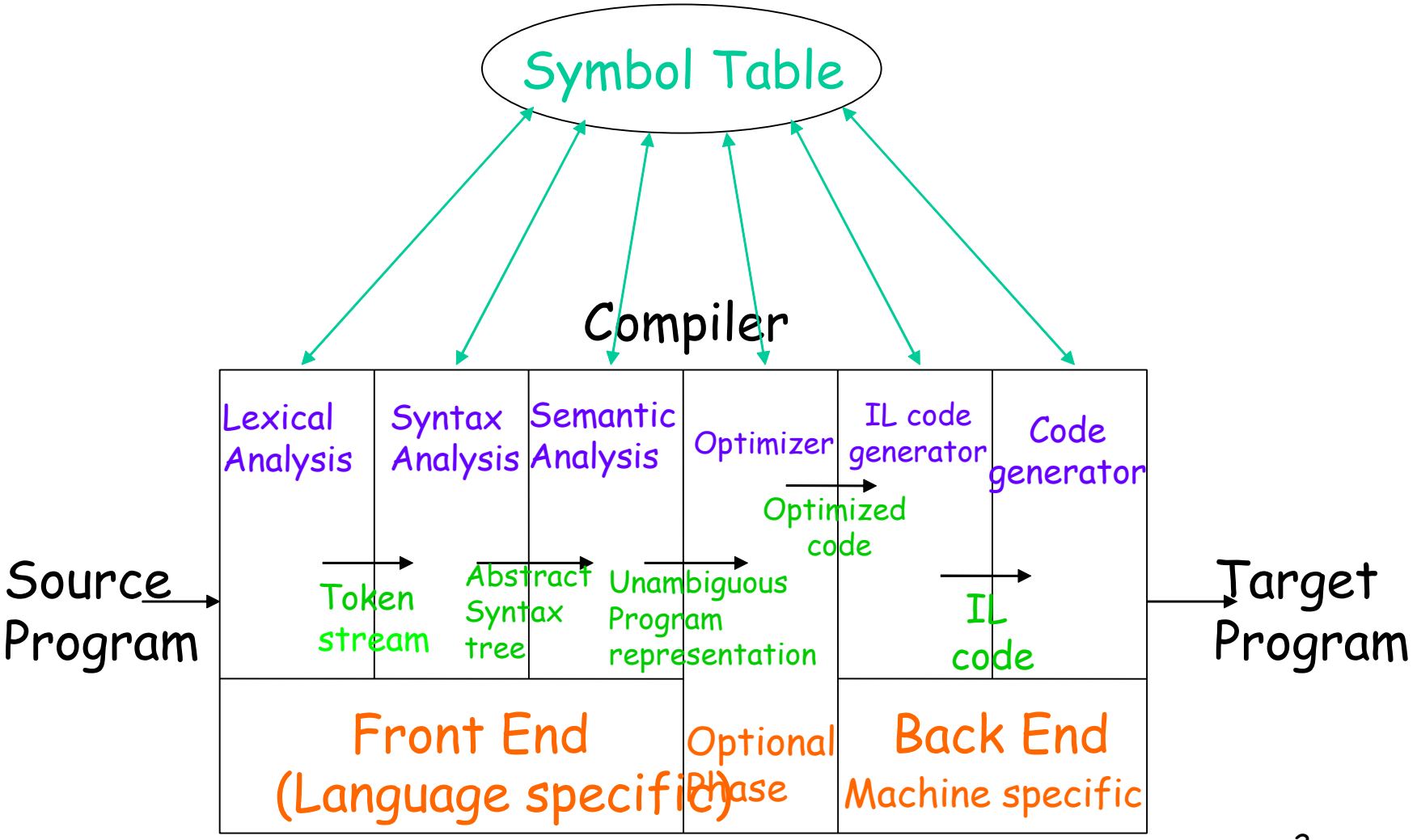
Acknowledgements

The slides for this lecture are modified versions of the offering by **Prof. Sanjeev K Aggarwal**

Compiler structure



Important Compiler Data-Structures

- Symbol-table
- Intermediate Representations
  - Abstract Syntax Tree
  - Three Address Code

Symbol Table

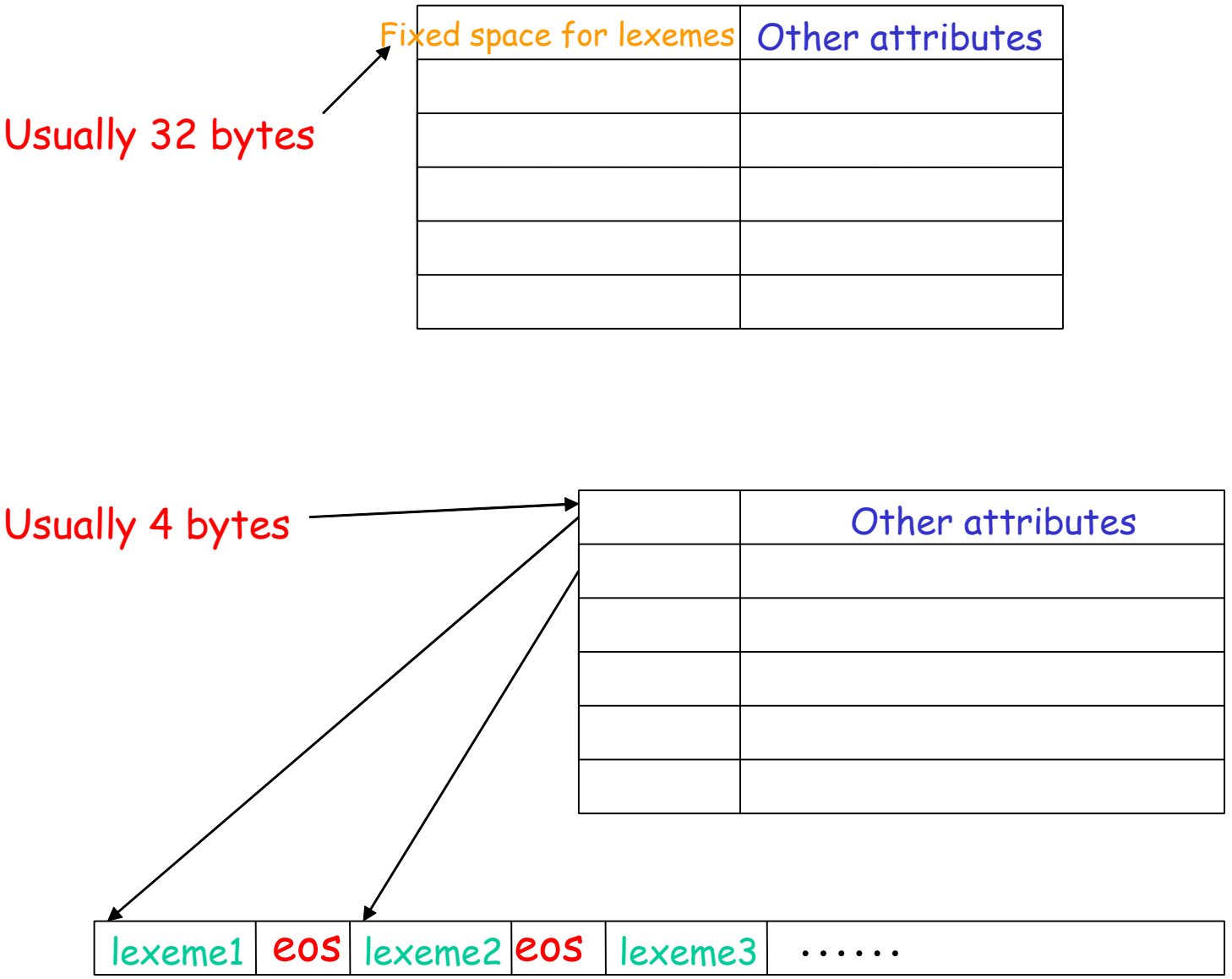
- Stores information for subsequent phases

- Interface to the symbol table
  - Insert(s,t): save lexeme s and token t and return pointer
  - Lookup(s): return index of entry for lexeme s or 0 if s is not found

Implementation of symbol table

- Fixed amount of space to store lexemes. Not advisable as it waste space.

- Store lexemes in a separate array. Each lexeme is separated by eos. Symbol table has pointers to lexemes.


Fixed space for lexemes
Other attributes
Usually 32 bytes
Usually 4 bytes
Other attributes
lexeme1
eos
lexeme2
eos
lexeme3
.....

Three address code

- It is a sequence of statements of the general form X := Y op Z where
  - X, Y or Z are names, constants or compiler generated temporaries
  - op stands for any operator such as a fixed- or floating-point arithmetic operator, or a logical operator

Three address code …

- Only one operator on the right hand side is allowed
- Source expression like x + y * z might be translated into

  $t_1 := y * z$

  $t_2 := x + t_1$

  where t_1 and t_2 are compiler generated temporary names
- Unraveling of complicated arithmetic expressions and of control flow makes 3-address code desirable for code generation and optimization
- The use of names for intermediate values allows 3-address code to be easily rearranged
- Three address code is a linearized representation of a syntax tree where explicit names correspond to the interior nodes of the graph

Three address instructions

- Assignment
  - x = y op z
  - x = op y
  - x = y
- Jump
  - goto L
  - if x relop y goto L
- Indexed assignment
  - x = y[i]
  - x[i] = y
- Function
  - param x
  - call p,n
  - return y
- Pointer
  - x = &y
  - x = *y
  - *x = y

In-memory representation

```
typedef struct {
        InstrType typ; // assign, goto...
        SymtabEntry *in1;
        SymtabEntry *in2
        SymtabEntry *out;
        int target;  // jump target
        Operator op
} Instruction3AC;

Instruction3AC ir[];  // full-program
```

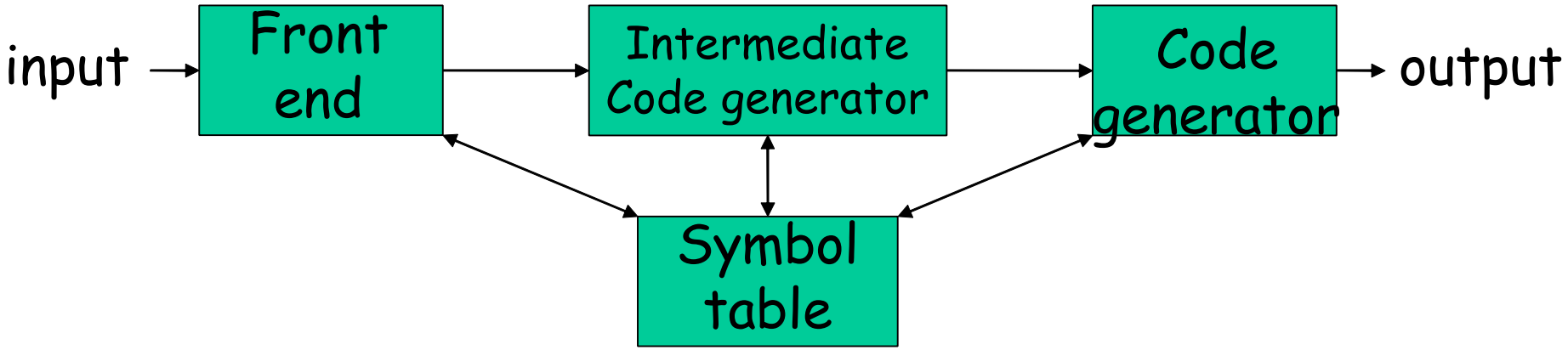
Other representations

- SSA: Single Static Assignment
- RTL: Register transfer language
- Stack machines: P-code
- CFG: Control Flow Graph
- Dominator Trees
- DJ-graph: dominator tree augmented with join edges
- PDG: Program Dependence Graph
- VDG: Value Dependence Graph
- GURRR: Global unified resource requirement representation. Combines PDG with resource requirements
- Java intermediate bytecodes
- The list goes on .....

float a[20][10];
use a[i][j+2]

HIR	MIR	LIR
t1←a[i,j+2]	t1← j+2	r1← [fp-4]
	t2← i*20	r2← r1+2
	t3← t1+t2	r3← [fp-8]
	t4← 4*t3	r4← r3*20
	t5← addr a	r5← r4+r2
	t6← t4+t5	r6← 4*r5
	t7←*t6	r7←fp-216
		f1← [r7+r6]

Code generation and Instruction Selection



- output code must be correct
- output code must be of high quality
- code generator should run efficiently

Issues in the design of code generator

- Input: Intermediate representation with symbol table

  assume that input has been validated by the front end

- target programs :
  - absolute machine language
    fast for small programs
  - relocatable machine code
    requires linker and loader
  - assembly code
    requires assembler, linker, and loader

Sub-tasks for Code Generation

- Instruction Selection
- Register Allocation
- Others: Instruction scheduling, peephole optimizations etc.

Instruction Selection

- Uniformity
- Completeness
- Instruction speed
- Register allocation
  Instructions with register operands are faster
  - store long life time and counters in registers
  - temporary locations
  - Even odd register pairs
- Evaluation order

Instruction Selection

- straight forward code if efficiency is not an issue

a=b+c	Mov b, R_0	
d=a+e	Add c, R_0	
	Mov R_0, a	
	Mov a, R_0	can be eliminated
	Add e, R_0	
	Mov R_0, d	
a=a+1	Mov a, R_0	Inc a
	Add #1, R_0	
	Mov R_0, a	

Target Machine

- Byte addressable with 4 bytes per word
- It has n registers R_0, R_1, ..., R_{n-1}
- Two address instructions of the form
  opcode source, destination
- Usual opcodes like move, add, sub etc.
- Addressing modes

MODE	FORM	ADDRESS
Absolute	M	M
register	R	R
index	c(R)	c+cont(R)
indirect register	*R	cont(R)
indirect index	*c(R)	cont(c+cont(R))
literal	#c	c

Register Allocation Problem

- Map a large number of program variables (virtual registers) to a small number of machine registers
- How many machine registers are required?
  - **Liveness** of variables
- What if the required machine registers are lesser than what is available?
  - **Splitting** live ranges by **spilling** registers
- Which registers to spill?
  - Heuristics: NextUse

Example

1: $t_1 = a * a$

2: $t_2 = a * b$

3: $t_3 = 2 * t_2$

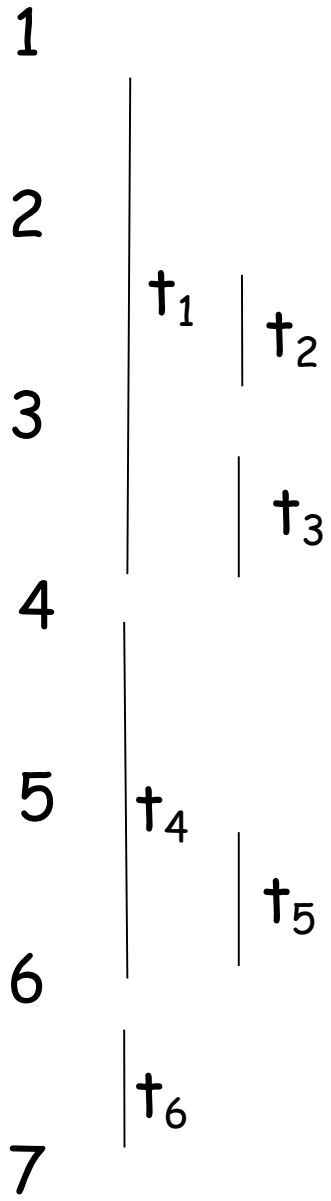
4: $t_4 = t_1 + t_3$

5: $t_5 = b * b$

6: $t_6 = t_4 + t_5$

7: $X = t_6$

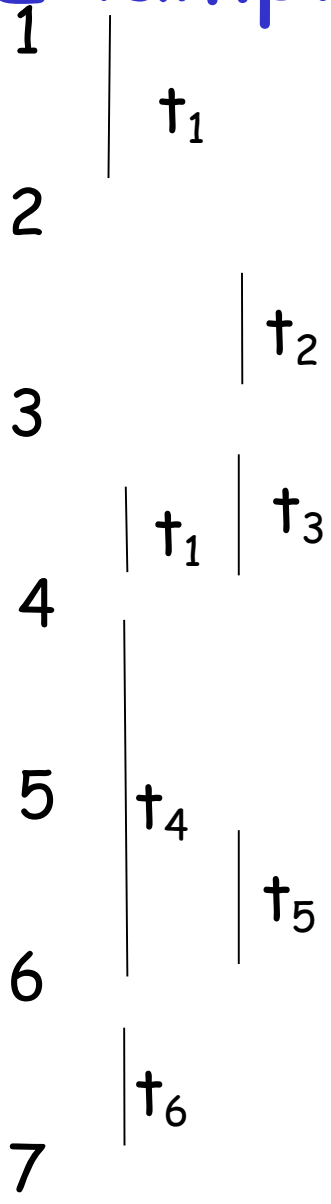
Example …



1: $t_1 = a * a$
2: $t_2 = a * b$
3: $t_3 = 2 * t_2$
4: $t_4 = t_1 + t_3$
5: $t_5 = b * b$
6: $t_6 = t_4 + t_5$
7: $X = t_6$

1: $t_1 = a * a$
2: $t_2 = a * b$
3: $t_2 = 2 * t_2$
4: $t_1 = t_1 + t_2$
5: $t_2 = b * b$
6: $t_1 = t_1 + t_2$
7: $X = t_1$

Example … Live Range Splitting



1: $t_1 = a * a$
2: $t_2 = a * b$
3: $t_3 = 2 * t_2$
4: $t_4 = t_1 + t_3$
5: $t_5 = b * b$
6: $t_6 = t_4 + t_5$
7: $X = t_6$

1: $t_1 = a * a$
1a: store t_1
2: $t_2 = a * b$
3: $t_3 = 2 * t_2$
4a: load t_1
4: $t_4 = t_1 + t_3$
5: $t_5 = b * b$
6: $t_6 = t_4 + t_5$
7: $X = t_6$

Example

1: $t_1 = a * a$
2: $t_2 = a * b$
3: $t_3 = 2 * t_2$
4: $t_4 = t_1 + t_3$
5: $t_5 = b * b$
6: $t_6 = t_4 + t_5$
7: $X = t_6$ STATEMENT

7: no temporary is live
6: t_6:use(7), t_4 t_5 not live
5: t_5:use(6)
4: t_4:use(6), t_1 t_3 not live
3: t_3:use(4), t_2 not live
2: t_2:use(3)
1: t_1:use(4)

Symbol Table

t_1	dead	Use in 4
t_2	dead	Use in 3
t_3	dead	Use in 4
t_4	dead	Use in 6
t_5	dead	Use in 6
t_6	dead	Use in 7

Basic blocks

- sequence of statements in which flow of control enters at the beginning and leaves at the end
- Algorithm to identify basic blocks
- determine leader
  - first statement is a leader
  - any target of a goto statement is a leader
  - any statement that follows a goto statement is a leader
- for each leader its basic block consists of the leader and all statements up to next leader

Flow graphs

- add control flow information to basic blocks
- nodes are the basic blocks
- there is a directed edge from B_1 to B_2 if B_2 can follow B_1 in some execution sequence
  - there is a jump from the last statement of B_1 to the first statement of B_2
  - B_2 follows B_1 in natural order of execution
- initial node: block with first statement as leader

Next use information (Local Register Allocation)

- for register and temporary allocation
- remove variables from registers if not used
- statement X = Y op Z
  defines X and uses Y and Z
- scan each basic blocks backwards
- assume all temporaries are dead on exit and all user variables are live on exit

Algorithm to compute next use information

- Suppose we are scanning

  i : X = Y op Z in backward scan

  - attach to i, information in symbol table about X, Y, Z
  - set X to not live and no next use in symbol table
  - set Y and Z to be live and next use in i in symbol table

Code Generator

- Consider each statement
- Remember if operand is in a register
- **Register descriptor**
  - Keep track of what is currently in each register.
  - Initially all the registers are empty
- **Address descriptor**
  - Keep track of location where current value of the name can be found at runtime
  - The location might be a register, stack, memory address or a set of those

Code Generation Algorithm

for each X = Y op Z do

- invoke a function getreg to determine location L where X must be stored. Usually L is a register.
- Consult address descriptor of Y to determine Y'. Prefer a register for Y'. If value of Y not already in L generate

  Mov Y', L
- Generate

  op Z', L

  Again prefer a register for Z. Update address descriptor of X to indicate X is in L. If L is a register update its descriptor to indicate that it contains X and remove X from all other register descriptors.
- If current value of Y and/or Z have no next use and are dead on exit from block and are in registers, change register descriptor to indicate that they no longer contain Y and/or Z.

Function getreg

1. If Y is in register (that holds no other values) and Y is not live and has no next use after

   X = Y op Z

   then return register of Y for L.

2. Failing (1) return an empty register

3. Failing (2) if X has a next use in the block or op requires register then get a register R (via register allocation heuristic), store its content into M (by Mov R, M) and use it.

4. else select memory location X as L

Example

Stmt	code	reg desc	addr desc
$t_1=a-b$	mov a,R_0	R_0 contains t_1	t_1 in R_0
	sub b,R_0		
$t_2=a-c$	mov a,R_1	R_0 contains t_1	t_1 in R_0
	sub c,R_1	R_1 contains t_2	t_2 in R_1
$t_3=t_1+t_2$	add R_1,R_0	R_0 contains t_3	t_3 in R_0
		R_1 contains t_2	t_2 in R_1
$d=t_3+t_2$	add R_1,R_0	R_0 contains d	d in R_0
	mov R_0,d		d in R_0 and memory

Conditional Statements

- branch if value of R meets one of six conditions negative, zero, positive, non-negative, non-zero, non-positive

if X < Y goto Z

```
Mov X, R0
Sub Y, R0
Jmp negative Z
```

- Condition codes: indicate whether last quantity computed or loaded into a location is negative, zero, or positive

Conditional Statements …

- Compare instruction: sets the codes without actually computing the value
- Cmp X, Y sets condition codes to positive if X > Y and so on

if X < Y goto Z	Cmp X, Y CJL Z

- maintain a condition code descriptor: tells the name that last set the condition codes

X =Y + Z
if X < 0 goto L
	Mov Y,R0
	Add Z, R0
	Mov R0, X
	CJN L