



# CODE GENERATION

## LECTURE 15-16

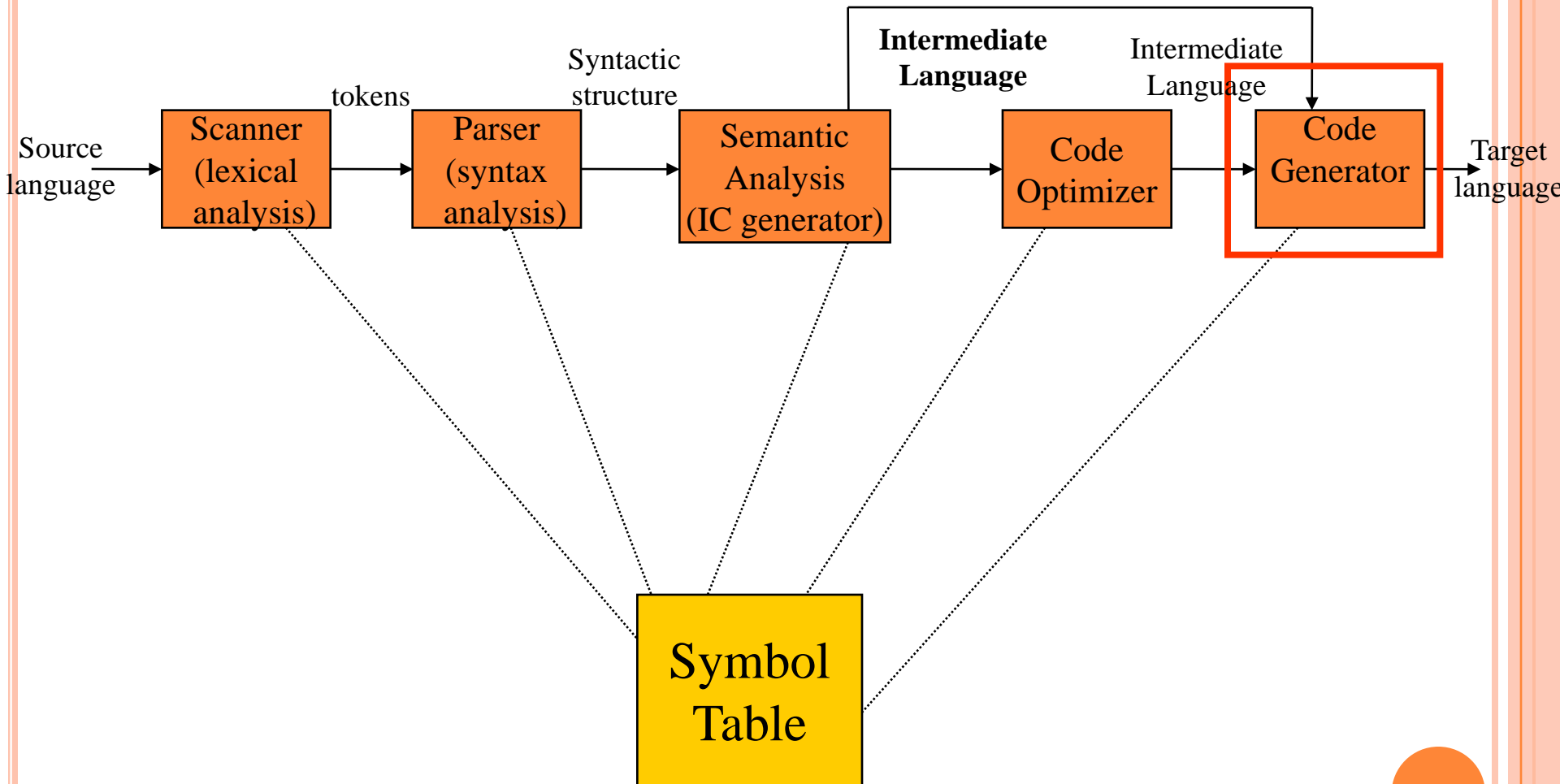
# CODE GENERATION

The code generation problem is the task of mapping intermediate code to machine code.

## Requirements:

- Correctness
  - Must preserve semantic meaning of source program
- Efficiency
  - Make effective use of available resources
  - Code Generator itself must run efficiently

# COMPILER ARCHITECTURE



# INPUT TO THE CODE GENERATOR

- We assume, front end has
  - Scanned, parsed and translate the source program into a reasonably detailed intermediate representations
  - Type checking, type conversion and obvious semantic errors have already been detected
  - Symbol table is able to provide run-time address of the data objects
  - Intermediate representations may be
    - Postfix notations
    - Three address representations
    - Syntax tree
    - DAG

# TARGET PROGRAMS

- The output of the code generator is the target program.
- Target architecture: must be **well** understood
  - Significantly influences the difficulty of code generation
  - RISC, CISC or stack based

# TARGET PROGRAMS

- RISC (Reduced Instruction Set Computer)
  - many registers
  - three address instructions
  - simple addressing modes
  - Simple instruction set architecture

# TARGET PROGRAMS

- Stack-Based Machine
  - operations are done by pushing operands in the stack
  - top of the stack is normally stored in a register
  - stack based machines are almost disappeared.
    - stack organization is too limiting
    - required too many swap and copy operations.

# TARGET PROGRAMS

- CISC (Complex Instruction Set Computer)
  - few registers
  - two address instructions
  - variety of addressing modes
  - several register classes
  - Variable length instructions



# TARGET PROGRAMS

- Target program may be
  - Absolute machine language
    - It can be placed in a fixed location of memory and immediately executed
  - Re-locatable machine language
    - Subprograms to be compiled separately
    - A set of re-locatable object modules can be linked together and loaded for execution by a linker

# TARGET PROGRAMS

- Here,
  - a very simple RISC like computer is assumed as target machine
  - some CISC like addressing mode is also added
  - assembly code is used as target language

# ISSUES IN THE DESIGN OF A CODE GENERATOR

- Instruction Selection
- Register Allocation
- Evaluation Order

# INSTRUCTION SELECTION

- There may be a *large number of ‘candidate’* machine instructions for a given IR instruction
  - Level of IR
    - High: Each IR translates into many machine instructions
    - Low: Reflects many low-level details of machine
  - Nature of the instruction set
    - Uniformity and completeness
  - The desired quality of generated code.
  - Each has own cost and constraints
    - Accurate cost information is difficult to obtain
    - Cost may be influenced by surrounding context

# INSTRUCTION SELECTION

- For each type of three-address statement, *a code skeleton* can be designed that outlines the target code to be generated for that construct.
  - Say,  $x := y + z$ 
    - Mov y, R0
    - Add z, R0
    - Mov R0, x

Statement by statement code generation often produces poor code

# INSTRUCTION SELECTION

$a := b + c$

$d := a + e$

MOV b, R0

ADD c, R0

MOV R0, a

MOV a, R0

ADD e, R0

MOV R0, d

→ If a is subsequently  
used

# INSTRUCTION SELECTION: MACHINE

IR Code:             $x := x + 5$

Target Code:        `mov     x, r0`  
                         `add     5, r0`  
                         `mov     r0, x`

---

IR Code:             $x := x + 1$

Target Code:        `mov     x, r0`  
                         `add     1, r0`  
                         `mov     r0, x`

Target Code:        `mov     x, r0`  
                         `inc     r0`  
                         `mov     r0, x`

Target Code:        `inc     x`

# REGISTER ALLOCATION

- How to best use the bounded number of registers.
- Use of registers
  - Register allocation
    - We select a set of variables that will reside in registers at each point in the program
  - Register assignment
    - We pick the specific register that a variable will reside in.
- Complications:
  - special purpose registers
  - operators requiring multiple registers.
  - Optimal assignment is NP-complete



# REGISTER ALLOCATION

- **Registers are complicated.**
  - x86: Each register made of several smaller registers; can't use a register and its constituent registers at the same time.
  - x86: Certain instructions must store their results in specific registers; can't store values there if you want to use those instructions.
  - MIPS: Some registers reserved for the assembler or operating system.
  - Most architectures: Some registers must be preserved across function calls.

# REGISTER ALLOCATION

Multiply Instruction

`mul y, r4` ← Must specify an even numbered register  
 $r4 \times y \rightarrow [r4, r5]$

Multiply Instruction

`div y, r4` ← Must specify an even numbered register  
 $[r4, r5] \div y \Rightarrow [r4, r5]$

SRDA: Shift Right Double Arithmetic

`srda 32, r6`



# REGISTER ALLOCATION

## IR Code:

```
t := a + b
t := t * c
t := t / d
```

## Target Code:

```
mov    a, r1
add    b, r1
mul    c, r0
div    d, r0
mov    r1, t
```

## IR Code:

```
t := a + b
t := t + c
t := t / d
```

## Target Code:

```
mov    a, r0
add    b, r0
add    c, r0
srda   32, r0
div    d, r0
mov    r1, t
```

## Conclusion:

*Where you put the result of  $t := a + b$  (either  $r0$  or  $r1$ ) depends on how it will be used later!!!*

*[A “chicken-and-egg” problem]*

# EXAMPLE TARGET MACHINE

## A 2-address Architecture

op source, destination

2 operands, at most

## Address Modes:

### *Absolute Memory Address*

mov x, y

sub x, y

$x \rightarrow y$

$y - x \rightarrow y$

### *Register*

mov r0, r1

sub r2, r3

$r3 - r2 \rightarrow r3$

### *Literal*

mov 39, r1

sub 47, r2

Data is included in the instruction directly

### *Indirect Register*

mov r0, [r1]

Register contains an address.  
Moves data in to word pointed to by r1

### *Indirect plus Index*

mov r0, [r1+48]

Use r1+48 as an address.

### *Double Indirect*

mov r0, [[r1+48]]

Go to memory and fetch a second address, "p".  
"p" points to the word.

### Op-Codes:

mov

add

sub

mul

...

# EVALUATION ORDER

- Choosing the order of instructions to best utilize resources
- Picking the optimal order is NP-complete problem
- Simplest Approach
  - Don't mess with re-ordering.
  - Target code will perform all operations in the same order as the IR code
- Trickier Approach
  - Consider re-ordering operations
  - May produce better code
    - ... Get operands into registers just before they are needed
    - ... May use registers more efficiently

# EVALUATION ORDER

$a+b-(c+d)*e$



```
t1:=a+b  
t2:=c+d  
t3:=e*t2  
t4:=t1-t3
```



```
MOV a,R0  
ADD b,R0  
MOV R0,t1  
MOV c,R1  
ADD d,R1  
MOV e,R0  
MUL R1,R0  
MOV t1,R1  
SUB R0,R1  
MOV R1,t4
```

*reorder*



```
t2:=c+d  
t3:=e*t2  
t1:=a+b  
t4:=t1-t3
```



```
MOV c,R0  
ADD d,R0  
MOV e,R1  
MUL R0,R1  
MOV a,R0  
ADD b,R0  
SUB R1,R0  
MOV R0,t4
```

# MOVING RESULTS BACK TO MEMORY

- When to move results from registers back into memory?
  - After an operation, the result will be in a register.
- **Immediately**
  - Move data back to memory just after it is computed.
  - May make more registers available for use elsewhere.
- ***Wait as long as possible before moving it back***
  - Only move data back to memory “at the end”
    - or “when absolutely necessary”
  - May be able to avoid re-loading it later!

# MEMORY TRADEOFFS

- There is an enormous tradeoff between *speed* and *size* in memory.
- SRAM is fast but very expensive:
  - Can keep up with processor speeds in the GHz.
  - As of 2007, cost is \$10/MB
  - Good luck buying 1TB of the stuff!
- Hard disks are cheap but very slow:
  - As of 2012, you can buy a 2TB hard drive for about \$100
  - As of 2012, good disk seek times are measured in ms (about two to four million times slower than a processor cycle!)



# EVALUATING A POTENTIAL CODE SEQUENCE

- Each instruction has a “*cost*”

**Cost = Execution Time**

- Execution Time is difficult to predict.

Pipelining, Branches, Delay Slots, etc.

- **Goal:** Approximate the real cost

A “*Cost Model*”

Simplest Cost Model:

Code Length  $\approx$  Execution Time

Just count the instructions!

# *COST MODEL*

- Depending on what aspect of a program are we interested in optimizing, common cost measures are
  - length of compilation time and size
  - running time
  - power consumption
- Determining the actual cost of compiling and running a program is a complex problem
- For simplicity, we assume
  - cost of an instruction to be one plus costs associated with the addressing modes of the operands.

# A BETTER COST MODEL

Look at each instruction.

Compute a cost (in “units”).

Count the number of memory accesses.

$$\text{Cost} = 1 + \text{Cost-of-operand-1} + \text{Cost-of-operand-2} + \text{Cost-of-result}$$

	<u>example</u>	<u>cost</u>
Absolute Memory Address	x	1
Register	r0	0
Literal	39	0
Indirect Register	[r1]	1
Indirect plus Index	[r1+48]	1
Double Indirect	[ [r1+48] ]	2

Example:    sub    97, r5                      r5 - 97 → r5

$$\text{Cost} = 1 + 0 + 0 + 0 = 1$$

Example:    sub    97, [r5]                      [r5] - 97 → [r5]

$$\text{Cost} = 1 + 1 + 0 + 1 = 3$$

Example:    sub    [r1], [ [r5+48] ]                      [[r5+48]] - [r1] → [[r5+48]]

$$\text{Cost} = 1 + 2 + 1 + 2 = 6$$

# COST GENERATION EXAMPLE

IR Code:     $x := y + z$

Translation #1:

mov	y, x	3	} Cost = 7
add	z, x	4	

Translation #2:

mov	y, r1	2	} Cost = 6
add	z, r1	2	
mov	r1, x	2	

Lesson #1:  
*Use Registers*

Translation #3:

*Assume “y” is in r1 and “z” is in r2*

*Assume “y” will not be needed again*

add	r2, r1	1	} Cost = 3
mov	r1, x	2	

Lesson #2:

*Keep variables in registers*

Translation #4:

*Assume “y” is in r1 and “z” is in r2*

*Assume “y” will not be needed again.*

*Assume we can keep “x” in a register.*

add	r2, r1	1	} Cost = 1

Lesson #4: (not illustrated)  
*Use different addressing modes effectively.*

# GOALS FOR TODAY

- Explore three algorithms for register allocation:
  - Naïve (“no”) register allocation.
  - Linear scan register allocation.
  - Graph-coloring register allocation.

# AN INITIAL REGISTER ALLOCATOR

- **Idea:** Store every value in main memory, loading values only when they're needed.
- To generate a code that performs a computation:
  - Generate **load** instructions to pull the values from main memory into registers.
  - Generate code to perform the computation on the registers.
  - Generate **store** instructions to store the result back into main memory.

# AN INITIAL REGISTER ALLOCATOR

a = b + c;

d = a;

**c = a + d;**

Param N	fp + 4N
...	...
Param 1	fp + 4
Stored fp	fp + 0
Stored ra	fp - 4
a	fp - 8
b	fp - 12
c	fp - 16
d	fp - 20

```
lw    $t0, -12(fp)
```

```
lw    $t1, -16(fp)
```

```
add   $t2, $t0, $t1
```

```
sw    $t2, -8(fp)
```

```
lw    $t0, -8(fp)
```

```
sw    $t0, -20(fp)
```

```
lw    $t0, -8(fp)
```

```
lw    $t1, -20(fp)
```

```
add   $t2, $t0, $t1
```

```
sw    $t2, -16(fp)
```

# AN INITIAL REGISTER ALLOCATOR

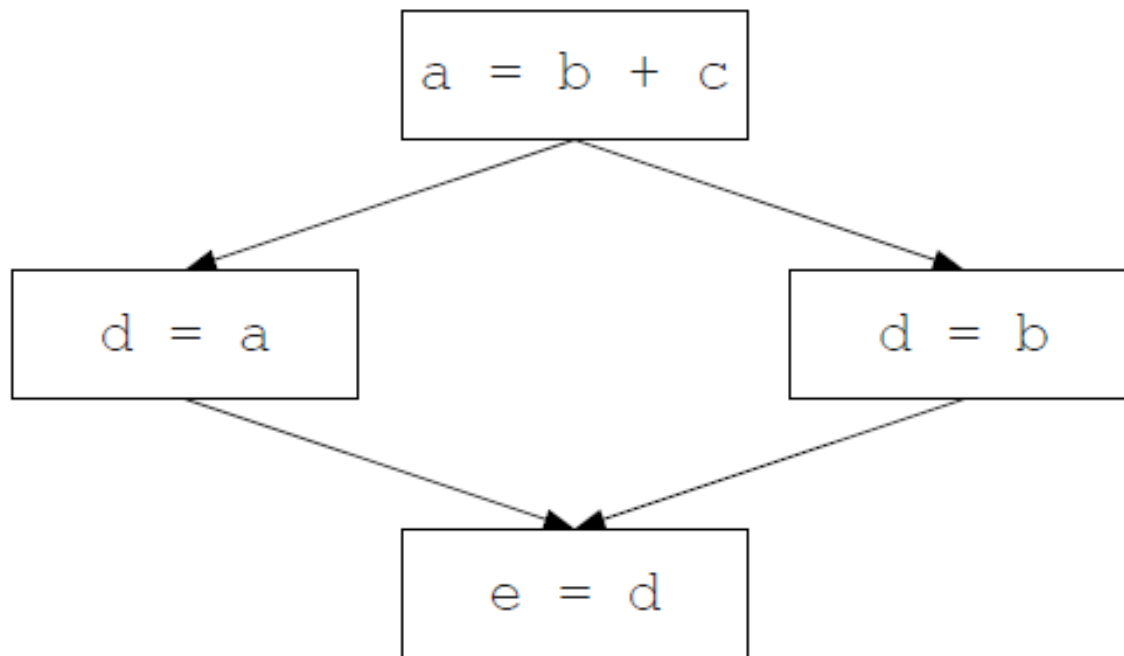
- Disadvantage: **Gross inefficiency.**
  - Issues unnecessary loads and stores by the dozen.
  - Wastes space on values that could be stored purely in registers.
  - Easily an order of magnitude or two slower than necessary.
  - Unacceptable in any production compiler.
- Advantage: **Simplicity.**
  - Can translate each piece of IR directly to assembly as we go.
  - Never need to worry about running out of registers.
  - Never need to worry about function calls or special-purpose registers.
  - Good if you just needed to get a prototype compiler up and running.



# BUILDING A BETTER ALLOCATOR

- **Goal:** Try to hold as many variables in registers as possible.
  - Reduces memory reads/writes.
  - Reduces total memory usage.
- We will need to address these questions:
  - Which registers do we put variables in?
  - What do we do when we run out of registers?

# REGISTER CONSISTENCY



# REGISTER CONSISTENCY

- At each program point, each variable must be in the same location.
  - Does **not** mean that each variable is always stored in the same location!
- At each program point, each register holds at most one live variable.
  - Can assign several variables the same register if no two of them ever will be read together.

# LIVE RANGES AND LIVE INTERVALS

- A variable is **live at a particular program** point if its value may be read later before it is written.
  - Can find this using global liveness analysis.
- The **live range for a variable is the set of program** points at which that variable is live.
- The **live interval for a variable is the smallest** subrange of the IR code containing all a variable's live ranges.
  - A property of the IR code, **not the CFG**.
  - Less precise than live ranges, but simpler to work with.

# BASIC BLOCKS

**Break IR code into blocks such that...**

**The block contains NO transfer-of-control instructions  
... except as the last instruction**

- A sequence of consecutive statements.
- Control enters only at the beginning.
- Control leaves only at the end.

# BASIC BLOCKS

```
      •  
      •  
      •  
Label_43:  t3 := t4 + 7  
          t5 := t3 - 8  
          if t5 < 9 goto Label_44  
          t6 := 1  
          goto Label_45  
Label_44:  t6 := 0  
Label_45:  t7 := t6 + 3  
          t8 := y + z  
          x := t8 - 4  
          y := t8 + x  
Label_46:  z := w + x  
          t9 := z - 5  
          •  
          •  
          •
```

# ALGORITHM TO PARTITION INSTRUCTIONS INTO BASIC BLOCKS

## Concept: “Leader”

*The first instruction in a basic block*

## Idea:

Identify “leaders”

- The first instruction of each routine is a leader.
- Any statement that is the target of a branch / goto is a leader.
- Any statement that immediately follows
  - a branch / goto
  - a call instruction... is a leader

A Basic Block consists of

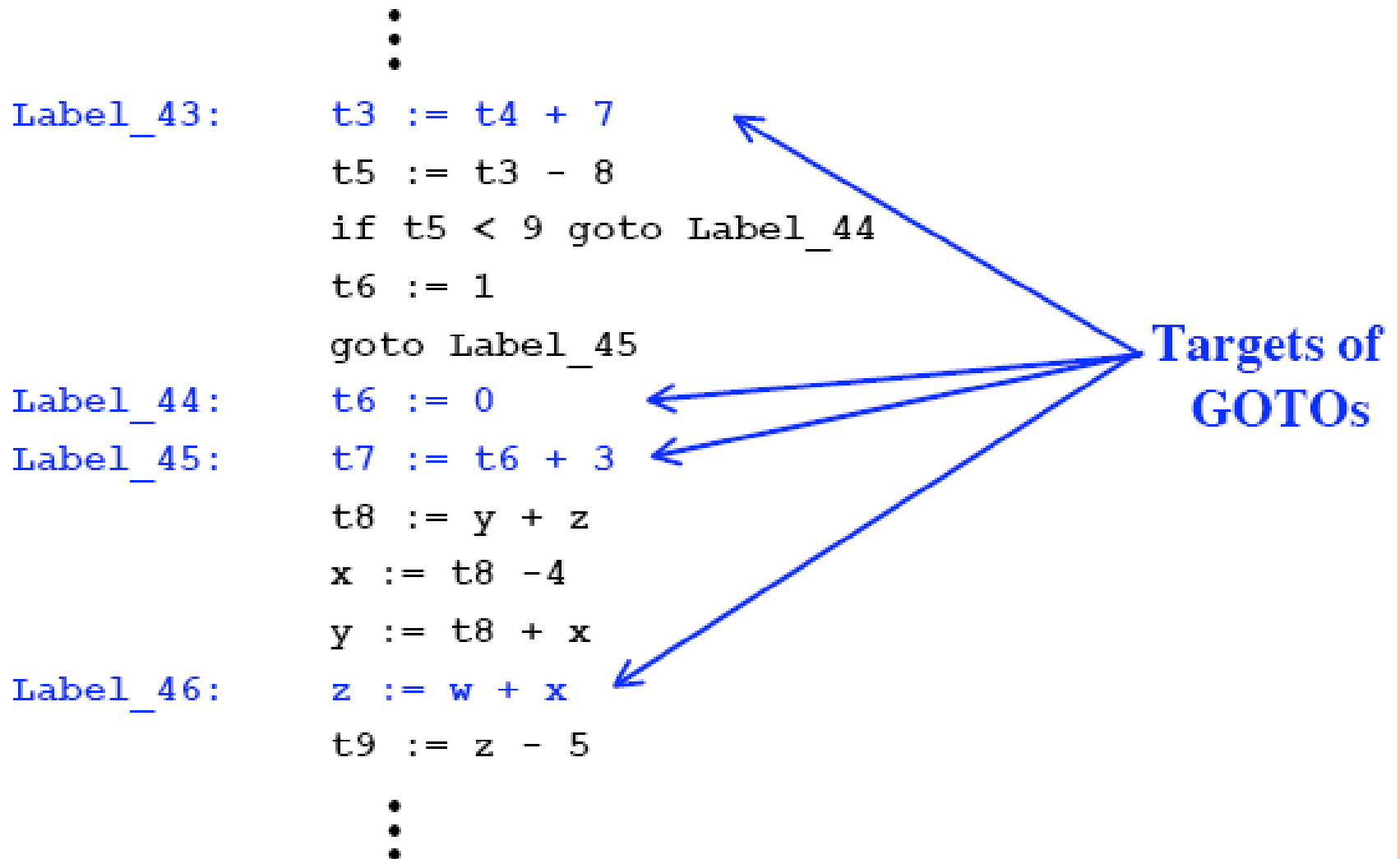
A leader and all statements that follow it  
... up to, but not including, the next leader

# IDENTIFY LEADERS – EXAMPLE 1:

```
      •
      •
      •
Label_43:  t3 := t4 + 7
          t5 := t3 - 8
          if t5 < 9 goto Label_44
          t6 := 1
          goto Label_45
Label_44:  t6 := 0
Label_45:  t7 := t6 + 3
          t8 := y + z
          x := t8 - 4
          y := t8 + x
Label_46:  z := w + x
          t9 := z - 5
          •
          •
          •
```



# IDENTIFY LEADERS – EXAMPLE 1:



# IDENTIFY LEADERS – EXAMPLE 1:

```
      •
      •
Label_43:  t3 := t4 + 7
          t5 := t3 - 8
          if t5 < 9 goto Label_44
          t6 := 1
          goto Label_45
Label_44:  t6 := 0
Label_45:  t7 := t6 + 3
          t8 := y + z
          x := t8 - 4
          y := t8 + x
Label_46:  z := w + x
          t9 := z - 5
          •
          •
          •
```

**Follows  
a GOTO**

# IDENTIFY LEADERS

•  
•  
•

---

```
Label_43:    t3 := t4 + 7  
            t5 := t3 - 8  
            if t5 < 9 goto Label_44
```

---

```
            t6 := 1  
            goto Label_45
```

---

```
Label_44:    t6 := 0
```

---

```
Label_45:    t7 := t6 + 3  
            t8 := y + z  
            x := t8 - 4  
            y := t8 + x
```

---

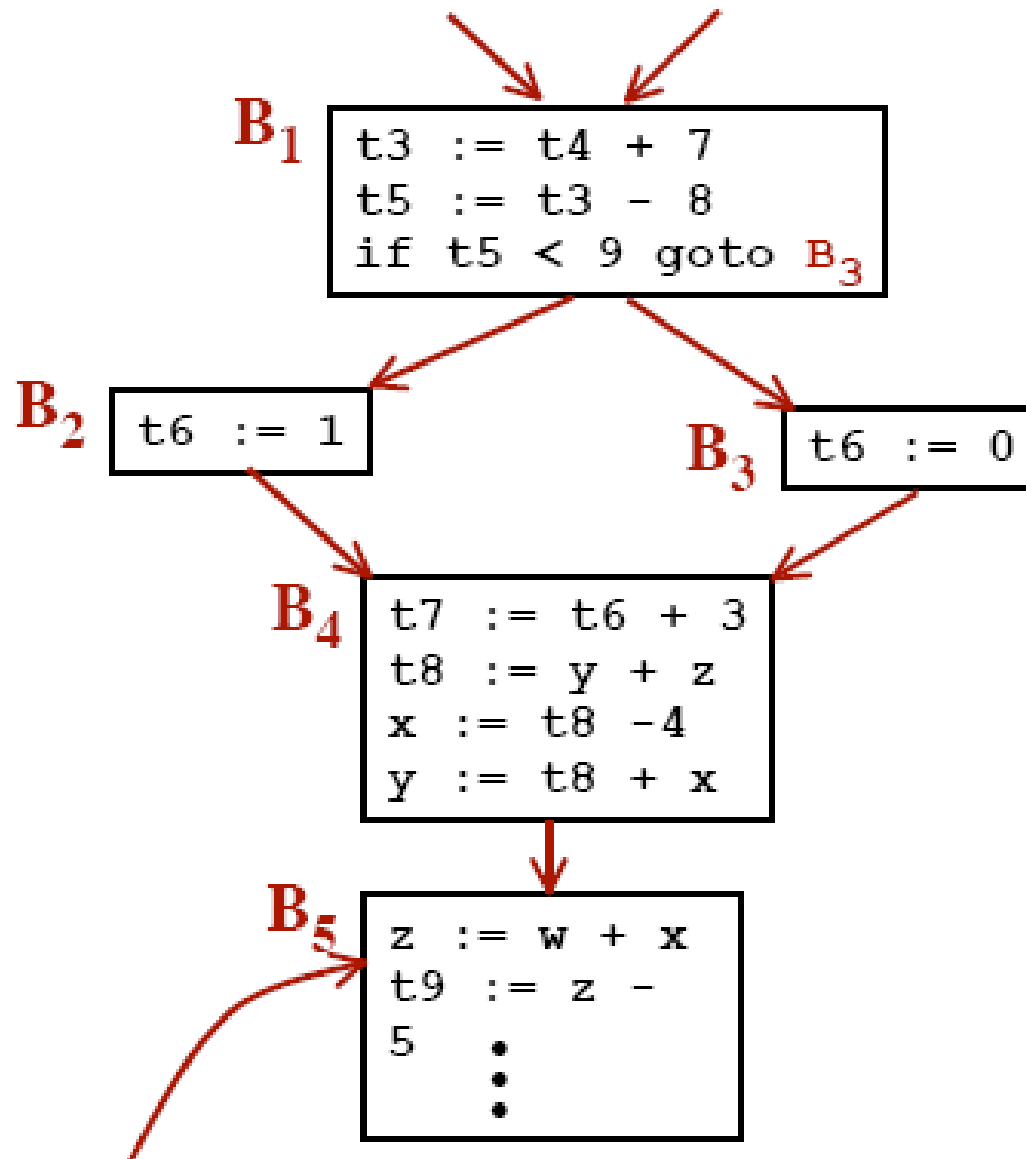
```
Label_46:    z := w + x  
            t9 := z - 5
```

•  
•  
•

# CONTROL-FLOW GRAPHS

- A **control-flow graph** (CFG) is a graph of the basic blocks in a function.
  - The term CFG is overloaded – from here on out, we'll mean “control-flow graph” and not “context-free grammar.”
- Each edge from one basic block to another indicates that control can flow from the end of the first block to the start of the second block.
- There is a dedicated node for the start and end of a function.

# CONTROL FLOW GRAPH



## IDENTIFY LEADERS – EXAMPLE 2:

```
1)  i = 1
2)  j = 1
3)  t1 = 10 * i
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)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

## IDENTIFY LEADERS – EXAMPLE 2:

```
1)  i = 1
2)  j = 1
3)  t1 = 10 * i
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)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

- According to rule 1 : 1
- According to rule 2 : 3, 2, 13
- According to rule 3 : 10, 12

# LOOK AT EACH BASIC BLOCK IN ISOLATION

## Use (B)

The set of variables used (i.e., read) by the Basic Block  
(... before being written / updated)

The “inputs” to the BB


## Def (B)

The set of variables in the Basic Block that are written / assigned to.

The “outputs” of the BB

**B<sub>7</sub>**

<b>x</b>	<b>:=</b>	<b>y</b>	<b>+</b>	<b>v</b>
<b>z</b>	<b>:=</b>	<b>x</b>	<b>*</b>	<b>y</b>
<b>v</b>	<b>:=</b>	<b>z</b>	<b>+</b>	<b>5</b>
<b>if w &lt; v goto B<sub>9</sub></b>				



**Use (B<sub>7</sub>) = y, v, w**

**Def (B<sub>7</sub>) = x, z, v**

View the basic block as a function

**< x, z, v > := f (y, v, w)**



# DEFINITION AND USE OF VARIABLES

A “**Definition**” of variable  $x$  is an instruction that changes the value

$x := \dots$

A “**Use**” of  $x$  is an instruction that reads or uses the value

$\dots := \dots x \dots$

```
104:  y := a + 5
105:  ...
106:  b := y * b
107:  ...
108:  ...
109:  x := b * y
110:  b := b - x
111:  ...
112:  c := y + b
```

This statement defines “y”...

What are the next uses of the  
variable it defines?  
(stmts 106, 109, 112)

*...if control flow could allow this  
value to reach these uses.*

# LIVE VARIABLES


*“Is some variable  $x$  live at some point  $P$  in the program?”*

Could the value of “ $x$ ” at point  $P$  ever be needed later in the execution?

*“Point in a program”*

A point in a program occurs between two statements.

```
...  
a := b + c  
d := e * f  
c := b - 5  
...
```



Point P

The diagram illustrates a point in a program between two statements. It shows a sequence of four lines of code: an ellipsis, 'a := b + c', 'd := e \* f', 'c := b - 5', and another ellipsis. Four red arrows point from the right towards the right-hand side of each line of code. The second arrow, pointing to the right-hand side of 'd := e \* f', is labeled 'Point P' in red text.

Is it possible that the program will ever read from  $x$  along a path from  $P$ ?

[... before “ $x$ ” is written / stored into]

# DEAD VARIABLES

A Variable is “*Dead at point P*”  
= Not Live

Value will definitely never be used.

*No need to compute it!*

*If value is in register, no need to store it!*

# LIVENESS EXAMPLE

→ a := b + c  
d := e \* f  
c := b - 5

*At this point...*

*Is b live? YES*

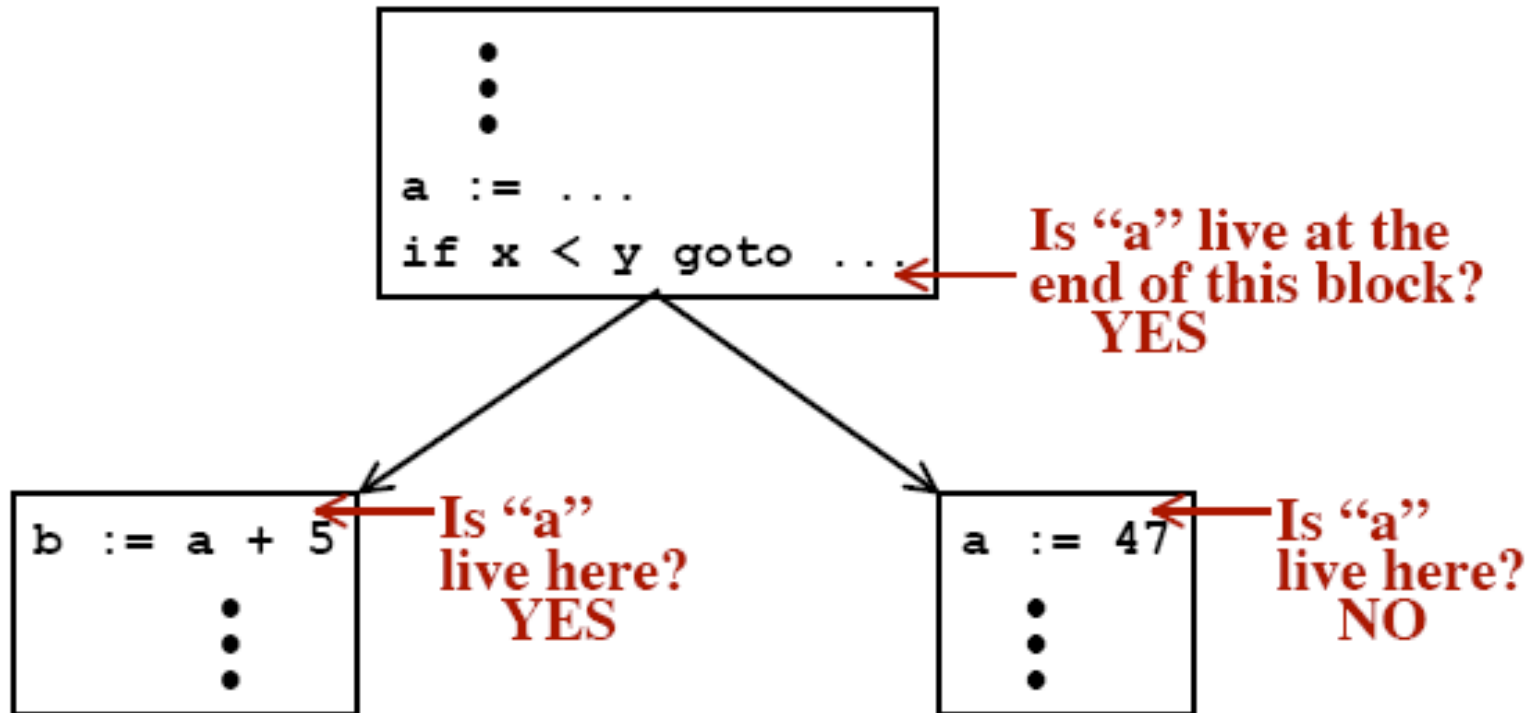
*Is c live? NO*

*Is a live? Don't Know*

*Is g live? Possibly!*

# LIVENESS EXAMPLE

*Must look at the whole “control flow graph” to determine liveness.*



# LIVE VARIABLE ANALYSIS

## Input:

The Control Flow Graph

$\left. \begin{array}{l} \text{Use}(B_i) \\ \text{Def}(B_i) \end{array} \right\} \text{ for all } B_i$

## Output:

$\text{Live}(B_i)$  = a list of all variables live at the end of  $B_i$

*Live Variable Analysis missing?*

Assume all variables are live at the end of each basic block.

# TEMPORARIES

## Assumption:

Each temporary is used in only one basic block  
(True of temps for expression evaluation)

```
...  
t5 := xxxx + xxxx  
...  
xxxx := t5 + xxxx  
...
```

*More precisely:  
No temp will ever  
be in  $Use(B_i)$  for any  $BB$*

## Conclusion:

Temps are never live at the end of a basic block.

*If Live-Variable-Analysis is missing...*

*this assumption can at least identify many dead variables.*

# DEAD CODE

## “Dead Code” (first meaning)

Any code that cannot be reached.  
(Will never be executed.)

```
    x := y + z
    goto Label_45
    a := b + c
    d := e * f
Label_45:
    z := x - a
```

**Dead Code (unreachable)**

## “Dead Code” (second meaning)

A statement which computes a dead variable.

### Example:

```
    b := x * y
    a := b + c
    ...
```

← If “a” is not live here...

*Then eliminate this statement!!!*



# TEMPORARIES

If you can identify a variable which is  
not in  $\text{Use}(B_i)$  for any basic block  
(e.g., a temporary used only in this basic block)

Then you may...

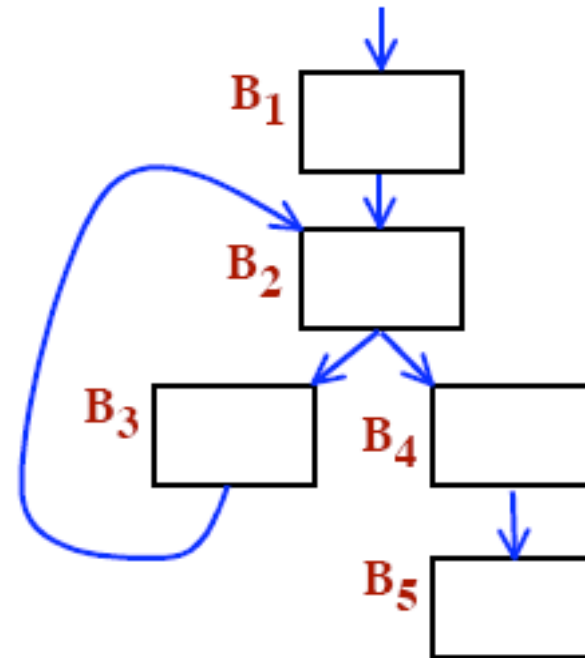
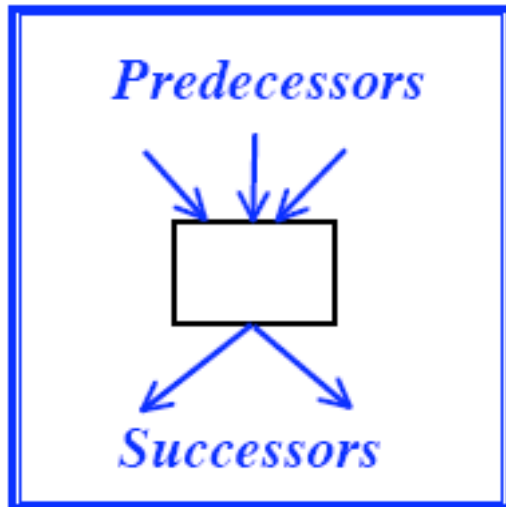
- Rename the variable
- Keep the variable in a register instead of in memory
- Eliminate it entirely (during some optimization)

*Must be careful that the variable*  
*is not used in other routines*  
(i.e., accessed as a non-local from another routine)

# CONTROL FLOW GRAPHS

## Definitions:

- Initial Block
- Predecessor Blocks
- Successor Blocks



# CODE GENERATION ALGORITHM #2

## Focus on one Basic Block at a time

- Ignore all other basic blocks
- Generate best possible code for the basic block

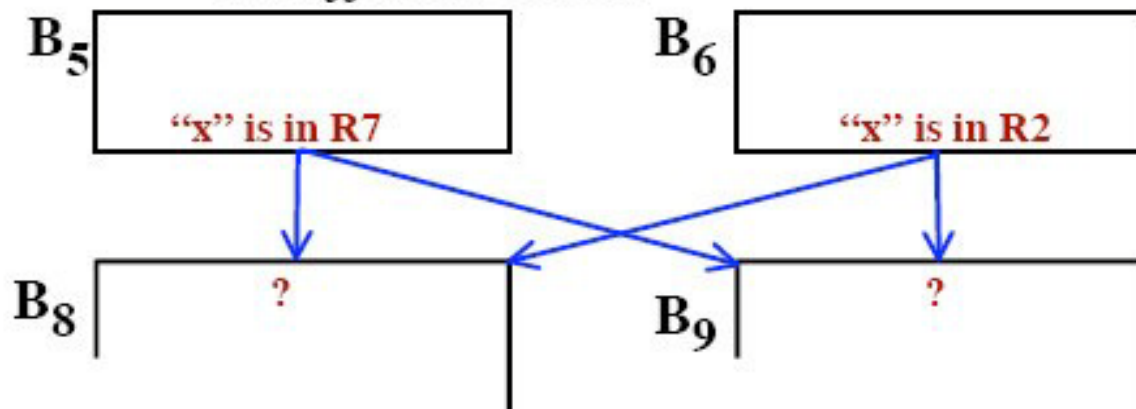
## Register Strategy:

- Store all LIVE variables in memory between basic blocks.
- Within each basic block...  
    Use registers for variables and computation, as necessary
- Each basic block will use registers independently  
    of other basic blocks

# CODE GENERATION ALGORITHM #2

**Q:** Why store all variables at the end of each Basic Block?  
*Why not leave them in registers?*

**A:** Each Basic Block is processed in isolation.  
*A variable might be put in different registers  
in different blocks*



## An Alternate Approach:

Assign "x" to one register for the entire routine  
*... But that ties up a register for too much time!*

# CODE GENERATION ALGORITHM #2

*We'll need to know which variables are LIVE  
at the end of each basic block*

## Option 1:

Perform live variable analysis beforehand  
(during optimization phase)

## Option 2:

Assume every variable is live  
at the end of every basic block

*We'll only store LIVE  
variables at the end of  
each Basic Block*

## Option 3:

Distinguish temporaries from normal variables...

Assume temps are not live between blocks.

Assume normal variables are live between blocks.

*(For more precision, we may want to  
distinguish which variables are in any  $Use(B_i)$  sets*



# THE “NEXT-USE” INFORMATION

Consider a bunch of statements.

(Some of the statements define variables.)

For each “definition”, we want to know...

What are its “**Next-Uses**”?

What statements “use” the value assigned in the definition?

Control flow must be able to go from the “definition”  
to the “use” without any intervening “definitions”.

For each statement, we want to know “What are its **Next-Uses**”?

	<u>Defs</u>	<u>Next-Uses</u>
104: <b>y</b> := a + 5	...	...
105:    ...	104	{106, 109, 112}
106: <b>b</b> := <b>y</b> * <b>b</b>	105	...
107:    ...	106	{109, 110}
108:    ...	107	...
109: <b>x</b> := <b>b</b> * <b>y</b>	108	...
110: <b>b</b> := <b>b</b> - <b>x</b>	...	...
111:    ...	...	...
112: <b>c</b> := <b>y</b> + <b>b</b>		

# THE “NEXT-USE” ALGORITHM

## Goal:

**Process a single basic block**

**Compute the Next-Use info**

**For each IR instruction...**

**$x := y + z$**

**For each variable in the instruction...**

**e.g.,  $x, y, z$**

**Determine...**

**Is the variable LIVE or DEAD after the instruction?**

**If it is LIVE, then...**

**Is it used again in this block?**

**If so, where is it used next?**

## Assumption:

**We already have LIVENESS info for all variables  
at the end of the block.**

# “NEXT-USE” EXAMPLE

1:	t1 := 4 * i	← t1:L(2)	i:L(3)	
2:	t2 := a[t1]	← t2:L(5)	a:L(0)	t1:D
3:	t3 := 4 * i	← t3:L(4)	i:L(8)	
4:	t4 := b[t3]	← t4:L(5)	b:L(0)	t3:D
5:	t5 := t2 * t4	← t5:L(6)	t2:D	t4:D
6:	t6 := prod + t5	← t6:L(7)	prod:D	t5:D
7:	prod := t6	← prod:L(0)	t6:D	
8:	t7 := i + 1	← t7:L(9)	i:D	
9:	i := t7	← i:L(10)	t7:D	
10:	if i <= 20 goto ..	← i:L(0)		

## Key:

L(4) *Live; next-use in statement 4*  
L(0) *Live; no next-use in this block*  
D *Dead*



# “NEXT-USE” ALGORITHM

- Identify all variables used in this block.

- Use a table

One entry for each variable

For each variable, store...

Its current status

LIVE or DEAD

If LIVE, its next-use in this block

(0=not used again in this block)

*A temporary data structure,  
used only for this algorithm*  
(Implementation Idea: Add fields  
to “VarDecl” to hold this info)

- Start with the LIVEness info  
at the BOTTOM of the block.
- Work through the block in reverse order  
instruction-by-instruction
- Update the table, as we go upward.



# “NEXT-USE” ALGORITHM

INITIALIZE the table

Use results from LIVE-VARIABLE ANALYSIS, if available

Else, set all variables to L(0) -- LIVE after this block

Go through the instructions in REVERSE order...

FOR each instruction DO

Let the instruction be:

5.        t5 := t2 \* t4

n.        x := y<sub>1</sub> ⊕ y<sub>2</sub>

*Let “x” be the variable DEFINED, in any.  
Let “y1, y2, ...” be any USED variables.*

Look up the current status of each variable (x, y<sub>1</sub>, y<sub>2</sub>, ...)

Fill in the NEXT-USE info for this instruction.

Set the status of “x” to “D”

Set the status of “y<sub>1</sub>” to “L(n)”

Set the status of “y<sub>2</sub>” to “L(n)”

*NOTE: Could have the same variable  
being DEFINED and USED:*

*i := i + 1*

*Must set status of the DEFINED  
variable first;*

*Then set/change the status of the  
USED variables.*

ENDFOR

# “NEXT-USE” ALGORITHM - EXAMPLE

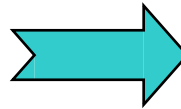
```
1:  t1 := 4 * i
2:  t2 := a[t1]
3:  t3 := 4 * i
4:  t4 := b[t3]
5:  t5 := t2 * t4
6:  t6 := prod + t5
7:  prod := t6
8:  t7 := i + 1
9:  i := t7
10: if i <= 20 goto...
```

t1:	D
t2:	D
t3:	D
t4:	D
t5:	D
t6:	D
t7:	D
a:	L(0)
b:	L(0)
prod:	L(0)
i:	L(0)

# “NEXT-USE” ALGORITHM - EXAMPLE

```
1:  t1 := 4 * i
2:  t2 := a[t1]
3:  t3 := 4 * i
4:  t4 := b[t3]
5:  t5 := t2 * t4
6:  t6 := prod + t5
7:  prod := t6
8:  t7 := i + 1
9:  i := t7
10: if i <= 20 goto .. ← i:L(0)
```

t1:	D
t2:	D
t3:	D
t4:	D
t5:	D
t6:	D
t7:	D
a:	L(0)
b:	L(0)
prod:	L(0)
i:	L(0)



t1:	D
t2:	D
t3:	D
t4:	D
t5:	D
t6:	D
t7:	D
a:	L(0)
b:	L(0)
prod:	L(0)
i:	L(10)

# “NEXT-USE” ALGORITHM - EXAMPLE

```
1:  t1 := 4 * i
2:  t2 := a[t1]
3:  t3 := 4 * i
4:  t4 := b[t3]
5:  t5 := t2 * t4
6:  t6 := prod + t5
7:  prod := t6
8:  t7 := i + 1
9:  i := t7
10: if i <= 20 goto ..
```

← i:L(10)      t7:D  
← i:L(0)

t1:	D
t2:	D
t3:	D
t4:	D
t5:	D
t6:	D
t7:	D
a:	L(0)
b:	L(0)
prod:	L(0)
i:	L(10)

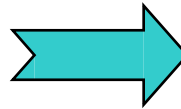


t1:	D
t2:	D
t3:	D
t4:	D
t5:	D
t6:	D
t7:	L(9)
a:	L(0)
b:	L(0)
prod:	L(0)
i:	D

# “NEXT-USE” ALGORITHM - EXAMPLE

```
1:  t1 := 4 * i
2:  t2 := a[t1]
3:  t3 := 4 * i
4:  t4 := b[t3]
5:  t5 := t2 * t4
6:  t6 := prod + t5
7:  prod := t6
8:  t7 := i + 1      ← t7:L(9)    i:D
9:  i := t7          ← i:L(10)   t7:D
10: if i <= 20 goto ← i:L(0)
```

t1:	D
t2:	D
t3:	D
t4:	D
t5:	D
t6:	D
t7:	L(9)
a:	L(0)
b:	L(0)
prod:	L(0)
i:	D



t1:	D
t2:	D
t3:	D
t4:	D
t5:	D
t6:	D
t7:	D
a:	L(0)
b:	L(0)
prod:	L(0)
i:	L(8)

# “NEXT-USE” ALGORITHM - EXAMPLE

```
1:  t1 := 4 * i
2:  t2 := a[t1]
3:  t3 := 4 * i
4:  t4 := b[t3]
5:  t5 := t2 * t4
6:  t6 := prod + t5
7:  prod := t6
8:  t7 := i + 1
9:  i := t7
10: if i <= 20 goto ..
```

← prod:L(0) t6:D  
← t7:L(9) i:D  
← i:L(10) t7:D  
← i:L(0)

t1:	D
t2:	D
t3:	D
t4:	D
t5:	D
t6:	D
t7:	D
a:	L(0)
b:	L(0)
prod:	L(0)
i:	L(8)



t1:	D
t2:	D
t3:	D
t4:	D
t5:	D
t6:	L(7)
t7:	D
a:	L(0)
b:	L(0)
prod:	D
i:	L(8)

# “NEXT-USE” ALGORITHM - EXAMPLE

```
1:  t1 := 4 * i
2:  t2 := a[t1]
3:  t3 := 4 * i
4:  t4 := b[t3]
5:  t5 := t2 * t4
6:  t6 := prod + t5 ← t6:L(7)    prod:D    t5:D
7:  prod := t6      ← prod:L(0)   t6:D
8:  t7 := i + 1     ← t7:L(9)    i:D
9:  i := t7         ← i:L(10)   t7:D
10: if i <= 20 goto ← i:L(0)
```

t1:	D
t2:	D
t3:	D
t4:	D
t5:	D
t6:	L(7)
t7:	D
a:	L(0)
b:	L(0)
prod:	D
i:	L(8)



t1:	D
t2:	D
t3:	D
t4:	D
t5:	L(6)
t6:	D
t7:	D
a:	L(0)
b:	L(0)
prod:	L(6)
i:	L(8)



# “NEXT-USE” ALGORITHM - EXAMPLE

```
1:  t1 := 4 * i
2:  t2 := a[t1]
3:  t3 := 4 * i
4:  t4 := b[t3]
5:  t5 := t2 * t4  ← t5:L(6)    t2:D    t4:D
6:  t6 := prod + t5 ← t6:L(7)    prod:D   t5:D
7:  prod := t6      ← prod:L(0)   t6:D
8:  t7 := i + 1     ← t7:L(9)    i:D
9:  i := t7         ← i:L(10)   t7:D
10: if i <= 20 goto ← i:L(0)
```

t1:	D
t2:	D
t3:	D
t4:	D
t5:	L(6)
t6:	D
t7:	D
a:	L(0)
b:	L(0)
prod:	L(6)
i:	L(8)



t1:	D
t2:	L(5)
t3:	D
t4:	L(5)
t5:	D
t6:	D
t7:	D
a:	L(0)
b:	L(0)
prod:	L(6)
i:	L(8)

# “NEXT-USE” ALGORITHM - EXAMPLE

```
1:  t1 := 4 * i
2:  t2 := a[t1]
3:  t3 := 4 * i
4:  t4 := b[t3]
5:  t5 := t2 * t4
6:  t6 := prod + t5
7:  prod := t6
8:  t7 := i + 1
9:  i := t7
10: if i <= 20 goto ..
```

← t4:L(5)    b:L(0)    t3:D  
← t5:L(6)    t2:D    t4:D  
← t6:L(7)    prod:D    t5:D  
← prod:L(0)    t6:D  
← t7:L(9)    i:D  
← i:L(10)    t7:D  
← i:L(0)

```
t1:  D
t2:  L(5)
t3:  D
t4:  L(5)
t5:  D
t6:  D
t7:  D
a:   L(0)
b:   L(0)
prod: L(6)
i:   L(8)
```



```
t1:  D
t2:  L(5)
t3:  L(4)
t4:  D
t5:  D
t6:  D
t7:  D
a:   L(0)
b:   L(4)
prod: L(6)
i:   L(8)
```

# “NEXT-USE” ALGORITHM - EXAMPLE

```
1:  t1 := 4 * i
2:  t2 := a[t1]
3:  t3 := 4 * i      ← t3:L(4)    i:L(8)
4:  t4 := b[t3]      ← t4:L(5)    b:L(0)    t3:D
5:  t5 := t2 * t4     ← t5:L(6)    t2:D      t4:D
6:  t6 := prod + t5   ← t6:L(7)    prod:D     t5:D
7:  prod := t6        ← prod:L(0)   t6:D
8:  t7 := i + 1       ← t7:L(9)    i:D
9:  i := t7           ← i:L(10)   t7:D
10: if i <= 20 goto... ← i:L(0)
```

t1:	D
t2:	L(5)
t3:	L(4)
t4:	D
t5:	D
t6:	D
t7:	D
a:	L(0)
b:	L(4)
prod:	L(6)
i:	L(8)



t1:	D
t2:	L(5)
t3:	D
t4:	D
t5:	D
t6:	D
t7:	D
a:	L(0)
b:	L(4)
prod:	L(6)
i:	L(3)

# “NEXT-USE” ALGORITHM - EXAMPLE

```
1:  t1 := 4 * i
2:  t2 := a[t1]
3:  t3 := 4 * i
4:  t4 := b[t3]
5:  t5 := t2 * t4
6:  t6 := prod + t5
7:  prod := t6
8:  t7 := i + 1
9:  i := t7
10: if i <= 20 goto ...
```

← t2:L(5)    a:L(0)    t1:D  
← t3:L(4)    i:L(8)  
← t4:L(5)    b:L(0)    t3:D  
← t5:L(6)    t2:D    t4:D  
← t6:L(7)    prod:D    t5:D  
← prod:L(0)    t6:D  
← t7:L(9)    i:D  
← i:L(10)    t7:D  
← i:L(0)

```
t1:  D
t2:  L(5)
t3:  D
t4:  D
t5:  D
t6:  D
t7:  D
a:   L(0)
b:   L(4)
prod: L(6)
i:   L(3)
```



```
t1:  L(2)
t2:  D
t3:  D
t4:  D
t5:  D
t6:  D
t7:  D
a:   L(2)
b:   L(4)
prod: L(6)
i:   L(3)
```

# “NEXT-USE” ALGORITHM - EXAMPLE

```
1:  t1 := 4 * i      ← t1:L(2)   i:L(3)
2:  t2 := a[t1]      ← t2:L(5)   a:L(0)   t1:D
3:  t3 := 4 * i      ← t3:L(4)   i:L(8)
4:  t4 := b[t3]      ← t4:L(5)   b:L(0)   t3:D
5:  t5 := t2 * t4     ← t5:L(6)   t2:D     t4:D
6:  t6 := prod + t5  ← t6:L(7)   prod:D    t5:D
7:  prod := t6        ← prod:L(0)  t6:D
8:  t7 := i + 1       ← t7:L(9)   i:D
9:  i := t7           ← i:L(10)  t7:D
10: if i <= 20 goto .. ← i:L(0)
```

t1:	L(2)
t2:	D
t3:	D
t4:	D
t5:	D
t6:	D
t7:	D
a:	L(2)
b:	L(4)
prod:	L(6)
i:	L(3)



t1:	D
t2:	D
t3:	D
t4:	D
t5:	D
t6:	D
t7:	D
a:	L(2)
b:	L(4)
prod:	L(6)
i:	L(1)

# WHY LIVE VARIABLE ANALYSIS?

`x := y - 68;`

- If the *defined* variable is DEAD after this statement...  
Eliminate the statement.
- If the *defined* variable is LIVE, but has no Next-Use in this block...  
No need to keep it in a register.  
Write back to memory immediately.
- If a *used* variable is DEAD...  
We can re-use its register.

## Example:

<Assume y is in R4>

SUB 68, R4

MOV R4, x

Otherwise, use another register:

MOV R4, R5

SUB 68, R5

MOV R5, x

# CODE GENERATION ALGORITHM #2

- Generate code for each Basic Block in isolation.
- Assume that Next-Use info. is available (see previous algorithm).
- Go through the statements (in FORWARD order).
- Try to keep variables in registers...
  - Leave as long as possible in register.
  - Store back to memory only when necessary.
  - Some variables may be left in registers for several instructions.
- At the end of the basic block,  
Move all LIVE variables back to memory.

## Data Structure:

From statement to statement, we need to remember...

For each variable:

*Is it in a register? Which one?*

For each register:

*Which variable(s) does it contain, if any?*

# CODE GENERATION ALGORITHM #2

IR Code	Code Gen. Alg. #1	Code Gen. Alg. #2
t1 := 43 * a	LD a,R1 MUL 43,R1 ST R1,t1	LD a,R1 MUL 43,R1
t1 := t1 + 7	LD t1,R1 ADD 7,R1 ST R1,t1	ADD 7,R1
a := t1 * 4	LD t1,R1 MUL 4,R1 ST R1,a	ST R1,t1 MUL 4,R1 ST R1,a

*Assuming t1 is LIVE at the end of the block, we need to store it.  
If t1 is DEAD, this instruction would be omitted!*



# DATA NEEDED DURING CODE GENERATION

## Register Descriptors

For each register, which variables  
are currently stored in the register?  
Initially, all registers are  
marked EMPTY.

R0	a
R1	EMPTY
R2	x
R3	y, t1
⋮	⋮
R31	t2

---

## Variable Descriptors

For each variable, where is its  
value currently stored?

- Register(s)
- Memory
- Some combination

Initially, all variables will be  
marked in MEMORY.

a	R0
b	MEM
x	MEM, R2
y	R3
t1	R3
t2	R4, R31
t3	<nowhere>
⋮	⋮



# EXAMPLE

**IR Instructions:**      **Target Code:**

---

t1 := b + c

*DEST := R0*

**Register Descriptors**

R0	empty
R1	empty
R2	empty
R3	empty
R4	empty
R5	empty

**Variable Descriptors**

a	MEM
b	MEM
c	MEM
d	MEM
t1	-
t2	-
t3	-

} Assume DEAD  
after block

# EXAMPLE

## IR Instructions:      Target Code:

t1 := b + c

**DEST := R0**

LD    b, R0

ADD   c, R0

## Register Descriptors

R0	empty
R1	empty
R2	empty
R3	empty
R4	empty
R5	empty

## Variable Descriptors

a	MEM
b	MEM
c	MEM
d	MEM
t1	-
t2	-
t3	-

} Assume DEAD  
after block

# EXAMPLE

**IR Instructions:**      **Target Code:**

t1 := b + c

*DEST := R0*

LD    b, R0

ADD   c, R0

**Register Descriptors**

R0	t1
R1	empty
R2	empty
R3	empty
R4	empty
R5	empty

**Variable Descriptors**

a	MEM
b	MEM
c	MEM
d	MEM
t1	R0
t2	-
t3	-

} Assume DEAD  
after block

# EXAMPLE

## IR Instructions:      Target Code:

t1 := b + c

*DEST := R0*

LD    b, R0

ADD   c, R0

t2 := b \* d

*DEST := R1*

## Register Descriptors

R0	t1
R1	empty
R2	empty
R3	empty
R4	empty
R5	empty

## Variable Descriptors

a	MEM
b	MEM
c	MEM
d	MEM
t1	R0
t2	-
t3	-

} Assume DEAD  
after block

# EXAMPLE

## IR Instructions:      Target Code:

t1 := b + c

**DEST := R0**

LD    b, R0

ADD   c, R0

t2 := b \* d

**DEST := R1**

LD    b, R1

MUL   d, R1

## Register Descriptors

R0	t1
R1	empty
R2	empty
R3	empty
R4	empty
R5	empty

## Variable Descriptors

a	MEM
b	MEM
c	MEM
d	MEM
t1	R0
t2	-
t3	-

} Assume DEAD  
after block

# EXAMPLE

## IR Instructions:      Target Code:

t1 := b + c

**DEST := R0**

LD    b, R0

ADD   c, R0

t2 := b \* d

**DEST := R1**

LD    b, R1

MUL   d, R1

## Register Descriptors

R0	t1
R1	t2
R2	empty
R3	empty
R4	empty
R5	empty

## Variable Descriptors

a	MEM
b	MEM
c	MEM
d	MEM
t1	R0
t2	R1
t3	-

} Assume DEAD  
after block

# EXAMPLE

## IR Instructions:      Target Code:

t1 := b + c

*DEST := R0*

LD b, R0

ADD c, R0

t2 := b \* d

*DEST := R1*

LD b, R1

MUL d, R1

t3 := t1 \* t2

*DEST := R0*

## Register Descriptors

R0	t1
R1	t2
R2	empty
R3	empty
R4	empty
R5	empty

## Variable Descriptors

a	MEM
b	MEM
c	MEM
d	MEM
t1	R0
t2	R1
t3	-

} Assume DEAD  
after block



# EXAMPLE

## IR Instructions:      Target Code:

t1 := b + c

**DEST := R0**

LD    b, R0

ADD   c, R0

t2 := b \* d

**DEST := R1**

LD    b, R1

MUL   d, R1

t3 := t1 \* t2

**DEST := R0**

MUL   R1, R0

## Register Descriptors

R0	t1
R1	t2
R2	empty
R3	empty
R4	empty
R5	empty

## Variable Descriptors

a	MEM
b	MEM
c	MEM
d	MEM
t1	R0
t2	R1
t3	-

} Assume DEAD  
after block

# EXAMPLE

## IR Instructions:      Target Code:

t1 := b + c

**DEST := R0**

LD    b, R0

ADD   c, R0

t2 := b \* d

**DEST := R1**

LD    b, R1

MUL   d, R1

t3 := t1 \* t2

**DEST := R0**

MUL   R1, R0

## Register Descriptors

R0	t3
R1	t2
R2	empty
R3	empty
R4	empty
R5	empty

## Variable Descriptors

a	MEM
b	MEM
c	MEM
d	MEM
t1	-
t2	R1
t3	R0

} Assume DEAD  
after block

# EXAMPLE

## IR Instructions:      Target Code:

t1 := b + c

*DEST := R0*

LD   b,R0

ADD   c,R0

t2 := b \* d

*DEST := R1*

LD   b,R1

MUL   d,R1

t3 := t1 \* t2

*DEST := R0*

MUL   R1,R0

a := t3 - t2

*DEST := R0*

## Register Descriptors

R0	t3
R1	t2
R2	empty
R3	empty
R4	empty
R5	empty

## Variable Descriptors

a	MEM
b	MEM
c	MEM
d	MEM
t1	-
t2	R1
t3	R0

} Assume DEAD  
after block

# EXAMPLE

## IR Instructions:      Target Code:

t1 := b + c

**DEST := R0**

LD    b, R0

ADD   c, R0

t2 := b \* d

**DEST := R1**

LD    b, R1

MUL   d, R1

t3 := t1 \* t2

**DEST := R0**

MUL   R1, R0

a := t3 - t2

**DEST := R0**

SUB   R1, R0

## Register Descriptors

R0	t3
R1	t2
R2	empty
R3	empty
R4	empty
R5	empty

## Variable Descriptors

a	MEM
b	MEM
c	MEM
d	MEM
t1	-
t2	R1
t3	R0

} Assume DEAD  
after block

# EXAMPLE

## IR Instructions:      Target Code:

t1 := b + c

**DEST := R0**

LD b, R0

ADD c, R0

t2 := b \* d

**DEST := R1**

LD b, R1

MUL d, R1

t3 := t1 \* t2

**DEST := R0**

MUL R1, R0

a := t3 - t2

**DEST := R0**

SUB R1, R0

## Register Descriptors

R0	a
R1	t2
R2	empty
R3	empty
R4	empty
R5	empty

## Variable Descriptors

a	R0
b	MEM
c	MEM
d	MEM
t1	-
t2	R1
t3	-

} Assume DEAD  
after block

# EXAMPLE

## IR Instructions:      Target Code:

t1 := b + c

**DEST := R0**

LD b, R0

ADD c, R0

t2 := b \* d

**DEST := R1**

LD b, R1

MUL d, R1

t3 := t1 \* t2

**DEST := R0**

MUL R1, R0

a := t3 - t2

**DEST := R0**

SUB R1, R0

<End of block>

## Register Descriptors

R0	a
R1	t2
R2	empty
R3	empty
R4	empty
R5	empty

## Variable Descriptors

a	R0
b	MEM
c	MEM
d	MEM
t1	-
t2	R1
t3	-

*Assume LIVE;  
need to save*

*Assume DEAD  
after block*

# EXAMPLE

## IR Instructions:      Target Code:

t1 := b + c

**DEST := R0**

LD    b, R0

ADD   c, R0

t2 := b \* d

**DEST := R1**

LD    b, R1

MUL   d, R1

t3 := t1 \* t2

**DEST := R0**

MUL   R1, R0

a := t3 - t2

**DEST := R0**

SUB   R1, R0

<End of block>

ST    R0, a

## Register Descriptors

R0	a
R1	t2
R2	empty
R3	empty
R4	empty
R5	empty

## Variable Descriptors

a	R0
b	MEM
c	MEM
d	MEM
t1	-
t2	R1
t3	-

*Assume LIVE;  
need to save*

*Assume DEAD  
after block*

ANY QUESTION ?