Reading Assignment

```
Fischer & LeBlanc

      Chapter  15

      Omit  15.7,  15.8
```


Code Generation

- Map IR (e.g. quadruples) to some real machine code.
  Assume input is semantically valid, all type conversions are explicit, control flow graph is valid.
- Allocate storage. For each scope, place all variables in the appropriate activation record. Assign an offset in the activation record to each variable.
- Can use record layout algorithms to layout activation records efficiently.
  Use indirection to deal with data objects that have dynamic sizes.
  Store literal constants in the code or in a constant area not in an activation record.
- Simple (unoptimized) case, make one pass over the IR transforming it to machine code. Need a mechanism for keeping track of addresses in the emitted code and for patching branches.

Code Generation Issues

- Code generation design is usually done on a per language construct basis or on a per IR quadruple basis.
- Instruction Selection
  - Map each quadruple to machine instructions
  - Optimizations: special cases, order of computation, context sensitive
  - Interacts with register allocation
- Register Allocation
  - Some registers will be dedicated to specific purposes like the display, routine call and return.
  - Need a mechanism for allocating available registers for use in expression evaluation and addressing.
  - Optimizations: minimize registers used, minimize stores of temporaries to memory.
- **You must know the target machine intimately before you can do a good job on code generation.**

Code Generation Mechanisms

- Need a mechanism for describing the location of operands and the location of results, something like the Fischer & LeBlanc Data Object.
- Need a mechanism for describing and manipulating target machine addresses.
- Need a mechanism for allocating registers and freeing them when they are no longer needed.
- Need a mechanism for managing temporary storage in memory, i.e. in the activation record of the current routine.
- Need mechanisms for managing the IR input and for managing the target machine code that is being produced.
- Need mechanisms for dealing with target machine branching instructions including patching of forward branches.

Table Driven - Code Generation Templates

- For each possible quadruple in the IR, design a template describing target machines instructions to implement the quadruple.
- Templates will have substitutable parameters for operands and results.
- Template expansion mechanism uses conditional selection to deal with possible operand/result locations and possibly operand type dependent code.
- Use conditional selection to generate special case code for local optimizations.
- Instruction selection can involve deep decision trees, e.g. large IBM mainframes have at least 17 instructions which perform addition, choice is based on operand types and operand locations.
- The simple versions of the approach are unable to use the *context* of the quadruple to optimize code selection, e.g. J = J + 1

Template Example

- Quadruple: (add, leftOperand, rightOperand, result)
- leftOperand and rightOperand could be
  - Literal Constants
  - Values in registers (result of previous quadruple)
  - Program variables in memory.
  - Temporary locations in memory.
  - The same.
  - Of different base types, e.g. short, integer, long, unsigned
- result could be
  - Temporary register
  - Dedicated register (e.g. function return value)
  - Temporary location in memory
  - Program variable in memory
  - The same as leftOperand or rightOperand

Template Example - (add , $left$, $right$, $result$)

$left$	$right$		
	literal	register	memory
literal	$result =$ $left + right$	LIT $tmp_{reg}, = left$ ADDR $right_{reg}, tmp_{reg}$ $result = right_{reg}$	LIT $tmp_{reg}, left$ ADD $tmp_{reg}, right$ $result = tmp_{reg}$
register	LIT $tmp_{reg}, right$ ADDR $left_{reg}, tmp_{reg}$ $result = left_{reg}$	ADDR $left_{reg}, right_{reg}$ $result = left_{reg}$	ADD $left_{reg}, right$ $result = left_{reg}$
memory	LIT $tmp_{reg}, right$ ADD $tmp_{reg}, left$ $result = tmp_{reg}$	ADD $right_{reg}, left$ $result = right_{reg}$	LOAD $tmp_{reg}, left$ ADD $tmp_{reg}, right$ $result = tmp_{reg}$

- Assumes registers can be overwritten, addition is commutative.
  Assumes result can be set in template (i.e. no targetting).

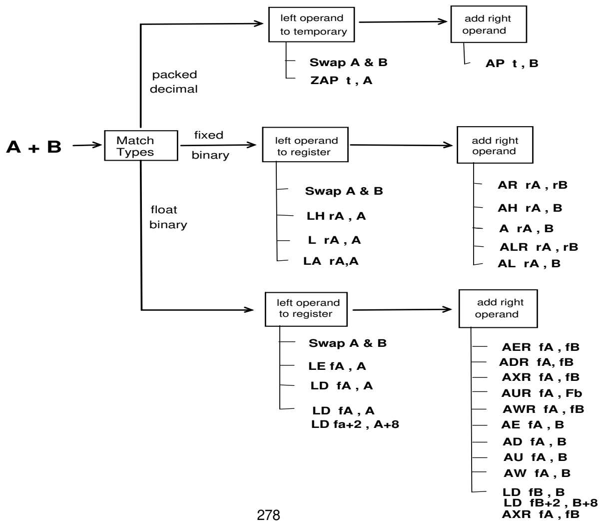
Code Generation Optimizations

- Defer operand evaluation, e.g constant folding.
- Use *targeting* to encourage results to end up in desirable locations (e.g. registers).
- Process expressions depth first as a heuristic to minimize register and temporary usage.
  - Constrained by operator precedence.
  - Constrained by language rules for expression evaluation order.
- Use special instructons (e.g. literal instructions) wherever possible for faster code. Examples for (add, left, right, result)
  - If left or right is literal zero, do nothing.
  - If left or right is literal one use INCR instruction
  - If left or right is small literal use LITADD instruction.
  - If left and right are the same and one is in a register, use ADDR instruction, e.g. ADDR right,right.

Implementing Code Selection

- For complicated (CISC) target machines, instruction selection during code generation can involve very complicated decision processes. Example in Slide 275 is very simplified.
- There are many systems for automating the code selection process
  - Code Generator Builders
  - Glanville-Graham code generation scheme based on parsing the IR to determine the appropriate template.
  - Chaitins register allocation scheme based on graph colouring.
  - Peephole optimization (discussed later) for local optimization on the fly.
  - Cattell's code generation scheme based on tree rewriting.
- RISC machines pose a different set of problems, instruction selection is often trivial, but optimal instruction ordering is a big issue.

Example - IBM 370 Code Generation for A + B



Data Descriptors

- A data structure invented by Prof. R.C. Holt[a] that provides a convenient way to represent and manipulate addresses during code generation. Similar to the Fischer/LeBlanc Data Object.
- Can be used to automate code generation for addressing including array subscripting and record field references.
- A Data Descriptor $*^k b.d$ has the interpretation $*^k(value(b) + d)$ i.e. add the displacement d to the contents of register b and apply k indirections to reach the data item that is the subject of the data descriptor.
- Examples

0.d	subject is literal constant d
b.0	subject is register b
*b.d	sum of register b and displacement d is address of subject
**b.0	register b contains address of pointer to the subject.

[a] R.C. Holt, Data Descriptors: a compile-time model of data and addressing, ACM Trans. on Programming Languages and Systems, v.9, n.3, July 1987, pp. 367-389

Data Descriptor Manipulation

- Holt developed a calculus of data descriptors that describes most address manipulations. It is used for integer add and for addition in addressing calculations.
- Optimized Add on the following slide is an example of data descriptor manipulation
  - It combines two data descriptors and produces a data descriptor describing its result.
  - It emits address calculation code only when required.
  - It takes advantage of existing temporary registers (t bases) whene ever it can.
  - It doesn't modify permanent base registers (b bases).
  - It takes advantage of constant information.
- The invariant for Optimal Add is that the result descriptor describes the same subject as the combination of left and right operand data descriptors.

Optimal Addition of Data Descriptors

Left Operand	Right Operand: 0	d_r	$t_r.d_r$	$b_r.d_r$	$^*t_r.d_r$	$^*b_r.d_r$
0	0	d_r	$t_r.d_r$	$b_r.d_r$	$^*t_r.d_r$	$^*b_r.d_r$
d_l	d_l	$d_l + d_r$	$t_r.(d_l + d_r)$	$b_r.(d_l + d_r)$	$t_r = *(t_r.d_r)$ $t_r.d_l$	$t = {}^*(b_r.d_r)$ $t.d_l$
$t_l.d_l$	$t_l.d_l$	$t_l.(d_l + d_r)$	$t_l += t_r$ $t_l.(d_l + d_r)$	$t_l += b_r$ $t_l.(d_l + d_r)$	$t_l += *(t_r.d_r)$ $t_l.d_l$	$t_l += *(b_r.d_r)$ $t_l.d_l$
$b_l.d_l$	$b_l.d_l$	$b_l.(d_l + d_r)$	$t_r += b_l$ $t_r.(d_l + d_r)$	$t = b_r$ $t += b_l$ $t.(d_l + d_r)$	$t_r = *(t_r.d_r)$ $t_r += b_l$ $t_r.d_l$	$t = *(b_r.d_r)$ $t += b_l$ $t.d_l$
$*t_l.d_l$	$*t_l.d_l$	$t_l = *(t_l.d_l)$ $t_l.d_r$	$t_r += *(t_l.d_l)$ $t_r.d_r$	$t_l = *(t_l.d_l)$ $t_l += b_r$ $t_l.d_r$	$t_l = *(t_l.d_l)$ $t_l += *(t_r.d_r)$ $t_l.0$	$t_l = *(t_l.d_l)$ $t_l += *(b_r.d_r)$ $t_l.0$
$*b_l.d_l$	$*b_l.d_l$	$t = *(b_l.d_l)$ $t.d_r$	$t_r += *(b_l.d_l)$ $t_r.d_r$	$t = *(b_l.d_l)$ $t += b_r$ $t.d_r$	$t_r = *(t_r.d_r)$ $t_r += *(b_l.d_l)$ $t_r.0$	$t = *(b_r.d_r)$ $t += *(b_l.d_l)$ $t.0$

Notes: The last line in each box is the result data descriptor.

t is a newly allocated temporary register.

$=$ and $+=$ indicate LOAD and ADD (ADDR) instructions emitted.

Reading Assignment

Fischer & LeBlanc

Chapter 16

Optimization

- Goals for Optimization
  - Make the object program smaller and/or faster **without changing their meaning.**
  - To achieve object code as good as that produced by the best assembly language programmers. (Optimizing compilers do better than assembly language programmers on modern multi-issue RISC machines).
  - To produce object code that is *optimal*, i.e. minimum execution time.
- Why optimization is needed
  - To improve on inefficient implementation of programming language constructs.
  - To mitigate the mismatch between high level programming language constructs and low level machine instructions.
  - To compensate for sloppy programming practices.
  - To improve programs in ways that can't be expressed in high level programming languages.
- Optimization **is not** a substitute for good algorithm selection.

Optimization Interactions

- Optimization interaction with Programming Languages
  - Non compact data structures, e.g. subarrays.
  - Exception handling.
  - Overly specific language standards.
  - Symbolic debugging.
  - Aliasing of variables
  - Concurrency.
- Optimization interaction with Hardware
  - Cache memory.
  - Virtual memory.
  - Register windows.
  - Instruction pipelining.
  - Super scalar, multi-issue instruction processors.
  - Multiple functional units, e.g. 2 FADD units.

When is Optimization Worthwhile ?

- Most expressions in typical programs are very simple.
  For example the average expression in Fortran contains less than 1 operator.
- Big optimization gains come from optimizing loops and subscript calculation.[a]
- Generating locally good code to begin with always wins.
- Most interesting optimization problems are NP-complete.
  There are useful heuristics in many cases.
- Optimizations are usually divided into two categories
  - Machine Independent Optimizations
  - Machine Dependent Optimizations
- Historically, optimization is notorious for introducing bugs into a compiler.

[a] Review the discussion of array subscript polynomials in Slides 162 .. 165

Optimizations

- Machine Independent Optimizations
  - Constant Folding
  - Common Subexpression Elimination
  - Algebraic Simplification
  - Code Motion (forward/backward)
  - Strength Reduction and Test Replacement
  - Dead Code Elimination
  - Loop Unrolling, Loop Fusion
  - Procedure/Function Integration
- Machine Dependent Optimizations
  - Register Allocation, allocating variables in registers
  - Use of hardware idioms
  - Peephole optimization
  - Instruction scheduling
  - Cache-sensitive code generation

Optimization Inhibitors

The programming artifacts listed below affect the ability of the compiler to perform optimization. If these effects cannot be ruled out then the compiler must optimize very pessimistically.

- *Side Effects* Any construct that changes the value of a variable has a side effect on that variable. An optimizer needs to know when the value of a variable might change in order to optimize the use of variables.
- *Aliasing* An alias exists when two identifiers refer to the same storage location. An optimizer needs to know the effect of every assignment statement. An alias causes an unexpected side effect.
- *Exception Handler* An exception handling mechanism can be invoked asynchronously during the execution of a program and can in general cause side effects on variables. The occurrence of exception-caused side effects is extremely difficult to predict.

Constant Folding

- If an expression involves only constants or other values known to the compiler, calculate the value of the expression at compile time.
- Issues
  - Compiler must contain a calculator for expressions.
    For cross compiler must calculate in the arithmetic of the target machine.
  - May want to extend to common builtin functions, e.g. max, sqrt.
  - Beware of arithmetic faults during constant expression evaluation, e.g. overflow, underflow, divide by zero.
  - May need to reorder expressions to facilitate constant folding, e.g. 3 + A + 4
- Example[a]

#define aSize 100	
int A[aSize] ;	
A[aSize-1] = aSize ∗ aSize ;	@ (&A[0] + 4∗ (100-1)) = 100 ∗ 100
	@ (&A[0] + 396) = 10000

[a] We use the operator @ as a C-style pointer dereferencing operator in these examples. Variables AP, BP, etc. are byte pointer variables. Most data items are 4 bytes wide.

Common Subexpression Elimination

- If the same expression is calculated more than once, save the value of the first calculation and use it place of the repeated calculations.
- Issues
  - Detection of common subexpressions. Use heuristics.
  - Must be able to detect when the values of the variables in an expression could be changed. The presence of aliasing and exception handling make this more difficult.
- Example

A[J] = A[J] + 1 ;	@ (&A[0] + 4 ∗ J) = @ (&A[0] + 4 ∗ J) + 1
A[J + 1] = A[J] ;	@ (&A[0] + 4 ∗ J + 4) = @ (&A[0] + 4 ∗ J)
	AP = (&A[0] + 4 ∗ J)
	@ AP = @ AP + 1
	@ (AP+4) = @ AP

Algebraic Simplification

- Use algebraic identities to simplify or reorder expressions. Use commutativity and associativity of operators. Often used to facilitate constant folding and common subexpression elimination.
- Issues
  - Must be careful not to change the meaning of the program.
- Examples

X + 0 $\rightarrow$ X	X - 0 $\rightarrow$ X	Y ∗ 1 $\rightarrow$ Y
Y / 1 $\rightarrow$ Y	X ∗ 0 $\rightarrow$ 0	-X + -Y $\rightarrow$ - (X + Y)
P **and** true $\rightarrow$ true	P **and** false $\rightarrow$ false	
P **or** true $\rightarrow$ true	P **or** false $\rightarrow$ P	

Code Motion

- Move code (statements, expressions or fragments thereof) out of loops (forward or backward) to places where they are executed less frequently.
- Code that is moved must be *loop invariant*, i.e. independent of the loop indices.
- Issues
  - Correctness (safety) of code motion.
  - Impacted by side effects, aliasing and exception handling.
  - Must preserve semantics of loops that never execute.
- Some optimizers do code motion by copy insertion followed by common subexpression elimination.

Code Motion Example

```
int J, K ;
float A[100][100], B[100] ;
for( J = 0 ; J < 100 ; J++ )
    for( K = 0 ; K < 100 ; K++ )
        A[J][K] = B[J] ;
```

```
for( J = 0 ; J < 100 ; J++ )
    for( K = 0 ; K < 100 ; K++ )
        @ (&A[0][0] + 400 ∗ J + 4 ∗ K)
            = @ (&B[0] + 4 ∗ J)
```

```
for( J = 0 ; J < 100 ; J++ ) {
    AP = &A[0][0] + 400 ∗ J
    BV = @ (&B[0] + 4 ∗ J)
    for( K = 0 ; K < 100 ; K++ )
        @ (AP + 4 ∗ K) = BV
}
```

Strength Reduction and Test Replacement

- Strength reduction is the replacement of "slow" operations (e.g. multiply and divide) with "faster" operations (e.g. add and subtract)
- Test Replacement occurs when (perhaps due to other optimizations) the body of a loop no longer contains any use of the loop index variable. Replace the loop index variable and the loop termination test.
- Issues
  - Deciding which operations are sufficiently "slow" as to warrant this optimization. Strength reduction was more important when multiply and divide were *really* slow relative to add and subtract.
  - Need to be careful not to change the meaning of the program.
- Strength reduction is often used to eliminate multiplications in array subscript polynomials. Often a precursor to machine dependent optimizations.

Strength Reduction Example

```
int J, K ;
float A[100][100], B[100] ;
for( J = 0 ; J < 100 ; J++ )
    for( K = 0 ; K < 100 ; K++ )
        A[J][K] = B[J] ;
```

```
for( J = 0 ; J < 100 ; J++ )
    for( K = 0 ; K < 100 ; K++ )
        @ (&A[0][0] + 400 * J + 4 * K)
            = @ (&B[0] + 4 * J)
```

```
for( J4 = 0 , J400 = 0 , J = 0 ; J < 100 ;
                J++ , J4 += 4, J400 += 400 ) {
    AP = &A[0][0] + J400
    BV = @ (&B[0] + J4)
    for( K = 0 , K4 = 4; K < 100
                ; K++, K4 += 4 )
        @ (AP + K4) = BV
}
```


Test Replacement Example

```
int J, K ;
float A[100][100], B[100] ;
for( J = 0 ; J < 100 ; J++ )
    for( K = 0 ; K < 100 ; K++ )
        A[J][K] = B[J] ;
```

```
for( J = 0 ; J < 100 ; J++ )
    for( K = 0 ; K < 100 ; K++ )
        @ (&A[0][0] + 400 * J + 4 * K)
            = @ (&B[0] + 4 * J)
```

```
for( J4 = 0 , J400 = 0 ; J4 < 400 ;
                J4 += 4, J400 += 400 ) {
    AP = &A[0][0] + J400
    BV = @ (&B[0] + J4)
    for( K4 = 0 ; K4 < 400 ; K4 += 4 )
        @ (AP + K4) = BV
}
J = 100
K = 100
```


Dead Code and Useless Computation Elimination

- Compiler detects and eliminates code that can never be executed (dead code) and computations of values that are never subsequently used (useless computations).
- Issues
  - Detection of dead code requires analysis of control flow graph.
  - Useless calculations may result from programmer error or from other optimizations. Test replacement is a form of useless calculation elimination.
- Examples

```
if( 1 )
    X = Y ;
else
    X = Z ;
return ;
X = Y ;
```

```
X = Y ;

return ;
```

Variable Folding

- If a variable is assigned a "computationally simple" value, replace subsequent uses of the variable by the value. Creates new opportunities for constant folding and common subexpression elimination.
- Issues: aliasing, side effects.
- Examples

J = 13 ;	
A[J] = B[J] ∗ C[J] ;	A[13] = B[13] ∗ C[13] ;
K = J + 1 ;	
A[K] = B[K] ;	A[J + 1] = B[J + 1] ;
A[∗] = B[∗] ; /* array assignment*/	
Z = SUM(A[∗]) ;	Z = SUM(B[∗]) ;

Loop Unrolling

- Replicate the body of a loop and adjust loop index. Reduces loop overhead relative to loop body. Enables common subexpression and constant folding optimizations.
- Issues: detecting unrollable loops, aliasing, side effects
  loop carried dependencies, handling loop remainder.
- Example

```
for( J = 0 ; j < 200 ; J++ )
    A[ J ] = B[ J ] * C[ J ] ;
```

```
for( J = 0 ; J < 200 ; J + = 4 ) {
    A[ J ] = B[ J ] * C[ J ] ;
    A[ J + 1 ] = B[ J + 1 ] * C[ J + 1 ] ;
    A[ J + 2 ] = B[ J + 2 ] * C[ J + 2 ] ;
    A[ J + 3 ] = B[ J + 3 ] * C[ J + 3 ] ;
}
```


Loop Fusion (Jamming)

- Combine and simplify two or more loops that share a common index or range. Leads to common subexpression optimizations.
- Issues: side effects, aliasing, detection of fusable loops.
- Example

```
for( J = 0 ; J < 100 ; J++ )
    for( K = 0 ; K < 100 ; K++ )
        A[ J ][ K ] = 0 ;
for( J = 0 ; J < 100 ; J++ )
    A[ J ][ J ] = 1 ;
```

```
for( J = 0 ; J < 100 ; J++ ) {
    for( K = 0 ; K < 100 ; K++ ) {
        A[ J ][ K ] = 0 ;
    }
    A[ J ][ J ] = 1 ;
}
```


Routine Inlining

- Replace the call of a procedure or function with an inline expansion of the routine body with actual parameters substituted for the formal parameters.
- Eliminates routine call and return overhead.
  Exposes code in the expanded body to further optimizations.
- Issues
  - Aliasing and side effects. Recursive procedures.
  - Extreme care must be taken to preserve the semantics of parameter passing.
  - Often only done on leaf routines (i.e. routines that don't call any other routines.
    Often candidate routines are ones with no (or very little) local storage.
  - Must systematically rename any variables local to the routine to avoid accidental synonyms.
- This is an important optimization for OO languages where Objects export a lot of very small Object access routines (e.g. inline functions in C++).
  For example, a stack object exporting Push, Pop, Top.

Routine Inlining Example

```
int max( int A , int B ) {
    if( A > B )
        return A ;
    else
        return B ;
}
...
X = max( X , 100 ) ;
```

```
if( X > 100 )
    X = X ;
else
    X = 100 ;
```


Tail Recursion Elimination

- If the body of a recursive routine ends with a recursive call to the routine, replace the call with a setting of parameters followed by a branch to the start of the routine body.
- Saves routine call and return overhead.
  Saves activation record allocation/deallocation overhead *and space*.
- Issues: Needs extreme care to preserve parameter passing semantics.
  Often done where parameters are passed by value.
- This is an especially important optimization in functional languages, e.g. Lisp, Scheme, ML

Tail Recursion Elimination Example

```
int Func( int A , int B ) {

    ...

    return Func( A - 1 , B + 1 );
}
```

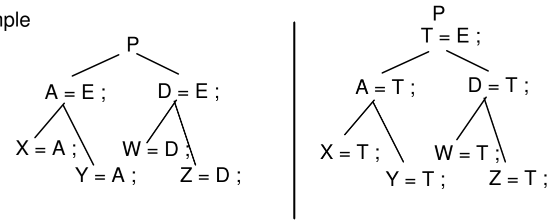
```
int Func( int A , int B ) {
    top:

    ...

    A = A - 1 ;
    B = B + 1 ;
    goto top ;
}
```


Code Hoisting

- If some expression is calculated on all paths leading *from* some point P in the programs control flow graph *and* the expression satisfies the conditions for common subexpression elimination then code for the expression can be moved (hoisted) to point P.
- Issues
  - Safety, aliasing, side effects.
  - Computationally expensive to detect hoisting opportunities.
  - Most useful if code is hoisted out of loops.
- Example



Machine Dependent Optimization

- Recognize specific programming language constructs that can be more efficiently implemented using specialized machine instructions
- Issues
  - Must maintain programming language semantics.
  - Be careful about interactions with other instructions, i.e. setting of condition codes.
- Examples

I = I + 1 ;	INC I	PDP-11, VAX
J = 0 ;	CLR J	PDP-11
A[$*$] = B[$*$] ;	MVCL A,B	System/370
for(I = 1 ; I <= 100 ; I++)	BXLE	System/370
X $*$ 2^n	SLA rX,n	System/370
C = 'A' ;	MVI 'A',C	System/370

Matrix Multiplication Example
Initial Program

- Matrix multiplication: $C_{mn} = A_{mp} \times B_{pn}$

$$c_{i,j} = \sum_{k=1}^{p} a_{i,k} \star b_{k,j}$$

- Naive program - Fortran

```
     INTEGER I, J, K
     FLOAT A[ 20 , 10 ] , B[ 10 , 20 ] , C[ 20 , 20 ]
     ...
     DO 300 I = 1 TO 20
        DO 300 J = 1 TO 20
           C[ I , J ] = 0.0
           DO 300 K = 1 TO 10
              C[ I , J ] = C[ I , J ] + A[ I , K ] ∗ B[ K , J ]
300     CONTINUE
```


Matrix Multiplication Example
Make Subscripts Explicit[a]

```
          @ (&C[1 , 1] + 4 ∗ (20 ∗ (I - 1) + (J - 1) ) ) = 0.0
          DO 300 K = 1 TO 10
             @ (&C[1 , 1] + 4 ∗ (20 ∗ (I-1) + (J-1) )) =
                @ (&C[1 , 1] + 4 ∗ (20 ∗ (I-1) + (J-1) ) ) +
                   @ (&A[1 , 1] + 4 ∗ (10 ∗ (I-1) + (K-1) ))
                      ∗ @ (&B[1 , 1] + 4 ∗ (20 ∗ (K-1) +(J-1) ) )
300       CONTINUE
```

[a] &X[1 , 1] is the base address of array X.
FORTRAN arrays are 1-origin and FLOATS are 4 bytes wide.

Matrix Multiplication Example
Fold and Propagate Constants

```
     @ (&C[1 , 1] + 80 ∗ I + 4 ∗ J - 84 ) ) = 0.0
     DO 300 K = 1 TO 10
        @ (&C[1 , 1] + 80 ∗ I + 4 ∗ J - 84 ) = @ (&C[1 , 1] + 80 ∗ I + 4 ∗ J - 84
) +
        @ (&A[1 , 1] + 40 ∗ I + 4 ∗ K - 44) ∗ @ (&B[1 , 1] + 80 ∗ K + 4 ∗ J - 84 )
300     CONTINUE
```

Matrix Multiplication Example
Eliminate Common Subexpression

```
      J484 = 4 * J - 84
      @ (&C[1 , 1] + 80 * I + J484 ) ) = 0.0
      DO 300 K = 1 TO 10
         @ (&C[1 , 1] + 80 * I + J484 ) = @ (&C[1 , 1] + 80 * I + J484 ) +
            @ (&A[1 , 1] + 40 * I + 4 * K - 44) * @ (&B[1 , 1] + 80 * K + J484 )
300      CONTINUE
```


Matrix Multiplication Example
Move Loop Independent Code Out of Loop

```
      J484 = 4 * J - 84
      AP = &A[ 1 , 1 ] + 40 * I - 44
      BP = &B[ 1 , 1 ] + J484
      CP = &C[ 1 , 1 ] + 80 * I + J484
      @ CP = 0.0
      DO 300 K = 1 TO 10
         @ CP = @ CP + @ (AP + 4 * K ) * @ (BP + 80 * K )
300      CONTINUE
```


Matrix Multiplication Example
Apply Strength Reduction

```
      J484 = 4 * J - 84
      AP = &A[ 1 , 1 ] + 40 * I - 44
      BP = &B[ 1 , 1 ] + J484
      CP = &C[ 1 , 1 ] + 80 * I + J484
      @ CP = 0.0
      K4 = 4
      K80 = 80
      DO 300 K = 1 TO 10
         @ CP = @ CP + @ (AP + K4 ) * @ (BP + K80 )
         K4 = K4 + 4
         K80 = K80 + 80
300      CONTINUE
```


Matrix Multiplication Example
Replace Test

```
      J484 = 4 * J - 84
      AP = &A[ 1 , 1 ] + 40 * I - 44
      BP = &B[ 1 , 1 ] + J484
      CP = &C[ 1 , 1 ] + 80 * I + J484
      @ CP = 0.0
      K80 = 80
      DO 300 K4 = 4 TO 40 BY 4
         @ CP = @ CP + @ (AP + K4 ) * @ (BP + K80 )
         K80 = K80 + 80
300      CONTINUE
```

Matrix Multiplication Example
Machine Dependent Optimizations - System/370

- Accumulate sum in a register
  Use register-register instructions, loop variable in registers.
  Use BXLE for loop control, base+index addressing.
- Assume register assignments

R1 ← AP	R8 ← 4	R2 ← BP	R9 ← 40
R3 ← CP	R10 ← 80	R6 ← K80	F0 ← sum
F0 ← sum	F2 ← product		

- Use LM instruction to initialize general registers.
- Similar optimizations can be performed on the outer loops.

Matrix Multiplication Example

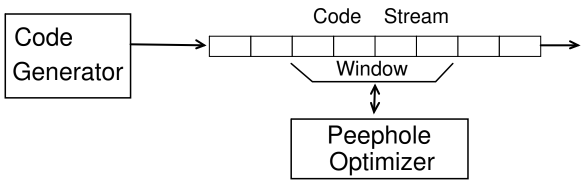
- System/370 Assembly Code for Loop

	SDR	F0,F0	F0 ← 0.0
KLOOP	LE	F2,0(R1,R7)	F2 ← @ (AP+K4)
	ME	F2,0(R2,R6)	F2 ← F2 ∗ @ (BP+K80)
	AER	F0,F2	F0 ← F0 + F2
	AR	R6,R10	K80 ← K80 + 80
	BXLE	R7,R8,KLOOP	K4 ← K4 + 4 ; if(K40 <= 40) goto KLOOP
	STE	F0,0(,R3)	@ CP ← F0

- Optimized loop is 7 instructions, 22 bytes. 5 instruction inner loop.
  Unoptimized loop is approximately 37 instructions, 146 bytes.
- PL/I and Fortran optimizing compilers for the IBM System/370 achieve this level of optimization starting from the naive program.

Peephole Optimization

- Peephole optimization is a technique for making local improvements to the code generated by a simple code generator.
- Peephole optimizer examine the code stream as it is generated.
  Uses a 2 .. 4 instruction window to search for particular instruction sequences that it can optimize. Usually table/template driven.
- Issues: Peephole optimizer must be aware of control flow.
  - Insert dummy nodes in code stream to mark branch targets.
  - Can't resolve branch addresses until after peephole optimization.



Peephole Optimization Examples

ST	3,X	→	ST 3,X	System/370
L	3,X			
MOV	X,R1	→	INC X	PDP-11
ADD	#1,R1			
MOV	R1,X			
L	2,Y	→	MVC Z,Y	System/370
ST	2,Z			
BR	∗ +1	→		All

Data Flow Analysis[a]

- Data Flow Analysis is a technique for discovering properties of the run-time behavior of programs during compilation.
- These properties are *universal*, i.e. they apply to all possible sequences of control and data flow.
- The information from data flow analysis is used to determine feasibility and safety of various optimizations.
- Data Flow Analysis and related optimizations are covered in much greater detail in the follow-on course ECE 540H.

[a] S.S. Muchnick and N.D. Jones (eds.), Program Flow Analysis: Theory and Applications, Prentice-Hall 1981

Definitions for Data Flow Analysis

- *Basic Block* - A sequence of instructions that is always executed from start to finish.
- *Forward Flow* - What can happen *before* a given point in a program.
- *Backward Flow* - What can happen *after* a given point in the program.
- *Definition Point* - Point where a given expression is calculated.
- *Use Point* - Point where a given expression is used.
- *Killed, Kill Point* - An expression is killed (rendered invalid) at any point where one of the variables in the expression is assigned a new value.

Data Flow Analysis Overview

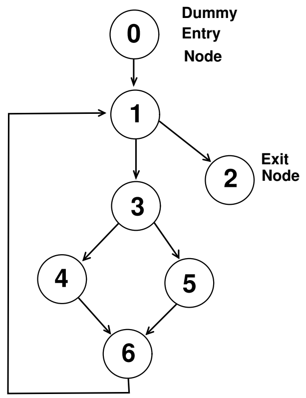
- Subdivide the program into Basic Blocks.
- Analyze the control structure of the program to determine the interrelationship of the basic blocks.
- Represent program control flow as a directed graph (G).
  - Each node in G is a basic block.
  - Each edge in G represents a possible block to block transfer of control.
- Analyze each basic block to determine where expressions are defined, used and killed.

Basic Block Example

Program and Basic Blocks

```
while ( 1 ) {
   scanf( " %f %f %f", &A, &B, &C ) ;      1
   if( A == 0.0 )
      break ;    2
   DISC = B * B - 4.0 * A * C ;      3
   if( DISC >= 0.0 ) {
      DROOT = sqrt( DISC ) ;
      R1 = ( - B + DROOT ) / ( 2.0 * A ) ;      4
      R2 = ( -B  - DROOT ) / ( 2.0 * A ) ;
   } else {
      DROOT = sqrt( - DISC ) ;
      R1 = - B / ( 2.0 * A ) ;      5
      R2 = DROOT / ( 2.0 * A ) ;
   } ;
   printf( "%f %f %f ", DISC, R1, R2 ) ;      6
} ;
```

Control Flow Graph



Available Expression Analysis
Forward Data Flow Example

- An expression is *available* at a point P in the program graph G if every path to P contains a definition of the expression which is not subsequently killed.
- Definitions
  - $Avail(b)$ - the set of expressions available on entry to basic block b .
  - $Predecessors(b)$ - the set of blocks with paths leading to basic block b
  - $NKill(b)$ - the set of expressions *not* killed in basic block b .
  - $Defined(b)$ - the set of expressions that are defined in basic block b and are not subsequently killed within the block.
- For all basic blocks b compute

$$Avail(b) = \bigcap_{x \, \varepsilon \, Predecessors(b)} (Defined(x) \cup (Avail(x) \cap NKill(x)))$$

Live Variable Analysis
Backward Data Flow Example

- A path in the graph G is $X - clear$ if it contains no assignments to the variable X.
  A variable X is *live* at a point P in the graph G if there is a $X - clear$ path from P to a use of X.
- Definitions
  - $Live(b)$ - the set of variables that are live on entry to block b .
  - $In(b)$ - the set of variables that are live on entry to b due to use of the variable in b.
  - $Thru(b)$ - the set of variables redefined (assigned) in b .
  - $Successors(b)$ - the set of successor blocks to block b.
- For all basic blocks b compute

$$Live(b) = In(b) \cup \bigcup_{x \, \varepsilon \, Successors(b)} (Thru(b) \cap Live(x))$$

Data Flow Analysis

- Data Flow equations similar to Available Expression and Live Variable analysis exist for other data flow problems.
- Solution techniques are typically $O(N^2)$ or $O(N)$ in the number of nodes N in G.
- Interprocedural Analysis examines similar questions between different routines.

Value Numbering

- Value numbering is a technique for doing constant and variable folding and common subexpression elimination in a basic block.
- Value Numbering technique
  - Each distinct value created or used within a basic block is assigned a unique value number.
    Two values have the same value number if and only if they are provably identical.
  - Use a hash-coded table of available expressions to detect common subexpressions. Use value number plus operator as the hash key.
  - Constants are handled by special flags in the symbol table and the tuples
  - Start at the beginning of a basic block, Consider each quadruple in the IR in order.
  - Look the tuple up in the hash table to see the same value has already been computed.
  - If a tuple has already been computed it is a common subexpression and can be replaced by the previous computation.

Value Numbering Example

- Basic Block Source

A = 4 ;

K = I ∗ J + 5 ;

L = 5 ∗ A ∗ K ;

M = I ;

B = M ∗ J + I ∗ A ;

- Tuples

1	(assign	, =4	,	, A	)
2	(mult	, I	, J	, R_1	)
3	(add	, R_1	, =5	, R_2	)
4	(assign	, R_2	,	, K	)
5	(mult	, =5	, A	, R_3	)
6	(mult	, R_3	, K	, R_4	)
7	(assign	, R_4	,	, L	)
8	(assign	, I	,	, M	)
9	(mult	, M	, J	, R_5	)
10	(mult	, I	, A	, R_6	)
11	(add	, R_5	, R_6	, R_7	)
12	(assign	, R_7	,	, B	)

Value Numbering Data Structures

- *Symbol Table*
  Name, Value#, Constant Flag
- *Tuple Extension*
  Result Value#, Constant Flag
- *Available Expression Table*
  Left Value#, Operator, Right Value#
  Result Value#, Tuple#
- *Constant Value Table*
  Value#, Constant Value
- The following slides show two intermediate snapshots during processing of the basic block in Slide 325.

Value Numbering up to Tuple T_5

Symbol

Nm	Val#	Con?
=4	1	Yes
A	1	Yes
I	2	No
J	3	No
=5	5	Yes
K	6	No

Tuple Extensions

Tup	Res#	Con?
T_1	1	Yes
T_2	4	No
T_3	6	No
T_4	6	No

Available Expressions

Left#	Op	Right#	Res#	Tuple
2	∗	3	4	T_2
4	+	5	6	T_3

Constants

Val#	Value
1	4
5	5

1	(assign	, =4	,	, A	)
2	(mult	, I	, J	, R_1	)
3	(add	, R_1	, =5	, R_2	)
4	(assign	, R_2	,	, K	)
5	(mult	, =5	, A	, R_3	)
6	(mult	, R_3	, K	, R_4	)

Action at T_5: Discover A and =5 are both constants
Add =20 to Constant Value Table. Delete T_5.
Modify T_6 to use new constant instead of R_3.

Value Numbering up to Tuple T_9

Symbol

Nm	Val#	Con?
=4	1	Yes
A	1	Yes
I	2	No
J	3	No
=5	5	Yes
K	6	No
=20	7	Yes
L	8	No
M	2	No

Tuple Extensions

Tup	Res#	Con?
T_1	1	Yes
T_2	4	No
T_3	6	No
T_4	6	No
T_5	7	Yes
T_6	8	No
T_7	8	No
T_8	2	No

Available Expressions

Left#	Op	Right#	Res#	Tuple
2	∗	3	4	T_2
4	+	5	6	T_3
7	∗	6	8	T_6

Constants

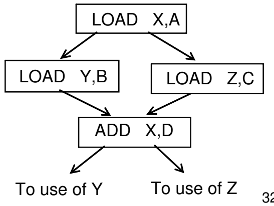
Val#	Value
1	=4
5	=5
7	=20

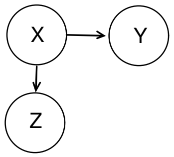
8	(assign	, I	,	, M	)
9	(mult	, M	, J	, R_5	)
10	(mult	, I	, A	, R_6	)
11	(add	, R_5	, R_6	, R_7	)

Action at T_9: Note that M has the same Value# as I.
Look up T_9 in Available Expressions table. Discover same value at T_2
Delete T_9, Replace reference to R_5 in T_{11} by R_1 .

Register Allocation

- Generate tuples assuming an infinite number of pseudo registers (the R_is)
- Perform a Live Variable Analysis (Slide 322) in the pseudo registers.
- Build an *interference graph* of the pseudo registers
  - A pseudo register X interferes with a pseudo register Y if X is live at the point of definition for Y.
  - An interference graph has a node for each pseudo register and an edge between each pair of registers that interfere with one another.
  - If there are N registers available, any node with less than N edges connected to it can be assigned a register.





Register Allocation by Graph Coloring[a]

- Attempt to color the interference graph with N colors where N is the number of available registers.
  - Nodes in the graph that are connected by an edge cannot have the same color.
  - A successful coloring of the graph corresponds to an assignment of the pseudo registers to real registers.
  - If N-coloring succeeds all registers have been assigned.
  - If N-coloring fails, find a region with high register requirements and *spill* some pseudo registers into memory.
  - Selection of registers to be spilled should take into account the cost of loading and storing registers as well as the gain from having data in a register.
- Graph Coloring is an NP-complete problem.
  Apparently linear heuristics have been developed by Chaitin, Chow and others.

[a] G.J. Chaitin, Register Allocation and Spilling via Graph Coloring, Proceedings Sigplan '82 Symposium on Compiler Construction, Sigplan Notices v.17 n.6, June 1982

RISC Machines

- Very simple instructions. Fairly uniform instruction size and timing.
  Only simple addressing modes. Often only register+displacement.
  Array subscripting requires explicit calculations.
  Often a Load/Store architecture. All arithmetic in registers.
- Many fast registers. Register window for parameter passing.
  Memory access is *very slow* relative to register access.
  Optimize usage of cache for maximum performance.
  Load and Store may have several instruction latency.
  Branching may involve several instruction latency and possibility of executing the instruction after the branch.
- Newer RISC machines (e.g. DEC Alpha) have deep instruction pipelines.
  Code generation needs to try to keep pipeline full and busy.
  Branches tend to cause pipeline flushes.
  Newer RISC machines (e.g. Sun SPARC) are multi-issue.
  Can issue (start) more than one instruction on each machine cycle.
  Usually complicated resource rules on which instructions can start together.

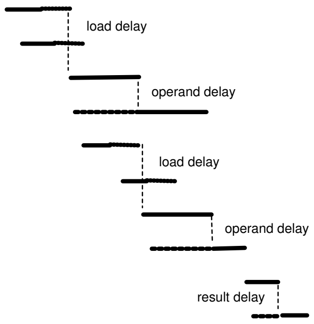
Optimizing for Pipelines

- Assume variable time instructions and a 4 stage pipeline
  Instruction fetch, operand fetch, execute, result store
- Goal is to schedule instructions so that the execution unit is always busy.
- Constraints on instruction scheduling
  - Operand wait (R_1)
    $R_1 \leftarrow R_x \; op_a \; R_y$
    $R_2 \leftarrow R_1 \; op_b \; R_z$
  - Delay store of result (R_1)
    $R_x \leftarrow R_y \; op_a \; R_1$
    $R_1 \leftarrow R_w \; op_b \; R_z$
  - Maintain order of writes (R_1)
    $R_1 \leftarrow R_w \; op_a \; R_x$
    $R_1 \leftarrow R_y \; op_b \; R_z$
  - Limits on available functional units (op_a)
    $R_u \leftarrow R_w \; op_a \; R_w$
    $R_x \leftarrow R_y \; op_a \; R_z$

Pipeline Execution Example

Y = A * X ** 2 + B * X + C

LOAD R1,X	R1 <- X
LOAD R2,A	R2 <- A
MR R0,R1,R1	R0 <- X**2
MR R6,R0,R2	R6 <- A*X**2
LOAD R3,B	R3 <- B
LOAD R4,C	R4 <- C
MR R3,R3,R1	R3 <- B*X
AR R5,R6,R3	R5 <- A*X**2+B*X
AR R7,R5,R4	R7 <- A*X**2+B*X+C
STORE R7,Y	Y <- R7



food for slide eater

food for slide eater

food for slide eater