**Administration**

- Prelim 1: Tuesday, 7:30-9:30PM
  - in Phillips 203 (here)
  - topics covered: regular expressions, tokenizing, context-free grammars, LL & LR parsers, static semantics, intermediate code generation
- Prelim 1 review session Monday in class

---

**Where we are**

- abstract syntax tree
- syntax-directed translation (IR generation)
- intermediate code
- syntax-directed translation (flattening)
- reordering with traces
- canonical intermediate code
- instruction selection
- abstract assembly code
- register allocation
- assembly code

---

**Conditional jumps**

- IR is now just a linear list of statements with one side effect per statement
- Still contains CJUMP nodes: two-way branches
- Real machines: fall-through branches (e.g., JZ, JNZ)

\[
\text{CJUMP}(e, t, f) \\
\text{...} \\
\text{LABEL}(t) \quad \text{if-true code} \\
\text{f:} \\
\text{evaluate e} \\
\text{JZ f} \\
\text{JMP f}
\]

---

**Simple Solution**

- Translate CJUMP into conditional branch followed by unconditional branch

\[
\text{CJUMP}(\text{TEMP}(t1)==\text{TEMP}(t2), t, f) \\
\text{CMP t1, t2} \\
\text{JZ t} \\
\text{JMP f}
\]

- JMP is usually gratuitous
- Code can be reordered so jump goes to next statement

---

**Basic blocks**

- Unit of reordering is a basic block
- A sequence of statements that is always begun at its start and always exits at the end:
  - starts with a LABEL statement (or beginning of all statements)
  - ends with a JUMP or CJUMP statement, or just before a LABEL statement
  - contains no other JUMP or CJUMP statement
  - contains no interior LABEL used as a jump target
- No point to breaking up a basic block during reordering
Basic block example

CJUMP(e, L2, L3)
LABEL(L1)
MOVE(TEMP(x), TEMP(y)
LABEL(L2)
MOVE(TEMP(x), TEMP(y) + TEMP(z))
JUMP(NAME(L1))
LABEL(L3)
EXP(CALL(NAME(f)), TEMP(x))

Control flow graph

- Control flow graph has basic blocks as nodes
- Edges show control flow between basic blocks

Fixing conditional jumps

- Reorder basic blocks so that (if possible)
  - the “false” direction of two-way jumps goes to
    the very next block
  - JUMPs go to the next block (are deleted)
- What if not satisfied?
  - For CJUMP add another JUMP immediately
    after to go to the right basic block
- How to find such an ordering of the basic blocks?

Traces

- Idea: order blocks according to a possible trace: a sequence of blocks that might (naively) be executed in sequence, never visiting a block more than once
- Algorithm:
  - pick an unmarked block (begin w/ start block)
  - run a trace until no more unmarked blocks can be
    visited, marking each block on arrival
  - repeat until no more unmarked blocks

Example

- Possible traces?

Arranging by traces

- Can use profiling information, heuristics to choose which branch to follow
Reordered code

CJUMP(e, L2, L3)
LABEL(L1)
MOVE(TEMP(x), TEMP(y))
LABEL(L2)
MOVE(TEMP(x), TEMP(y) + TEMP(z))
JUMP(L1)
LABEL(L3)
EXP(CALL(NAME(f)), TEMP(x))

Reversing sense of jumps

CJUMP(e, L2, L3)
LABEL(L1)
MOVE(TEMP(x), TEMP(y))
JUMP(L2)
LABEL(L3)
EXP(CALL(NAME(f)), TEMP(x))

Progress

abstract syntax tree
syntax-directed translation (IR generation)
intermediate code
syntax-directed translation (flattening)
reordering with traces
canonical intermediate code
instruction selection (tiling)
abstract assembly code

Abstract Assembly

- Abstract assembly = assembly code w/ infinite register set
- Canonical intermediate code = abstract assembly code – except for expression trees
- MOVE(e1, e2) ⇒ mov e1, e2
- JUMP(e) ⇒ jmp e
- CJUMP(e, l) ⇒ cmp e1, e2
  [jne|je|jgt|…] l
- CALL(e, e1, …) ⇒ 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

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

MOV

TMP(t1) ADD

MEM

ADD

4

TEMP(FP)

ADD

t1, t3

mov t2, fp
add t2, 4
mov t3, [t2]
add t1, t3
add t1, [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, jec, &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)

```
mov eax, 1
add ebx, ecx
sub esi, [ebp]
add [ecx+16*edi], edi
```

Tiling

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

```
mov t2, ebp
add t2, 4
mov t3, [t2]
```

```
ADD
```

Some tiles

```
MOVE
```

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

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

```
add [ecx+4], eax
mov edx, [ecx+4]
add edx, eax
```

Branches

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

```
test t1
jns 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
More handy tiles

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

- `lea t_x, [t_1*t_2]` (t_x a fresh temporary)
- `lea t_x, [t_1+k_1*t_2]` (k_1 one of 2,4,8,16)

Maximal Munch Algorithm

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

Implementing tiles

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

How good is it?

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