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

• $\text{MOVE}(e_1, e_2) \Rightarrow \text{mov} \ e_1, \ e_2$

• $\text{JUMP}(e) \Rightarrow \text{jmp} \ e$

• $\text{CJUMP}(e, l) \Rightarrow \text{cmp} \ e_1, \ e_2$

• $\text{[jne|je|jgt|…]} \ l$

• $\text{CALL}(e, e_1, \ldots) \Rightarrow \text{push} \ e_1; \ldots; \text{call} \ e$

• $\text{LABEL}(l) \Rightarrow \text{l:}$
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))

MOVE
  
  TEMP(t1)  ADD
    
    TEMP(t1)  MEM
                  
                  ADD
                    
                    TEMP(fp)  4

mov %rbp, t2
add $4, t2
mov (t2), t3
add t3, t1
add 4(%rbp), t1
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 (r, n, (r), k(r), (r1,r2), (r1,r2,w), k(r1,r2,w))
    (may also be an operand)
  – source (any legal destination, or a constant $k$)

  \[
  \text{mov} \ $1,%rax \quad \text{add} \ %rcx,%rbz \\
  \text{sub} \ %rbp,%esi \quad \text{add} \ %edi, (%rcx,%edi,16) \\
  \text{je label1} \quad \text{jmp} \ 4(%rbp)
  \]
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*

```assembly
mov %rbp, t2
add $4, t2
mov (t2), t3
add t3, t1
```

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

\begin{align*}
\text{MOVE} & \quad \text{TEMP}(t_1) \\

\text{ADD} & \quad t_f \quad t_1 \quad t_2 \\

\text{MOVE} & \quad \text{MEM} \quad \text{CONST}(i) \\

\end{align*}

\begin{align*}
\text{mov} & \quad t_2, t_1 \\
\text{mov} & \quad t_1, t_f \quad (t_f \text{ a fresh temporary}) \\
\text{add} & \quad t_2, t_f \\
\text{mov} & \quad $i, (t_1, t_2)
\end{align*}
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

\[
\begin{align*}
\text{add} & \%rax,4(\%rcx) \iff \text{mov} 4(\%rcx) , \%rdx \\
& \text{add} \ %rdx , \%rax \\
& \text{mov} \ %rax , 4(\%rcx)
\end{align*}
\]
An annoying instruction

- Pentium mul instruction multiples single operand by $\text{rax}$, puts result in $\text{rax}$ (low 32 bits), $\text{rdx}$ (high 32 bits).

- Solution: add extra $\text{mov}$ instructions, let register allocation deal with $\text{rdx}$ overwrite.

```
mov rax, t1
mul t2
mov t_f, rax
```
Branches

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

```markdown
CJUMP
  \( t_1 \)
  \( l1 \) \( (l2) \)
```

```markdown
test t1
jnz l1
```

```markdown
CJUMP
  \( t_1 \)
  \( t_2 \)
  EQ
  \( l1 \) \( (l2) \)
```

```markdown
cmp t1, t2
je l1
```
More handy tiles

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

\[
\text{ADD} \quad \text{lea} \ (t_1, t_2), t_f
\]

\[
\text{ADD} \quad \text{lea} \ (t_1, t_2, k_1), t_f \quad (t_f \text{ a fresh temporary})
\]

\[
\text{ADD} \quad \text{CONST}(k_1) \quad (k_1 \text{ one of } 2, 4, 8)
\]
Greedy tiling

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

```
MOVE
   MEM 4
   ADD
      MEM 4
      ADD 8
      FP 12
MUL
   MEM
   ADD
      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

- 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

```plaintext
mov t2, t1
add t2, imm8
```
Another example

\[ x = x + 1; \]

\[
\text{MOVE} \\
\text{MEM} \\
\text{FP} + x \\
\text{MEM} \\
\text{FP} + x \\
\text{MEM} + 1
\]
Example

\[ x = x + 1; \]

\[ \text{mov } t1, [ebp+x] \]
\[ \text{mov } t2, t1 \]
\[ \text{add } t2, 1 \]
\[ \text{mov } [ebp+x], t2 \]

**ebp**: Pentium frame pointer register
Alternate (non-RISC) tiling

\[ x = x + 1; \]

```
MOVE MEM [ebp+x], 1
MOVE FP x
MEM FP +
```

```
MOVE r/m32 +
MOVE r/m32 +
CONST(k)
```
ADD expression tiles

\[
\begin{align*}
\text{mov} & \ t1, \ t2 \\
\text{add} & \ t1, \ r/m32 \\
\text{mov} & \ t1, \ t2 \\
\text{add} & \ t1, \ \text{imm32}
\end{align*}
\]
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.

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

![Diagram of a single-node tree with nodes t1, t2, and t3 connected by an addition operation.](image)
More handy tiles

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

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

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

• As defined in lecture, have

\[
\text{CJUMP}\left(\text{cond}, \text{destination}\right)
\]

• Appel: \(\text{CJUMP}(\text{op, } e_1, e_2, \text{destination})\)
  where \(\text{op}\) is one of \(==, !=, <, <=, =>, >\)

• Our \text{CJUMP} translates easily to RISC ISAs that have explicit comparison result

\[
\begin{align*}
\text{MIPS} & \quad \text{CJUMP}\left(\text{cond}, \text{destination}\right) \\
& \quad \text{NAME}(n) \\
& \quad \text{t}1 \quad \text{br} \quad \text{t}1, \text{n} \\
& \quad \text{t}2 \quad \text{cmplt} \quad \text{t}2, \text{t}3, \text{t}1
\end{align*}
\]
Condition code ISA

- Appel’s CJUMP corresponds more directly to Pentium conditional jumps
  
  \[
  \text{CJUMP} < \text{NAME}(n) \]
  
  \[
  \text{cmp } t1, t2 \quad \text{jl } n \]

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

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

```
comp t_1, t_2
je l1
```
Fixed-register instructions

mul \textit{r/m32}

Sets eax to low 32 bits of eax * operand, edx to high 32 bits

jecxz \textit{label}

Jump to \textit{label} if ecx is zero

add eax, \textit{r/m32}

Add to eax

No fixed registers in IR except \texttt{TEMP(FP)}!
Strategies for fixed regs

- Use extra MOV’s and temporaries

\[
\begin{align*}
\text{mov} & \text{ eax, t2} \\
\text{mul} & \text{ t3} \\
\text{mov} & \text{ t1, eax}
\end{align*}
\]

- 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)

• 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

MOVE

dest source

ADD

source

source =

CONST(i)

MEM

ADD

CONST(i)

MEM

TEMP(t)
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

Total cost: 5
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
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