CS 322: Languages and Compiler Design II

Mark P Jones, Portland State University

Spring 2015

Week 5: Assembly code generation

Why translation is needed

- We like to write programs at a higher-level than the machine can execute directly
  - Spreadsheet: `sum [A1:A3]`
  - Java: `a[1] + a[2] + a[3]`
  - Machine language:
    ```
    movl $0, %eax
    addl 4(a), %eax
    addl 8(a), %eax
    addl 12(a), %eax
    ```
- High-level languages let us describe what is to be done without worrying about all the details
- In machine languages, every step must be carefully spelled out

Translating from source to target

- The source language is the driving force in the earlier compiler phases, from source input to semantic analysis
- As we move to the backend, the target language becomes much more important
- But the source language still has a role to play:
  - How do we map source language datatypes and primitives to the target machine?
  - How do we map source program constructs to the target machine instruction set?

Our source

- To begin with, we will assume a very simple imperative programming language as our source language:

```
int i = 0;     // initialize
while (i <= 10) {
  print i*i;  // print a square
  i = i + 1;
}
```

- Later, we'll extend this to take a close look at the details of how functions and objects are implemented

Our target

- To begin with:
  - we will look at general techniques and concepts
  - we will generate x86-64 assembly language directly from abstract syntax trees
- Later, we will consider an alternative approach to code generation that introduces an *intermediate language* that allows better code generation and optimization (but requires a more sophisticated compiler)

First steps to code generation

"Doing" vs "Thinking about doing"

- Compilers translate programs (turning syntax to syntax)
- Interpreters run programs (turning syntax to semantics)
- Example:
  - Use your calculator to evaluate (1+2)+(3+4):
    Answer: 10
  - Tell me what buttons to press to evaluate (1+2)+(3+4):
    Answer: 1 + 2 = M 3 + 4 + MR =
- We'll focus on compilers for now, but will use many of the same tools to work with interpreters later.

Thinking about translating

There are two levels of translation that we need to consider:

- How will values of types in the source language be mapped to values of types in the target language?
- How will the constructs of the source language be mapped to the constructs of the target language?

Translating values

- How do we represent source language values in the target language?
- We only have a few types to worry about so far:
  - We'll represent integers by 32 bit words
  - We'll represent booleans by 32 bit words using
    - 0 to represent false
    - 1 to represent true
- As we extend the language to work with other kinds of values (e.g., arrays, objects, functions), we'll need to provide corresponding representations.

Translating statements & expressions

- A statement describes an action:
  - so compilation of a statement should produce code to execute that action
- An expression describes a value:
  - so compilation of an expression should produce code that can be executed to calculate that value
- To make this work, we need:
  - a general strategy
  - a mapping of that strategy onto our target machine

A general strategy

- Our initial strategy for compiling an expression will be
  - produce code that will evaluate the expression and leave the result in `eax`
  - use the stack to store temporary values
- We won't be making much use of other registers yet
  - a reasonable choice on a target machine with very few registers
  - an obvious area for improvement later on ...

Compiling addition

- For example, to compile the expression e_1+e_2, we need to produce code of the form:

```
code to evaluate e1, leaving the result in eax
pushl %eax
code to evaluate e2, leaving the result in eax
popl %edi
addl %edi, %eax
```

- The final result is in `eax`
- Temporary values were saved on the stack
- Each recursive call follows exactly the same pattern

A more formal/concise description

- E⟦e⟧ generates code that evaluates the expression e and leaves the result in `eax`
  - E⟦e_1+e_2⟧ = E⟦e_1⟧
    ```
    pushl  %eax
    ```
    E⟦e_2⟧
    ```
    popl   %edi
    addl   %edi, %eax
    ```
  - E⟦var⟧ = `movl   var,%eax`
  - E⟦num⟧ = `movl   $num, %eax`

... continued

- E⟦e_1 && e_2⟧

  = E⟦e_1⟧
  ```
  cmpl   $1, %eax
  jnz    lab1
  ```
  a new label

  E⟦e_2⟧
  ```
  lab1:
  ```
- and so on ...

Or, in Java

- Add an abstract method to the `Expr` class:

```
abstract void compile(Assembly a);
```

- Add the following to the `Add` class:

```
void compile(Assembly a) {
  left.compile(a);
  a.emit("pushl", "%eax");
  right.compile(a);
  a.emit("popl", "%edi");
  a.emit("addl", "%edi", "%eax");
}
```


... continued

- Add the following to the `IntLiteral` class:

```
void compile(Assembly a) {
  a.emit("movl", a.immed(num), "%eax");
}
```

- Add the following to the `Var` class:

```
void compile(Assembly a) {
  a.emit("movl", varName, "%eax");
}
```

- And so on ...

The `Assembly` class

- We will use objects of type `Assembly` to represent assembly output files:

```
class Assembly {
  static Assembly assembleToFile(String name);

  String newLabel();
  void emitLabel(String lab);

  void emit(String op);
  void emit(String op, String op1);
  void emit(String op, String op1, String op2);

  String immed(int v);                    // $v
  String indirect(int n, String s);       // n(s)
}
```


Compiling statements

- S⟦s⟧ generates code that executes the statement s
  - S⟦e;⟧ = E⟦e⟧
  - S⟦if (e) s_1 else s_2⟧

    = E⟦e⟧
    ```
    cmpl   $1,%eax
    jnz    lab1
    ```
    S⟦s_1⟧
    ```
    jmp    lab2
    lab1:
    ```
    S⟦s_2⟧
    ```
    lab2:
    ```

... continued

- A rule for compiling a while loop:
  - S⟦while (e) s⟧

```
        = lab1:
              E[[e]]
              cmpl   $1,%eax
              jnz    lab2
              S[[s]]
              jmp    lab1
          lab2:
```

(lab1, lab2 are new labels)

Our first optimization!

- An alternative rule for compiling a while loop:
  - S⟦while (e) s⟧

```
        =      jmp    lab2
          lab1:
               S[[s]]
          lab2:
               E[[e]]
               cmpl   $1,%eax
               jz     lab1
```

- Question: when is this an improvement? Why?

For example: (2+4)*(3+5)

```
                     # eax   edi   stack
movl   $2, %eax      #  2
pushl  %eax          #  2           2
movl   $4, %eax      #  4           2
popl   %edi          #  4      2
addl   %edi, %eax    #  6      2
pushl  %eax          #  6      2    6
movl   $3, %eax      #  3      2    6
pushl  %eax          #  3      2    6 3
movl   $5, %eax      #  5      2    6 3
popl   %edi          #  5      3    6
addl   %edi, %eax    #  8      3    6
popl   %edi          #  8      6
imull  %edi, %eax    # 48      6
```


We need register allocation!

- Why use the stack? We have more than two registers!

```
movl  $2, %eax
movl  $4, %edi
addl  %edi, %eax
movl  $3, %edi
movl  $5, %esi
addl  %esi, %edi
imull %edi, %eax
```

- How can we modify our compilation schemes to make better use of machine registers?

We need better compilation schemes!

- Why stick to these simple instructions? Other choices can reduce the size of the generated code:

```
movl  $2, %eax
addl  $4, %eax
movl  $3, %edi
addl  $5, %edi
imull %edi, %eax
```

- This just requires a specialized version of our original compilation scheme:
  - E⟦e+n⟧ = E⟦e⟧

```
            addl $n,%eax
```


Summary

- The constructs of high-level languages can be implemented using sequences of machine language instructions
- It is important to select good target instructions for each source construct
- Compilation schemes can be used to describe the mapping between languages
- To get better code quality, we should:
  - Make good use of the target machine's instructions, registers, etc...
  - Use more refined compilation schemes to deal with special cases

Improving code quality

Code quality

- Our simple assembly code generator does ok ...
- But the quality of the generated code is poor:
  - Program size suffers because the generated instruction sequences are quite long
  - Program execution time suffers because there are redundant instructions/memory transfers, and because there is poor use of CPU registers
- In short, the code generator does not make good use of the facilities that the machine provides

An example

- With the E compilation scheme, the statement

```
    total = total + count * count;
```

compiles to

```
movl    total, %eax
pushl   %eax
movl    count, %eax
pushl   %eax
movl    count, %eax
popl    %edi
imull   %edi, %eax
popl    %edi
addl    %edi, %eax
movl    %eax, total
```

- 10 instructions, 8 of which are accessing memory

An example

- With the E compilation scheme, the statement

```
    total = total + count * count;
```

compiles to

```
movl    total, %eax
pushl   %eax
movl    count, %eax
pushl   %eax
movl    count, %eax
popl    %edi
imull   %edi, %eax
popl    %edi
addl    %edi, %eax
movl    %eax, total
```

- Some of the memory accesses are unavoidable ...

An example

- With the E compilation scheme, the statement

```
    total = total + count * count;
```

compiles to

```
movl    total, %eax
pushl   %eax
movl    count, %eax
pushl   %eax
movl    count, %eax
popl    %edi
imull   %edi, %eax
popl    %edi
addl    %edi, %eax
movl    %eax, total
```

- But some of them seem avoidable if we use registers ...

Making better use of registers

- Up to now, we've used only two registers in generated code
- Imagine that we can have as many registers as we need. Call them r(0), r(1), r(2), r(3), r(4), r(5), ...
- What code might we generate then?
- Instead of storing our stack of intermediate results in memory, we could save them in registers ...
- Instead of using a stack pointer at run time, the compiler can figure out where the stack pointer will be ... at compile time ...

Another compilation scheme

- Let's introduce a new compilation scheme to generate code using our stack of registers.
- $E_r[\![e]\!]$(free) generates code that will evaluate the expression e and leave the result in register r(free) without changing any of the values in any of the lower numbered registers.
- For example:

$E_r[\![e_1+e_2]\!]$(free) = $E_r[\![e_1]\!]$(free)
$E_r[\![e_2]\!]$(free+1)
`addl r(free+1), r(free)`

Register allocation

- The process of arranging for intermediate values to be held in registers rather than memory locations is known as register allocation
- That's why I've called the new compilation scheme E_r:
  E for expressions and r for register allocation
- There are many different schemes/algorithms for register allocation
- The approach that I'm using here is fairly simple (i.e., good for understanding the concepts), but real compilers use more sophisticated algorithms that can make significantly better use of registers.

More examples

$E_r[\![num]\!]$(free) = `movl $num, r(free)`

$E_r[\![var]\!]$(free) = `movl var, r(free)`

$E_r[\![e_1\&\&e_2]\!]$(free) = $E_r[\![e_1]\!]$(free)
```
        cmpl   $1, r(free)
        jnz    lab1
```
$E_r[\![e_2]\!]$(free)
```
lab1:   ...
```

$E_r[\![var=e]\!]$(free) = $E_r[\![e]\!]$(free)
```
        movl   r(free), var
```


Compiling statements

- The rules for compiling statements need to be modified too...
- But, with our current approach, intermediate values are used only while evaluating an expression, so we can assume that the "register stack" is empty at the start of each statement

$S_r[\![e;]\!]$ = $E_r[\![e]\!](0)$

$S_r[\![$while (e) s$]\!]$ =
```
        jmp    lab2
lab1:   Sr[[s]]
lab2:   Er[[e]](0)
        cmpl   $1, r(0)
        jz     lab1
```

etc...

Back to our example

$S_r[\![$`total = total + count * count`$]\!]$

= $E_r[\![$`total = total + count * count`$]\!](0)$

Back to our example

$S_r[\![$`total = total + count * count`$]\!]$

= $E_r[\![$`total + count * count`$]\!](0)$
`movl r(0), total`

Back to our example

S_r⟦total = total + count * count⟧

```
= Er⟦total⟧(0)
  Er⟦count * count⟧(1)
  addl   r(1), r(0)
  movl   r(0), total
```


Back to our example

S_r⟦total = total + count * count⟧

```
= movl   total, r(0)
  Er⟦count * count⟧(1)
  addl   r(1), r(0)
  movl   r(0), total
```


Back to our example

S_r⟦total = total + count * count⟧

```
= movl   total, r(0)
  Er⟦count⟧(1)
  Er⟦count⟧(2)
  imull  r(2), r(1)
  addl   r(1), r(0)
  movl   r(0), total
```


Back to our example

S_r⟦total = total + count * count⟧

```
= movl   total, r(0)
  movl   count, r(1)
  Er⟦count⟧(2)
  imull  r(2), r(1)
  addl   r(1), r(0)
  movl   r(0), total
```


Back to our example

S_r⟦total = total + count * count⟧

```
= movl   total, r(0)
  movl   count, r(1)
  movl   count, r(2)
  imull  r(2), r(1)
  addl   r(1), r(0)
  movl   r(0), total
```


Back to our example

S_r⟦total = total + count * count⟧

```
= movl   total, %eax
  movl   count, %edi
  movl   count, %esi
  imull  %esi, %edi
  addl   %edi, %eax
  movl   %eax, total
```

- Now we're making better use of registers: any code to implement this operation would need at least two loads, one store, one add, and one multiply
- So now we're just one instruction away from optimal ...

Stack simulation

- To make this compilation scheme work, we added a compile-time parameter to tell us what the stack size will be at the corresponding point at run time
- Suppose that we also try to track the run time contents of the stack at compile time ...
- (we won't be able to do this with complete accuracy)
- We'll write a stack in the form [e_0, e_1, e_2, ..., e_n] where e_0, e_1, e_2, ..., e_n are either expressions, or else the special symbol ? that represents "unknown"

Back to our example (again!)

S_c⟦`total = total + count * count`⟧

  = E_c⟦`total = total + count * count`⟧ []

We use the name S_c:

S for "statement"

c for "cache"

Back to our example (again!)

S_c⟦`total = total + count * count`⟧

  = E_c⟦`total + count * count`⟧ []

```
movl   r(0), total
```


Back to our example (again!)

S_c⟦`total = total + count * count`⟧

  = E_c⟦`total`⟧ []

  E_c⟦`count * count`⟧ [`total`]

```
addl   r(1), r(0)
movl   r(0), total
```


Back to our example (again!)

S_c⟦`total = total + count * count`⟧

  = `movl   total, r(0)`

  E_c⟦`count * count`⟧ [`total`]

```
addl   r(1), r(0)
movl   r(0), total
```


Back to our example (again!)

S_c⟦`total = total + count * count`⟧

  = `movl   total, r(0)`

  E_c⟦`count`⟧ [`total`]

  E_c⟦`count`⟧ [`total, count`]

```
imull  r(2), r(1)
addl   r(1), r(0)
movl   r(0), total
```

Back to our example (again!)

$S_c[\![$`total = total + count * count`$]\!]$

```
= movl   total, r(0)
  movl   count, r(1)
  Ec[[count]] [total,count]
  imull  r(2), r(1)
  addl   r(1), r(0)
  movl   r(0), total
```

The stack description here shows that count has already been loaded in to register r(1) so it does not need to be reloaded from memory

$E_c[\![v]\!]\ [e_0, e_1, e_2, \ldots, e_n]$ = `movl r(i), r(n+1)`, if e_i = v

= `movl v, r(n+1)`, otherwise

Back to our example (again!)

$S_c[\![$`total = total + count * count`$]\!]$

```
= movl   total, r(0)
  movl   count, r(1)
  movl   r(2), r(1)
  imull  r(2), r(1)
  addl   r(1), r(0)
  movl   r(0), total
```

- One more instruction than we needed
- But we've eliminated the duplicated memory load
- The redundant instruction can be eliminated too by using an optimization called "copy propagation"

Keeping the "cache" accurate

- Some instructions can invalidate cache entries:
  `(total + (total = 2) + total)`
- Naively applying the previous compilation scheme would give us code like the following:

```
movl   total, r(0)
movl   $2, r(1)
movl   r(1), total
addl   r(1), r(0)
movl   r(0), r(1)
addl   r(1), r(0)
```

total has been changed so we shouldn't use r(0) as a shortcut to total after this

and, in fact, we've overwritten r(0) with another value anyway

so using the value in r(0) here for total is definitely a mistake!

Keeping the "cache" accurate

- To avoid such problems, we must track the cache more carefully:

```
movl   total, r(0)    [total]
movl   $2, r(1)       [total, 2]
movl   r(1), total    [?, total=2]
addl   r(1), r(0)     [?, total=2]
#movl  r(1), r(1)
addl   r(1), r(0)     [?]
```

the cache has two symbolic representations of the value in r(1) at this point

but the original value for total has been overwritten

- (Reminder: we use ? to represent unknown values)

Lessons to be learned

- We can determine quite a lot about the runtime behavior of a program by using symbolic data at compile time
- Beware subtle pitfalls and oversights. It's easy to make mistakes if you do this by hand/without a "safety net". This is one area where formal methods can help.
- A law of diminishing returns applies ...

Registers, in practice

- In the real world, no machine has infinitely many registers!
- x86-64 has quite a few …
- but some are used for special purposes …
- so we will need to be careful about how we use them!

rax	return result	eax	ax
rbx	callee saved 1	ebx	bx
rcx	argument 4	ecx	cx
rdx	argument 3	edx	dx
rsi	argument 2	esi	si
rdi	argument 1	edi	di
rbp	base pointer	ebp	bp
rsp	stack pointer	esp	sp
r8	argument 5	r8d	r8w
r9	argument 6	r9d	r9w
r10	caller saved 1	r10d	r10w
r11	caller saved 2	r11d	r11w
r12	callee saved 2	r12d	r12w
r13	callee saved 3	r13d	r13w
r14	callee saved 4	r14d	r14w
r15	callee saved 5	r15d	r15w

Register pressure

- Programs that require larger number of registers are sometimes described as creating high "register pressure"
- High register pressure occurs most frequently when we are compiling for machines with very few registers
- We can deal with register pressure by:
  - using registers more carefully
  - using register spilling when all else fails

Step 1: use registers carefully

S⟦total = total + count * count⟧

```
= movl   total, r(0)
  movl   count, r(1)
  imull  r(1), r(1)
  addl   r(1), r(0)
  movl   r(0), total
```

- We use register r(0) to hold the value of `total`
- But we don't actually need the value of total until after we've calculated `count*count`
- So why don't we delay the load of `total` ...

Step 1: use registers carefully

S⟦total = total + count * count⟧

```
= movl   count, r(0)
  imull  r(0), r(0)
  movl   total, r(1)
  addl   r(1), r(0)
  movl   r(0), total
```

- This hasn't changed the number of registers that we need
- But it has exposed a new opportunity for better code
- Instead of loading total into r(1), add it directly to r(0) ...

Step 1: use registers carefully

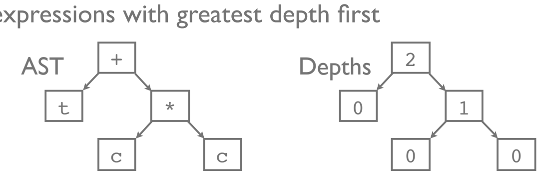
S⟦total = total + count * count⟧

```
= movl   count, r(0)
  imull  r(0), r(0)
  addl   total, r(0)
  movl   r(0), total
```

- Now we're using only one register, and fewer instructions too!
- These improvements would be particularly important if we were dealing with a subexpression of some larger statement where more registers were needed

Minimizing depth

- It can be proved that we use fewest registers if we evaluate subexpressions with greatest depth first



- If we evaluate the arguments of a subtraction in the reverse order, then we will have to use the `xchgl` instruction before the `subl`
- If we evaluate the arguments of an addition in the reverse order, then an `addl` by itself will do just fine (addition is *commutative*)

Beware of side-effects

- If the arguments of an operator (might) have side-effects, then we might not be able to change the order of evaluation
- For example, suppose that `f(x)` prints the value of `x` on the console, and then consider the expression `f(3)+(1+f(2))`
- Some language specify an explicit order of evaluation that the programmer can expect will be followed
- Others leave the choice unspecified and give the implementor more flexibility (while making the programmer's job more difficult)

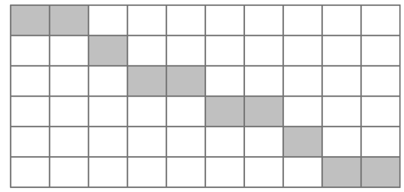
Instruction scheduling

- Finding a good execution order for the (assembly language) instructions in a program is called instruction scheduling
- What we've just seen is one example of how scheduling can make a difference
- Scheduling is perhaps more commonly associated with execution of programs on pipelined machines

Pipelining

- A pipelined machine can begin executing the next instruction before the current instruction has finished
- But, if the next instruction requires data from the current, then we have to wait until the current instruction completes
- Suppose that it takes twice as long to execute an instruction that uses memory as a register only instruction

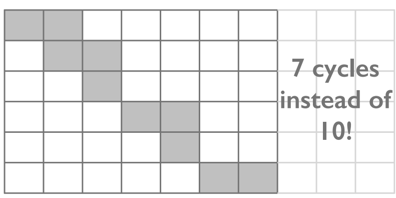
```
movl v,%eax
addl $3, %eax
movl %eax, v
movl u, %eax
subl $3, %eax
movl %eax, u
```



Pipelining (continued)

- We can reduce the time taken by scheduling the instructions in a different order ... but with the same final result

```
movl v,%eax
movl u,%ebx
addl $3,%eax
movl %eax, v
subl $3, %ebx
movl %ebx, u
```



- We may often need to change the register allocation to avoid unnecessary data dependencies
- Here we trade register pressure against scheduling; such tradeoffs are quite common in practice

Step 2: Register spilling

- The term register spilling refers to the process of moving temporary values from registers in to the stack/memory
- Suppose we limit ourselves to using only `eax` and `edi`
- We can use our finite set of registers, together with the memory stack, to simulate the infinite family r(0), r(1), r(2), ...

logical	`r(0)`	`r(1)`	`r(2)`	`r(3)`	`r(4)`	`r(5)`	`r(6)`	...
physical	`eax`	`edi`	`eax`	`edi`	`eax`	`edi`	`eax`	...

- At any given time, at most the top two r(i) values are in physical registers; the rest are saved in memory

Spilling during compilation

- In practice, we modify our compilation schemes to use spilling to reserve space when they need a "new" register
- For example:

$$\begin{aligned} E_{rs}[\![e_1+e_2]\!](\text{free}) \;=\; & E_{rs}[\![e_1]\!](\text{free}) \\ & \text{spill}(\text{free}+1) \\ & E_r[\![e_2]\!](\text{free}+1) \\ & \texttt{addl}\ \ r(\text{free}+1),\ \ r(\text{free}) \\ & \text{unspill}(\text{free}+1) \end{aligned}$$

where:

$$\begin{aligned} \text{spill}(\text{free}) &= \texttt{pushl r(free)}, && \text{if free} \geq 2 \\ &= \varepsilon, && \text{otherwise} \\ \text{unspill}(\text{free}) &= \texttt{popl r(free)}, && \text{if free} \geq 2 \\ &= \varepsilon, && \text{otherwise} \end{aligned}$$

With only two registers

$S_{rs}[\![$`total = total + count * count`$]\!]$

```
= movl   total, r(0)
  movl   count, r(1)
  pushl  r(2)
  movl   count, r(2)
  imull  r(2), r(1)
  popl   r(2)
  addl   r(1), r(0)
  movl   r(0), total
```

- If we had just two registers and didn't make other improvements, this is the code that we'd end up with
- Note that `r(2)` = `r(0)` in this setting

With only two registers

$S_{rs}[\![$`total = total + count * count`$]\!]$

```
= movl   total, %eax
  movl   count, %edi
  pushl  %eax
  movl   count, %eax
  imull  %eax, %edi
  popl   %eax
  addl   %edi, %eax
  movl   %eax, total
```

- If we had just two registers and didn't make other improvements, this is the code that we'd end up with
- Note that `r(2)` = `r(0)` in this setting

Use registers when we can; use spilling when we have to

An implementation in Java

```
public class Assembly { ...
    private String[] regs
       = new String[] { "%eax", "%edi" };
    private int numRegs = regs.length;

    /** Return the name of the physical register corresponding to a
     *  specific logical register.
     */
    public String reg(int free) {
        return regs[free % numRegs];
    }

    /** Output code to preserve the value in a register that is
     *  currently in use by pushing its value to the stack.  This
     *  allows that same register to be used temporarily to hold
     *  the value for a different logical register.
     */
    public void spill(int free) {
        if (free>=numRegs) {
            emit("pushl", reg(free));
        }
    }
    ...
}
```

maps logical register numbers to physical register names

free tells us the number of the next free register

Compiling binary expressions

```
void compileOp(Assembly a, Expr left, Expr right, int free) {

    if (left.getDepth()>right.getDepth() || right.getDepth()>=DEEP) {
        left.compileExpr(a, free);
        a.spill(free+1);
        right.compileExpr(a, free+1);

    } else {

        right.compileExpr(a, free);
        a.spill(free+1);
        left.compileExpr(a, free+1);
        a.emit("xchgl", a.reg(free+1), a.reg(free));

    }
    a.emit(op, a.reg(free+1), a.reg(free));
    a.unspill(free+1);
}
```

`DEEP` is a special (large) value assigned for expressions that have a potential side-effect

in practice, an `xchgl` is only needed if the operation is not commutative

Taking advantage of context

- As we moved to add register allocation, we got better code by adding information to our compilation schemes about the context in which the expression appears (i.e., which registers are already in use)
- Without context information, a compilation scheme must work in *all* places where that expression could appear.
- With context information, the compilation scheme only needs to produce code that will work in a specific context

Example: integer comparisons

- Code produced by E_{rs} has to work in all circumstances

$E_{rs}[\![e_1<e_2]\!](\text{free})$ = $E_{rs}[\![e_1]\!](\text{free})$
$E_{rs}[\![e_2]\!](\text{free}+1)$

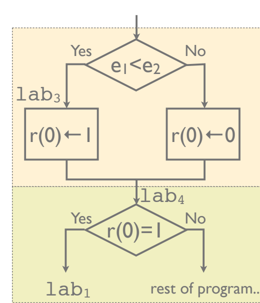
```
        cmpl  r(free+1), r(free)
        jl    lab1
        movl  $0, r(free)
        jmp   lab2
lab1:   movl  $1, r(free)
lab2:   ...
```

- In particular, this code has to produce a 0 or 1 result in r(free) because those are the only possible values for a Boolean expression

We don't always need a 0 or a 1

$S_{rs}[\![\text{while } (e_1<e_2)\ s]\!]$

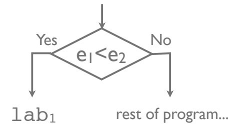
```
=          jmp    lab2
lab1:      Srs[[s]]
lab2:      Ers[[e1]](0)
           Ers[[e2]](1)
           cmpl   r(1), r(0)
           jl     lab3
           movl   $0, r(0)
           jmp    lab4
lab3:      movl   $1, r(0)
lab4:      cmpl   $1, r(0)
           jz     lab1
           ...
```



We don't always need a 0 or a 1

$S_{rs}[\![\text{while } (e_1<e_2)\ s]\!]$

```
=          jmp     lab₂
   lab₁:   Srs[[s]]
   lab₂:   Ers[[e₁]](0)
           Ers[[e₂]](1)
           cmpl  r(1), r(0)
           jl    lab₁
           ...
```



- There are times when we need a value for a Boolean expression (e.g., `boolean b = x < y;`)
- But often, we're just using a Boolean to decide whether we need to make a jump, and we don't need a 0 or 1 result.

New compilation schemes

- This motivates the introduction of two new compilation schemes for Boolean expressions:
  - *Branch to* lab *if expression* e *is true*:

    branchTrue$[\![e]\!]$(free,lab) = $E_{rs}[\![e]\!]$(free)
    `cmpl    $1,` r(free)
    `jz` lab
  - *Branch to* lab *if expression* e *is false*:

    branchFalse$[\![e]\!]$(free,lab) = $E_{rs}[\![e]\!]$(free)
    `cmpl    $1,` r(free)
    `jnz` lab
- Note that we've added a label (extra context) as an argument to the compilation scheme ...

Use the new schemes

- We can update our original compilation schemes to use branchTrue and branchFalse:

$S_{rs}[\![\text{while (e) s}]\!]$ =

```
           jmp     lab₂
   lab₁:   Srs[[s]]
   lab₂:   branchTrue[[e]](0, lab₁)
```

$S_{rs}[\![\text{if (e) } s_1 \text{ else } s_2]\!]$

```
=          branchFalse[[e]](0, lab₁)
           Srs[[s₁]]
           jmp     lab₂
   lab₁:   Srs[[s₂]]
   lab₂:
```


Override for special cases

branchTrue$[\![e_1<e_2]\!]$(free, lab) = $E_{rs}[\![e_1]\!]$(free)
$E_{rs}[\![e_2]\!]$(free+1)
`cmpl` r(free+1), r(free)
`jl` lab

ignoring spilling, to simplify presentation

branchTrue$[\![\text{true}]\!]$(free, lab) = `jmp` lab

branchTrue$[\![\text{false}]\!]$(free, lab) = `/* no code */`

branchTrue$[\![e_1||e_2]\!]$(free, lab) = branchTrue$[\![e_1]\!]$(free, lab)
branchTrue$[\![e_2]\!]$(free, lab)

branchTrue$[\![e_1\&\&e_2]\!]$(free, lab) = branchFalse$[\![e_1]\!]$(free, lab_1)
branchTrue$[\![e_2]\!]$(free, lab)
lab_1:

etc...

Benefit from the new schemes

- With the original compilation schemes, we get:

$S_{rs}[\![$`while (e₁ && e₂) s`$]\!]$

```
=          jmp     lab₂
   lab₁:   Srs[[s]]
   lab₂:   Ers[[e₁]](0)
           cmpl    $1, r(0)
           jnz     lab₃
           Ers[[e₂]](0)
           cmpl    $1, r(0)
           jz      lab₁
   lab₃:   ...
```


... continued

- With the new compilation schemes, we get:

$S_{rs}[\![$`while (e₁ && e₂) s`$]\!]$

```
=          jmp     lab₂
   lab₁:   Srs[[s]]
   lab₂:   branchFalse[[e₁]](0, lab₃)
           branchTrue[[e₂]](0, lab₁)
   lab₃:   ...
```

- If either of e_1 or e_2 fits the special cases that we have defined for branchFalse or branchTrue, then we will get better code!

Yet more compilation schemes

- The increment operator, ++, in C/C++/Java can be implemented using:

```
var++:  movl  var, %eax     ++var:  incl  var
        incl  var                   movl  var, %eax
```

- If `var++` or `++var` appears in an expression, then the final value in `eax` is important
- If either appears as a statement, then the final value is discarded and we can compile either as `incl var`
- We can use a compilation scheme to detect the special case in the compilation of statements

... and then some more

- Java/C/C++ provide two ways of escaping from a loop:

```
while (...) {          while (...) {
    ...                    ...
}                      cont:
                       }
                       past:
```

- A `break` statement in the first loop is equivalent to a "`goto past`" in the second
- A `continue` statement in the first loop is equivalent to a "`goto cont`" in the second
- How will we know where to branch when we compile a `break` or `continue` statement?

Summary

- With careful selection of compilation schemes, we can produce good quality assembly code directly from the validated AST (output from static analysis) of a program.
- Register allocation makes a big difference to performance and code size. Register spilling is used to cope with limited numbers of registers.
- Instruction scheduling can make a difference to execution time on modern machines.
- The more information that we have about the context in which a source language construct appears, the better job we can do in translating it to good quality target code.