Abstract Assembly

- Abstract assembly = assembly code w/ infinite register set
- Canonical intermediate code = abstract assembly code + expression trees

\[
\begin{align*}
\text{MOVE}(e_1, e_2) & \Rightarrow \text{mov } e_1, e_2 \\
\text{JUMP}(e) & \Rightarrow \text{jmp } e \\
\text{CJUMP}(e, l) & \Rightarrow \text{cmp } e_1, e_2 \\
& \quad [\text{jne|je|jgt}|...] l \\
\text{CALL}(e, e_1, \ldots) & \Rightarrow \text{push } e_1; \ldots; \text{call } e \\
\text{LABEL}(l) & \Rightarrow l:
\end{align*}
\]

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)
ADD
TEMP(t1)
MEM
?
ADD
TEMP(ip)
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)
  - 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 $k$

```
opcode src/srg/dest
mov $1, rax
add $rcx, rdx
sub $rspb, rsi
add $edi, ($rcx, $edi, 16)
je labell
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, ..., %r8, %r9, ..., %r15
  - 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
  - mov %rbp, t2
  - add $4, t2
  - mov (t2), t3
  - add t3, t1

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

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

Some tiles

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

More handy tiles

- lea instruction computes a memory address but doesn’t actually load from memory
  - lea (t1, t2), t3
  - lea (t1, t2, k1), t3 (k1 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 \(==, !=, <, <=, =>, >\)
- Our CJUMP translates easily to RISC ISAs
  that have explicit comparison result

#### MIPS

- \text{cmp} t2, t3, t1
- \text{br } t1, n

#### RISC

- \text{CJUMP NAME(n) t1 br t1, n}

### Condition code ISA

- Appel’s CJUMP corresponds more directly
to Pentium conditional jumps

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

### Branches

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

#### MIPS

- \text{cmp t1, t2}
- \text{jl } n

#### RISC

- \text{CJUMP EQ t1, t2}
- \text{je } l1

### 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

#### RISC

- \text{mov t1, %eax}
- \text{mul t2}
- \text{mov %eax, t_f}

### Tiling 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

#### Example

- \text{add %rax,4(%rcx)} \Leftrightarrow \text{mov 4(%rcx),%rdx}
  \text{add %rdx,%rax}
  \text{mov %rax,4(%rcx)}

- \text{mov (%ebp,x), t1}
- \text{mov t1, t2}
- \text{add $1, t2}
- \text{mov t2, (%ebp,x)}
Alternate (non-RISC) tiling

\[ x = x + 1; \]

\[ \text{add } \$1, (ebp, x) \]

Greedy tiling

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

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

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
Problems with model

- Modern processors:
  - execution time not sum of tile times
  - instruction order matters
    - Processors are 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

Finding matching 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

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, can also be implemented recursively using memoization
- Real optimization will require instruction scheduling