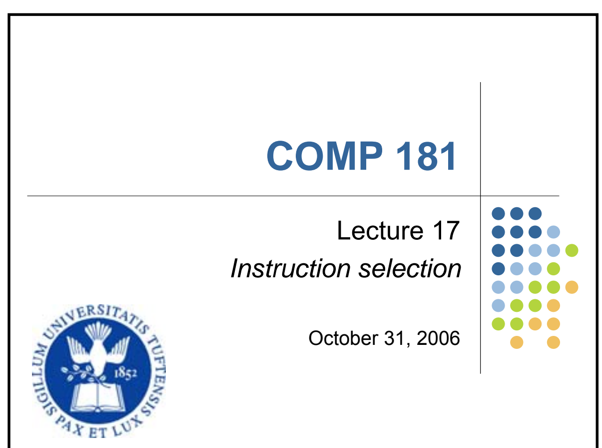
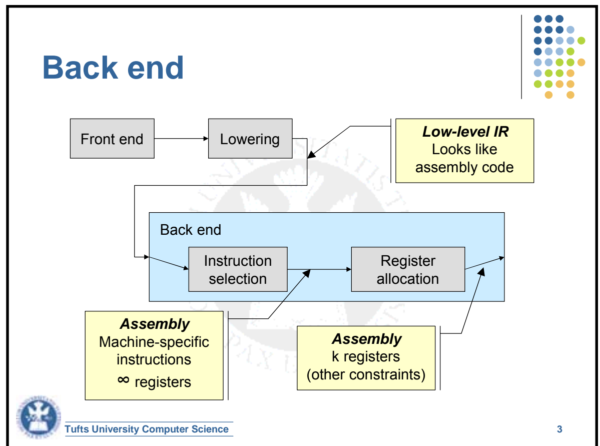
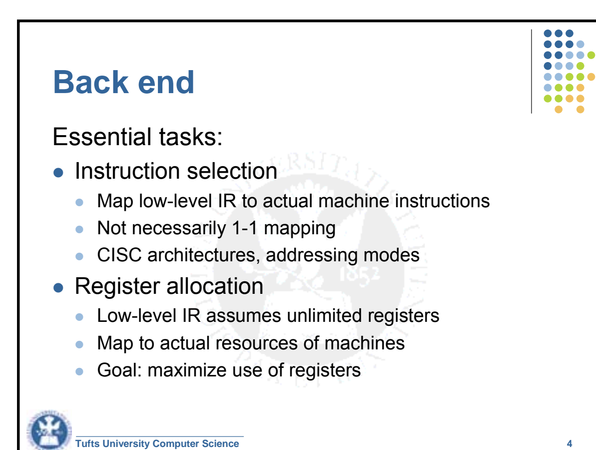
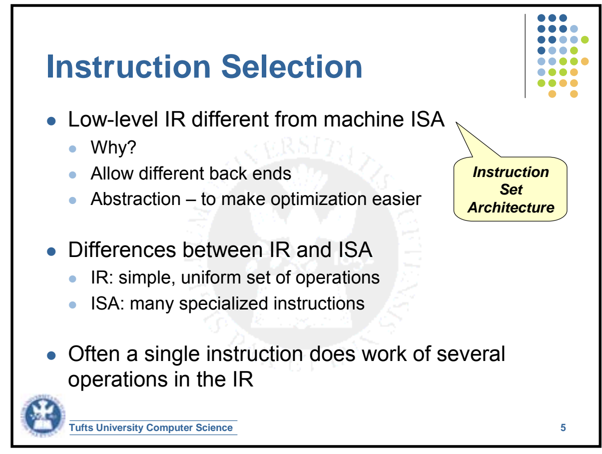
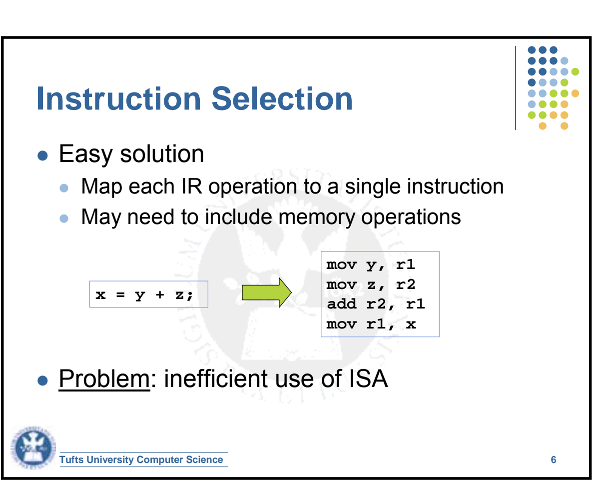

COMP 181
Lecture 17
Instruction selection
October 31, 2006
SIGILLUM UNIVERSITATIS TUFTENSIS
PAX ET LUX
1852

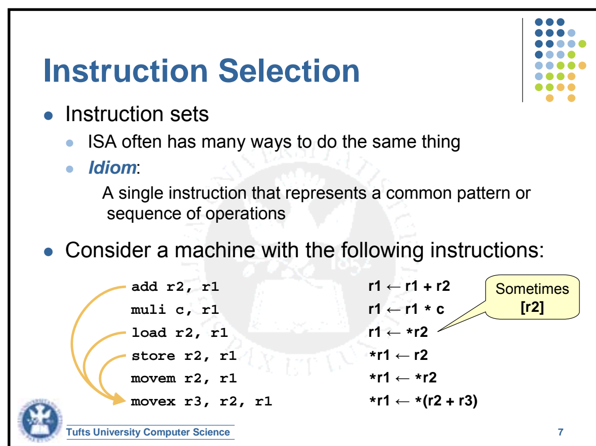
Prelude
What is "Halloween"?
All Hallows Eve – the night before All Hallows Day
Classic holiday formula
Ancient pagan (Celtic) holiday – Samhain, end of summer
Christian holiday – dedication of All Saints Chapel
Dressing up?
Part of honoring dead, warding off evil spirits
Trick-or-treating?
"souling" – ritual begging for soul cakes, prayers
In the US?
Brought from Ireland in early 1900's
Tufts University Computer Science
2


Back end
Front end
Lowering
Low-level IR
Looks like assembly code
Back end
Instruction selection
Register allocation
Assembly
Machine-specific instructions
∞ registers
Assembly
k registers
(other constraints)
Tufts University Computer Science
3

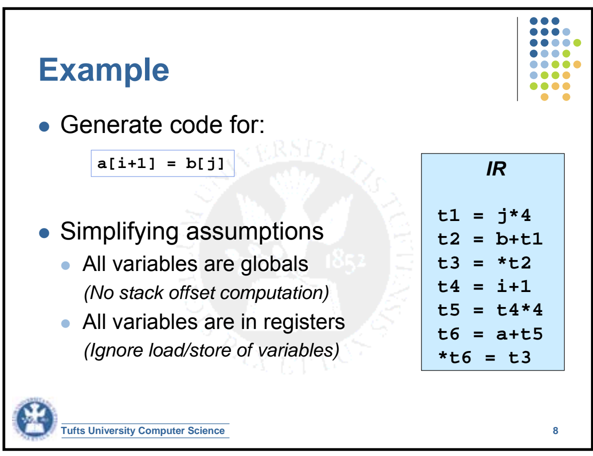

Back end
Essential tasks:
Instruction selection
Map low-level IR to actual machine instructions
Not necessarily 1-1 mapping
CISC architectures, addressing modes
Register allocation
Low-level IR assumes unlimited registers
Map to actual resources of machines
Goal: maximize use of registers
Tufts University Computer Science
4


Instruction Selection
Low-level IR different from machine ISA
Why?
Allow different back ends
Abstraction – to make optimization easier
Instruction Set Architecture
Differences between IR and ISA
IR: simple, uniform set of operations
ISA: many specialized instructions
Often a single instruction does work of several operations in the IR
Tufts University Computer Science
5

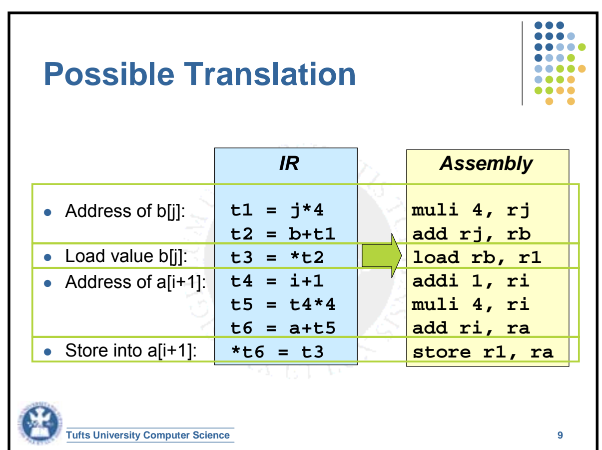

Instruction Selection
Easy solution
Map each IR operation to a single instruction
May need to include memory operations
x = y + z;
mov y, r1
mov z, r2
add r2, r1
mov r1, x
Problem: inefficient use of ISA
Tufts University Computer Science
6



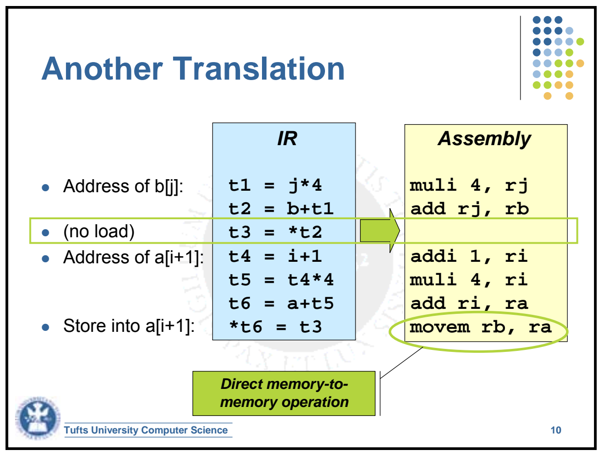
Instruction Selection
Instruction sets
ISA often has many ways to do the same thing
Idiom:
A single instruction that represents a common pattern or sequence of operations
Consider a machine with the following instructions:
add r2, r1
r1 ← r1 + r2
muli c, r1
r1 ← r1 * c
load r2, r1
r1 ← *r2
store r2, r1
*r1 ← r2
movem r2, r1
*r1 ← *r2
movex r3, r2, r1
*r1 ← *(r2 + r3)
Sometimes
[r2]
Tufts University Computer Science
7



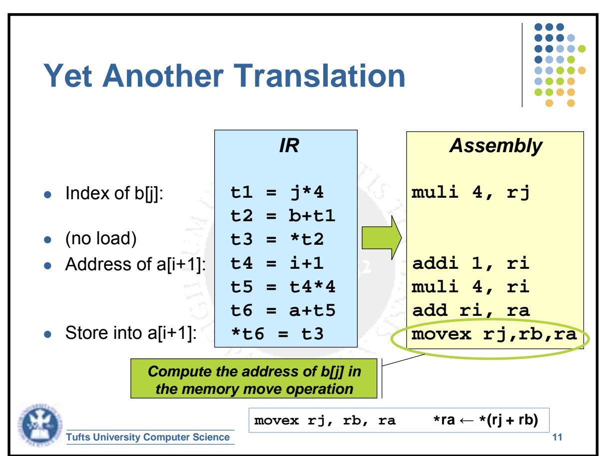
Example
Generate code for:
a[i+1] = b[j]
Simplifying assumptions
All variables are globals
(No stack offset computation)
All variables are in registers
(Ignore load/store of variables)
IR
t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t3
Tufts University Computer Science
8



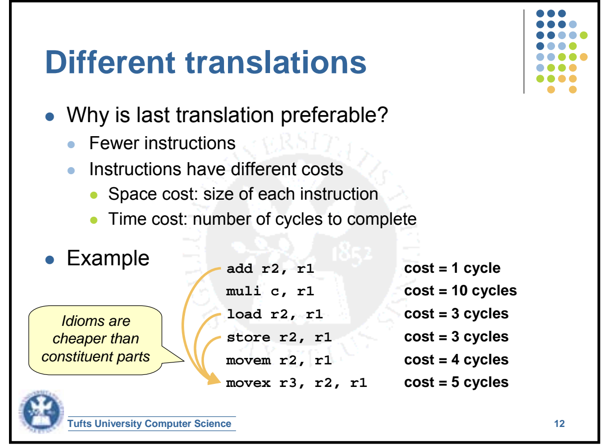
Possible Translation
IR
Assembly
Address of b[j]:
t1 = j*4
t2 = b+t1
muli 4, rj
add rj, rb
Load value b[j]:
t3 = *t2
load rb, r1
Address of a[i+1]:
t4 = i+1
t5 = t4*4
t6 = a+t5
addi 1, ri
muli 4, ri
add ri, ra
Store into a[i+1]:
*t6 = t3
store r1, ra
Tufts University Computer Science
9



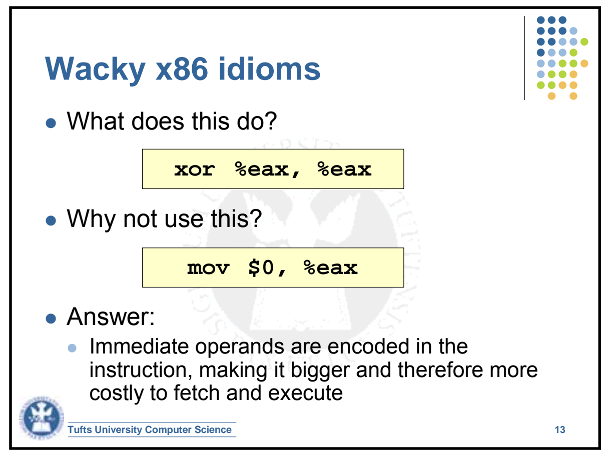
Another Translation
IR
Assembly
Address of b[j]:
t1 = j*4
t2 = b+t1
muli 4, rj
add rj, rb
(no load)
t3 = *t2
Address of a[i+1]:
t4 = i+1
t5 = t4*4
t6 = a+t5
addi 1, ri
muli 4, ri
add ri, ra
Store into a[i+1]:
*t6 = t3
movem rb, ra
Direct memory-to-memory operation
Tufts University Computer Science
10

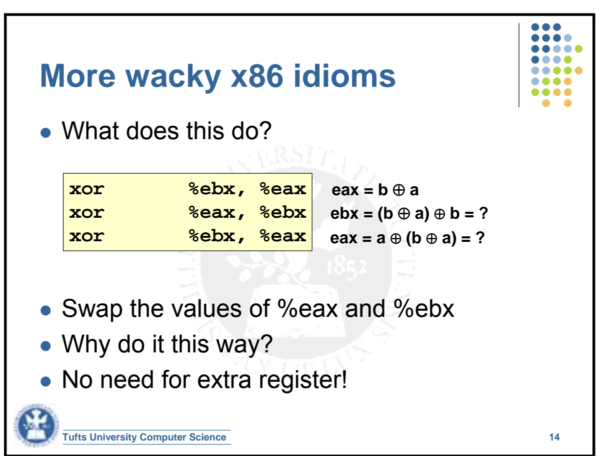


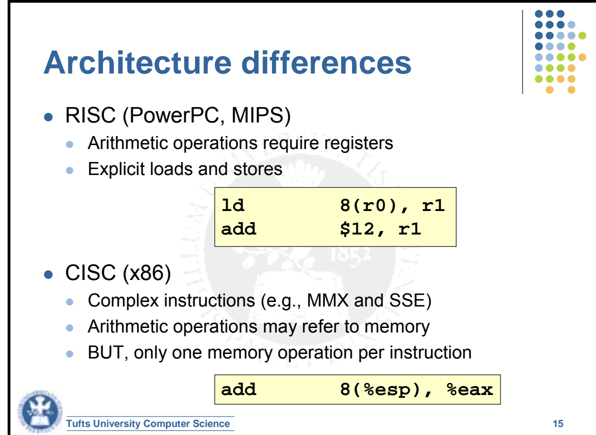
Yet Another Translation
IR
Assembly
Index of b[j]:
t1 = j*4
t2 = b+t1
muli 4, rj
(no load)
t3 = *t2
Address of a[i+1]:
t4 = i+1
t5 = t4*4
t6 = a+t5
addi 1, ri
muli 4, ri
add ri, ra
Store into a[i+1]:
*t6 = t3
movex rj,rb,ra
Compute the address of b[j] in the memory move operation
movex rj, rb, ra
*ra ← *(rj + rb)
Tufts University Computer Science
11

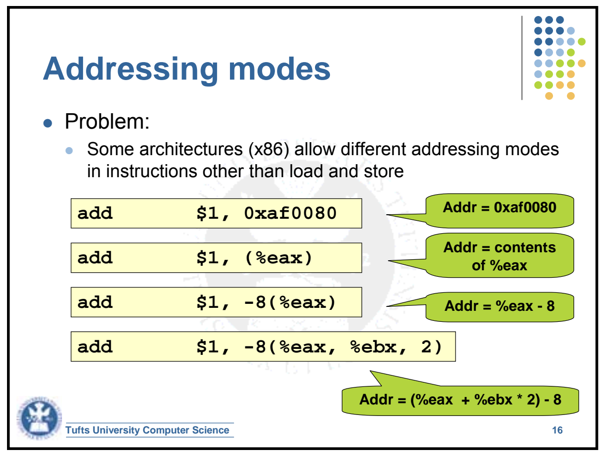


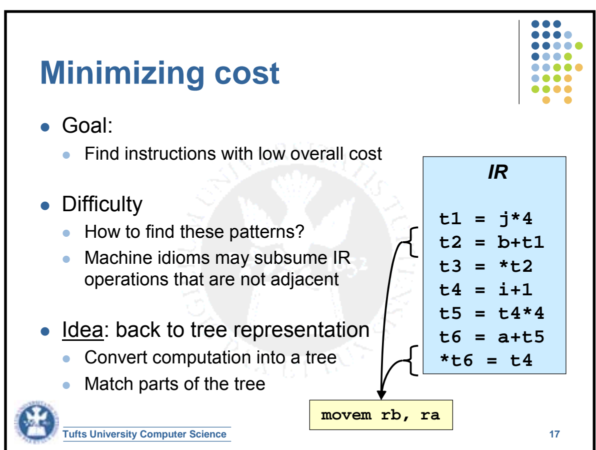
Different translations
Why is last translation preferable?
Fewer instructions
Instructions have different costs
Space cost: size of each instruction
Time cost: number of cycles to complete
Example
add r2, r1
cost = 1 cycle
muli c, r1
cost = 10 cycles
load r2, r1
cost = 3 cycles
store r2, r1
cost = 3 cycles
movem r2, r1
cost = 4 cycles
movex r3, r2, r1
cost = 5 cycles
Idioms are cheaper than constituent parts
Tufts University Computer Science
12

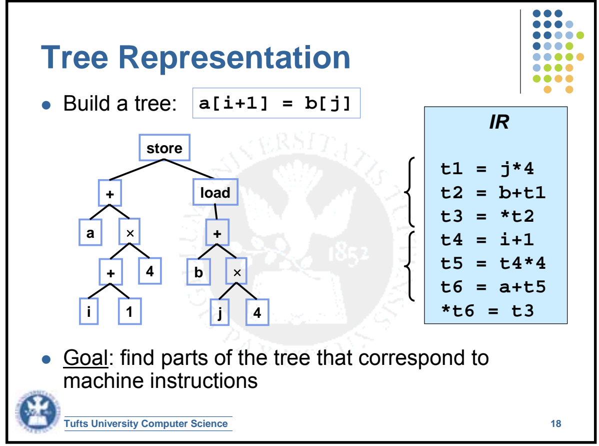

Wacky x86 idioms
What does this do?
xor %eax, %eax
Why not use this?
mov $0, %eax
Answer:
Immediate operands are encoded in the instruction, making it bigger and therefore more costly to fetch and execute
Tufts University Computer Science
13

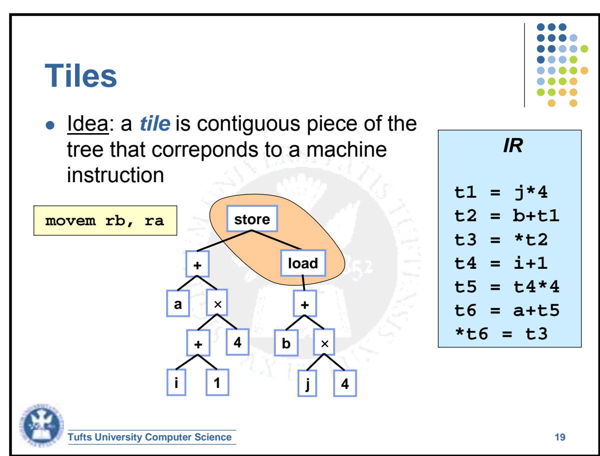

More wacky x86 idioms
What does this do?
xor %ebx, %eax
xor %eax, %ebx
xor %ebx, %eax
eax = b ⊕ a
ebx = (b ⊕ a) ⊕ b = ?
eax = a ⊕ (b ⊕ a) = ?
Swap the values of %eax and %ebx
Why do it this way?
No need for extra register!
Tufts University Computer Science
14

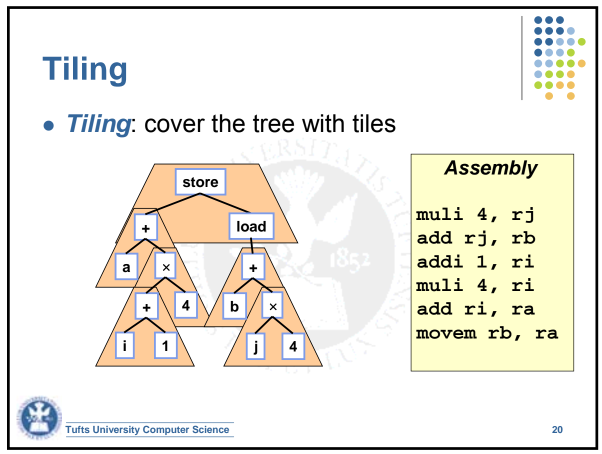

Architecture differences
RISC (PowerPC, MIPS)
Arithmetic operations require registers
Explicit loads and stores
ld 8(r0), r1
add $12, r1
CISC (x86)
Complex instructions (e.g., MMX and SSE)
Arithmetic operations may refer to memory
BUT, only one memory operation per instruction
add 8(%esp), %eax
Tufts University Computer Science
15

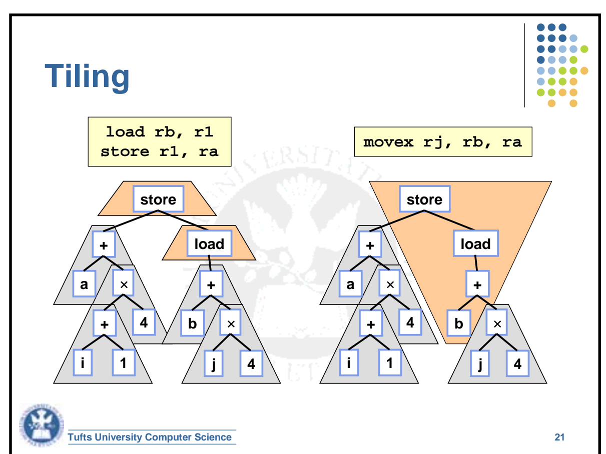

Addressing modes
Problem:
Some architectures (x86) allow different addressing modes in instructions other than load and store
add $1, 0xaf0080
Addr = 0xaf0080
add $1, (%eax)
Addr = contents of %eax
add $1, -8(%eax)
Addr = %eax - 8
add $1, -8(%eax, %ebx, 2)
Addr = (%eax + %ebx * 2) - 8
Tufts University Computer Science
16

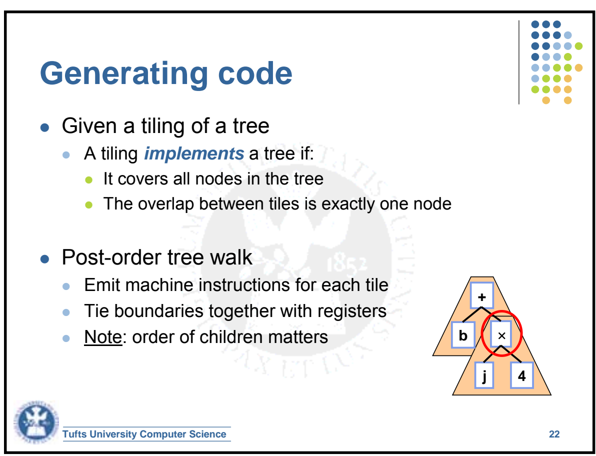

Minimizing cost
Goal:
Find instructions with low overall cost
Difficulty
How to find these patterns?
Machine idioms may subsume IR operations that are not adjacent
Idea: back to tree representation
Convert computation into a tree
Match parts of the tree
IR
t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t4
movem rb, ra
Tufts University Computer Science
17

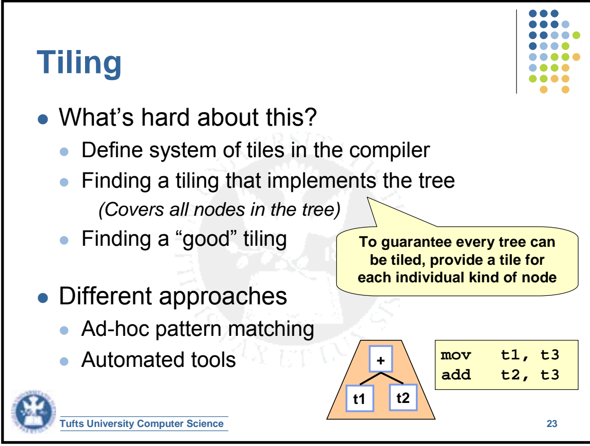

Tree Representation
Build a tree: a[i+1] = b[j]
store
+
load
a
×
+
+
4
b
×
i
1
j
4
IR
t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t3
Goal: find parts of the tree that correspond to machine instructions
Tufts University Computer Science
18

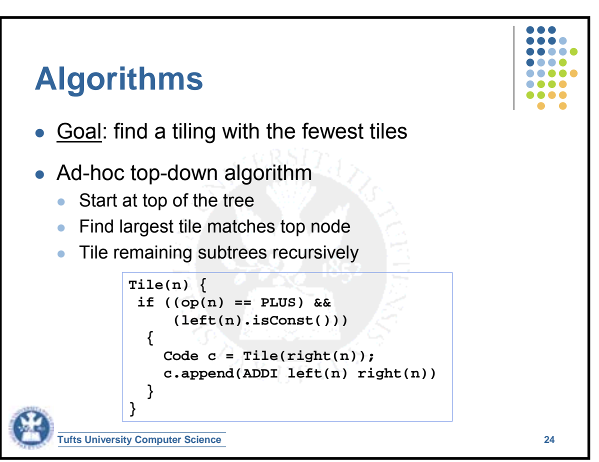

Tiles
Idea: a tile is contiguous piece of the tree that correponds to a machine instruction
movem rb, ra
store
load
+
a
×
+
4
i
1
+
b
×
j
4
IR
t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t3
Tufts University Computer Science
19


Tiling
Tiling: cover the tree with tiles
store
load
+
a
×
+
4
i
1
+
b
×
j
4
Assembly
muli 4, rj
add rj, rb
addi 1, ri
muli 4, ri
add ri, ra
movem rb, ra
Tufts University Computer Science
20


Tiling
load rb, r1
store r1, ra
movex rj, rb, ra
store
load
+
a
×
+
4
i
1
+
b
×
j
4
store
load
+
a
×
+
4
i
1
+
b
×
j
4
Tufts University Computer Science
21


Generating code
Given a tiling of a tree
A tiling implements a tree if:
It covers all nodes in the tree
The overlap between tiles is exactly one node
Post-order tree walk
Emit machine instructions for each tile
Tie boundaries together with registers
Note: order of children matters
+
b
×
j
4
Tufts University Computer Science
22


Tiling
What's hard about this?
Define system of tiles in the compiler
Finding a tiling that implements the tree
(Covers all nodes in the tree)
Finding a "good" tiling
To guarantee every tree can be tiled, provide a tile for each individual kind of node
Different approaches
Ad-hoc pattern matching
Automated tools
+
t1
t2
mov t1, t3
add t2, t3
Tufts University Computer Science
23


Algorithms
Goal: find a tiling with the fewest tiles
Ad-hoc top-down algorithm
Start at top of the tree
Find largest tile matches top node
Tile remaining subtrees recursively
Tile(n) {
if ((op(n) == PLUS) &&
(left(n).isConst()))
{
Code c = Tile(right(n));
c.append(ADDI left(n) right(n))
}
}
Tufts University Computer Science
24

Ad-hoc algorithm

- <u>Problem</u>: what does tile size mean?
  - Not necessarily the best fastest code
    (Example: multiply vs add)
  - How to include cost?
- <u>Idea</u>:
  - Total cost of a tiling is sum of costs of each tile
- <u>Goal</u>: find a minimum cost tiling

Including cost

- Algorithm:
  - For each node, find minimum total cost tiling for that node and the subtrees below
- Key:
  - Once we have a minimum cost for subtree, can find minimum cost tiling for a node by trying out all possible tiles matching the node
- Use dynamic programming

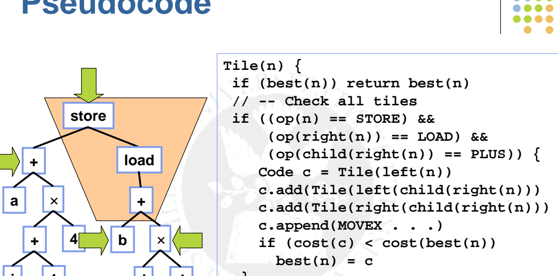
Dynamic programming

- Idea
  - For problems with ***optimal substructure***
  - Compute optimal solutions to sub-problems
  - Combine into an optimal overall solution
- How does this help?
  - Use ***memoization***:
    Save previously computed solutions to sub-problems
  - Sub-problems recur many times
  - Can work top-down or bottom-up

Recursive algorithm

- Memoization
  - For each subtree, record best tiling in a table
  - (*Note*: need a quick way to find out if we've seen a subtree before – some systems use DAGs instead of trees)
- At each node
  - First check table for optimal tiling for this node
  - If none, try all possible tiles, remember lowest cost
  - Record lowest cost tile in table
  - Greedy, top-down algorithm
- We can emit code from table

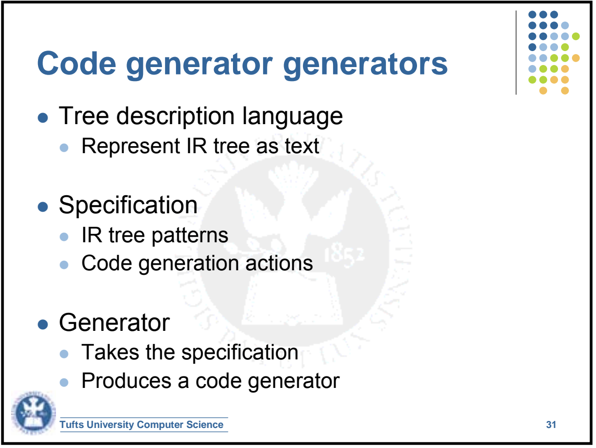
Pseudocode

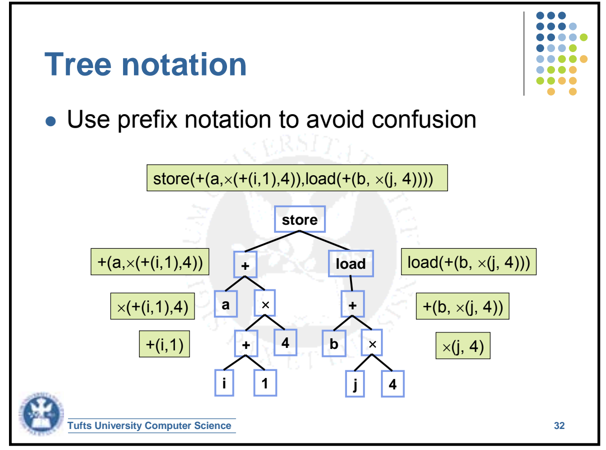


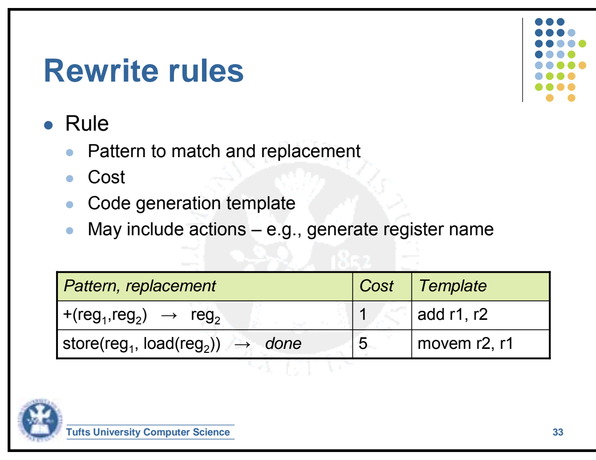

```
Tile(n) {
 if (best(n)) return best(n)
 // -- Check all tiles
 if ((op(n) == STORE) &&
     (op(right(n)) == LOAD) &&
     (op(child(right(n)) == PLUS)) {
    Code c = Tile(left(n))
    c.add(Tile(left(child(right(n)))
    c.add(Tile(right(child(right(n)))
    c.append(MOVEX . . .)
    if (cost(c) < cost(best(n))
      best(n) = c
  }
 // . . . and all other tiles . . .
 return best(n)
}
```


Ad-hoc algorithm

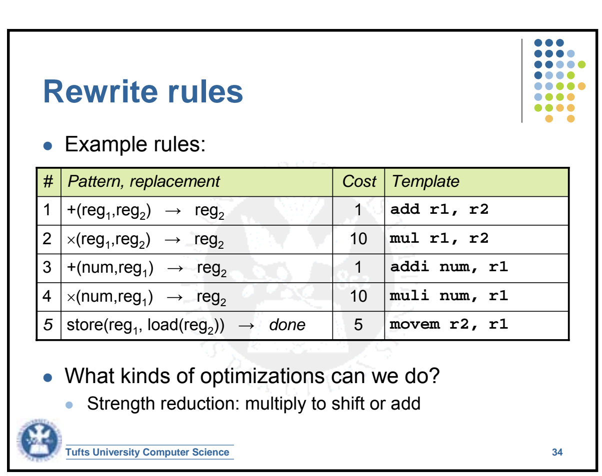
- Problem:
  - Hard-codes the tiles in the code generator
- Alternative:
  - Define tiles in a separate specification
  - Use a generic tree pattern matching algorithm to compute tiling
  - Tools: *code generator generators*
  - Probably overkill for RISC



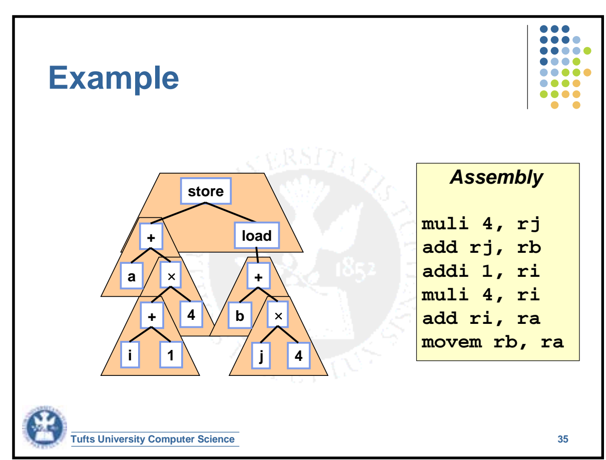


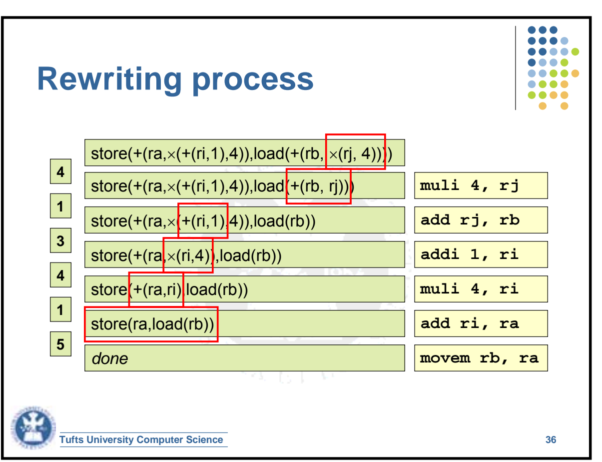


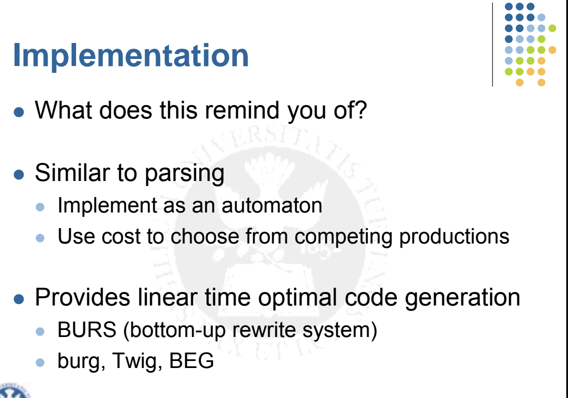
Pattern, replacement	Cost	Template
$+(reg_1, reg_2) \rightarrow reg_2$	1	add r1, r2
$store(reg_1, load(reg_2)) \rightarrow$ *done*	5	movem r2, r1

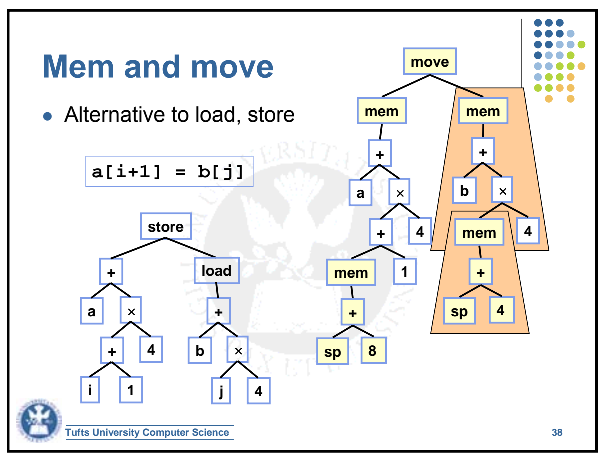


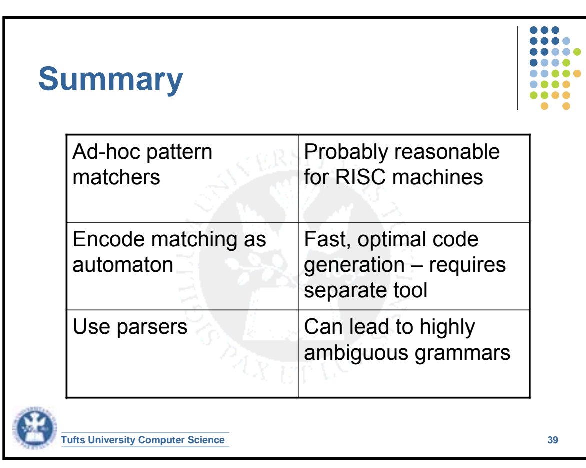
#	Pattern, replacement	Cost	Template
1	$+(reg_1, reg_2) \rightarrow reg_2$	1	`add r1, r2`
2	$\times(reg_1, reg_2) \rightarrow reg_2$	10	`mul r1, r2`
3	$+(num, reg_1) \rightarrow reg_2$	1	`addi num, r1`
4	$\times(num, reg_1) \rightarrow reg_2$	10	`muli num, r1`
5	$store(reg_1, load(reg_2)) \rightarrow$ *done*	5	`movem r2, r1`

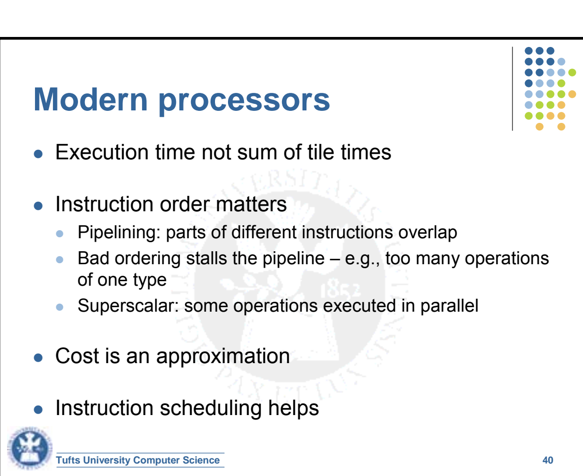


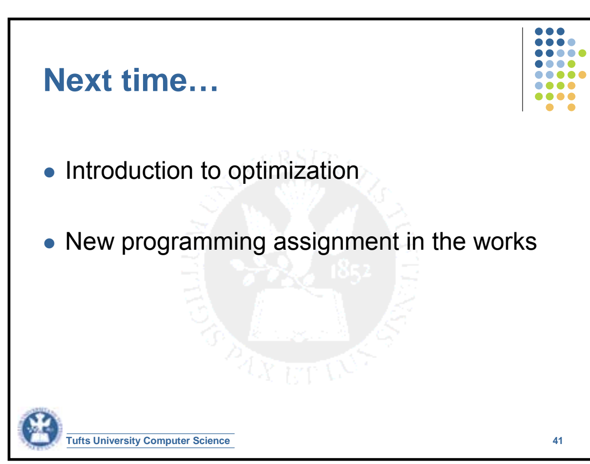



Implementation
What does this remind you of?
Similar to parsing
Implement as an automaton
Use cost to choose from competing productions
Provides linear time optimal code generation
BURS (bottom-up rewrite system)
burg, Twig, BEG
Tufts University Computer Science
37


Mem and move
Alternative to load, store
a[i+1] = b[j]
store
load
move
mem
mem
mem
mem
sp
8
sp
4
Tufts University Computer Science
38


Summary
Ad-hoc pattern matchers
Probably reasonable for RISC machines
Encode matching as automaton
Fast, optimal code generation – requires separate tool
Use parsers
Can lead to highly ambiguous grammars
Tufts University Computer Science
39


Modern processors
Execution time not sum of tile times
Instruction order matters
Pipelining: parts of different instructions overlap
Bad ordering stalls the pipeline – e.g., too many operations of one type
Superscalar: some operations executed in parallel
Cost is an approximation
Instruction scheduling helps
Tufts University Computer Science
40


Next time…
Introduction to optimization
New programming assignment in the works
Tufts University Computer Science
41