Code generation and local optimization

Generating assembly

- How do we convert from three-address code to assembly?
- Seems easy! But easy solutions may not be the best option
- What we will cover:
  - Instruction selection
  - Peephole optimizations
  - “Local” common subexpression elimination
  - “Local” register allocation

Naïve approach

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

ADD A, B, C  \rightarrow  LD A, R1
                \quad LD B, R2
                \quad ADD R1, R2, R3
                \quad ST R3, C

MUL A, 4, B  \rightarrow  LD A, R1
                \quad MOV 4, R2
                \quad MUL R1, R2, R3
                \quad ST R3, B

Why is this bad? (I)

Too many instructions
Should use a different instruction type

MUL A, 4, B  \rightarrow  LD A, R1
                \quad MOV 4, R2
                \quad MULI R1, 4, R3
                \quad ST R3, B

Why is this bad? (I)

Too many instructions
Should use a different instruction type

MUL A, 4, B  \rightarrow  LD A, R1
                \quad MOV 4, R2
                \quad MUL R1, R2, R3
                \quad ST R3, B

Why is this bad? (I)

Too many instructions
Should use a different instruction type

MUL A, 4, B  \rightarrow  LD A, R1
                \quad MULI R1, 4, R3
                \quad ST R3, B
Why is this bad? (II)

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

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

Why is this bad? (II)

ADD A, B, C
ADD A, B, D

Why is this bad? (II)

ADD A, B, C
ADD A, B, D

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
  
  + A B C
  
  - Can be indirect, register, memory address, indexed, etc.
  - Can be literal, register, memory address, indexed, etc.
  - Can be literal, register, memory address, indexed, etc.

- 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
  
  LDC R1 R2 4 /load from address in R1 then inc by 4

**Peephole optimizations**

- Constant folding
  
  ADD lit1, lit2, Rx \(\rightarrow\) MOV lit1 + lit2, Rx
  
  MOV lit1, Rx \(\rightarrow\) MOV lit1 + lit2, Ry

- Strength reduction
  
  MUL operand, 2, Rx \(\rightarrow\) SHIFTL operand, 1, Rx
  
  DIV operand, 4, Rx \(\rightarrow\) SHIFTR operand, 2, Rx

- Null sequences
  
  MUL operand, 1, Rx \(\rightarrow\) MOV operand, Rx
  
  ADD operand, 0, Rx \(\rightarrow\) MOV operand, Rx

---

**Peephole optimizations**

- Combine operations
  
  EQ L1
  
  JMP L2
  
  L1: ...

- Simplifying
  
  SUB operand, 0, Rx \(\rightarrow\) NEG Rx

- Special cases (taking advantage of ++/--)
  
  ADD 1, Rx, Rx \(\rightarrow\) INC Rx
  
  SUB Rx, 1, Rx \(\rightarrow\) DEC Rx

- Address mode operations
  
  MOV A R1
  
  ADD 0(R1) R2 R3 \(\rightarrow\) ADD @A R2 R3

---

**Superoptimizerization**

- 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: \( A = B \times C \)  
  2: \( E = B \times C \)  
  
  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: \( A = B \times C \)  
  2: \( B = \text{<new value>} \)  
  3: \( E = B \times C \)  
  
  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?)

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

<table>
<thead>
<tr>
<th>Three address code</th>
<th>Generated code</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ A B T1</td>
<td></td>
</tr>
<tr>
<td>+ T1 C T2</td>
<td></td>
</tr>
<tr>
<td>+ A B T3</td>
<td></td>
</tr>
<tr>
<td>+ T1 T2 C</td>
<td></td>
</tr>
<tr>
<td>+ T1 C T4</td>
<td></td>
</tr>
<tr>
<td>+ T3 T2 D</td>
<td></td>
</tr>
</tbody>
</table>

Available expressions: "A+B"

Example

<table>
<thead>
<tr>
<th>Three address code</th>
<th>Generated code</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ A B T1</td>
<td>ADD A B R1</td>
</tr>
<tr>
<td>+ T1 C T2</td>
<td></td>
</tr>
<tr>
<td>+ A B T3</td>
<td></td>
</tr>
<tr>
<td>+ T1 T2 C</td>
<td></td>
</tr>
<tr>
<td>+ T1 C T4</td>
<td></td>
</tr>
<tr>
<td>+ T3 T2 D</td>
<td></td>
</tr>
</tbody>
</table>

Available expressions: "A+B"
### Example

<table>
<thead>
<tr>
<th>Three address code</th>
<th>Generated code</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+A\ B\ T1)</td>
<td>(\text{ADD}\ A\ B\ R1)</td>
</tr>
<tr>
<td>(+T1\ C\ T2)</td>
<td>(\text{ADD}\ R1\ C\ R2)</td>
</tr>
<tr>
<td>(+A\ B\ T3)</td>
<td>(\text{ADD}\ R1\ R2\ R5;\ \text{ST}\ R5\ C)</td>
</tr>
<tr>
<td>(+T1\ T2\ C)</td>
<td>(\text{ADD}\ R1\ C\ R4)</td>
</tr>
<tr>
<td>(+T3\ T2\ D)</td>
<td>(\text{ADD}\ R3\ R2\ R6;\ \text{ST}\ R6\ D)</td>
</tr>
</tbody>
</table>

Available expressions: “\(+A\ B\)”, “\(+T1\ C\)”, “\(+T1\ T2\)”, “\(+T3\ T2\)”, “\(+T1\ T2\)”, “\(+T1\ C\)”.

### Example

<table>
<thead>
<tr>
<th>Three address code</th>
<th>Generated code</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+A\ B\ T1)</td>
<td>(\text{ADD}\ A\ B\ R1)</td>
</tr>
<tr>
<td>(+T1\ C\ T2)</td>
<td>(\text{ADD}\ R1\ C\ R2)</td>
</tr>
<tr>
<td>(+A\ B\ T3)</td>
<td>(\text{MOV}\ R1\ R3)</td>
</tr>
<tr>
<td>(+T1\ T2\ C)</td>
<td>(\text{ADD}\ R1\ R2\ R5;\ \text{ST}\ R5\ C)</td>
</tr>
<tr>
<td>(+T3\ T2\ D)</td>
<td>(\text{ADD}\ R3\ R2\ R6;\ \text{ST}\ R6\ D)</td>
</tr>
</tbody>
</table>

Available expressions: “\(+A\ B\)”, “\(+T1\ T2\)”, “\(+T1\ C\)”, “\(+T3\ T2\)”, “\(+T1\ T2\)”, “\(+T1\ C\)”.

### Example

<table>
<thead>
<tr>
<th>Three address code</th>
<th>Generated code</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+A\ B\ T1)</td>
<td>(\text{ADD}\ A\ B\ R1)</td>
</tr>
<tr>
<td>(+T1\ C\ T2)</td>
<td>(\text{ADD}\ R1\ C\ R2)</td>
</tr>
<tr>
<td>(+A\ B\ T3)</td>
<td>(\text{ADD}\ R1\ R2\ R5;\ \text{ST}\ R5\ C)</td>
</tr>
<tr>
<td>(+T1\ T2\ C)</td>
<td>(\text{ADD}\ R1\ C\ R4)</td>
</tr>
<tr>
<td>(+T3\ T2\ D)</td>
<td>(\text{ADD}\ R3\ R2\ R6;\ \text{ST}\ R6\ D)</td>
</tr>
</tbody>
</table>

Available expressions: “\(+A\ B\)”, “\(+T1\ T2\)”, “\(+T1\ C\)”, “\(+T3\ T2\)”, “\(+T1\ T2\)”, “\(+T1\ C\)”.

### Example

<table>
<thead>
<tr>
<th>Three address code</th>
<th>Generated code</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+A\ B\ T1)</td>
<td>(\text{ADD}\ A\ B\ R1)</td>
</tr>
<tr>
<td>(+T1\ C\ T2)</td>
<td>(\text{ADD}\ R1\ C\ R2)</td>
</tr>
<tr>
<td>(+A\ B\ T3)</td>
<td>(\text{MOV}\ R1\ R3)</td>
</tr>
<tr>
<td>(+T1\ T2\ C)</td>
<td>(\text{ADD}\ R1\ R2\ R5;\ \text{ST}\ R5\ C)</td>
</tr>
<tr>
<td>(+T1\ C\ T4)</td>
<td>(\text{ADD}\ R1\ C\ R4)</td>
</tr>
<tr>
<td>(+T3\ T2\ D)</td>
<td>(\text{ADD}\ R3\ R2\ R6;\ \text{ST}\ R6\ D)</td>
</tr>
</tbody>
</table>

Available expressions: “\(+A\ B\)”, “\(+T1\ T2\)”, “\(+T1\ C\)”, “\(+T3\ T2\)”, “\(+T1\ T2\)”, “\(+T1\ C\)”.

### Downsides

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

<table>
<thead>
<tr>
<th>Three address code</th>
<th>Generated code</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+A\ B\ T1)</td>
<td>(\text{ADD}\ A\ B\ R1)</td>
</tr>
<tr>
<td>(+T1\ C\ T2)</td>
<td>(\text{ADD}\ R1\ C\ R2)</td>
</tr>
<tr>
<td>(+A\ B\ T3)</td>
<td>(\text{MOV}\ R1\ R3)</td>
</tr>
<tr>
<td>(+T1\ T2\ C)</td>
<td>(\text{ADD}\ R1\ R2\ R5;\ \text{ST}\ R5\ C)</td>
</tr>
<tr>
<td>(+T1\ C\ T4)</td>
<td>(\text{ADD}\ R1\ C\ R4)</td>
</tr>
<tr>
<td>(+T3\ T2\ D)</td>
<td>(\text{ADD}\ R3\ R2\ R6;\ \text{ST}\ R6\ D)</td>
</tr>
</tbody>
</table>

Available expressions: “\(+A\ B\)”, “\(+T1\ T2\)”, “\(+T1\ C\)”, “\(+T3\ T2\)”, “\(+T1\ T2\)”, “\(+T1\ C\)”.
### Downsides
- What are some downsides to this approach? Consider the two highlighted operations

**Three address code**
- ADD A B R1
- ADD R1 R2
- MOV R1 R3
- ADD R1 R2 R5; ST RS C
- ADD R1 C R4
- ST RS D

**Generated code**

- This can be handled by an optimization called **value numbering**, which we will not cover now (although we may get to it later)

### Aliasing
- One of the biggest problems in compiler analysis is to recognize aliases – different names for the same location in memory
- Aliases can occur for many reasons
  - Pointers referring to same location, arrays referencing the same element, function calls passing the same reference in two arguments, explicit storage overlapping (unions)
  - Upshot: 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

### Register allocation
- Simple code generation: use a register for each temporary, load from a variable on each read, store to a variable at each write

- **Problems**
  - Real machines have a limited number of registers – one register per temporary may be too many
  - Loading from and storing to variables on each use may produce a lot of redundant loads and stores
  - Goal: allocate temporaries and variables to registers to:
    - Use only as many registers as machine supports
    - Minimize loading and storing variables to memory (keep variables in registers when possible)
    - Minimize putting temporaries on stack

### Global vs. local
- Same distinction as global vs. local CSE
- Local register allocation is for a single basic block
- Global register allocation is for an entire function (but not interprocedural – why?)

- Will cover some local allocation strategies now, global allocation later

### Top-down register allocation
- For each basic block
  - Find the number of references of each variable
  - Assign registers to variables with the most references

- **Details**
  - Keep some registers free for operations on unassigned variables and spilling
  - Store dirty registers at the end of BB (i.e., registers which have variables assigned to them)
  - Do not need to do this for temporaries (why?)
**Bottom-up register allocation**

- Smarter approach:
  - Free registers once the data in them isn’t used anymore

- Requires calculating **liveness**
  - A variable is live if it has a value that may be used in the future

- Easy to calculate if you have a single basic block:
  - Start at end of block, all local variables marked dead
    - If you have multiple basic blocks, all local variables should be live (they may be used in the future)
  - When a variable is used, mark as live, record use
  - When a variable is defined, record def, variable dead above this
  - Creates chains linking uses of variables to where they were defined
- We will discuss how to calculate this across BBs later

**Liveness example**

- What is live in this code?

```
1:  A = B + C
2:  C = A + B
3:  T1 = B + C
4:  T2 = T1 + C
5:  D = T2
6:  E = A + B
7:  B = E + D
8:  A = C + D
9:  T3 = A + B
10: WRITE(T3)
```

**Liveness example**

- What is live in this code?

```
1:  A = B + C
2:  C = A + B
3:  T1 = B + C
4:  T2 = T1 + C
5:  D = T2
6:  E = A + B
7:  B = E + D
8:  A = C + D
9:  T3 = A + B
10: WRITE(T3)
```

**Bottom-up register allocation**

- Free registers once the data in them isn’t used anymore

- Requires calculating **liveness**
  - A variable is live if it has a value that may be used in the future

- Easy to calculate if you have a single basic block:
  - Start at end of block, all local variables marked dead
    - If you have multiple basic blocks, all local variables should be live (they may be used in the future)
  - When a variable is used, mark as live, record use
  - When a variable is defined, record def, variable dead above this
  - Creates chains linking uses of variables to where they were defined
- We will discuss how to calculate this across BBs later

**Liveness example**

- What is live in this code?

```
1:  A = B + C
2:  C = A + B
3:  T1 = B + C
4:  T2 = T1 + C
5:  D = T2
6:  E = A + B
7:  B = E + D
8:  A = C + D
9:  T3 = A + B
10: WRITE(T3)
```

**Bottom-up register allocation**

- Free registers once the data in them isn’t used anymore

- Requires calculating **liveness**
  - A variable is live if it has a value that may be used in the future

- Easy to calculate if you have a single basic block:
  - Start at end of block, all local variables marked dead
    - If you have multiple basic blocks, all local variables should be live (they may be used in the future)
  - When a variable is used, mark as live, record use
  - When a variable is defined, record def, variable dead above this
  - Creates chains linking uses of variables to where they were defined
- We will discuss how to calculate this across BBs later

---

**Bottom-up register allocation**

```
ensure(opr)
if opr is already in register r
  return r
else
  r = allocate(opr)
generate load from opr into r
return r
```

```free(r)
if r is marked dirty and variable is live
  generate store
mark r as free
```

```allocate(opr)
if there is a free r
  choose r
else
  choose r with most distant use
mark r associated with opr
return r
```

**Example**

- Perform register allocation for this code:

```
1:  A = B + C
2:  C = A + B
3:  T1 = B + C
4:  T2 = T1 + C
5:  D = T2
6:  E = A + B
7:  B = E + D
8:  A = C + D
9:  T3 = A + B
10: WRITE(T3)
```
### Aliasing, as usual, is a problem

- What happens with this code?
  ```c
  // a and b are aliased
  LD a R1
  LD b R2
  ADD R1 R2 R3
  ST R3 c // c = a + b
  R1 = 7 // a = 7
  ADD R1 R2 R4
  ST R4 d // d = a + b
  ```

### Dealing with aliasing

- Immediately before loading a variable \( x \)
- For each variable aliased to \( x \) that is already in a dirty register, save it to memory (i.e., perform a store)
- This ensures that we load the right value
- Immediately before writing to a register holding \( x \)
- For each register associated with a variable aliased to \( x \), mark it as invalid
- So next time we use the variable, we will reload it
- Conservative approach: assume all variables are aliased (in other words, reload from memory on each read, store to memory on each write)
- Better alias analysis can improve this
- At subroutine boundaries, still often use conservative analysis

### Allocation considerations

- Use register coloring to perform global register allocation
- Will see this next
- Find right order of optimizations and register allocation
- Peephole optimizations can reduce register pressure, can make allocation better
- CSE can actually increase register pressure
- Different orders of optimization produce different results
- Register allocation still an open research area
- For example, how to do allocation for JIT compilers