CS 4240: Compilers

Lecture 16: Calling Convention (contd), Part 1 of Midterm Review

Instructor: Vivek Sarkar (vsarkar@gatech.edu)

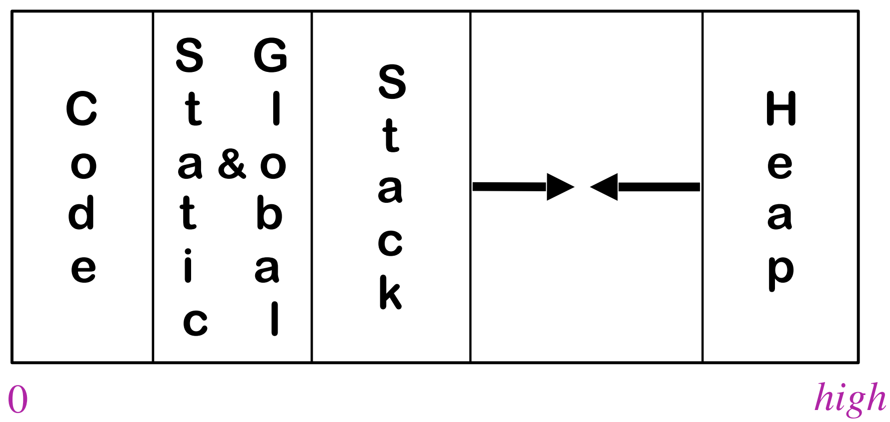
March 6, 2019

ANNOUNCEMENTS & REMINDERS

- **Project 2 due by 11:59pm on Wednesday, April 3rd**
  - 15% of course grade
- Homework 1 solution posted, along with worksheet solutions
- **MIDTERM EXAM: Wednesday, March 13, 4:30pm - 5:45pm**
  - 20% of course grade
  - Scope of exam: Lectures 1-8, 10-14 (Lecture 9 on MIPS processor is excluded)
  - Chapters 5 and 8-13 of textbook (restricted to sections covered in class)
  - **March 6th and March 11th lectures will review midterm material**
  - **Practice midterm will be released by tonight**
- FINAL EXAM: Wednesday, May 1, 2:40 pm - 5:30 pm
  - 30% of course grade

Placing Run-time Data Structures

Classic Organization



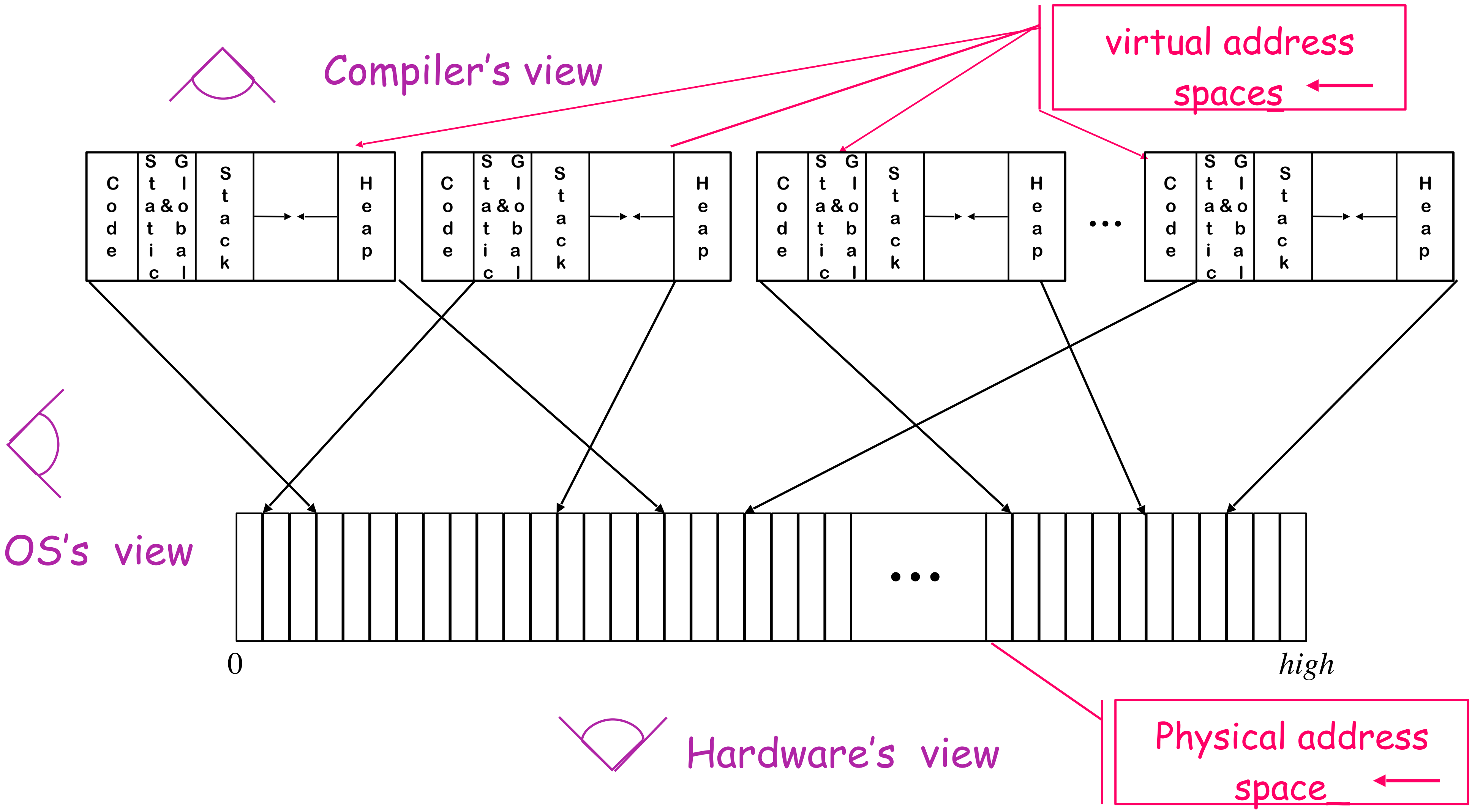
Single Logical Address Space

- Better utilization if stack & heap grow toward each other
- Very old result (Knuth)
- Code & data separate or interleaved
- Uses address space, not allocated memory

- Code, static, & global data have known size
  - Use symbolic labels in the code
- Heap & stack both grow & shrink over time
- This is a <u>virtual</u> address space

How Does This Really Work?

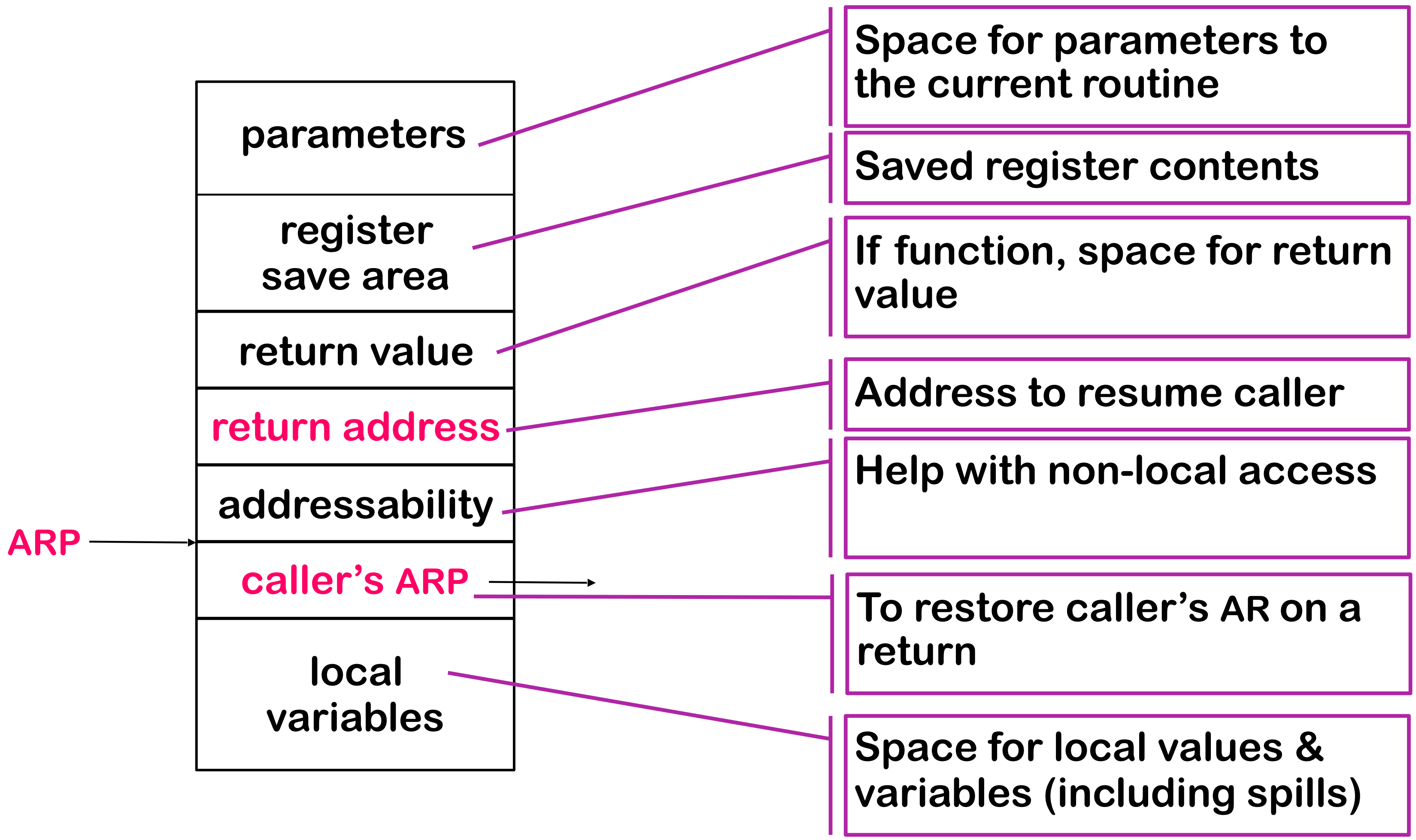
The Big Picture



On many modern processors, L1 cache uses physical addresses

L2 caches typically use virtual addresses

Activation Record Basics



One AR for each invocation of a procedure

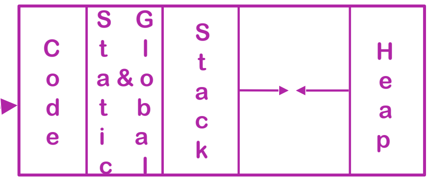
ARP ≈ Activation Record Pointer

Activation Record Details

Where do activation records live?

- If lifetime of AR matches lifetime of invocation, AND
- If code normally executes a “return”

⇒Keep ARs on a stack



Yes! This stack.

- If a procedure can outlive its caller, OR
- If it can return an object that can reference its execution state

⇒ ARs must be kept in the heap

- If a procedure makes no calls

⇒ AR can be allocated statically

Efficiency prefers static, stack, then heap

What is a Calling Convention?

- It's a **protocol** about *how* you call functions
  and *how* you are supposed to return from them

- Every CPU architecture has one
  - They can differ from one arch. to another

- 3 Reasons why ***we*** care:
  - Because it makes programming a lot easier if everyone agrees to the same consistent (i.e. reliable) methods
  - Makes **testing** a whole lot easier

Source: https://ucsb-cs64-f18.github.io/lectures/CS64_Lecture09.pdf

More on the "Why"

- Have a way of implementing functions in assembly
  - But not a clear, easy-to-use way to do <u>complex</u> functions
- In MIPS, we do not have an *inherent* way of doing **nested/recursive functions**
  - Example: Saving an *arbitrary amount* of variables
  - Example: Jumping back to a place in code *recursively*
- There <u>*is*</u> more than one way to do things
  - But we often need a <u>***convention***</u> to set **working parameters**
  - Helps facilitate things like testing and inter-compatibility
  - This is partly why MIPS has different registers for different uses

Instructions to Watch Out For

- **`jal <label>`** and **`jr $ra`** always go together
- Function *arguments* have to be stored ONLY in
  $a0 thru $a3
- Function *return values* have to be stored ONLY in
  $v0 and $v1
- If functions need additional registers *whose values we don't care about keeping after the call*, then they can use
  $t0 thru $t9
- What about **$s** registers? AKA the ***preserved registers***
  - We must save them… more on that…

The MIPS Convention In Its Essence

Preserved vs **Unpreserved** Regs

- **Preserved**: **\$s0 - \$s7**, and **\$sp , \$ra**
- **Unpreserved**: **\$t0 - \$t9**, **\$a0 - \$a3**, and **\$v0 - \$v1**

- Values held in **Preserved Regs** immediately before a function call ***MUST be the same*** immediately after the function returns.

- Values held in **Unpreserved Regs** must always be assumed to change after a function call is performed.
  - \$a0 - \$a3 are for passing arguments into a function
  - \$v0 - \$v1 are for passing values from a function

What Saves What?

- By MIPS convention, certain registers are ***designated*** to be **preserved** across a call

- Preserved registers are saved by the ***function called*** (e.g., \$s0 - \$s7)
  - So these should be saved at the start of every function

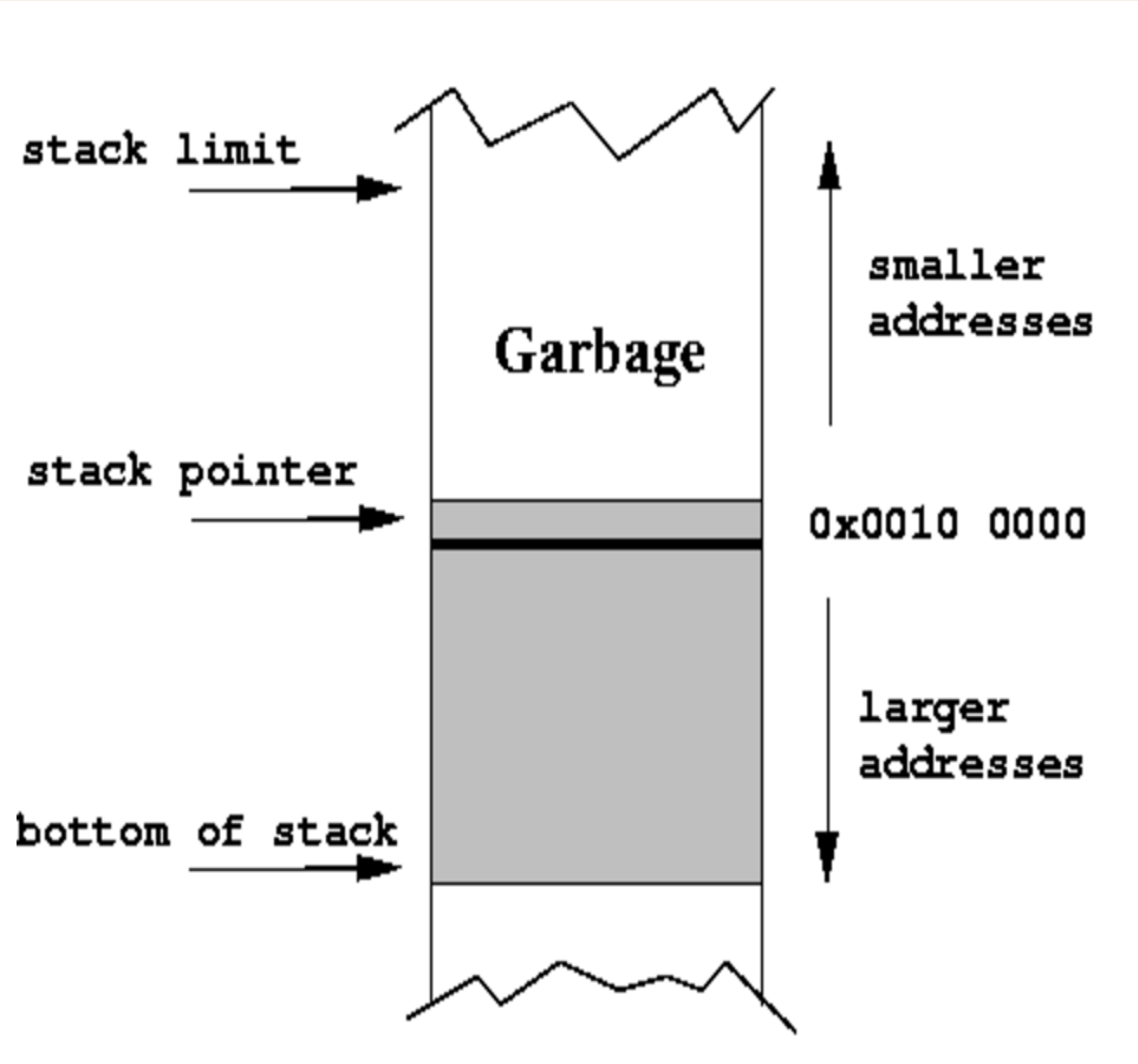
- Non-preserved registers are saved by the ***caller of the function*** (e.g., \$t0 - \$t9)
  - So these should be saved by the function's caller
  - Or not… (they can be ignored under certain circumstances)

And Where is it Saved?

- Register values are saved on the **stack**
- The top of the stack is held in **$sp** (**stackpointer**)
- The stack grows
  from high addresses *to* low addresses

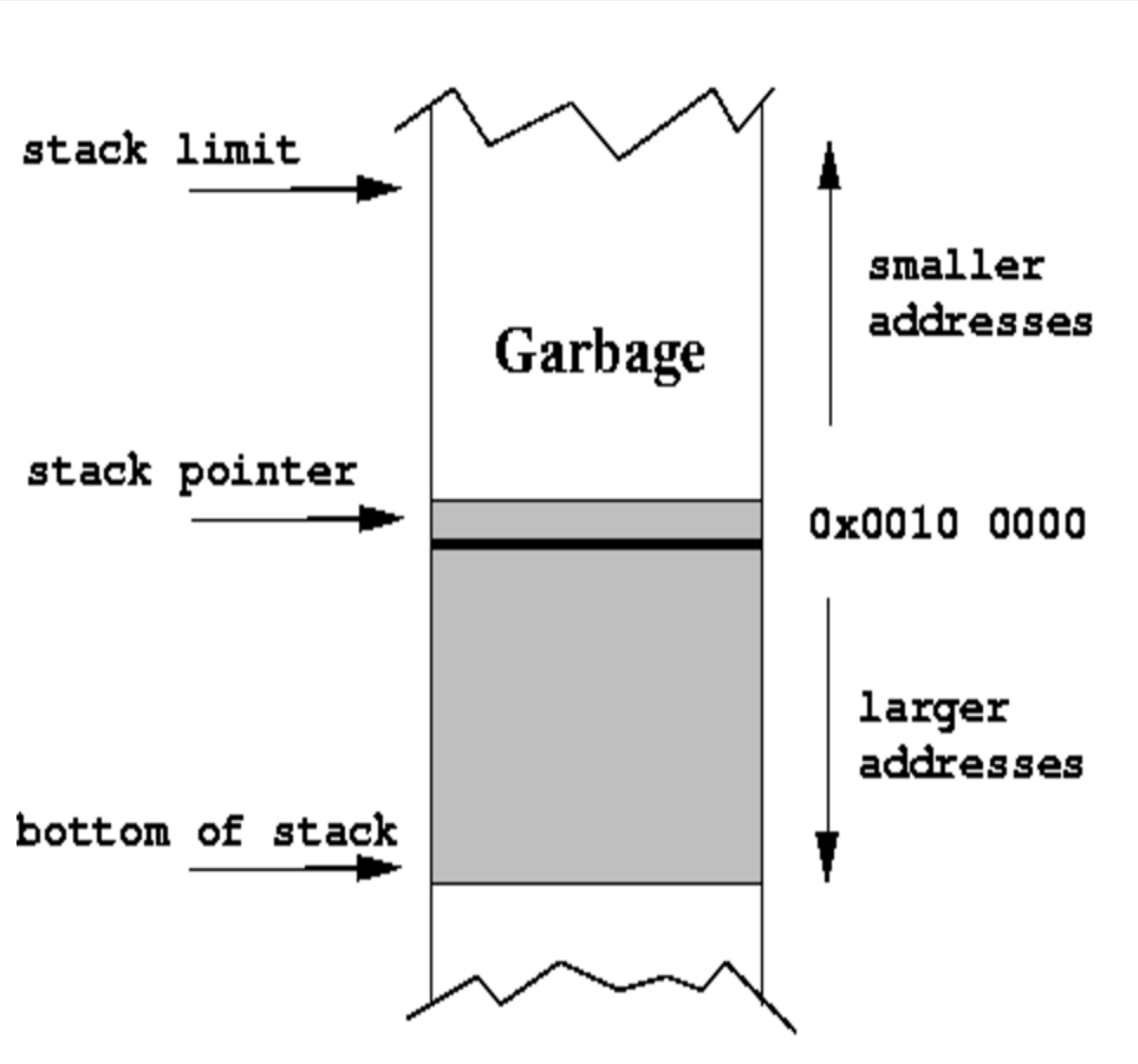
The Stack

When a program starts executing, a certain *contiguous* section of memory is set aside for the program called the **stack**.



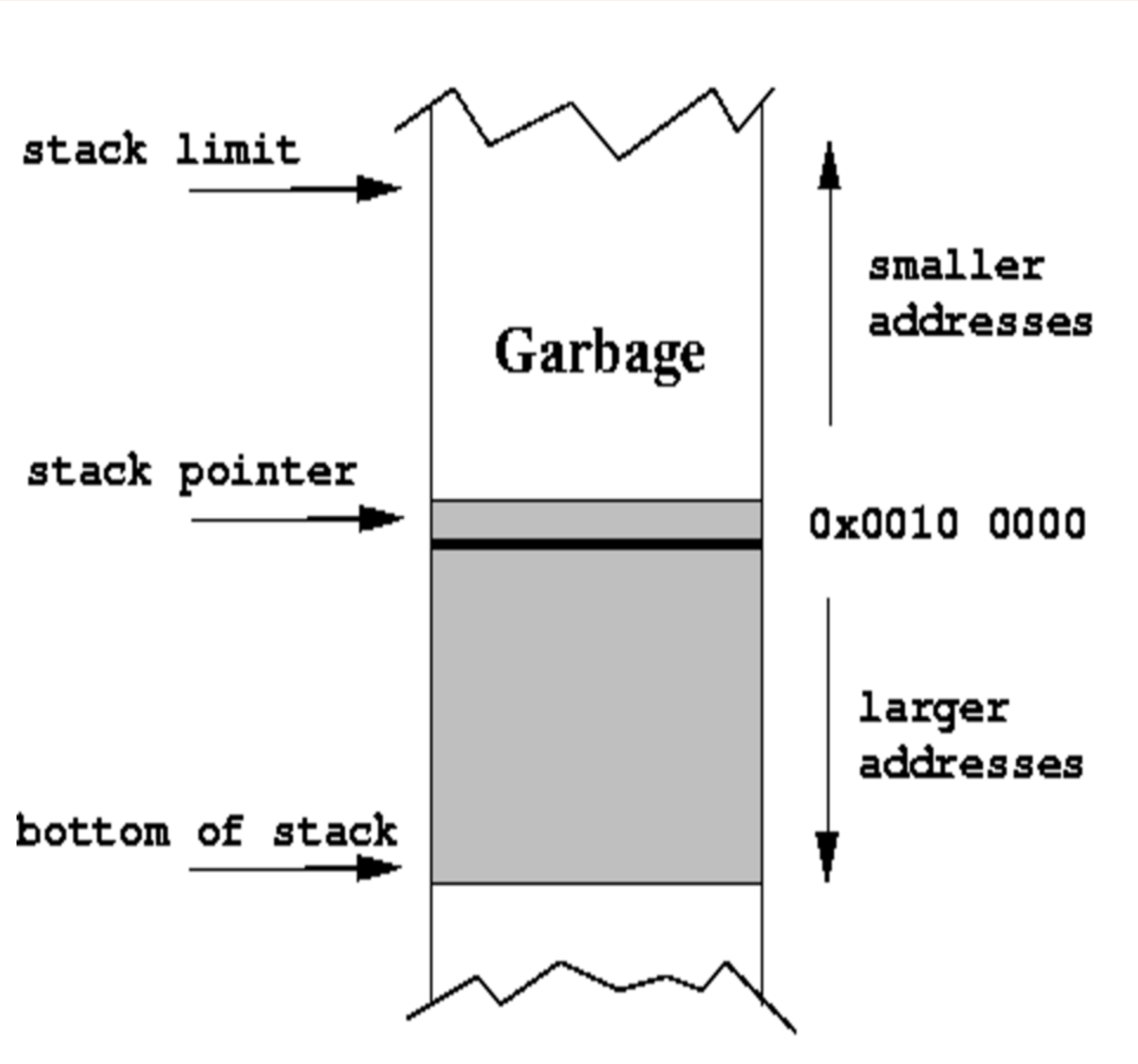
The Stack

- The **stack pointer** is a register (`$sp`) that contains the **top of the stack**.
- `$sp` contains the *smallest address* **x** such that any address smaller than **x** is considered ***garbage***, and any address greater than or equal to **x** is considered ***valid***.



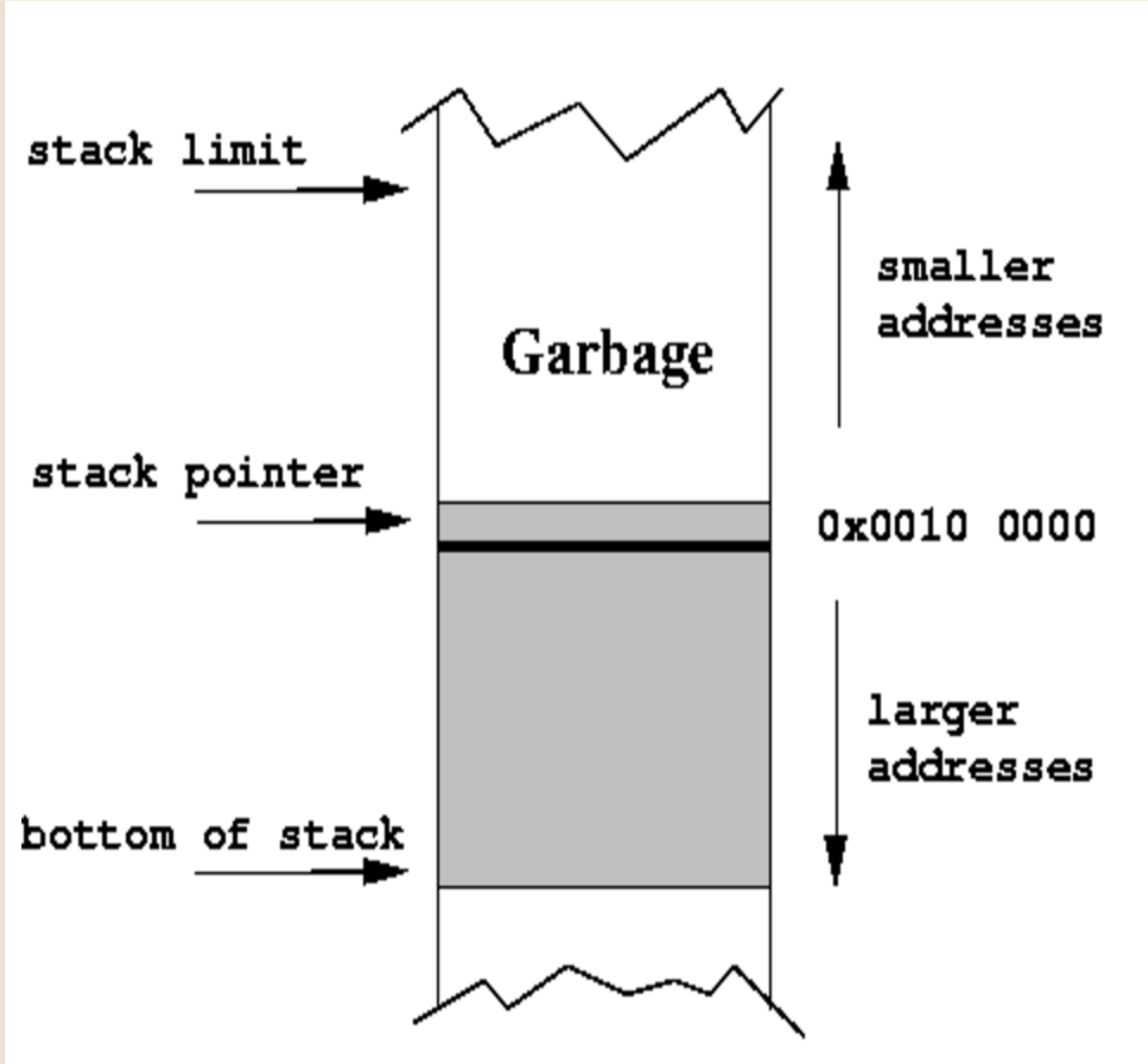
The Stack

- In this example, **`$sp`** contains the value **0x0000 1000**.
- The shaded region of the diagram represents **valid** parts of the stack.



The Stack

- **<u>Stack Bottom</u>:** The *largest* valid address of a stack.
- When a stack is initialized, $sp points to the stack bottom.
- **<u>Stack Limit</u>:** The *smallest* valid address of a stack.
- If $sp gets smaller than this, then we get a **stack overflow error**



MIPS Call Stack

- We know what a Stack is…
- A **"Call Stack"** is used for storing ***the return addresses*** of the various **functions** which have been *called*

- When you **call** a function (e.g. **`jal funcA`**), the address that we need to return to is **pushed** into the call stack.

…

funcA **does its thing… then…**

…

The function needs to return.

So, the address is **popped** off the call stack

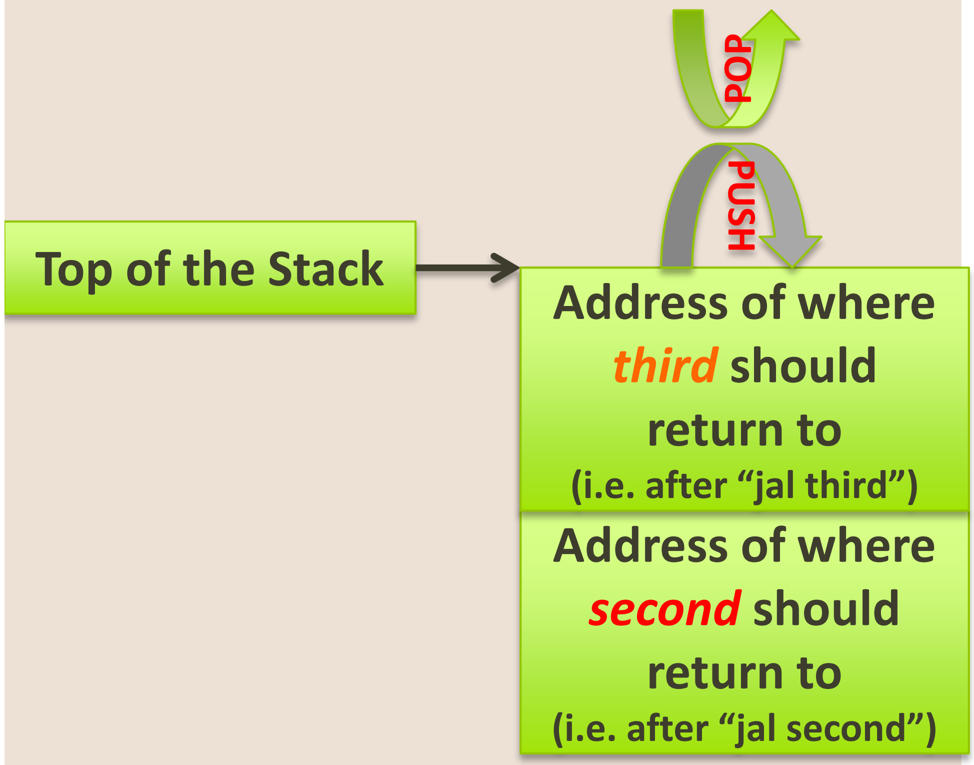
MIPS Call Stack

```
void first()
{
   second()
   return; }

void second()
{
   third ();
   return; }

void third()
{
   fourth ();
   return; }

void forth()
{
   return; }
```




```
fourth:
   jr $ra

third:
   push $ra
   jal fourth
   pop $ra
   jr $ra

second:
   push $ra
   jal third
   pop $ra
   jr $ra

first:
   jal second

li $v0, 10
syscal
```

Lecture 1 review: Types of Intermediate Representations

Three major categories

» Structural
- Graphically oriented
- Heavily used in source-to-source translators
- Tend to be large

Example:
Abstract Syntax Tree (AST)

» Linear
- Pseudo-code for an abstract machine
- Level of abstraction varies
- Simple, compact data structures
- Easier to rearrange

Examples:
3 address code
Stack machine code

» Hybrid
- Combination of graphs and linear code

Example:
Control-flow graph (CFG)

Generating 3-address Code: example

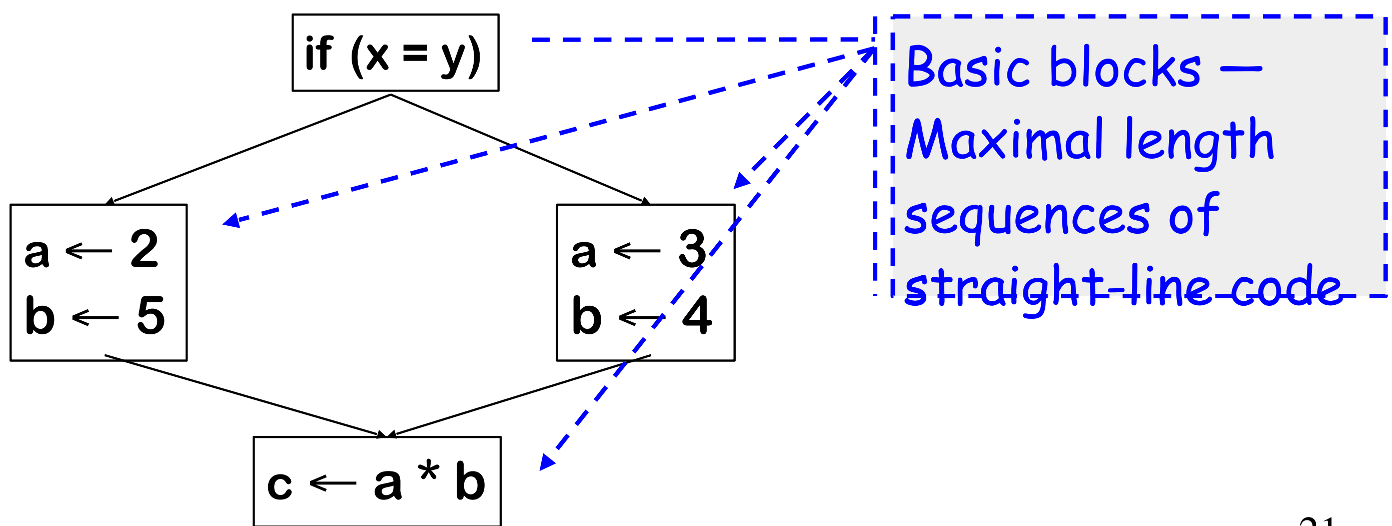
```
if (c == 0) {
   while (c < 20) {
      c = c + 2;
   }
}
else
   c = n * n + 2;
```

```
1.t1 = c == 0
2.br t1, lab1
3.t2 = n * n
4.c = t2 + 2
5.goto end
6.lab1:
7.t3 = c >= 20
8.br t3, end
9.c = c + 2
10.goto lab1
11.end:
```

Control-flow Graph

Models the transfer of control in the procedure

- Nodes in the graph are basic blocks
  - Can be represented with quads or any other linear representation
- Edges in the graph represent control flow
- Potential for exceptions can reduce basic block size in some languages, e.g., NullPointerException in Java
- Example



Dead code elimination

DEAD

- Conceptually similar to mark-sweep garbage collection
  - Mark useful operations
  - Everything not marked is useless
- Need an efficient way to find and to mark useful operations
  - Start with critical operations
  - Work back up data flow edges to find their antecedents

Define critical

- I/O statements, linkage code (entry & exit blocks), return values, calls to other procedures
- Global variables that can be visible on program exit

Dead code elimination

Mark

```
for each op i
    clear i's mark
    if i is critical then
        mark i
        add i to WorkList

while (Worklist ≠ Ø)
    remove i from  WorkList
        (i has form "x←y op z" )
    for each instruction j that
    writes to y or z
      if j is not marked then
        mark j
        add j to WorkList
```

Sweep

```
for each op i
    if i is not marked then
        delete i
```

NOTES:

1) Not all instructions that write to y or z need to be marked. We can only focus on "reaching definitions" (next lecture).

2) Branch instructions need special handling in general. A simple approach is to mark all branch instructions as critical. See textbook for more sophisticated approaches.

Consider the following source code program written in a high level programming language. Assume that multiplication is left-associative and has higher precedence than addition.

```
int w, x, y;
x = 1;
y = 2;
w = x * y * x + 7 * x
print y
```

Problem 1. Convert the program to three-address code, introducing temporary variables as necessary.

One possible solution:

$$
\begin{aligned}
&x \leftarrow 1\\
&y \leftarrow 2\\
&t_1 \leftarrow x * y\\
&t_2 \leftarrow t_1 * x\\
&t_3 \leftarrow 7 * x\\
&w \leftarrow t_2 + t_3\\
&print\ y
\end{aligned}
$$

Most students submitted correct IR for the source code, with the following variations, all of which are acceptable:

- Different students had different (legal) orderings of IR statements, e.g., some students generated IR for the left operand (`x*y*x`) first, whereas some generated IR for the right operand (`7*x`) first.
- Some students reused temporary variables to reduce the number of temporaries, whereas others created a new temporary in each instruction.

One common mistake among the (few) incorrect solutions was that some students calculated multiplication in a non-left-associative manner.

Worksheet 1, Problem 2

Problem 2. Assuming that all print instructions are critical instructions, show the output IR after dead code elimination is performed.

Solution:

$$y \leftarrow 2$$
$$print\ y$$

Almost all students answered this problem correctly. A few students were not sure what *critical instruction* meant in the context of dead code elimination. (Recall from the lecture that critical instructions are roots of computations that are necessary.)

Lecture 2: Improved Dead-code elimination algorithm

Mark

```
for each op i
    clear i's mark
    if i is critical then
        mark i
        add i to WorkList

while (Worklist ≠ Ø)
    remove i from  WorkList
        (i has form "x←y op z" )
    for each instruction j that
    writes to y (or z), and is not
    followed by a subsequent
    write of y (or z) before i
      if j is not marked then
        mark j
        add j to WorkList
```

Sweep

```
for each op i
    if i is not marked then
        delete i
```

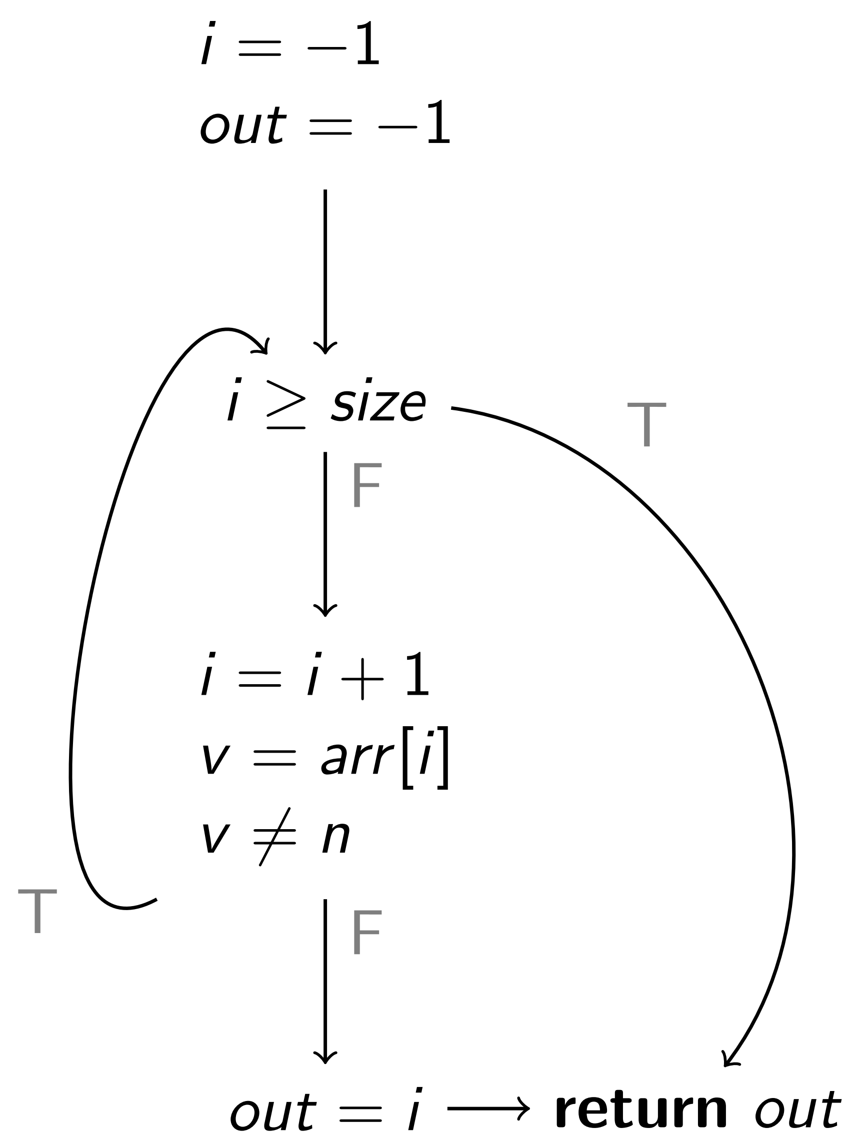
NOTES:

1) This is simple to do if there is a "straight line" control from instruction j to i, with no intervening branch instructions
2) Identifying minimum set of instructions j that contribute to inputs of instruction i is more complicated in the presence of control flow ==> ***need to build control flow graph***

Example of converting IR region to a CFG

1: $\textsc{Search'}(arr, size, n)$
2: $\quad i = -1$
3: $\quad out = -1$
4: $\quad$ **branch** $(i \geq size)$ 11
5: $\quad i = i + 1$
6: $\quad v = arr[i]$
7: $\quad$ **branch** $(v \neq n)$ 10
8: $\quad out = i$
9: $\quad$ **goto** 11
10: $\quad$ **goto** 4
11: $\quad$ **return** out

$$\{2, 4, 5, 8, 10, 11\}$$



Reaching Definitions

- Def = Write to a variable in an IR instruction
  - An IR instruction typically has a single def, but there may be exceptions, e.g., a procedure call that updates multiple global variables
- Use = Read of a variable in an IR instruction
  - It is common for an IR instruction to have more than one use
- A definition **d** reaches program point **u** if there is a control-flow path from **d** to **u** that **does not** contain an intervening definition of the same variable as **d**
  - Implies that there may be some program execution in which the value of d may reach u; this is not a requirement for all program executions
  - Definition applies to any program point u, but we will be especially interested in the case when u corresponds to a use of the variable written by d

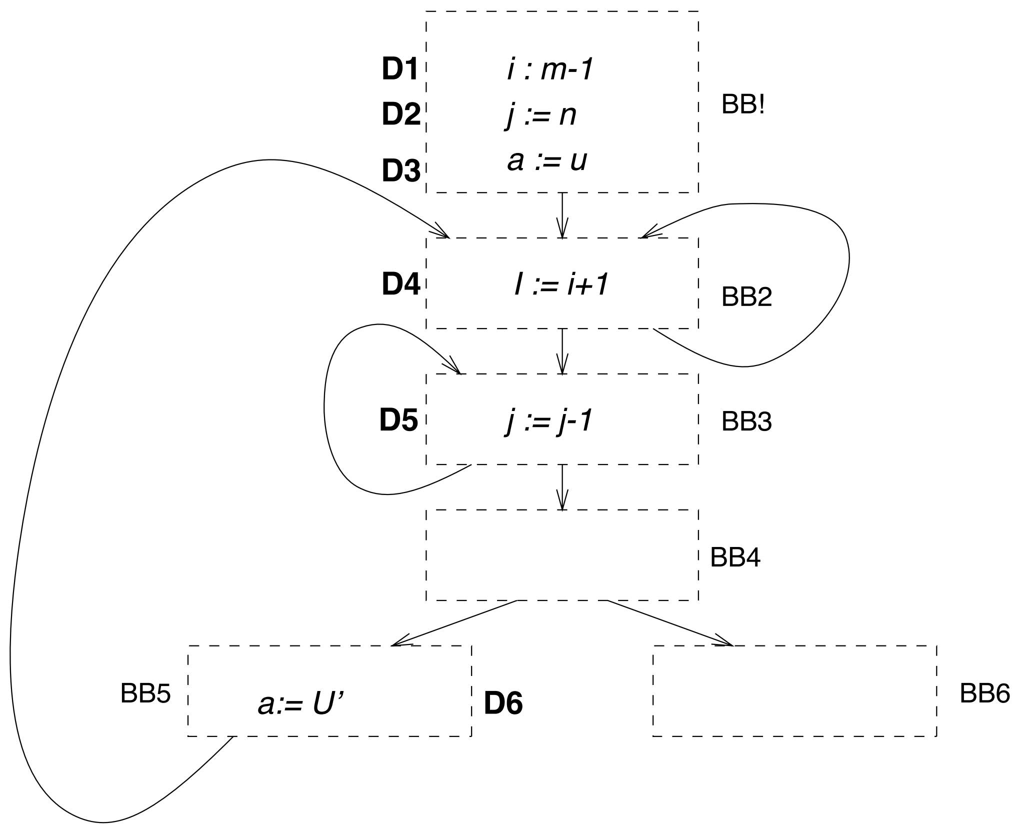
Formalizing a Solution to the Reaching Definitions Problem

- Given a statement/instruction S, define
  - Local sets that can be extracted from S
    - GEN[S] = set of definitions in S ("generated" by S)
    - KILL[S] = set of definitions that may be overwritten by S (e.g., all definitions in program that write to S's lval, whether or not they reach S)
  - Global sets to be computed using CFG
    - IN[S] = set of definitions that reach the entry point of S
    - OUT[S] = set of definitions in S as well as definitions from IN[S] that go beyond S (are not "killed" by S)
- Data flow equations (invariants) for these sets

$$\text{OUT[S]} = \text{GEN[S]} \cup (\text{IN[S]} - \text{KILL[S]})$$

$$\text{IN[S]} = \cup_{p \in predecessors} \text{OUT[p]}$$

Example



- **D1** reaches **D4** but *not* beyond; why?
  Think of the "kill" sets of **D4**
- **D4** reaches itself due to cyclic dependences in the control-flow
- **D1** reaches **D6** and so on

Worksheet-2 Solution

(From Lecture 2 given on 01/09/2019)

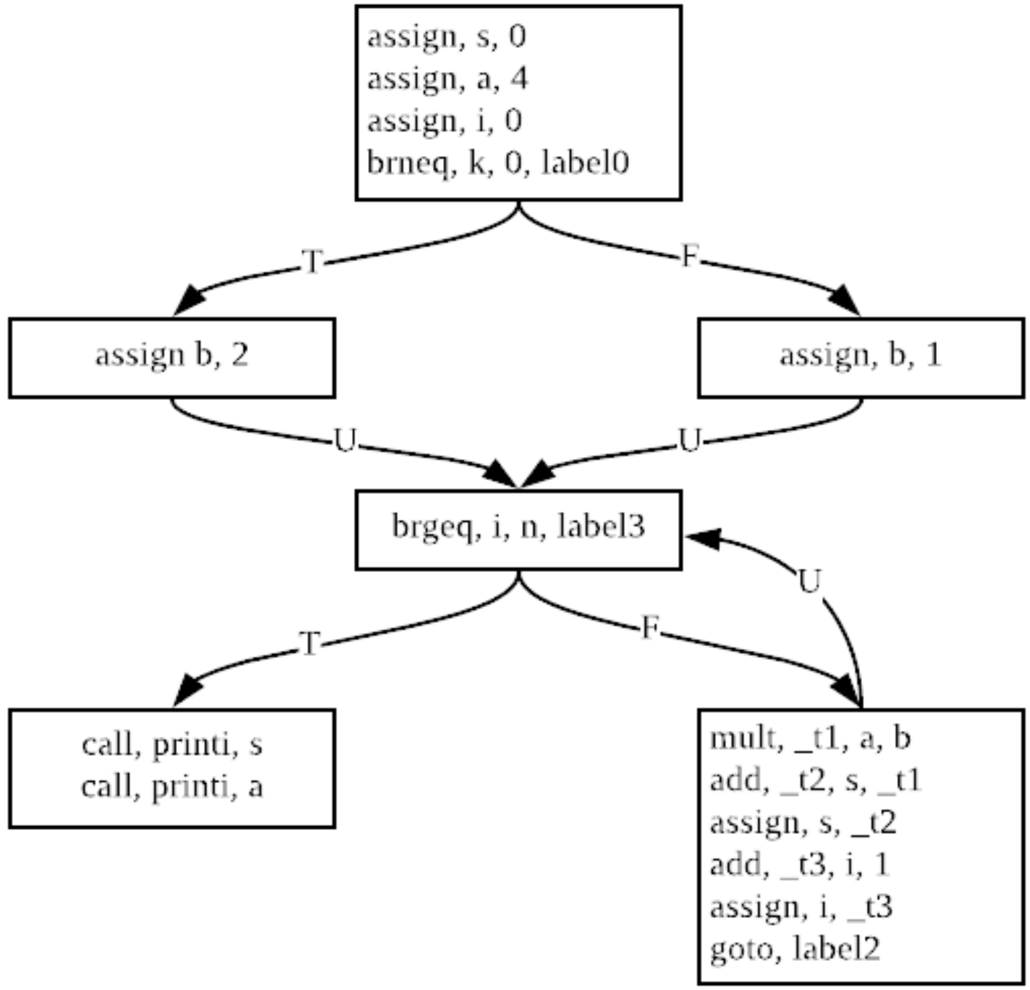
Q1

- Construct a Control Flow Graph for the IR (Intermediate Representation) segment shown below, and draw it on the right of the IR. Each vertex can be a single IR instruction, or a basic block containing of a straight-line sequence of multiple IR instructions.

```
   assign s, 0
   assign a, 4
   assign i, 0
// branch if (arg1 != arg2)
   brneq k, 0, label0
   assign, b, 1
   goto, label1
label0:
   assign, b, 2
label1:
label2:
// branch if (arg1 >= arg2)
   brgeq, i, n, label3
   mult, _t1 a, b
   add, _t2, s, _t1
   assign s, _t2
   add _t3, i, 1
   assign i, _t3
   goto, label2
label3:
   call, printi, s
   call, printi, a
```

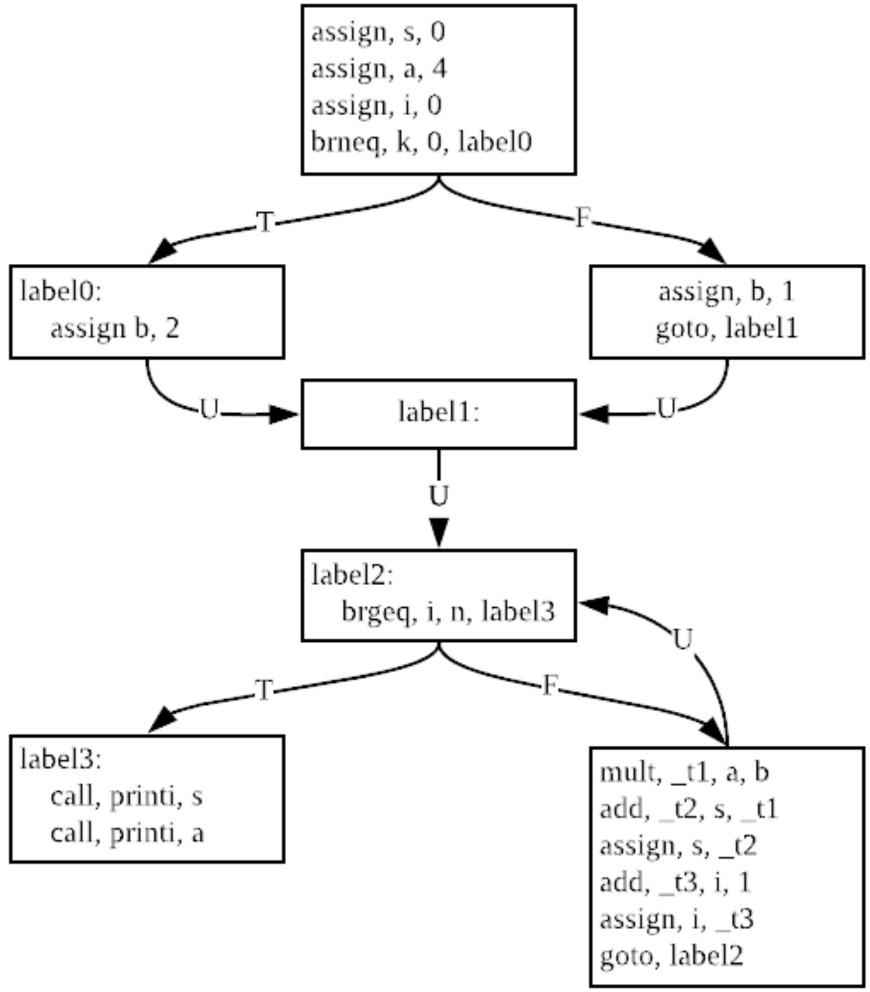
Q1 Sample Solution1: labels are not instructions

```
  assign s, 0
  assign a, 4
  assign i, 0
// branch if (arg1 != arg2)
  brneq k, 0, label0
  assign, b, 1
  goto, label1
label0:
  assign, b, 2
label1:
label2:
// branch if (arg1 >= arg2)
  brgeq, i, n, label3
  mult, _t1 a, b
  add, _t2, s, _t1
  assign s, _t2
  add _t3, i, 1
  assign i, _t3
  goto, label2
label3:
  call, printi, s
  call, printi, a
```



Q1 Sample Solution2: labels are no-op instructions

```
  assign s, 0
  assign a, 4
  assign i, 0
// branch if (arg1 != arg2)
  brneq k, 0, label0
  assign, b, 1
  goto, label1
label0:
  assign, b, 2
label1:
label2:
// branch if (arg1 >= arg2)
  brgeq, i, n, label3
  mult, _t1 a, b
  add, _t2, s, _t1
  assign s, _t2
  add _t3, i, 1
  assign i, _t3
  goto, label2
label3:
  call, printi, s
  call, printi, a
```



Q2, Q3

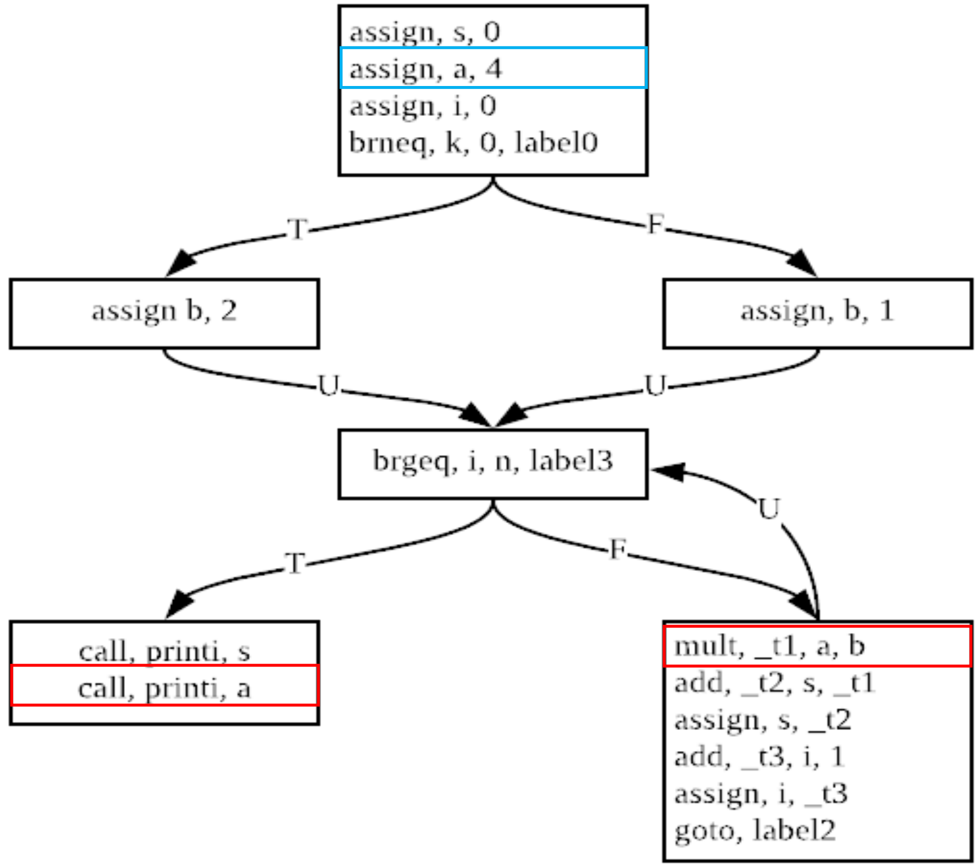
- Q2
  Which uses of ***a*** are reached by the def of ***a*** in line 2?

- Q3
  Which defs of ***b*** reach the use of ***b*** in line 12?

```
1    assign s, 0
2    assign a, 4
3    assign i, 0
// branch if (arg1 != arg2)
4    brneq k, 0, label0
5    assign, b, 1
6    goto, label1
7  label0:
8    assign, b, 2
9  label1:
10 label2:
// branch if (arg1 >= arg2)
11   brgeq, i, n, label3
12   mult, _t1 a, b
13   add, _t2, s, _t1
14   assign s, _t2
15   add _t3, i, 1
16   assign i, _t3
17   goto, label2
18 label3:
19   call, printi, s
20   call, printi, a
```

Q2 Solution

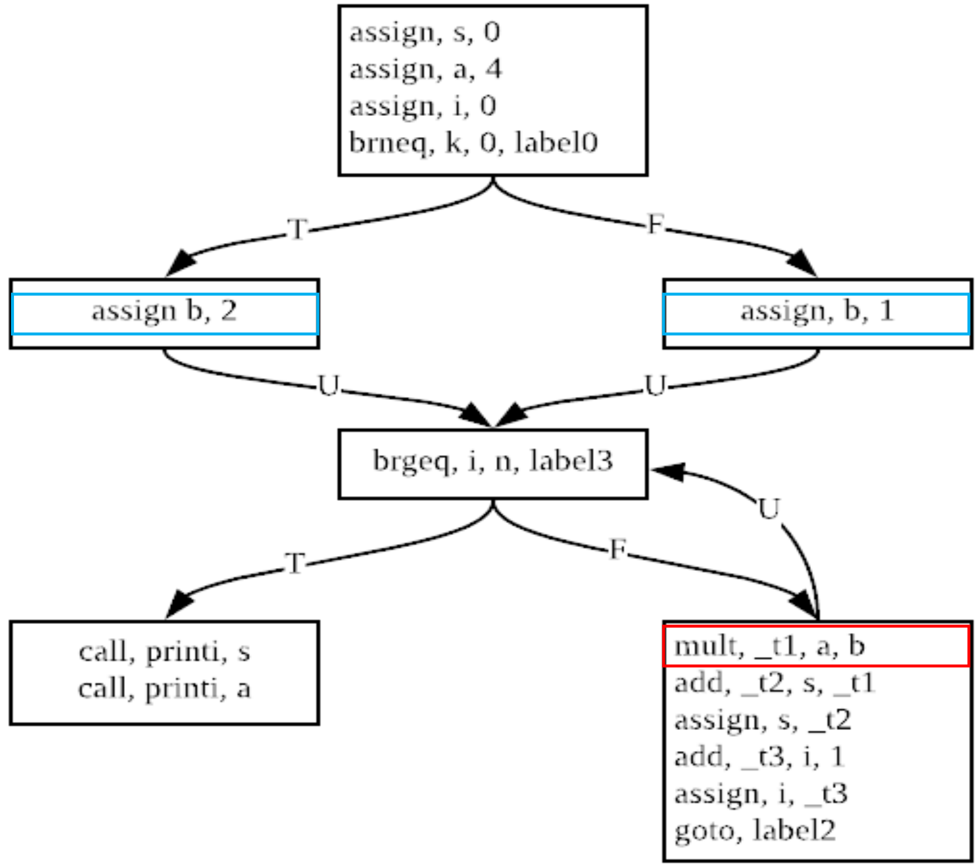
```
1     assign s, 0
2     assign a, 4
3     assign i, 0
// branch if (arg1 != arg2)
4     brneq k, 0, label0
5     assign, b, 1
6     goto, label1
7   label0:
8     assign, b, 2
9   label1:
10  label2:
// branch if (arg1 >= arg2)
11    brgeq, i, n, label3
12    mult, _t1 a, b
13    add, _t2, s, _t1
14    assign s, _t2
15    add _t3, i, 1
16    assign i, _t3
17    goto, label2
18  label3:
19    call, printi, s
20    call, printi, a
```



In line 2, variable ‘a’ is defined as 4. <u>The definition of ‘a’ in line 2 reaches line **12 & 20** without any intervening ‘def of a’.</u>

Q3 Solution

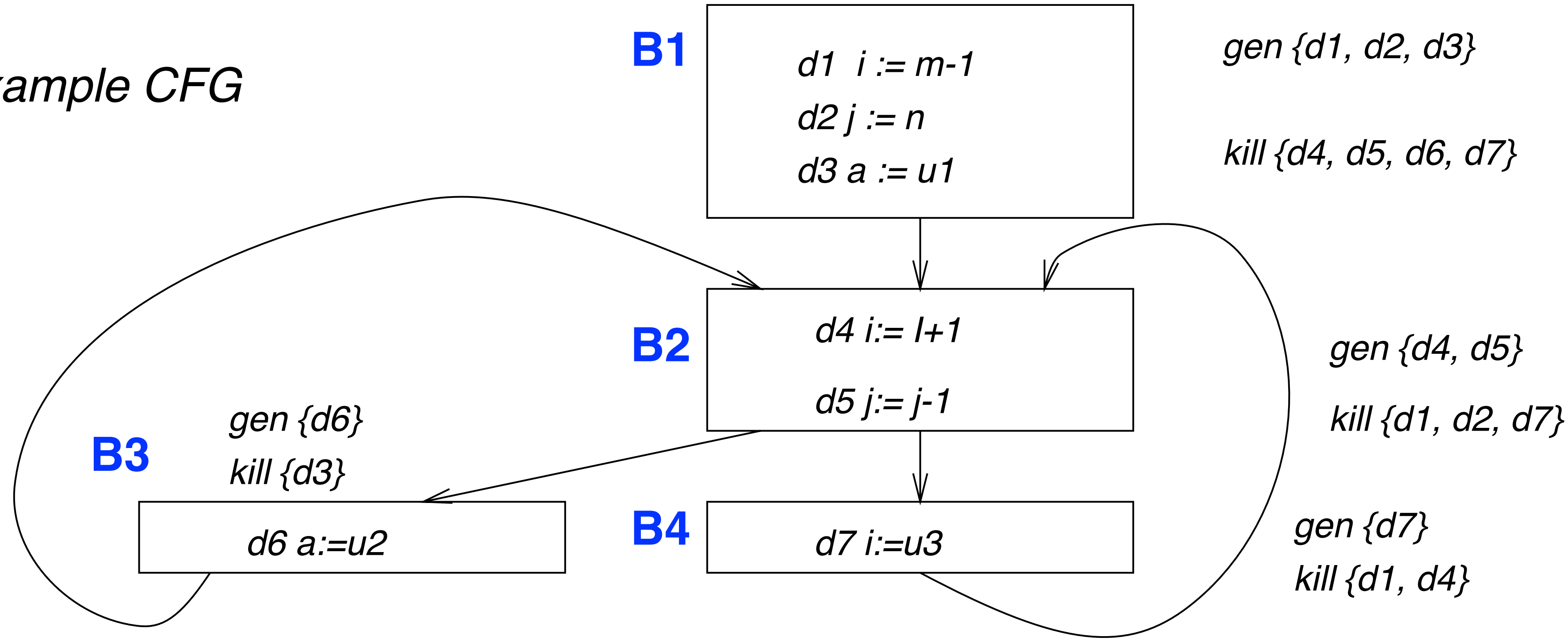
```
1     assign s, 0
2     assign a, 4
3     assign i, 0
// branch if (arg1 != arg2)
4     brneq k, 0, label0
5     assign, b, 1
6     goto, label1
7   label0:
8     assign, b, 2
9   label1:
10  label2:
// branch if (arg1 >= arg2)
11    brgeq, i, n, label3
12    mult, _t1 a, b
13    add, _t2, s, _t1
14    assign s, _t2
15    add _t3, i, 1
16    assign i, _t3
17    goto, label2
18  label3:
19    call, printi, s
20    call, printi, a
```



There is no intervening 'def of b' in the control flow between line 5 and line 12. Same for line 8 and line 12. The defs of b in lines **5 & 8** both reach the use of b in line 12.

Lecture 3: Data Flow Equations are Recursive!

Example CFG



OUT[B1] = GEN[B1] U (IN[B1] - KILL[B1])
IN[B1] = { } // empty set

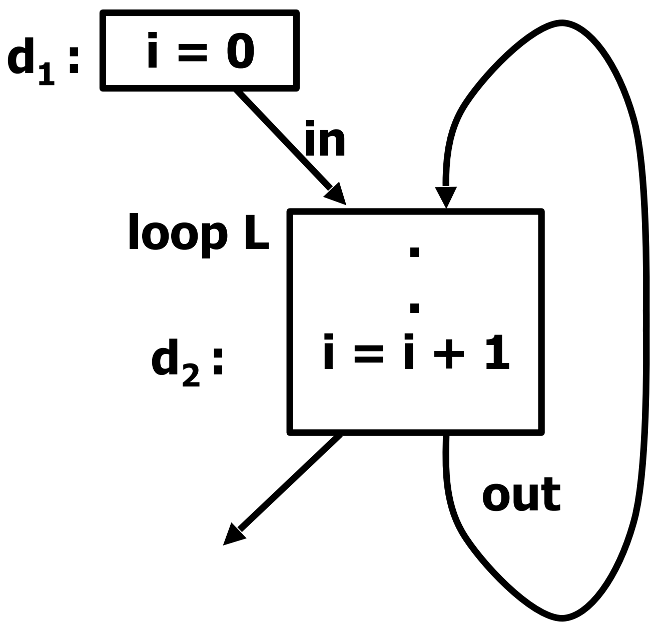
OUT[B2] = GEN[B2] U (IN[B2] - KILL[B2])
IN[B2] = OUT[B1] U OUT[B3] U OUT[B4]

OUT[B3] = GEN[B3] U (IN[B3] - KILL[B3])
IN[B3] = OUT[B2]

OUT[B4] = GEN[B4] U (IN[B4] - KILL[B4])
IN[B4] = OUT[B2]

Reaching Definitions as an example of Data Flow Analysis

Data Flow Analysis = finding solution to recursive data flow equations



Question:

What is the set of reaching definitions at the exit of the loop L?

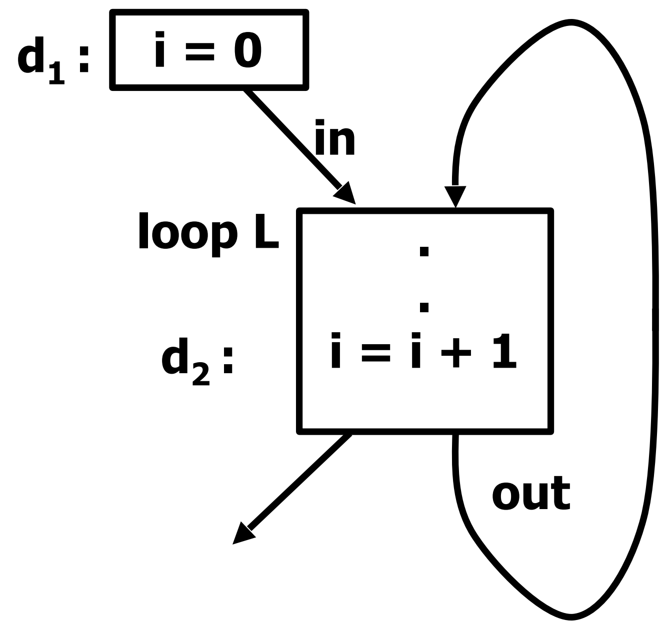
in [L] = $\{d_1\} \cup$ out[L]
gen [L] = $\{d_2\}$
kill [L] = $\{d_1\}$
out [L] = gen [L] $\cup$ {in [L] - kill[L]}

in[L] depends on out[L], and out[L] depends on in[L]!!

Solution?

Initialization

$out[L] = \emptyset$



First iteration

$in[L] = \{d_1\} \cup out[L]$

$= \{d_1\}$

$out[L] = gen\ [L] \ \cup \ (in\ [L] - kill\ [L])$

$= \{d_2\} \cup (\{d_1\} - \{d_1\})$

$= \{d_2\}$

in [L] = $\{d_1\} \cup$ out[L]
gen [L] = $\{d_2\}$
kill [L] = $\{d_1\}$
out [L] = gen [L] $\cup$ {in [L] - kill[L]}

Solution

First iteration

$out[L] = \{d_2\}$

Second iteration

$in[L] = \{d_1\} \cup out[L]$

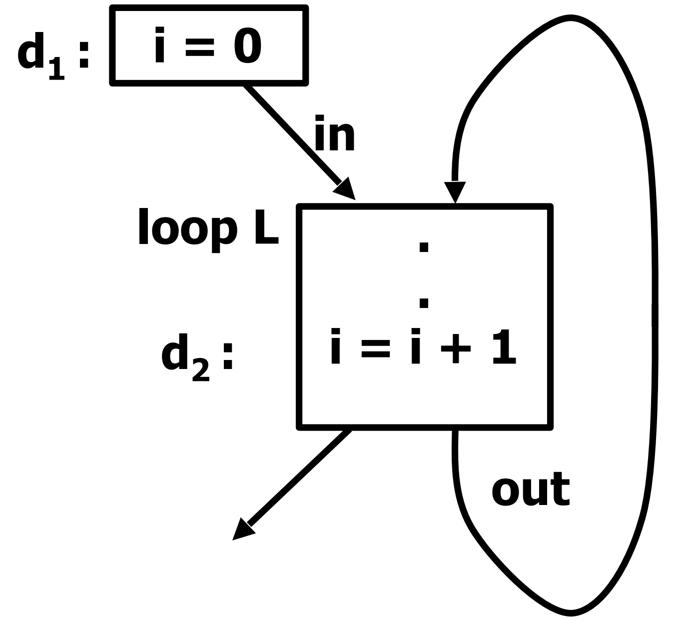
$= \{d_1, d_2\}$

$out[L] = gen\ [L] \cup (in\ [L] - kill\ [L])$

$= \{d_2\} \cup \{\{d_1, d_2\} - \{d_1\}\}$

$= \{d_2\} \cup \{d_2\}$

$= \{d_2\}$



We reached the fixed point!

$in\ [L] = \{d_1\} \cup out[L]$
$gen\ [L] = \{d_2\}$
$kill\ [L] = \{d_1\}$
$out\ [L] = gen\ [L] \cup \{in\ [L] - kill[L]\}$

Algorithm Summary: Inputs and Outputs

- *Input:* A flow graph for which $kill[B]$ and $gen[B]$ have been computed for each basic block B
- *Output:* $in[B]$ and $out[B]$ for each block B
- The Idea: Use an iterative approach where the "initial" in and out information is *propagated* across edges and along the paths of the graph *until* none of the outs change

  All computation is at the granularity of basic-blocks

Algorithm Summary: Overall Steps

Reaching Definitions:

- // Initialize out under the assumpt1¡ion that $in = \emptyset$ by setting $out[B] := gen[B]$ for all the blocks //
- $change$:= **true**

  // This initiates the iteration and if there is a change after the iteration in *any* of the out sets, then it remains true//
- While $change$ remains **true** compute
  - $in[B] = \cup_{p \in P} out[p]$ where P is the set of all predecessors of block B
- $tempout := out[B]$
- $out[B] := gen[B] \cup (in[B] - kill[B])$
- if $out[B] \neq temoput\ change$:= **true**

Lecture 4: Using Reaching Definitions to improve Dead-code Elimination algorithm

Mark

```
for each op i
    clear i's mark
    if i is critical then
        // for simplicity, assume all
        // branch instructions are critical
        mark i
        add i to WorkList

while (Worklist ≠ Ø)
    remove i from  WorkList
        (i has form "x ← op y" or
         "x←y op z" )
    for each instruction j that
    contains a def of y or z that
    reaches i
        if j is not marked then
            mark j
            add j to WorkList
```

Sweep

```
for each op i
    if i is not marked then
        delete i
```

NOTES:

- A def reaches instruction i
  1) if it is in the IN set for the basic block B(i) containing i, and
  2) the def is not killed locally within B(i) before instruction i
- Condition 2) above can be omitted if reaching definitions analysis is performed on an instruction-level CFG
- Additional smarts are needed to also avoid marking branch instructions as critical

Redundancy Elimination as an Example

An expression x+y is redundant if and only if, along every path from the procedure's entry, it has been evaluated, and its constituent subexpressions (x & y) have not been re-defined.

If the compiler can prove that an expression is redundant

- It can preserve the results of earlier evaluations
- It can replace the current evaluation with a reference

Two pieces to the problem

- Proving that x+y is redundant, or available
- Rewriting the code to eliminate the redundant evaluation

One technique for accomplishing both is called value numbering

Local Value Numbering

The Algorithm

For each operation o = <operator, o_1, o_2> in a basic block, in order

1 Get value numbers for operands from hash lookup

2 Hash <operator,VN(o_1),VN(o_2)> to get a value number for o

3 If o already had a value number, replace o with a reference

4 If o_1 & o_2 are constant, evaluate it & replace with a `loadI`

If hashing behaves, the algorithm runs in linear time

Handling algebraic identities

- Case statement on operator type
- Handle special cases within each operator

Local Value Numbering

An example (superscripts are value numbers, and are not part of the IR)

Original Code

$a \leftarrow x + y$
* $b \leftarrow x + y$
$a \leftarrow 17$
* $c \leftarrow x + y$

With VNs

$a^3 \leftarrow x^1 + y^2$
* $b^3 \leftarrow x^1 + y^2$
$a^4 \leftarrow 17$
* $c^3 \leftarrow x^1 + y^2$

Rewritten

$a^3 \leftarrow x^1 + y^2$
* $b^3 \leftarrow a^3$
$a^4 \leftarrow 17$
* $c^3 \leftarrow a^3$ (oops!)

Two redundancies

- Eliminate stmts with a *
- Coalesce results ?

Corrected

$t \leftarrow x^1 + y^2$
$a \leftarrow t$
* $b^3 \leftarrow a^3$
$a^4 \leftarrow 17$
* $c^3 \leftarrow t$

Options

- Use $c^3 \leftarrow b^3$
- Save a^3 in t^3
- Rename around it (next slide)
- Introduce a temporary (corrected code on left)
- . . .

Local Value Numbering

The LVN Algorithm, with bells & whistles

Block is a sequence of n operations of the form

$T_i \leftarrow L_i \ Op_i \ R_i$

for i ← 0 to n-1

1. get the value numbers V_1 and V_2 for L_i and R_i
2. if L_i and R_i are both constant then

   evaluate Li Op_i R_i, assign it to T_i, and mark T_i as a constant

   Constant folding
3. if Li Op_i R_i matches an identity then

   replace it with a copy operation or an assignment

   Algebraic identities
4. if Op_i commutes and $V_1 > V_2$ then

   swap V_1 and V_2

   Commutativity
5. construct a hash key $<V_1, Op_i, V_2>$
6. if the hash key is already present in the table then

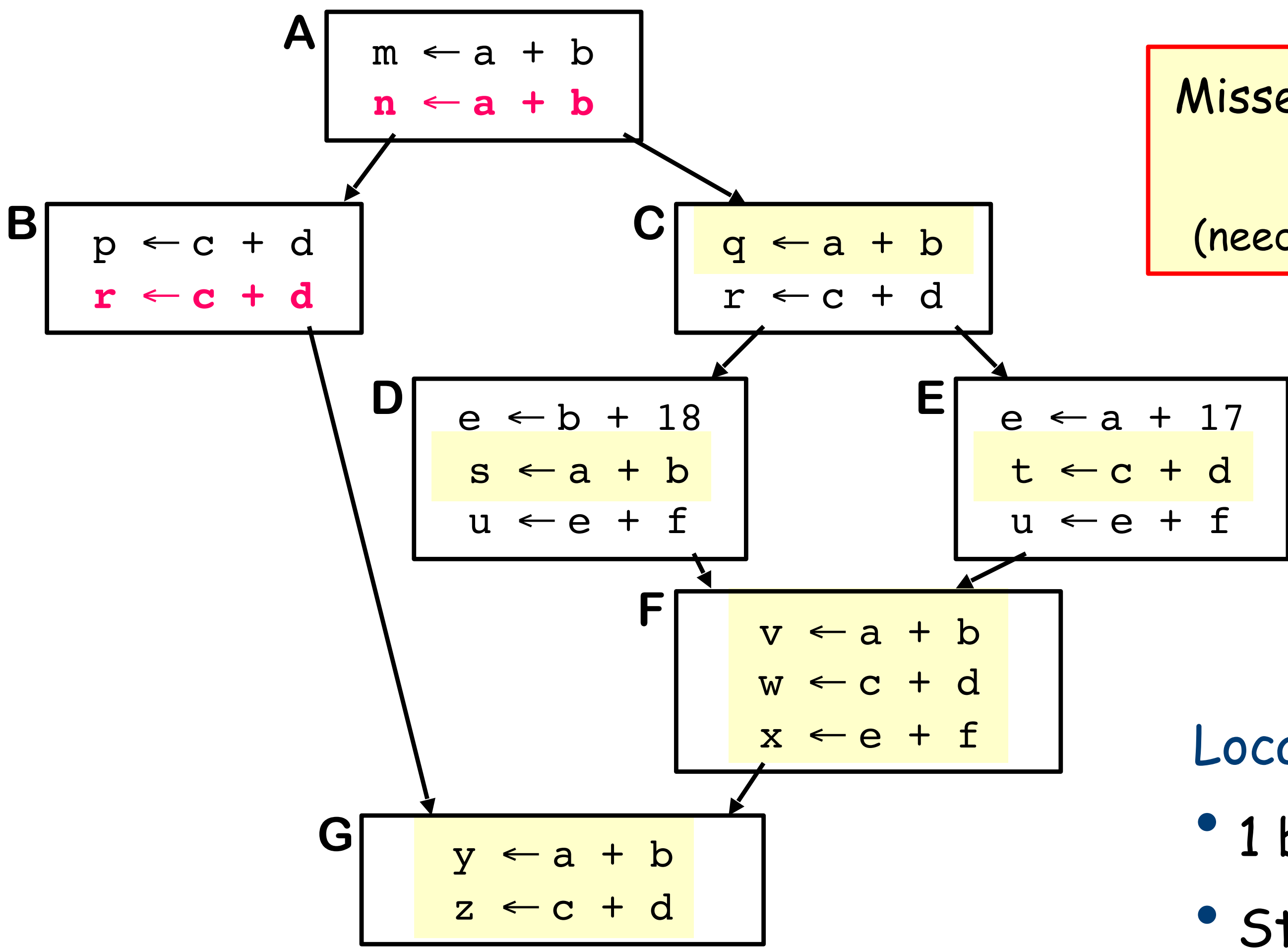
   replace operation I with a copy into T_i and mark T_i with the VN

   else

   insert a new VN into table for hash key & mark T_i with the VN

Limitations of Local Value Numbering

LVN finds redundant ops in red



Missed opportunities in yellow

(need stronger methods)

Local Value Numbering

- 1 block at a time
- Strong local results
- No cross-block effects

Scope of Optimization

A basic block is a sequence of straight-line code.

Local optimization

- Operates entirely within a single basic block
- Properties of block lead to strong optimizations

Regional optimization

- Operate on a region in the CFG that contains multiple blocks
- Loops, trees, paths, extended basic blocks

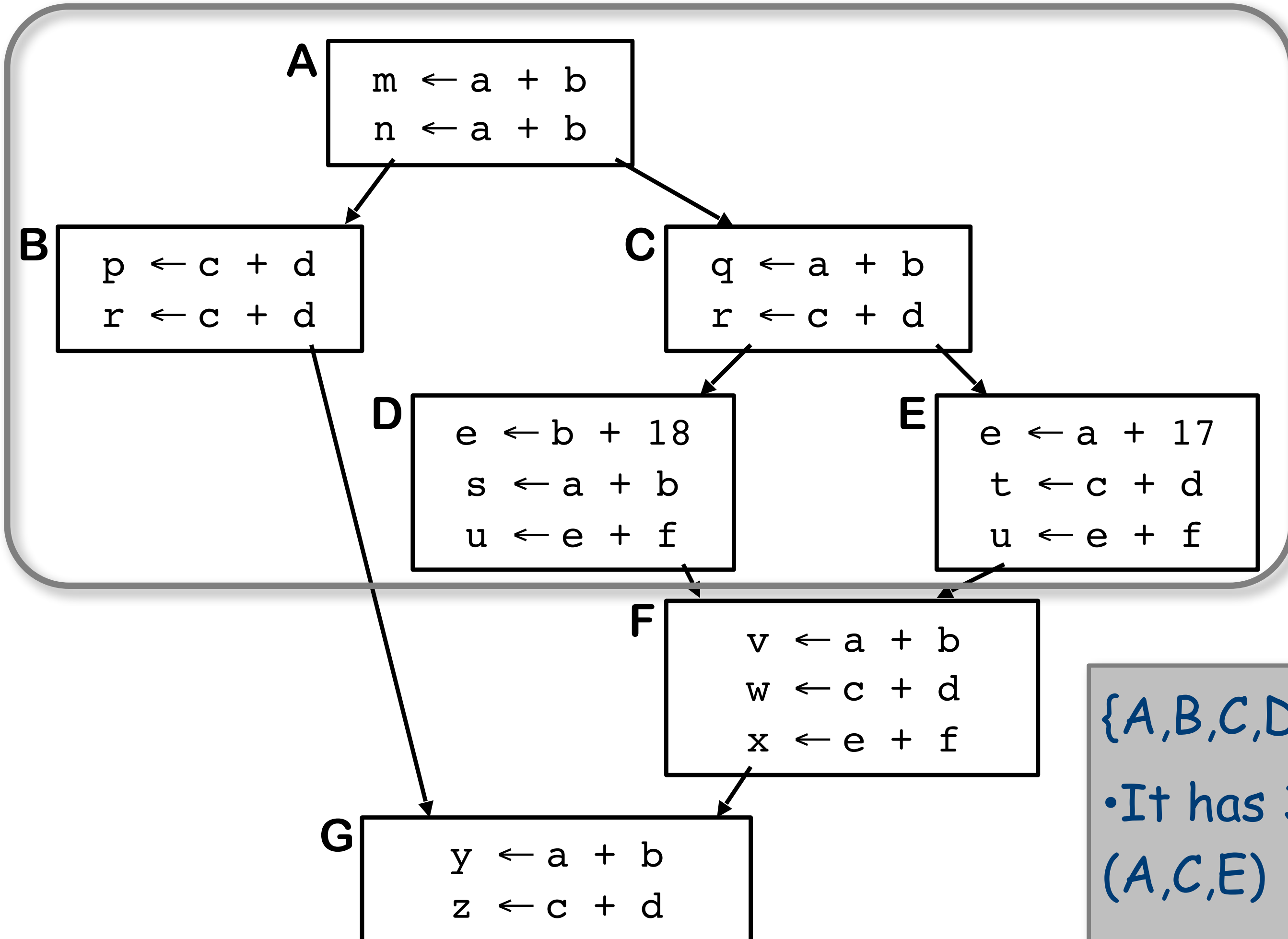
Whole procedure optimization (intraprocedural)

- Operate on entire CFG for a procedure
- Presence of cyclic paths forces analysis then transformation

Whole program optimization (interprocedural)

- Operate on some or all of the call graph (multiple procedures)
- Must contend with call/return & parameter binding

Superlocal Value Numbering (SVN)



EBB: A maximal set of blocks $B_1, B_2, \ldots, B_n$ where each B_i, except B_1, has only exactly one predecessor and that block is in the EBB.

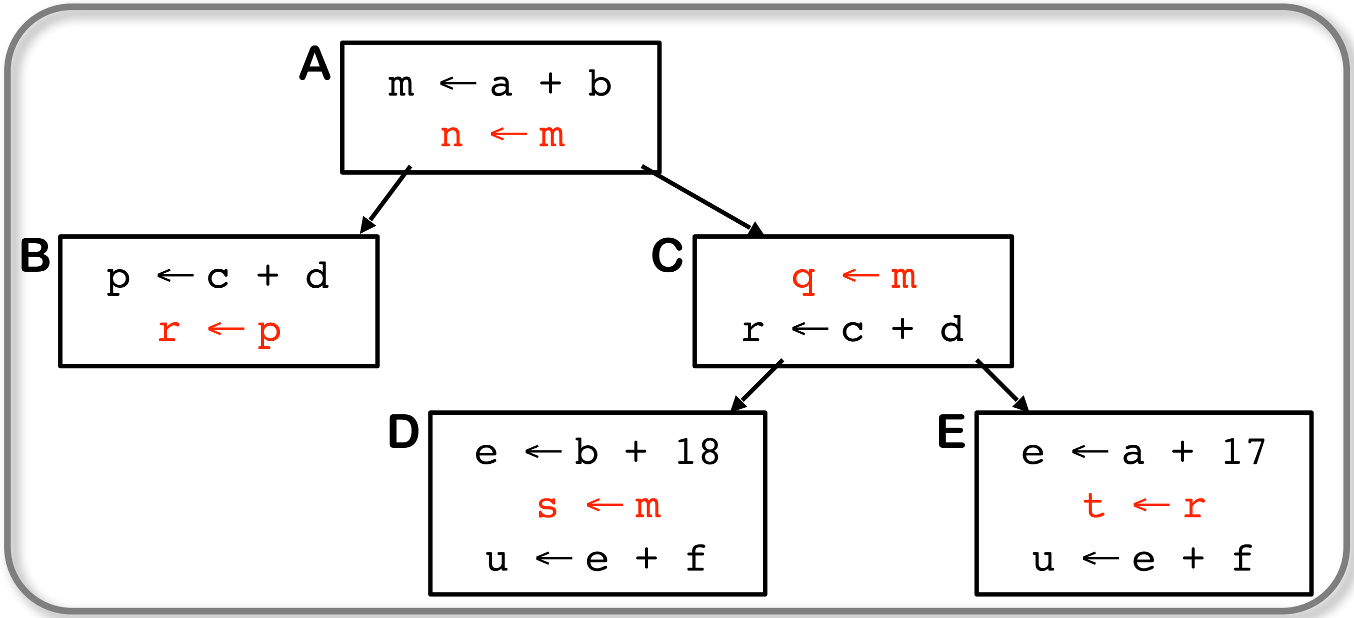
{A,B,C,D,E} is an EBB

- It has 3 paths: (A,B), (A,C,D), & (A,C,E)
- Can sometimes treat each path as if it were a block

{F} & {G} are degenerate EBBs

Superlocal: "applied to an EBB"

After Superlocal Value Numbering (SVN)



EBB: A maximal set of blocks B_1, B_2, …, B_n where each B_i, except B_1, has only exactly one predecessor and that block is in the EBB.

- Capture expression in temporaries to avoid bugs if variable m is rewritten

{A,B,C,D,E} is an EBB

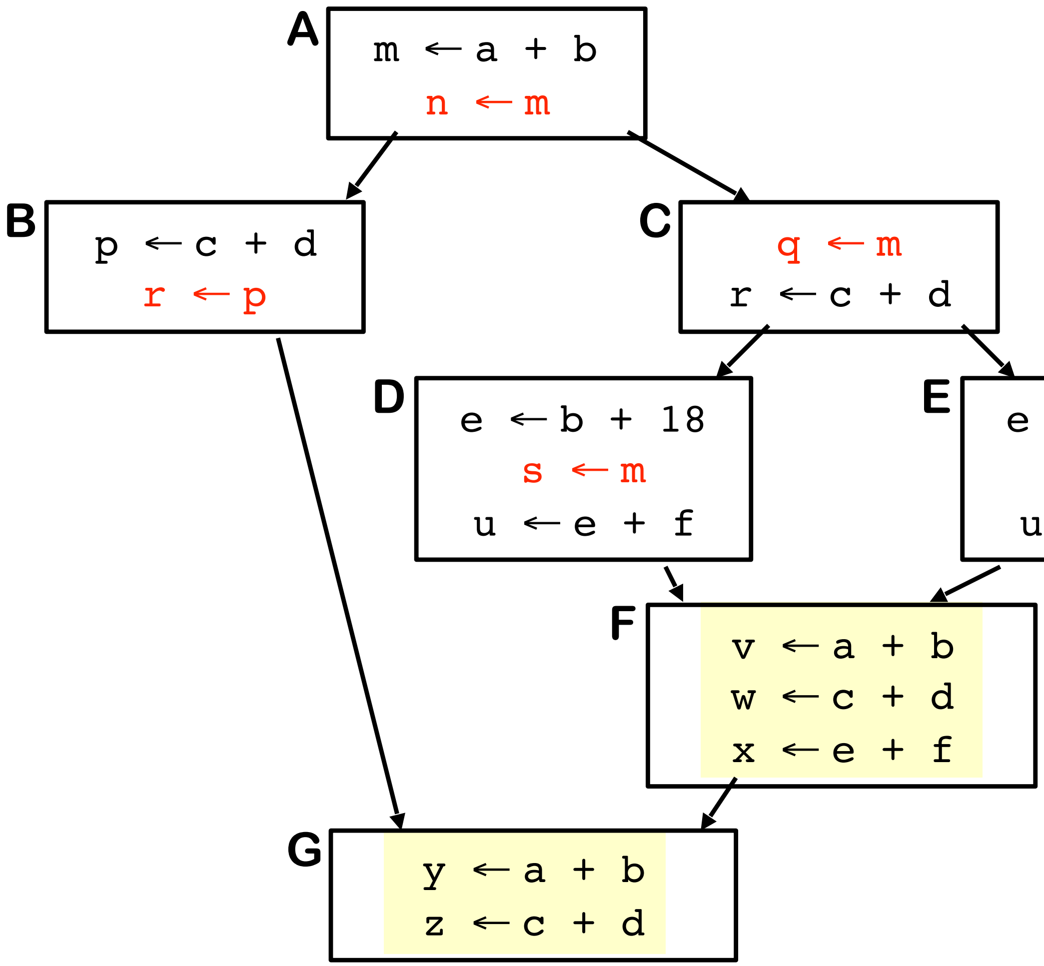
- It has 3 paths: (A,B), (A,C,D), & (A,C,E)
- Can sometimes treat each path as if it were a block

{F} & {G} are degenerate EBBs

Superlocal: "applied to an EBB"

Limitations of Superlocal Value Numbering

SVN finds redundant ops in red



Missed opportunities in yellow remain!

(need stronger methods)

Dominators

Definitions

x dominates y if and only if every acyclic path from the entry of the control-flow graph to the node for y includes x

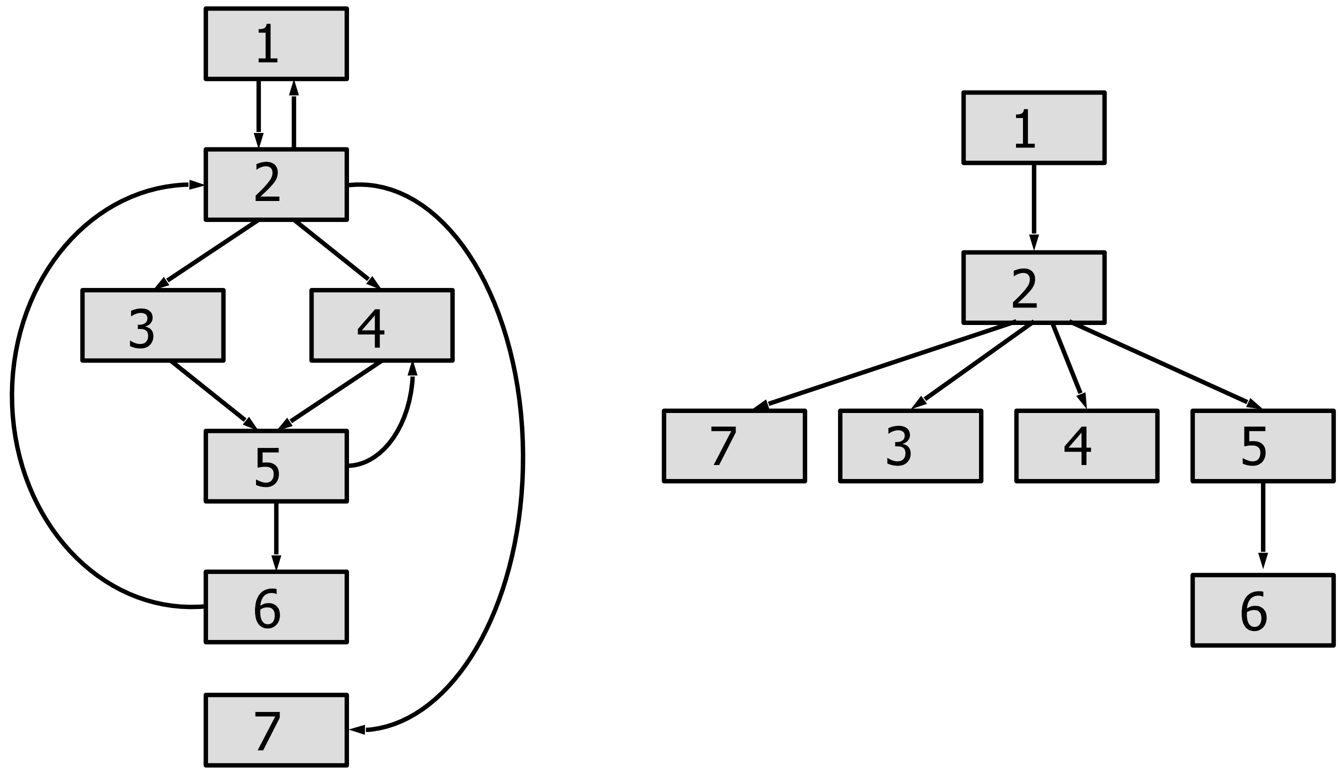
- By definition, x dominates x
- We associate a Dom set with each node
- |Dom(x)| ≥ 1

Immediate dominators

- For any node x, there must be a y in Dom(x) closest to x
- We call this y the immediate dominator of x
- As a matter of notation, we write this as IDom(x)
- Dominator Tree defined with root = entry, and IDom has parent map

Dominator Tree

- Build a dominator tree as follows:
  - Root is CFG entry node n_0
  - m is child of node n iff n=idom(m)
- Example:

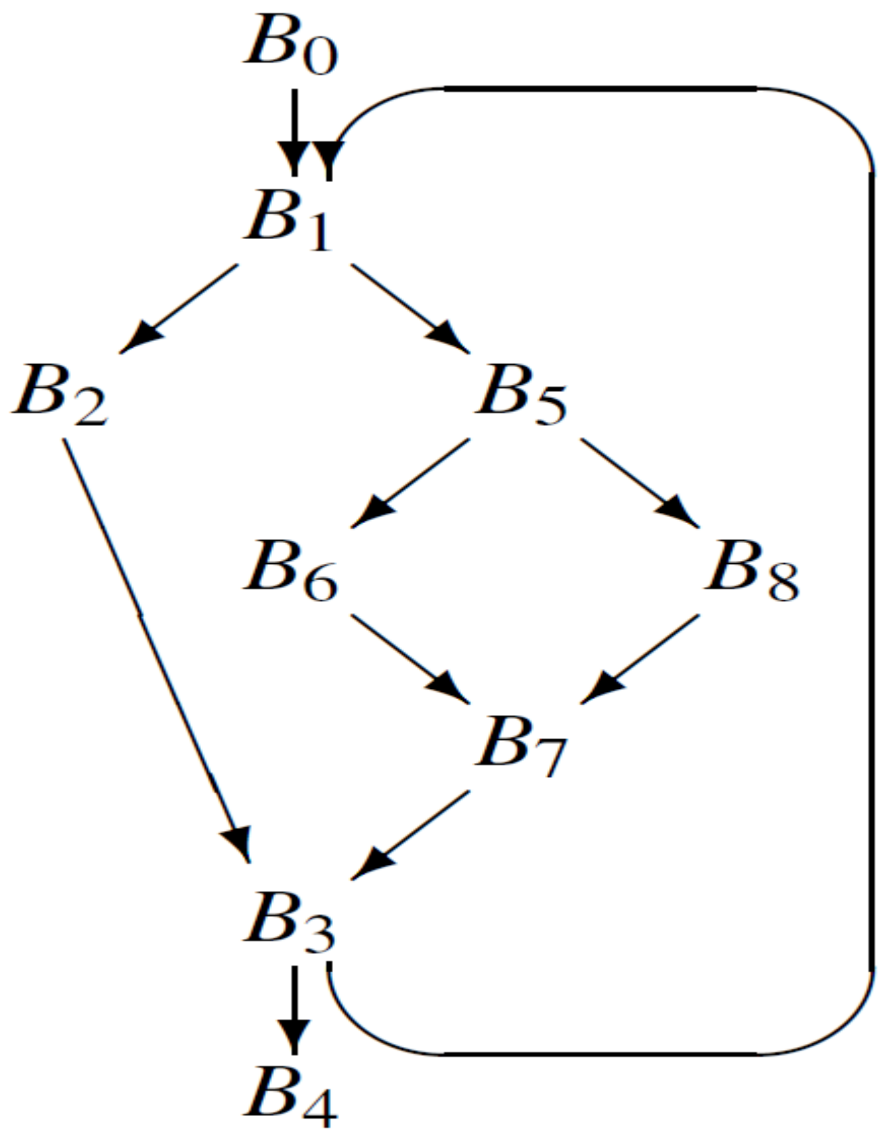


Worksheet-4 Solution

(From Lecture 4 given on 01/16/2019)

- Q1
  Compute **dominator sets** for each node in the CFG.

Dom(n)	Set of basic blocks that dominate B_n
Dom(B_0)	{B_0}
Dom(B_1)	{B_0, B_1}
Dom(B_2)	{B_0, B_1, B_2}
Dom(B_3)	{B_0, B_1, B_3}
Dom(B_4)	{B_0, B_1, B_3, B_4}
Dom(B_5)	{B_0, B_1, B_5}
Dom(B_6)	{B_0, B_1, B_5, B_6}
Dom(B_7)	{B_0, B_1, B_5, B_7}
Dom(B_8)	{B_0, B_1, B_5, B_8}

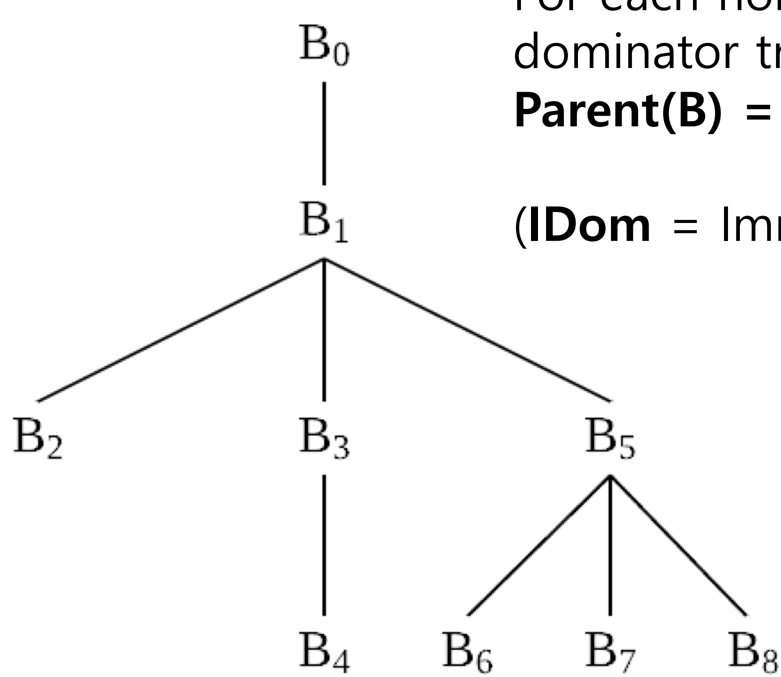


Q2: Draw the dominator tree of the CFG

Observations:

- The immediate dominator idom(n) of a node n is the unique last strict (different from n) dominator on any path from ENTRY to n
- The dominator tree is an efficient encoding of dominator sets; the dominator set of node n can be obtained by enumerating all nodes from n to the root of the tree.

Dom(n)	Set of basic blocks that dominate B_n
Dom(B_0)	{B_0}
Dom(B_1)	{B_0, B_1}
Dom(B_2)	{B_0, B_1, B_2}
Dom(B_3)	{B_0, B_1, B_3}
Dom(B_4)	{B_0, B_1, B_3, B_4}
Dom(B_5)	{B_0, B_1, B_5}
Dom(B_6)	{B_0, B_1, B_5, B_6}
Dom(B_7)	{B_0, B_1, B_5, B_7}
Dom(B_8)	{B_0, B_1, B_5, B_8}



For each non-entry node B in the dominator tree,
Parent(B) = IDom(B)

(**IDom** = Immediate Dominator)

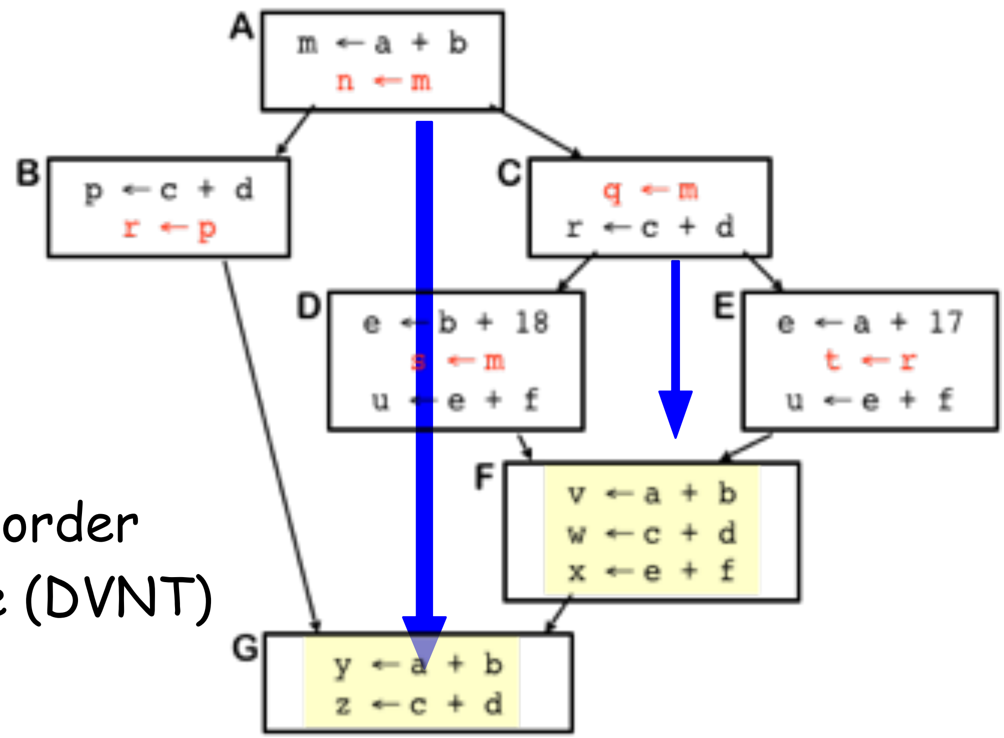
Lecture 5: Motivation for Dominators (Recap)

We have not helped with F or G

» Multiple predecessors

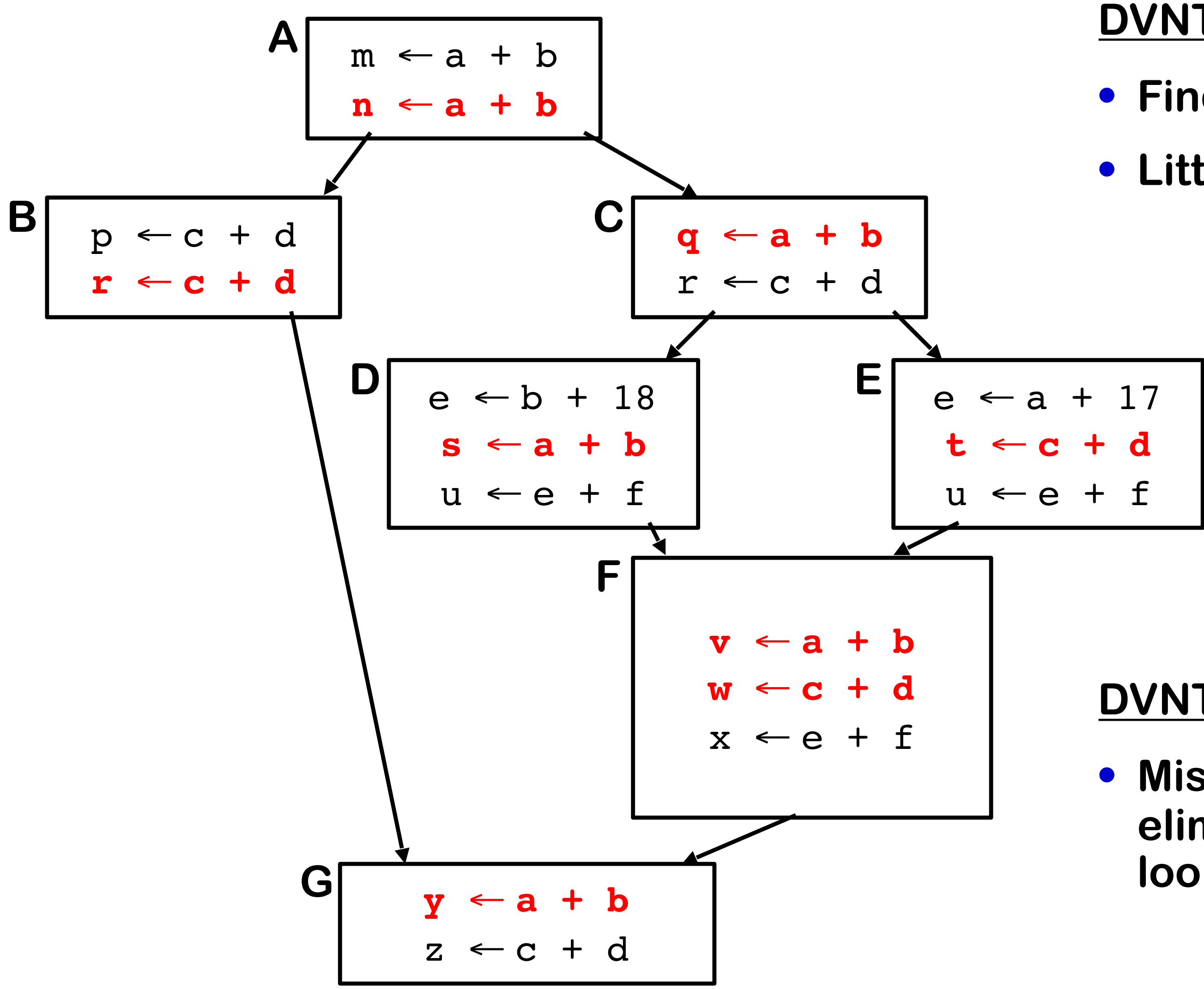
» Must decide what facts hold in F and in G

- For G, combine B & F?
- Merging state is expensive
- Fall back on what's known



- Can use value numbers from block IDom(x) when processing x, e.g.,
  - Use C for F and A for G
- Imposes a Dom-based application order
- Leads to Dominator VN Technique (DVNT)

Dominator Value Numbering: Example



DVNT advantages

- **Find more redundancy**
- **Little additional cost**

DVNT shortcomings

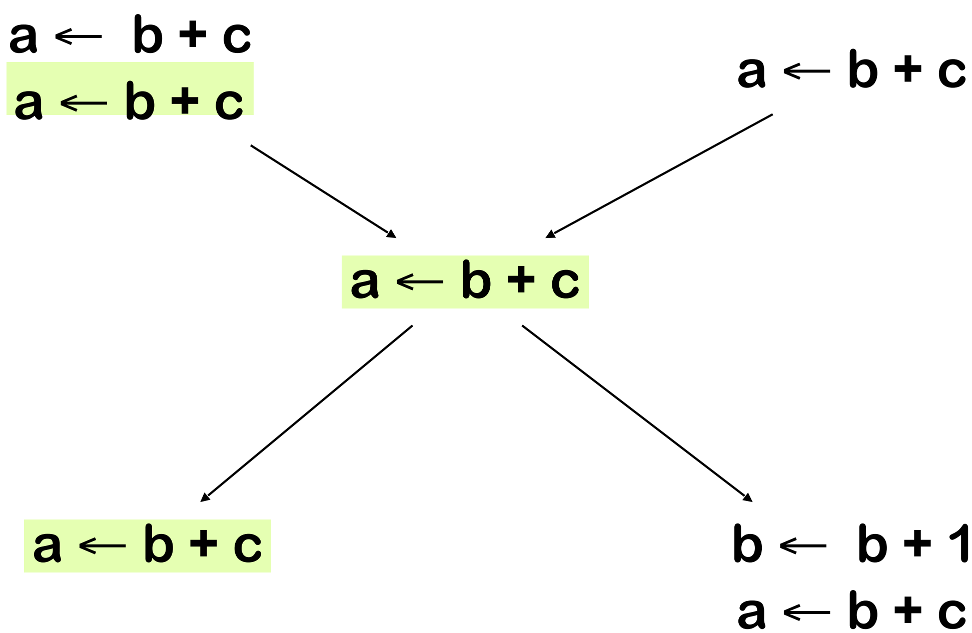
- **Misses redundancy elimination opportunities in loops**

Redundant Expression: General Definition

An expression is redundant at point p if, on every path to p

1. It is evaluated before reaching p, and
2. None of its constitutent values is redefined before p

Example

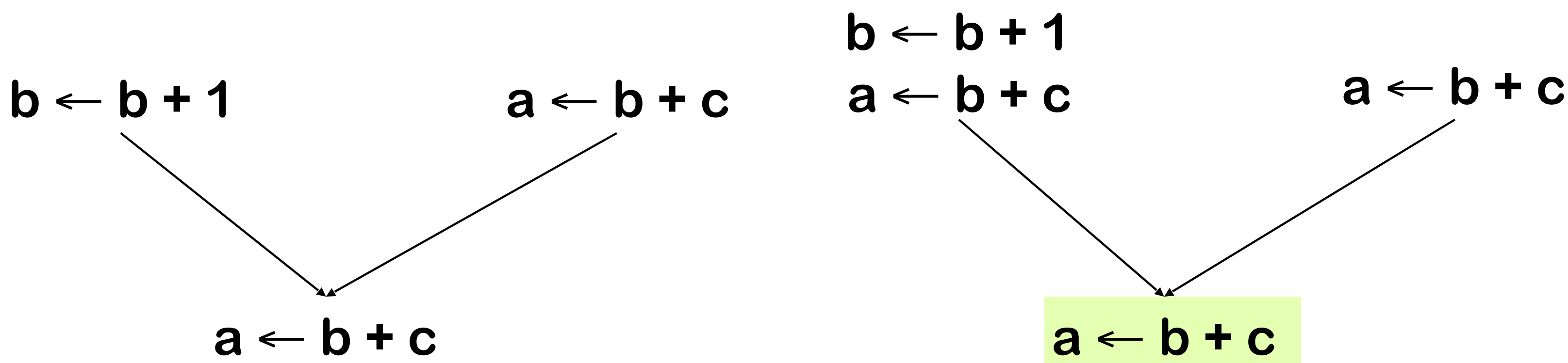


Some occurrences of b+c are redundant

Partially Redundant Expression

An expression is partially redundant at p if it is redundant along some, but not all, paths reaching p

Example



Two steps:

1) **Insert a copy of “a ← b + c” after the definition of b**

2) **Delete the highlighted computation of “a ← b + c” since it is now redundant**

Loop Invariant Expression

Another example

Loop invariant expressions are partially redundant

- Partial redundancy elimination performs code motion
- Major part of the work is figuring out where to insert operations
- Question: what if we had a while loop, instead of a do-while loop as in the above example?

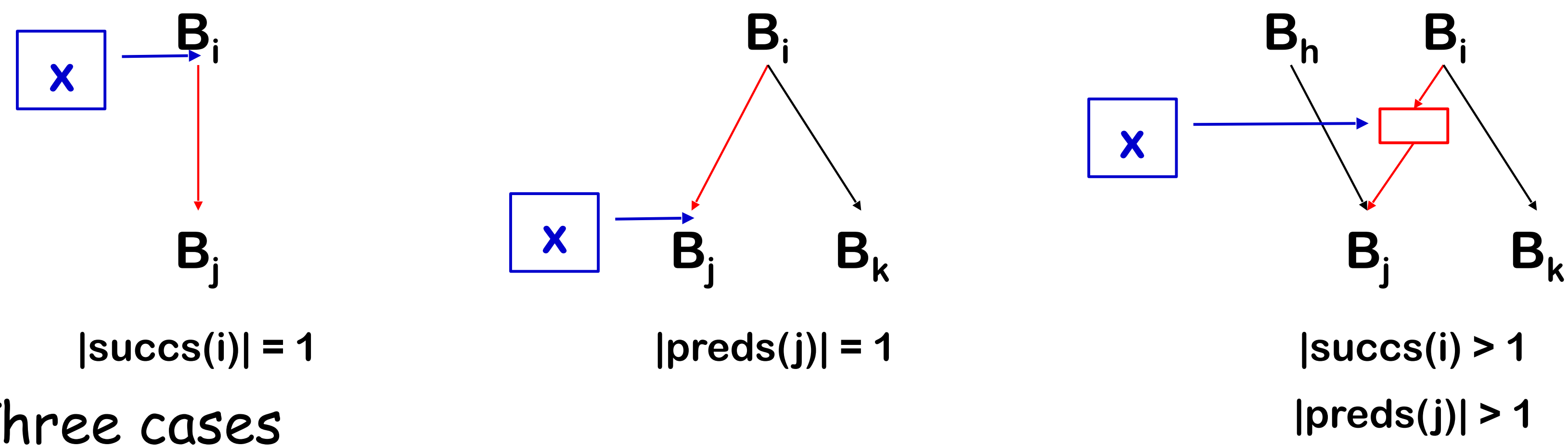
Lazy Code Motion

The concept

- Compute INSERT & DELETE sets by solving data flow subproblems
  - Compute available expressions (AVAIL)
    - Can be extended with value numbering
  - Compute anticipable expressions (ANT)
  - AVAIL and ANT are advanced concepts that are not required for this class
- Linear pass to rewrite code using INSERT & DELETE sets
  - $x \in \text{INSERT}(i,j) \Rightarrow$ insert x at start of j, end of i, or new block
  - $x \in \text{DELETE}(k) \Rightarrow$ delete first evaluation of x in k
- Lazy Code Motion (LCM) extends earlier work on Partial Redundancy Elimination (PRE)
- See textbook for details

Lazy Code Motion: Placement Rules

- $x \in \textsc{Insert}(i,j)$

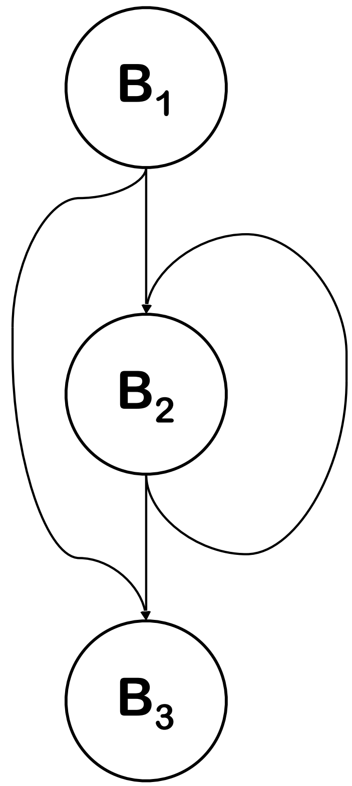


Three cases

- |succs(i)| = 1 $\Rightarrow$ insert at end of i
- | succs(i)| > 1, but |preds(j)| = 1$\Rightarrow$ insert at start of j
- | succs(i)| > 1, & |preds(j)| > 1 $\Rightarrow$ create new block in ‹i,j› for x
  - Modify CFG by adding a new basic block in this case

Lazy Code Motion: Example

B_1: $r_1 \leftarrow 1$
$r_2 \leftarrow r_0 + @m$
if $r_1 < r_2 \rightarrow B_2, B_3$
B_2: ...
$r_{20} \leftarrow r_{17} * r_{18}$
...
$r_4 \leftarrow r_1 + 1$
$r_1 \leftarrow r_4$
if $r_1 < r_2 \rightarrow B_2, B_3$
B_3: ...



Insert(1,2) = { r20 ← r17 * r18 }
Delete(2) = { r20 ← r17 * r18 }
==>
Move r20 ← r17 * r18 from start of B_2 to B_1—> B_2 edge

Available Expression Analysis

An expression x+y is available if and only if, along every path from the procedure's entry, it has been evaluated, and its constituent subexpressions (x & y) have not been re-defined.

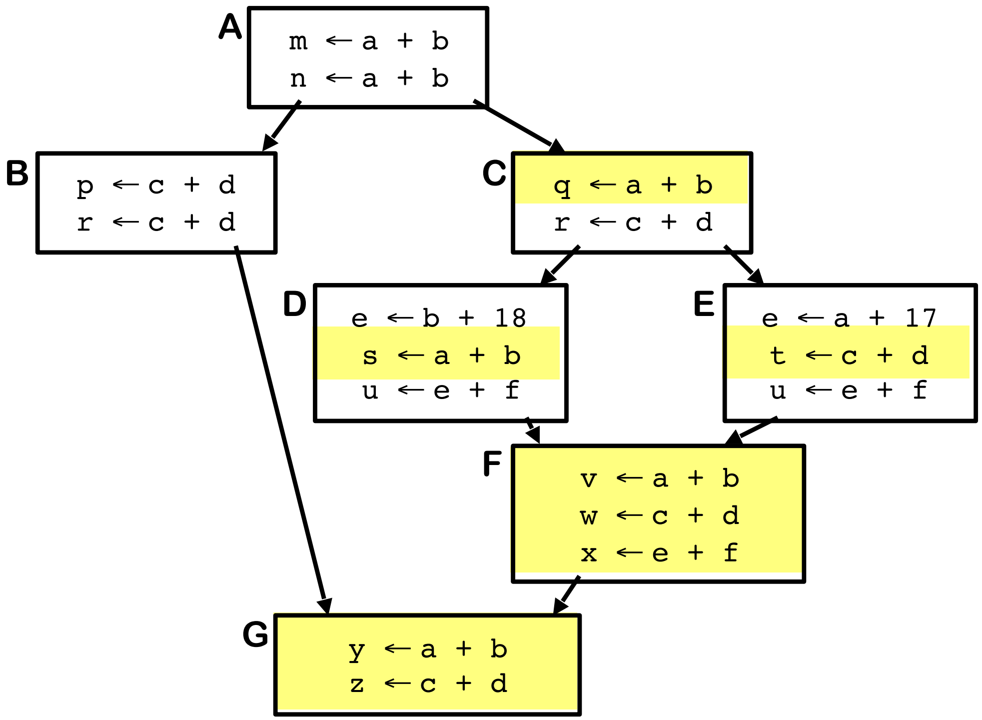
If the compiler can prove that an expression is available

- It can preserve the results of earlier evaluations
- It can replace the current evaluation with a reference

Two pieces to redundancy elimination

- Proving that x+y is available
- Rewriting the code to eliminate the redundant evaluation

Example



AVAIL(A) = Ø

AVAIL(B) = {a+b}

AVAIL(C) = {a+b}

AVAIL(D) = {a+b, c+d}

AVAIL(E) = {a+b, c+d}

AVAIL(F) = {a+b, c+d, e+f}

AVAIL(G) = {a+b, c+d}

Formal Definition of Available Expressions

For each block b, let

- » Avail(b) be the set of expressions available on entry to b
- » DEExpr(b) be the set of expressions computed in b and available on exit (Downward Exposed Expressions)
- » ExprKill(b) be these set of expressions that are killed in b
  - » An expression is killed one of its inputs is assigned a value

Now, Avail(b) can be defined as:

$$Avail(b) = \cap_{x \in pred(b)} (DEExpr(x) \cup (Avail(x) - ExprKill(x)))$$

preds(b) is the set of b's predecessors in the control-flow graph

- This system of simultaneous equations forms a data-flow problem, and can be solved as past data-flow problems that we've seen (reaching definitions, dominators)

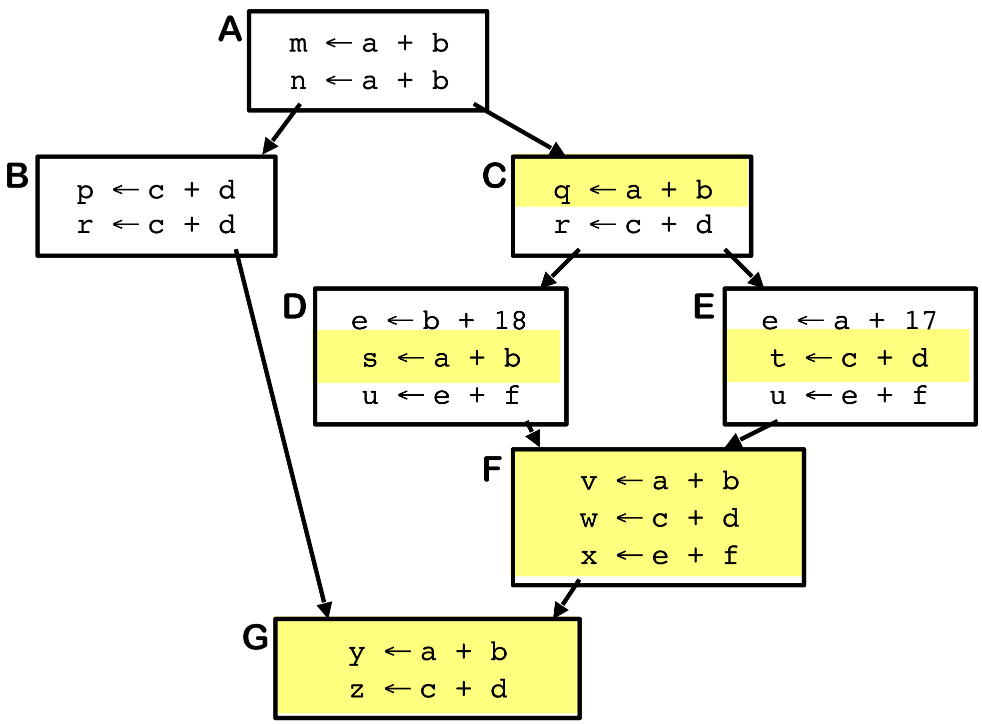
Computing Available Expressions

The Big Picture

1. Build a control-flow graph
2. Gather the initial (local) data — DEExpr(b) & ExprKill(b)
3. Propagate information around the graph, evaluating the equation
4. Use output of Available Expression analysis as needed, e.g., for lazy code motion or redundancy elimination

Computing Avail for the Example

	A	B	C	D	E	F	G
DEExpr	a+b	c+d	a+b, c+d	b+18, a+b, e+f	a+17, c+d, e+f	a+b, c+d, e+f	a+b, c+d
ExprKill	∅	∅	∅	e+f	e+f	∅	∅



AVAIL(A) = Ø

AVAIL(B) = {a+b} ∪ (ø ∩ all)

= {a+b}

AVAIL(C) = {a+b}

AVAIL(D) = {a+b, c+d} ∪ ({a+b} ∩ all)

= {a+b, c+d}

AVAIL(E) = {a+b, c+d}

AVAIL(F) = [{b+18, a+b, e+f} ∪

({a+b, c+d} ∩ {all − e+f})]

∩ [{a+17, c+d, e+f} ∪

({a+b, c+d} ∩ {all − e+f})]

= {a+b, c+d, e+f}

AVAIL(G) = [{c+d} ∪ ({a+b} ∩ all)]

∩ [{a+b, c+d, e+f} ∪

({a+b, c+d, e+f} ∩ all)]

= {a+b, c+d}

$$\textbf{Avail(b)} = \bigcap_{x \in pred(b)} (\textbf{DEExpr}(x) \cup (\textbf{Avail}(x) - \textbf{ExprKill}(x)))$$

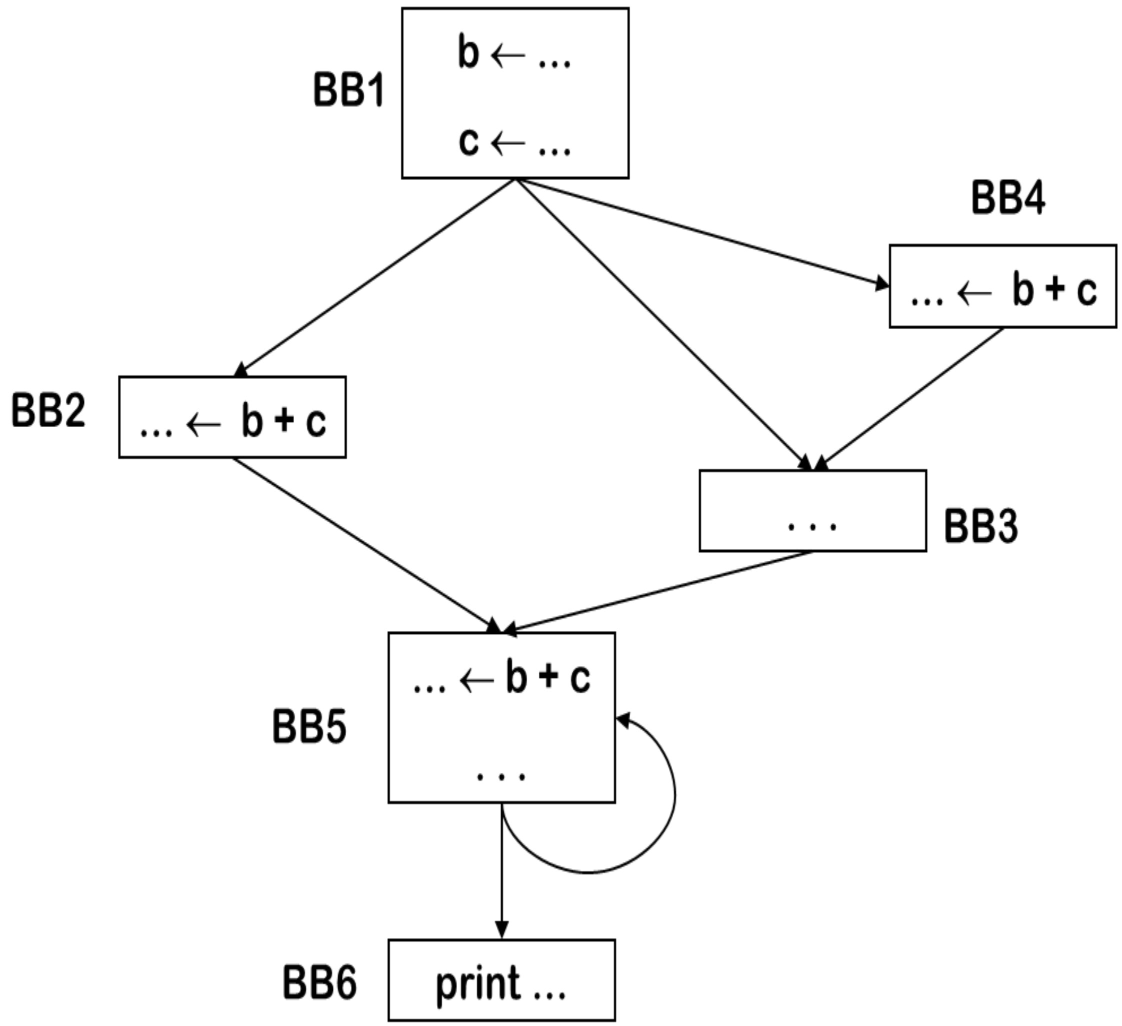
Algorithm halts in one pass for this example, because graph is acyclic

Worksheet–5 Solution

From lecture given on 01/23/2019

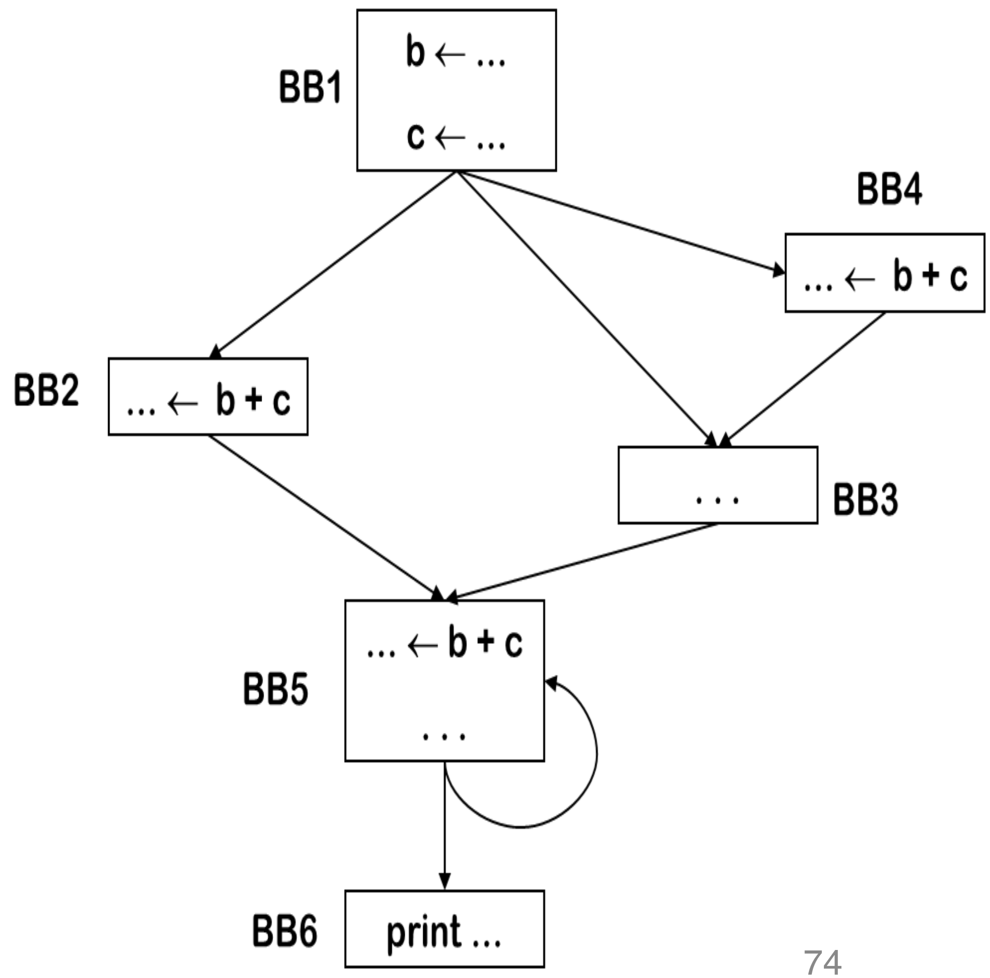
Question1.

Consider the control flow graph shown below. Indicate where computations of **b+c** can be inserted and deleted to minimize the number of times it is computed. Assume that there are no other defs of b and c, and do not worry about dead code elimination in this example.



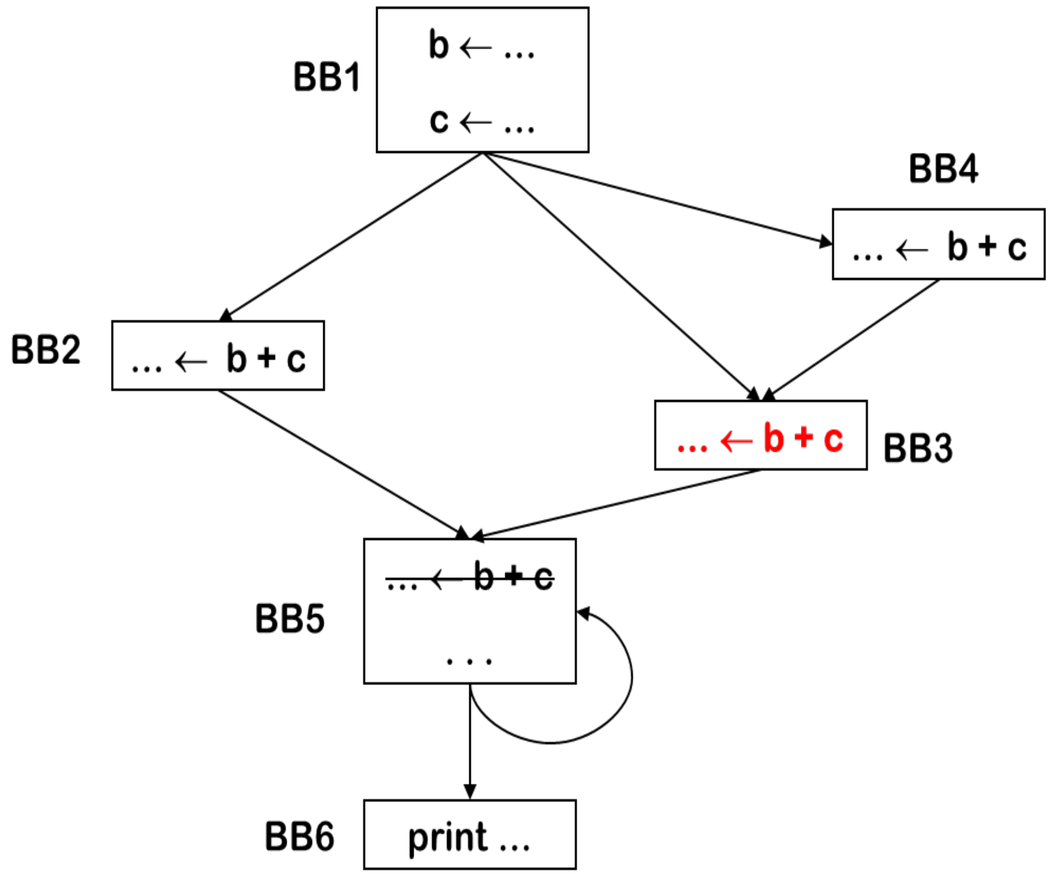
Sample solution : 1st step

- Computation of **b+c** in BB5 is partially redundant.
- We can remove the redundancy by moving the computation of **b+c** from BB5 to a location before BB5.



Sample solution : 2nd step

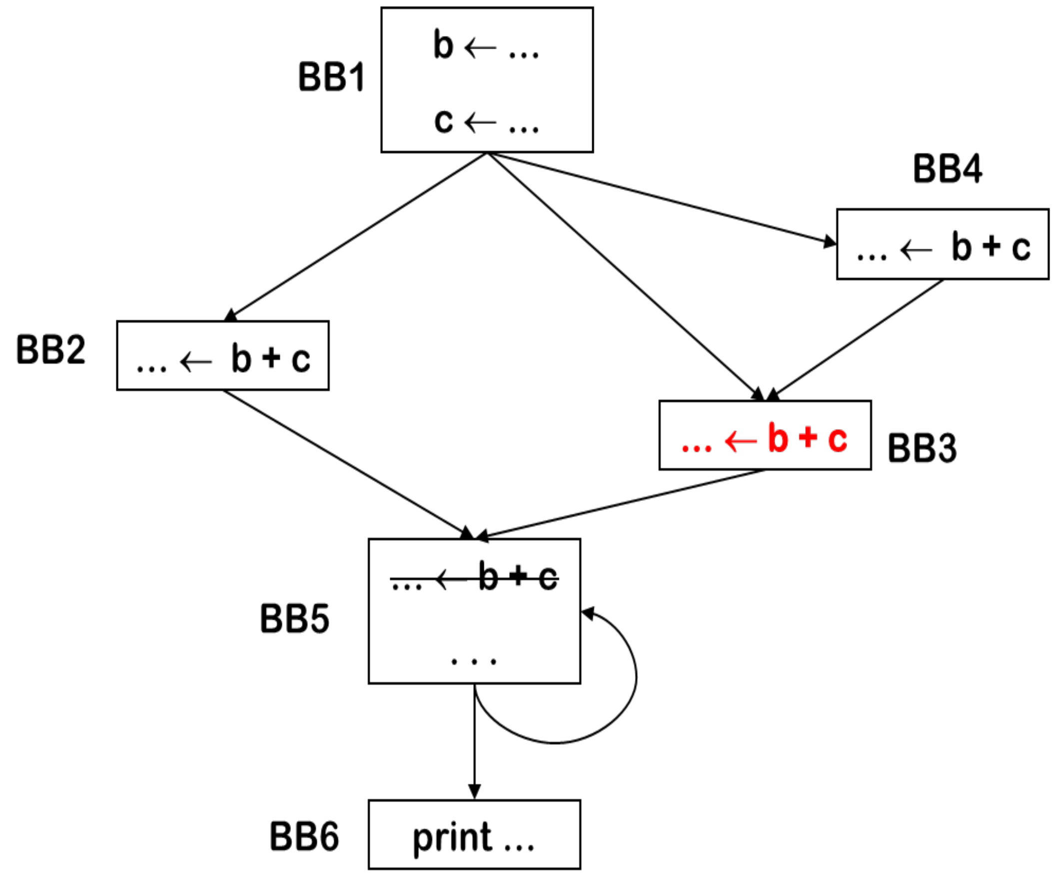
- By moving the computation of **b+c** to the end of BB3, redundancy of computing **b+c** along the path **[BB1→BB2→BB5 →* BB6]** is removed.
- Redundancy still remains along the path **[BB1→BB4→BB3→BB5 →* BB6]**.



Sample solution : 2nd step

- **b+c** is computed on every path that leaves ***BB1*** and produces the same value at each of those computations.

  (= **b+c** is an <u>anticipable expression</u> from the end of BB1)



Sample solution : 3rd step

- Since **b+c** is anticipable from the end of BB1, it is safe to append the computations of **b+c** to the end of BB1, and delete others.
- After the modification, there are no redundancies remaining in any control path.