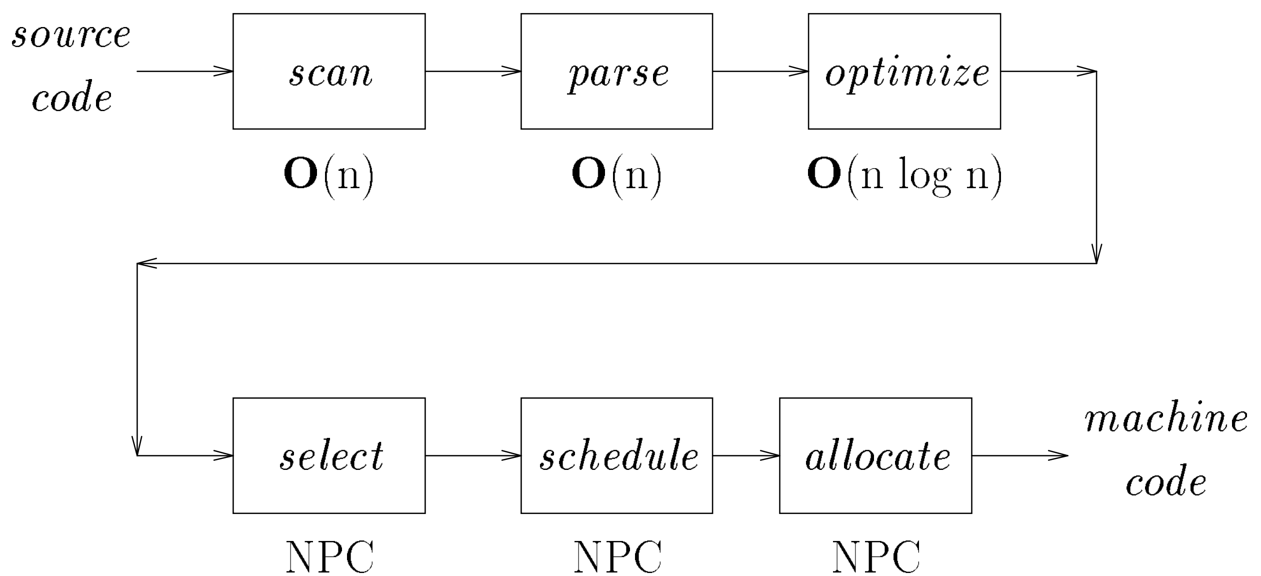
The structure of a compiler

Reality



A compiler is a lot of fast stuff followed by hard problems

Code generation

AS&U think of *intermediate code* generation and *machine code* generation as two problems

- in reality, the differences can be large or small

  AST→RISC ISA (*very different*)

  quads→VAX ISA (*quite similar*)

- changes in machine and compiler architecture make the distinction unimportant

Superscalar instruction issue ⇒ dominant issues

- sequencing instructions (*scheduling*)
- combining instructions (*multiple issue*)
- keeping values in registers (*register allocation*)

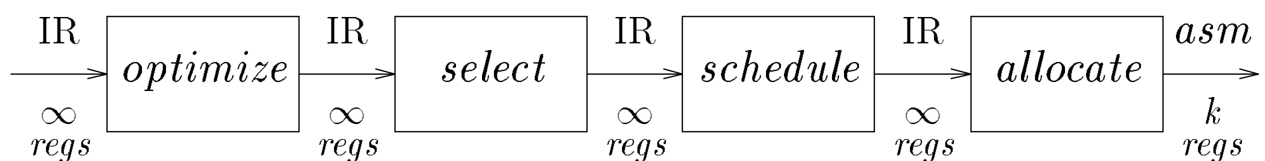
Code generation

Chapters 8 and 9

- really deal with the same set of problems
- lectures will skip back and forth between them

Compilation model

- low-level, RISC-like intermediate language
- separate *selection*, *scheduling*, and *allocation*
- assume a sufficient number of registers in early phases



⇒ *select* is fairly simple

⇒ *allocate* and *schedule* are complex

Definitions

Instruction selection

- the process of mapping IR into assembly code
- assumes a *fixed* storage mapping (*code shape*)
- combining instructions, using address modes

Register allocation

- the process of deciding which values reside in registers
- changes storage mapping (*and the code*)
- concern about placement of data

Instruction scheduling

- the process of reordering instructions to hide latencies
- assumes a *fixed* program
- changes demand for registers

Of course, the problems are *very* inter-related.

The big picture

How hard are these problems?

Instruction selection

- can make locally optimal choices (*BURS*)
- can automate construction

Register allocation

- one basic block: (*no spilling*)
  - one register size ⇒ linear time
  - two register sizes ⇒ NP-complete
- whole procedure : NP-complete (*no spilling*)

Instruction scheduling

- basic block ⇒ polynomial time
- across blocks ⇒ *extremely* hard

Before looking at code-generator generators (CGG)*,*
look at code generators (CG)

The big picture

Conventional wisdom says that we can (and should) *attack each of these problems independently*

Instruction selection

- use either tree-matching, or instruction matching
- either:
  1. assume "enough" registers, or
  2. target "important" values into registers

Register allocation

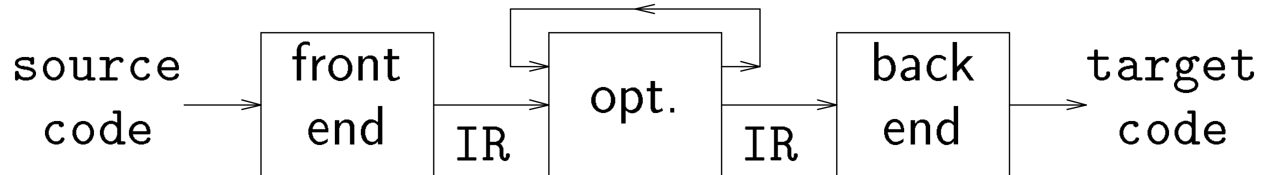
- virtual registers $\Rightarrow$ map "enough" into reality
- targeting $\Rightarrow$ prioritize by "good" metric

Instruction scheduling

- within a block, use list-scheduling
- across blocks (small loops), use *software pipelining*

Note the large number of "fuzzy" terms!

Code generation



1. Source code → intermediate representation
   (a) storage layout
   (b) code for simple expressions
   (c) code for control structures
   (d) code for procedure calls
   (e) code for complex expressions
   (f) better code for expressions

2. Intermediate representation → target code
   (a) instruction selection
   (b) instruction scheduling
   (c) register allocation

We will be covering each issue in order

Intermediate representation

We'll be targeting RISC-like processors

- load-store architecture
- register-transfer language
- three-address code
- explicit loads and stores

Examples

```
load r1, <addr>        $ r1 ← value at <addr>
loadi r1, <const>      $ r1 ← value of <const>
store r1, <addr>       $ <addr> ← r1
move r1, r2            $ r1 ← r2
add r1, r2, r3         $ r1 ← r2 + r3
sub r1, r2, r3         $ r1 ← r2 - r3
mult r1, r2, r3        $ r1 ← r2 * r3
jmp <addr>             $ jump to <addr>
```

Storage layout

Local, non-static storage

- stash them in the frame
- keep frames on the stack
- assign offsets from the frame pointer
- pad for word alignment

Global or static storage

- offset from procedure's static data area (static)
- offset from known global label (global)

Varying sized storage

- *local, non-static* $\Rightarrow$ top of stack frame
- *other cases* $\Rightarrow$ allocate on the heap

Storage layout

Key Issue

- what variables can be *safely* allocated to registers?
- what variables *should* be allocated to registers?

Encoding the decision

- a "code shape" issue
- assign some variables to virtual registers
- treat those that cannot be in a register carefully

Depends on the philosophy of the register allocator

Simple expressions

Expression trees:

- adopt a simple treewalk scheme
- assign a virtual register to each operator
- emit code in postorder walk

Support routines:

- `base( str )` — returns the name of a virtual register that contains the base address for `str`
- `offset( str )` — returns the name of a virtual register that contains the offset of `str`
- `newtemp()` — returns a new virtual register name

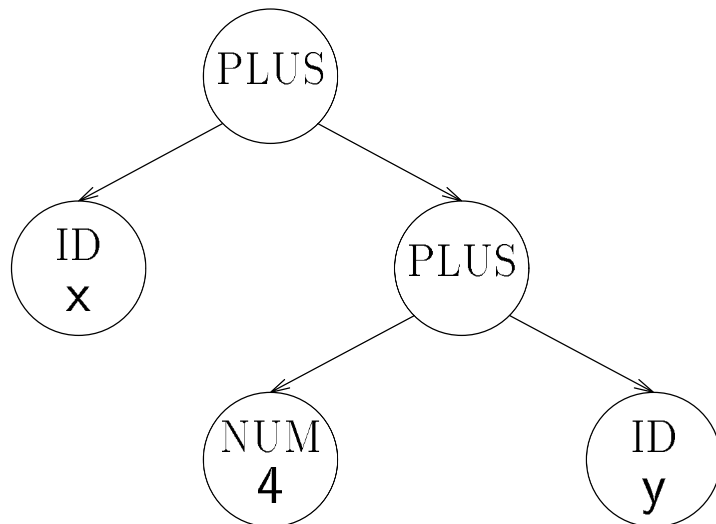
Assume:

- assume tree reflects precedence, associativity
- assume all operands are integers

Simple expressions

```
expr( node )
   int result, t1, t2, t3;
   switch( type of node )

      case PLUS:
         t1 = expr( left child of node );
         t2 = expr( right child of node );
         result = newtemp();
         emit( add, result, t1, t2 );
         break;

      case ID:
         t1 = base( node.val );
         t2 = offset( node.val );
         t3 = newtemp();
         emit( add, t3, t1, t2 );
         result = newtemp();
         emit( load, result, t3 );
         break;

      case NUM:
         result = newtemp();
         emit( loadi, result, node.val );
         break;

   return result
```

Simple expressions




```
loadi r1, base of x      $ base(x)
loadi r2, offset of x    $ offset(x)
add r3, r1, r2           $ addr = base+offset
load r4, r3              $ r4 ← x
loadi r5, 4              $ constant
loadi r6, base of y      $ base(y)
loadi r7, offset of y    $ offset(y)
add r8, r6, r7           $ addr = base+offset
load r9, r8              $ r9 ← y
add r10, r5, r9          $ r10 ← 4 + y
add r11, r4, r10         $ r11 ← x + (4 + y)
```

Control structures

Assignment statement

$$lhs \leftarrow rhs$$

Strategy

- evaluate *rhs* to a value (*an rvalue*)
- evaluate *lhs* to an address (*an lvalue*)
  i) *lvalue* is register $\Rightarrow$ `move` it
  ii) *lvalue* is address $\Rightarrow$ `store` it

Registers versus memory

- non-aliased scalars $\Rightarrow$ can go in a register
- aggregate or potentially aliased $\Rightarrow$ in memory

Control structures

Basic blocks

- a *basic block* is a sequence of straight line code
- if one instruction executes, they all execute
- a maximal sequence of instructions without branches
- a label starts a new basic block

Early work in code optimization focused on basic blocks.

- common subexpression elimination
- constant folding
- "optimal" code generation
- list scheduling

Control structures

Control structure examples

- `if-then-else`
- `while loop`
- `case statement`

Overview

- control flow links up the basic blocks
- ideas are simple
- implementation requires bookkeeping
- some care is required for good code
- *design-time* vs. *compile-time* vs. *run-time*

Early optimizing compilers generated good code for basic blocks and linked them together carefully.

Control structures

`if-then-else`

1. evaluate the expression to `true` or `false`
2. if `true`, fall through to `then` part
   branch around `else` part
3. if `false`, branch to `else` part
   fall through to next statement

Example

Code	Comment
`r1` ← *expr*	*evaluate the expression*
`if not(r1) br L1`	*compare and branch*
`...`	*stmts for* `then` part
`br L2`	*branch to exit*
`L1: ...`	*stmts for* `else` part
`L2: ...`	*following stmt*

Control structures

`while` *loop or* `do` *loop*

1. evaluate the control expression
2. if `false`, branch beyond end of loop
   if `true`, fall through into loop body
3. at end, re-evaluate the control expression
4. if `true`, branch to top of loop body
   if `false`, fall through

Example

Code	Comment
`    r1 ← expr`	*evaluate the expression*
`    if not(r1) br L2`	*compare and branch*
`L1: ...`	*loop body*
`    r1 ← expr`	
`    if r1, br L1`	
`L2: ...`	*following stmt*

Test at end ⇒ *simple loop is one block*

Control structures

`case statement`

1. evaluate the controlling expression
2. branch to the selected case
3. execute its code
4. branch to the following statement

Key Issue:

⇒ *finding the right case*

Method	*When*	*Cost*
linear search	few cases	$\mathbf{O}(\mid cases \mid)$
binary search	sparse	$\mathbf{O}(\log_2(\mid cases \mid))$
jump table	dense	$\mathbf{O}(1)$

Procedure calls

Code for procedure calls [†]

	Code	Comment
	`save registers`	*prolog code*
	`extend basic frame`	*for local data*
	`find static data area`	*if needed*
	`initialize locals`	
	`...`	
	`allocate child's frame`	*start of a* `call`
	`evaluate & store params.`	
	`store` FP `and` RA	*in child's frame*
	`set` FP `for child`	
	`jump to child`	*may handle* RA
	`...`	
RA:	`copy return value`	*post-*`call` *code*
	`restore params.`	*if needed*
	`free child's frame`	
	`...`	
	`store return value`	*epilog code*
	`unextend basic frame`	
	`restore registers`	
	`restore parent's` FP	
	`jump to` RA	*hardware* `ret`

[†] FP is *frame pointer*, RA is *return address*

Procedure calls

All the critical issues are in the linkage convention.

Issues

- caller saves or callee saves
- no need for a static data area
- parameters on stack or in registers
- return address in stack or register
- where's the frame pointer
- static links, frame display, or global display
- return value on stack or in register

The first test of a linkage convention is that it works