COEN 175

Lecture 17: More Code Generation for Expressions

Sethi-Ullman Algorithm

- The **Sethi-Ullman algorithm** generates code for an expression using the fewest number of registers.
- The key insight is to evaluate the subtree that requires the most registers first.
- Thus, we must be able to generate code in an order different from the left-to-right parse order.
- The algorithm works on the abstract syntax tree (AST) and requires two passes over the tree:
  1. Label the abstract syntax tree.
  2. Generate optimal code based on the labeled tree.

Labeling

- Each node is assigned an integer **label** representing the number of registers it requires.
- The label of each node is computed bottom-up:
  - The label of a left leaf is 1.
  - The label of a right leaf is 0.
  - The label of a binary node, *parent*, with children *left* and *right* is computed as follows:

```
if label(left) = label(right) then
    label(parent) ← label(left) + 1
else
    label(parent) ← max(label(left), label(right))
```

Example 1: Labels

- Label the expression `a * b + c * d`.
  1. label(`a`) = 1
  2. label(`b`) = 0
  3. label(`a * b`) = max(1, 0) = 1
  4. label(`c`) = 1
  5. label(`d`) = 0
  6. label(`c * d`) = max(1, 0) = 1
  7. label(`a * b + c * d`) = 1 + 1 = 2
- The Sethi-Ullman algorithm tells us that two registers are required to evaluate this expression.

Example 2: Labels

- Label the expression `a + b * c – (a + b + c)`.
  1. label(a) = 1
  2. label(b) = 1
  3. label(c) = 0
  4. label(b * c) = max(1, 0) = 1
  5. label(a + b * c) = 1 + 1 + 2
  6. label(a) = 1
  7. label(b) = 0
  8. label(a + b) = max(1, 0) = 1
  9. label(c) = 0
  10. label(a + b + c) = max(1, 0) = 1
  11. label(a + b * c – (a + b + c)) = max(2, 1) = 2

Example 3: Labels

- Label the expression `a * b - (c * d + (a – b * c))`.
  1. label(a) = 1
  2. label(b) = 0
  3. label(a * b) = max(1, 0) = 1
  4. label(c) = 1
  5. label(d) = 0
  6. label(c * d) = max(1, 0) = 1
  7. label(a) = 1
  8. label(b) = 1
  9. label(c) = 0
  10. label(b * c) = max(1, 0) = 1
  11. label(a – b * c) = 1 + 1 = 2
  12. label(c * d + (a – b * c)) = max(1, 2) = 2
  13. label(a * b – (c * d + (a – b * c))) = max(1, 2) = 2

Generating Optimal Code

- To generate code requiring the fewest number of registers, we always generate code for the child with the **larger label first**.
  - If both children have the same label, the order is irrelevant.
- Suppose one child requires m registers and the other child requires n registers, where $m < n$.
  - The larger child requires n registers, but in the end the result is always held in a single register.
  - The smaller child requires $m + 1$ registers: m registers for itself and 1 register to hold the larger child's result.
  - Since $m < n$, we will use at most n registers.

Example 1: Code

- Generate code for `a * b + c * d`.
  - Since the two children of the addition have the same label, it does not matter which child we do first:

```
movl   a, %eax
imull  b, %eax
movl   c, %ecx
imull  d, %ecx
addl   %ecx, %eax
```

Left operand of addition done first

```
movl   c, %eax
imull  d, %eax
movl   a, %ecx
imull  b, %ecx
addl   %eax, %ecx
```

Right operand of addition done first

- Either choice requires two registers.

Example 2: Code

- Generate code for `a + b * c – (a + b + c)`.
  - Except for the tie between `a` and `b * c` in the left operand of the subtraction, there are no choices.

```
movl   b, %eax
imull  c, %eax
movl   a, %ecx
addl   %eax, %ecx
movl   a, %eax
addl   b, %eax
addl   c, %eax
subl   %eax, %ecx
```

- This is the same code we generated previously.

Example 3: Code

- Generate code for `a * b - (c * d + (a – b * c))`.
  - Except for the tie between `a` and `b * c` in the right operand of the addition, there are no choices.

```
movl   b, %eax
imull  c, %eax
movl   a, %ecx
subl   %eax, %ecx
movl   c, %eax
imull  d, %eax
addl   %ecx, %eax
movl   a, %ecx
imull  b, %ecx
subl   %eax, %ecx
```

Simple Arithmetic Operators

- Most binary operators follow the same template:
  1. Generate code for one child.
  2. Generate code for the other child.
  3. If the left child is not in a register, then load it.
  4. Perform the operation, overwriting the left child's register.
  5. Deallocate any register for the right operand.
- For addition, subtraction, and multiplication, performing the operation itself is simple.
  - Addition = `add`, subtraction = `sub`, multiplication = `imul`

Division and Remainder

- Division and remainder are slow operations.
  - Many compilers avoid division or remainder by a constant.
- Dividing two 32-bit operands requires a 64-bit dividend, and two 64-bit operands requires a 128-bit dividend.
- Intel requires special registers for division.
  - The `EDX:EAX` pair holds the 64-bit dividend.
  - Similarly, the `RDX:RAX` pair holds the 128-bit dividend.
  - After division, `EAX` (or `RAX` if 64-bit) holds the quotient and `EDX` (or `RDX`) holds the remainder.

Division: Example

- Generate code for `x / y + z`, using 32-bit operands.

```
movl    x, %eax         # load: %eax allocated
movl    %eax, %edx      # sign extend %eax into %edx
sarl    $31, %edx
idivl   y               # %edx:%eax / y
addl    z, %eax
```

- Note that the divide instruction, `idiv`, requires only the divisor as its single operand.
- The dividend is implicitly specified as the `%edx:%eax` register pair.

Division: Quirks

- Special instructions are available for sign-extension: `cltd` (32-bit to 64-bit) and `cqto` (64-bit to 128-bit).
- The Intel division instruction allows as an operand a register or memory reference, but not an immediate.
  - So, `x / 7` will not work using our template.
  - We must first load the immediate into a register.
- Generate code for `x / 7`.

```
movl   x, %eax
cltd
movl   $7, %ecx
idivl  %ecx
```

Comparison Operators

- The comparison operators in C yield 1 if the comparison is true, and 0 if the comparison is false.
- In Intel assembly, the `cmp` instruction sets internal condition codes or **flags**, which can then be tested:
  - `sete` – set if equal
  - `setne` – set if not equal
  - `setl` – set if less than
  - `setle` – set if less than or equal
  - `setg` – set if greater than
  - `setge` – set if greater than or equal

Comparison: Example

- The various `set` instructions require a byte register, which can then be zero-extended using `movzbl`.
- Generate code for `(x > y) + z`.

```
movl   x, %eax
cmpl   y, %eax    # test and set condition codes
setg   %al        # set %al based on condition codes
movzbl %al, %eax  # move zero-extend byte to long
addl   z, %eax
```

- Note that `AL` is the low byte of `AX`/`EAX`/`RAX`.
  - Not all registers have an analogous byte register.
  - `ESI` and `EDI` do have have a byte register.

Intel Register Summary

64-bit	32-bit	16-bit	8-bit
RAX	EAX	AX	AL
RBX	EBX	BX	BL
RCX	ECX	CX	CL
RDX	EDX	DX	DL
RSI	ESI	SI	SIL
RDI	EDI	DI	DIL
RBP	EBP	BP	BPL
RSP	ESP	SP	SPL
R8–R15	R8d–R15d	R8w–R15w	R8b–R15b

`R8–R15` are new registers available only in 64-bit mode.
`SIL`, `DIL`, `BPL`, and `SPL` are only available in 64-bit mode, not 32-mode.