# CS 4120
# Introduction to Compilers

Andrew Myers
Cornell University

## Lecture 17: Instruction Selection

7 Mar 2018

# 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

- MOVE($e_1$, $e_2$) $\Rightarrow$ `mov` $e_1$, $e_2$
- JUMP($e$) $\Rightarrow$ `jmp` $e$
- CJUMP($e$,$l$) $\Rightarrow$ `cmp` $e_1$, $e_2$

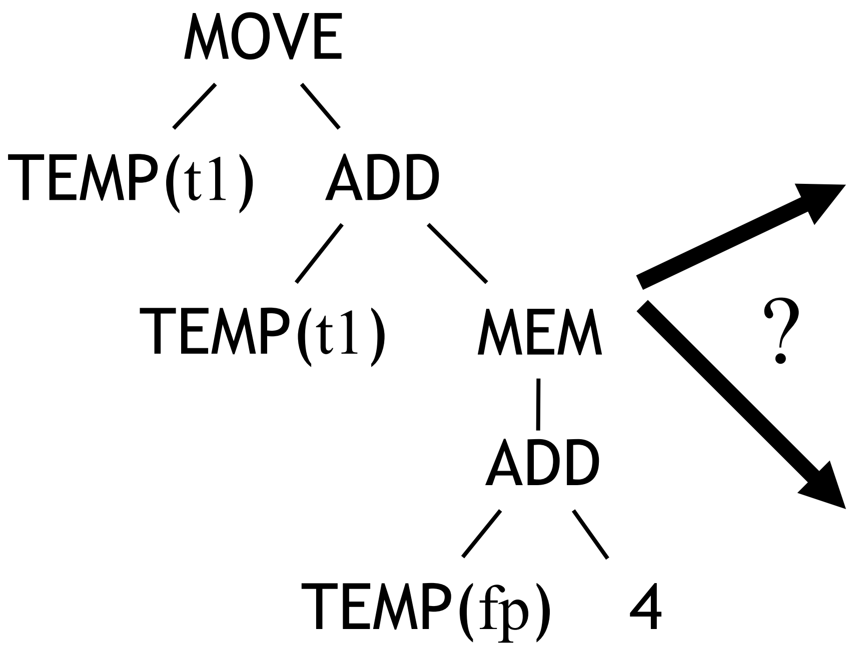
  `[jne|je|jgt|…]` $l$
- CALL($e$, $e_1$,…) $\Rightarrow$ `push` $e_1$; … ; `push` $e_n$; `call` $e$
- LABEL($l$) $\Rightarrow$ `l:`

# 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))




```
mov r2, rbp
add t2, 4
mov r3, [t2]
add t1, t3
```

```
add t1, [rbp + 4]
```

# 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`**, &c.)

– *destination* (may also be an operand): **`r, [n],[r], [r+k], [r1+r2], [r1 + r2*w],[r1 + r2*w + k]`**

- (AT&T notation: **`r,n,(r),k(r),(r1,r2),(r1,r2,w), k(r1,r2,w)`** )

– *source* (any legal destination, or a constant **`$k`**)

*opcode* *src/dest* *src*

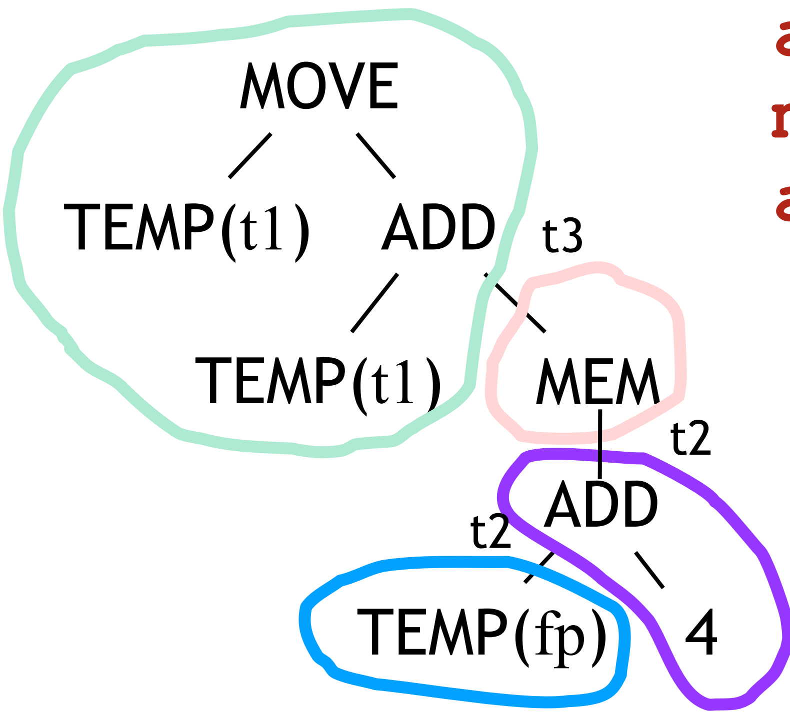
```
mov rax, 1          add rcx,rbz
sub rbp, esi        add edi, [rcx + edi * 8]
je label1           jmp [rbp + 4]
```

# AT&T vs Intel

- Intel syntax: (`gcc -masm=intel`)
  - opcode dest, src
  - Registers rax, rbx, rcx,…r8, r9, …, r15
  - constants k
  - memory operands [n], [r+k], [r1+w*r2], ...
- AT&T syntax (gcc default):
  - opcode src, dest; opcode includes width (addq)
  - registers %rax, %rbx,…
  - constants $k
  - memory operands n, k(r), (r1,r2,w), ...

# Tiling

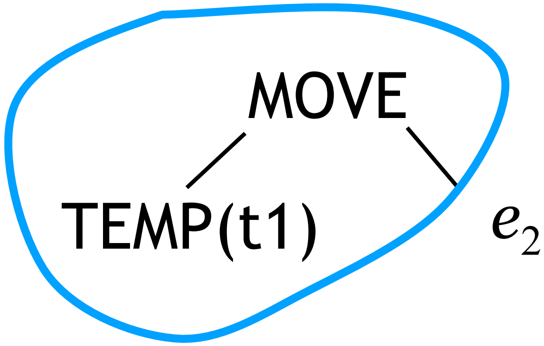
- Idea: each Pentium instruction performs computation for a piece of the IR tree: a *tile*



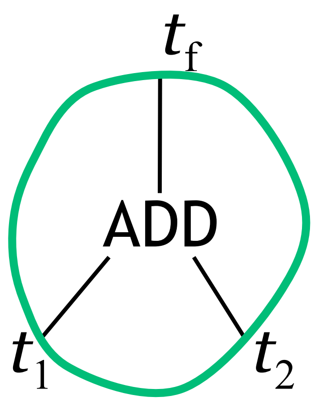

```
mov t2, rbp
add t2, 4
mov t3, [t2]
add t1, t3
```

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

# Some tiles

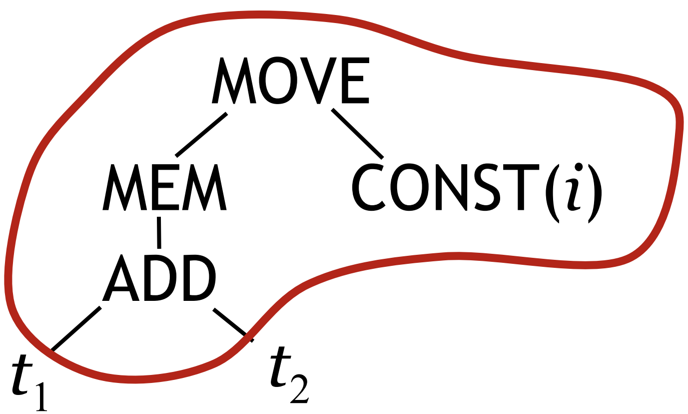



`mov t1, t`$_2$




`mov t`$_f$`,t`$_1$
`add t`$_f$`,t`$_2$

(`t`$_f$ a *fresh* temporary)




`mov [t`$_1$`+t`$_2$`], i`

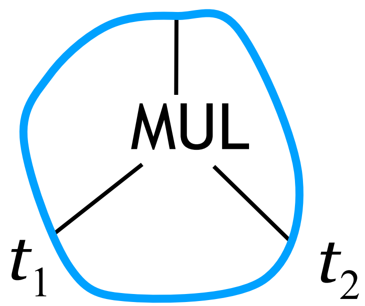
# Problem

- 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 [rcx + 4], rax        ⇔        mov rdx, [rcx+4]
                                   add rax, rdx
                                   mov [rcx+4], rax
```

# An annoying instruction

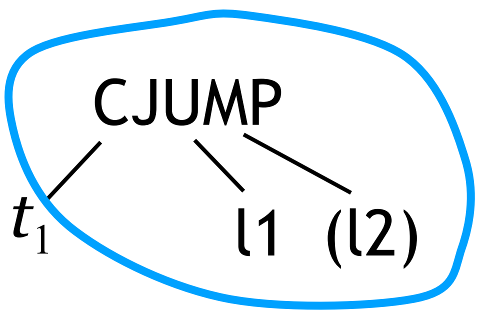
- Pentium mul instruction multiples single operand by **`rax`**, puts result in **`rax`** (low 32 bits), **`rdx`** (high 32 bits)
- Solution: add extra **`mov`** instructions, let register allocation deal with **`rdx`** overwrite



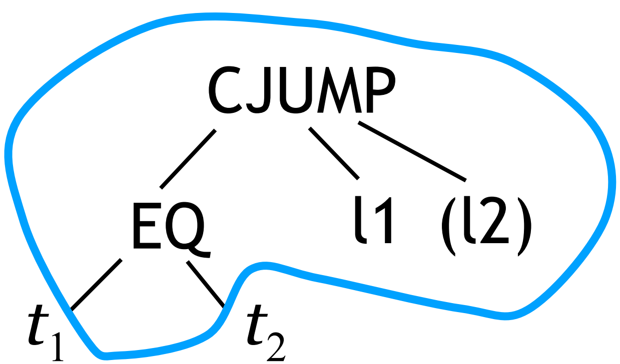

**`mov rax, t1`**
**`mul t2`**
**`mov`** $\mathbf{t_f}$**`, rax`**

# Branches

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



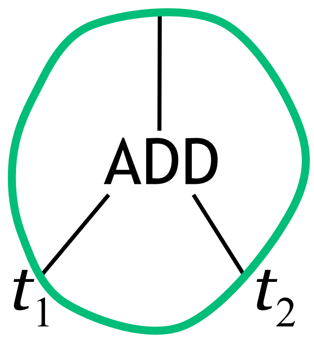

```
test t1
jnz l1
```




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

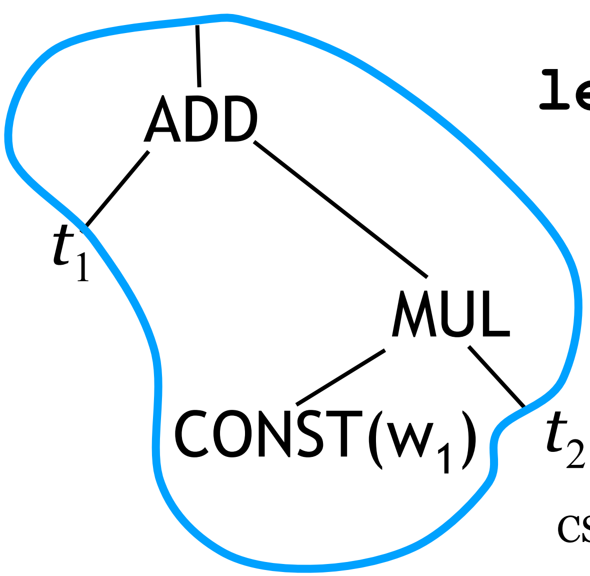
# More handy tiles

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




**lea** $\mathbf{t_f}$, [$\mathbf{t_1}$+$\mathbf{t_2}$]

(**$\mathbf{t_f}$** a *fresh* temporary)

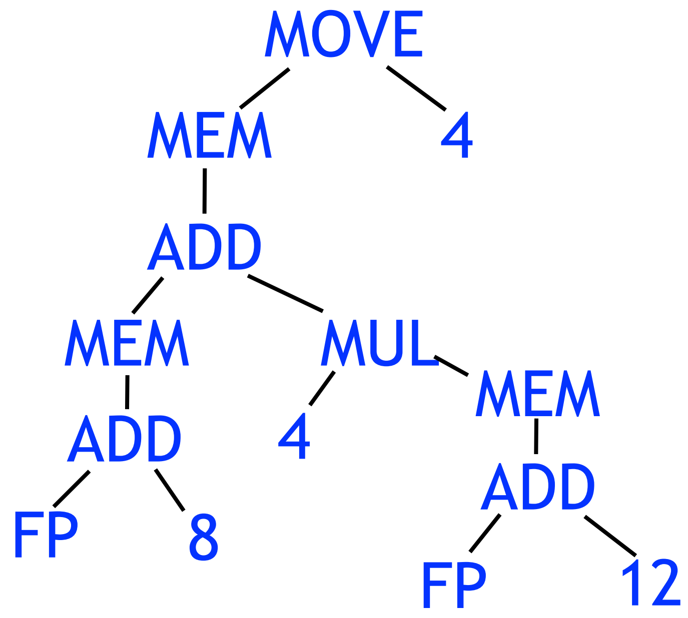



**lea** $\mathbf{t_f}$, [$\mathbf{t_1}$+$\mathbf{t_2}$*$\mathbf{w_1}$]

(**$\mathbf{w_1}$** 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



# How good is it?

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

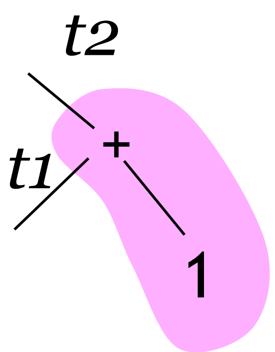
Greedy tiling (Appel: "maximal munch") finds an *optimal* but not necessarily *optimum* tiling: cannot combine two tiles into a lower-cost tile

- 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

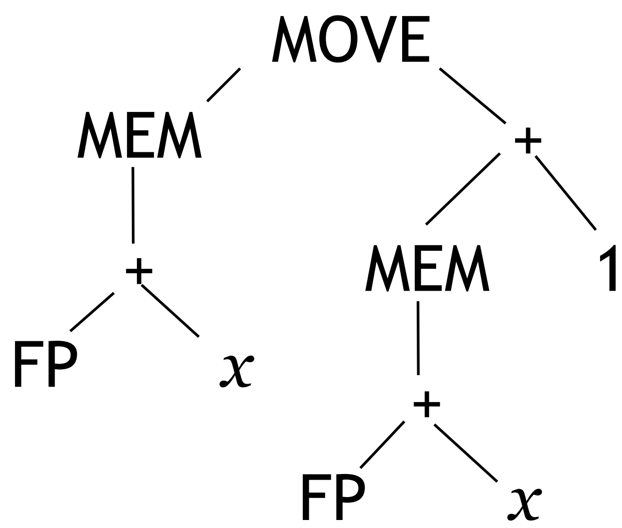



mov t2, t1
add t2, *imm8*

- 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

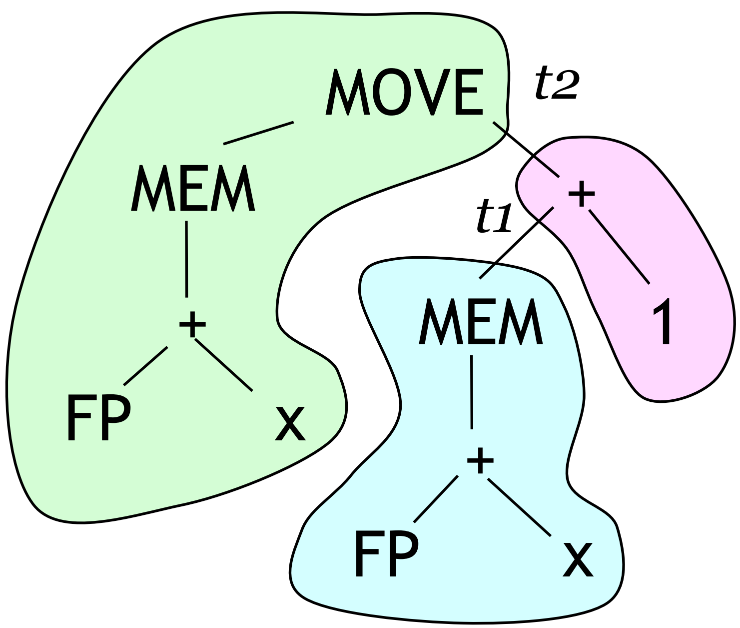
x = x + 1;



# Example

x = x + 1;

**`rbp`**: frame pointer register



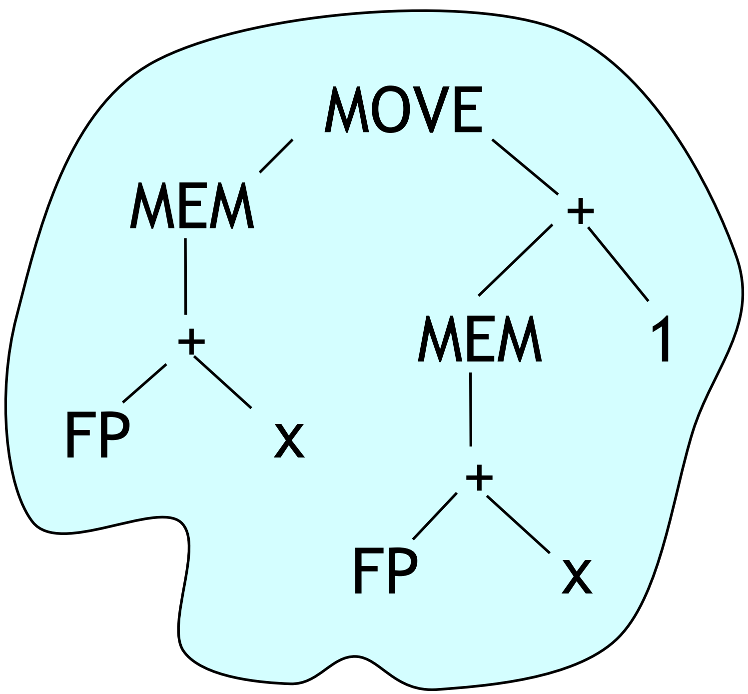

```
mov t1, [rbp+x]
```

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

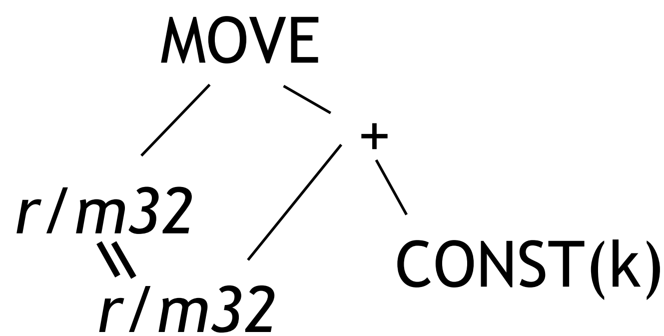
```
mov [rbp+x], t2
```

# Alternate (non-RISC) tiling

x = x + 1;

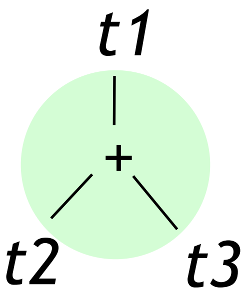


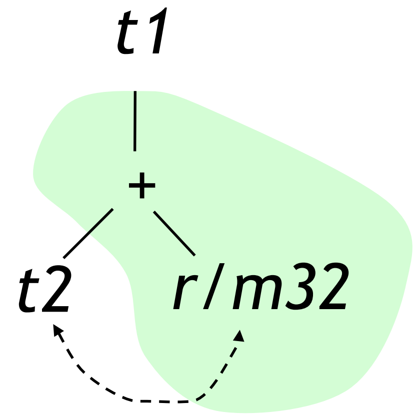

```
add [rbp+x], 1
```



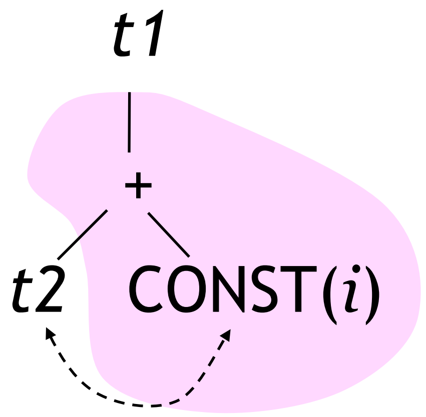
# ADD expression tiles

```
mov t1, t2
add t1, r/m32
```




```
mov t1, t2
add t1, imm32
```



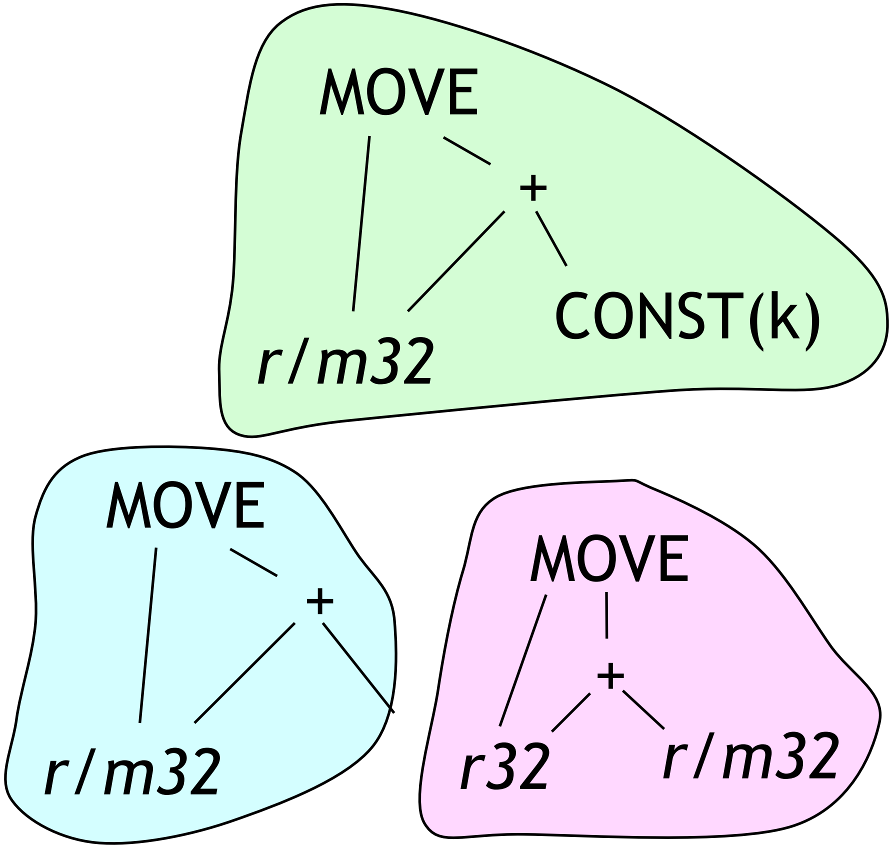
# ADD statement tiles

Intel Architecture Manual, Vol 2, 3-17:

```
add rax, imm32
add r/m32, imm32
add r/m32, imm8
```

```
add r/m32, r32
```

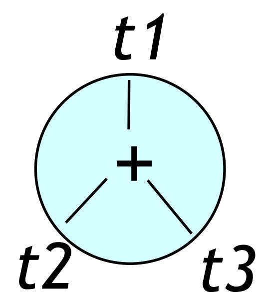
```
add r32, r/m32
```



# Designing tiles

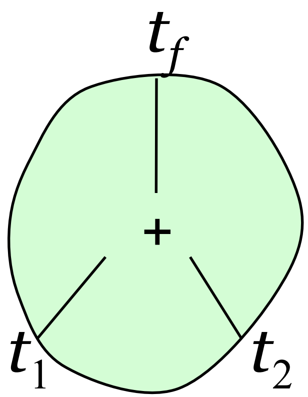
- Only add tiles that are useful to compiler
- Many instructions will be too hard to use effectively or will offer no advantage
- Need tiles for all single-node trees to guarantee that every tree can be tiled, e.g.

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

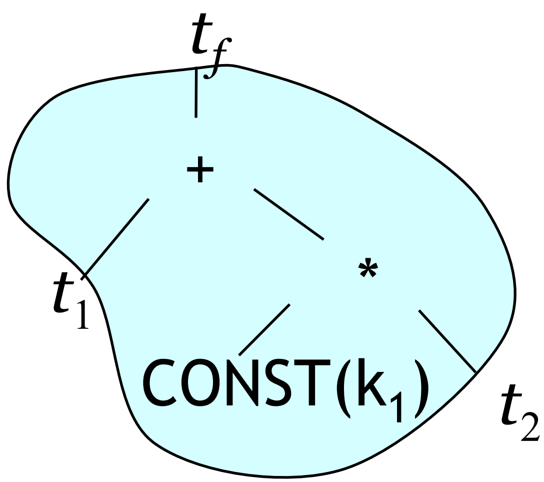


# More handy tiles

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




**`lea tf, [t1+t2]`** ($t_f$ a *fresh* temporary)




**`lea tf, [t1+k1*t2]`** ($k_1$ **one of 2,4,8,16**)

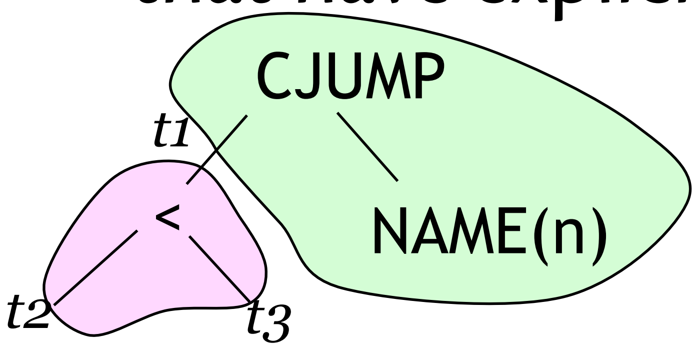
# Matching CJUMP for RISC

- 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

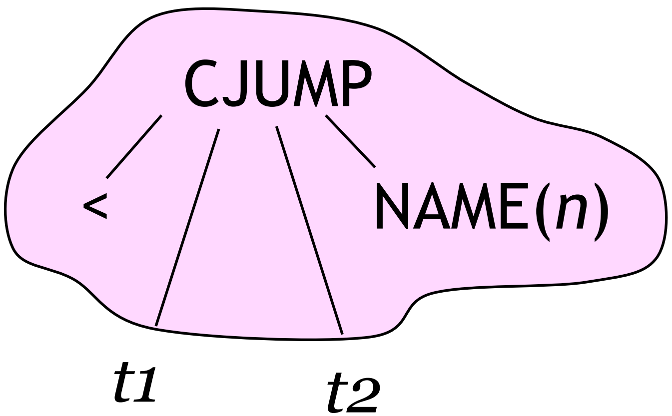



MIPS

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

# Condition code ISA

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




*set condition codes*

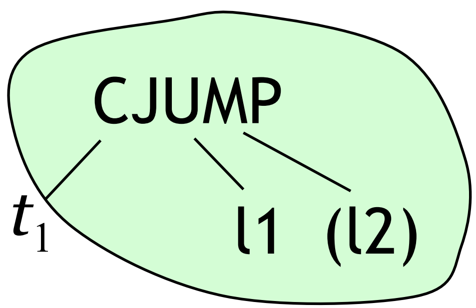
```
cmp t1, t2
jl n
```

*test condition codes*

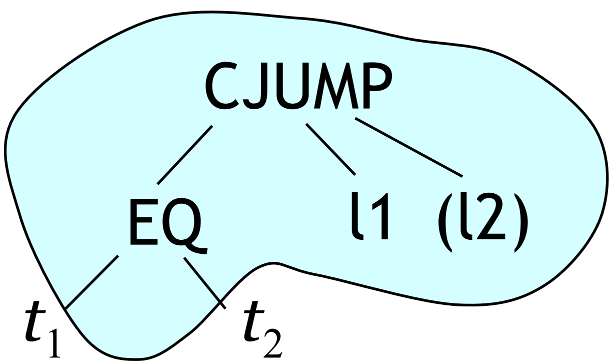
- However, can handle Pentium-style jumps with lecture IR with appropriate tiles

# Branches

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




```
test t1
jnz l1
```




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

# 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*

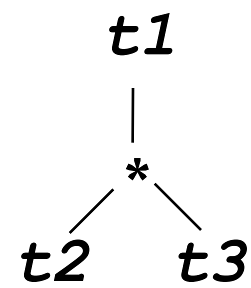
Add to eax

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
```



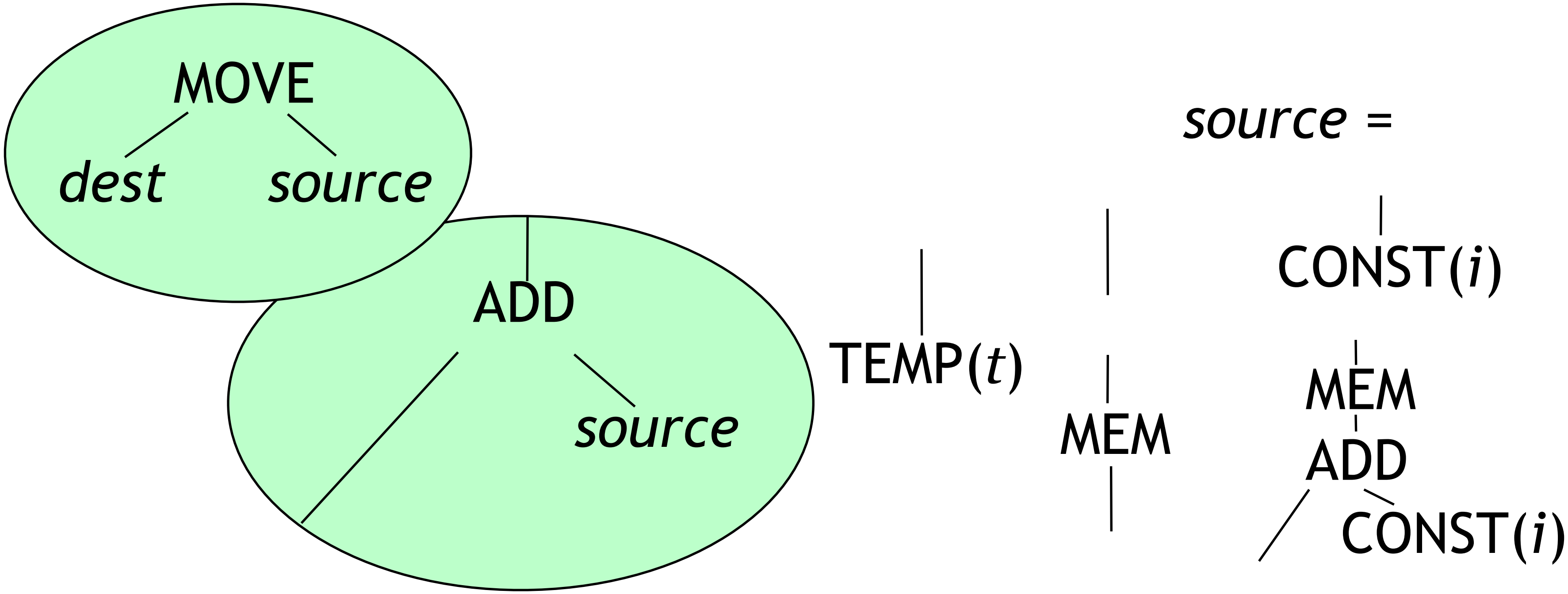

- Don't use instruction (jecxz)
- Let assembler figure out when to use (add eax, ...), bias register allocator

# Implementation

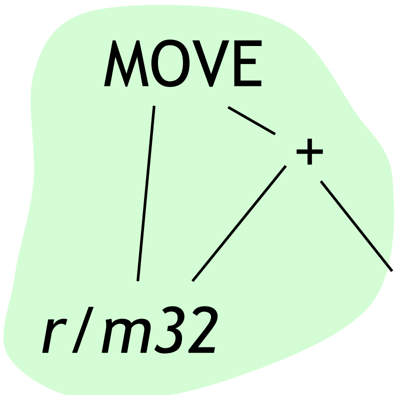
- Maximal Munch: start from statement node
- Find largest tile covering top node and matching all children
    - Invoke recursively on all children of *tile*
    - Generate code for this tile (code for children will have been generated already in recursive calls)
- How to find matching tiles?

# 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

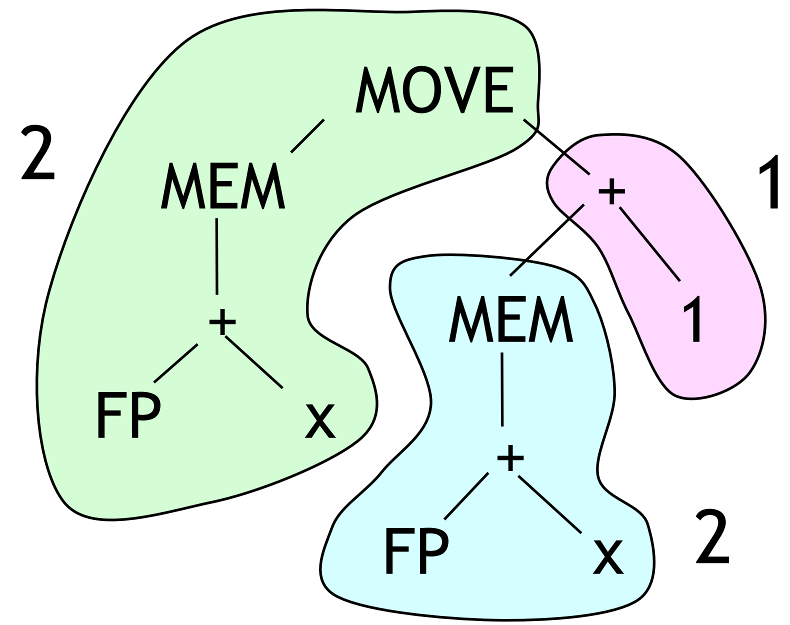
# Improving instruction selection

- 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

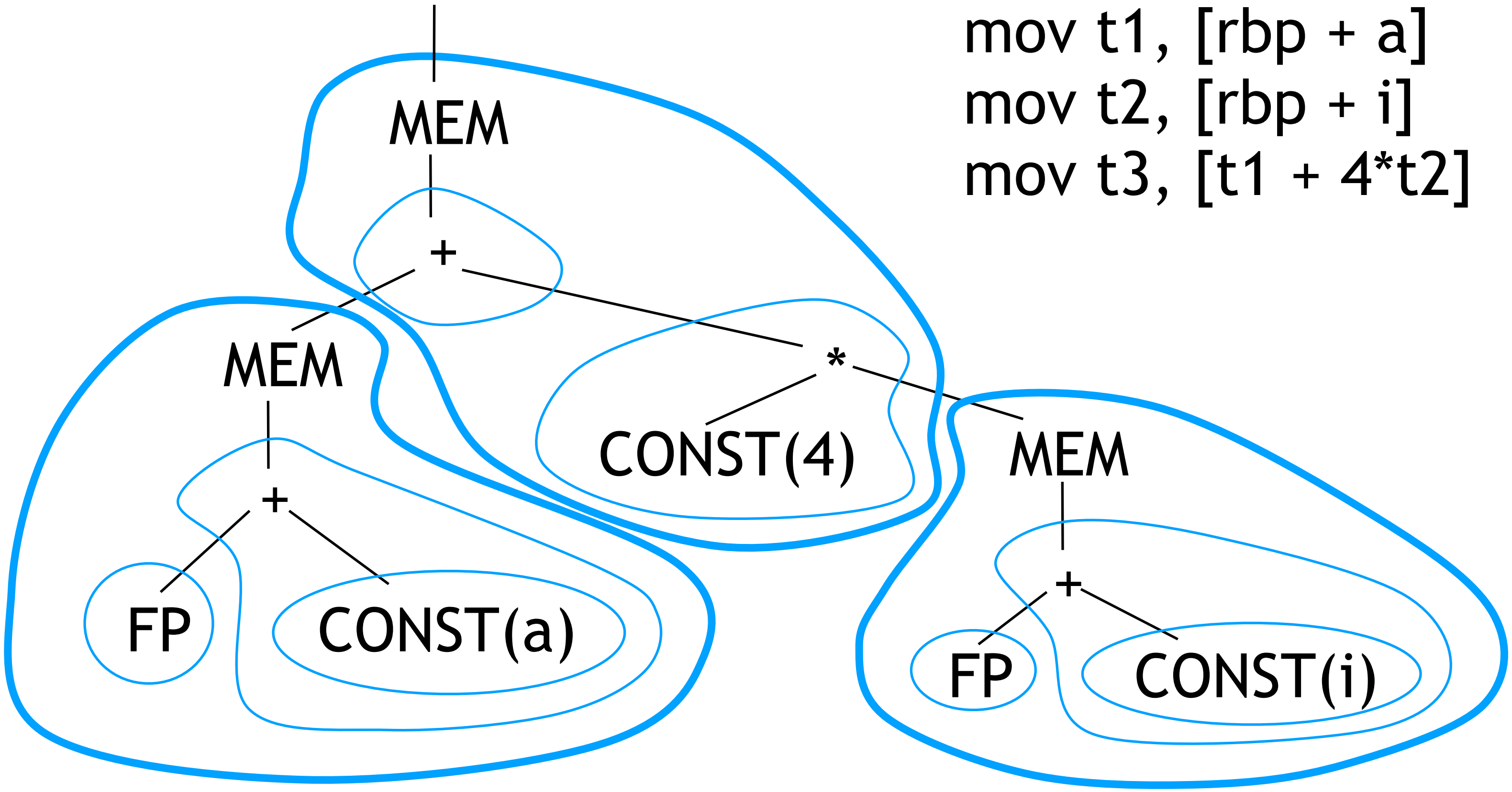
Total cost: 5



# 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]



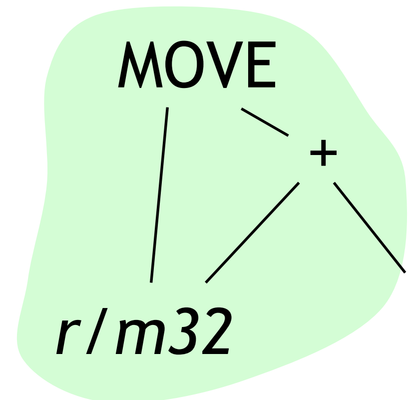

mov t1, [rbp + a]
mov t2, [rbp + i]
mov t3, [t1 + 4*t2]

# 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*