

## More “optimal code generation”

---

Sethi-Ullman is optimal on simple machine model

- minimizes register use
- minimizes execution time

What about a more realistic machine model?

Delayed-load architectures are more complex

- issue load, result appears *delay* cycles later
- execution continues unless result is referenced
- premature reference causes hardware to stall (*interlock*)

Many microprocessor-based systems have this property

See: T.A. Proebsting and C.N. Fischer, “Linear-time, optimal code scheduling for delayed-load architectures”, in *Proceedings of the SIGPLAN 91 Conference on Programming Language Design and Implementation*”

## More “optimal code generation”

---

### Assumptions

1. input is an *expression tree*
  - not a *dag*
  - represents entire basic block
2. delayed-load, RISC architecture
  - register-to-register ops, load, & store
  - 1 cycle/instruction
  - non-blocking, 2 cycle load

### The big picture

*interlocks* waste resources

rearrange instructions to fill *delay* slots

⇒ *move loads as early as possible*

## So what's wrong with Sethi-Ullman

---

### Quick review

#### 1. modify labeling scheme for RISC

- left & right leaves labeled with 1
- interior labels as before
$$\begin{cases} l_1 = l_2 \Rightarrow l_1 + 1 \\ l_1 \neq l_2 \Rightarrow \max(l_1, l_2) \end{cases}$$

#### 2. code generation

- more demanding subtree first
- if  $l > \mathcal{R}$ , spill ( $\mathcal{R}$  is # regs.)

This is *optimal* if  $delay = 0$

### The problem

- loads will interlock if  $delay > 0$

### Overview of the solution

- move loads back at least  $delay$  slots from *ops*
- this increases register pressure
- want to minimize this extra register pressure

## Brute force solution

---

### Obvious approach

- issue all the loads
- execute all the operators

Unfortunately, this can create *too much* register pressure

### Phase ordering

- allocate first  $\Rightarrow$  poor schedule
- schedule first  $\Rightarrow$  poor allocation

*Scheduling & allocation are like oil and water*

### Proebsting and Fischer

- retain good properties of Sethi-Ullman
  - contiguous evaluation
  - minimal register use
- consider two problems together<sup>†</sup>

<sup>†</sup> *a recurrent theme for the nineties*

## The DLS algorithm

---

### The big picture

- schedule the operations (*à la* Sethi-Ullman)
- schedule the loads

### Legal ordering

- children of an operator appear before it
- each load appears before operator that uses it

### The final schedule

- preserves relative order of operations  
(*ops*  $\leftrightarrow$  *ops*)
- preserves relative order of loads  
(*loads*  $\leftrightarrow$  *loads*)
- changes relative order of loads to operations

## The DLS algorithm

---

The canonical order

Given  $\mathcal{R}$  registers

1. schedule  $\mathcal{R}$  loads
2. schedule a series of  $(op, load)$  pairs
3. schedule the remaining  $\mathcal{R} - 1$  ops

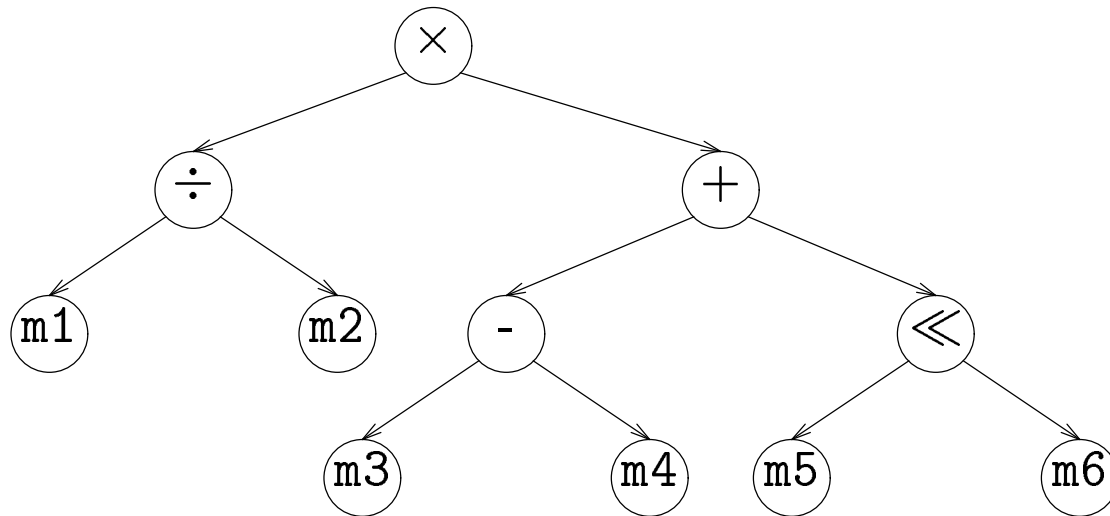
This keeps extra register pressure down

The algorithm

1. run Sethi-Ullman algorithm
  - calculate  $minReg$  for each subtree
  - create an ordering of the operators
2. put loads into canonical order
  - uses  $minReg + 1$  regs
  - requires some renaming

# Example

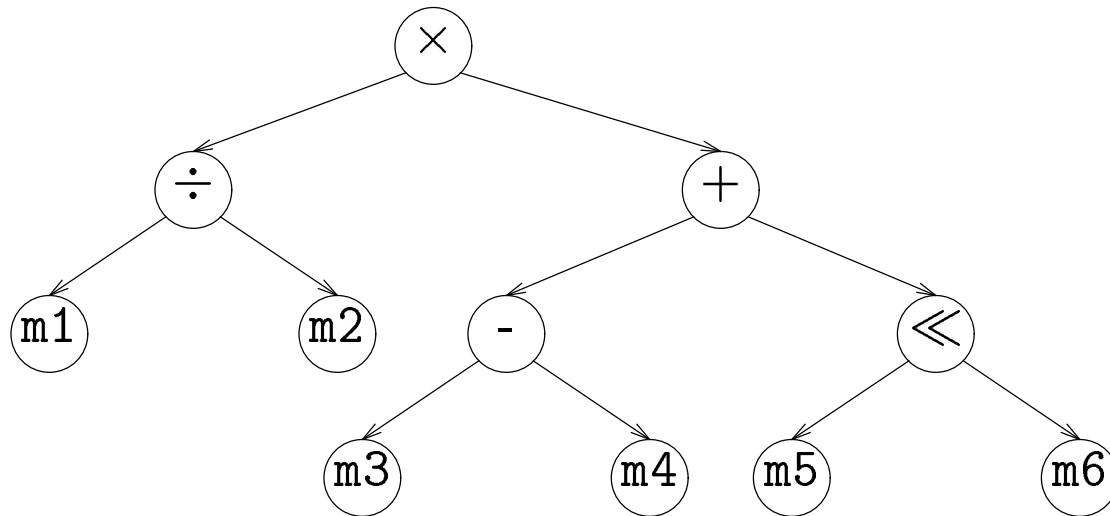
---



*Canonical ordering*

<i>Operators</i>		<i>Loads</i>	
1.	sub	1.	load m3
2.	shift	2.	load m4
3.	add	3.	load m5
4.	div	4.	load m6
5.	mult	5.	load m1
		6.	load m2

## Example



	Sethi-Ullman	DLS(3)	DLS(4)
1.	load m3, r1	load m3, r1	load m3, r1
2.	load m4, r2	load m4, r2	load m4, r2
3.	<i>-stall-</i>	load m5, r3	load m5, r3
4.	sub r1, r2, r2	sub r1, r2, r2	load m6, r4
5.	load m5, r1	load m6, r1	sub r1, r2, r2
6.	load m6, r3	<i>-stall-</i>	load m1, r1
7.	<i>-stall-</i>	shift r1, r3, r3	shift r3, r4, r4
8.	shift r1, r3, r3	load m1, r3	load m2, r3
9.	add r2, r3, r3	add r2, r1, r1	add r2, r4, r4
10.	load m1, r1	load m2, r2	div r1, r3, r3
11.	load m2, r2	<i>-stall-</i>	mult r4, r3, r3
12.	<i>-stall-</i>	div r3, r2, r2	
13.	div r1, r2, r2	mult r1, r2, r2	
14.	mult r3, r2, r2		

## Limitations

---

### Input

(*like* Sethi-Ullman)

- handles *trees*, not *dags*
- limited to a single basic block
- values not kept in registers

### Output

- $delay > 1 \Rightarrow$  optimality not guaranteed
- non-constant *delay* causes deeper problems

### Strengths

- fast, simple algorithm
- clever metric for spilling
- no excuse to do worse

*This work raises the bar for non-optimizing compilers*

## Code generator generators

---

### Automating the process

- would like a description-based tool
- machine description + IR description give code generator (cg)
- resulting cg should produce great code
- resulting cg should run quickly

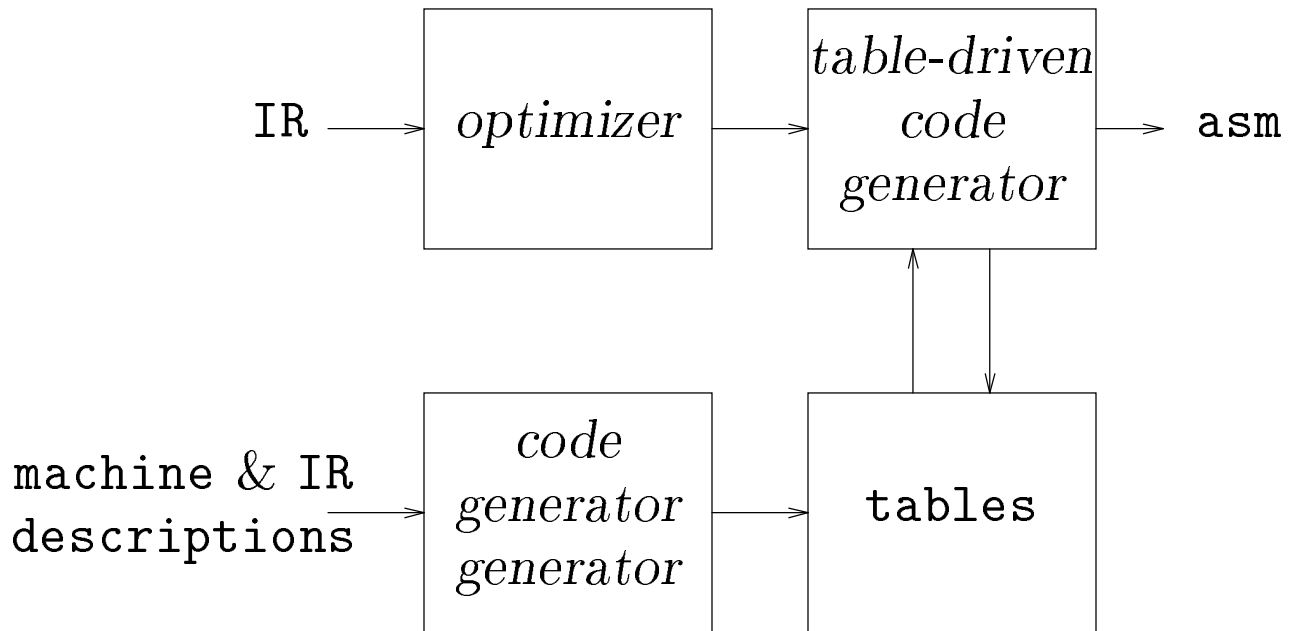
### Two major schools

- tree pattern matching
- instruction matching

# Code generator generators

---

The big picture



*This scheme should look familiar*

## Tree pattern matching

---

Assume that the program is represented as a set of trees.

### *Tree rewriting schemes (BURS)*

- machine description is
  1. mapping of subtree into single node
  2. associated code (to be emitted)
  
- *example pattern:*
  - $r_i \leftarrow + a b$
  - `{load r1,a; load r2,b; add r1,r1,r2}`
  
- paradigm is
  - find a pattern to match subtree
  - replace *rhs* pattern with *lhs* node
  - emit the associated code

## Tree rewriting schemes

---

### Several basic techniques

- work from a simple tree walk
  - depth-first traversal
  - simple local choice criterion
- adopt Aho & Corasick string matching (TWIG)
  - matches multiple string patterns
  - translate to/from linear form
- adopt Aho & Johnson (dynamic programming)
  - run rewriting and cost computation concurrently
  - choose low-cost alternative at each point
- use a real tree pattern matching algorithm
  - generate all subtree matches concurrently
  - pick the best overall match

## Tree parsing schemes

---

### Use LR parsers

- encode pattern matching into parsing problem
  - use well understood technology
  - write grammar to describe target machine
  
- reductions emit code
  - attributed-style specification
  - lots of contextual knowledge available
  
- grammars are *very* ambiguous
  - reduce/reduce  $\Rightarrow$  pick longer reduction
  - shift/reduce  $\Rightarrow$  shift
  
- linear time scheme!

## Instruction matching

---

Assume the program is represented in some low-level IR.

### Peephole optimization (§9.9)

- find logically adjacent instructions that can be combined
  - use a very small context (3-10 instructions)
  - combining  $i_1$  and  $i_2 \Rightarrow$  faster  $i_3$
- work at register-transfer language (*rtl*) level
  - machine description in *rtl*
  - low-level IR description in *rtl*
- using pattern matching, synthesize more complex instructions
- useful for implementing many machine dependent optimizations

## Instruction matching

---

### Generating “peephole” code generators

- provide a one-to-one translation (one-to-one) for IR
- add patterns to improve code  
(more complex instructions and addressing modes)

### Training generator

- feed a set of representative programs to the trainer and let it build a table by exhaustive search
- one time expense *(and it is expensive)*
- use a linear time pattern matcher run from the tables produced by the trainer

# Instruction matching

---

## Typical machines

- RT/PC w/o floating point - 70-100 instructions
- MC68020 - millions of possible instructions

## Recent work

- these two camps have merged  
(Proebsting, *92 PLDI*)
- highly efficient BURS systems
- combines tree-matching and peephole

## Compiler design philosophy

---

The Rice Compiler group's philosophy

Instruction selection

- simple pass in the optimizer
- use *global peephole optimization* to find *address modes*
- performed before scheduling & allocation

Perspective

- emphasis on instruction selection is dated (*architecturally*)
- real problems today are instruction scheduling and register allocation
- RISC makes instruction selection simpler

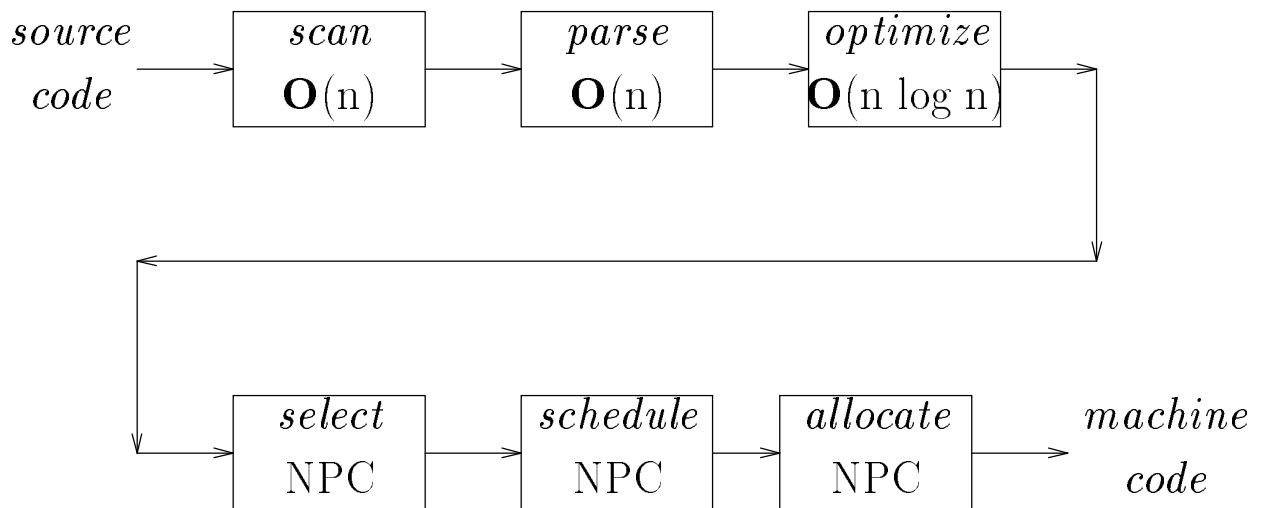
Myth: RISC makes compiling easier

Truth: RISC limits choices & shifts emphasis to more productive (from a speed perspective) issues

# Compiler design philosophy

---

## Compiler structure



## Economics

- spend your time (& money) in *optimize*
- spend your time (& money) in *schedule*
- spend your time (& money) in *allocate*
- do not attempt “optimal” instruction selection

## Match-book version

1. eliminate as many instructions as possible
2. place the remaining ones carefully
3. keep the compiler from adding extras