CSE 304
Compiler Design
Code Generation

TONY MIONE

Overview

Goals of a Code Generator

Issues in Design

The Target Language

Addresses in Target Code

Basic Blocks and Flow Graphs

Optimizing Basic Blocks

A Simple Code Generator

Peephole Optimizations

Goals of a Code Generator

Code generators must generate **correct** code

Code generators should generate reasonably efficient code

Issues in Design

- Intermediate Form
  - Quadruples, Triples, Indirect Triples
  - Syntax Trees, DAGS
  - Postfix notation
- Target Architecture
  - CISC – Complex Instruction Set
  - RISC – Reduced Instruction Set
  - Stack-based Architectures (JVM, etc)
- Instruction Selection
  - Complexity affected by
    - Level of the IR
    - Nature of the Instruction Set Architecture
    - Desired quality of code
- Register Allocation
  - Register Allocation
  - Register Assignment
- Evaluation Order

The Target Language

Example machine from Aho:

- Instructions
  - Load Operations
  - Store Operations
  - OP dst, src1, src2
  - Unconditional Jumps
  - Conditional Jumps
- Addressing Modes
  - Variable name/memory address
  - Indexted [a(r)]
  - Offset - 100(R2) ➔ contents(100 + contents(R2))
  - Indirect - *100(R2) ➔ contents(contents(100 + contents(R2)))
  - Immediate - #100
- Simple Instruction cost model

Addresses in Target Code

- Most executables are comprised of 4 different regions
  - **Code** – Executable code lives here. Size can be determined at compile time
  - **Static** – An area for global constants and data generated by compiler. Size can be determined at compile time
  - **Heap** – Dynamically managed area holding data objects allocated and freed during run. Size cannot be determined at compile time.
  - **Stack** – Dynamically managed region holding activation records. Size cannot be determined at compile time.

Stack Allocation

- Access via offset from Stack Pointer (sp) or base/frame pointer (bp)
  - SP moved by size of procedure's activation record at start of procedure code
  - Return address stored at bottom location in activation record
  - SP returned to original value at end of procedure before return
  - Local variables addressed by offset from SP

Example: Calling a procedure

```
ADD SP, SP #caller.recordSize // Adjust stack pointer
ST    0(SP), #here+16             // store return address
BR    callee.codeArea              // jump to procedure code
```

Example: Return from procedure

```
BR   *0(SP)                      // Returns to caller
```

[in caller:

```
SUB SP, SP, #caller.recordSize   // Restore SP to value before call
```

Run time Address for Names

- Code generated uses offsets from start of a region (like static)
  - Initially, intermediate code may express an offset from the start of a region

Example: x is at 12 bytes after the start of static

May express this as static[12]

```
static[12] = 0
```

until the address of the static region is known late in code generation. For example, if static starts at 1000, then x is at 1012.

```
LD   1012, #0
```

Basic Blocks and Flow Graphs

- **Basic Blocks** – Sequence of code that has no transfers into it and no transfers out
- Marking basic blocks help give context to analysis of the IR
  - Can easily mark uses
  - Can easily track which variables/values are 'live'
- Basic Blocks can be linked in a Flow Graph
  - The flow graph indicates which blocks flow into other blocks
  - This helps with doing more global optimization

Basic Blocks and Flow Graphs

- Construction of Basic Blocks:
- Algorithm: Partition three address instructions into basic blocks
- INPUT: A sequence of three-address instructions
- OUTPUT: A list of basic blocks for the sequence where each instruction is assigned to exactly 1 basic block
- METHOD: First, determine which instructions are *leaders,* the first instruction in some basic block
    1. First instruction in the sequence is a *leader*
    2. Any instruction that is the target of a conditional or unconditional jump is a *leader*
    3. Any instruction that follows a conditional or unconditional jump is a *leader*

Example: Basic Block Construction

```
1)  i = 1                    ← Leader (1)
2)  j = 1                    ← Leader (2)
3)  t1 = 10 * i              ← Leader (2)
4)  t2 = t1 + j
5)  t3 = 8 * t2
6)  t4 = t3 - 88
7)  a[t4] = 0.0
8)  j = j + 1
9)  if j <= 10 goto (3)      ← Jump
10) i = i + 1                ← Leader (3)
11) if i <= 10 goto (2)      ← Jump
12) i = 1                    ← Leader (3)
13) t5 = i - 1               ← Leader (2)
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)     ← Jump
```

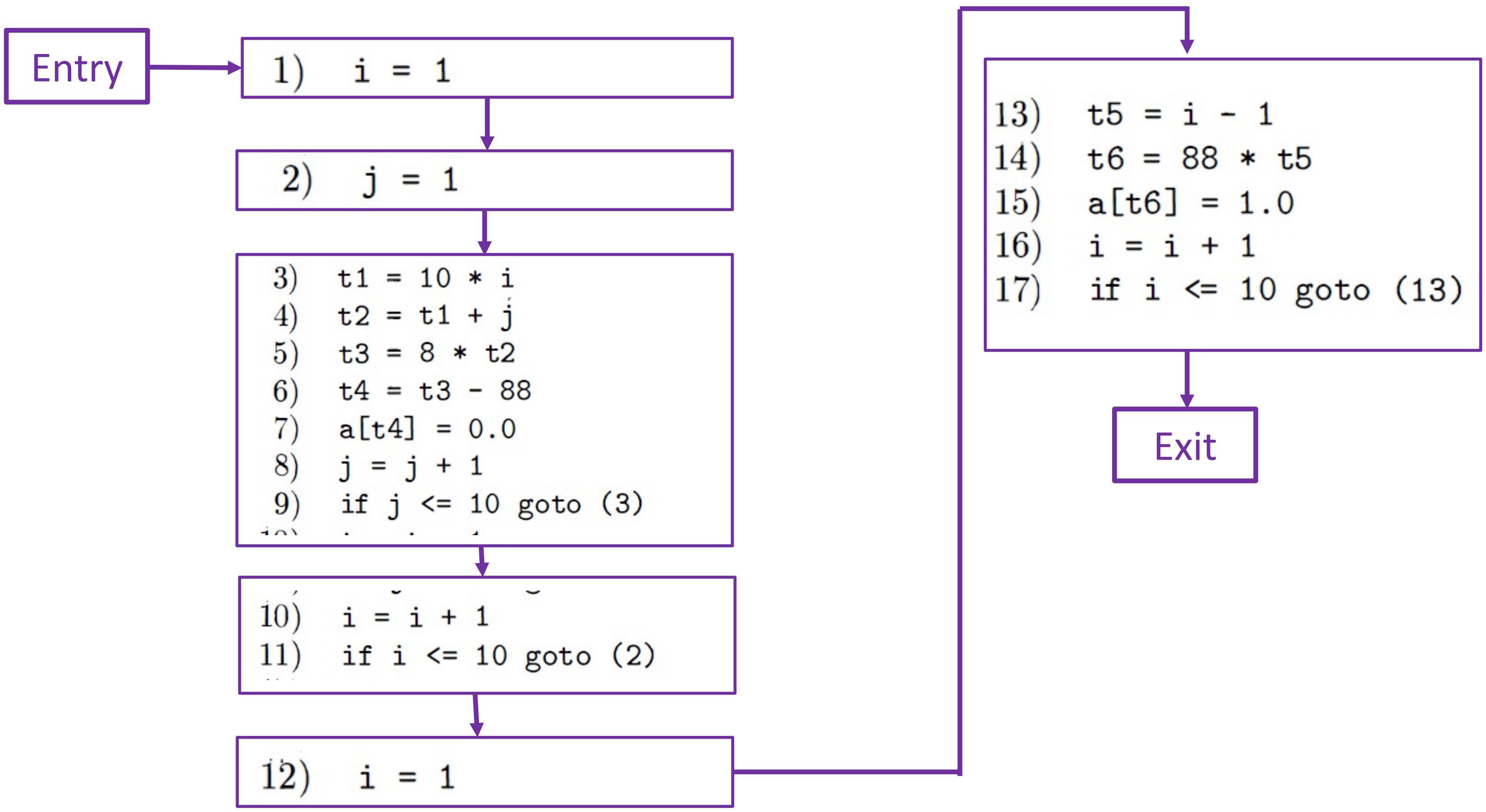
Leader (1)

Leader (2)

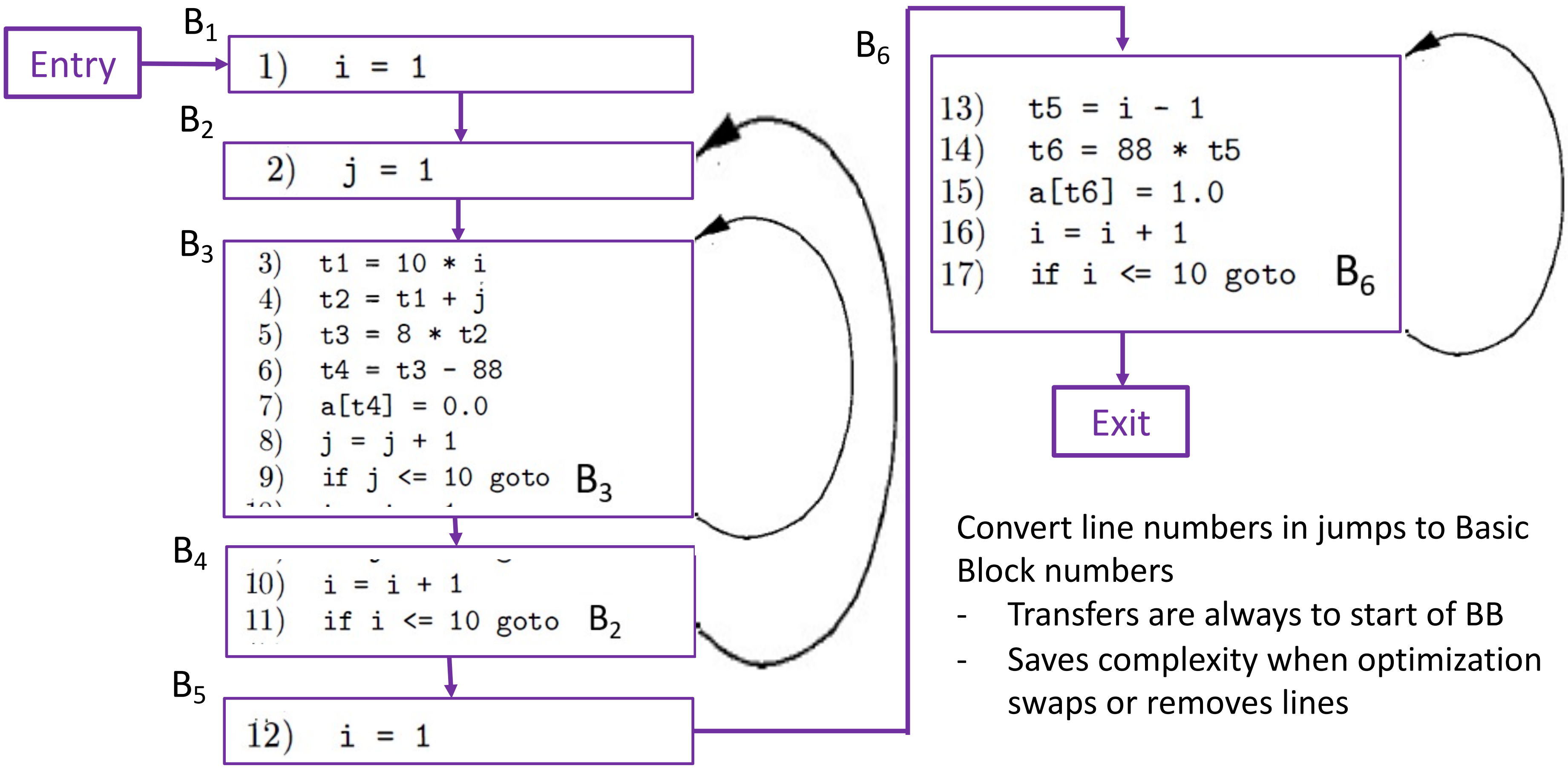
Leader (3)

Jumps

Example: Basic Block Construction



Example: Flow Graph Construction



Convert line numbers in jumps to Basic Block numbers

- Transfers are always to start of BB
- Saves complexity when optimization swaps or removes lines

Determining Next-Use and Liveness

- To generate correct code, we need:
  - Information on a variable's next use in the code
  - Information on a variable's 'liveness'
  - Information on the current location(s) of a variable
- Can generate the information using a reverse scan of a basic block

Determining Next-Use and Liveness

Algorithm: Determining the liveness and next-use information for each statement in a basic block.

INPUT: A basic block of three-address statements. Assume the symbol table shows all non temporary variables as being live on exit from the BB

OUTPUT: At each statement *i* : `x=y OP z` in BB, we attach to *i* the liveness and next-use information of `x`, `y`, and `z`.

METHOD: Start at last statement of BB and scan backwards to the beginning of BB. At each statement, do:

1. Attach to *i* the information currently found in the symbol table regarding next-use and liveness of `x`, `y`, and `z`.
2. In the symbol table, set `x` to 'not live' and 'no next use'.
3. In the symbol table, set `y` and `z` to 'live' and 'next-use' to *i*.

Optimizing Basic Blocks

- Represent Basic Blocks as DAGs
  - Create a node (N) in the DAG for each initial value of the variables in the BB
  - Create a node (N) for each statement (s) within the BB.
    - Children of N are nodes corresponding to statements that are the last definitions (prior to s) of the operands used by s
  - Node (N) is labeled by the operator applied in the statement. Also, attached is a list of variables for which this is the last definition in the BB
  - Certain nodes are output nodes.
    - These are nodes whose variables are *live on exit* from the BB [values may be used later in other successor blocks of the flow graph[
    - Calculation of these variables is based on global data flow analysis
- Four immediate benefits
  - Can eliminate local common *subexpressions*
  - Can eliminate dead code (instructions computing a value that is never used)
  - Can reorder statements that do not depend on each other
  - Can apply algebraic laws to reorder operands ➔ simplify a computation

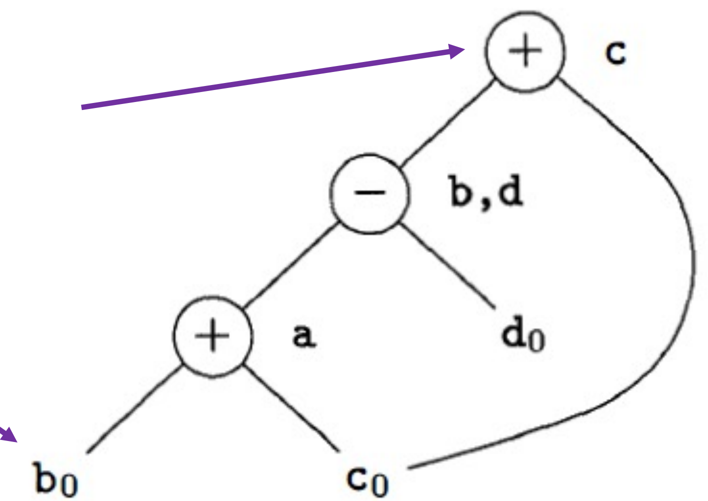
Optimizing Basic Blocks : Common Subexpressions

Using Value-Number method but being careful to get the latest definition of a variable:

```
a = b + c
b = a - d
c = b + c
d = a - d
```

will create the following DAG:

This node with '+' uses the newer definition of b in the '-' node. Not the original 'b'



Optimizing Basic Blocks : Common Subexpressions

Note that the basic construction on the previous slide will not recognize that a and e are the same value in:

```
a = b + c;
b = b - d
c = c + d
e = b + c
```

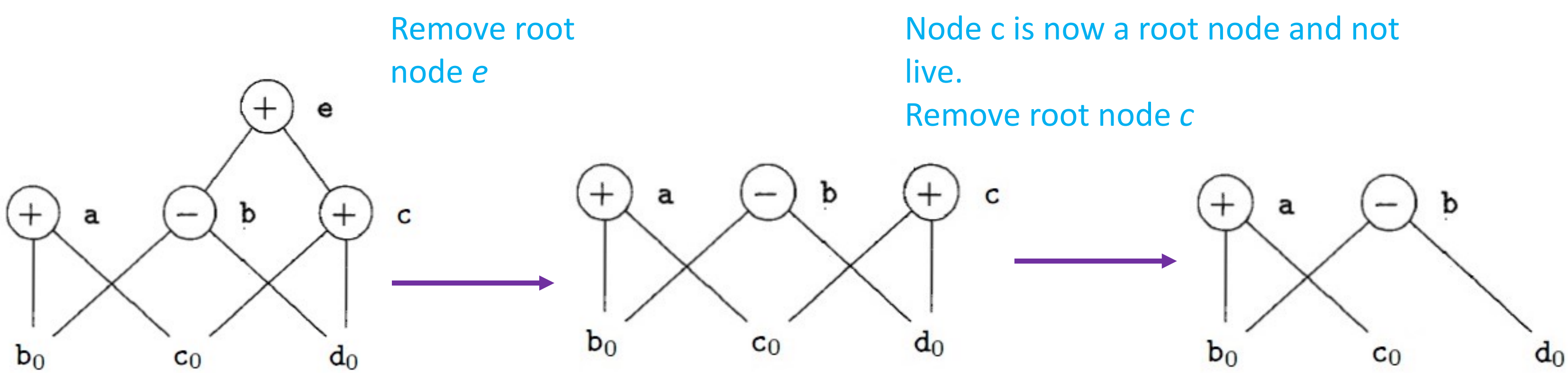
This is because b + c == (b - d) + (c + d)

Using algebraic identities on the DAG may reveal this equivalence.

Optimizing Basic Blocks: Dead Code Elimination

- Dead Code Elimination
  - Procedure:
    - Delete any **root** node from the DAG whose variables are not live at the end of the BB
    - Repeat with any new **root** nodes.

Example: a and b are live, c and e are not live



Optimizing Basic Blocks: Algebraic Optimizations

- Algebraic Identities
  - Ex:
    - x+0 = 0+x = x, x * 1 = 1 * x = x,
    - x – 0 = x, x / 1 = x

- Reduction in Strength

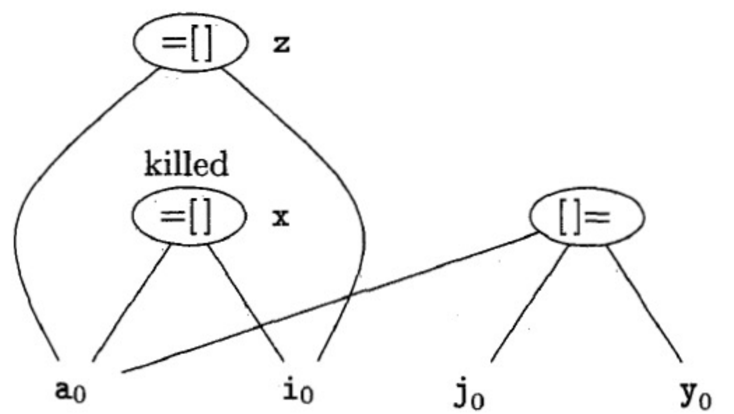
Expensive		Cheaper
x^2	=	x * x
2 * x	=	x + x
x / 2	=	x * 0.5

- Constant Folding
  - Ex: 2 * 3.14   can be replaced at compile time with: 6.28

Optimizing Basic Blocks: Array References

- Operators :
  - =[] (assignment from an array element)
  - []= (assignment to an array element) [3 operands]
- Assigning to an array element 'kills' nodes constructed from the same array
  - 'Kill' means the nodes can have no additional variables attached so cannot be a 'common subexpression'
  - Reason: indices may be the same and so refer to the same element.

```
x = a[i]
a[j] = y
z = a[i]
```



Optimizing Basic Blocks: Pointer Assignments

- Operators :
  - =* (assignment from a pointer derefernce)
  - *= (assignment to a dereferenced pointer)
- *= kills all nodes currently constructed in the DAG!

Optimizing Basic Blocks: Reassembling from DAGs

Basic idea: For each node that has 1 or more attached variables

- construct a three-address statement that computes the value of one of the variables
  - Prefer a variable that is live at end of BB
  - In absence of global data-flow info: assume all variables are live
- For additional variables on a node, generate copy instructions

Optimizing Basic Blocks: Reassembling from DAGs

Additionally:

- Order of instructions must respect order in the DAG (cannot compute Node's value till all its children Nodes are computed)
- Assignment to array must follow all previous assignments/evaluations to/from same array according to order in original BB
- Evaluations of array elements must follow any previous assignments to same array according to order in original BB
- Any variable use must follow all previous procedure calls or indirect assignment through pointers according to order in original BB
- Any procedure call or indirect assignment through a pointer must follow all previous evaluations of any variable according to order in original BB.

A Simple Code Generator

- Issues:
  - Efficient Register Usage
    - Operands for most instructions include registers
    - Registers are useful to hold temporary values
    - Registers may be needed to hold global values for use in another basic block
    - Registers are needed for runtime storage management (SP, FP, etc)
  - Machine Instructions
    - Load values into registers
    - Perform computations
    - Store values into memory
    - For our discussion:
      - LD reg, mem # Loads memory into a register
      - ST mem, reg # Stores a value in a register back into memory
      - OP reg, reg, reg # Performs an operation with values in registers

A Simple Code Generator

- Need a data structure to track where values currently live during code generation
  - Register Descriptors
    - One per register
    - Indicates which variable(s) are currently in the register
    - Initially, all registers are empty
  - Address Descriptors
    - Indicates where a variable’s value is currently
      - Memory
      - Register
      - Stack location
    - Can hold multiple locations
    - Can be maintained in symbol table entry

A Simple Code Generator : The Algorithm

- Use a function getReg() to select registers for each variable in three address instruction ➔ details later
- Traverse 1 BB at a time.
- Consider:
  - Operation type instructions
  - Copy instructions
  - End of BB actions

A Simple Code Generator : The Algorithm

- Operation Instructions
  1. Use **getReg(x=y+z)** to select registers for **x**, **y**, and **z** [R_x, R_y, R_z]
  2. Check R_y register descriptor.
     - If y is not in R_y, issue **LD R_y , y'**
     - Pick y' from one of the locations of y in its address descriptor
  3. Follow the same procedure for R_z
  4. Issue **ADD R_x, R_y, R_z**
- Copy Instructions
  1. Assume getReg() will return same register for x and y
  2. Check register descriptor for R_y. If y is not in that register, issue: **LD R_y, y**
  3. Adjust register descriptor for R_y by adding x
- End of BB code
  - For any non-temporary variable that is live at the end of BB
    - If the variable's address descriptor does NOT list its memory location, then issue **ST x, R**

A Simple Code Generator : The Algorithm

- Managing Register and Address descriptors…use the following actions
  1. For instructions like LD R, x
     a) Change R's register descriptor to hold ony x
     b) Change address descriptor for x by adding register R
  2. For instructions like ST x, R
     a) Change address descriptor for x to include its own memory location
  3. For operations like ADD R_x, R_y, R_z
  4. Copy statements, after managing descriptors for all loads (1)
     a) Add x to the register descriptor for R_y
     b) Change address descriptor for x so its ONLY location is R_y

Example Code Generation

```
t = a - b
u = a - c
v = t + u
a = d
d = v + u
```

R1	R2	R3	a	b	c	d	t	u	v
			a	b	c	d			

```
t = a - b
    LD R1, a
    LD R2, b
    SUB R2, R1, R2
```

R1	R2	R3	a	b	c	d	t	u	v
a	t		a, R1	b	c	d	R2		

```
u = a - c
    LD R3, c
    SUB R1, R1, R3
```

R1	R2	R3	a	b	c	d	t	u	v
u	t	c	a	b	c, R3	d	R2	R1	

. . .

Example Code Generation

```
t = a - b
u = a - c
v = t + u
a = d
d = v + u
```

```
v = t + u
    ADD R3, R2, R1
```

u	t	v

a	b	c	d	R2	R1	R3

```
a = d
    LD R2, d
```

u	a, d	v

R2	b	c	d, R2		R1	R3

Example Code Generation

```
t = a - b
u = a - c
v = t + u
a = d
d = v + u
```

```
d = v + u
     ADD R1, R3, R1
```

d	a	v

R2	b	c	R1			R3

```
exit
     ST a, R2
     ST d, R1
```

d	a	v

a, R2	b	c	d, R1			R3

getReg()

- getReg(I) analyzes instruction i. It returns 2 or 3 registers based on instruction type
- For y and z (illustrating process with y)
  1. If y is currently in a register (R), pick that register. No instruction generation
  2. If y is not in a register but a register is 'empty', pick the empty register
  3. If y is not in a register and no registers are free (hard case) need to select a register and make it 'safe' to use. Examine register descriptors to see which variable(s) (*v*) are held there. Cases:
     a) If address descriptor for *v* says that *v* is somewhere else besides R, then use of R is okay
     b) If *v* is *x* (variable being computed by the instruction) and x is not also the other operand (*z*), then use of R is okay [We know the value of *x* in R is never needed again]
     c) If *v* is not used later and is live on exit from block, then it must be computed elsewhere in the block so use of R is okay
     d) If not okay by one of the first 3 cases, need to generate ST v, R to place v back in its memory location.

     Must repeat d) for each variable held by the register. Count ST instructions generated and that is R's ***score***. Pick the register with the lowest score and use that.

getReg()

- Finally, consider x, the value being computed. Almost the same issues as for y and z. But here are the differences
  - Since *x* is being computed, a Register holding only *x* is fine. This applies even when *x* is also *y* or *z*.
  - If *y* is not used after the instruction I (see 3c), and R_y holds ONLY y, then use R_y as R_x also.
- Special case for copy instructions
  - Pick R_y as described above
  - Use R_y as R_x also

Peephole Optimizations

- Peephole optimization scans a small window of instructions at 1 time.
- Good for the following improvements:
  - Eliminate redundant loads and stores
  - Eliminate unreachable code
  - Flow-of-control optimizations
  - Algebraic simplification and strength reduction

Peephole Optimizations

- Eliminate redundant loads and stores (Example)

```
LD  R0, a
ST a, R0
```

Can eliminate ST instruction (since R0 was loaded from *a* immediately before)

➔BUT, not if ST instruction has a label (instructions must be in same BB)

Peephole Optimizations

- Eliminating unreachable code (Example)

```
        if debug == 1 goto L1
           goto L2
L1:     //print debug information
L2:
```

[eliminate jump over jump…this can be:]

```
 if debug != 1 goto L2
L1:     //print debug information
L2:
```

[If *debug* is set to 0 at start of program, using constant propagation, we can get:

```
      if 0 != 1 goto L2
```

Which really is…

```
      goto L2                    # Allows us to drop all print debug code
```

Peephole Optimizations

- Flow-of-control optimizations (Example)
  - Sometimes we generate various jumps to jumps (conditional or unconditional)

```
goto L1
...
L1: goto L2
```

Becomes

```
goto L2
...
L1: goto L2
```

And also...

```
If a < b goto L1
.
L1: goto L2
```

Becomes

```
If a < b goto L2
.
L1: goto L2
```

Peephole Optimizations

- Algebraic Simplification and Strength Reduction (Example)
  - Identities:

    `x = x + 0`

    `x = x * 1`

  Can simply be removed

`x = y * z` where z is a power of 2 (2^n)

- Can replace multiplication by a shift instruction by n bits

Questions?