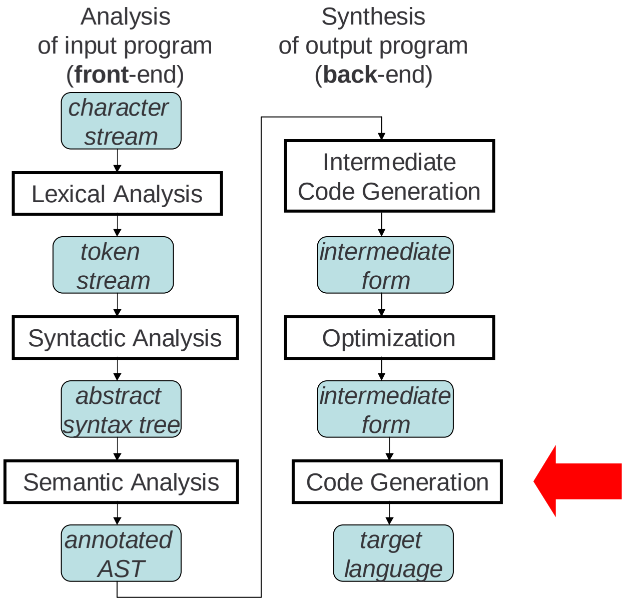
Target Code Generation

Using the generated intermediate code, covert to instructions and memory characteristics of the target machine.

Compiler Passes



Analysis
of input program
(**front**-end)

character stream

Lexical Analysis

token stream

Syntactic Analysis

abstract syntax tree

Semantic Analysis

annotated AST

Synthesis
of output program
(**back**-end)

Intermediate Code Generation

intermediate form

Optimization

intermediate form

Code Generation

target language

Target Code Generation

Input: intermediate language (IL)

Output: target language program

Target languages:

- absolute binary (machine) code
- relocatable binary code
- assembly code
- C

Target code generation must bridge the gap

The gap, if target is machine code

IL	Machine Code
global variables	global static memory
unbounded number of interchangeable local variables	fixed number of registers, of various incompatible kinds, plus unbounded number of stack locations
built-in parameter passing & result returning	calling conventions defining where arguments & results are stored and which registers may be overwritten by callee
statements	machine instructions
statements can have arbitrary subexpression trees	instructions have restricted operand addressing
conditional branches based on integers representing Boolean values	conditional branches based on condition codes (maybe)

Tasks of Code Generator

Register allocation

- for each IL variable, select register/stack location/global memory location(s) to hold it
    - can depend on type of data, which operations manipulate it

- Stack frame layout
    - compute layout of each function's stack frame
- Instruction selection
    - for each IL instruction (sequence), select target language instruction (sequence)
        - includes operand addressing mode selection
- Can have complex interactions
    - instruction selection depends on where operands are allocated
    - some IL variables may not need a register, depending on the instructions & addressing modes that are selected

Register Allocation

Intermediate language uses unlimited temporary variables

- makes ICG easy

Target machine has fixed resources for representing "locals" plus other internal things such as stack pointer

- MIPS, SPARC: 31 registers + 1 always-zero register
- 68k: 16 registers, divided into data and address regs
- x86: 8 word-sized integer registers (with a number of instruction-specific restrictions on use) plus a stack of floating-point data manipulated only indirectly

Registers are *much* faster than memory

Must use registers in load/store RISC machines

Consequences:

- should try to keep values in registers if possible
- must reuse registers, implies free registers after use
- must handle more variables than registers, implies spill
- Interacts with instruction selection on CISC, implies it's a real pain

Classes of Registers

What registers can the allocator use?

Fixed/dedicated registers

- stack pointer, frame pointer, return address, ...
- claimed by machine architecture, calling convention, or internal convention for special purpose
- not easily available for storing locals

- Scratch registers
    - couple of registers kept around for temp values e.g. loading a spilled value from memory in order to operate on it
- Allocatable registers
    - remaining registers free for register allocator to exploit
- Some registers may be overwritten by called procedures implies caller must save them across calls, if allocated
    - caller-saved registers vs. callee-saved registers

Classes of Variables

What variables can the allocator try to put in registers?

Temporary variables: easy to allocate

- defined & used exactly once, during expression evaluation implies allocator can free up register when done
- usually not too many in use at one time implies less likely to run out of registers

Local variables: hard, but doable

- need to determine **last use** of variable in order to free reg
- can easily run out of registers implies need to make decision about which variables get register allocation
- what about assignments to local through pointer?
- what about debugger?

- Global variables:
    - really hard, but doable as a research project

Register Allocation in MiniJava

Don’t do any analysis to find last use of local variables implies allocate all local variables to stack locations

- each read of the local variable translated into a load from stack
- each assignment to a local translated to a store into its stack location

Each IL expression has exactly one use implies allocate result value of IL expression to register

- maintain a set of allocated registers
- allocate an unallocated register for each expr result
- free register when done with expr result
- not too many IL expressions "active" at a time implies unlikely to run out of registers, even on x86
- MiniJava compiler dies if it runs out of registers for IL expressions :(

- X86 register allocator:
  - `eax, ebx, ecx, edx`: allocatable, caller-save registers
  - `esi, edi`: scratch registers
  - `esp`: stack pointer; `ebp`: frame pointer
  - floating-point stack, for `double` values

Stack Frame Layout

Need space for

- formals
- local variables
- return address
- (maybe) dynamic link (ptr to calling stack frame)
- (maybe) static link (ptr to lexically-enclosing stack frame)
- other run-time data (e.g. caller-saved registers)

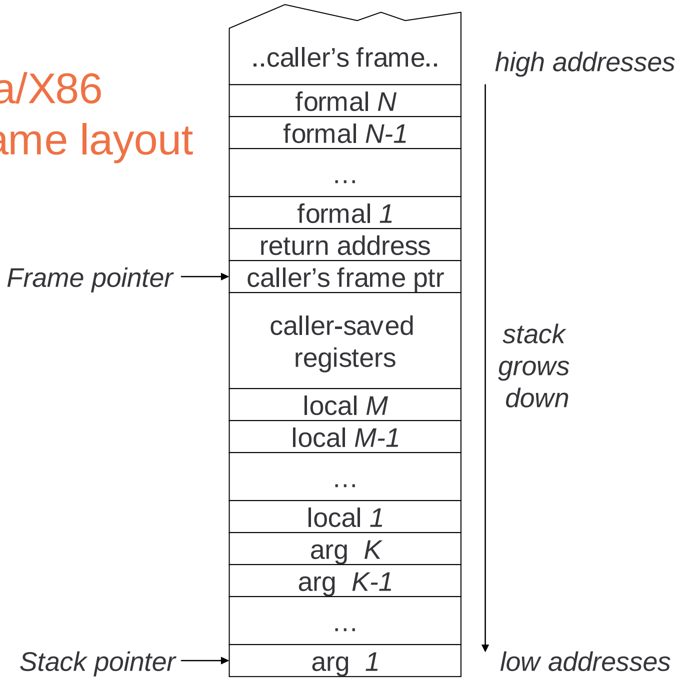
Assign dedicated register(s) to support access to stack frames

- frame pointer (FP): ptr to beginning of stack frame (fixed)
- stack pointer (SP): ptr to end of stack (can move)

Key property: all data in stack frame is **at fixed, statically** computed offset from FP

- easy to generate fast code to access data in stack frame, even lexically enclosing stack frames
- compute all offsets solely from symbol tables

MiniJava/X86 stack frame layout



..caller's frame..

formal *N*

formal *N-1*

…

formal *1*

return address

caller's frame ptr

caller-saved registers

local *M*

local *M-1*

…

local *1*

arg *K*

arg *K-1*

…

arg *1*

Calling Conventions

Need to define responsibilities of caller and callee in setting up, tearing down stack frame

Only caller can do some things

Only callee can do other things

Some things could be done by both

Need a protocol

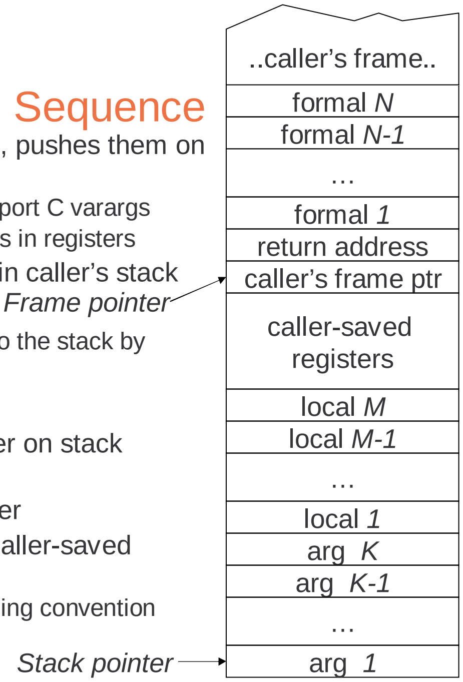
X86 Calling Sequence

Caller:

- evaluates actual arguments, pushes them on stack
  - in right-to-left order, to support C varargs
  - alternative: 1st *k* arguments in registers
- saves caller-save registers in caller's stack
- executes call instruction
  - return address pushed onto the stack by hardware

Callee:

- pushes caller's frame pointer on stack
  - the dynamic link
- sets up callee's frame pointer
- allocates space for locals, caller-saved registers
  - order doesn't matter to calling convention
- starts running callee's code


..caller's frame..

formal *N*

formal *N-1*

…

formal *1*

return address

caller's frame ptr

caller-saved registers

local *M*

local *M-1*

…

local *1*

arg *K*

arg *K-1*

…

arg *1*

Frame pointer

Stack pointer

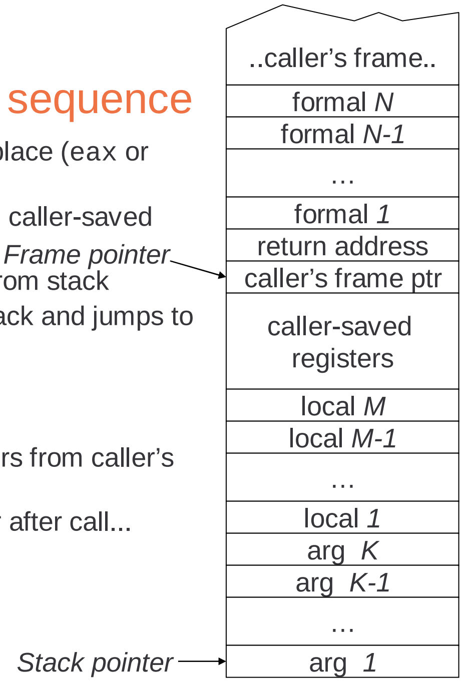
X86 return sequence

Callee:

- puts returned value in right place (eax or floating-point stack)
- deallocates space for locals, caller-saved regs
- pops caller's frame pointer from stack
- pops return address from stack and jumps to it

Caller:

- deallocates space for args
- restores caller-saved registers from caller's stack
- continues execution in caller after call...


..caller's frame..

formal *N*

formal *N-1*

…

formal *1*

return address

caller's frame ptr

caller-saved registers

local *M*

local *M-1*

…

local *1*

arg *K*

arg *K-1*

…

arg *1*

Frame pointer

Stack pointer

Instruction Selection

Given one or more IL instructions, pick "best" sequence of target machine instructions with same semantics

"best" = fastest, shortest, lowest power, ...

Difficulty depends on nature of target instruction set

- RISC: easy
  - usually only one way to do something
  - closely resembles IL instructions
- CISC: hard to do well
  - lots of alternative instructions with similar semantics
  - lots of possible operand addressing modes
  - lots of tradeoffs among speed, size
  - simple RISC-like translation may not be very efficient
- C: easy, as long as C appropriate for desired semantics
  - can leave optimizations to C compiler

Correctness a big issue, particularly if codegen complex

Example

IL code:

```
t3 = t1 + t2;
```

Target code (MIPS):

```
add $3,$1,$2
```

Target code (SPARC):

```
add %1,%2,%3
```

Target code (68k):

```
mov.l d1,d3
add.l d2,d3
```

Target code (x86):

```
movl %eax,%ecx
addl %ebx,%ecx
```

1 IL instruction may expand to several target instructions

Another Example

IL code:

```
t1 = t1 + 1;
```

Target code (MIPS):

```
add $1,$1,1
```

Target code (SPARC):

```
add %1,1,%1
```

Target code (68k):

```
add.l #1,d1 …or…
inc.l d1
```

Target code (x86):

```
addl $1,%eax …or…
incl %eax
```

Can have choices

- it’s a pain to have choices; requires making decisions

Yet another example

IL code:

```
// push x onto stack
sp = sp - 4;
*sp = t1;
```

Target code (MIPS):

```
sub $sp,$sp,4
sw $1,0($sp)
```

Target code (SPARC):

```
sub %sp,4,%sp
st %1,[%sp+0]
```

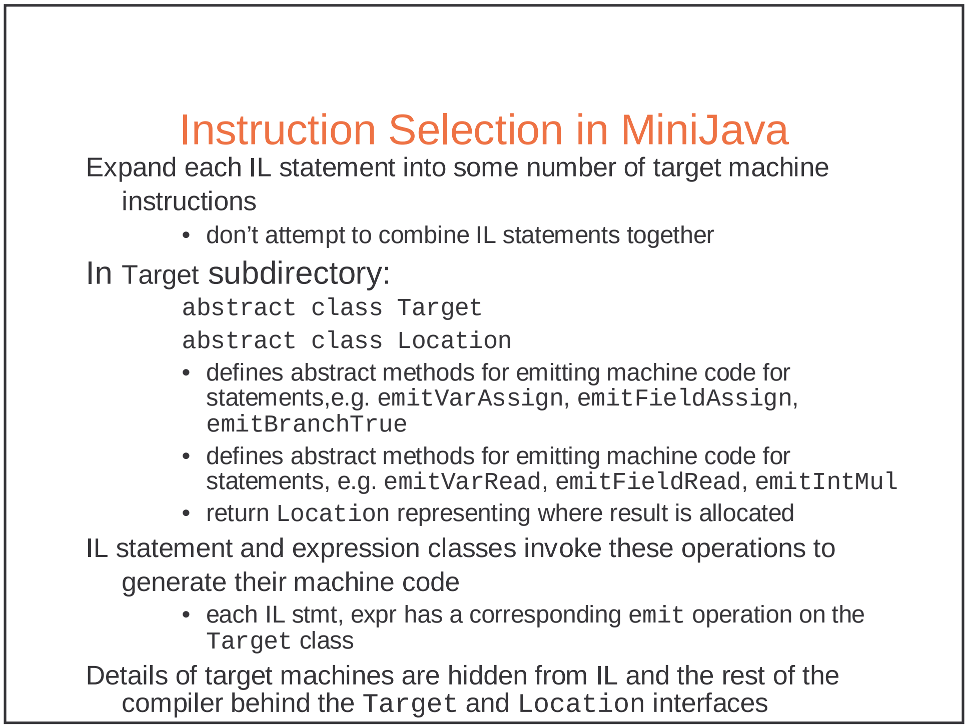
Target code (68k):

```
mov.l d1,-(sp)
```

Target code (x86):

```
pushl %eax
```

Several IL instructions can combine to 1 target instruction



Instruction Selection in MiniJava

Expand each IL statement into some number of target machine instructions

- don't attempt to combine IL statements together

In Target subdirectory:

`abstract class Target`

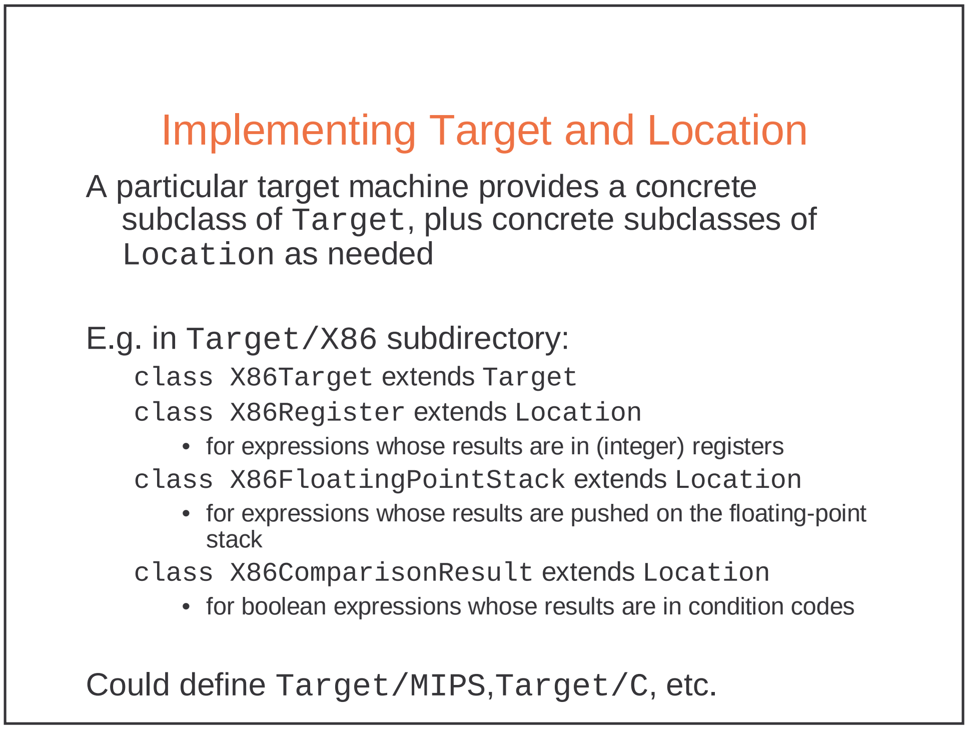
`abstract class Location`

- defines abstract methods for emitting machine code for statements,e.g. `emitVarAssign`, `emitFieldAssign`, `emitBranchTrue`
- defines abstract methods for emitting machine code for statements, e.g. `emitVarRead`, `emitFieldRead`, `emitIntMul`
- return `Location` representing where result is allocated

IL statement and expression classes invoke these operations to generate their machine code

- each IL stmt, expr has a corresponding `emit` operation on the `Target` class

Details of target machines are hidden from IL and the rest of the compiler behind the `Target` and `Location` interfaces



Implementing Target and Location

A particular target machine provides a concrete subclass of `Target`, plus concrete subclasses of `Location` as needed

E.g. in `Target/X86` subdirectory:

`class X86Target` extends `Target`

`class X86Register` extends `Location`

- for expressions whose results are in (integer) registers

`class X86FloatingPointStack` extends `Location`

- for expressions whose results are pushed on the floating-point stack

`class X86ComparisonResult` extends `Location`

- for boolean expressions whose results are in condition codes

Could define `Target/MIPS`,`Target/C`, etc.

An Example X86 emit method

```
Location emitIntConstant(int value) {
   Location result_location =
      allocateReg(ILType.intILType());
      emitOp("movl",
        intOperand(value),
        regOperand(result_location));
   return result_location;
}
```

`Location allocateReg(ILType):`

allocate a new register to hold a value of the given type

`void emitOp(String opname, String arg1, ...):`

emit assembly code

`String intOperand(int):`

return the asm syntax for an int constant operand

`String regOperand(Location):`

return the asm syntax for a reference to a register

An Example X86 Target emit method

What x86 code to generate for `arg1 +.int arg2`?

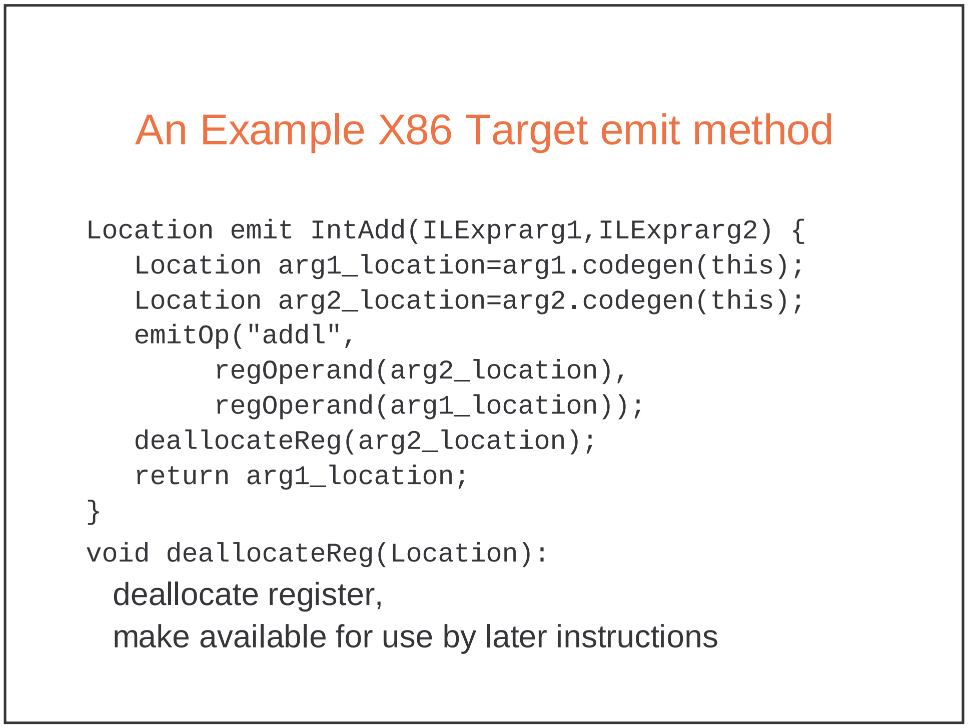
x86 int add instruction: `addl %arg, %dest`

– semantics: `%dest = %dest + %arg;`

emit *`arg1`* into register`%arg1`

emit *`arg2`* into register`%arg2`

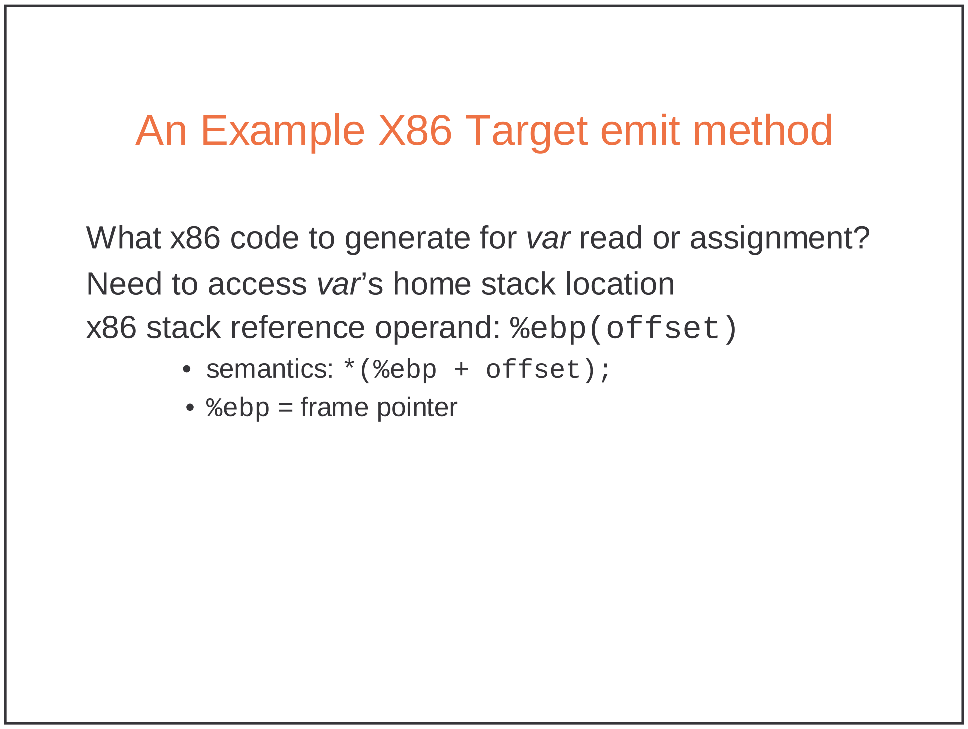
then?



An Example X86 Target emit method

```
Location emit IntAdd(ILExprarg1,ILExprarg2) {
   Location arg1_location=arg1.codegen(this);
   Location arg2_location=arg2.codegen(this);
   emitOp("addl",
        regOperand(arg2_location),
        regOperand(arg1_location));
   deallocateReg(arg2_location);
   return arg1_location;
}
void deallocateReg(Location):
```

deallocate register,
make available for use by later instructions



An Example X86 Target emit method

What x86 code to generate for *var* read or assignment?

Need to access *var*’s home stack location

x86 stack reference operand: `%ebp(offset)`

- semantics: `*(%ebp + offset);`
- `%ebp` = frame pointer

An Example X86 Target emit method

```
Location emitVarRead(ILVarDecl var) {
  int var_offset = var.getByteOffset(this);
  ILType var_type = var.getType();
  Location result_location =
      allocateReg(var_type);
  emitOp("movl",
         ptrOffsetOperand(FP, var_offset),
         regOperand(result_location));
  return result_location;
}
void emitVarAssign(ILVarDecl var,
                   Location rhs_location) {
  int var_offset = var.getByteOffset(this);
  emitOp("movl",
         regOperand(rhs_location),
         ptrOffsetOperand(FP, var_offset));
}
String ptrOffsetOperand(Location, int):
```

return the asm syntax for a reference to a "ptr + offset" memory location

An Example X86 Target emit method

```
void emitAssign(ILAssignableExpr lhs,
                ILExpr rhs) {
  Location rhs_location =
      rhs.codegen(this);
  lhs.codegenAssign(rhs_location, this);
  deallocateReg(rhs_location);
}
```

Each `ILAssignableExpr` implements `codegenAssign`

- invokes appropriate `emitAssign` operation, e.g. `emitVarAssign`

Target Code Generation for Comparisons

What code to generate for `arg1` <.int `arg2`?

- produce zero or non-zero int value into some result register

MIPS: use an `slt` instruction to compute boolean-valued int result into a register

x86 (and most other machines): no direct instruction

Have comparison instructions, which set condition codes

- e.g. `cmpl %arg2, %arg1`

Later conditional branch instructions can test condition codes

e.g. `jl, jle, jge, jg, je, jne label`

What code to generate?

Target Code Generation for Compares (1)

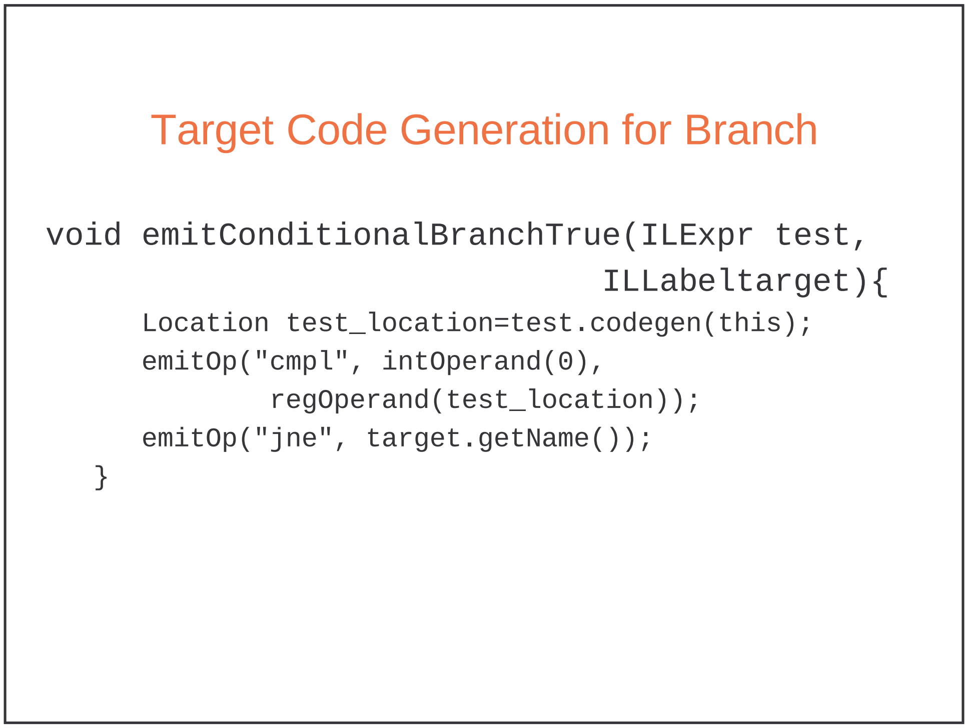
```
Location emitIntLessThanValue(ILExpr arg1,
                              ILExpr arg2) {
   Location arg1_location=arg1.codegen(this);
   Location arg2_location=arg2.codegen(this);
   emitOp("cmpl",
           regOperand(arg2_location),
           regOperand(arg1_location));
   deallocateReg(arg1_location);
   deallocateReg(arg2_location);
   Location result_location =
     allocateReg(ILType.intILType());
```

Target Code Generation for Compars (2)

```
   String true_label = getNewLabel();
   emitOp("jl", true_label);
   emitOp("movl", intOperand(0),
           regOperand(result_location));
   String done_label = getNewLabel();
   emitOp("jmp", done_label);
   emitLabel(true_label);
   emitOp("movl", intOperand(1),
           regOperand(result_location));
   emitLabel(done_label);
   return result_location;
}
```

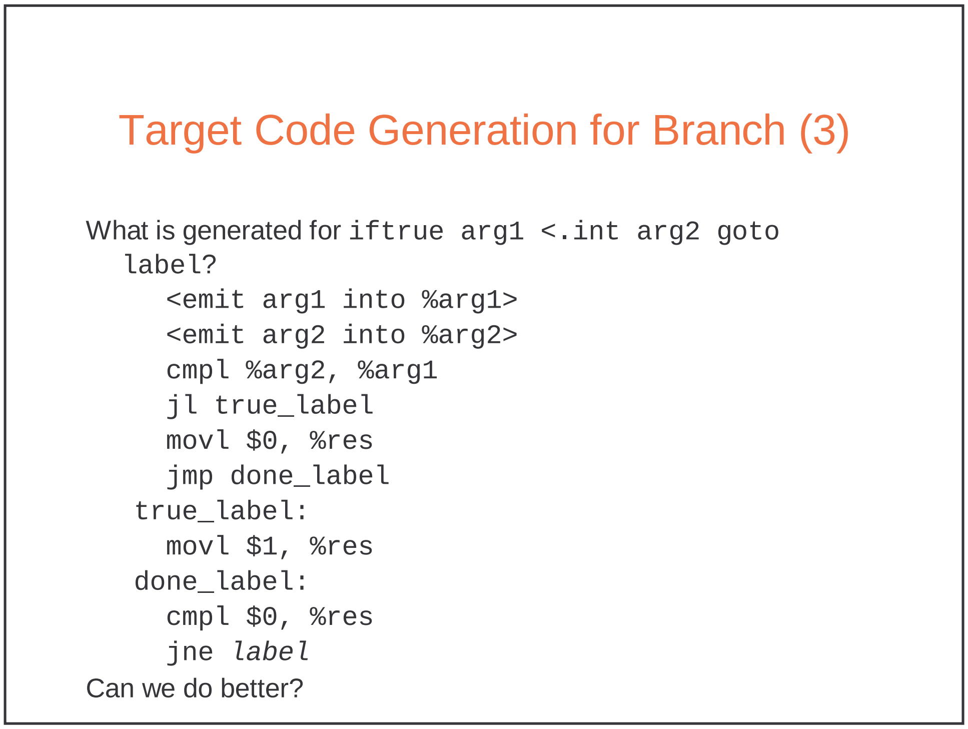
Target Code Generation for Branch

What code to generate for `iftrue test goto label`?



Target Code Generation for Branch

```
void emitConditionalBranchTrue(ILExpr test,
                             ILLabeltarget){
      Location test_location=test.codegen(this);
      emitOp("cmpl", intOperand(0),
              regOperand(test_location));
      emitOp("jne", target.getName());
   }
```



Target Code Generation for Branch (3)

What is generated for `iftrue arg1 <.int arg2 goto label`?

```
     <emit arg1 into %arg1>
     <emit arg2 into %arg2>
     cmpl %arg2, %arg1
     jl true_label
     movl $0, %res
     jmp done_label
   true_label:
     movl $1, %res
   done_label:
     cmpl $0, %res
     jne label
```

Can we do better?

Optimized Code Gen for Branches(1)

Idea: boolean-valued IL expressions can be generated two ways, depending on their consuming context

- for their value
- for their "condition code"

Existing code gen operation on IL expression produces its value

New codegenTest operation on IL expression produces its condition code

- X86ComparisonResultLocation represents this result

Now conditional branches evaluate their test expression in the "for condition code" style

Optimized Code Gen for Branches (2)

```
void emitConditionalBranchTrue(ILExpr test,
                               ILLabeltarget){
   Location test_location=test.codegen(this);
   X86ComparisonResultLoc cc =
       (X86ComparisonResultLoc) test_location;
   emitOp("j" + cc.branchTrueOp(),
   target.getName());
}
```

IL codegenTest Default Behavior

```
class ILExpr extends ILExpr {
   ...
   Location codegenTest(Target target) {
      return target.emitTest(this);
   }
}
In X86Target class:
  Location emitTest(ILExpr arg) {
    Location arg_location = arg.codegen(this);
    emitOp("cmpl", intOperand(0),
           regOperand(arg_location));
    deallocateReg(arg_location);
    return new X86ComparisonResultLoc("ne");
  }
```

IL codegenTest Specialized Behavior

```
class ILIntLessThanExpr extends ILExpr {
  ...
  Location codegenTest(Target target) {
     return target.emitIntLessThanTest(arg1, arg2);
  }
}
In X86Target class:
  Location emitIntLessThanTest(ILExpr arg1,
                               ILExpr arg2) {
    Location arg1_location=arg1.codegen(this);
    Location arg2_location=arg2.codegen(this);
    emitOp("cmpl",
           regOperand(arg2_location),
           regOperand(arg1_location));
    deallocateReg(arg1_location);
    deallocateReg(arg2_location);
    return new X86ComparisonResultLoc("l");
}
```

Register Allocation -- A Cool Algorithm

- How to convert the infinite sequence of temporary data references, t1, t2, … into finite assignment register numbers $8, $9, …, $25
- Goal: Use available registers with minimum spilling
- Problem: Minimizing the number of registers is NP-complete … it is equivalent to chromatic number--minimum colors to color nodes of graph so no edge connects same color

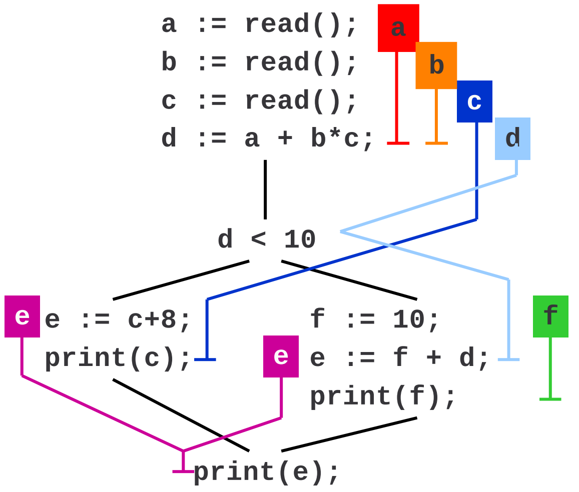
Begin With Data Flow Graph

- procedure-wide register allocation
- only live variables require register storage

dataflow analysis: a variable is live at node N if *the value* it holds is used on some path further down the control-flow graph; otherwise it is dead

- two variables(values) interfere when their live ranges overlap

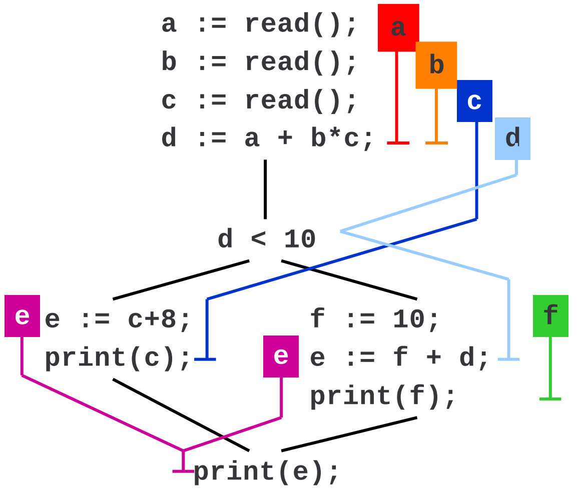
Live Variable Analysis



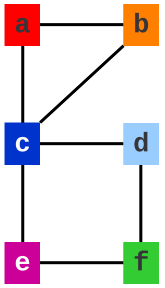
```
a := read();
b := read();
c := read();
d := a + b*c;

d < 10

e := c+8;          f := 10;
print(c);          e := f + d;
                   print(f);

print(e);
```

```
a := read();
b := read();
c := read();
d := a + b*c;
if (d < 10 ) then
   e := c+8;
   print(c);
else
   f := 10;
   e := f + d;
   print(f);
fi
print(e);
```

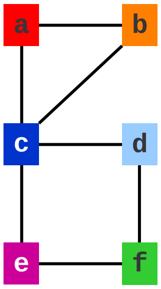
Register Interference Graph



```
a := read();
b := read();
c := read();
d := a + b*c;

d < 10

e := c+8;          f := 10;
print(c);          e := f + d;
                   print(f);

print(e);
```

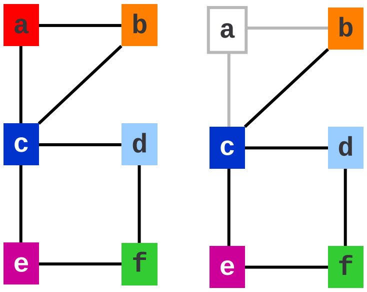


Graph Coloring

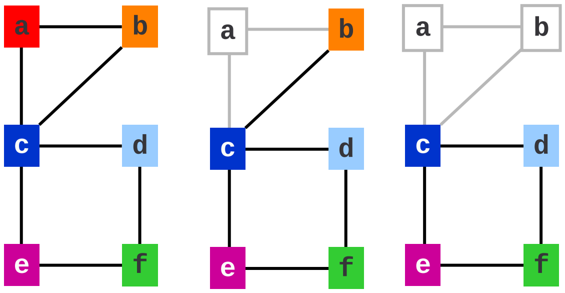


- NP complete problem
- heuristic: color easy nodes last
  - find node *N* with lowest degree
  - remove *N* from the graph
  - color the simplified graph
  - set color of *N* to the first color that is not used by any of *N*’s neighbors

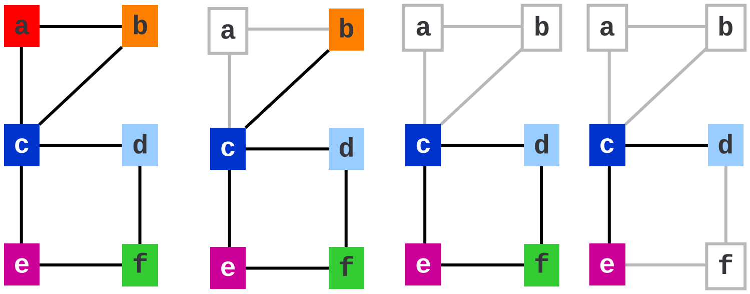
Apply Heuristic



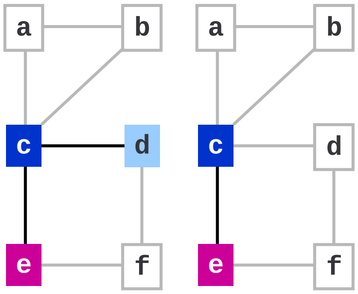
Apply Heuristic



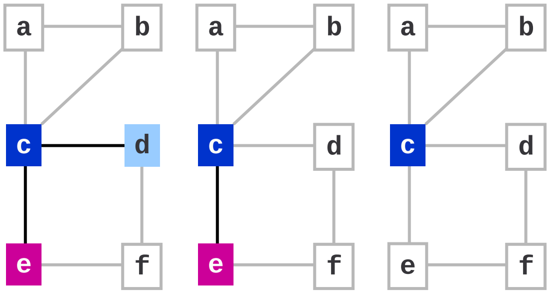
Apply Heuristic



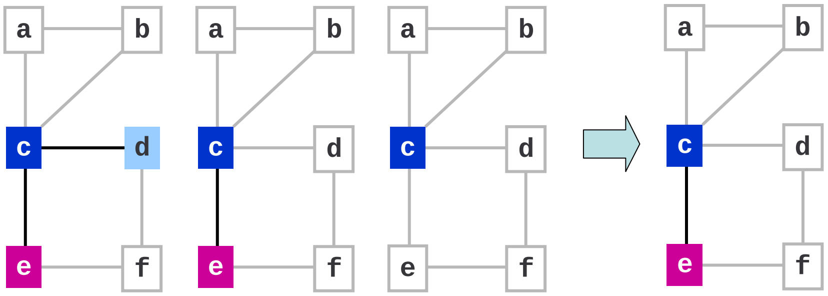
Continued



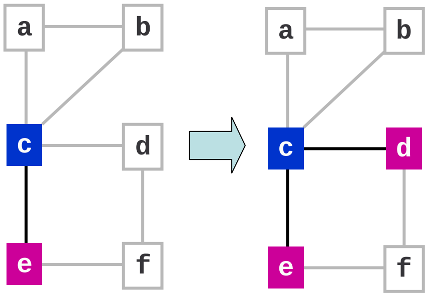
Continued



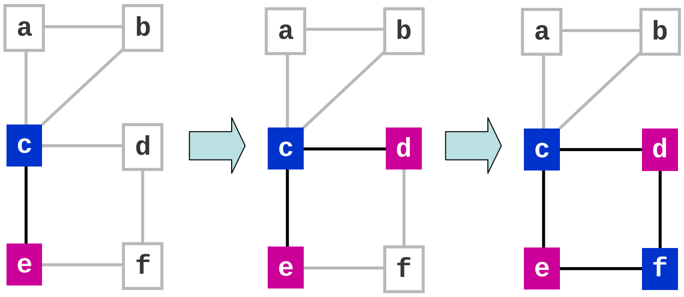
Continued



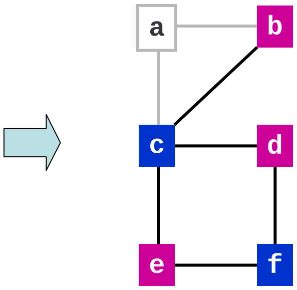
Continued



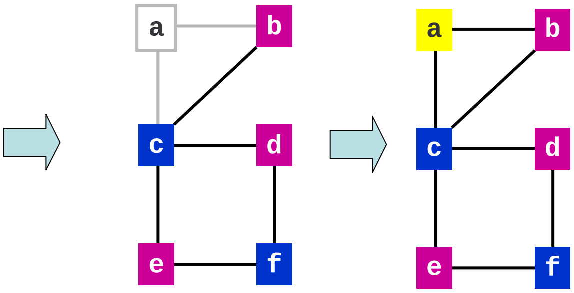
Continued



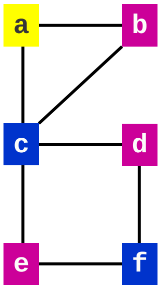
Continued



Continued



Final Assignment



```
a := read();
b := read();
c := read();
d := a + b*c;
if (d < 10 ) then
   e := c+8;
   print(c);
else
   f := 10;
   e := f + d;
   print(f);
fi
print(e);
```

Example

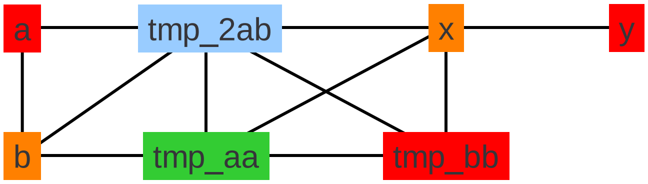
```
{  int tmp_2ab = 2*a*b;
   int tmp_aa = a*a;
   int tmp_bb = b*b;

   x := tmp_aa + tmp_2ab + tmp_bb;
   y := tmp_aa - tmp_2ab + tmp_bb;
}
```

given that a and b are live on entry and dead on exit,
and that x and y are live on exit:

(a) construct the register interference graph

(b) color the graph; how many registers are needed?

4 Registers Needed



Code Generation Summary

- Code generation is
  - Machine specific
  - Error prone
  - Least “elegant” of the compilation process
- Code generation is
  - Place where key transformation takes place in the compiler
  - Most visible impact on performance