Where we are

Intermediate code
  syntax-directed translation
  reordering with traces

Canonical intermediate code
  tiling
  dynamic programming

Abstract assembly code
  register allocation

Assembly code

Abstract Assembly

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

Instruction selection

- Conversion to abstract assembly is problem of 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

MOVE(TEMP(t1), TEMP(t1) + MEM(TEMP(FP)+4))

Pentium ISA

- Need to map IR tree to actual machine instructions – need to know how instructions work
- Pentium is two-address CISC architecture
- Typical instruction has
  - opcode (mov, add, sub, shl, shr, mul, div, jmp, jcc, push, pop, test, enter, leave, &c.)
  - destination ([r], [r], [k], [r+k], [r1+r2],
    [r1+w*r2], [r1+w*r2+k])
    (may also be an operand)
  - source (any legal destination, or a constant)

Tiling

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

Some tiles

- Tiles connected by new temporary registers (t2, t3) that hold result of tile
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
- Pentium: RISC core instructions take 1 cycle, other instructions may take more

\[
\text{add } [\text{ecx+4}], \text{ eax} \leftrightarrow \text{mov edx, } [\text{ecx+4}]
\]
\[
\text{add edx, eax}
\]
\[
\text{mov } [\text{ecx+4}], \text{ eax}
\]

An annoying instruction

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

\[
\text{mov eax, t1}
\]
\[
\text{mul t2}
\]
\[
\text{mov } t_\ell, \text{ eax}
\]

Branches

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

\[
\text{test } t_1
\]
\[
\text{jnz } l1
\]
\[
\text{cmp } t_1, t_2
\]
\[
\text{je } l1
\]

More handy tiles

\text{lea} instruction computes a memory address but doesn’t actually load from memory

\[
\text{lea } t_\ell, [t_1+t_2]
\] \hspace{1cm} \text{\textbf{(t_\ell \text{ a fresh temporary})}}

\[
\text{lea } t_\ell, [t_1+k_1*t_2]
\] \hspace{1cm} \text{\textbf{(k_1 \text{ one of } 2,4,8,16)}}
Greedy tiling

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

```
  MOVE
  MEM
  ADD
  MEM
  ADD
  MUL
  MEM
  ADD
  FP
  8
  FP
  12
```

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

```
  mov t2, t1
  add t2, imm8
```

- 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

\[ x = x + 1; \]

Example

\[ x = x + 1; \]

Alternate (non-RISC) tiling

\[ x = x + 1; \]

ADD expression tiles

\[ \text{mov } t1, t2 \]
\[ \text{add } t1, r/m32 \]
\[ \text{mov } t1, t2 \]
\[ \text{add } t1, \text{imm32} \]
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

```
CJUMP(cond, destination)
```

- Appel: `CJUMP(op, e1, e2, destination)`
where `op` is one of `==, !=, <, <=, =>, >`

- Our `CJUMP` translates easily to RISC ISAs that have explicit comparison result

```
cmplt t2, t3, t1
```

```
br t1, n
```
**Condition code ISA**

- Appel’s CJUMP corresponds more directly to Pentium conditional jumps
  
  ```
  CJUMP
  NAME(n)
  <
  t1  t2
  cmp t1, t2
  jl n.
  ```

- However, can handle Pentium-style jumps with lecture IR with appropriate tiles

**Branches**

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

  ```
  CJUMP
  l1 (l2)
  cmp t1, t2
  je l1
  ```

**Fixed-register instructions**

- `mul r/m32`
  - Sets eax to low 32 bits of eax * operand, edx to high 32 bits

- `jecxz label`
  - Jump to `label` if ecx is zero

- `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

  ```
  mov eax, t2
  mul t3
  mov t1, eax
  ```

- 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)

• How to find matching tiles?

Implementing tiles

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

Matching tiles

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 AddIns(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: $a[i]$
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

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);
    }
    ...consider all other tiles...
    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