**LEC 11 Compiler Optimization**

* Constant folding – don’t need to wait until runtime to compute constants
  + Ex: 11 + 2
  + Following all the rules:

lis $3

.word 11

sw $3, -4($30)

sub $30, $30, $4

lis $3

.word 2

lw $5, 0($30)

add $30, $30, $4

add $3, $5, $3

* + Using constant folding:

lis $3

.word 13

* Constant propagation
  + Ex:

int x = 1;

x + x;

* + Using constant propagation:

push x to stack ; x might still be used elsewhere

lis $3 ; load constant value instead of calculating x + x at runtime

.word 2

* + If x is not used elsewhere, don’t need to push x to stack
* Common sub-expression elimination
  + Ex:

int x = 1;

int \*y = NULL;

y = &x;

\*y = 2;

…

x + x ; recognize that $3 already contains x

; don’t need to lw it again

* + Ex:

(a + b) \* (a + b) ; use register to hold a + b, then just multiply it by itself

* + Watch out for side effects
    - E.g. f() + f()
    - The second call may have different result than the first call
* Dead code elimination
  + Don’t generate code for a branch of the program that can be shown to never run
* Register allocation
  + It is cheaper to use registers for variables instead of the stack
    - In real world – registers are much faster
    - In this class – using registers saves lw & sw
  + E.g. registers $12 – 28 are unused by our code generation scheme
  + Use these for variables before using the stack
  + Choose the 17 most-used variables
  + Can also use these registers to store temporaries in evaluating expressions
  + Can determine if the use of two variables is disjoint in the program
    - i.e. if the first time y is used is after the last time x is used, they can share the same register
  + Issue: the & operator assumes the variable is in memory
    - If the variable is stored in a register it has no address
    - Can “fake” the address when this is encountered
* Strength reduction
  + In real world – add runs faster than mult
  + In this class – add uses less lines than mult
  + For multiplications by 2 – use add instead
  + Ex:

lis $2 vs. add $3, $3, $3

.word 2

mult $3, $2

mflo $3

* **Procedure-specific optimization**
* Inlining
  + Ex:

int f(int x) { x + x; }

int wain(int a, int b) { return f(a); }

* + vs.

int wain(int a, int b) { return a + a; }

* + Replace function call with the function’s body within the caller
  + Saves the overhead of a function call
  + If all calls to f are inlined, the procedure f doesn’t need to be generated
  + Downsides
    - If f is called many times, many copies of its body will be generated
    - The body of f may be longer than the code to call f
    - Recursive functions cannot be inlined
* Tail recursion
  + Ex: a tail recursive function in C

int fact(int n, int a) {

if (n == 0) return a;

else return fact(n – 1, n \* a);

}

* + Note that the recursive call is the last thing this procedure does
    - i.e. there is nothing left to do (except return itself) when the recursive call returns
    - i.e. contents of the current stack frame will not be used again
    - ∴ the current frame can be reused for recursive calls
    - ∴ no new stack space is being used – the recursion can run infinitely – can be compiled as a loop instead of procedure
  + This can’t be done in WLP4 because only one return statement is allowed at the end of a procedure
  + Recursive call is no longer the last thing done by the procedure
  + Transformation into tail recursion:
    - If returns immediately follows if-else, push it inside both branches
    - Ex:

int f() {

int ret = 0;

if (…) {

if (…) { … }

else { … }

}

else { … }

return ret;

}

* Becomes →

int f() {

int ret = 0;

if (…) {

if (…) {

…

return ret;

}

else {

…

return ret;

}

}

else {

…

return ret;

}

}

* + - If return ret immediately follows an assignment to ret, merge the two

ret = f(…);

return ret;

* + - Becomes → return f(…);
* Generalization – tail call optimization
  + When a function’s last action is a function call (recursive or not) – reuse the stack frame
* Experiment – method overloading
  + i.e.

int f(int a) { … }

int f(int a, int \*b) { … }

* + How to deal with duplicate labels?
  + Name mangling – encode the types of params (signature) as part of the label
    - Example convention: F + types + “\_” + name

int f() { … } F\_f:

int f(int a) { … } Fi\_f:

int f(int a, int \*b) { … } Fip\_f:

* + - C++ compilers do this
    - There is no standard convention
      * This makes linking code from different compilers hard/impossible since labels may be named differently
      * This is by design because compilers differ in other aspects of calling conventions
      * Prevent successful linking if conventions don’t match
  + C has no overloading & name mangling – but how do C & C++ code call each other?
    - Mangling in C++ must be suppressed
    - Calling C from C++:

extern “C” int f(int n);

* + - * “C” specifies a linkage convention – in this case, specifies to look for an unmangled name when linking
    - Calling C++ from C:

extern “C” { // can also extern multiple functions like this

int g(int \*p) { … }

…

}

* + - * Specifies to compile g with unmangled name so it can be called by C