Abstract Assembly

- Abstract assembly = assembly code w/ infinite register set
- Canonical intermediate code = abstract assembly code – except for expression trees
  - $\text{MOVE}(e_1, e_2) \Rightarrow \text{mov } e_1, e_2$
  - $\text{JUMP}(e) \Rightarrow \text{jmp } e$
  - $\text{CJUMP}(e, l) \Rightarrow \text{cmp } e_1, e_2$
    \[ \begin{cases} 
    \text{jne} & | \text{jel} & \text{jgt} & \ldots \\ 
    l 
    \end{cases} \]
  - $\text{CALL}(e, e_1, \ldots) \Rightarrow \text{push } e_1; \ldots; \text{call } e$
  - $\text{LABEL}(l) \Rightarrow l$

Instruction selection

- Conversion to abstract assembly is problem of \textit{instruction selection} for a single IR statement node
- Full abstract assembly code: glue translated instructions from each of the statements
- Problem: more than one way to translate a given statement. How to choose?
**Example**

\[
\text{MOVE}(\text{TEMP}(t1), \text{TEMP}(t1) + \text{MEM}(\text{TEMP}(\text{FP})+4))
\]

\[
\begin{align*}
\text{MOVE} \\
\text{TEMP}(t1) \\
\text{ADD} \\
\text{MEM} \\
\text{ADD} \\
\text{TEMP}(\text{fp}) \\
4
\end{align*}
\]

\[
\begin{align*}
\text{move} & \ %rbp,t2 \\
\text{add} & \ $4,t2 \\
\text{mov} & \ (t2),t3 \\
\text{add} & \ t3,t1
\end{align*}
\]

\[
\begin{align*}
\text{add} & \ 4(%rbp),t1
\end{align*}
\]

**x86-64 ISA**

- Need to map IR tree to actual machine instructions – need to know how instructions work.
- A two-address CISC architecture (inherited from 4004, 8008, 8086...)
- Typical instruction has
  - opcode (mov, add, sub, shl, shr, mul, div, jmp, jcc, push, pop, test, enter, leave, &c.)
  - destination \((x, n, (x), k(x), (r1, r2), (r1, r2, w), k(r1, r2, w))\)
    (may also be an operand)
  - source (any legal destination, or a constant \(k\))

\[
\begin{align*}
\text{mov} & \ %rbp,t2 \\
\text{add} & \ $4,t2 \\
\text{mov} & \ (t2),t3 \\
\text{add} & \ t3,t1
\end{align*}
\]

**AT&T vs Intel**

- Intel syntax:
  - opcode dest, src
  - Registers rax, rbx, rcx,...r8,r9,...r15
  - constants \(k\)
  - memory operands \([n], [r+k], [r1+w*r2], ...\)
- AT&T syntax (gnu assembler default):
  - opcode src, dest
  - %rax, %rbx,...
  - constants \(k\)
  - memory operands \(n, k(r), (r1,r2,w), ...\)

**Tiling**

- Idea: each Pentium instruction performs computation for a piece of the IR tree: a tile

\[
\begin{align*}
\text{mov} & \ %rbp,t2 \\
\text{add} & \ $4,t2 \\
\text{mov} & \ (t2),t3 \\
\text{add} & \ t3,t1
\end{align*}
\]

- Tiles connected by new temporary registers \((t2, t3)\) that hold result of tile
Some tiles

- **MOVE**
  - TEMP(t1)

- **ADD**
  - t_1 + t_2 → t_f
  - **mov** $t_1, t_2$

- **MEM**
  - **mov** $\text{MEM}_{\text{CONST}(i)}$, (t_1, t_2)

Problem

- How to pick tiles that cover IR statement tree with minimum execution time?
- Need a good selection of tiles
  - small tiles to make sure we can tile every tree
  - large tiles for efficiency
- Usually want to pick large tiles: fewer instructions
- Instructions ≠ cycles: RISC core instructions take 1 cycle, other instructions may take more

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>add %rax, 4(%rcx)</td>
<td>mov 4(%rcx), %rdx</td>
</tr>
<tr>
<td>add %rdx, %rax</td>
<td>mov %rax, 4(%rcx)</td>
</tr>
</tbody>
</table>

An annoying instruction

- Pentium mul instruction multiples single operand by eax, puts result in eax (low 32 bits), edx (high 32 bits)
- Solution: add extra mov instructions, let register allocation deal with edx overwrite

<table>
<thead>
<tr>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov eax, t1</td>
</tr>
<tr>
<td>mul t2</td>
</tr>
<tr>
<td>mov t_f, eax</td>
</tr>
</tbody>
</table>

Branches

- How to tile a conditional jump?
- Fold comparison operator into tile

<table>
<thead>
<tr>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>test t1</td>
</tr>
<tr>
<td>jnz 11</td>
</tr>
<tr>
<td>cmp t1, t2</td>
</tr>
<tr>
<td>je 11</td>
</tr>
</tbody>
</table>
More handy tiles

**lea** instruction computes a memory address but doesn’t actually load from memory

\[
\text{lea } (t_1, t_2), t_e \quad \text{ (} t_e \text{ a fresh temporary)}
\]

\[
\text{lea } (t_1, t_2, k_1), t_e \quad \text{ (} k_1 \text{ one of } 2, 4, 8)
\]

Greedy tiling

- Assume larger tiles = better
- Greedy algorithm: start from top of tree and use largest tile that matches tree
- Tile remaining subtrees recursively

How good is it?

Very rough approximation on modern pipelined architectures: execution time is number of tiles

Greedy tiling (Appel: “maximal munch”) finds an **optimal** but not necessarily **optimum** tiling: cannot combine two tiles into a lower-cost tile

- We can find the optimum tiling using dynamic programming!

Instruction Selection

- Current step: converting canonical intermediate code into abstract assembly
  - implement each IR statement with a sequence of one or more assembly instructions
  - sub-trees of IR statement are broken into **tiles** associated with one or more assembly instructions
Tiles

- Tiles capture compiler’s understanding of instruction set
- Each tile: sequence of instructions that update a fresh temporary (may need extra mov’s) and associated IR tree
- All outgoing edges are temporaries

Another example

Example

alternate (non-RISC) tiling
### ADD expression tiles

- `mov t1, t2`
- `add t1, r/m32`

### ADD statement tiles

Intel Architecture Manual, Vol 2, 3-17:
- `add eax, imm32`
- `add r/m32, imm32`
- `add r/m32, imm8`
- `add r/m32, r32`
- `add r32, r/m32`

### Designing tiles

- Only add tiles that are useful to compiler
- Many instructions will be too hard to use effectively or will offer no advantage
- Need tiles for all single-node trees to guarantee that every tree can be tiled, e.g.

```
mov t1, t2
add t1, t3
```

### More handy tiles

lea instruction computes a memory address but doesn’t actually load from memory

```
lea t_f, [t_1+t_2]  \quad (t_f \text{ a fresh temporary})
```

```
lea t_f, [t_1+k_1*t_2]  \quad (k_1 \text{ one of } 2, 4, 8, 16)
```
Matching CJUMP for RISC

- As defined in lecture, have
  \[
  \text{CJUMP}(\text{cond}, \text{destination})
  \]

- Appel: \(\text{CJUMP}(\text{op}, e_1, e_2, \text{destination})\)
  where \(\text{op}\) is one of \(==, \neq, <, \leq, \geq, >\)

- Our CJUMP translates easily to RISC ISAs that have explicit comparison result
  
  \[
  \begin{align*}
  \text{MIPS} &\quad \text{cmplt } t_2, t_3, t_1 \\
  \text{CJUMP} &\quad \text{br } t_1, n
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{CJUMP NAME}(n) &\quad \text{t1} \text{br } \text{t1, n} \\
  \text{cmplt } t_2, t_3, t_1 &\quad \text{test condition codes}
  \end{align*}
  \]

Condition code ISA

- Appel’s CJUMP corresponds more directly to Pentium conditional jumps
  \[
  \begin{align*}
  \text{CJUMP} &\quad \text{cmp } t_1, t_2 \\
  \text{NAME}(n) &\quad \text{jl n}
  \end{align*}
  \]

- However, can handle Pentium-style jumps with lecture IR with appropriate tiles
  
  \[
  \begin{align*}
  \text{CJUMP} &\quad \text{cmp } t_1, t_2 \\
  \text{NAME}(n) &\quad \text{jl n}
  \end{align*}
  \]

Branches

- How to tile a conditional jump?
- Fold comparison operator into tile

\[
\begin{align*}
\text{CJUMP} &\quad \text{jnz } l1 \\
\text{t1} &\quad \text{test } t_1 \\
\text{EQ} &\quad \text{je } l1 \\
\text{CJUMP} &\quad \text{cmp } t_1, t_2
\end{align*}
\]

Fixed-register instructions

- \(\text{mul } r/m32\)
  - Sets eax to low 32 bits of eax * operand,
  - edx to high 32 bits

- \(\text{jecxz label}\)
  - Jump to label if ecx is zero

- \(\text{add eax, } r/m32\)
  - Add to eax

No fixed registers in IR except TEMP(FP)!
**Strategies for fixed regs**

- Use extra `mov`'s and temporaries
  
  \[
  \begin{align*}
  &\text{mov eax, t2} \\
  &\text{mul t3} \\
  &\text{mov t1, eax}
  \end{align*}
  \]

- Don’t use instruction (`jecxz`)
- Let assembler figure out when to use (`add eax, ...`), bias register allocator

**Implementation**

- Maximal Munch: start from statement node
- Find largest tile covering top node and matching all children
- Invoke recursively on all children of `tile`
- Generate code for this tile (code for children will have been generated already in recursive calls)

**Implementing tiles**

- Explicitly building every tile: tedious
- Easier to write subroutines for matching Pentium source, destination operands
- Reuse matcher for all opcodes

**Matching tiles**

```java
abstract class IR_Stmt {
    Assembly munch();
}

class IR_Move extends IR_Stmt {
    IR_Expr src, dst;
    Assembly munch() {
        if (src instanceof IR_Plus && ((IR_Plus)src).lhs.equals(dst) && is_regmem32(dst)) {
            Assembly e = (IR_Plus)src).rhs.munch();
            return e.append(new AddrInsn(dst, e.target()));
        }
        else if ...}
    }
```
Tile Specifications

- Previous approach simple, efficient, but hard-codes tiles and their priorities
- Another option: explicitly create data structures representing each tile in instruction set
  - Tiling performed by a generic tree-matching and code generation procedure
  - Can generate from instruction set description
  - generic back end!
- For RISC instruction sets, over-engineering

Improving instruction selection

- Greedy tiling may not generate best code
  - Always selects largest tile, not necessarily fastest instruction
  - May pull nodes up into tiles when better to leave below
- Can do better using dynamic programming algorithm

Timing model

- Idea: associate cost with each tile (proportional to # cycles to execute)
  - caveat: cost is fictional on modern architectures
- Estimate of total execution time is sum of costs of all tiles

Finding optimum tiling

- **Goal:** find minimum total cost tiling of tree
- **Algorithm:** for every node, find minimum total cost tiling of that node and sub-tree.
- **Lemma:** once minimum cost tiling of all children of a node is known, can find minimum cost tiling of the node by trying out all possible tiles matching the node
- **Therefore:** start from leaves, work upward to top node
Dynamic programming: $a[i]$

```
mov t1, [bp + a]
mov t2, [bp + i]
mov t3, [t1 + 4*t2]
```

Recursive implementation

- Any dynamic programming algorithm equivalent to a memoized version of same algorithm that runs top-down
- For each node, record best tile for node
- Start at top, recurse:
  - First, check in table for best tile for this node
  - If not computed, try each matching tile to see which one has lowest cost
  - Store lowest-cost tile in table and return
- Finally, use entries in table to emit code

Greedy → Memoization

```java
class IR_Move extends IR_Stmt {
    IR_Expr src, dst;
    Assembly best; // initialized to null
    int optTileCost() {
        if (best != null) return best.cost();
        if (src instanceof IR_Plus &&
            ((IR_Plus)src).lhs.equals(dst) && is_regmem32(dst)) {
            int src_cost = ((IR_Plus)src).rhs.optTileCost();
            int cost = src_cost + CISC_ADD_COST;
            if (cost < best.cost())
                best = new AddIns(dst, e.target);
        }...
        return best.cost();
    }
}
```

A small tweak to greedy algorithm!

Problems with model

- Modern processors:
  - execution time not sum of tile times
  - instruction order matters
    - Processors is pipelining instructions and executing different pieces of instructions in parallel
    - bad ordering (e.g. too many memory operations in sequence) stalls processor pipeline
    - processor can execute some instructions in parallel (super-scalar)
  - cost is merely an approximation
  - instruction scheduling needed
Summary

- Can specify code generation process as a set of tiles that relate IR trees to instruction sequences
- Instructions using fixed registers problematic but can be handled using extra temporaries
- Greedy algorithm implemented simply as recursive traversal
- Dynamic programming algorithm generates better code, also can be implemented recursively using memoization
- Real optimization will require instruction scheduling