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

Instruction Selection
- Low-level IR different from machine ISA
  - Why?
  - Allow different back ends
  - Abstraction – to make optimization easier
- 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

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:

Example
- Generate code for:

```plaintext
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)

```plaintext
LIR

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 = j \times 4 )</td>
<td>( t_1 = j \times 4 )</td>
</tr>
<tr>
<td>( t_2 = b + t_1 )</td>
<td>( t_2 = b + t_1 )</td>
</tr>
<tr>
<td>( t_3 = t_2 )</td>
<td>( t_3 = t_2 )</td>
</tr>
<tr>
<td>( t_4 = i + 1 )</td>
<td>( t_4 = i + 1 )</td>
</tr>
<tr>
<td>( t_5 = t_4 \times 4 )</td>
<td>( t_5 = t_4 \times 4 )</td>
</tr>
<tr>
<td>( t_6 = a + t_5 )</td>
<td>( t_6 = a + t_5 )</td>
</tr>
<tr>
<td>( *t_6 = t_3 )</td>
<td>( *t_6 = t_3 )</td>
</tr>
</tbody>
</table>
```
Possible Translation

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

Another Translation

- Address of b[j]:
  - t1 = j*4
  - t2 = b + t1
- Load value b[j]:
  - (no load)
- Address of a[i+1]:
  - t4 = i + 1
  - t5 = t4 * 4
  - t6 = a + t5
- Store into a[i+1]:
  - IR: t1 = j*4
t1 = j*4  
t2 = b + t1  
t3 = *t2  
t4 = i + 1  
t5 = t4 * 4  
t6 = a + t5
  - Assembly: muli 4, rj
  - add r1, rb
  - load rb, r1
  - addi 1, ri
  - muli 4, ri
  - add ri, ra
  - movem rb, ra
  - Direct memory-to-memory operation

Yet Another Translation

- Index of b[j]:
  - t1 = j*4
  - t2 = b + t1
  - t3 = *t2
  - t4 = i + 1
  - t5 = t4 * 4
  - t6 = a + t5
- (no load)
- Address of a[i+1]:
  - *t6 = t3
- Store into a[i+1]:
  - IR: t1 = j*4
t1 = j*4  
t2 = b + t1  
t3 = *t2  
t4 = i + 1  
t5 = t4 * 4  
t6 = a + t5
  - Assembly: muli 4, rj
  - add i, r1
  - muli 4, ri
  - add ri, ra
  - movex rj, rb, ra
  - Compute the address of b[j] in the memory move operation

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, ri
  - 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

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

More wacky x86 idioms

- What does this do?
  - xor %ebx, %eax
  - eax = b @ a
  - ebx = (b @ a) @ b = ?
  - xor %eax, %ebx
  - eax = a @ (b @ a) = ?
  - Swap the values of %eax and %ebx
  - Why do it this way?
  - No need for extra register!
Architecture differences

- **RISC (PowerPC, MIPS)**
  - Arithmetic operations require registers
  - Explicit loads and stores
  - Example: `ld 8(r0), r1` (load 8 bytes starting at address r0 into register r1)
  - `add $12, r1` (add 12 to register r1)

- **CISC (x86)**
  - Complex instructions (e.g., MMX and SSE)
  - Arithmetic operations may refer to memory
  - BUT, only one memory operation per instruction
  - Example: `add 8(%esp), %eax`

Addressing modes

- **Problem:**
  - Some architectures (x86) allow different addressing modes in instructions other than load and store

- Example:
  - `add %1, 0xaf0080`
  - `add %1, (%eax)`
  - `add %1, -8(%eax)`
  - `add %1, -8(%eax, %ebx, 2)`

Address = 0xaf0080
Address = contents of %eax
Address = %eax - 8
Address = (%eax + %ebx * 2) - 8

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

Tree Representation

- **Build a tree:**
  - `a[i+1] = b[j]`

- Example:
  - `t1 = j*4`
  - `t2 = b+t1`
  - `t3 = *t2`
  - `t4 = i+1`
  - `t5 = t4 + t5`
  - `t6 = a+t5`
  - `t6 = t4`

- **Goal:** find parts of the tree that correspond to machine instructions

Tiles

- **Idea:** a **tile** is contiguous piece of the tree that corresponds to a machine instruction

- Example:
  - `movem rb, ra`

Tiling

- **Tiling:** cover the tree with tiles

- Example:
  - `muli 4, rj` (multiply by 4)
  - `add rj, rb` (add result to rb)
  - `addi i, ri` (add immediate value i to ri)
  - `muli 4, ri` (multiply by 4)
  - `add ri, ra` (add result to ra)
  - `movem rb, ra` (move data between registers)
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

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
- Different approaches
  - Ad-hoc pattern matching
  - Automated tools

Interesting result (Dias and Ramsey): in general, undecidable

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

\[
\begin{align*}
\text{Tile}(n) \{ \\
\quad & \text{if } (\text{op}(n) == \text{PLUS}) \&\& \text{left}(n).isConst() \} \\
\quad & \{ \text{Code } c = \text{Tile(right}(n)) ; \\
\quad & \hspace{1em} c.\text{append}(\text{ADDI } \text{left}(n) \text{ right}(n)) \} \\
\}
\end{align*}
\]

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

Ad-hoc algorithm

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

Including cost:
- 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

Code generator generators

- Tree description language
  - Represent IR tree as text
- Specification
  - IR tree patterns
  - Code generation actions
- Generator
  - Takes the specification
  - Produces a code generator

Tree notation

- Use prefix notation to avoid confusion

```
store+(a,x+(i,1),4),load+(b, xj, 4))
```

```
store

store

store
```

```
+(a(+(i,1),4))
```

```
+(i,1)
```

```
+(b, xj, 4))
```

```
+(j, 4)
```

```
+(0,11)
```

```
+(b, xj, 4))
```

```
+(j, 4)
```

```
+(b, xj, 4))
```

```
+(j, 4)
```

```
+(b, xj, 4))
```

```
+(j, 4)
```
Rewrite rules

- Rule
  - Pattern to match and replacement
  - Cost
  - Code generation template
  - May include actions – e.g., generate register name

<table>
<thead>
<tr>
<th>Pattern, replacement</th>
<th>Cost</th>
<th>Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>*(reg, reg) → reg 2</td>
<td>1</td>
<td>add r1, r2</td>
</tr>
<tr>
<td>store(reg, load(reg)) → done</td>
<td>5</td>
<td>movem r2, r1</td>
</tr>
</tbody>
</table>

Example

```assembly
muli 4, rj
add rj, rb
addi 1, ri
muli 4, ri
add ri, ra
movem rb, ra
```

Rewriting process

```assembly
store(+(ra, +(ri, 4)), load(+rb, +(rj, 4)));
muli 4, rj
store(+ra, +(ri, 4)), load(rb)
add rj, rb
addi 1, ri
muli 4, ri
store(ra, load(rb))
add ri, ra
done
```

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

Mem and move

- Alternative to load, store
Summary

<table>
<thead>
<tr>
<th>Ad-hoc pattern matchers</th>
<th>Probably reasonable for RISC machines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encode matching as automaton</td>
<td>Fast, optimal code generation – requires separate tool</td>
</tr>
<tr>
<td>Use parsers</td>
<td>Can lead to highly ambiguous grammars</td>
</tr>
</tbody>
</table>

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

Next time…

- Introduction to optimization
- New programming assignment in the works