Code Generation

Part II

Code Generation Algorithm #2 and #3

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

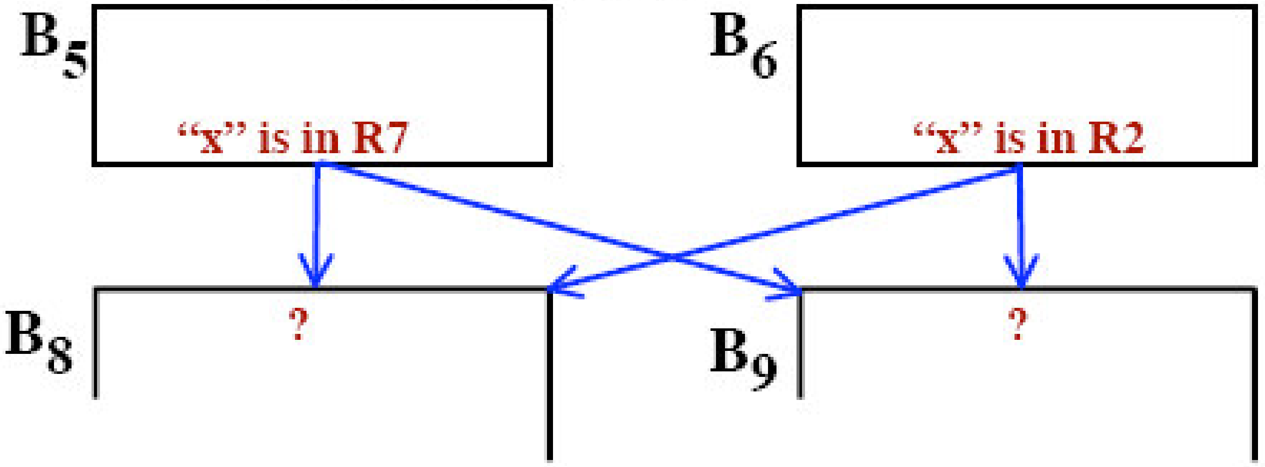
Code Generation Algorithm #2

Q: Why store <u>all</u> 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



<u>An Alternate Approach:</u>

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)

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

Option 2:

Assume every variable is live
at the end of every 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*

Definition and Use of variables

A "**Definition**" of variable x is an instruction that changes the value

```
x := ...
```

A "**Use**" of x is an instruction that reads or uses the value

```
... := ... x ...
```

```
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.

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**"?

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

Defs	Next-Uses
...	...
104	{106,109,112}
105	...
106	{109,110}
107	...
108	...
...	...

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 := y1 ⊕ y2
    Look up the current status of each variable (x, y1, y2, ...)
        Fill in the NEXT-USE info for this instruction.
    Set the status of “x” to “D”
    Set the status of “y1” to “L(n)”
    Set the status of “y2” to “L(n)”
ENDFOR
```

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

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.

“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)

$\Rightarrow$

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                  <-------- 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:	D
a:	L(0)
b:	L(0)
prod:	L(0)
i:	L(10)

$\Rightarrow$

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

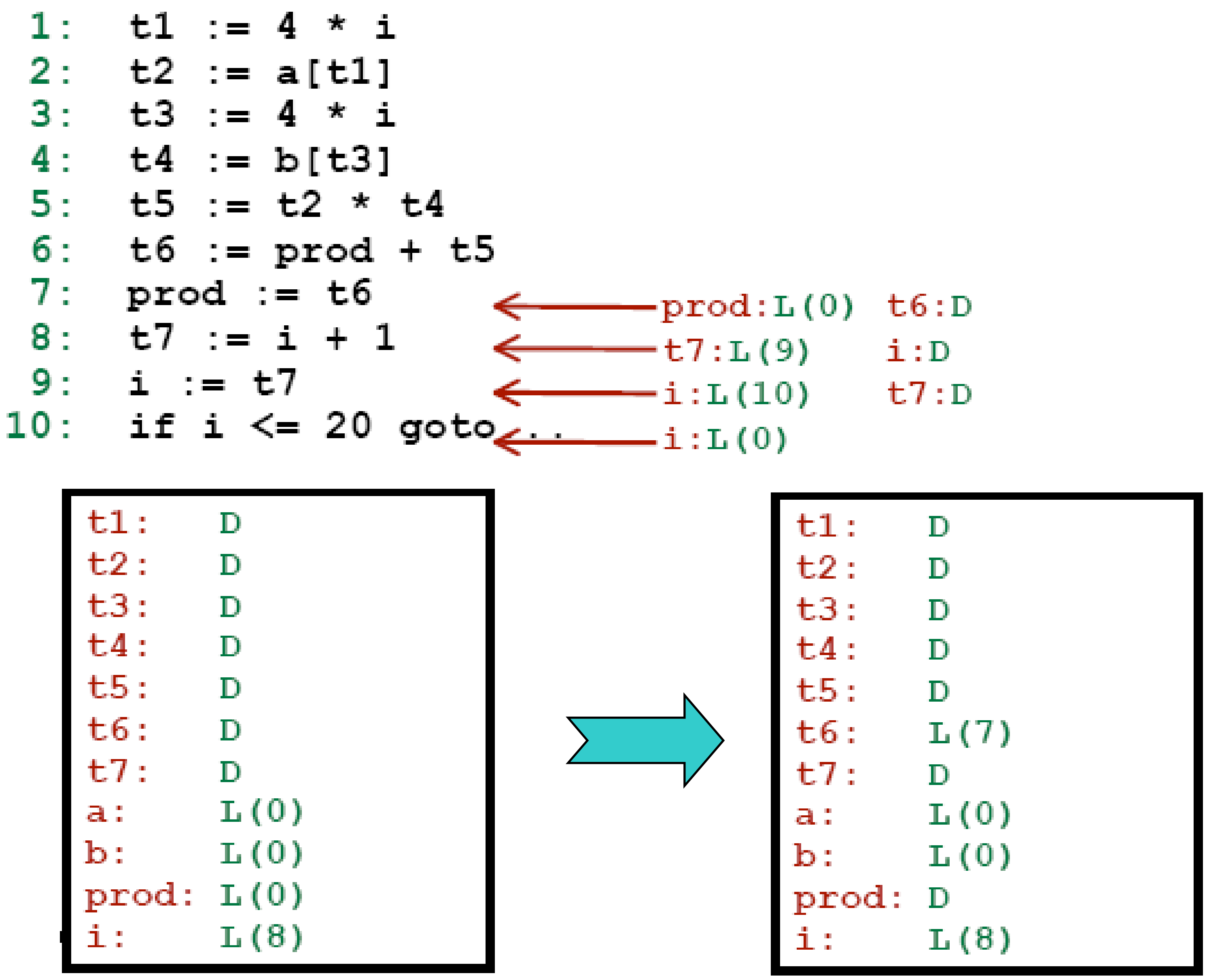
```
 1:   t1  := 4 * i
 2:   t2  := a[t1]
 3:   t3  := 4 * i
 4:   t4  := b[t3]
 5:   t5  := t2 * t4
 6:   t6  := prod + t5
 7:   prod := t6
 8:   t7  := i + 1            <------ t7:L(9)    i:D
 9:   i  := t7                <------ i:L(10)    t7:D
10:   if i <= 20 goto ...     <------ i:L(0)
```

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

$\Rightarrow$

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            <----- 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:	D
t7:	D
a:	L(0)
b:	L(0)
prod:	L(0)
i:	L(8)

$\Rightarrow$

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)

$\Rightarrow$

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]           <------ 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:	D
t4:	L(5)
t5:	D
t6:	D
t7:	D
a:	L(0)
b:	L(0)
prod:	L(6)
i:	L(8)

$\Rightarrow$

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)

$\Rightarrow$

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]              <------ 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:	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)

$\Longrightarrow$

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)

$\Rightarrow$

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>
⋮	⋮

Code Generation Algorithm #2 (Overview)

Focus on binary operators (others are similar)

```
x := y - z
x := y
if y < z goto ...
```

```
Initialize REGISTER-DESCRIPTORS to "EMPTY."
Initialize VARIABLE-DESCRIPTORS to in "MEMORY."
FOR EACH IR Statement DO
    Let x be the defined variable (if any).
    Let y and z be the used variables (if any).
    (At this point, the REGISTER-DESCRIPTORS
        and VARIABLE-DESCRIPTORS tell
        what is in regs and where the variables are stored.)

    Step 1: Determine where we will be storing the result value.
            Call it "DEST"                              DEST = "R5"
    Step 2: Move "y" into "DEST".                       LD   y,R5
    Step 3: Figure out where "z" is.
            Generate the instruction.                   SUB  z,R5
    Step 4: Update REGISTER-DESCRIPTORS
              and VARIABLE-DESCRIPTORS.
END FOR
Generate stores for all LIVE variables.                 ST   R5,x
```

Code Generation Algorithm #2

Step1: Determine where to put the result...

In Register?
In Memory?

Example IR instruction: `a := b - c`

Might generate this:

```
        SUB     ...,R7
```

Set DEST = "R7"

Or this:

```
        SUB     ...,a
```

Set DEST = "a"

Code Generation Algorithm #2

Step1: Determine where to put the result...

```
IF y is already in a register (call it $R_i$) AND
        y is DEAD after this statement AND
        $R_i$ holds no other variables THEN
    DEST := $R_i$
    Modify the descriptors to say that y is not in $R_i$ anymore.
ELSE IF any register is empty THEN
    Let DEST := $R_j$ (where $R_j$ is an empty register)
ELSE IF x has a Next-Use in this block OR
        the operator $\oplus$ requires a register for its destination THEN
    Select an occupied register; call it $R_k$
        How to choose $R_k$?
            If the vars in some reg are also in mem, no spills necessary.
            If the Next-Uses of vars in some reg are distant, choose it.
    Generate SPILL instructions, as necessary.
        Assume that REG-CONTENTS [$R_k$] = {v,w}
        Generate:    ST   $R_k$,v
                     ST   $R_k$,w
    DEST := $R_k$
ELSE
    No Next-Use in this block...
    Put the result straight into memory.
    DEST := "x"
END IF
```

Code Generation Algorithm #2

Step 2: **Determine the location of “y” and get “y” into DEST, if not already there.**

```
IF y is in a register THEN
    $LOC_y$ := $R_y$              The register containing “y”
ELSE
    $LOC_y$ := “y”                The memory address of “y”
ENDIF

IF $LOC_y \neq$ DEST THEN
    Generate the instruction:
                LD/MOV     $LOC_y$,DEST
ENDIF
```

Code Generation Algorithm #2

Step 3: Generate the instruction that performs the actual operation

Let LOC_z be the location of “z”

(If “z” is both in memory and a register, we prefer to use the register.)

Generate the instruction

```
        SUB LOCz,DEST
```

(Or whatever operation is involved)

Update the VARIABLE-DESCRIPTOR for “x”
...to show that it is in DEST only.

If DEST is a register, update its REGISTER-DESCRIPTOR
...to show that it contains only “x”.

Code Generation Algorithm #2

Step 4: **Update REGISTER-DESCRIPTORs and VARIABLE-DESCRIPTORs for “y” and “z”.**

```
IF y is in a register (call it R_y) AND
        y has no Next-Use in this block THEN
   IF y is LIVE THEN
        Generate a “SPILL” instruction
                ST     R_y,y
   END
   Modify Descriptors to say that y is no longer in any register.
END

IF z is in a register ... AND
        z has no Next-Use ...
   IF z is LIVE ...
        Generate a “SPILL” instruction
                ST      . . .
   END
   Modify ...
END
```

Same for “z”

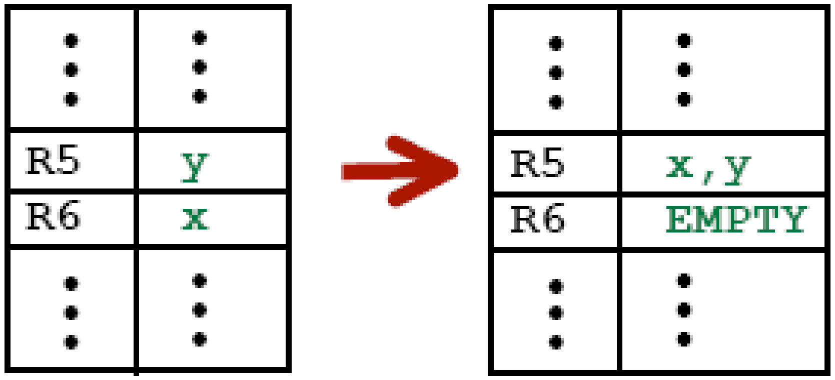
Special Case

The "assign" IR Instruction:

```
x  := y
```

If y is in a register...
Don't generate any code.
Just modify the descriptors!

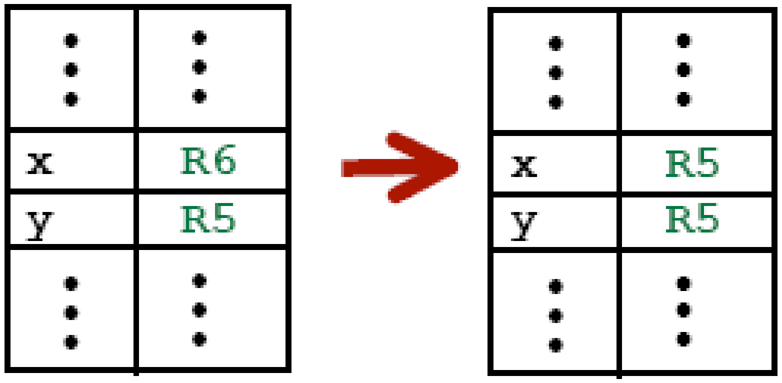
Register Descriptors



⋮	⋮
R5	y
R6	x
⋮	⋮

⋮	⋮
R5	x , y
R6	EMPTY
⋮	⋮

Variable Descriptors



⋮	⋮
x	R6
y	R5
⋮	⋮

⋮	⋮
x	R5
y	R5
⋮	⋮

Code Generation Algorithm #2

At the End of the Basic Block...

After processing all IR statements in the Basic Block...
generate "SPILL" instructions for any LIVE variables.

```
FOR each variable "x" that is LIVE
                    at the end of the Basic Block...
    Look at x's Variable Descriptor
    IF x is only in a register THEN
        Generate
                ST    R_i,x
    END
END
```

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	–

$\left.\begin{array}{l} t1 \\ t2 \\ t3 \end{array}\right\}$ *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	-

$\left.\begin{matrix}t1\\t2\\t3\end{matrix}\right\}$ *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	-

$\left.\begin{array}{l} t1 \\ t2 \\ t3 \end{array}\right\}$ *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	-	*Assume DEAD after block*
t2	R1	
t3	R0	

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

$\left.\begin{matrix}t1\\t2\\t3\end{matrix}\right\}$ *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 (t1, t2, t3)

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	*Assume LIVE; need to save*
b	MEM	
c	MEM	
d	MEM	
t1	-	*Assume DEAD after block*
t2	R1	
t3	-	

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	*Assume LIVE; need to save*
b	MEM	
c	MEM	
d	MEM	
t1	-	*Assume DEAD after block*
t2	R1	
t3	-	