CSC 488S/CSC 2107S Lecture Notes

These lecture notes are provided for the personal use of students taking CSC488H1S or CSC2107S in the
Winter 2012/2013 term at the University of Toronto

Reading Assignment

```
Fischer, Cytron and LeBlanc

Chapter  13

Omit     13.3.3, 13.3.4, 13.4.1, 13.4.2
         13.5.1, 13.5.2, 13.5.3
         13.6
         MIPS details
```

Code Generation

- Map IR (e.g. quadruples) to some real machine code. Assume
  - Program has passed semantic analysis.
  - All type conversions are explicit in the IR.
  - All address calculations (e.g. array subscripts) are explicit in the IR.
  - Control flow graph (IR branches) is correct.
- Code Generation Actions
  - Perform final storage allocation. Assign hardware addresses (i.e. offsets in activation records) for all variables.
  - Transform IR to machine instructions. Usually one linear pass over IR.
  - Need mechanisms to handle branch address fixups, hardware register allocation.
- Must be able to generate code to handle most general case of the programming language.
- Design IR so that translation to machine code is easy.

Code Generation - Correctness and Consistency

- The code generation design must correctly implement all of the programming language.
- All of the individual code generation designs choices must lead to a *complete* and *consistent* whole.
- The key issue is that the *entire* hardware state at any point in code generation must be correct for the code generation that follows. State includes not only named registers but also internal state (e.g. condition codes).
- Example: implementation of Boolean expressions must lead to correct implementation of conditional statements and Boolean variables.

Code Generation Design

- 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 one or more machine instructions
  - Interacts with register allocation, i.e instructions requiring specific registers.
- Register Allocation
  - Some registers will be dedicated to specific purposes like the display, routine call and return. Often determined by hardware/OS conventions.
  - Need a mechanism for allocating available registers for use in expression evaluation and addressing.
  - Must map large number of pseudo registers used in the IR into a smaller number of physical registers.
  - 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.**

Tuple to Code Example

(A + B) <= (C − D) && (X == Y || Y != Z)

1 (add , A , B , R_1)
2 (sub , C , D , R_2)
3 (leq , R_1 , R_2 , R_3)
4 (branch , R_3 , T5 , ?)
5 (eq , X , Y , R_4)
6 (branch , R_4, ? , T7)
7 (neq , Y , Z , R_5)
8 (branch , R_5 , ? , ?)

falseList: $T4_{false}$, $T8_{false}$

trueList: $T6_{true}$, $T8_{true}$

Tuples from Slides 357 . . . 364

Tuple to Code Example

(A + B) <= (C − D) && (X == Y || Y != Z)

1 (add , A , B , R_1)
2 (sub , C , D , R_2)
3 (leq , R_1 , R_2 , R_3)
4 (branch , R_3 , T5 , ?)
5 (eq , X , Y , R_4)
6 (branch , R_4, ? , T7)
7 (neq , Y , Z , R_5)
8 (branch , R_5 , ? , ?)

```
LOAD    A
LOAD    B
ADD
```

falseList: $T4_{false}$, $T8_{false}$

trueList: $T6_{true}$, $T8_{true}$

Tuple to Code Example

(A + B) <= (C − D) && (X == Y || Y != Z)

1 (add , A , B , R_1)
2 (sub , C , D , R_2)
3 (leq , R_1 , R_2 , R_3)
4 (branch , R_3 , T5 , ?)
5 (eq , X , Y , R_4)
6 (branch , R_4, ? , T7)
7 (neq , Y , Z , R_5)
8 (branch , R_5 , ? , ?)

```
LOAD    A
LOAD    B
ADD
LOAD    C
LOAD    D
SUB
```

falseList: $T4_{false}$, $T8_{false}$

trueList: $T6_{true}$, $T8_{true}$

Tuple to Code Example

(A + B) <= (C − D) && (X == Y || Y != Z)

1 (add , A , B , R_1)

2 (sub , C , D , R_2)

3 (leq , R_1 , R_2 , R_3)

4 (branch , R_3 , T5 , ?)

5 (eq , X , Y , R_4)

6 (branch , R_4, ? , T7)

7 (neq , Y , Z , R_5)

8 (branch , R_5 , ? , ?)

```
LOAD    A
LOAD    B
ADD
LOAD    C
LOAD    D
SUB
CMP
```

falseList: $T4_{false}$, $T8_{false}$

trueList: $T6_{true}$, $T8_{true}$

Tuple to Code Example

(A + B) <= (C − D) && (X == Y || Y != Z)

1 (add , A , B , R_1)

2 (sub , C , D , R_2)

3 (leq , R_1 , R_2 , R_3)

4 (branch , R_3 , T5 , ?)

5 (eq , X , Y , R_4)

6 (branch , R_4, ? , T7)

7 (neq , Y , Z , R_5)

8 (branch , R_5 , ? , ?)

```
         LOAD    A
         LOAD    B
         ADD
         LOAD    C
         LOAD    D
         SUB
         CMP
         BRLEQ   L4178
         BR      L4179
L4178:
```

falseList: $T4_{false}$, $T8_{false}$

trueList: $T6_{true}$, $T8_{true}$

Tuple to Code Example

(A + B) <= (C − D) && (X == Y || Y != Z)

1 (add , A , B , R_1)

2 (sub , C , D , R_2)

3 (leq , R_1 , R_2 , R_3)

4 (branch , R_3 , T5 , ?)

5 (eq , X , Y , R_4)

6 (branch , R_4, ? , T7)

7 (neq , Y , Z , R_5)

8 (branch , R_5 , ? , ?)

```
         LOAD    A
         LOAD    B
         ADD
         LOAD    C
         LOAD    D
         SUB
         CMP
         BRLEQ   L4178
         BR      L4179
L4178:   LOAD    X
         LOAD    Y
         CMP
```

falseList: $T4_{false}$, $T8_{false}$

trueList: $T6_{true}$, $T8_{true}$

Tuple to Code Example

(A + B) <= (C − D) && (X == Y || Y != Z)

1 (add , A , B , R_1)

2 (sub , C , D , R_2)

3 (leq , R_1 , R_2 , R_3)

4 (branch , R_3 , T5 , ?)

5 (eq , X , Y , R_4)

6 (branch , R_4, ? , T7)

7 (neq , Y , Z , R_5)

8 (branch , R_5 , ? , ?)

```
         LOAD    A
         LOAD    B
         ADD
         LOAD    C
         LOAD    D
         SUB
         CMP
         BRLEQ   L4178
         BR      L4179
L4178:   LOAD    X
         LOAD    Y
         CMP
         BREQ    L4180
         BR      L4181
L4181:
```

falseList: $T4_{false}$, $T8_{false}$

trueList: $T6_{true}$, $T8_{true}$

Tuple to Code Example

(A + B) <= (C − D) && (X == Y || Y != Z)

1 (add , A , B , R_1)
2 (sub , C , D , R_2)
3 (leq , R_1 , R_2 , R_3)
4 (branch , R_3 , T5 , ?)
5 (eq , X , Y , R_4)
6 (branch , R_4, ? , T7)
7 (neq , Y , Z , R_5)
8 (branch , R_5 , ? , ?)

falseList: $T4_{false}$, $T8_{false}$

trueList: $T6_{true}$, $T8_{true}$

```
         LOAD     A
         LOAD     B
         ADD
         LOAD     C
         LOAD     D
         SUB
         CMP
         BRLEQ    L4178
         BR       L4179
L4178:   LOAD     X
         LOAD     Y
         CMP
         BREQ     L4180
         BR       L4181
L4181:   LOAD     Y
         LOAD     Z
         CMP
```

Tuple to Code Example

(A + B) <= (C − D) && (X == Y || Y != Z)

1 (add , A , B , R_1)
2 (sub , C , D , R_2)
3 (leq , R_1 , R_2 , R_3)
4 (branch , R_3 , T5 , ?)
5 (eq , X , Y , R_4)
6 (branch , R_4, ? , T7)
7 (neq , Y , Z , R_5)
8 (branch , R_5 , ? , ?)

falseList: $T4_{false}$, $T8_{false}$

trueList: $T6_{true}$, $T8_{true}$

false target: L4179
true target: L4180

```
         LOAD     A
         LOAD     B
         ADD
         LOAD     C
         LOAD     D
         SUB
         CMP
         BRLEQ    L4178
         BR       L4179
L4178:   LOAD     X
         LOAD     Y
         CMP
         BREQ     L4180
         BR       L4181
L4181:   LOAD     Y
         LOAD     Z
         CMP
         BRNEQ    L4180
         BR       L4179
```

Code Generation Design

- Constants, Variables & Expressions
  - Access to all types of variables, including arrays, records.
  - Evaluation of all types of expressions
    numeric expressions, boolean expressions, string expressions
  - Strategy for storing and accessing constants.
- Control flow for all statements
  - Conditional and unconditional branching for control constructs.
  - Scope exiting (non-local) branches.
  - Branches to implement exception handling.
- Run Time Environment
  - Program initialization
  - Program termination
  - Out-of-line support functions
  - Dynamic memory support
  - Exception handling support

- Input & Output
  - Code generation for IO constructs.
  - Support library to implement complex input/output operations
- Procedures & Functions
  - Activation record layout
  - Activation record addressing (i.e. a display)
  - Parameter Passing
  - Call and return
  - Non-local branching and data access
- Modules, Classes & Packages
  - Access to exported data
  - Invocation of member functions
  - Dynamic binding, polymorphism
  - Run time type resolution.

Run Time Environments

- The runtime environment (**RTE**)is the envelope of data structures and algorithms in which the compiled program executes.
- A typical RTE might include
  - Program initialization and termination
  - IO support
  - Exception handling support
  - Support of complex data types, e.g. strings
  - Dynamic semantic analysis checking
  - Support for dynamic program constructs, e.g. dynamic arrays
  - Support for complex statements, e.g switches
  - Support for multi-threading.

Run Time Environment Design Issues

- **Correct** for the general case.
- Interaction/tradeoff with code generation.
  - Generate inline code to implement *thing*.
  - Branch to support function in RTE that implements *thing*.
  - Examples: string operations, dynamic arrays, dynamic memory allocation.
- Space (memory) efficiency.
- Speed (time) efficiency.
- Optimize for common special cases.
- Ideally common core for all compiled programs. e.g. `libctr1.o`

Storage Allocation Issues

- For each scope, place all variables in the appropriate activation record. Assign an offset (address) in the activation record to each variable.
- Storage allocation strategies
  - Assume programmer cannot determine order of variable allocation in activation record.
  - Allocate in order of decreasing alignment constraint (e.g. doublewords, fullwords, halfwords, bytes) to eliminate fill in the activation record.
  - Could allocate in order of declaration and use the record/structure layout algorithms to pack the activation record efficiently.
- Use indirection to deal with data objects that have dynamic sizes. Allocate a pointer in the activation record and emit code to do the dynamic allocation at runtime and fill in the pointer.
- Store literal constants in the code or in a constant area separate from the activation records.

Code Generation Mechanisms

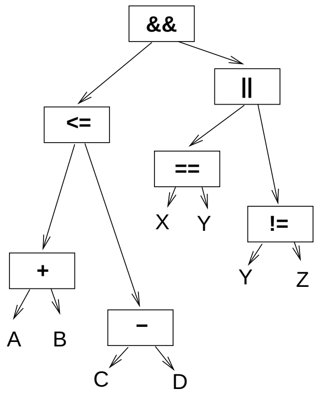
- Need a mechanism for describing the location of operands and the location of results, something like the Data Object in Slide 341 .
- Need a mechanism for describing and manipulating target machine addresses.
- Need a mechanism for allocating registers and freeing them when they are no longer needed.
- 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.

Code Generation for Expressions

- Code for expressions can be generated by doing a *depth first walk* of an AST for the expression.
- Processing expressions depth first is a useful heuristic to minimize register and temporary usage.
  - Constrained by operator precedence.
  - Constrained by language rules for expression evaluation order.
- At each operator node, can also choose order of operand evaluation (subject to language constraints) to minimize register and temporary usage.

Tree Code Generation Example

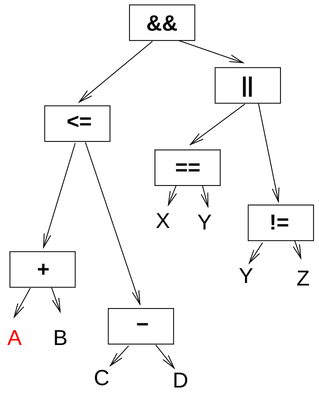
(A + B) <= (C − D) && (X == Y || Y != Z)



Compare with tuple driven code generation example in Slides 410 to 418

Tree Code Generation Example

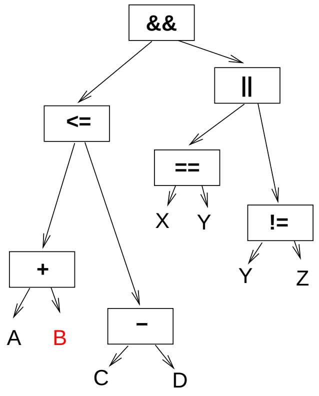
(A + B) <= (C − D) && (X == Y || Y != Z)



```
LOAD   A
```

Tree Code Generation Example

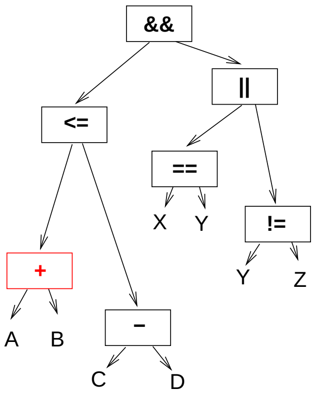
(A + B) <= (C − D) && (X == Y || Y != Z)



```
LOAD   A
LOAD   B
```

Tree Code Generation Example

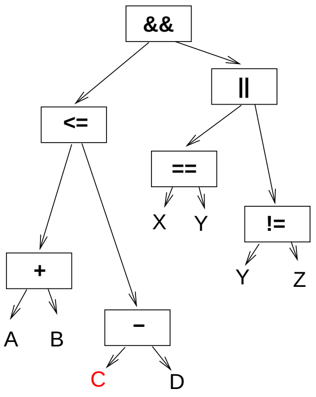
(A + B) <= (C − D) && (X == Y || Y != Z)



```
LOAD   A
LOAD   B
ADD
```

Tree Code Generation Example

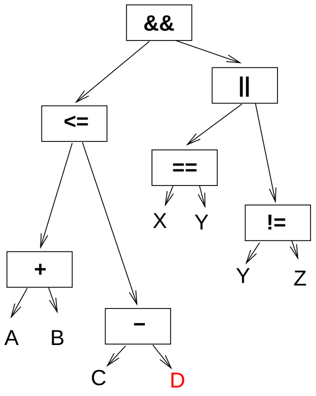
(A + B) <= (C − D) && (X == Y || Y != Z)



```
LOAD   A
LOAD   B
ADD
LOAD   C
```

Tree Code Generation Example

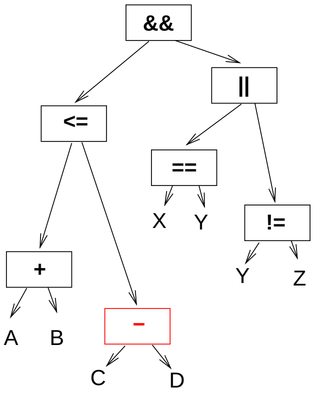
(A + B) <= (C − D) && (X == Y || Y != Z)



```
LOAD   A
LOAD   B
ADD
LOAD   C
LOAD   D
```

Tree Code Generation Example

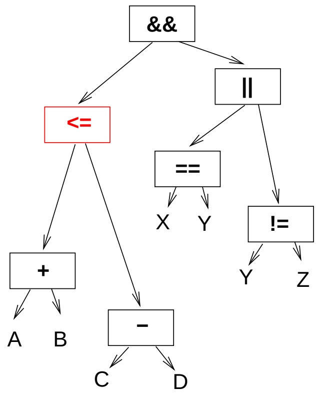
(A + B) <= (C − D) && (X == Y || Y != Z)



```
LOAD   A
LOAD   B
ADD
LOAD   C
LOAD   D
SUB
```

Tree Code Generation Example

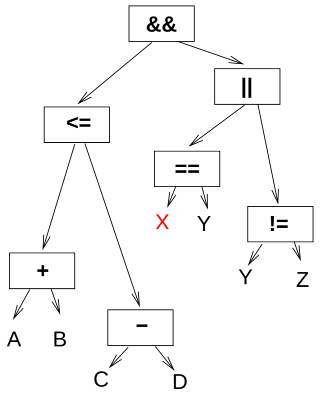
(A + B) <= (C − D) && (X == Y || Y != Z)



```
LOAD   A
LOAD   B
ADD
LOAD   C
LOAD   D
SUB
CMP
BRLEQ   ?
BR      ?
```

Tree Code Generation Example

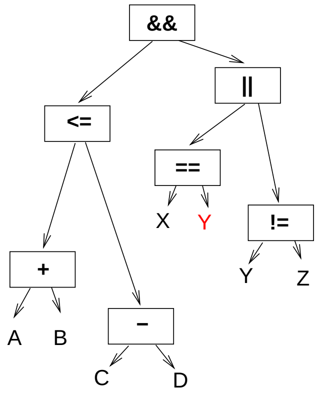
(A + B) <= (C − D) && (X == Y || Y != Z)



```
LOAD   A
LOAD   B
ADD
LOAD   C
LOAD   D
SUB
CMP
BRLEQ   ?
BR      ?
LOAD   X
```

Tree Code Generation Example

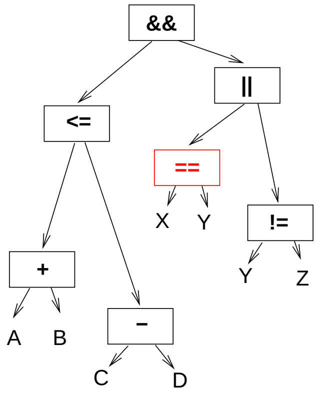
(A + B) <= (C − D) && (X == Y || Y != Z)



```
LOAD   A
LOAD   B
ADD
LOAD   C
LOAD   D
SUB
CMP
BRLEQ   ?
BR      ?
LOAD   X
LOAD   Y
```

Tree Code Generation Example

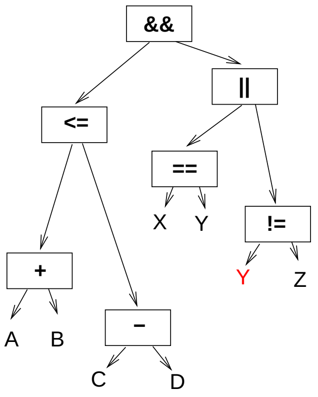
(A + B) <= (C − D) && (X == Y || Y != Z)



```
LOAD   A
LOAD   B
ADD
LOAD   C
LOAD   D
SUB
CMP
BRLEQ   ?
BR      ?
LOAD   X
LOAD   Y
CMP
BREQ    ?
BR      ?
```

Tree Code Generation Example

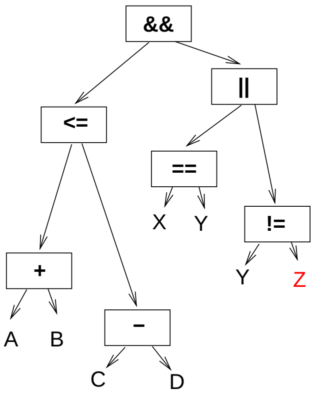
(A + B) <= (C – D) && (X == Y || Y != Z)



```
LOAD   A
LOAD   B
ADD
LOAD   C
LOAD   D
SUB
CMP
BRLEQ    ?
BR       ?
LOAD   X
LOAD   Y
CMP
BREQ     ?
BR       ?
LOAD   Y
```

Tree Code Generation Example

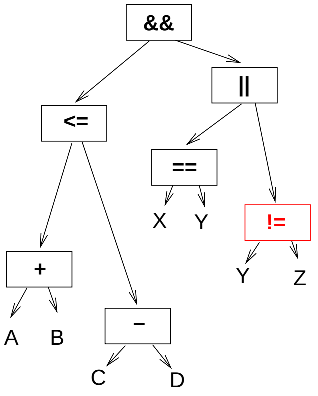
(A + B) <= (C – D) && (X == Y || Y != Z)



```
LOAD   A
LOAD   B
ADD
LOAD   C
LOAD   D
SUB
CMP
BRLEQ    ?
BR       ?
LOAD   X
LOAD   Y
CMP
BREQ     ?
BR       ?
LOAD   Y
LOAD   Z
```

Tree Code Generation Example

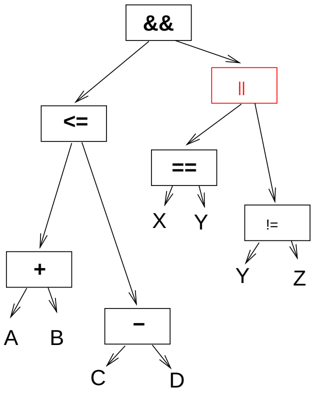
(A + B) <= (C – D) && (X == Y || Y != Z)



```
LOAD   A
LOAD   B
ADD
LOAD   C
LOAD   D
SUB
CMP
BRLEQ    ?
BR       ?
LOAD   X
LOAD   Y
CMP
BREQ     ?
BR       ?
LOAD   Y
LOAD   Z
CMP
BRNEQ    ?
BR       ?
```

Tree Code Generation Example

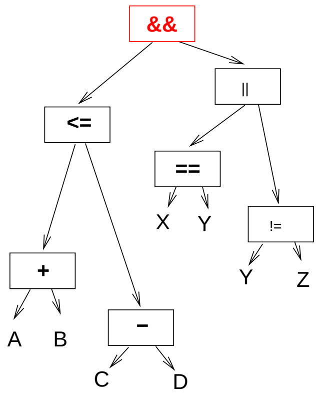
(A + B) <= (C – D) && (X == Y || Y != Z)



```
        LOAD   A
        LOAD   B
        ADD
        LOAD   C
        LOAD   D
        SUB
        CMP
        BRLEQ    ?
        BR       ?
        LOAD   X
        LOAD   Y
        CMP
        BREQ     ?
        BR     L3175
L3175:  LOAD   Y
        LOAD   Z
        CMP
        BRNEQ    ?
        BR       ?
```

Tree Code Generation Example

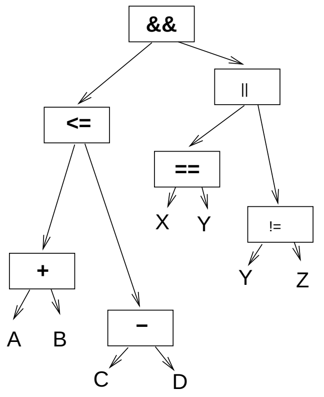
(A + B) <= (C − D) && (X == Y || Y != Z)



```
         LOAD   A
         LOAD   B
         ADD
         LOAD   C
         LOAD   D
         SUB
         CMP
         BRLEQ  L3176
         BR     ?
L3176:   LOAD   X
         LOAD   Y
         CMP
         BREQ   ?
         BR     L3175
L3175:   LOAD   Y
         LOAD   Z
         CMP
         BRNEQ  ?
         BR     ?
```

Tree Code Generation Example

(A + B) <= (C − D) && (X == Y || Y != Z)



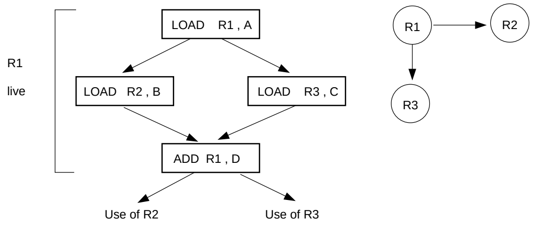
```
         LOAD   A
         LOAD   B
         ADD
         LOAD   C
         LOAD   D
         SUB
         CMP
         BRLEQ  L3176
         BR     L3177
L3176:   LOAD   X
         LOAD   Y
         CMP
         BREQ   L3178
         BR     L3175
L3175:   LOAD   Y
         LOAD   Z
         CMP
         BRNEQ  L3178
         BR     L3177
```

false target: L3177
true target: L3178

Register Allocation

- Generate tuples assuming an infinite number of pseudo registers (the R_is)
- Perform a Live Variable Analysis on the pseudo registers.

  A register is live from where it is given a value to where it is last used.
- 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.


R1 live

LOAD R1 , A

LOAD R2 , B

LOAD R3 , C

ADD R1 , D

Use of R2

Use of R3

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.
- Graph Coloring is an NP-hard 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

Register Spilling

- 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.
- Spilling strategy:
  - Pick a register *R* to be spilled.
  - Allocate a memory location (*Rtmp*) to spill into. Usually in the current activation record.
  - Before each operation that reads R insert

    ```
    LOAD R,Rtmp
    ```
  - After each operation that writes to R insert

    ```
    STORE R,Rtmp
    ```
  - These insertions reduce the live range of R, it is now only live between:
    the `LOAD R,Rtmp` and following instruction.
    the preceding instruction and the `STORE R,Rtmp`
  - Recalculate the interference graph and try recoloring
  - Multiple spills might be required to achieve colorability.

- Spilling a register R reduces its live range and thus its interference with other registers.
- Any choice of register to spill is correct, but the choice may impact program performance.
- Possible spill heuristics
  - Avoid spilling in inner loops.
  - Spill registers with the largest number of conflicts.
  - Spill registers with the smallest number of definitions and uses.
  - ...

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. See Slide 451
- The simple versions of this approach are unable to use the *context* of the quadruple to optimize code selection
  For example could generate INC J for 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

Quadruple Translation Example

subsaddr[a] (*var* , *subReg* , textit resultReg)

Generate code to place address of var [subReg] into resultReg[b]

```
R = targetReg( subReg )                                   Allocate register, preference subReg
if isRegister( subReg ) then                              Is subReg already in a register
    if registerNumber( subReg ) != R then                 targeting failed
        emit( COPYR, R , registerNumber( subReg ))        copy subReg value to register R
else                                                      subReg not in a register
    emit( LOAD, R, subReg )                               load subReg into register R
T = makeConstant( getScale ( var ) )                      run time constant in memory
emit( MUL , R , T )                                       scale subscript by element size
if getLowerBound( var ) != 0 then
    T = makeConstant( getLowerBound( var ) * getScale ( var ) )
    emit( SUB , R, T )                                    normalize subscript
emit(ADDREG , R, getBase( var ) )                         R = baseRegister( var ) + subscript
setResult( resultReg , R , getOffset( var ) )             @var[ subReg ] = R + offset(var)
```

[a] Slide 348

[b] See Slides 261, 262. Assume constant lower and upper bounds.

Examples from Slide 368

Assume array A has a constant lower bound of 1 and an element size of 4 bytes. Assume A is contained in an activation record addressed by D_2 and that the offset of A in this activation record is 240.

Expression:	A[K]	A [K + 1]
Quadruples:	(subsaddr, A , K, R_3)	(add , K , =1 , R_1) (subsaddr , A , R_1 , R_2)
Code:	LOAD R_4,K MULT R_4,=4 SUB R_4,=4 ADDREG R_4,D_2	 MULT R_1,=4 SUB R_1,=4 ADDREG R_1,D_2
Result:	Base: R_4 Offset 240	Base R_1 Offset 240

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

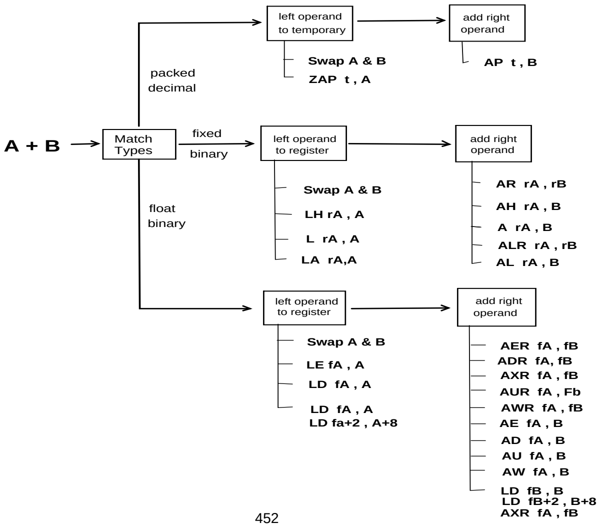
– Perform addition of $left$ and $right$, $result$ describes output

– Assumes registers can be overwritten, addition is commutative.

– Assumes result can be set in template (i.e. no targeting).

$left$	$right$		
	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 $tmp_{reg}, right$ $result = tmp_{reg}$

Example - IBM 370 Code Generation for A + B



Implementing Code Selection

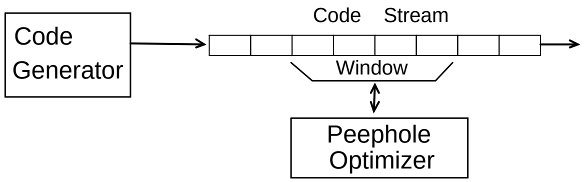
- For complicated (CISC) target machines, instruction selection during code generation can involve very complicated decision processes. The example in Slide 451 is very simplified.
- There are many systems for automating the code selection process
  - Code Generator Builders
  - Glanville-Graham code generation scheme based on parsing the IR to determine the appropriate template.
  - Peephole optimization (Slides 455, 456) 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 and optimizing around load/store and branch latencies is a major issue.

Code Generation Optimizations

- Generating *locally good* code is always a winning strategy.
- Take advantage of constant information known to the compiler.
- Use targeting to encourage results to end up in desirable locations (e.g. registers for parameter passing).
- Use special instructions (e.g. literal instructions, register to register instructions) wherever possible for faster code.
  - Use literal instructions (e.g. AddImmediate) for faster arithmetic.
  - Use register-to-register instructions for speed.
  - Be clever about using instructions in unobvious ways.
    e.g. SDR R,R to set register to zero, BCTR 0,R to decrement register.

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

RTE Design Example - Exception Handling

Modula-3/Java/C++ like mechanism:

```
try {
    . . .
    raise exceptExpression ;
    . . .
}
catch( parm : ExceptType ){
    . . .
}
```

Exception Handling Design Issues

- Associate exception with *dynamically closest* handler.
- Identifying exceptions uniquely at runtime. Efficient handler search.
- RTE generated exceptions, e.g. stack overflow.
- Want non-exception case to be efficient.
  Don't make programmers who don't use them pay for exceptions.
- Separate compilation.
- Efficiency of exception handling can be an issue *if* exceptions are used heavily as an error handling mechanism.
- Exception semantics - *resume* .vs *terminate*
- Correct handling of nested exceptions and reraised exceptions.
- Multi-threaded code.

Exception Handling Design Example

- Exception Identification
  - Assign unique index to RTE generated exceptions
  - Assign symbolic name to user declared exceptions
  - Linker or program initialization maps symbolic exception names into unique runtime indices.
- Exception Handler Location
  - Add pointer field to each activation record. **handler list pointer**
  - This pointer field (if non-null) points to a logical stack of handlers active in the routine.
  - The handler list pointer for the main program, points to a default system provided handler that will catch all exceptions not otherwise caught.

Exception Handling Design Example

- Exception Handler Search Algorithm
  - An exception is raised
  - Start at activation record for the routine in which the exception was raised.
  - Search the list of active handlers pointed at by the handler list pointer.
  - If the signature of the handler matches the exception,
    call the handler like a routine, passing the exception as an argument.
  - If no matching handler is found on this list, chain back on the *dynamic link* to the immediately preceding activation record.
  - Keep searching down the chain of activation records until a handler is found.
  - With *resume semantics* leave the activations records on the stack in place and keep a pointer to the activation record in which the exception occurred.
    With *terminate semantics*, deallocate activation records on the fly as you chain backward.
  - This search is guaranteed to terminate (at the man program)

Exception Handling Design Example

- Code generation for **try**
  - *Prelude:* Generate code to push the addresses of the associated **catch** handler(s) onto the **head** of the list pointed to by the handler list pointer.
  - *Postlude:* generate code to remove the catch handle(s) from the list pointed to by the handle list pointer.
- Code generation for **catch**
  - Generate code for the handler as if it was a routine with the exception as its formal parameter.
  - Handler can exit by: **return**, **go to** or by raising the same or another exception.
  - If handler reraises the same exception, search must start in the current activation record, but *after* the reraising handler.
- Code Generation for **raise**
  - Generate code the save value of the exception expression.
  - Generate branch (terminate semantics) or call (resume semantics) to exception handler search routine in the RTE.