Sources of Inefficiency in Syntax-Based Translations. Optimizations.

– lecture 6 pre-recording –

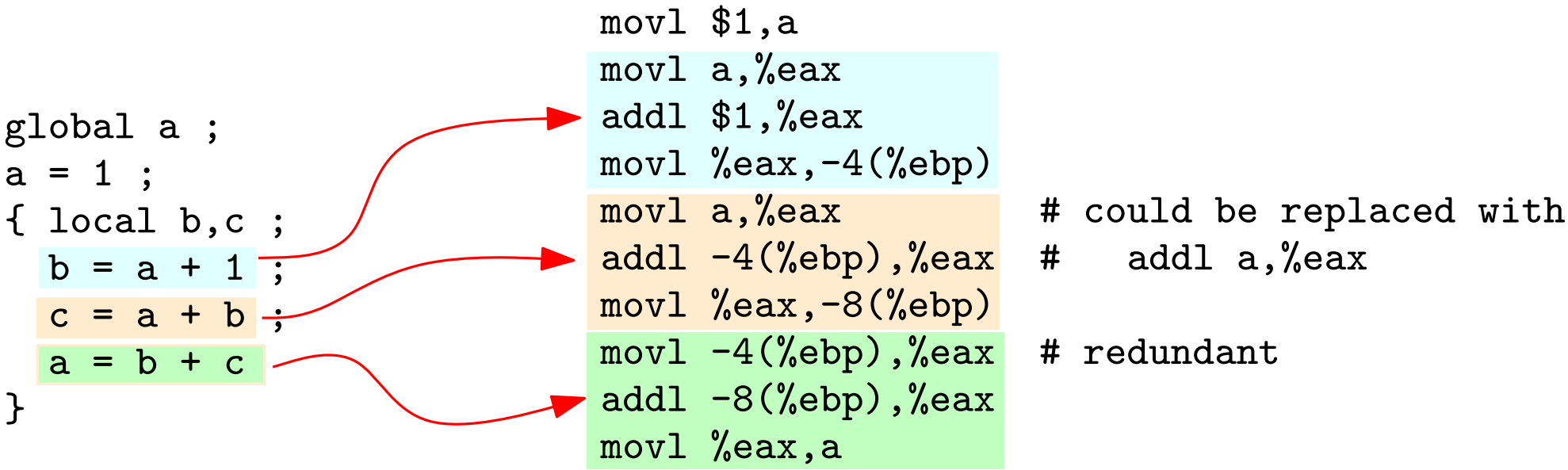
Sources of Inefficiency in Generated Code

- Inefficient code:
  - Redundant instructions (which can be removed without changing the functionality of the code)
  - Groups of instructions that can be replaced by smaller, faster, but functionally equivalent instructions.

- Appear at the boundary between code fragments generated for different components.
  - Translation scheme has to assume worst-case scenario
  - Cannot safely assume availability of data in registers; always loads/saves to memory
  - These loads/saves are often redundant

- Important to understand what kind of code will be inefficient
  - Gives us a chance to correctly detect opportunities for optimizations.

Inefficient Instruction Groups / Redundant Loads

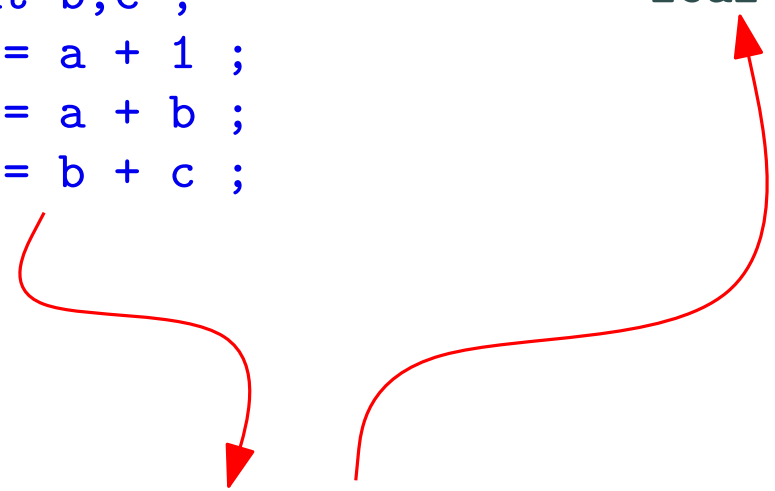


GCC's Optimization

Obtained with `gcc -O2`

```
int a ;
void f(int x) {
 a = x ;
 {      int b,c ;
        b = a + 1 ;
        c = a + b ;
        a = b + c ;
    }
}
```

```
leal 2(%eax,%eax,2), %eax
```



Equivalent to `a = 3*a+2` (local variables optimized away)

Compiler performs algebraic reconstruction, applies algebraic laws, and recompiles resulting formula

Jump Into Jump / Consec Uncond Jumps

```
global i,j ;
i = 10 ; j = 100 ;
if ( i < j ) then {
   j = j - 1
} ;
while i > 0 do {
   if i == 5
   then { i = i - 2  }
   else { j = j + 1 ;
          continue } ;
   i = i - 1
}
```

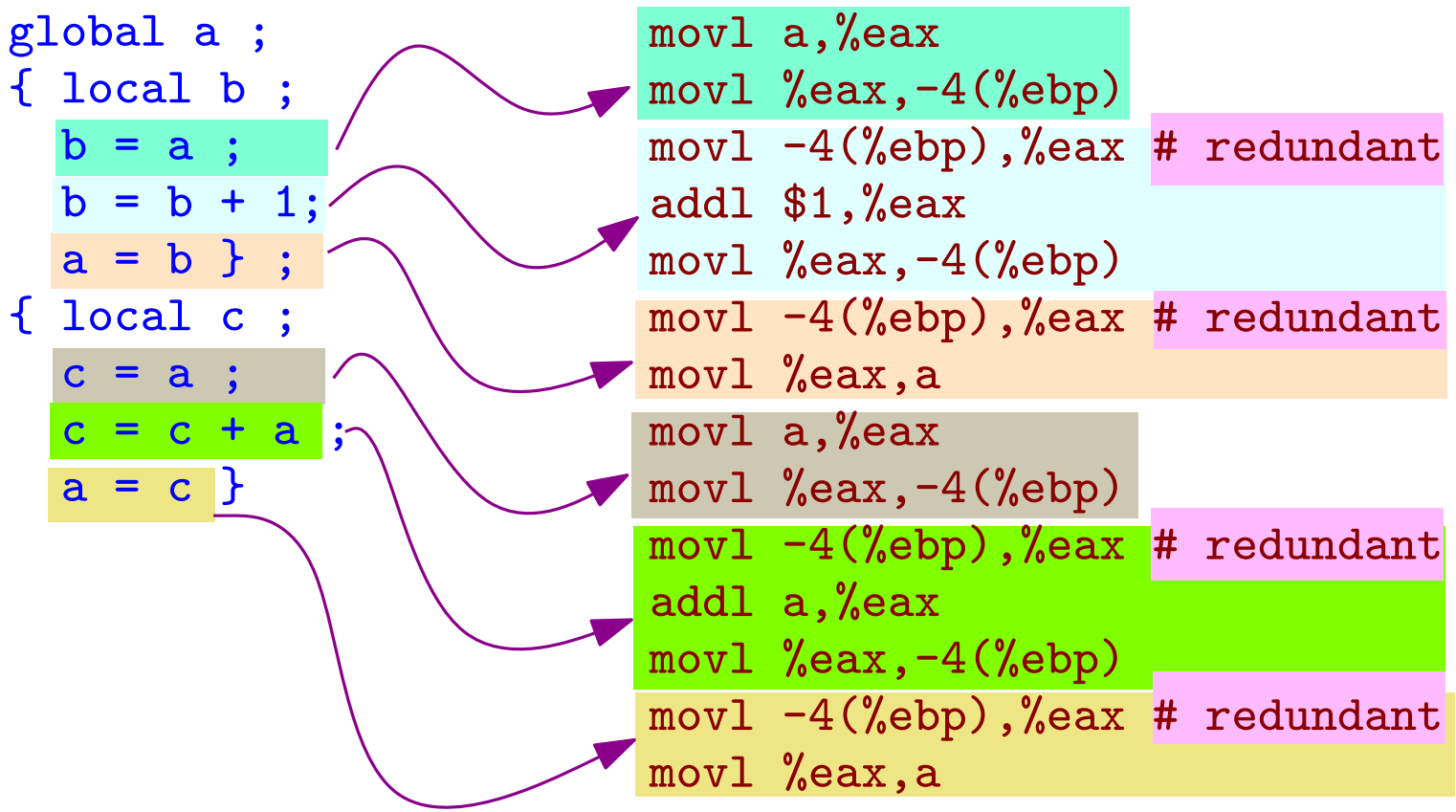
```
   movl $10,i
   movl $100,j
   movl i,%eax
   cmpl j,%eax
   jge L0        # could be replaced by jge L5
   movl j,%eax
   subl $1,%eax
   movl %eax,j
L0:
   jmp L5
L4:
   movl i,%eax
   cmpl $5,%eax
   je L2
   movl j,%eax
   addl $1,%eax
   movl %eax,j
   jmp L1
   jmp L3        # redundant, never executed
L2:
   movl i,%eax
   subl $2,%eax
   movl %eax,i
L3:
   movl i,%eax
   subl $1,%eax
   movl %eax,i
L5:
L1:
   movl i,%eax
   cmpl $0,%eax
   jg L4
```

GCC's Optimization

```
int i,j ;
void f(c10,c100) {
  i = c10 ; j = c100 ;
  if ( i < j ) {
   j = j - 1 ;
  }
  while (i > 0) {
   if (i == 5) {
    i = i - 2 ;
   } else {
    j = j + 1 ;
    continue ;
   }
   i = i - 1 ;
  }
}
```

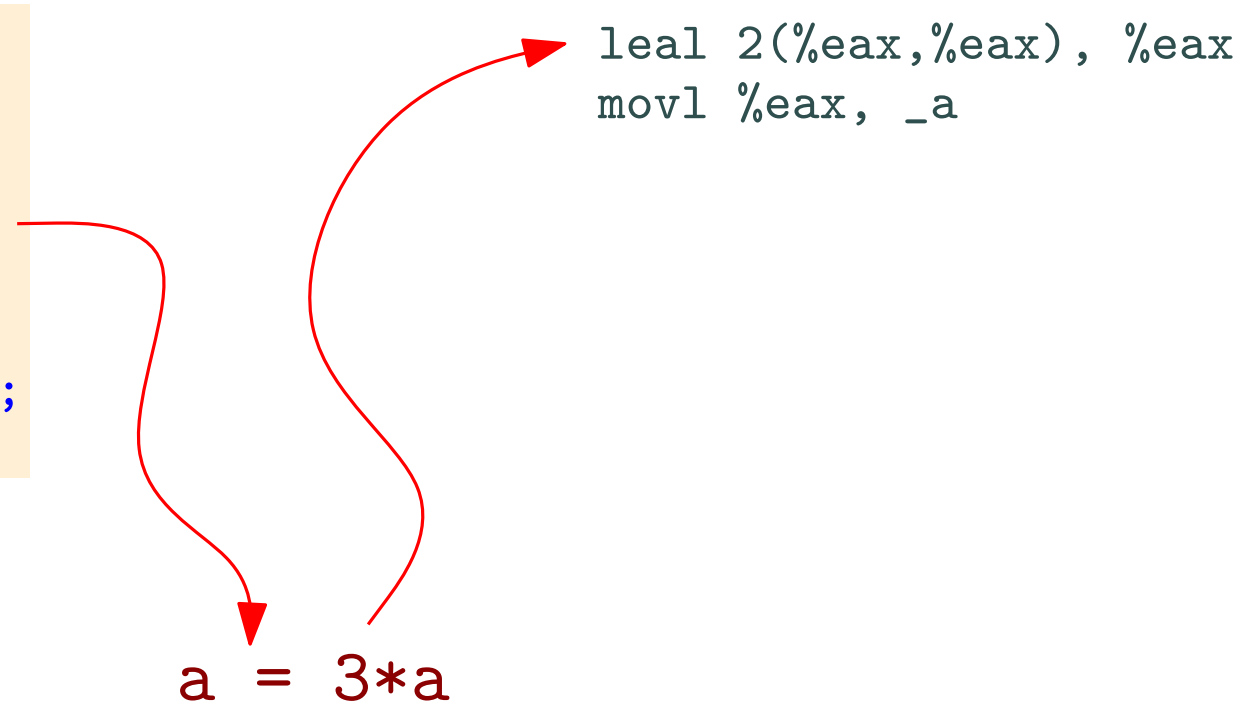
```
 movl 8(%ebp), %eax
 movl 12(%ebp), %edx
 movl %eax, _i
 cmpl %edx, %eax
 jge L2
 subl $1, %edx
L2:
 testl %eax, %eax
 movl _i, %ecx
 jle L8
L5:
 cmpl $5, %eax
 je L6
 addl $1, %edx
 movl %ecx, %eax
L9:
 testl %eax, %eax
 jg L5
L8:
 movl %ecx, _i
 movl %edx, _j
 popl %ebp
 ret
L6:
 movl $2, %ecx
 movl %ecx, %eax
 jmp L9
```

Redundant Loads



GCC's Optimization

```
int a ;
void f() {
{ int b ;
  b = a ;
  b = b + 1;
  a = b ; }
{ int c ;
  c = a ;
  c = c + a ;
  a = c ; }
}
```



Conclusion

- Syntax-based translation leads to inherent inefficiency
- In designing translation schemes for components, worst case scenarios have to be assumed.
- Optimizations:
  - Peephole: directly on generated code, looking at short groups of instructions
  - Algebraic: reconstructing general expressions from generated code, applying algebraic laws, and re-translating