CS406: Compilers

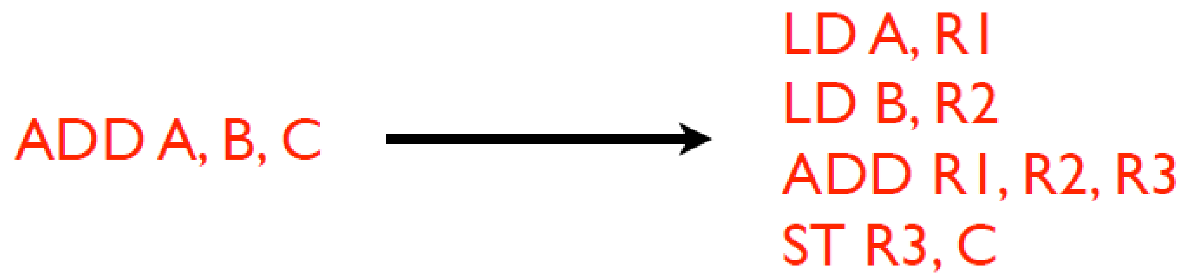
Spring 2021

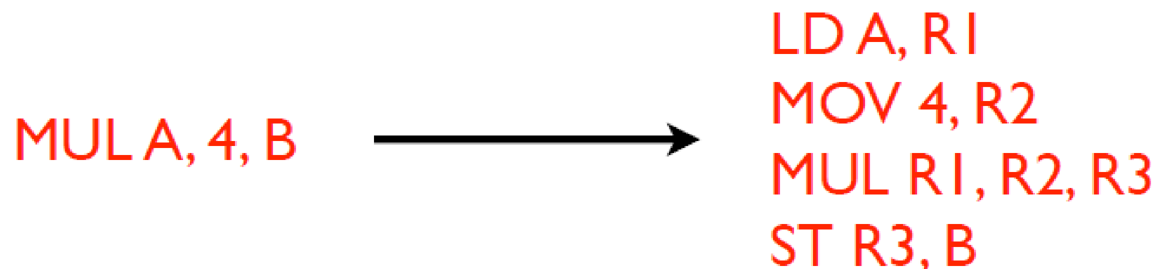
Week 10: Local Optimizations

(slide courtesy: Prof. Milind Kulkarni)

Naïve approach

- "Macro-expansion"
  - Treat each 3AC instruction separately, generate code in isolation





Why is this bad? (I)

MUL A, 4, B →

LD A, R1
MOV 4, R2
MUL R1, R2, R3
ST R3, B

MUL A, 4, B →

LD A, R1
MULI R1, 4, R3
ST R3, B

Too many instructions
Should use a different instruction type

Why is this bad? (II)

ADD A, B, C

LD A, R1
LD B, R2
ADD R1, R2, R3
ST R3, C

ADD A, B, C
ADD C, A, E

LD A, R1
LD B, R2
ADD R1, R2, R3
ST R3, C
LD C, R4
LD A, R5
ADD R4, R5, R6
ST R6, E

Redundant load of C
Redundant load of A
Uses a lot of registers

Why is this bad? (III)

ADD A, B, C →

```
LD A, R1
LD B, R2
ADD R1, R2, R3
ST R3, C
```

ADD A, B, C
ADD A, B, D →

```
LD A, R1
LD B, R2
ADD R1, R2, R3
ST R3, C
LD A, R4
LD B, R5
ADD R4, R5, R6
ST R6, D
```

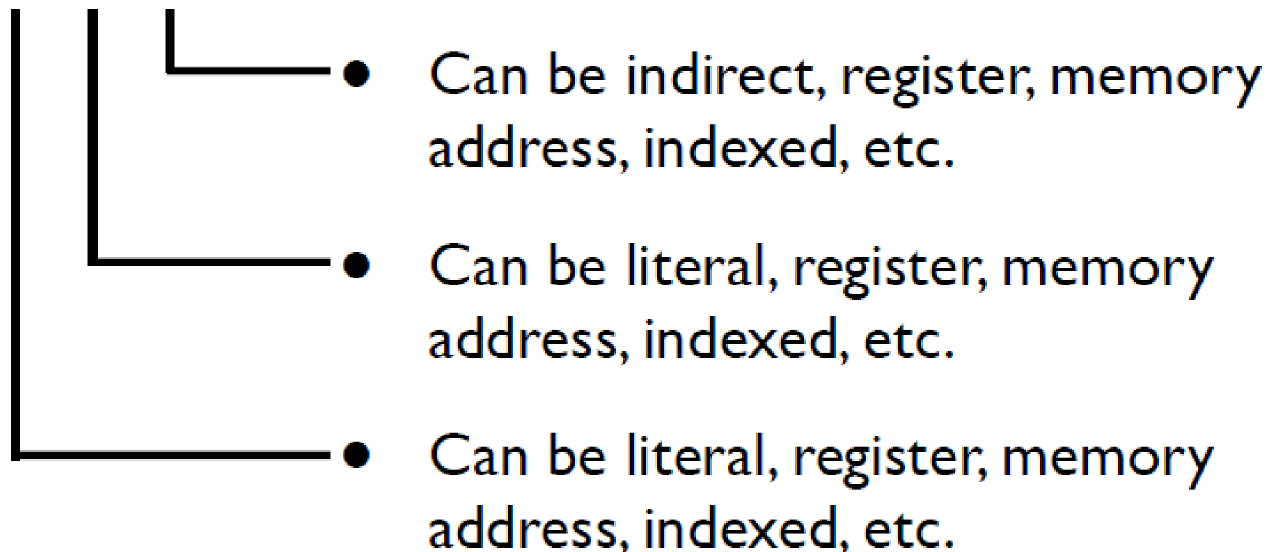
Wasting instructions recomputing A + B

How do we address this?

- Several techniques to improve performance of generated code
  - *Instruction selection* to choose better instructions
  - *Peephole optimizations* to remove redundant instructions
  - *Common subexpression elimination* to remove redundant computation
  - *Register allocation* to reduce number of registers used

Instruction selection

- Even a simple instruction may have a large set of possible address modes and combinations



- Dozens of potential combinations!

More choices for instructions

- Auto increment/decrement (especially common in embedded processors as in DSPs)
  - e.g., load from this address and increment it
  - Why is this useful?
- Three-address instructions
- Specialized registers (condition registers, floating point registers, etc.)
- "Free" addition in indexed mode

  MOV (R1)offset R2

  - Why is this useful?

Peephole optimizations

- Simple optimizations that can be performed by pattern matching
  - Intuitively, look through a "peephole" at a small segment of code and replace it with something better
  - Example: if code generator sees `ST R X; LD X R`, eliminate load
- Can recognize sequences of instructions that can be performed by single instructions

  `LDI R1 R2; ADD R1 4 R1` replaced by

  `LDINC R1 R2 4` //load from address in R1 then inc by 4

Peephole optimizations

- Simple optimizations that can be performed by pattern matching
  - Intuitively, look through a "peephole" at a small segment of code and replace it with something better
  - Example: if code generator sees `ST R X; LD X R`, eliminate load

Get the data present at address in R2 and put it in R1

be

`LDI R1 R2; ADD R1 4 R1` replaced by

`LDINC R1 R2 4` //load from address in R1 then inc by 4

Peephole optimizations

- Constant folding

```
ADD lit1, lit2, Rx  →  MOV lit1 + lit2, Rx
```

```
MOV lit1, Rx
ADD li2, Rx, Ry     →  MOV lit1 + lit2, Ry
```

- Strength reduction

```
MUL operand, 2, Rx  →  SHIFTL operand, 1, Rx
DIV operand, 4, Rx  →  SHIFTR operand, 2, Rx
```

- Null sequences

```
MUL operand, 1, Rx  →  MOV operand, Rx
ADD operand, 0, Rx  →  MOV operand, Rx
```

Peephole optimizations

- Combine operations

```
JEQ L1
JMP L2          →   JNE L2
L1: ...
```

- Simplifying

```
SUB operand, 0, Rx  →  NEG Rx
```

- Special cases (taking advantage of ++/--)

```
ADD 1, Rx, Rx   →   INC Rx
SUB Rx, 1, Rx   →   DEC Rx
```

- Address mode operations

```
MOV A R1
ADD 0(R1) R2 R3   →   ADD @A R2 R3
```

Superoptimization

- Peephole optimization/instruction selection writ large
- Given a sequence of instructions, find a different sequence of instructions that performs the same computation in less time
- Huge body of research, pulling in ideas from all across computer science
  - Theorem proving
  - Machine learning

Common subexpression elimination

- Goal: remove redundant computation, don't calculate the same expression multiple times

1: A = B * C
2: E = B * C

Keep the result of statement 1 in a temporary and reuse for statement 2

- Difficulty: how do we know when the same expression will produce the same result?

1: A = B * C
2: B = <new value>
3: E = B * C

B is "killed." Any expression using B is no longer "available," so we cannot reuse the result of statement 1 for statement 3

- This becomes harder with pointers (how do we know when B is killed?)

Common subexpression elimination

- Two varieties of common subexpression elimination (CSE)
- Local: within a single basic block
  - Easier problem to solve (why?)
- Global: within a single procedure or across the whole program
  - Intra- vs. inter-procedural
  - More powerful, but harder (why?)
  - Will come back to these sorts of “global” optimizations later

CSE in practice

- Idea: keep track of which expressions are "available" during the execution of a basic block
    - Which expressions have we already computed?
    - Issue: determining when an expression is no longer available
        - This happens when one of its components is assigned to, or "killed."
- Idea: when we see an expression that is already available, rather than generating code, copy the temporary
    - Issue: determining when two expressions are the same

Maintaining available expressions

- For each 3AC operation in a basic block
  - Create name for expression (based on lexical representation)
  - If name not in available expression set, generate code, add it to set
    - Track register that holds result of and any variables used to compute expression
  - If name in available expression set, generate move instruction
  - If operation assigns to a variable, kill all dependent expressions

Example

3 Address Code	Available expression(s)	Killed expression(s)	Generated Code (assembly)
ADD A B T1	{}		ld a r1; ld b r2; add r1 r2 r1
ADD T1 C T2	{“A + B”}		add r1 c r2
ADD A B T3	{“A + B”, “T1 + C”}		mov r1 r3
ADD T1 T2 C	{“A + B”, ~~“T1 + C”}~~	{“T1+C”}	add r1 r2 r5 st r5 c
ADD T1 C T4	{“A + B”, “T1 + T2”}		add r1 c r4
ADD T3 T2 D	{“A + B”, “T1 + T2”, “T1 + C”}		add r3 r2 r6 st r6 d
	{“A + B”, “T1 + T2”, “T1 + C”, “T3 + T2”}		

Downsides (CSE)

- What are some downsides to this approach? Consider the two highlighted operations

Three address code

```
+ A B T1
+ T1 C T2
+ A B T3
+ T1 T2 C
+ T1 C T4
+ T3 T2 D
```

Generated code

```
ADD A B R1
ADD R1 C R2
MOV R1 R3
ADD R1 R2 R5; ST R5 C
ADD R1 C R4
ST R5 D
```

T1 and T3 compute the same expression. This can be handled by an optimization called *value numbering.*

Aliasing

- One of the biggest problems in compiler analysis is to recognize aliases – different names for the same location in memory

exercise: are T1 and T3 aliased in previous example?

- Why do aliases occur?
  - Pointers referring to the same location
  - Function calls passing the same reference in two arguments
  - Arrays referencing the same element
  - Unions
- What problems does aliasing pose for CSE?
  - when talking about "live" and "killed" values in optimizations like CSE, we're talking about particular variable names
  - In the presence of aliasing, we may not know which variables get killed when a location is written to

Memory disambiguation

- Most compiler analyses rely on *memory disambiguation*
  - Otherwise, they need to be too conservative and are not useful
- Memory disambiguation is the problem of determining whether two references point to the same memory location
  - *Points-to* and *alias* analyses try to solve this
  - Will cover basic pointer analyses in a later lecture