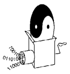


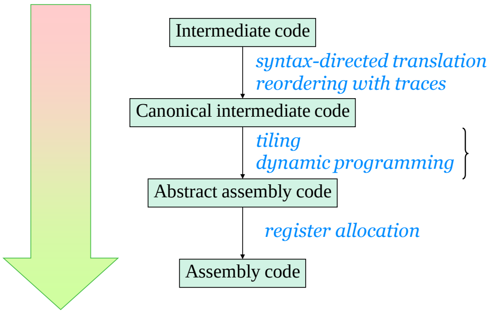
CS 4120
Introduction to Compilers

Ross Tate

Cornell University

Lecture 18: Instruction Selection

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 + expression trees

MOVE(e_1, e_2) ⇒ **`mov e1, e2`**

JUMP(e) ⇒ **`jmp e`**

CJUMP(e,l) ⇒ **`cmp e1, e2`**
`[jne|je|jgt|…] l`

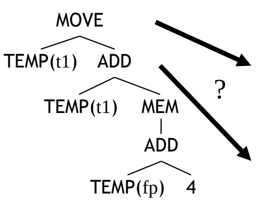
CALL(e, e_1,…) ⇒ **`push e1`**; … ; **`call e`**

LABEL(l) ⇒ **`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



```
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`**, **`j`***cc*, **`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* *src/dest*

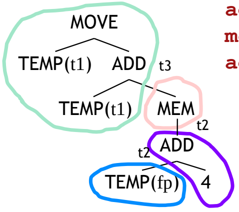
```
mov $1,%rax          add %rcx,%rbz
sub %rbp,%esi        add %edi, (%rcx,%edi,16)
je label1            jmp 4(%rbp)
```

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

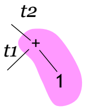
- Idea: each Pentium instruction performs computation for a piece of the IR tree: a *tile*



```
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

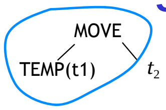
Tiles



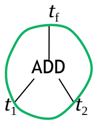
```
mov t1, t2
add $1, t2
```

- 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

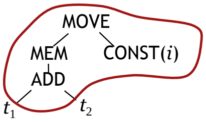
Some tiles



mov t_2,**t1**



mov t_1,t_f
add t_2,t_f

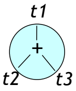


**mov $i,**($t_1$,$t_2$)

Designing tiles

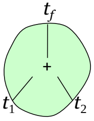
- Only add tiles that are useful to compiler
- Many instructions will be too hard to use effectively or will offer no advantage
- Need tiles for all single-node trees to guarantee that every tree can be tiled, e.g.

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

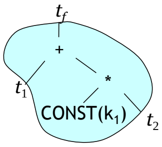


More handy tiles

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



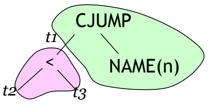
lea (t_1,t_2), t_f



lea (t_1,t_2,k_1), t_f (k_1 **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

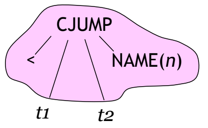


MIPS

```
cmplt t2, t3, t1
br    t1, n
```

Condition code ISA

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



set condition codes

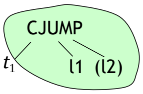
cmp *t1*, *t2*
jl *n*

test condition codes

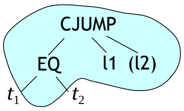
- However, can handle Pentium-style jumps with lecture IR with appropriate tiles

Branches

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



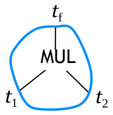
```
test t1
jnz l1
```



```
cmp t1, t2
je l1
```

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



```
mov t1, %eax
mul t2
mov %eax, tf
```

Tiling Problem

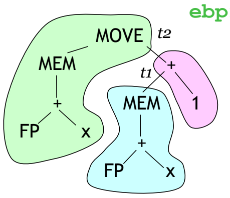
- How to pick tiles that cover IR statement tree with minimum execution time?
- Need a good selection of tiles
  - small tiles to make sure we can tile every tree
  - large tiles for efficiency
- Usually want to pick large tiles: fewer instructions
- instructions ≠ cycles: RISC core instructions take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

```
add %rax,4(%rcx)    ⇔    mov 4(%rcx),%rdx
                         add %rdx,%rax
                         mov %rax,4(%rcx)
```

Example

x = x + 1;

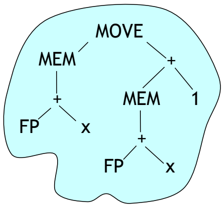
`ebp`: Pentium frame-pointer register



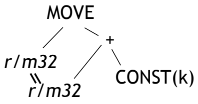
```
mov (%ebp,x), t1
mov t1, t2
add $1, t2
mov t2, (%ebp,x)
```

Alternate (non-RISC) tiling

x = x + 1;

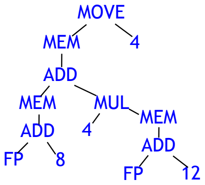


`add $1, (ebp,x)`



Greedy tiling

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



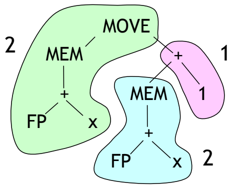
Improving instruction selection

- 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

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

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

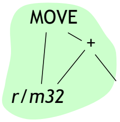
Problems with model

- 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