the stack bounds.

- Q3: what is activation record?

  Info of a procedure execution, i.e. stack frame.

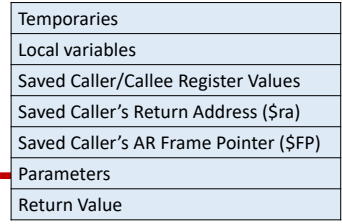
- Q4: why a calling convention is needed?

  Caller and callee cooperate to effectively save registers across function calling.

- Q5: explain preserved registers.

  Registers that a function (callee) promises to leave unchanged. So the caller can directly use them after a function call ends.

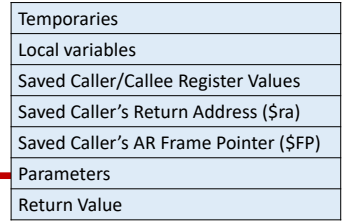
Detailed Calling Steps



- The **caller** sets up for the call via these steps[调用者]
  - 1) Make space on stack for and save any caller-saved registers
  - 2) Pass arguments by pushing them on the stack, one by one, right to left[传参数]
  - 3) Execute a jump to the function (saves the next inst in *ra*)

- The **callee** takes over and does the following[被调用者]
  - 4) Make space on stack for and save values of fp and ra
  - 5) Configure frame pointer by setting fp to base of frame
  - 6) Allocate space for stack frame (total space required for all local and temporary variables)
  - 7) 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 following[调用退出]
  - 8) Assign the return value (if any) to a0[返回值]
  - 9) Pop stack frame off the stack (locals/temps/saved regs)
  - 10) Restore the value of fp and ra
  - 11) 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]
  - 12) Pop the parameters from the stack
  - 13) Restore value of any caller-saved registers, pops spill space from stack

Code Generation for Function Call

- The calling sequence is the 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*
  - Actually, *jal ra label*
    - Store PC+4 in *ra*
    - Similar to *jal x0 label* for jumping inside a loop

```
cgen(f(e1, …, en)):
        # pushes arguments (reverse order)
        cgen(en)
        addiu sp sp -4
        sw a0 4(sp)
        …
        cgen(e1)
        addiu sp sp -4
        sw a0 4(sp)
        # saves FP
        addiu sp sp -4
        sw fp 4(sp)
        # pushes return address
        addiu sp, sp, -4
        sw ra, 4(sp)
        # begins new AR in stack
        mv fp, sp
        # jumps to func entry (update ra)
        jal f_entry
```


https://drive.google.com/file/d/1qe7it1bz7Ioaa8UBduaAv08XU8AFzpbt/view

Code Generation for Function Definition

- New instruction: *jr reg*
  - Jump to address in register reg
  - Acutally, *jalr ra rd rmm*, jump to *rd + imm*
    - Set the PC to *rd + imm*

```
cgen(def f(x1,…,xn) = e):
f_entry:  # save registers ra and si
          cgen(e)
          # removes AR from stack
          mv sp fp
          # pops return address
          sw ra 4(sp)
          addiu sp sp 4
          # pops old FP
          lw fp 4(sp)
          addiu sp sp 4
          # jumps to return address
          jr ra
```

RISC-V uses *jal* to invoke a function and *jr* to return from a function

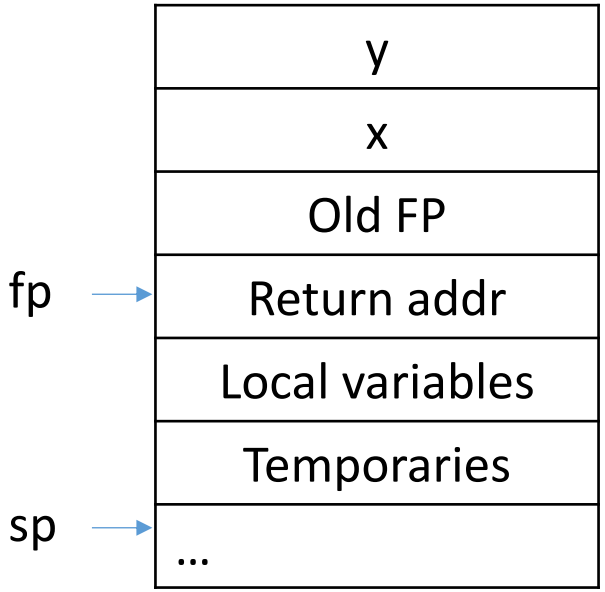
https://drive.google.com/file/d/1qe7it1bz7Ioaa8UBduaAv08XU8AFzpbt/view

Code Generation for Variables

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



x: +8(fp)

y: +12(fp)

First local variable: -4(fp)

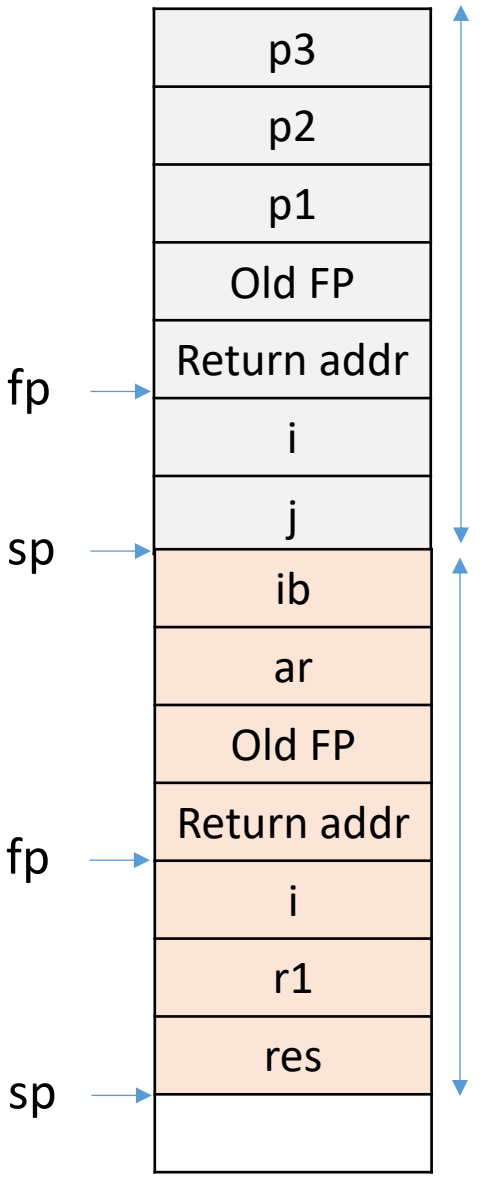
The parameters are pushed right to left by the caller

The locals are pushed left to right by the callee

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 structs in C
  - Objects are laid out in contiguous memory
  - Each member variable is stored at a fixed offset in object
- Unlike structs, objects have member methods
- 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
    ```

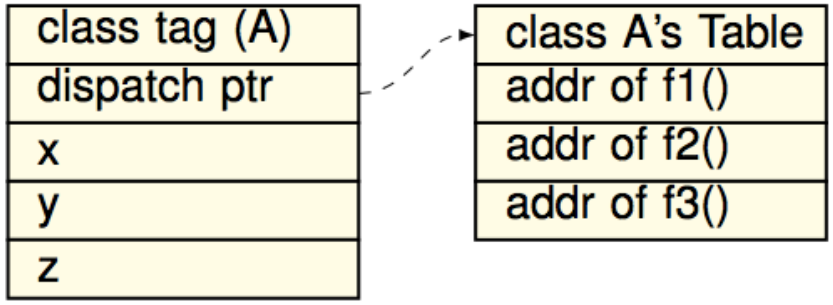
    Method called depends on (runtime) type of object

    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
- 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

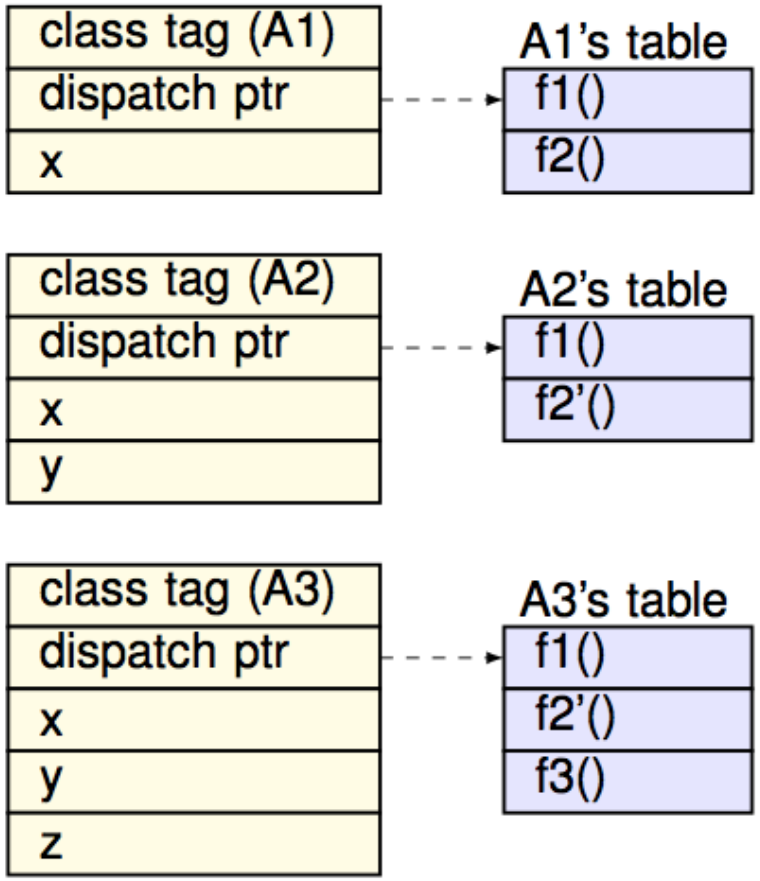
```
if(...) obj = new Parent()
else  obj = new Child();
obj.x = 10;          // move 10, x_offset(obj)
obj.f2();            // call f2_offset(obj.dispatch_ptr)
```

- Offsets must remain identical regardless of object type
  - 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() { ... }
}
```



A Question …

```
1  #include <iostream>
2  using namespace std;
3
4  class A1 {
5    public:
6      virtual void f1() { cout << "base.f1\n"; }
7      virtual void f2() { cout << "base.f2\n"; }
8      void f3() { cout << "base.f3\n"; }
9    private:
10     char a;
11     int x;
12     int y;
13     static int z;
14 };
15
16 int main(int argc, char* argv[]) {
17     A1 a1;
18     cout << "sizeof(a1) = " << sizeof(a1) << "\n";
19
20     return 0;
21 }
```

- What is the output?
  - **24** (on my 64-bit MBA)
- How come?
  - Fields (12B)
    - char a: 1 --> 4
    - int x: 4
    - int y: 4
  - Functions (8B)
    - virtual: 8B
  - Alignment
    - 12+8 --> 24

[1] Determining the Size of a Class Object
[2] sizeof class in C++

Heap Memory Management

- Heap data
  - Lives beyond the lifetime of the procedure that creates it

```
TreeNode* createTREE() {
    TreeNode* p = (TreeNode*)malloc(sizeof(TreeNode));
    return p;
}
```

  - Cannot reclaim memory automatically using a stack
- Problem: when and how do we reclaim that memory?
- Two approaches
  - **Manual** memory management
    - Programmer inserts deallocation calls. E.g. “free(p)”
  - **Automatic** memory management
    - Runtime code automatically reclaims memory when it determines that data is no longer needed

Heap Memory Management (cont.)

- Manual memory management is typically more efficient
  - Programmers know when data is no longer needed
    - With automatic management, runtime must somehow detect when data is no longer needed and recycle it, incurring overheads
- Automatic management leads to fewer bugs
  - Disallowing programmer free() calls is essential for security
- Common functionality in both automatic and manual
  - Runtime code maintains used/unused spaces in heap (e.g. linked together in the form of a list)
  - malloc(int size): move size bytes from unused to used
  - free(void *p): move given memory from used to unused
- Only in automatic memory management
  - Routines to perform detection of unused memory

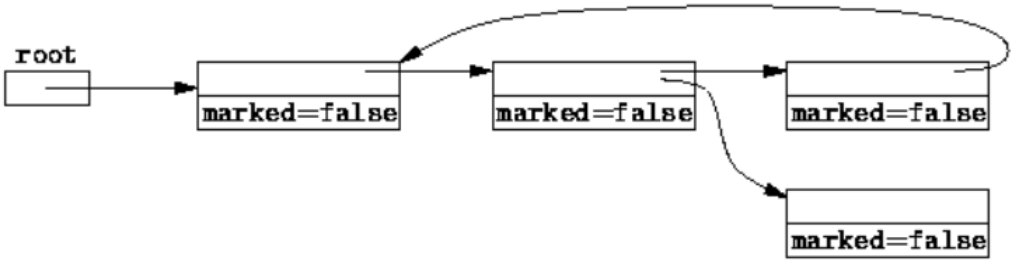
Heap Memory Management (cont.)

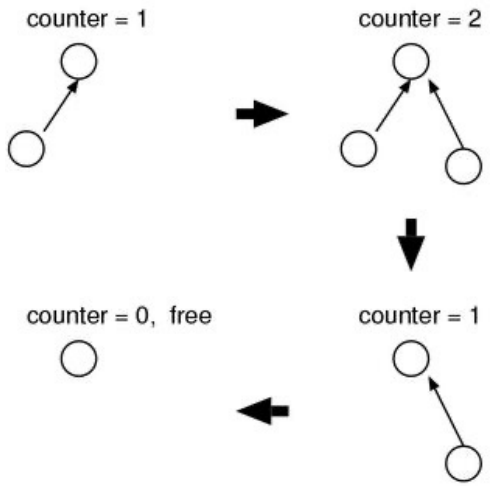
- Detection: determining an object will no longer be used
  - In general, impossible for compiler to tell exactly
    - Requires knowledge of program beyond what compiler has
  - But compiler can tell when it can no longer be used

- An object x is **reachable** iff
  - A named object contains a reference to x, or
  - A reachable object y contains a reference to x
- An unreachable object is referred to as **garbage**
  - Garbage can no longer be used and its memory can be reclaimed
  - This reclamation is process is called **garbage collection**

Garbage Collection Schemes

- Reference Counting[引用计数]
  - Maintain a reference counter inside each object
    - Counts the number of references to object
  - When counter becomes 0, the object is no longer usable
    - Garbage collect unreachable object
- Tracing[追踪/标记清除]
  - When the heap runs out of memory to allocate:
    - 1. Pause the program
    - 2. Trace through all reachable objects
    - 3. Garbage collect remaining objects
    - 4. Restart the program





Machine Optimizations[机器相关优化]

- After performing IR optimizations
  - We need to further convert the optimized IR into the target language (e.g. assembly, machine code)
- Specific machines 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**: select and reorder insts to implement the operators in IR
  - **Register allocation**: map values to registers and manage
  - **Peephole optimization**: locally improve the target code

Instruction Selection[指令选取]

- To find an efficient mapping from the IR of a program 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 *inc*, *add*, or *lea* instruction

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**))	3

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)
  - Cannot use results until ready, can do something else
  - 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]

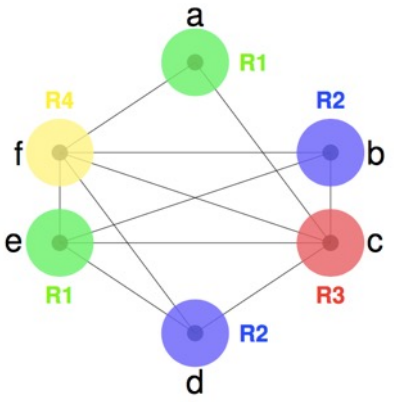
- In TAC, there are an unlimited number of variables
  - On a physical machine there are a small number of registers
- **Register allocation** is the process of assigning variables to registers and managing data transfer in and out of registers
  - How to assign variables to finitely many registers?
  - What to do when it can't be done?
  - 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]
  - LP (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 close to optimal allocations 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 var stays in memory and is not allocated a reg
- 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 inst selection and register allocation
  - Alternative: generate naïve target code and then improve
- A simple but effective technique for locally improving the target code[很局部的优化，但可能带来性能的极大提升]
  - Done by examining a sliding window of target instructions (called **peephole**) and replacing instruction sequences within the peephole by a shorter or faster sequence, whenever psbl
  - Can also be applied directly after IR generation to improve IR
- Example transformations
  - Redundant-instruction elimination
  - Flow-of-control optimizations
  - Algebraic simplifications
  - Use of machine idioms

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

Jump to jumps

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

LLVM

- llc: LLVM static compiler
  - Input: *.ll* or *.bc*
  - Output: assembly language for a specified architecture
- End-user options

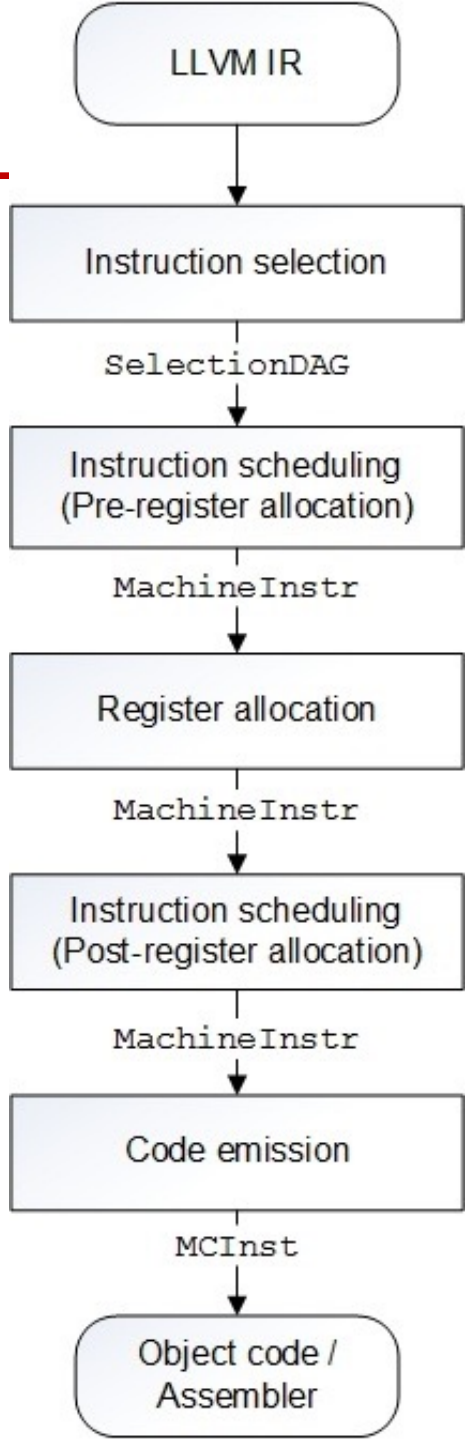
  -march=<arch>: e.g., x86

  -mcpu=<cpuname>: e.g., corei7-avx
- Tuning/Configuration Options

  --print-after-isel: print generated machine code after instruction selection (useful for debugging)

  --regalloc=<allocator>: specify the register allocator to use, basic/fast/greedy/pdqp

  --spiller=<spiller>: simple/local



https://www.llvm.org/docs/CommandGuide/llc.html

Optimizations[总结]

- Code can be optimized at different levels with various techniques
  - Peephole, local, loop, global
  - IR: local, global, common subexpression elimination, constant folding and propagation, …
  - Target: instruction, register, peephole, …
- Interactions between the various optimization techniques
  - Some transformations may expose possibilities for others
  - One opt. may obscure or remove possibilities for others
- Affect of compiler opts are intertwined and hard to separate
  - Finding optimal opt combinations is in itself research
  - Compilers package opts that typically go together into levels (e.g -O1, -O2, -O3)

Final Exam

- 考试时间:
  - 6.27/周二，14:30 – 16:30
- 关于试卷
  - 中文（专业术语标注英文）
  - A、B卷，学院指定
- 成绩计算
  - 期末：60%
  - 平时：40%
    - 课堂：15%
    - 作业：25%
- 题型及分值
  - 一、判断题（10分）
    - 10小题，每小题1分
  - 二、填空题（10分）
    - 几个小题，10个空，每空1分
  - 三、简答题（20 - 25分）
    - 3小题，每小题5 - 10分
  - 四、应用题（55 - 60分）
    - 3小题，每小题10 - 25分
- 主要内容
  - 词法分析
  - 语法分析
  - 语义分析
  - 代码生成及优化