CS536

Instruction Selection and Processor Type

A Sahu

CSE, IIT Guwahati

Outline

- Instruction Selection
- Methods for Instruction Selection

Three Basic Back-End Optimization

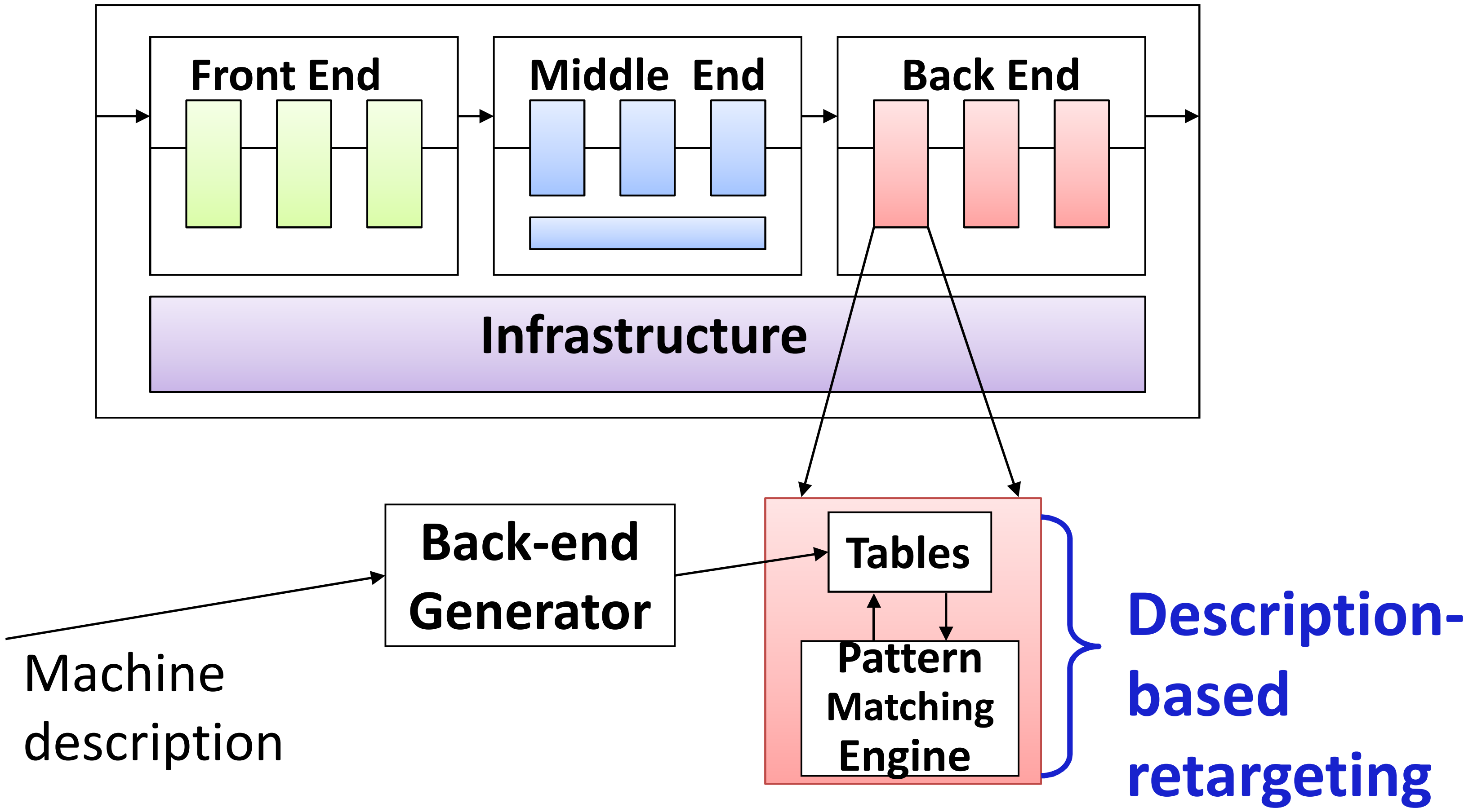
- Instruction selection
  - Mapping IR into assembly code
  - Assume a fixed storage mapping & code shape
  - Combining operations, using address modes
- Instruction scheduling
  - Reordering operations to hide latencies
  - Assume a fixed program (set of operations)
  - Changes demand for registers
- Register allocation
  - Deciding which values will reside in registers
  - Changes the storage mapping may add false sharing
  - Concerns about placement of data & memory operations

Instruction Selection

Objectives

- Introduce the complexity and importance of instruction selection
- Study practical issues and solutions

Instruction Selection: Re-targetable



Machine description should also help with scheduling & allocation

This is simplistic but useful view

Processor Types

- CISC: X86, X86-64 : **3000 instructions types**
- RISC: 80 INS; simple addressing mode
  - Similar to compiler LLIR
  - MIPS, ARM, PowerPC
- Application Specific Instruction Processor (ASIP) or Domain Specific Inst. Processor
  - **Digital Signal Processor / MAC Ins**
  - **Image Processor**
  - **Network Processor //Match+Action, P4 Language**
  - **Crypto Processor**
  - **Media Processor (MMX)**
  - **Vector/Matrix Processor (Google TPU)**
  - **GPU**
  - **Tensor Core**

Complexity of Instruction Selection

Modern computers have many ways to do things.

Consider a register-to-register copy

- Obvious operation is: mv r_j, r_i
- Many others exist
  - **add rj, ri,0** **sub rj, ri, 0** **rshiftI rj, ri, 0**

    mul rj, ri, 1 **or rj, ri, 0** **divI rj, r, 1**

    xor rj, ri, 0 **others …**

Complexity of Instruction Selection (Cont.)

- Multiple addressing modes
- Each alternate sequence has its cost
  - Complex ops (mult, div): several cycles
  - Memory ops: latency vary
- Sometimes, cost is context related
- Use under-utilized FUs
- Dependent on objectives: speed, power, code size

Complexity of Instruction Selection (Cont.)

- **Additional constraints on specific operations**
  - Load/store multiple words: contiguous registers
  - Multiply: need special register Accumulator
- **Interaction between instruction selection, instruction scheduling, and register allocation**
  - For scheduling, instruction selection predetermines latencies and function units
  - For register allocation, instruction selection pre-colors some variables. e.g. non-uniform registers (such as registers for multiplication)

Instruction Selection Techniques

- **Tree Pattern-Matching**
  - Tree-oriented IR suggests pattern matching on trees
  - Tree-patterns as input, matcher as output
  - Each pattern maps to a target-machine instruction sequence
  - Use dynamic programming or bottom-up rewrite systems
- **Peephole-based Matching**
  - Linear IR suggests using some sort of string matching
  - Inspired by peephole optimization
  - Strings as input, matcher as output
  - Each string maps to a target-machine instruction sequence
- **In practice, both work well; matchers are quite different.**

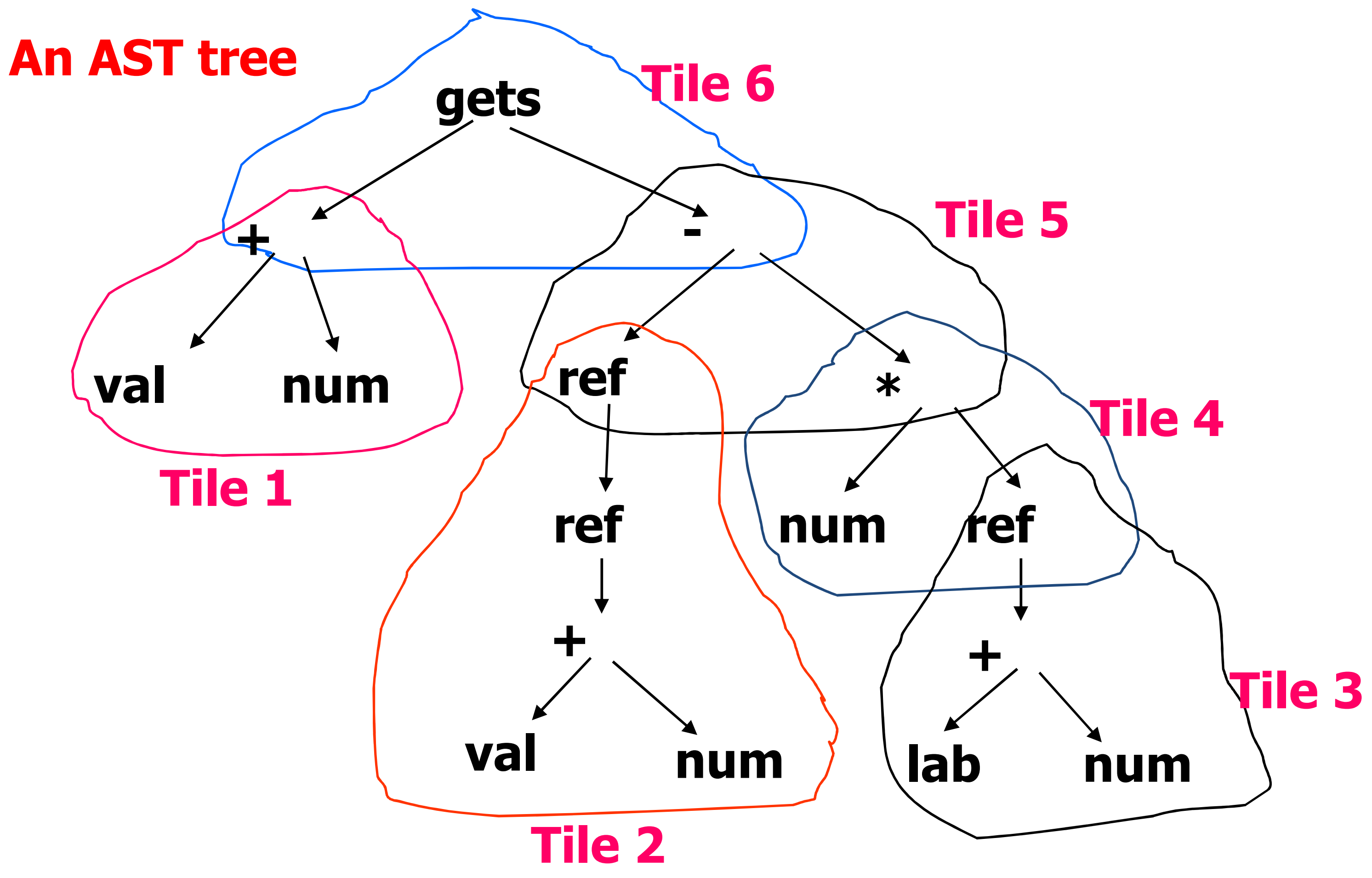
A Simple Tree-Walk Code Generation Method

- Assume starting with a Tree-like IR
- Starting from the root, recursively walking through the tree
- At each node use a simple (unique) rule to generate a low-level instruction

Tree Pattern-Matching

- **Assumptions**
  - tree-like IR - an AST
  - Assume each subtree of IR – there is a corresponding set of *tree patterns (or "operation trees"* - low-level abstract syntax tree)
- **Problem formulation:** Find a best mapping of the AST to operations by "tiling" the AST with operation trees (where tiling is a collection of (AST-node, operation-tree) pairs).

Tile AST



An AST tree
gets
+
-
val
num
ref
*
ref
num
ref
+
+
val
num
lab
num
Tile 1
Tile 2
Tile 3
Tile 4
Tile 5
Tile 6

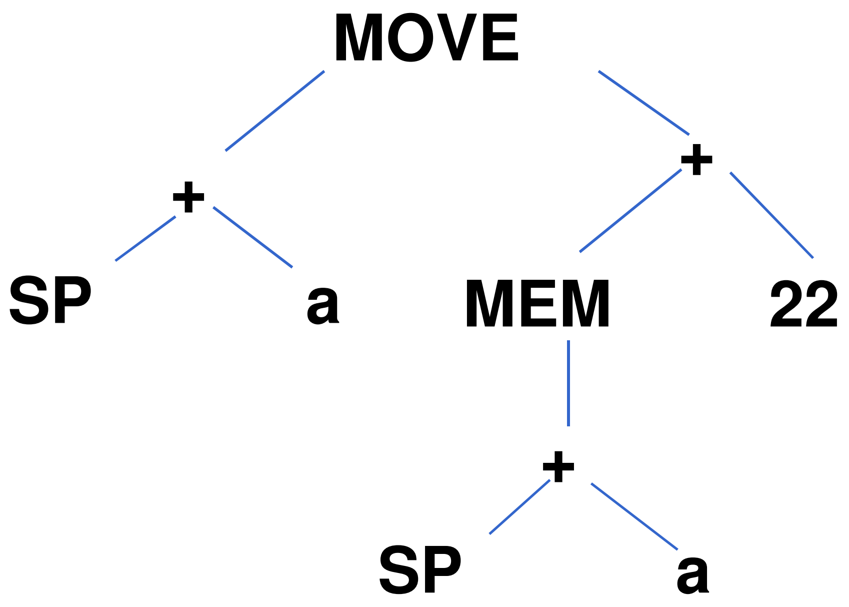
Tile AST with Operation Trees

Goal is to “tile” AST with operation trees.

- A tiling is collection of <ast-node, op-tree > pairs
  - **ast-node** is a node in the AST
  - **op-tree** is an operation tree
  - **<ast-node, op-tree>** means that op-tree could implement the subtree at ast-node
- A tiling ‘implements” an AST if it covers every node in the AST and the overlap between any two trees is limited to a single node
  - **<ast-node, op-tree>** tiling means ast-node is also covered by a leaf in another operation tree in the tiling, unless it is the root
  - Where two operation trees meet, they must be compatible (expect the value in the same location)

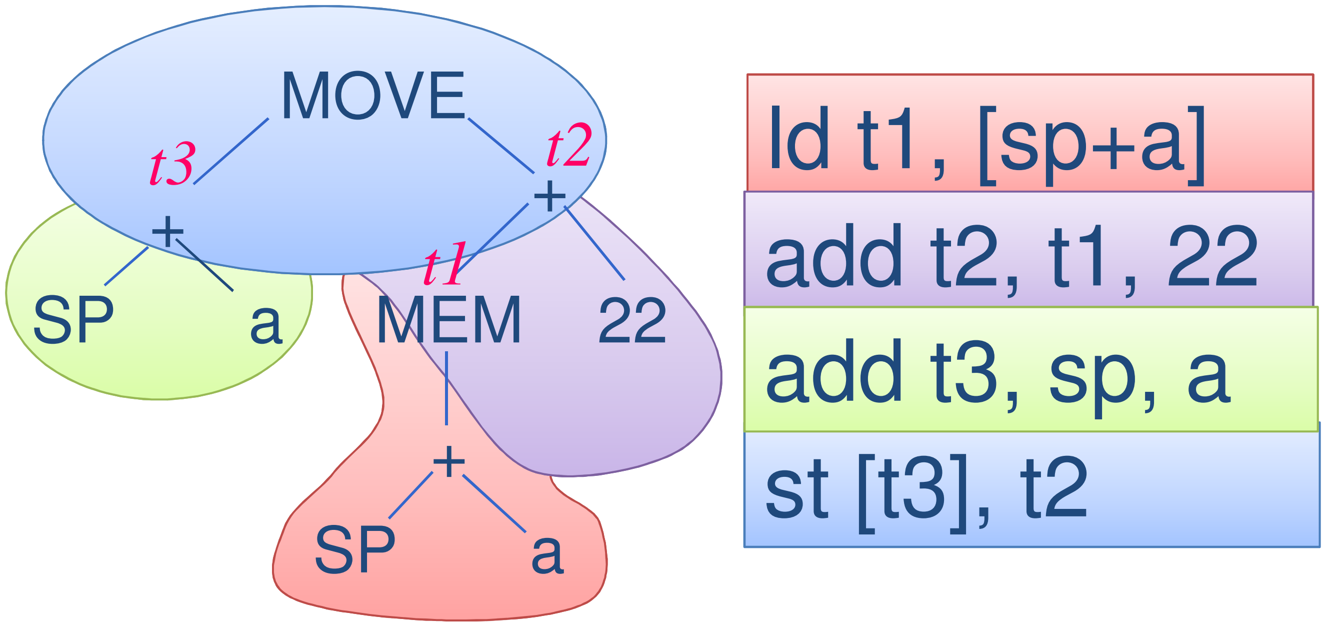
Tree Walk by Tiling: An Example

a = a + 22;



Tree Walk by Tiling: An Example

a = a + 22;



ld t1, [sp+a]

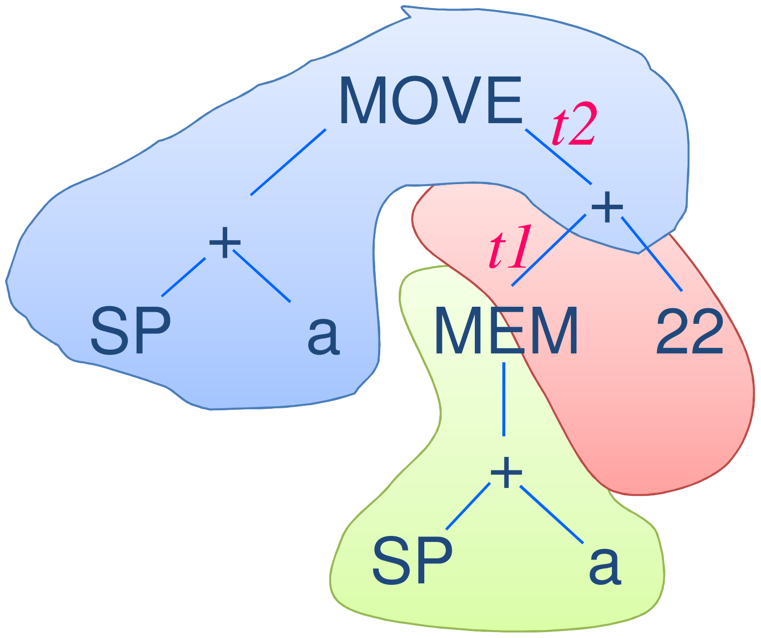
add t2, t1, 22

add t3, sp, a

st [t3], t2

Example: An Alternative

a = a + 22;



ld t1, [sp+a]
add t2, t1, 22
st [sp+a], t2

Finding Matches to Tile the Tree

- Compiler writer connects operation trees to AST subtrees
  - Provides a set of rewrite rules
  - Encode tree syntax, in linear form
  - Associated with each is a code template

Generating Code in Tilings

Given a tiled tree

- Postorder treewalk, with node-dependent order for children
    - Do right child before its left child
- Emit code sequence for tiles, in order
- Tie boundaries together with register names
    - Can incorporate a "real" register allocator or can simply use "NextRegister++" approach

Optimal Tilings

- Best tiling corresponds to least cost instruction sequence
- Optimal tiling
  - no two adjacent tiles can be combined to a tile of lower cost

Dynamic Programming for Optimal Tiling

- For a node *x*, let *f(x)* be the cost of the optimal tiling for the whole expression tree rooted at *x*. Then

$$f(x) = \min_{\forall \text{tile } T \text{ covering } x} \left(\text{cost}(T) + \sum_{\forall \text{child } y \text{ of tile } T} f(y) \right)$$

Dynamic Programming for Optimal Tiling (Con't)

- Maintain a table: node x→ the optimal tiling covering node x and its cost
- Start from root recursively:
  - check in table for optimal tiling for this node
  - If not computed, try all possible tiling and find the optimal, store lowest-cost tile in table and return
- Finally, use entries in table to emit code

Peephole-based Matching

- Basic idea inspired by peephole optimization
- Compiler can discover local improvements locally
    - Look at a small set of adjacent operations
    - Move a “peephole” over code & search for improvement

A Classic example is store followed by load

Original code

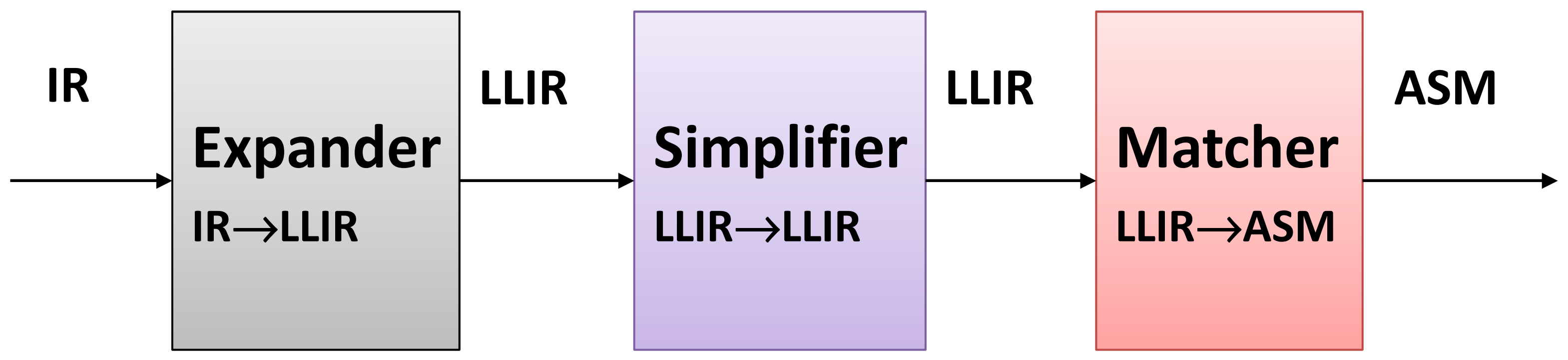
```
st $r1,($r0)
ld $r2,($r0)
```

Improved code

```
st $r1,($r0)
move $r2,$r1
```

Implementing Peephole Matching

- Early systems used limited set of hand-coded patterns
- Window size ensured quick processing
- Modern peephole instruction selectors break problem into three tasks



LLIR: Low Level IR

ASM: Assembly Code