Static Program Analysis

Yue Li and Tian Tan

2020 Spring

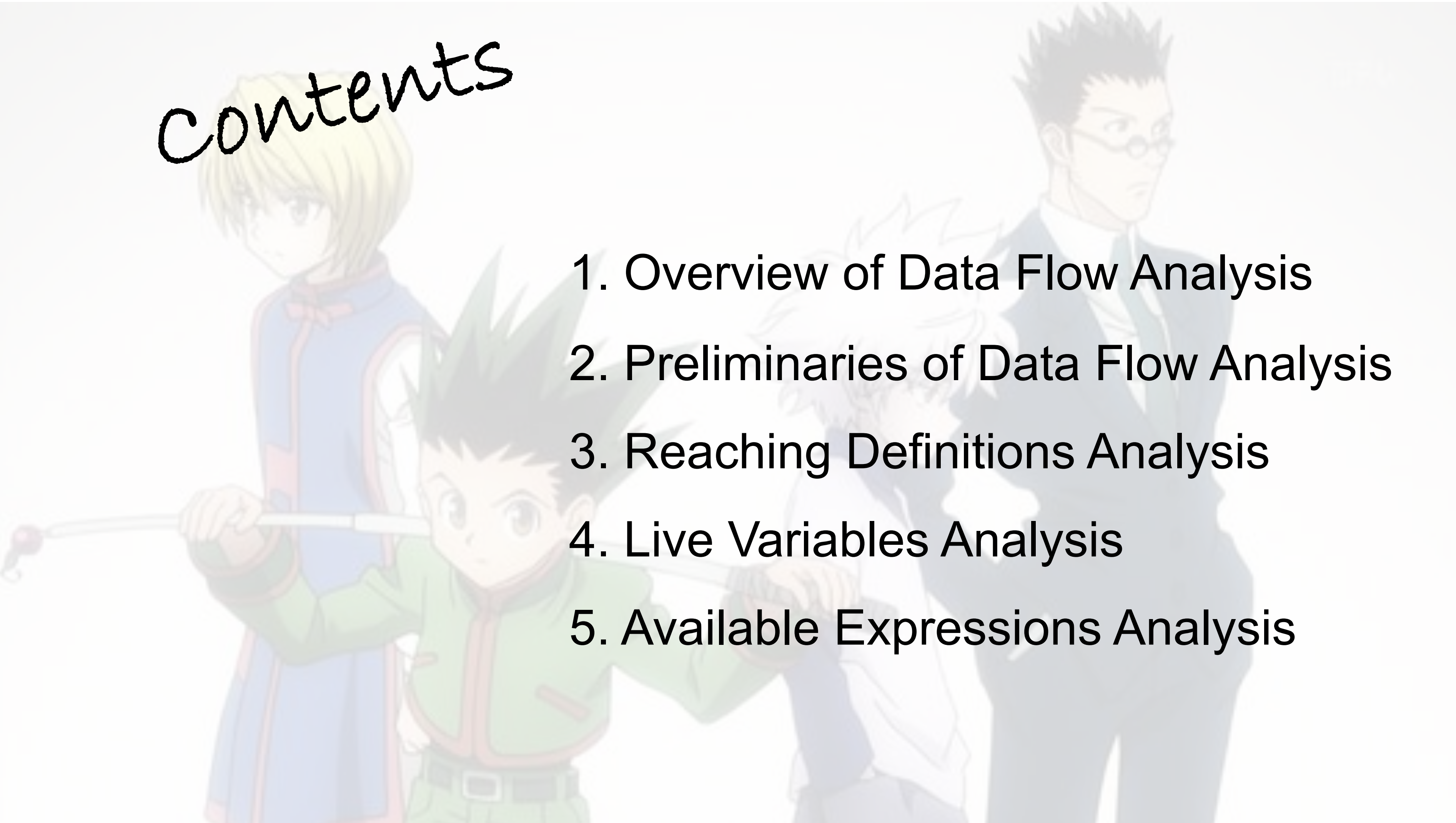
Static Program Analysis

Data Flow Analysis — Applications

Nanjing University

Yue Li

2020



Contents

Data Flow Analysis

Data Flow Analysis

How Data Flows on CFG?

Data Flow Analysis

How Data Flows on CFG?

How Data

Flows through the

CFG (a program)?

Data Flow Analysis

How Data Flows on CFG?

How application-specific Data
Flows through the

CFG (a program)?

Data Flow Analysis

How Data Flows on CFG?

How application-specific Data Flows through the Nodes (BBs/statements) and Edges (control flows) of CFG (a program)?

Recall 1st lecture

Data Flow Analysis

How Data Flows on CFG?

How application-specific Data ← Abstraction

Over-approximation → Flows through the

Nodes (BBs/statements) and

Edges (control flows) of

CFG (a program)?

Recall 1st lecture

Data Flow Analysis

How Data Flows on CFG?

How application-specific Data ← Abstraction

Over-approximation → Flows through the

Nodes (BBs/statements) and

Edges (control flows) of

CFG (a program)?

for most static analyses (may analysis)

Recall 1st lecture

Data Flow Analysis

How Data Flows on CFG?

How application-specific Data ← Abstraction

Over-approximation → Flows through the

for most static analyses (may analysis)

Nodes (BBs/statements) and

Edges (control flows) of

CFG (a program)?

may analysis:
outputs information that may be true (over-approximation)
must analysis:
outputs information that must be true (under-approximation)
Over- and under-approximations are both for safety of analysis

Data Flow Analysis

How Data Flows on CFG?

How application-specific Data ← Abstraction

Safe-approximation → Flows through the

may analysis: safe=over

must analysis: safe=under

Nodes (BBs/statements) and

Edges (control flows) of

CFG (a program)?

Data Flow Analysis

How Data Flows on CFG?

How application-specific Data ← Abstraction

Safe-approximation → Flows through the

Transfer function → Nodes (BBs/statements) and

Edges (control flows) of

Control-flow handling → CFG (a program)?

Recall 1st lecture

Data Flow Analysis

How Data Flows on CFG?

How application-specific Data ← Abstraction

Safe-approximation → Flows through the

Transfer function → Nodes (BBs/statements) and

Control-flow handling → Edges (control flows) of

CFG (a program)?

Recall 1^{st} lecture

Data Flow Analysis

How Data Flows on CFG?

$+, -, 0, \bot, \top$

How application-specific Data ← Abstraction

Safe-approximation → Flows through the

Transfer function → Nodes (BBs/statements) and

Control-flow handling → Edges (control flows) of

CFG (a program)?

Recall 1st lecture

Data Flow Analysis

How Data Flows on CFG?

$+, -, 0, \perp, \top$

How application-specific Data ← Abstraction

Safe-approximation → Flows through the

Transfer function → Nodes (BBs/statements) and

$+ \; op \; - = -; \; + \; op \; + = +$

Control-flow handling → Edges (control flows) of

CFG (a program)?

Data Flow Analysis

Recall 1st lecture

How Data Flows on CFG?

$+, -, 0, \bot, \top$

How application-specific Data ← Abstraction

Safe-approximation → Flows through the

Transfer function → Nodes (BBs/statements) and

$+ \; op \; - = -; \; + \; op \; + = +$

Control-flow handling → Edges (control flows) of

union the signs at merges

CFG (a program)?

Recall 1st lecture

Data Flow Analysis

How Data Flows on CFG?

How application-specific Data ← Abstraction

+,-,0,⊥,⊤

Safe-approximation → Flows through the

Transfer function → Nodes (BBs/statements) and

+ *op* − = − ; + *op* + = +

Control-flow handling → Edges (control flows) of

CFG (a program)?

union the signs at merges

different data-flow analysis applications have
different data abstraction and
different flow safe-approximation strategies, i.e.,
different transfer functions and control-flow handlings

Recall 1st lecture

Data Flow Analysis

How Data Flows on CFG?

+,-,0,⊥,⊤

How application-specific Data ← Abstraction

Safe-approximation → Flows through the

Transfer function → Nodes (BBs/statements) and

+ *op* − = −; + *op* + = +

Control-flow handling → Edges (control flows) of

union the signs at merges

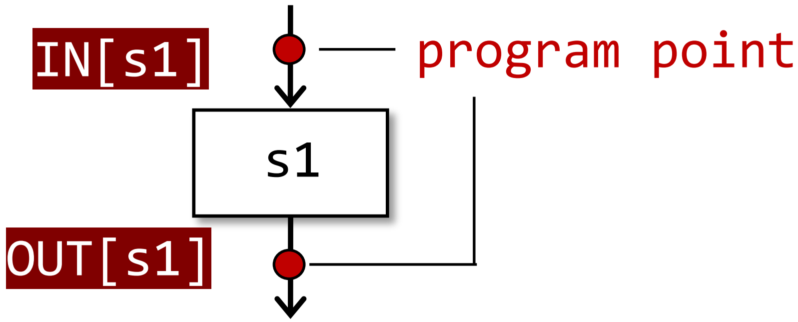
CFG (a program)?

different data-flow analysis applications have
different data abstraction and
different flow safe-approximation strategies, i.e.,
different transfer functions and control-flow handlings

Preliminaries of Data Flow Analysis

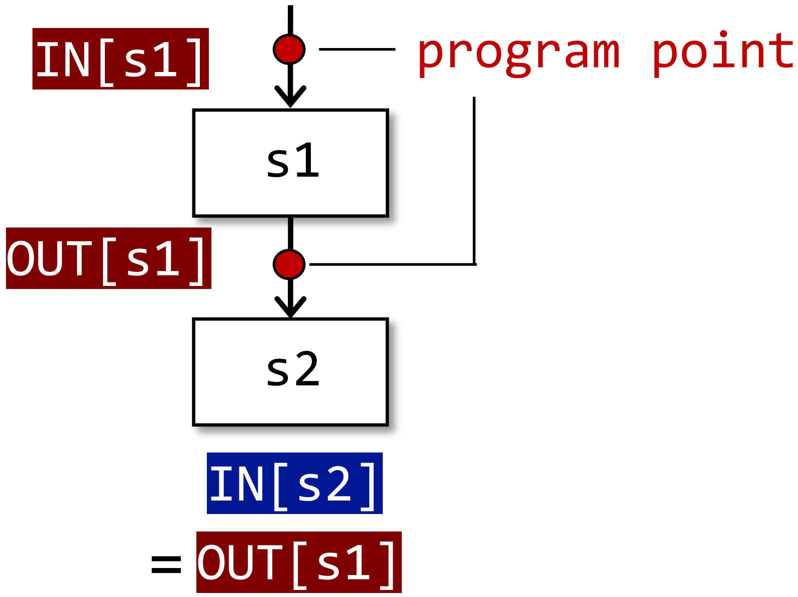
Input and Output States

- Each execution of an IR statement transforms an input state to a new output state
- The input (output) state is associated with the program point before (after) the statement



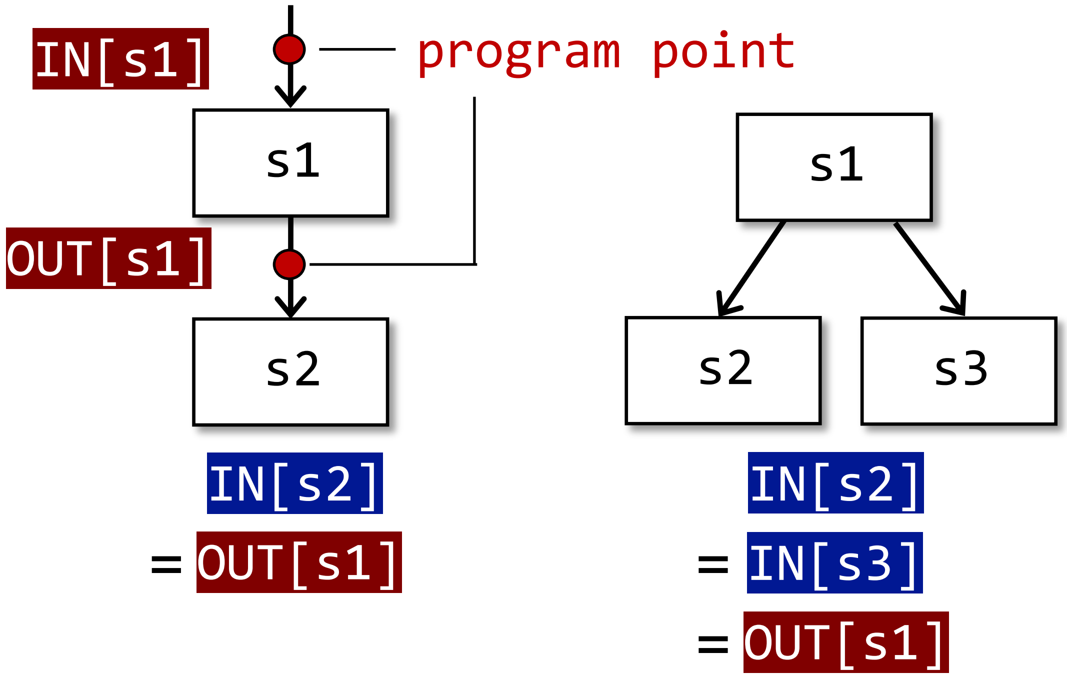
Input and Output States

- Each execution of an IR statement transforms an input state to a new output state
- The input (output) state is associated with the program point before (after) the statement



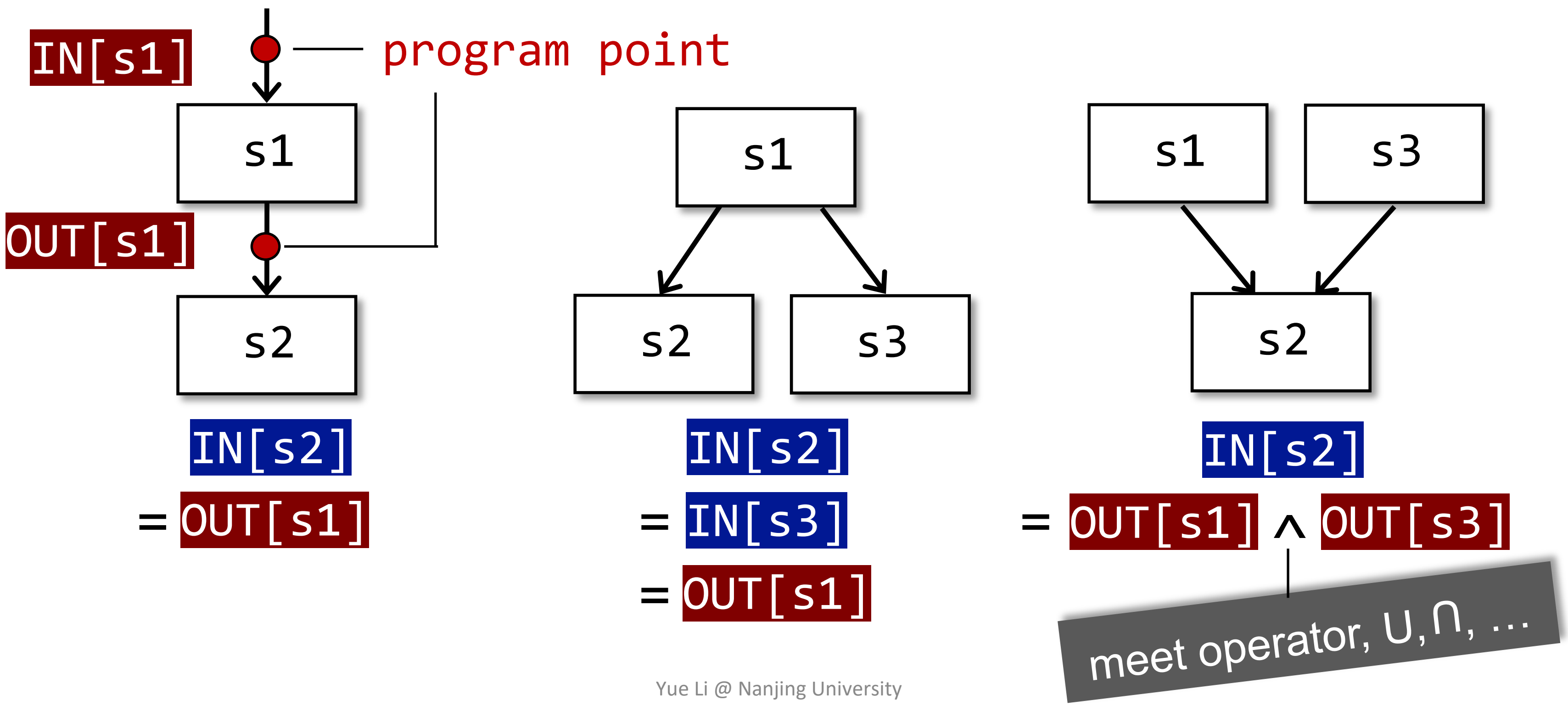
Input and Output States

- Each execution of an IR statement transforms an input state to a new output state
- The input (output) state is associated with the program point before (after) the statement



Input and Output States

- Each execution of an IR statement transforms an input state to a new output state
- The input (output) state is associated with the program point before (after) the statement



In each data-flow analysis application, we associate with every program point a data-flow value that represents an *abstraction* of the set of all possible program states that can be observed for that point.

In each data-flow analysis application, we associate with every program point a data-flow value that represents an *abstraction* of the set of all possible program states that can be observed for that point.

Recall 1st lecture

```
x = 10;
   ●
y = -1;
   ●
x = y;
   ●
x = x / 6;
   ●
```

In each data-flow analysis application, we associate with every program point a data-flow value that represents an *abstraction* of the set of all possible program states that can be observed for that point.

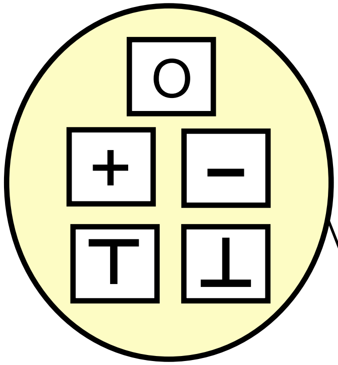
Recall 1st lecture

`x = 10;`	x = $\boxed{+}$ y = $\boxed{\perp}$
`y = -1;`	x = $\boxed{+}$ y = $\boxed{-}$
`x = y;`	x = $\boxed{-}$ y = $\boxed{-}$
`x = x / y;`	x = $\boxed{+}$ y = $\boxed{-}$

In each data-flow analysis application, we associate with every program point a data-flow value that represents an *abstraction* of the set of all possible program states that can be observed for that point.

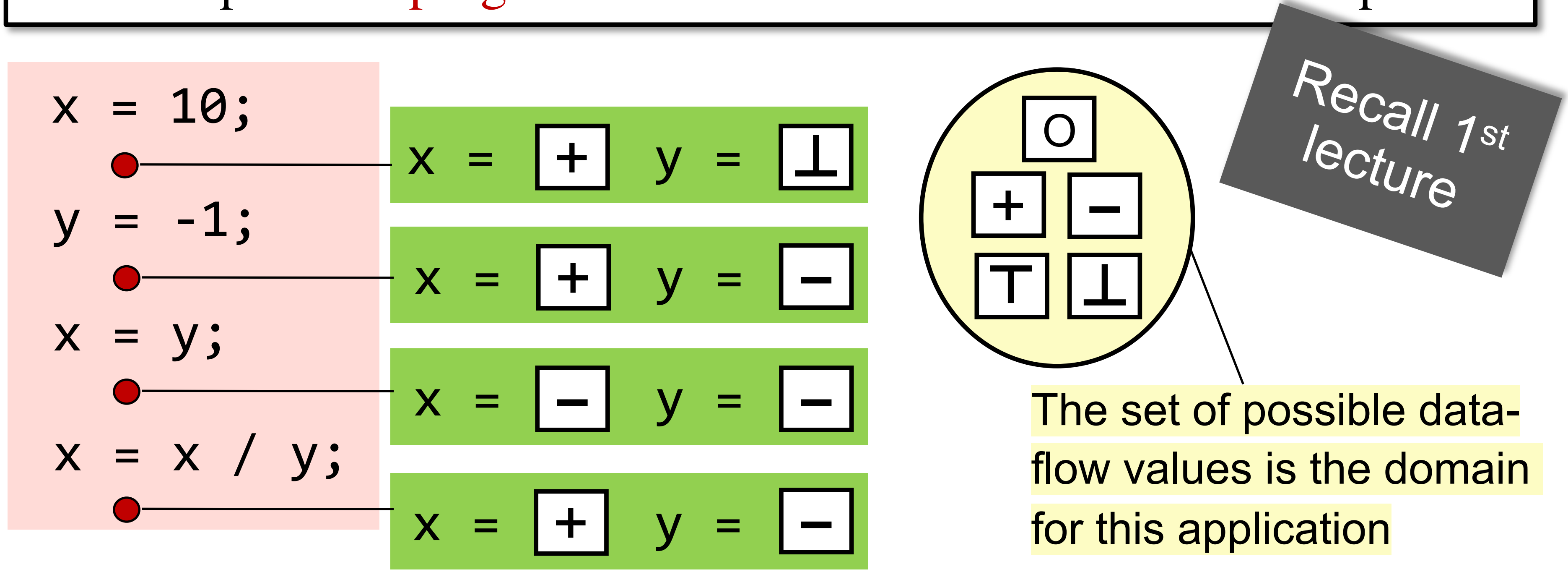
Recall 1st lecture

`x = 10;`	
●	x = $\boxed{+}$ y = $\boxed{\bot}$
`y = -1;`	
●	x = $\boxed{+}$ y = $\boxed{-}$
`x = y;`	
●	x = $\boxed{-}$ y = $\boxed{-}$
`x = x / y;`	
●	x = $\boxed{+}$ y = $\boxed{-}$



The set of possible data-flow values is the domain for this application

In each data-flow analysis application, we associate with every program point a data-flow value that represents an *abstraction* of the set of all possible program states that can be observed for that point.



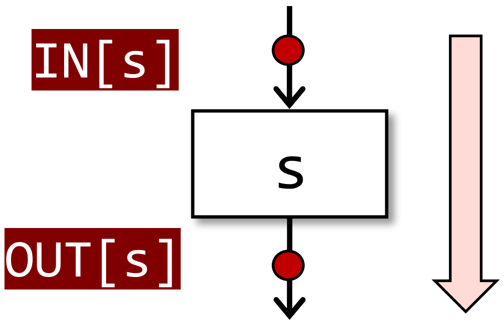
Data-flow analysis is to find a solution to a set of *safe-approximation-directed constraints* on the IN[s]'s and OUT[s]'s, for all statements s.

- *constraints* based on semantics of statements (*transfer functions*)
- *constraints* based on the *flows of control*

Notations for Transfer Function's Constraints

- Forward Analysis

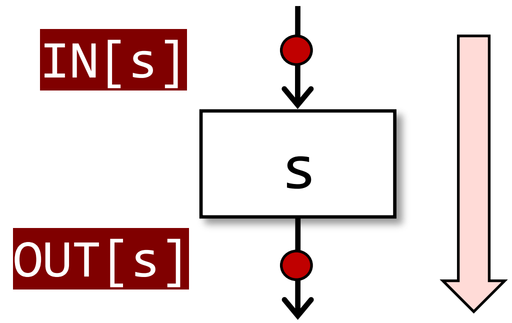
$$\mathrm{OUT}[s] = f_s(\mathrm{IN}[s])$$



Notations for Transfer Function's Constraints

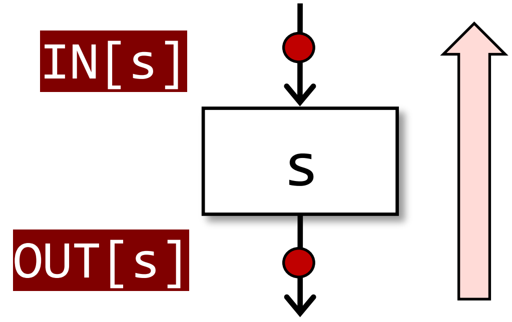
- Forward Analysis

$$\mathrm{OUT}[s] = f_s(\mathrm{IN}[s])$$



- Backward Analysis

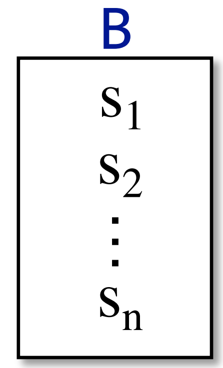
$$\mathrm{IN}[s] = f_s(\mathrm{OUT}[s])$$



Notations for Control Flow’s Constraints

- Control flow within a BB

- Control flow among BBs



Notations for Control Flow's Constraints

- Control flow within a BB

$$\boxed{\mathrm{IN}[s_{i+1}] = \mathrm{OUT}[s_i],\ \textit{for all } i = 1, 2, \ldots, n\text{-}1}$$

B

$$\boxed{\begin{array}{c} s_1 \\ s_2 \\ \vdots \\ s_n \end{array}}$$

- Control flow among BBs

Notations for Control Flow's Constraints

- Control flow within a BB

$$\boxed{\mathrm{IN}[s_{i+1}] = \mathrm{OUT}[s_i],\ \textit{for all } i = 1, 2, \ldots, n\text{-}1}$$

- Control flow among BBs

$$\boxed{\mathrm{IN}[B] = \mathrm{IN}[s_1]} \quad \boxed{\mathrm{OUT}[B] = \mathrm{OUT}[s_n]}$$

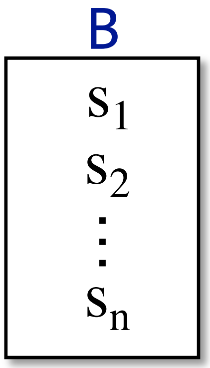
B

$$\boxed{\begin{array}{c} s_1 \\ s_2 \\ \vdots \\ s_n \end{array}}$$

Notations for Control Flow's Constraints

- Control flow within a BB

$$\mathrm{IN}[s_{i+1}] = \mathrm{OUT}[s_i],\ \textit{for all } i = 1, 2, \ldots, \mathrm{n}\text{-}1$$

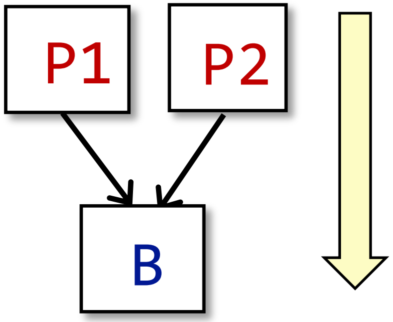


- Control flow among BBs

$$\mathrm{IN}[B] = \mathrm{IN}[s_1] \qquad \mathrm{OUT}[B] = \mathrm{OUT}[s_n]$$

$$\mathrm{OUT}[B] = f_B(\mathrm{IN}[B]),\ f_B = f_{s_n} \circ \ldots \circ f_{s_2} \circ f_{s_1}$$

$$\mathrm{IN}[B] = \bigwedge_{P \text{ a predecessor of } B} \mathrm{OUT}[P]$$

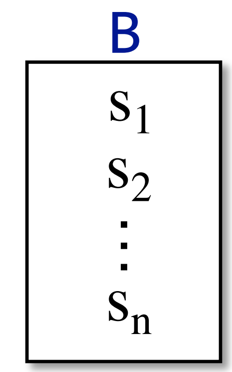


The meet operator $\wedge$ is used to summarize the contributions from different paths at the confluence of those paths

Notations for Control Flow's Constraints

- Control flow within a BB

$$\mathrm{IN}[s_{i+1}] = \mathrm{OUT}[s_i],\ \textit{for all } i = 1, 2, \ldots, n\text{-}1$$

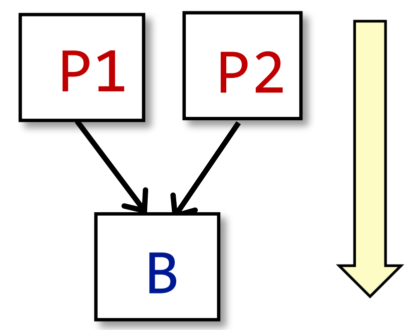


- Control flow among BBs

$$\mathrm{IN}[B] = \mathrm{IN}[s_1] \qquad \mathrm{OUT}[B] = \mathrm{OUT}[s_n]$$

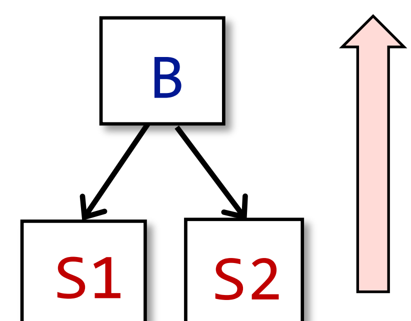
$$\mathrm{OUT}[B] = f_B(\mathrm{IN}[B]),\ f_B = f_{s_n} \circ \ldots \circ f_{s_2} \circ f_{s_1}$$

$$\mathrm{IN}[B] = \bigwedge_{P \text{ a predecessor of } B} \mathrm{OUT}[P]$$



$$\mathrm{IN}[B] = f_B(\mathrm{OUT}[B]),\ f_B = f_{s_1} \circ \ldots \circ f_{s_{n-1}} \circ f_{s_n}$$

$$\mathrm{OUT}[B] = \bigwedge_{S \text{ a successor of } B} \mathrm{IN}[S]$$



Data Flow Analysis Applications

(I) Reaching Definitions Analysis

(II) Live Variables Analysis

(III) Available Expressions Analysis

Issues Not Covered

- Method Calls

  - Intra-procedural CFG

  - Will be introduced in lecture: Inter-procedural Analysis

- Aliases

  - Variables have no aliases

  - Will be introduced in lecture: Pointer Analysis

Reaching Definitions

A definition d at program point p *reaches* a point q if there is a path from p to q such that d is not "killed" along that path

Reaching Definitions

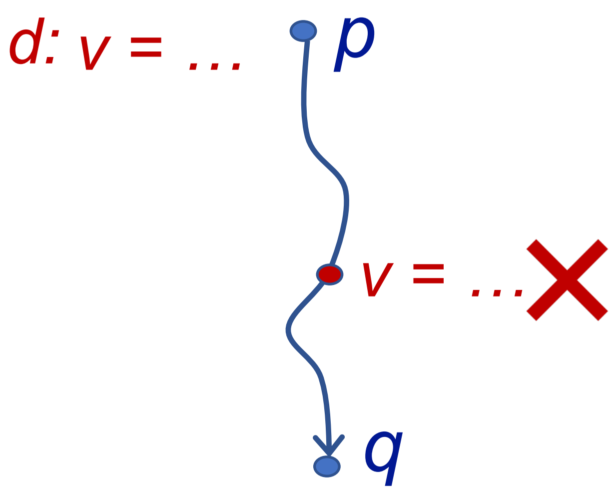
A definition d at program point p *reaches* a point q if there is a path from p to q such that d is not “killed” along that path

- A definition of a variable v is a statement that assigns a value to v

Reaching Definitions

A definition d at program point p *reaches* a point q if there is a path from p to q such that d is not “killed” along that path

- A definition of a variable v is a statement that assigns a value to v
- Translated as: definition of variable v at program point p reaches point q if there is a path from p to q such that no new definition of v appears on that path



Reaching Definitions

A definition d at program point p *reaches* a point q if there is a path from p to q such that d is not "killed" along that path

- A definition of a variable v is a statement that assigns a value to v
- Translated as: definition of variable v at program point p reaches point q if there is a path from p to q such that no new definition of v appears on that path
- Reaching definitions can be used to detect possible undefined variables. e.g., introduce a dummy definition for each variable v at the entry of CFG, and if the dummy definition of v reaches a point p where v is used, then v may be used before definition (as *undefined* reaches v)

Understanding Reaching Definitions

Abstraction

- Data Flow Values/Facts

  - The definitions of all the variables in a program

Understanding Reaching Definitions

Abstraction

- Data Flow Values/Facts

  - The definitions of all the variables in a program

  - Can be represented by bit vectors

    e.g., D1, D2, D3, D4, …, D100 (100 definitions)

$$\underbrace{00000\ldots0}_{\text{100 bits}}$$

Bit i from the left represents definition Di

Understanding Reaching Definitions

Safe-approximation

- Transfer Function

- Control Flow

Understanding Reaching Definitions

Safe-approximation

```
D: v = x op y
```

This statement "generates" a definition D of variable v and "kills" all the other definitions in the program that define variable v, while leaving the remaining incoming definitions unaffected.

- Transfer Function

- Control Flow

Understanding Reaching Definitions

Safe-approximation

```
D: v = x op y
```

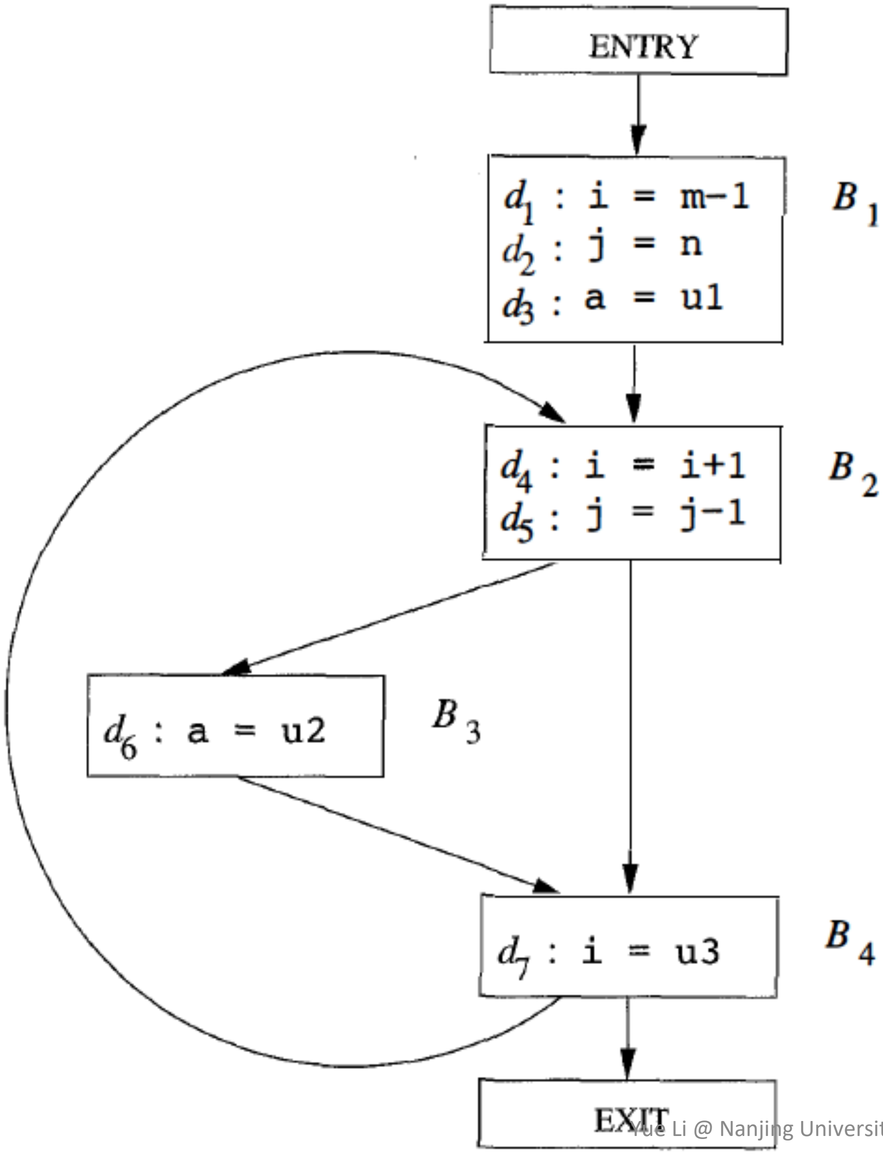
This statement "generates" a definition D of variable v and "kills" all the other definitions in the program that define variable v, while leaving the remaining incoming definitions unaffected.

- Transfer Function

$$\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B)$$

- Control Flow

$$\mathrm{OUT}[B] = gen_B \cup (\mathrm{IN}[B] - kill_B)$$



$gen_{B_1} = \{ d_1, d_2, d_3 \}$

$kill_{B_1} = \{ d_4, d_5, d_6, d_7 \}$

$gen_{B_2} = \{ d_4, d_5 \}$

$kill_{B_2} = \{ d_1, d_2, d_7 \}$

$gen_{B_3} = \{ d_6 \}$

$kill_{B_3} = \{ d_3 \}$

$gen_{B_4} = \{ d_7 \}$

$kill_{B_4} = \{ d_1, d_4 \}$

Understanding Reaching Definitions

```
D: v = x op y
```

Safe-approximation

This statement "generates" a definition D of variable v and "kills" all the other definitions in the program that define variable v, while leaving the remaining incoming definitions unaffected.

- Transfer Function

$$\boxed{OUT[B] = gen_B \cup (IN[B] - kill_B)}$$

- Control Flow

Understanding Reaching Definitions

Safe-approximation

`D: v = x op y`

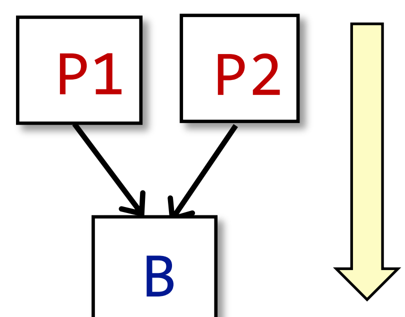
This statement "generates" a definition D of variable v and "kills" all the other definitions in the program that define variable v, while leaving the remaining incoming definitions unaffected.

- Transfer Function

$$OUT[B] = gen_B \cup (IN[B] - kill_B)$$

- Control Flow

$$IN[B] = \bigcup_{P \text{ a predecessor of } B} OUT[P]$$



Understanding Reaching Definitions

Safe-approximation

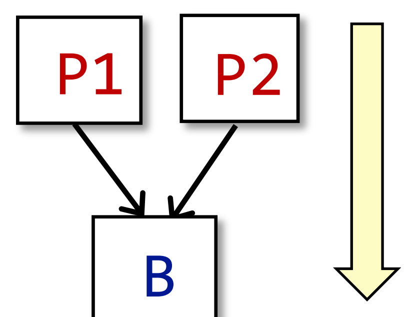
```
D: v = x op y
```

This statement "generates" a definition D of variable v and "kills" all the other definitions in the program that define variable v, while leaving the remaining incoming definitions unaffected.

A definition reaches a program point as long as there exists at least one path along which the definition reaches.

- Control Flow

$$IN[B] = \bigcup_{P \text{ a predecessor of } B} OUT[P]$$



Understanding Reaching Definitions

Safe-approximation

$$\boxed{\texttt{D: v = x } \mathit{op} \texttt{ y}}$$

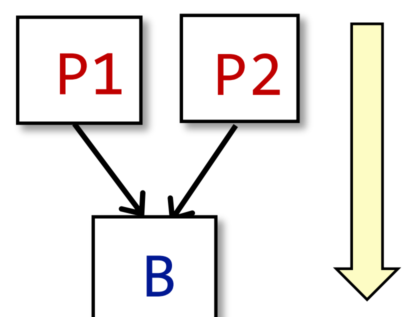
This statement "generates" a definition D of variable v and "kills" all the other definitions in the program that define variable v, while leaving the remaining incoming definitions unaffected.

- Transfer Function

$$\boxed{\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B)}$$

- Control Flow

$$\boxed{\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P]}$$



Algorithm of Reaching Definitions Analysis

INPUT: CFG ($kill_B$ and gen_B computed for each basic block B)

OUTPUT: IN[B] and OUT[B] for each basic block B

METHOD:

OUT[*entry*] = ∅;
for (each basic block *B\entry*)
    OUT[B] = ∅;
while (changes to any OUT occur)
    for (each basic block *B\entry*) {
        $\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P]$;
        $\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B)$;
    }

Algorithm of Reaching Definitions Analysis

INPUT: CFG ($kill_B$ and gen_B computed for each basic block B)

OUTPUT: IN[B] and OUT[B] for each basic block B

METHOD:

OUT[*entry*] = ∅; **?**

for (each basic block $B \setminus entry$)

    OUT[B] = ∅;

while (changes to any OUT occur)

    for (each basic block $B \setminus entry$) {

        IN[B] = $\bigcup_{P \text{ a predecessor of } B}$ OUT[P];

        OUT[B] = $gen_B \cup (\text{IN}[B] - kill_B)$;

    }

Algorithm of Reaching Definitions Analysis

INPUT: CFG ($kill_B$ and gen_B computed for each basic block B)

OUTPUT: IN[B] and OUT[B] for each basic block B

METHOD:

OUT[*entry*] = $\emptyset$;
for (each basic block $B \setminus entry$) ?
    OUT[B] = $\emptyset$;
while (changes to any OUT occur)
    for (each basic block $B \setminus entry$) {
        IN[B] = $\bigcup_{P \text{ a predecessor of } B}$ OUT[P];
        OUT[B] = $gen_B \cup$ (IN[B] - $kill_B$);
    }

Why entry is excluded?

Algorithm of Reaching Definitions Analysis

INPUT: CFG ($kill_B$ and gen_B computed for each basic block B)

OUTPUT: IN[B] and OUT[B] for each basic block B

METHOD:

```
OUT[entry] = ∅;
for (each basic block B\entry)
    OUT[B] = ∅; ?
while (changes to any OUT occur)
    for (each basic block B\entry) {
        IN[B] = ∪_{P a predecessor of B} OUT[P];
        OUT[B] = gen_B ∪ (IN[B] - kill_B);
    }
```

$$\mathrm{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \mathrm{OUT}[P]$$

$$\mathrm{OUT}[B] = gen_B \cup (\mathrm{IN}[B] - kill_B)$$

Algorithm of Reaching Definitions Analysis

INPUT: CFG ($kill_B$ and gen_B computed for each basic block B)

OUTPUT: IN[B] and OUT[B] for each basic block B

METHOD:

OUT[*entry*] = ∅;
for (each basic block $B \backslash entry$)
    OUT[B] = ∅;
while (changes to any OUT occur) ?
    for (each basic block $B \backslash entry$) {
        $\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P]$;
        $\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B)$;
    }

Why this iterative algorithm can finally stop?

Algorithm of Reaching Definitions Analysis

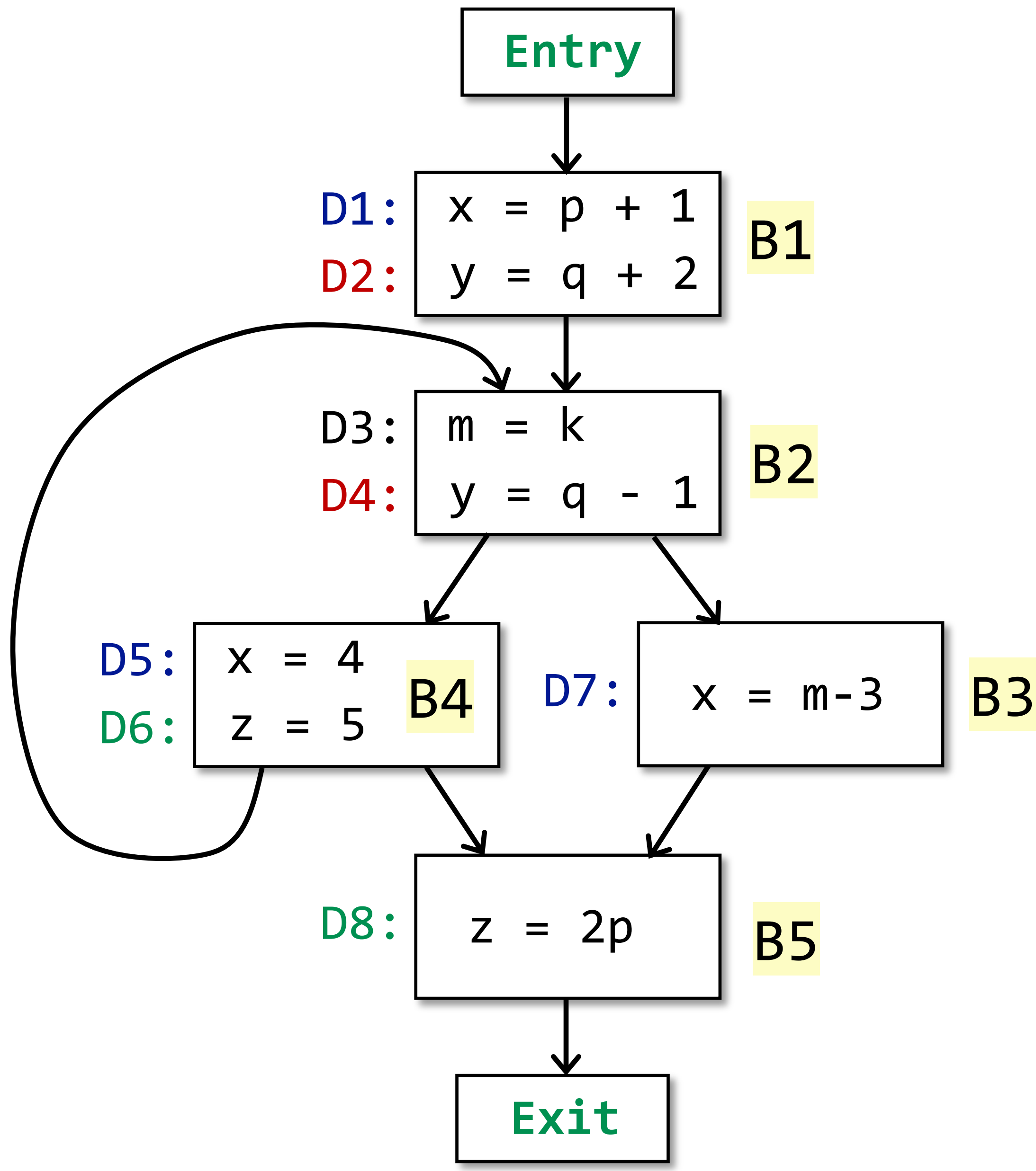
INPUT: CFG ($kill_B$ and gen_B computed for each basic block B)

OUTPUT: IN[B] and OUT[B] for each basic block B

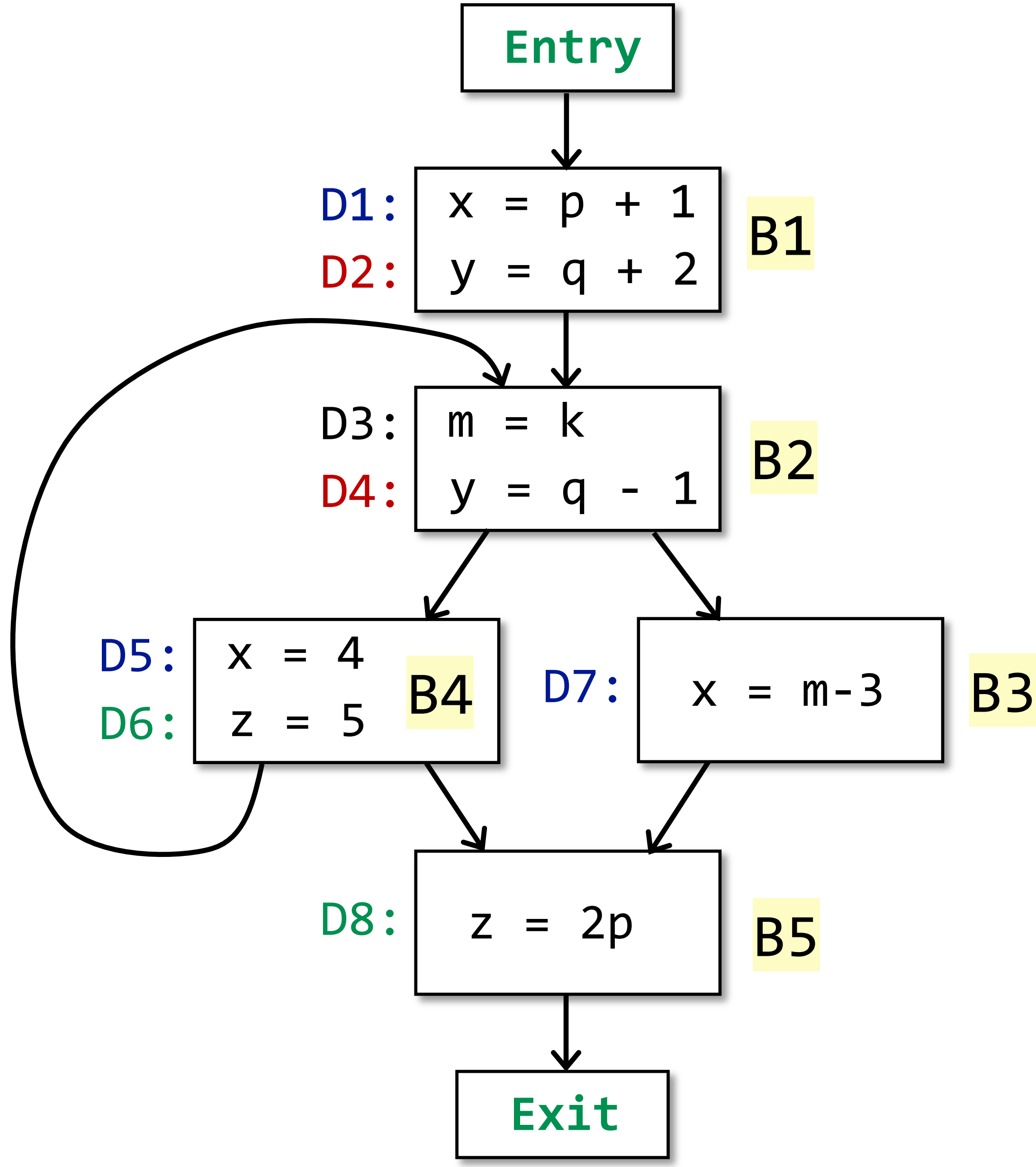
METHOD:

```
OUT[entry] = ∅;
for (each basic block B\entry)
    OUT[B] = ∅;
while (changes to any OUT occur)
    for (each basic block B\entry) {
        IN[B] = ∪_{P a predecessor of B} OUT[P];
        OUT[B] = gen_B ∪ (IN[B] - kill_B);
    }
```

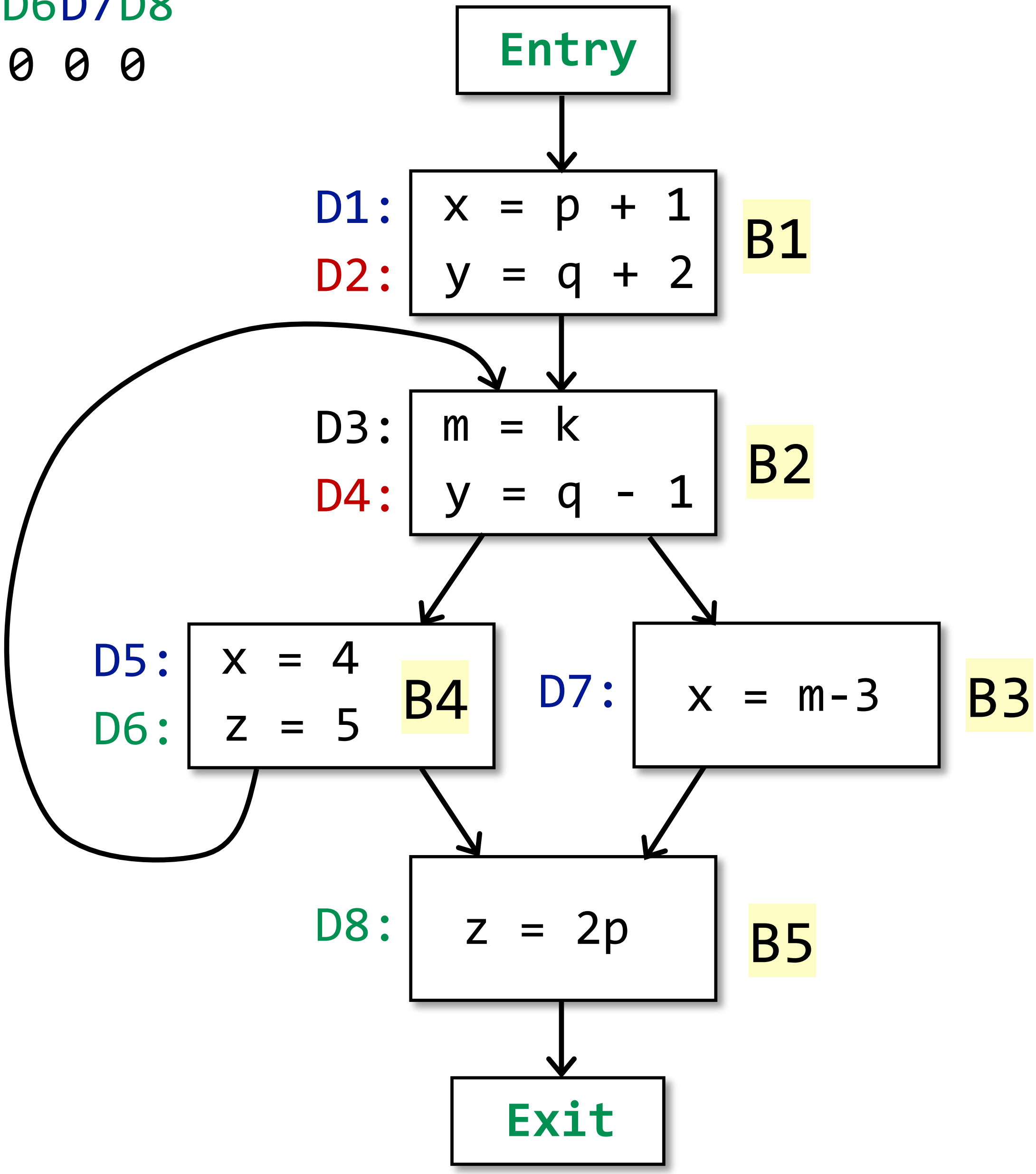
$$\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P]; \qquad \text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B);$$


Entry
D1:
x = p + 1
D2:
y = q + 2
B1
D3:
m = k
D4:
y = q - 1
B2
D5:
x = 4
D6:
z = 5
B4
D7:
x = m-3
B3
D8:
z = 2p
B5
Exit

```
D1
D2
do {
  D3
  D4
  if(…) {
    D5
    D6
  } else {
    D7
    break
  }
} while(…)
D8
```



D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0



Algorithm of Reaching Definitions Analysis

INPUT: CFG ($kill_B$ and gen_B computed for each basic block B)

OUTPUT: IN[B] and OUT[B] for each basic block B

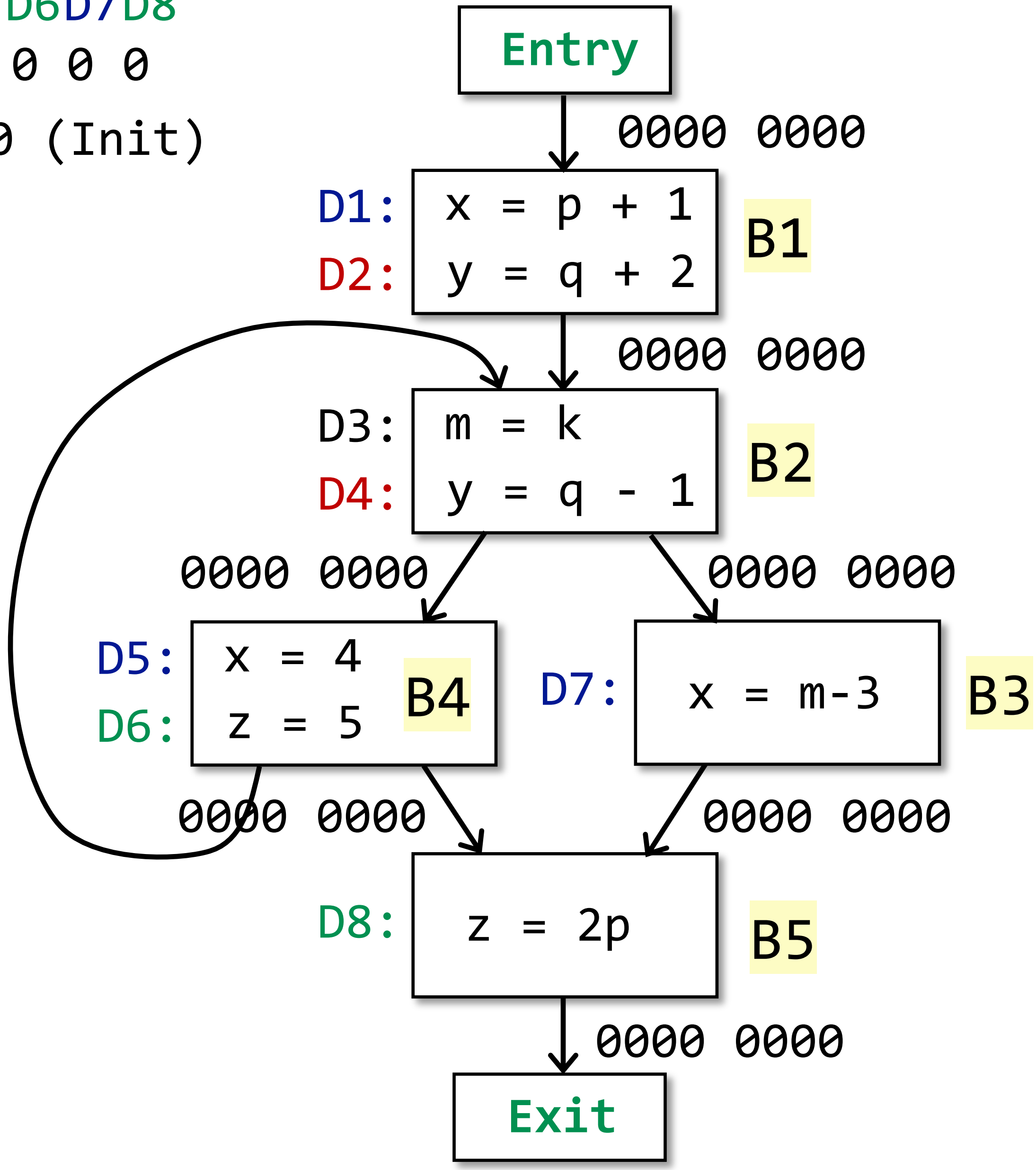
METHOD:

OUT[*entry*] = ∅;
for (each basic block *B\entry*)
    OUT[*B*] = ∅;
while (changes to any OUT occur)
    for (each basic block *B\entry*) {
        $\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P]$;
        $\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B)$;
    }

D1D2D3D4D5D6D7D8

0 0 0 0 0 0 0 0

Iteration 0 (Init)



Algorithm of Reaching Definitions Analysis

INPUT: CFG ($kill_B$ and gen_B computed for each basic block B)

OUTPUT: IN[B] and OUT[B] for each basic block B

METHOD:

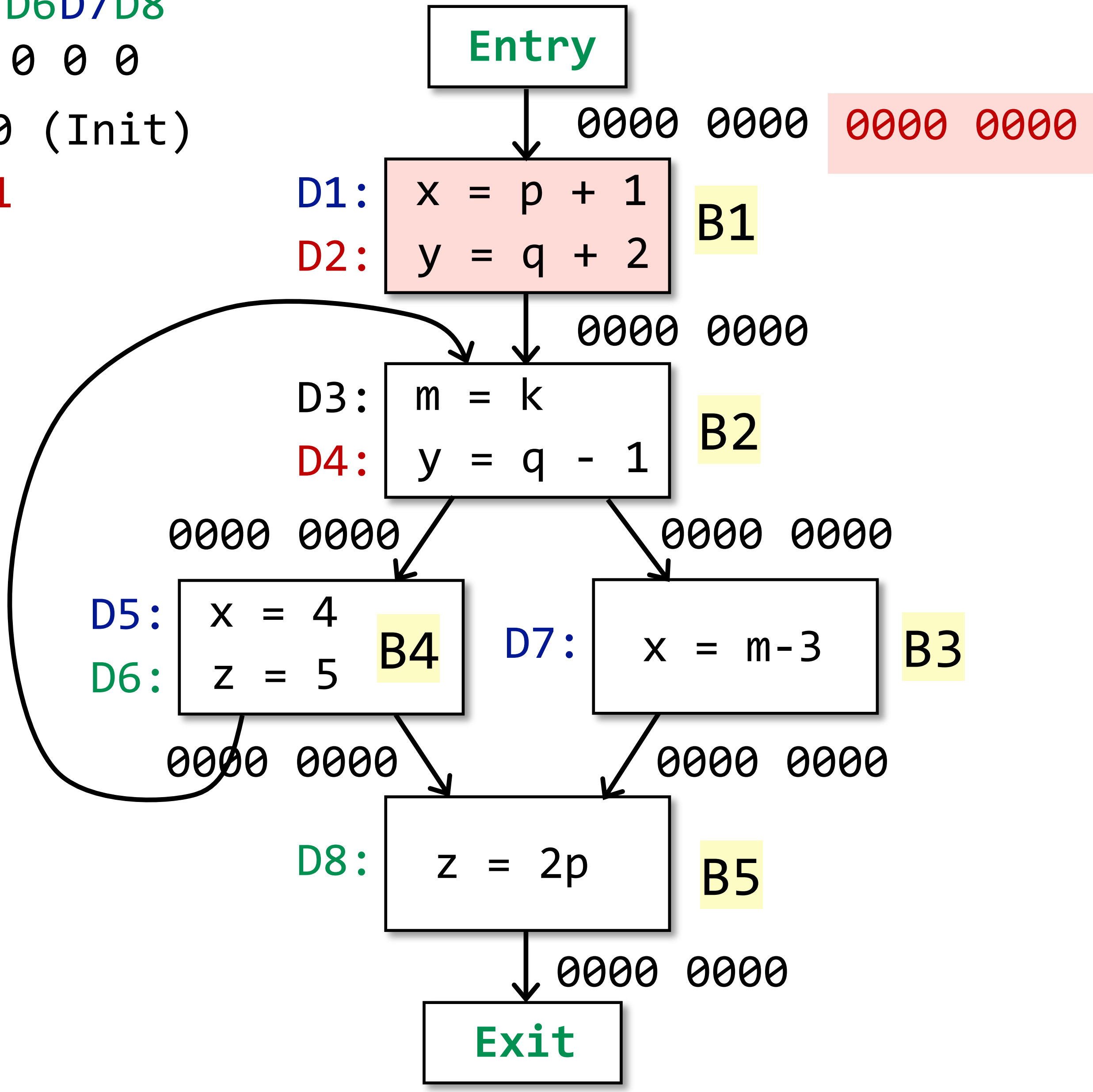
OUT[*entry*] = $\emptyset$;
for (each basic block $B \backslash entry$)
    OUT[B] = $\emptyset$;
while (changes to any OUT occur)
    for (each basic block $B \backslash entry$) {
        IN[B] = $\bigcup_{P \text{ a predecessor of } B}$ OUT[P];
        OUT[B] = $gen_B \cup$ (IN[B] - $kill_B$);
    }

D1D2D3D4D5D6D7D8

0 0 0 0 0 0 0 0

Iteration 0 (Init)

Iteration 1

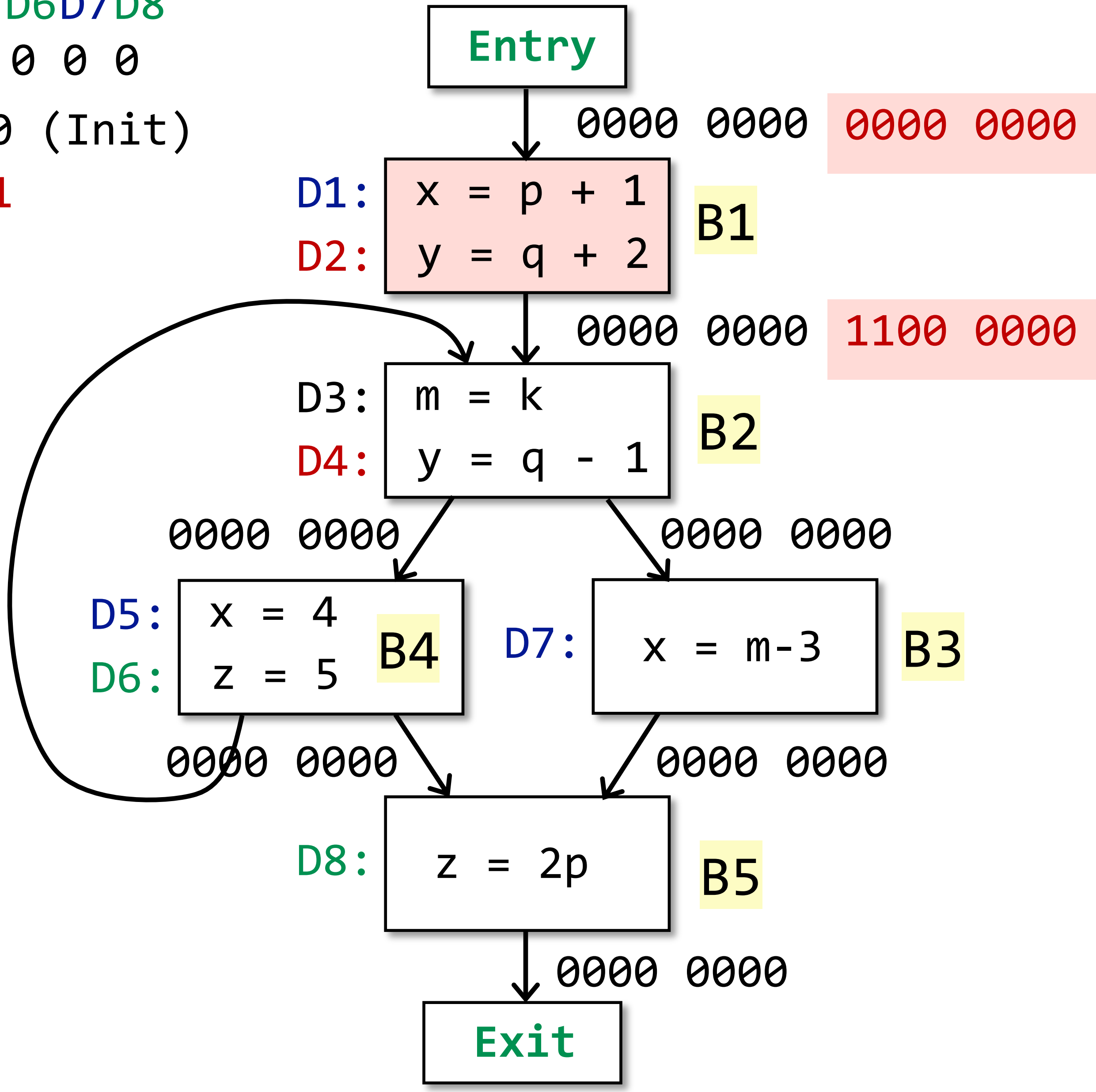


$\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B);$

D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0

Iteration 0 (Init)

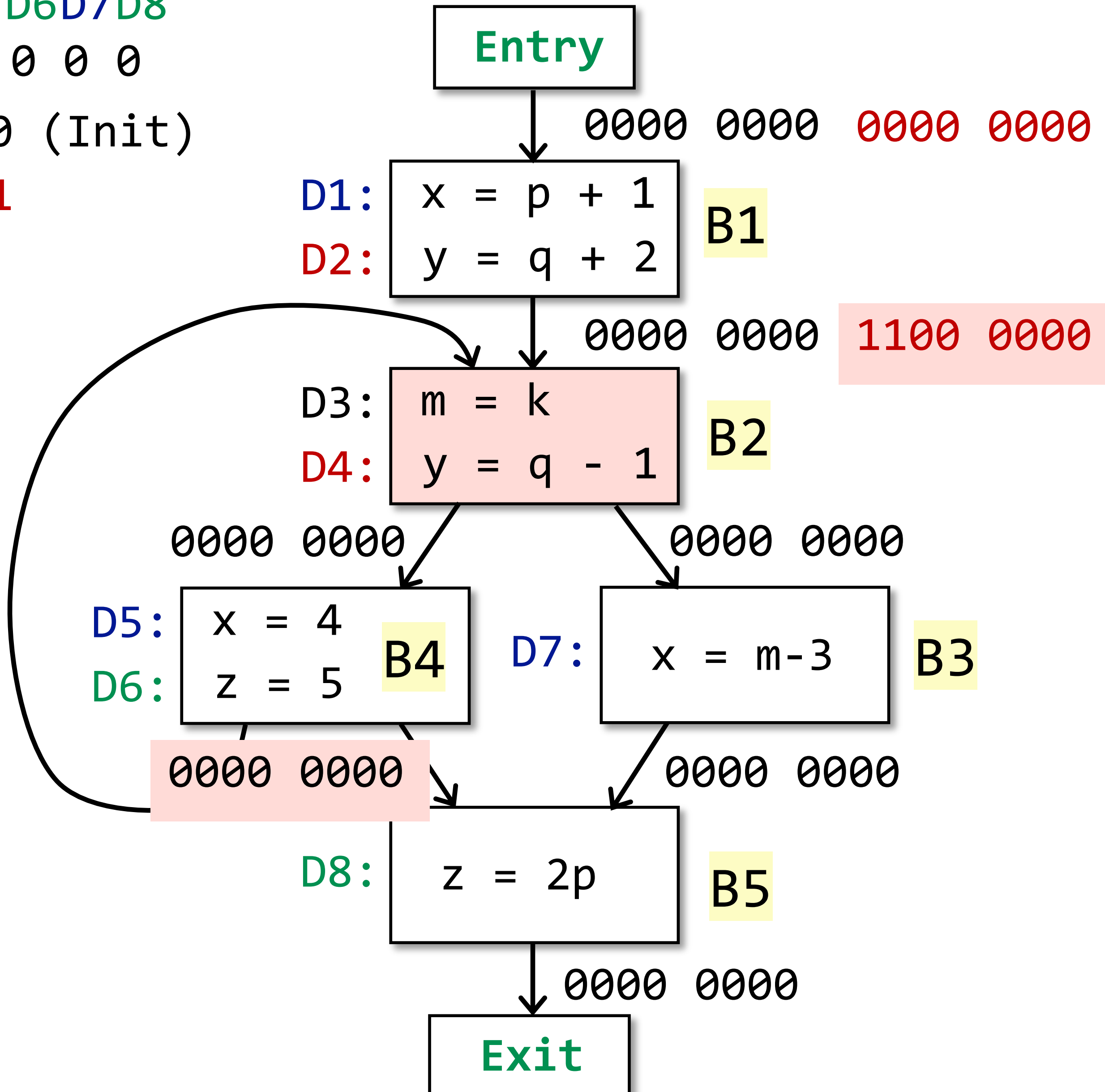
Iteration 1



D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0

Iteration 0 (Init)

Iteration 1

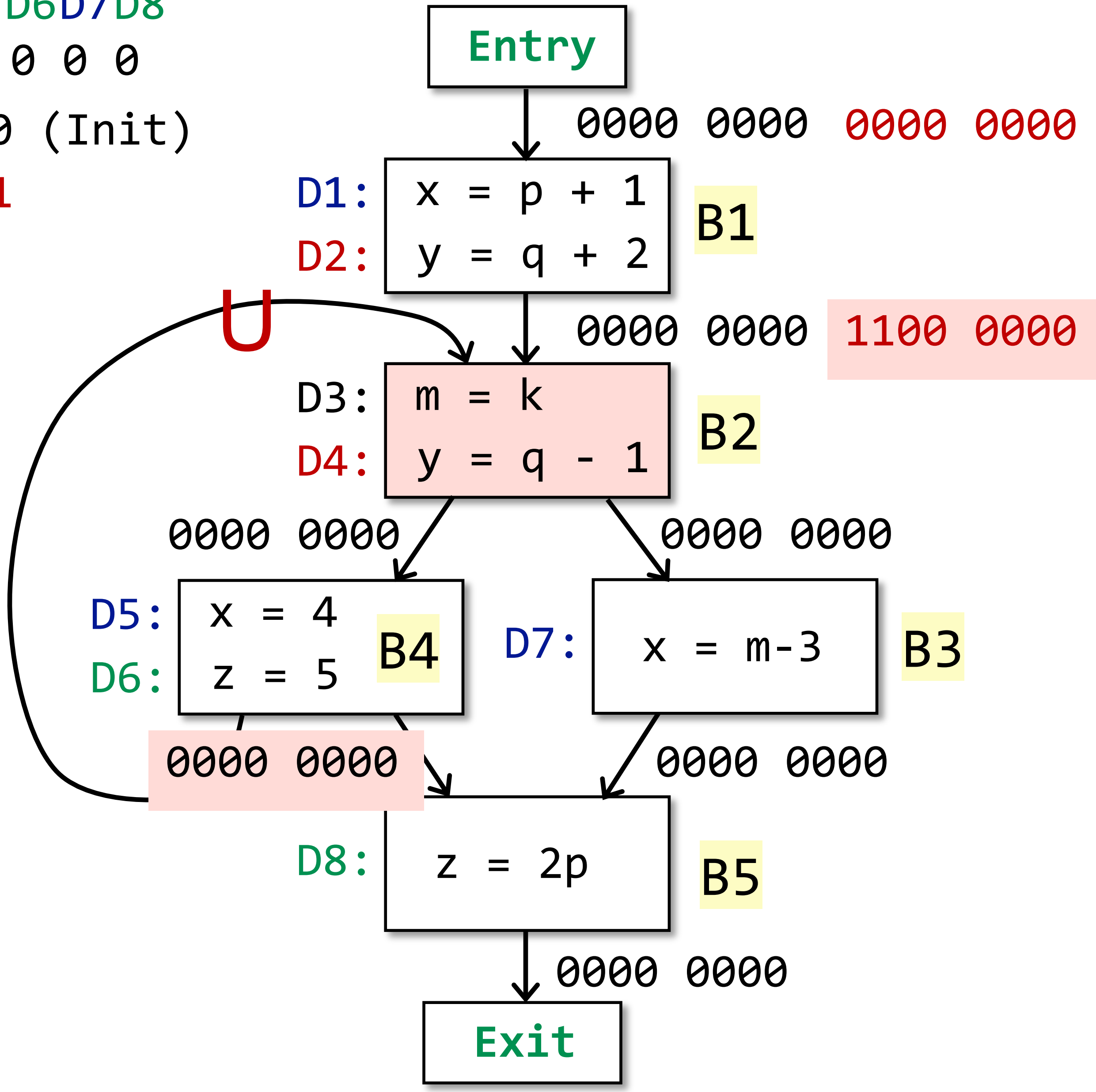


$$IN[B] = \bigcup_{P \text{ a predecessor of } B} OUT[P];$$

D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0

Iteration 0 (Init)

Iteration 1

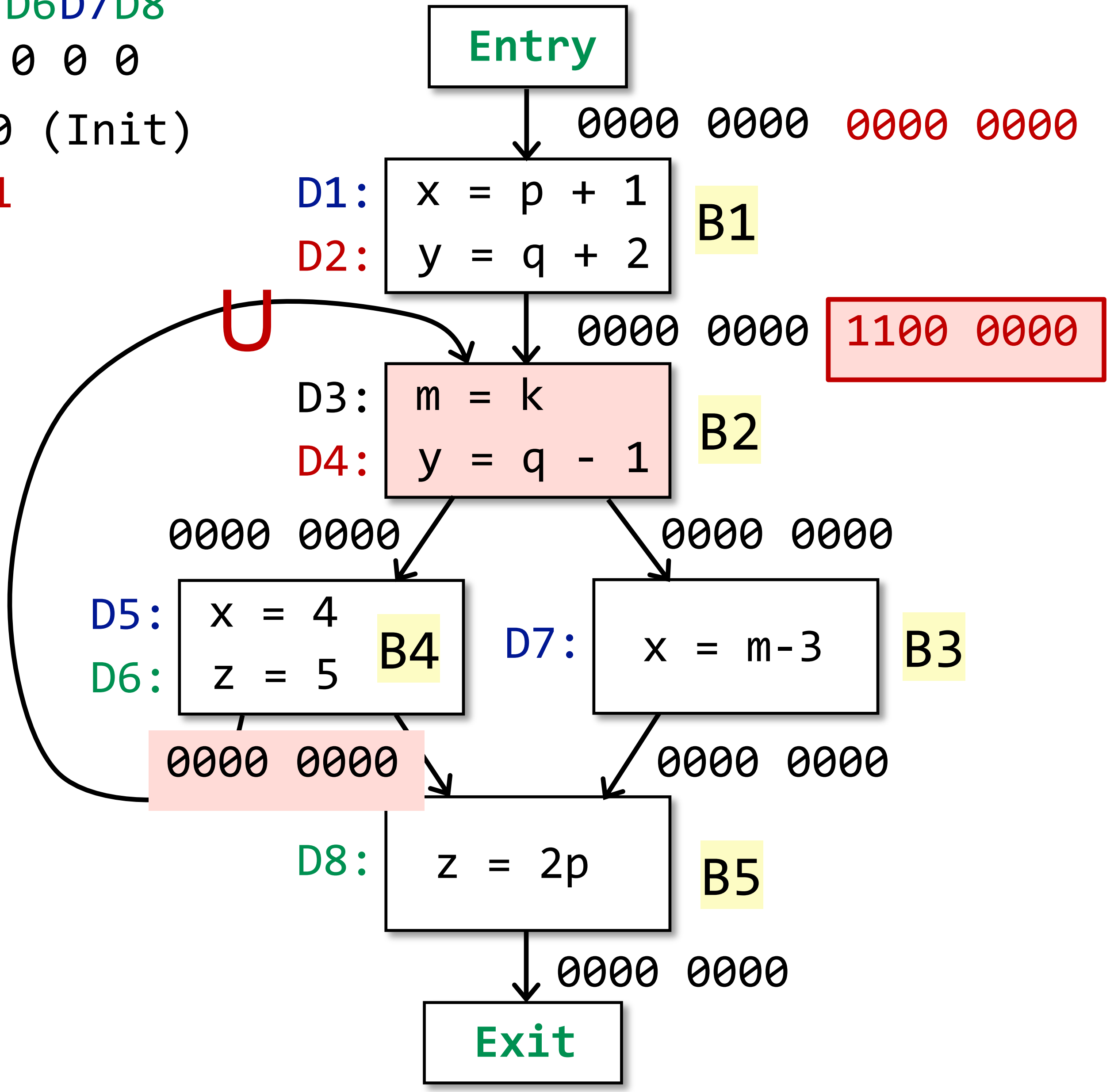


D1D2D3D4D5D6D7D8

0 0 0 0 0 0 0 0

Iteration 0 (Init)

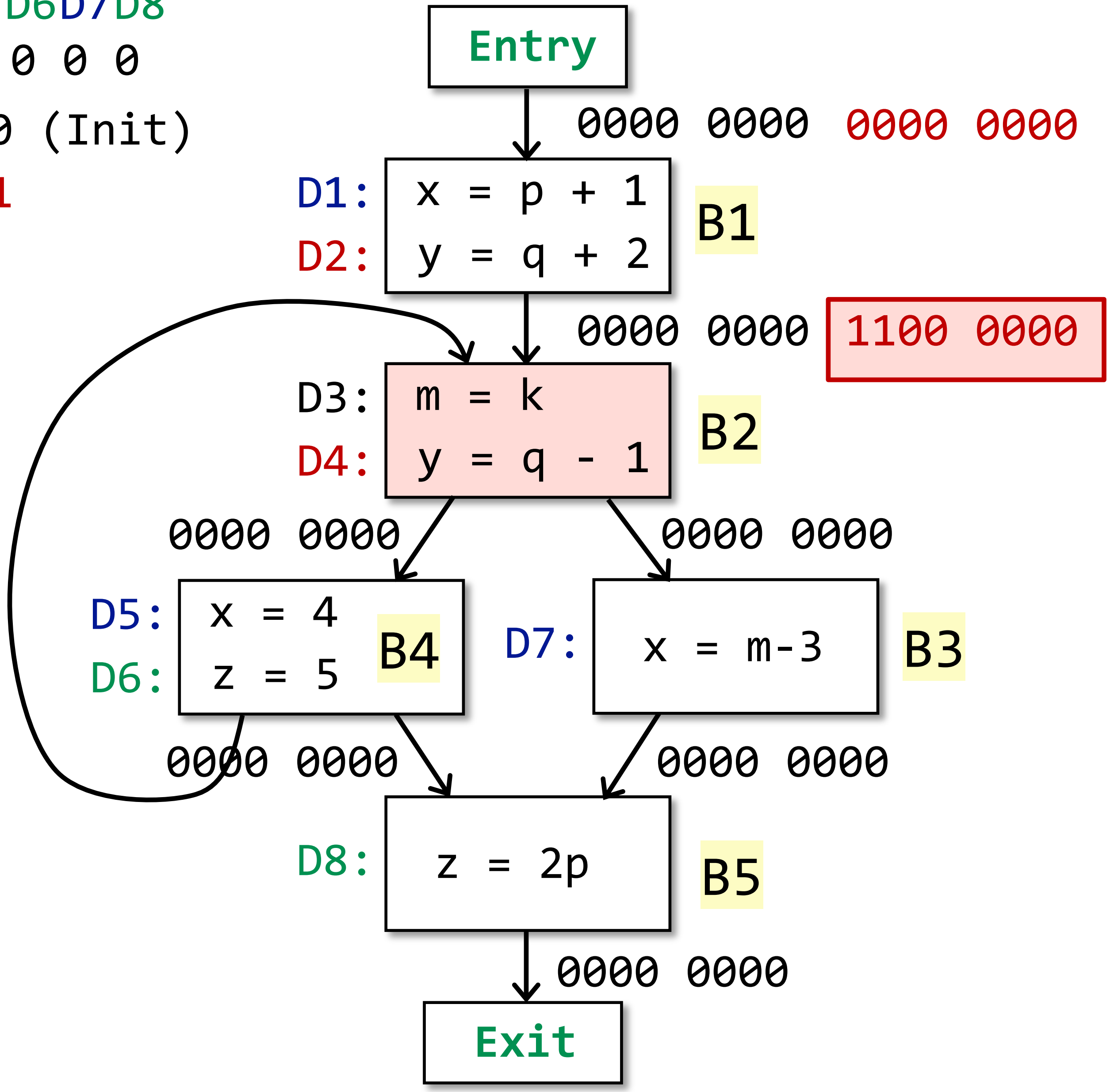
Iteration 1



D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0

Iteration 0 (Init)

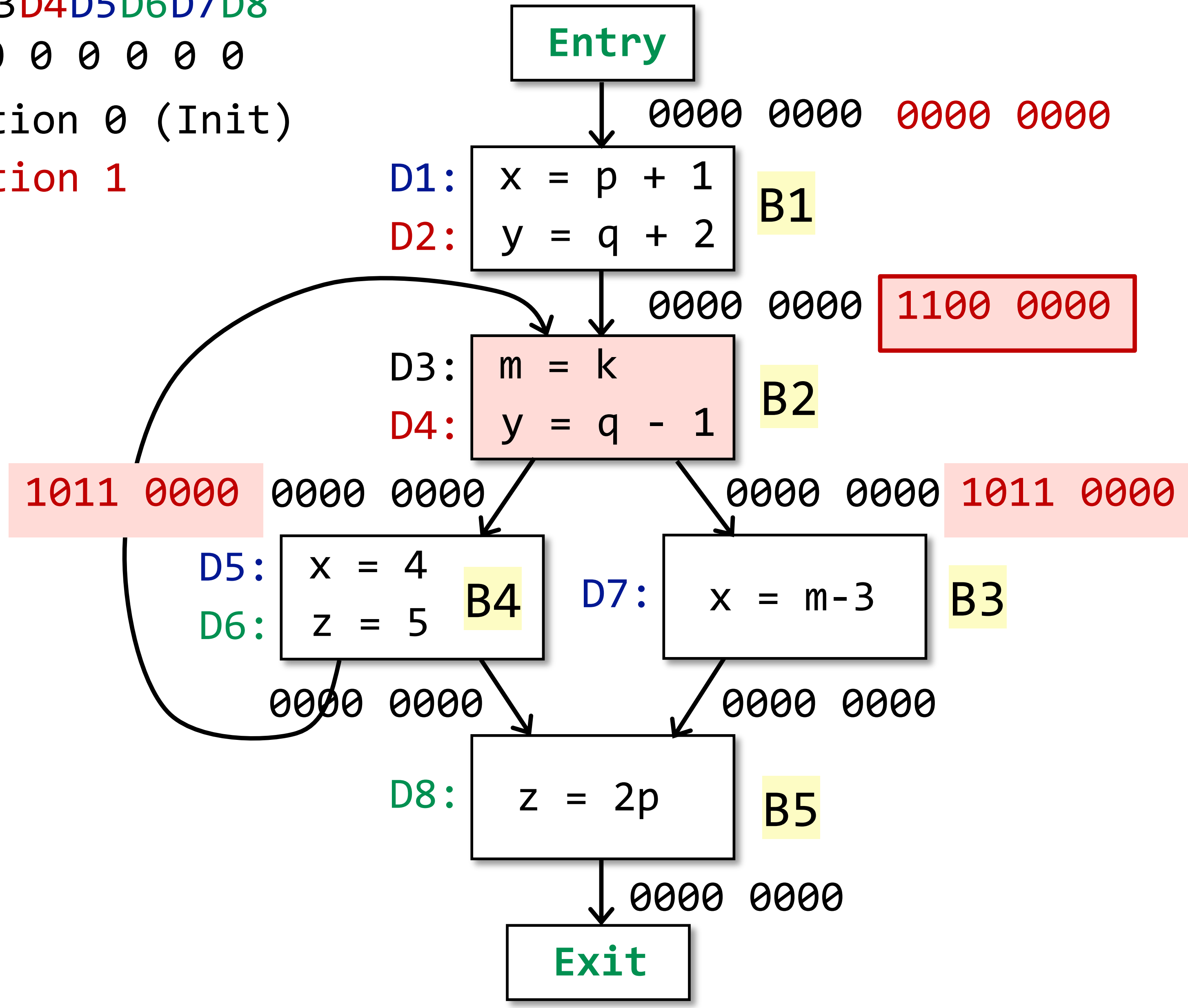
Iteration 1



D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0

Iteration 0 (Init)

Iteration 1

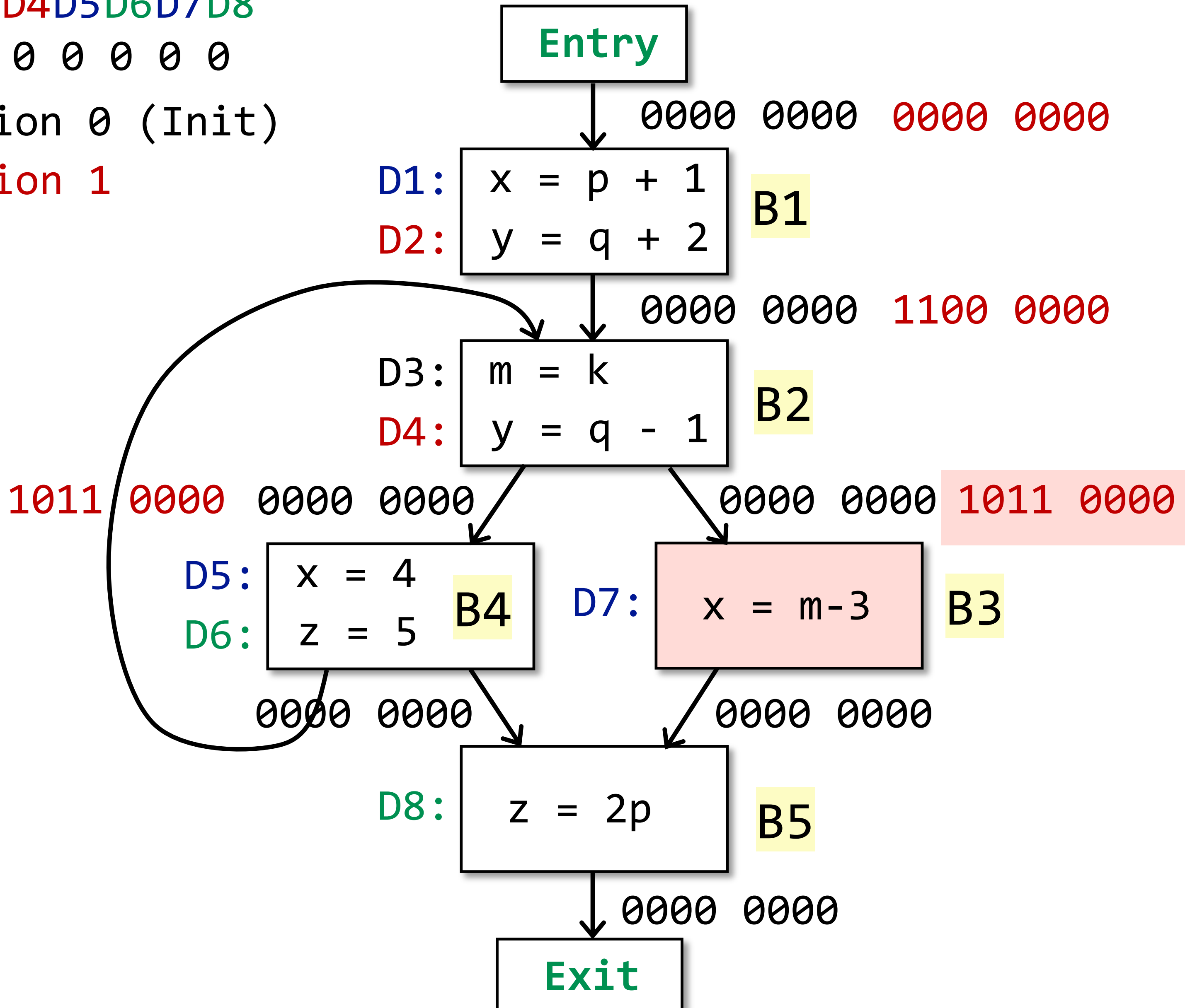


D1D2D3D4D5D6D7D8

0 0 0 0 0 0 0 0

Iteration 0 (Init)

Iteration 1

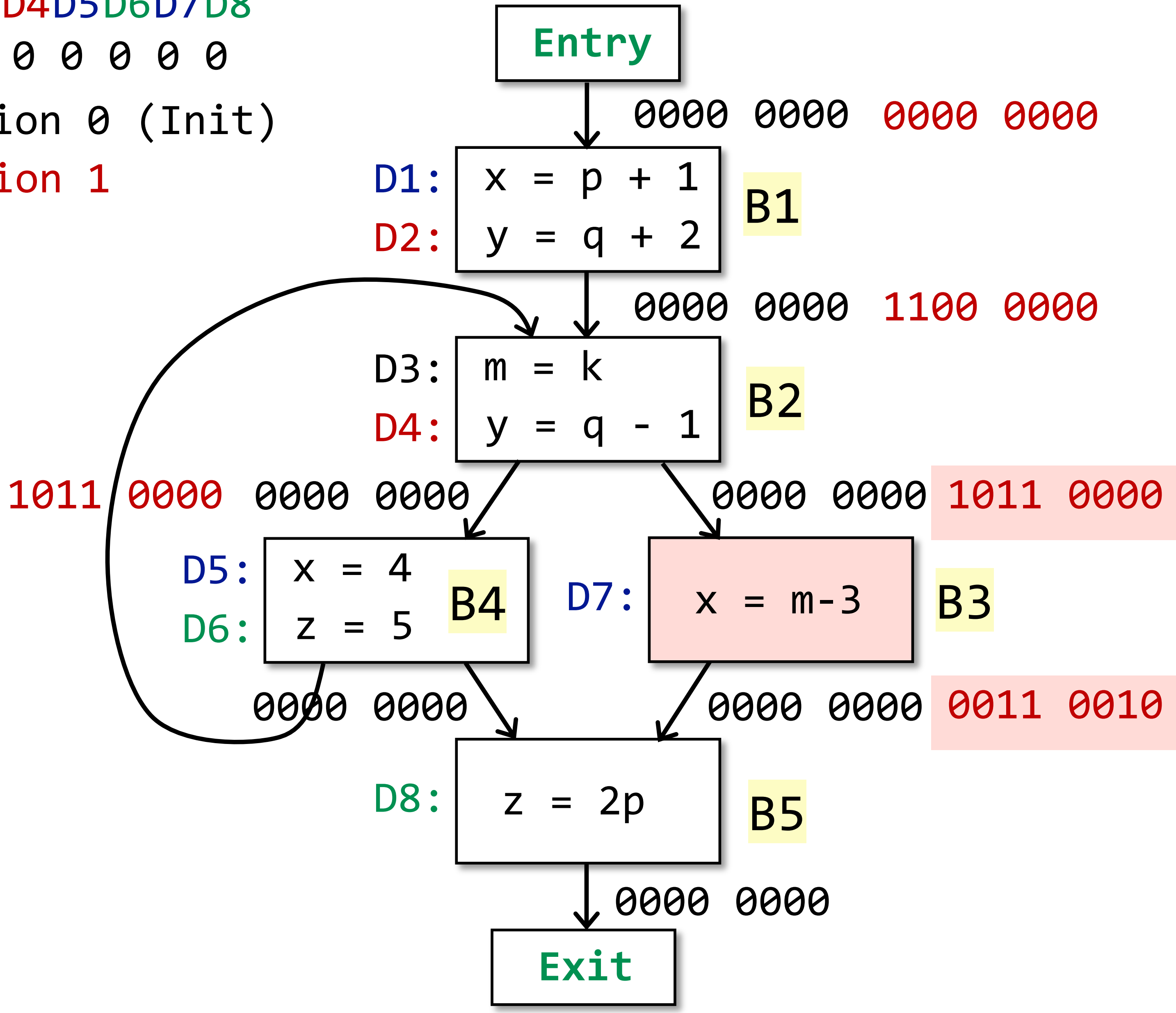


D1D2D3D4D5D6D7D8

0 0 0 0 0 0 0 0

Iteration 0 (Init)

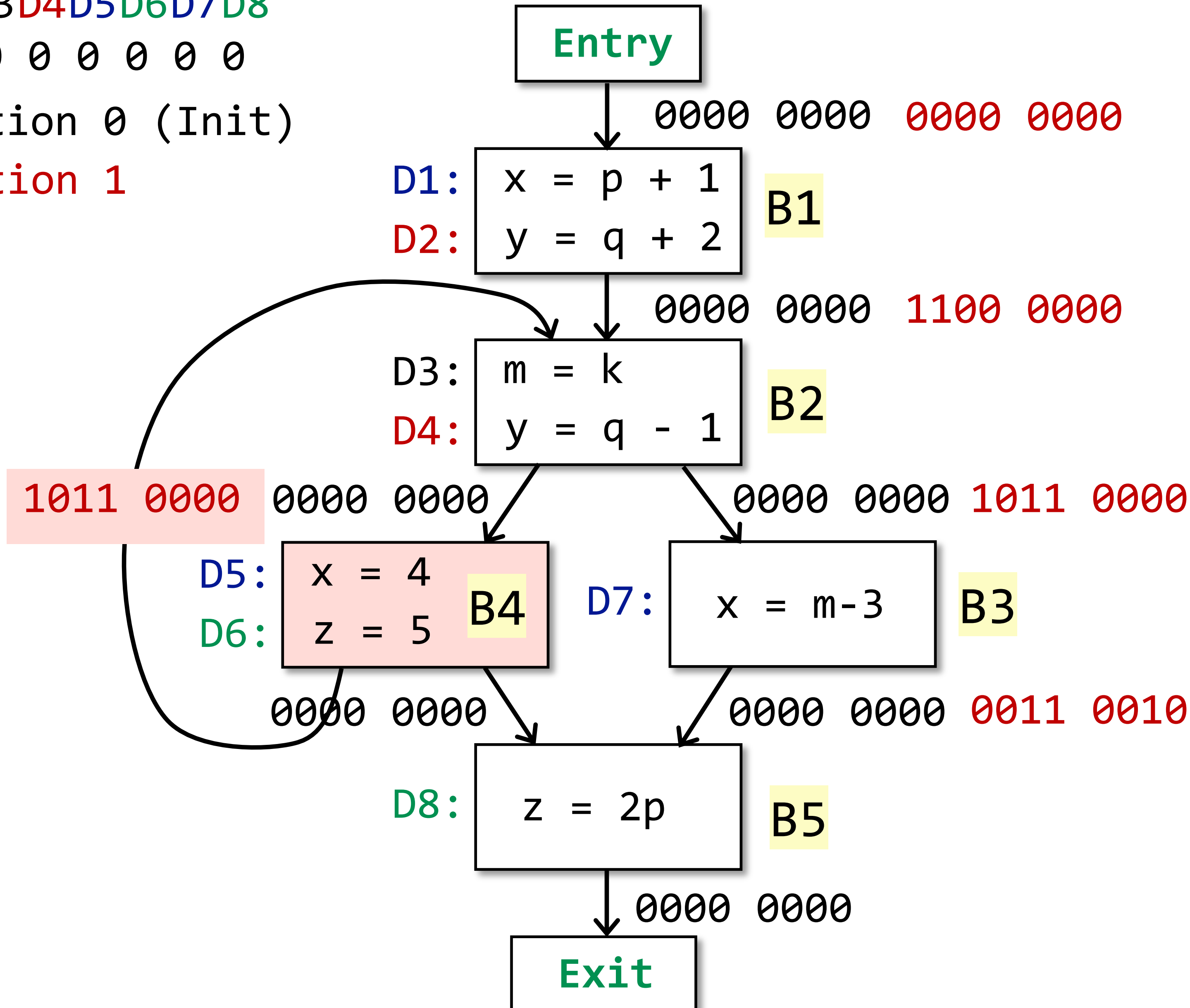
Iteration 1



D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0

Iteration 0 (Init)

Iteration 1

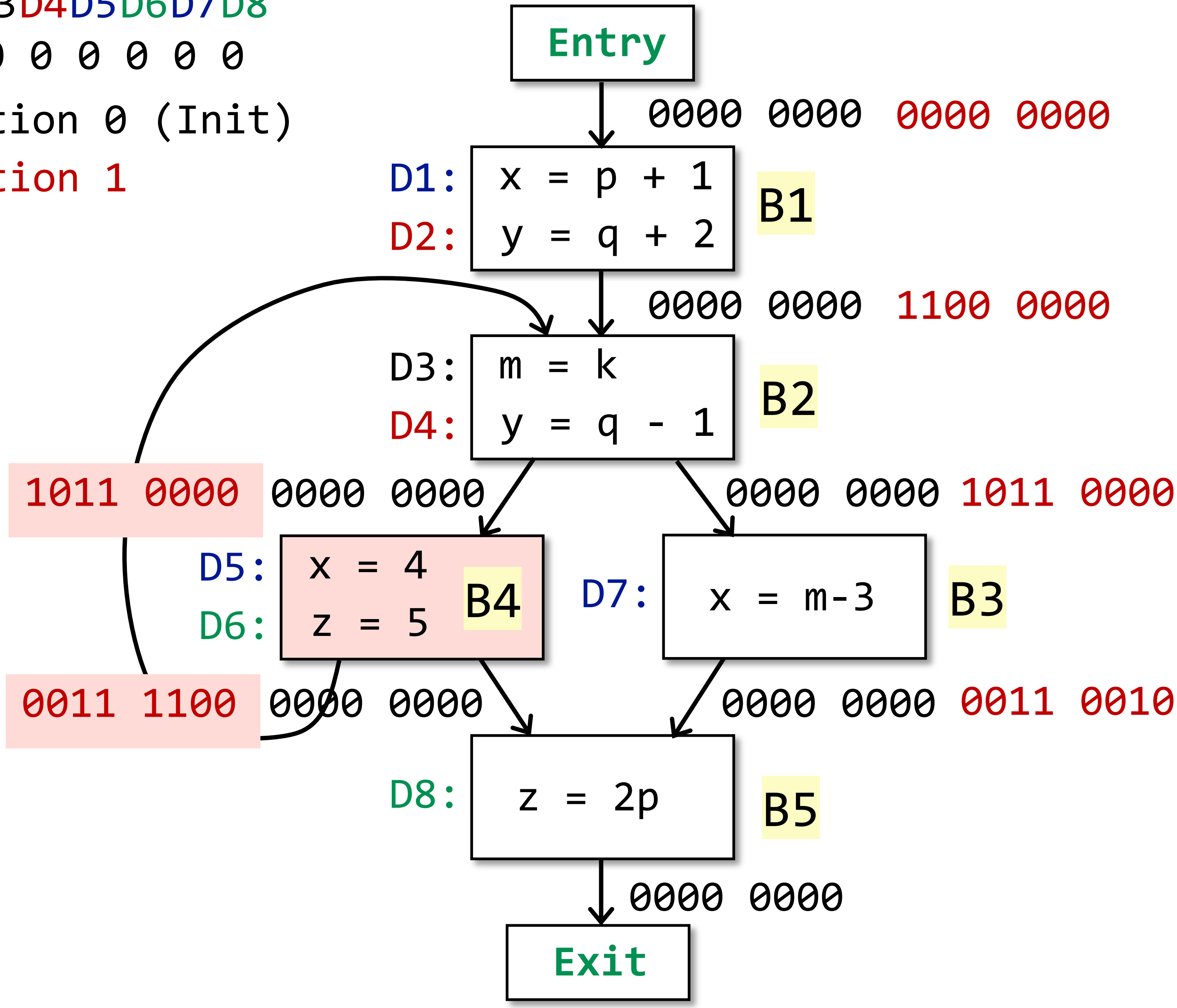


D1D2D3D4D5D6D7D8

0 0 0 0 0 0 0 0

Iteration 0 (Init)

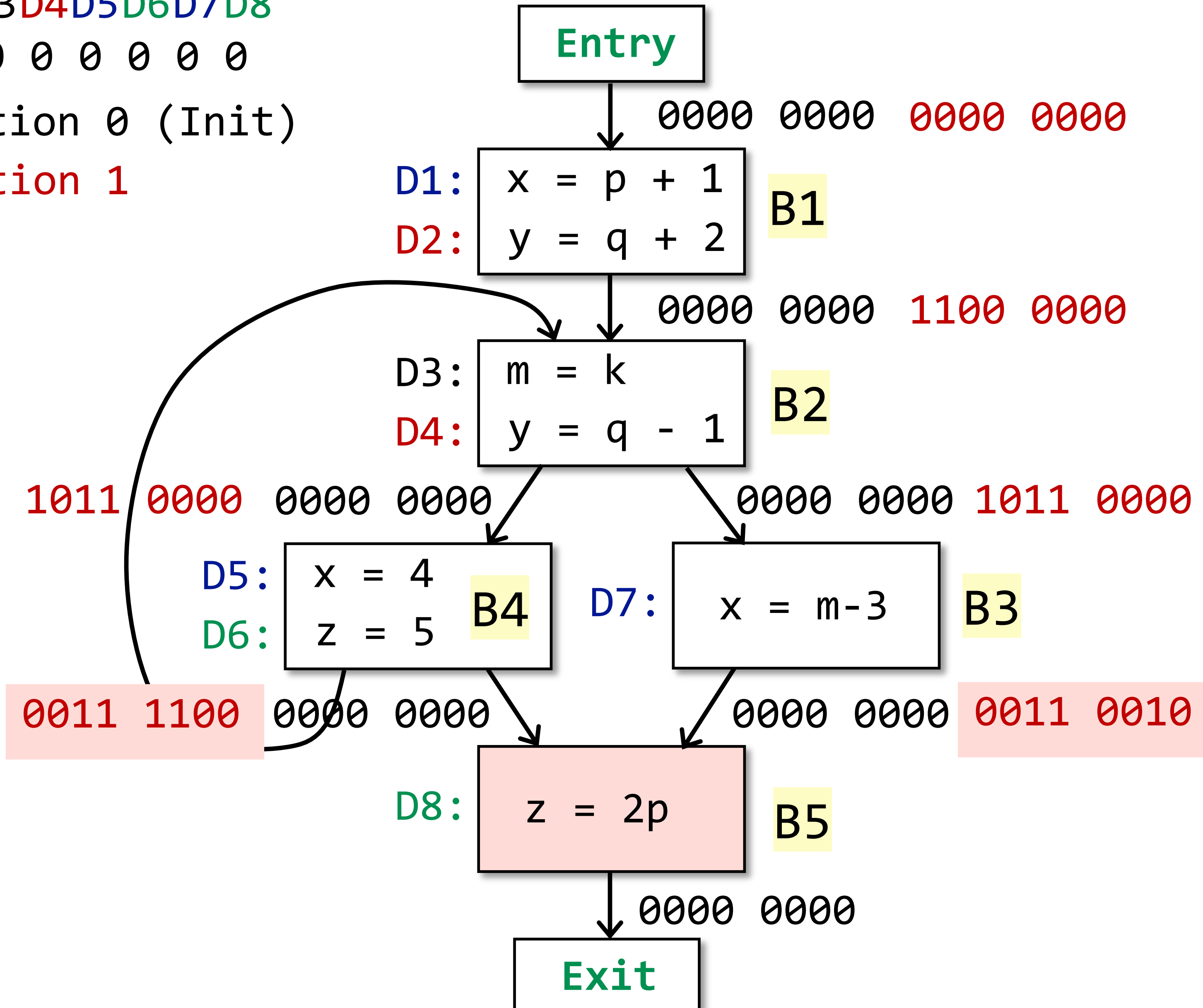
Iteration 1



D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0

Iteration 0 (Init)

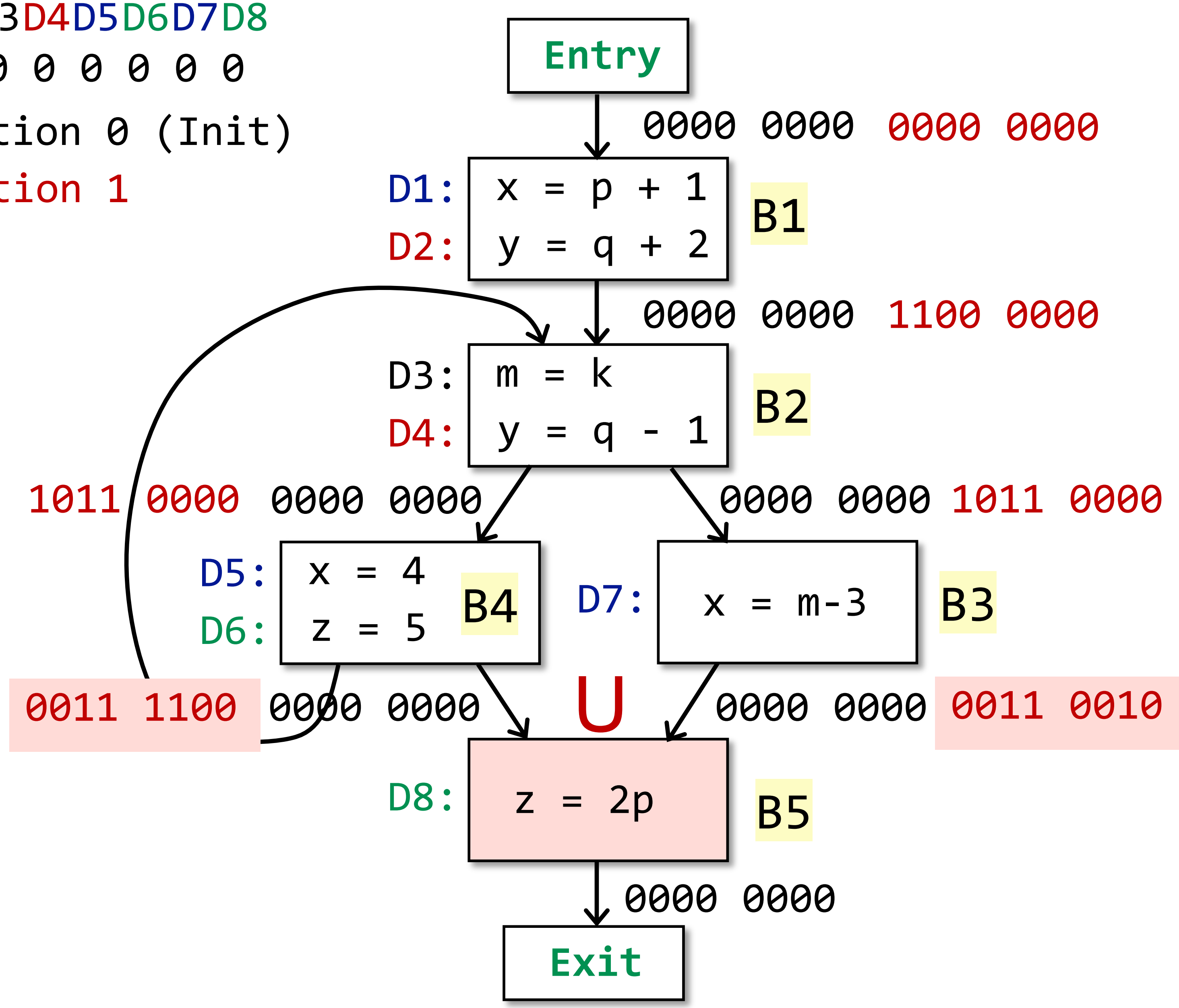
Iteration 1



D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0

Iteration 0 (Init)

Iteration 1

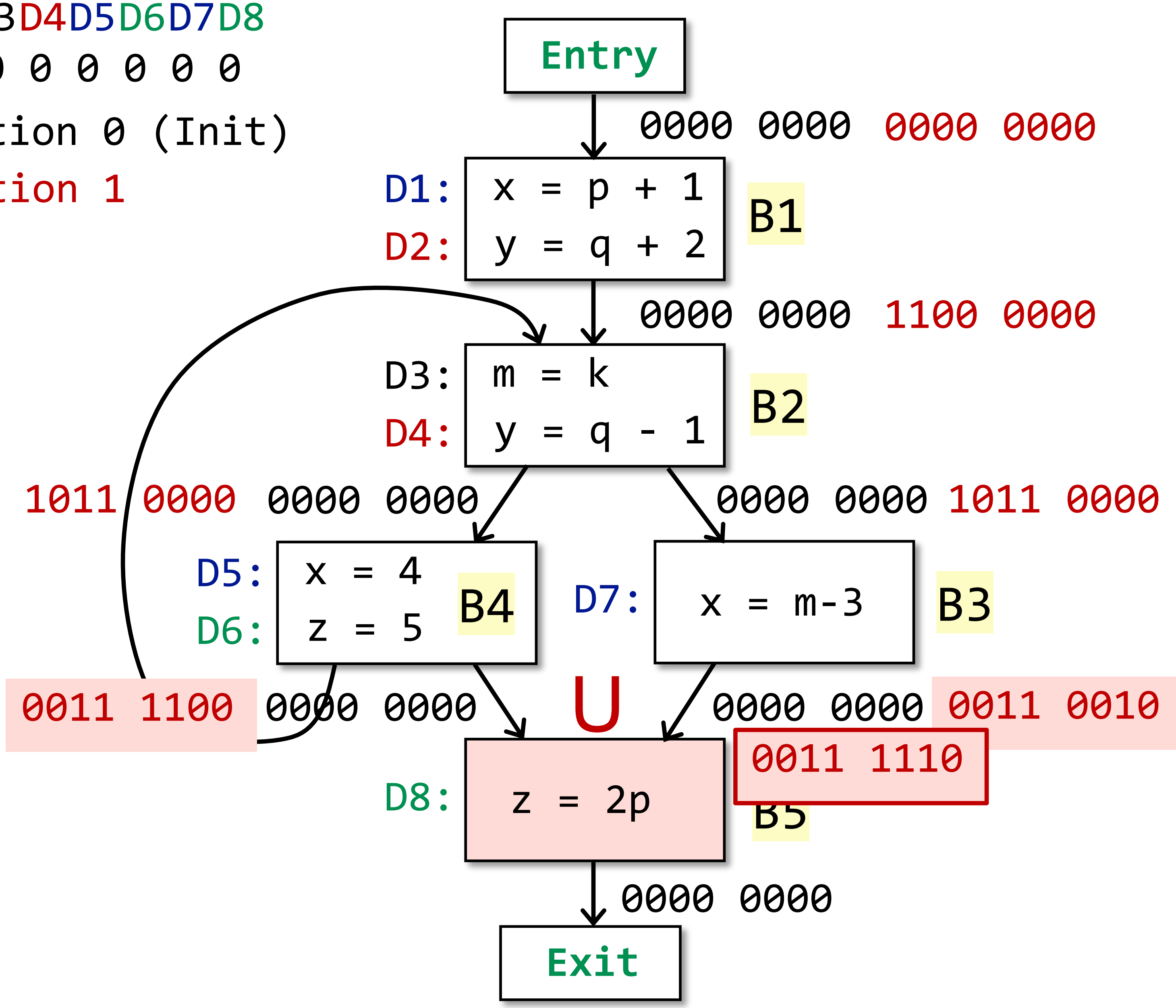


D1D2D3D4D5D6D7D8

0 0 0 0 0 0 0 0

Iteration 0 (Init)

Iteration 1

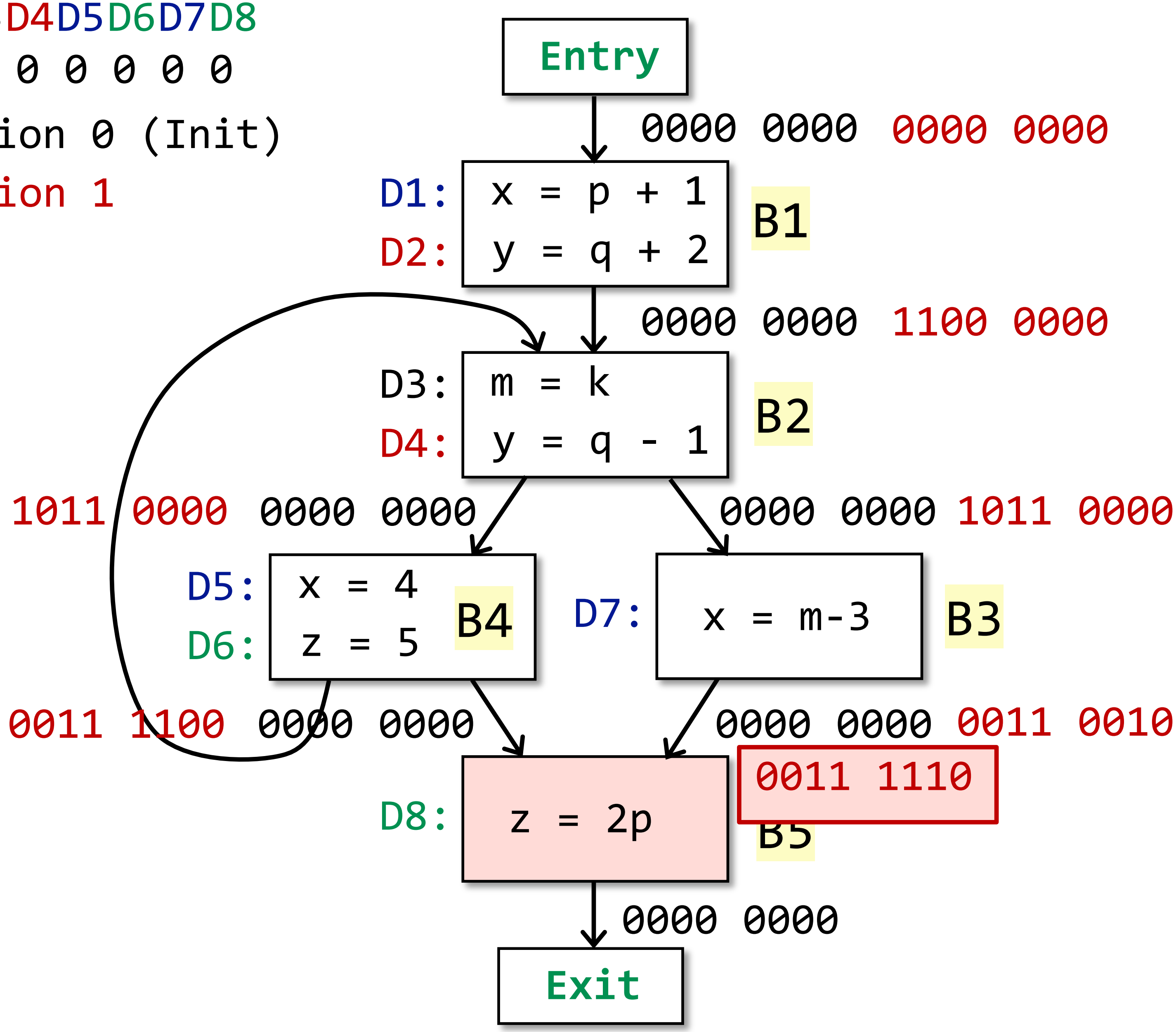


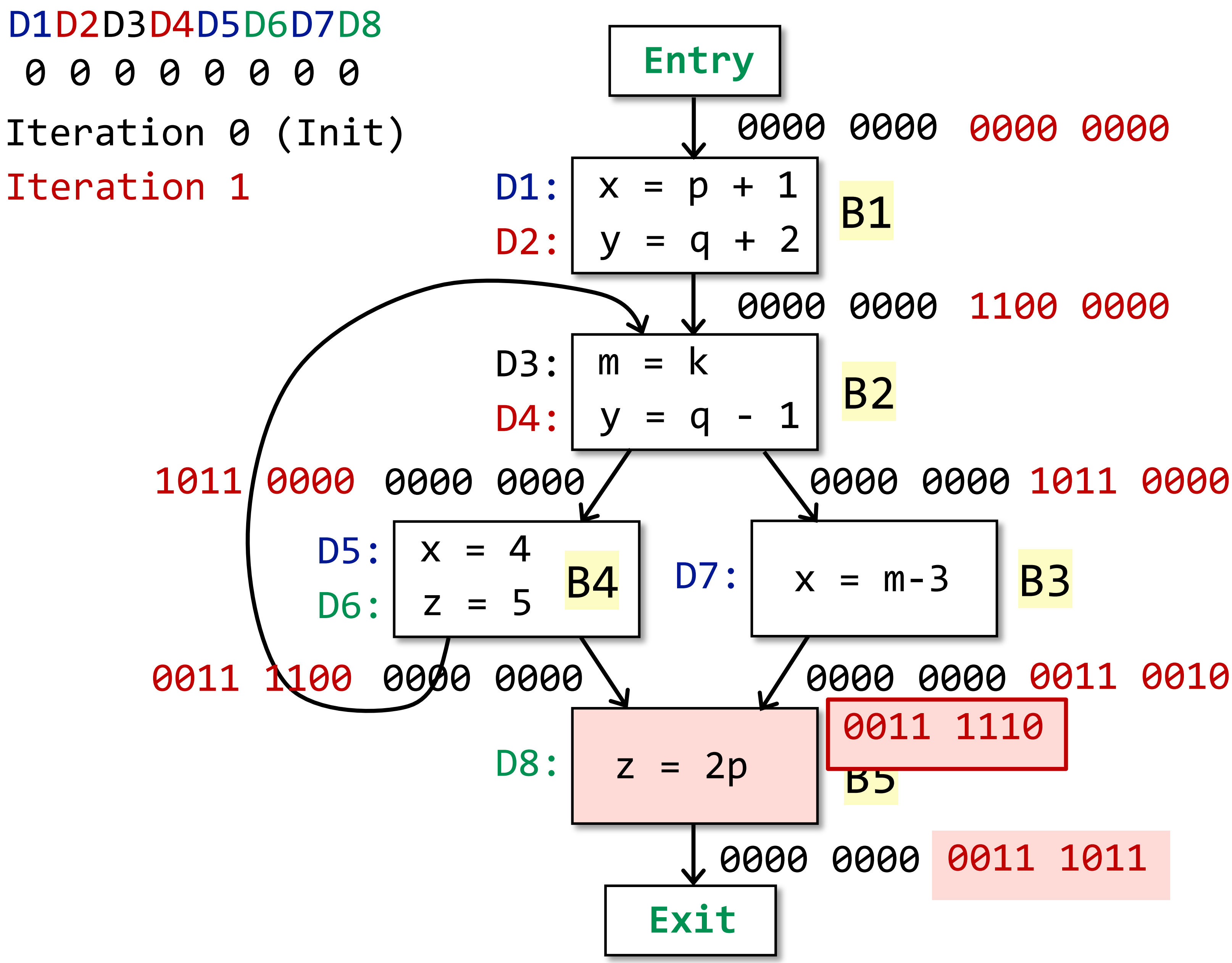
D1D2D3D4D5D6D7D8

0 0 0 0 0 0 0 0

Iteration 0 (Init)

Iteration 1





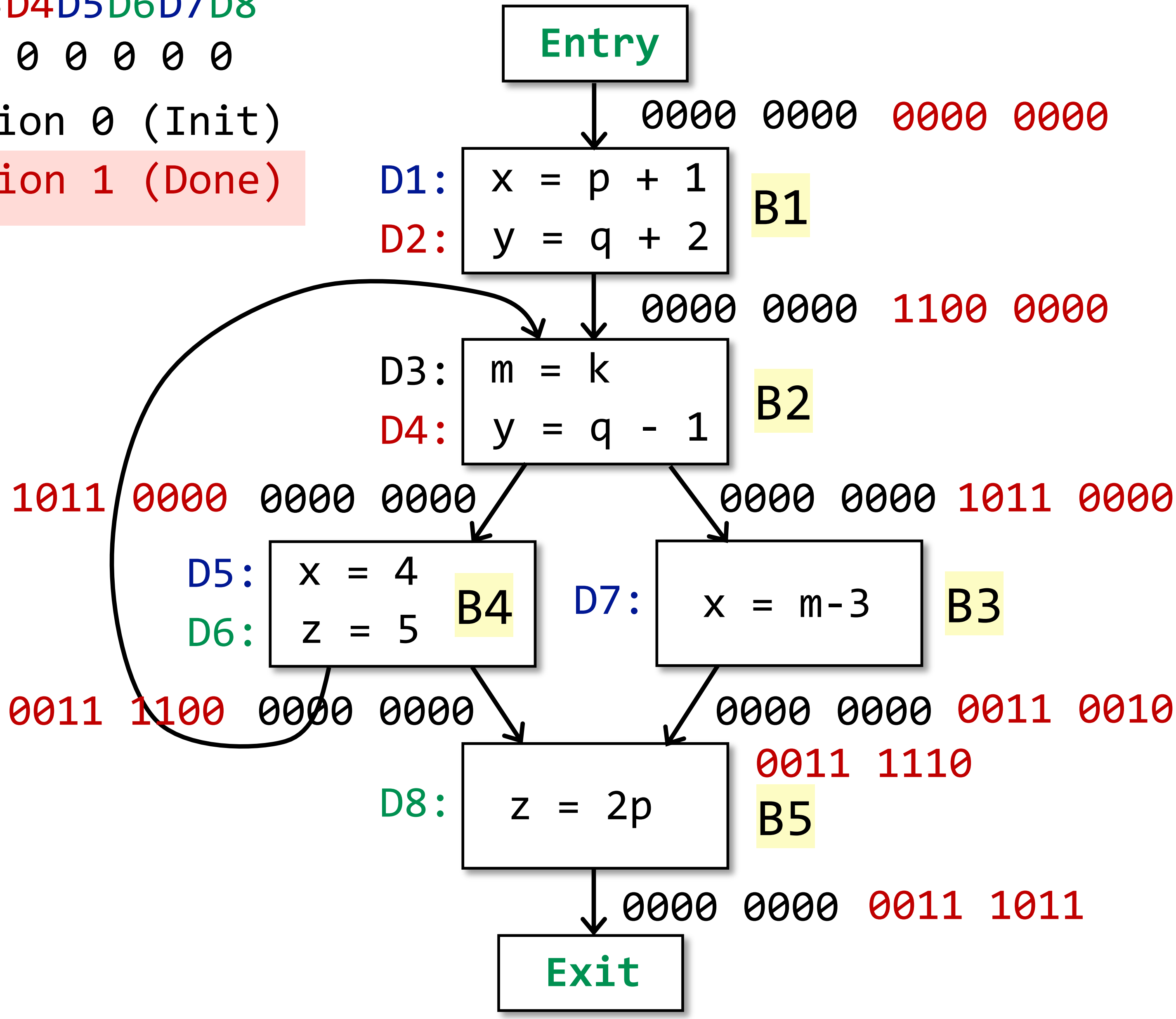
D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1
Entry
0000 0000 0000 0000
D1: x = p + 1
D2: y = q + 2
B1
0000 0000 1100 0000
D3: m = k
D4: y = q - 1
B2
1011 0000 0000 0000
0000 0000 1011 0000
D5: x = 4
D6: z = 5
B4
D7: x = m-3
B3
0011 1100 0000 0000
0000 0000 0011 0010
0011 1110
D8: z = 2p
B5
0000 0000 0011 1011
Exit

D1D2D3D4D5D6D7D8

0 0 0 0 0 0 0 0

Iteration 0 (Init)

Iteration 1 (Done)



Algorithm of Reaching Definitions Analysis

INPUT: CFG ($kill_B$ and gen_B computed for each basic block B)

OUTPUT: IN[B] and OUT[B] for each basic block B

METHOD:

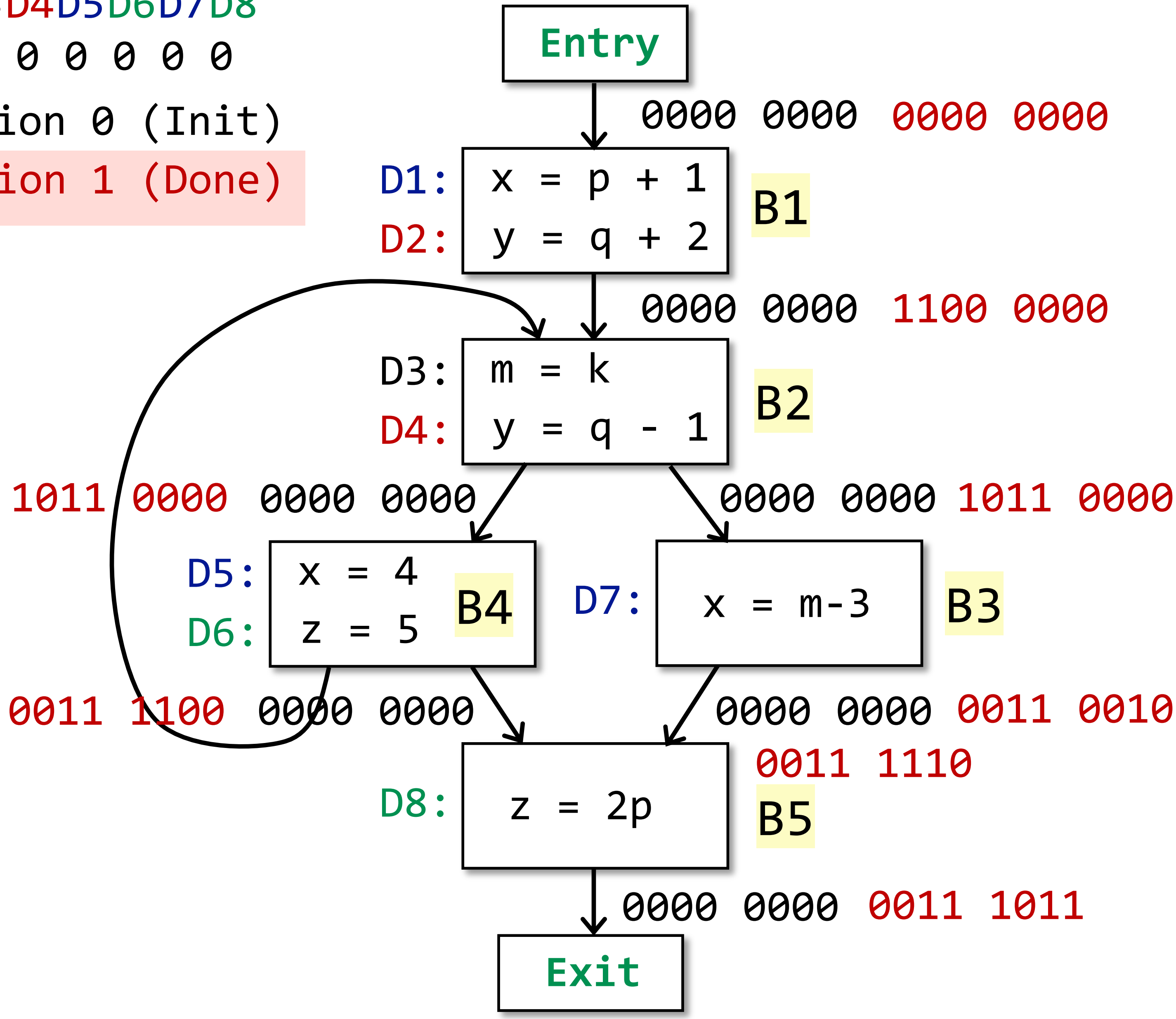
OUT[*entry*] = ∅;
for (each basic block $B \backslash entry$)
    OUT[B] = ∅;
while (changes to any OUT occur)
    for (each basic block $B \backslash entry$) {
        $\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P]$;
        $\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B)$;
    }

D1D2D3D4D5D6D7D8

0 0 0 0 0 0 0 0

Iteration 0 (Init)

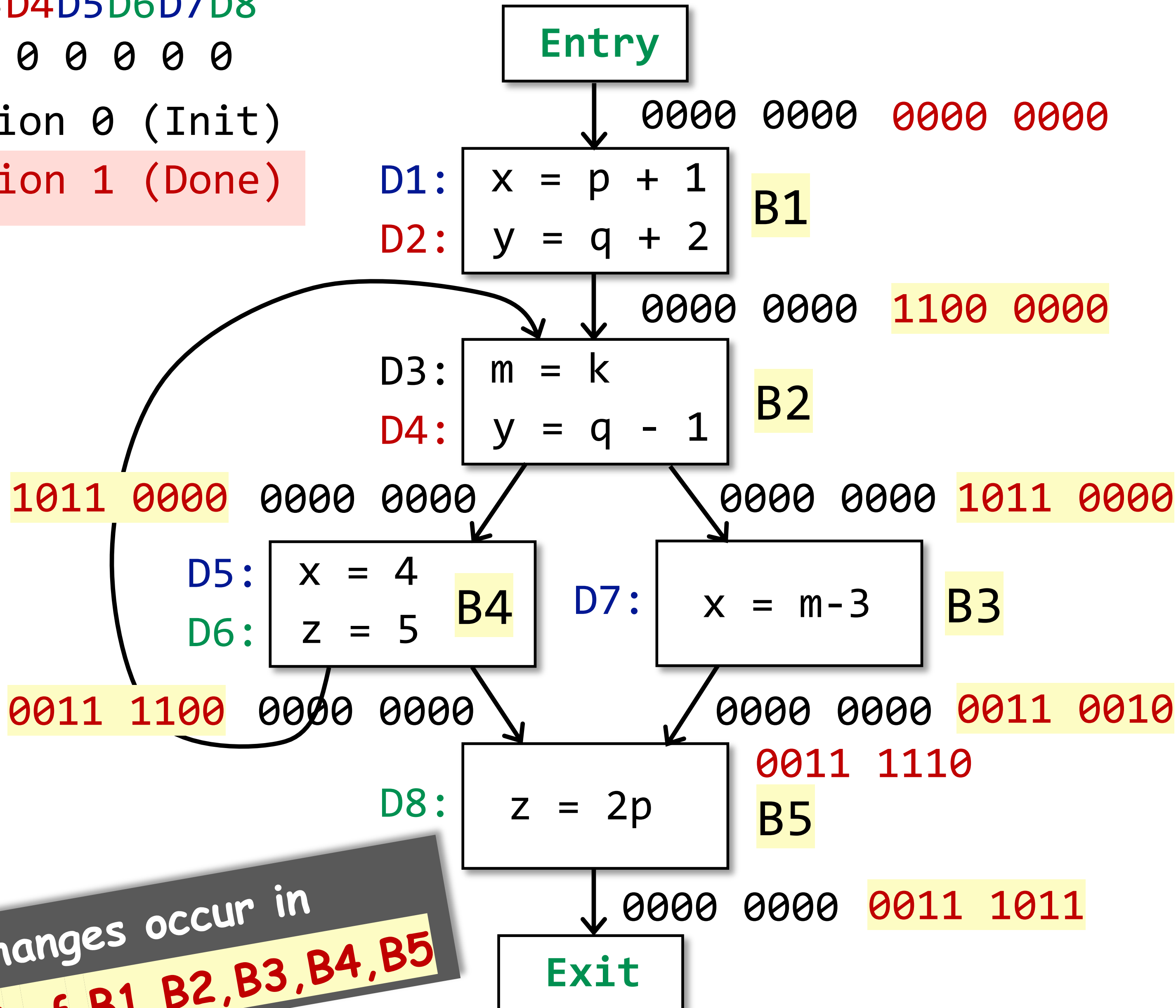
Iteration 1 (Done)



D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0

Iteration 0 (Init)

Iteration 1 (Done)



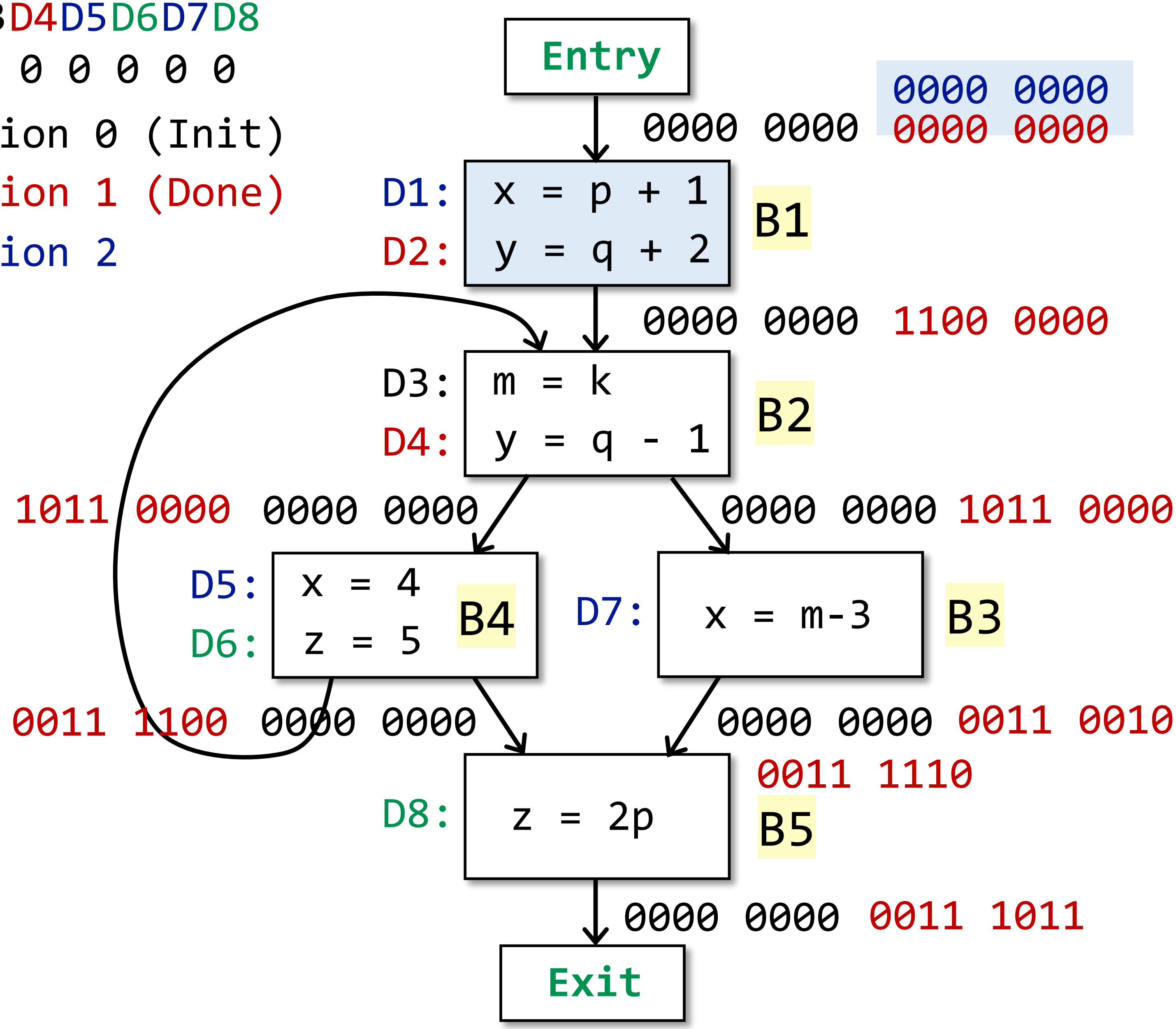
Changes occur in OUT[] of B1,B2,B3,B4,B5

D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0

Iteration 0 (Init)

Iteration 1 (Done)

Iteration 2

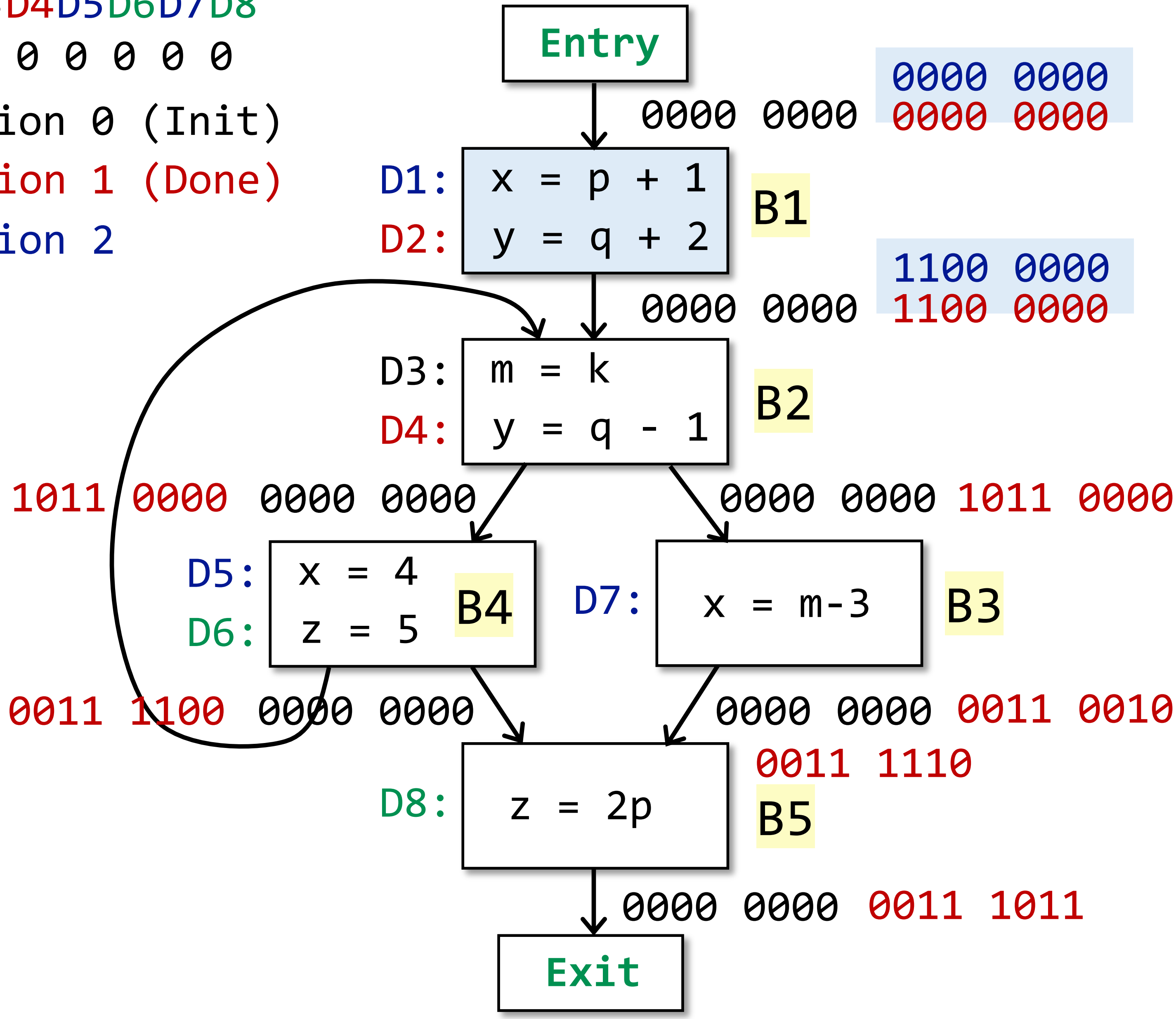


D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0

Iteration 0 (Init)

Iteration 1 (Done)

Iteration 2



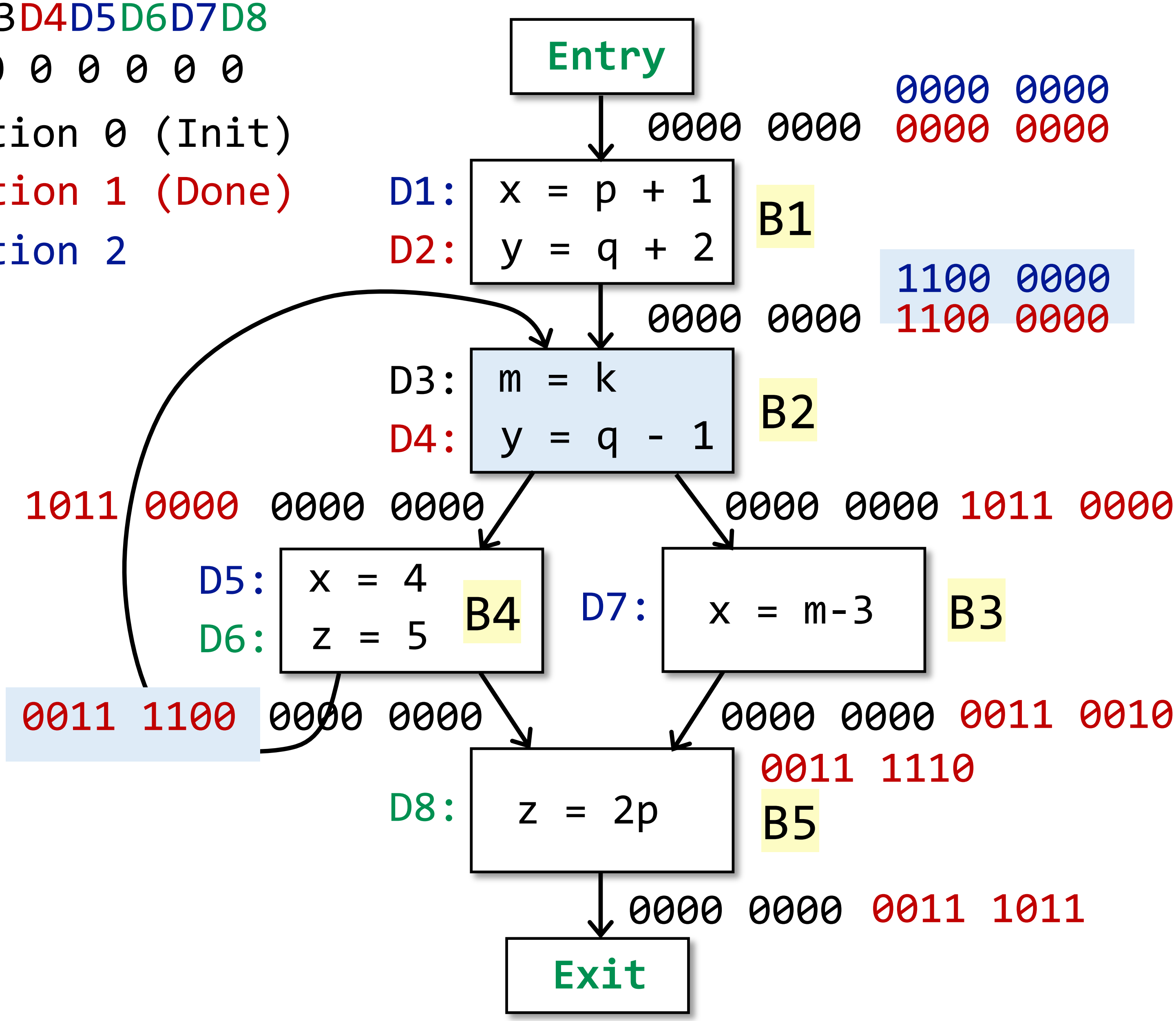
D1D2D3D4D5D6D7D8

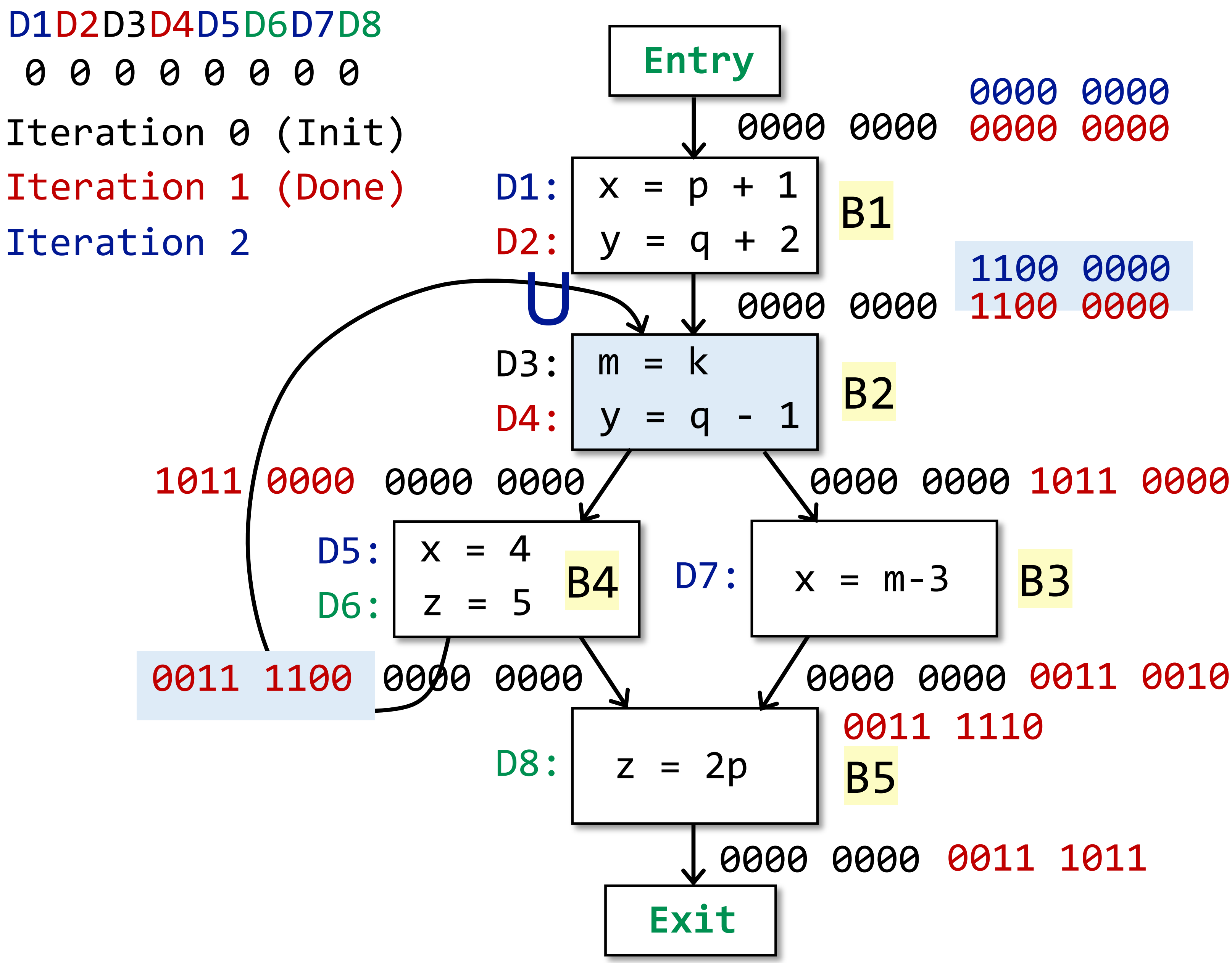
0 0 0 0 0 0 0 0

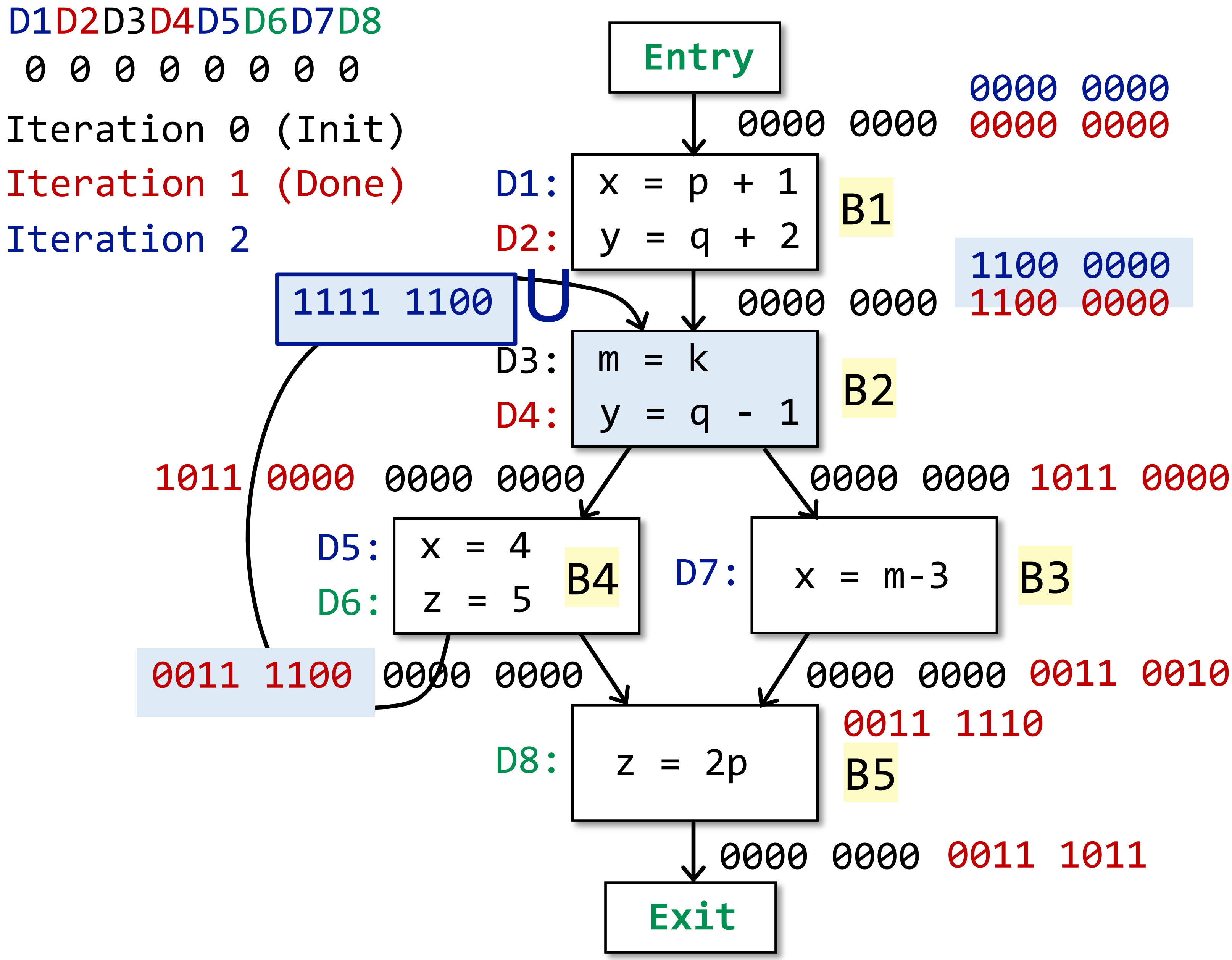
Iteration 0 (Init)

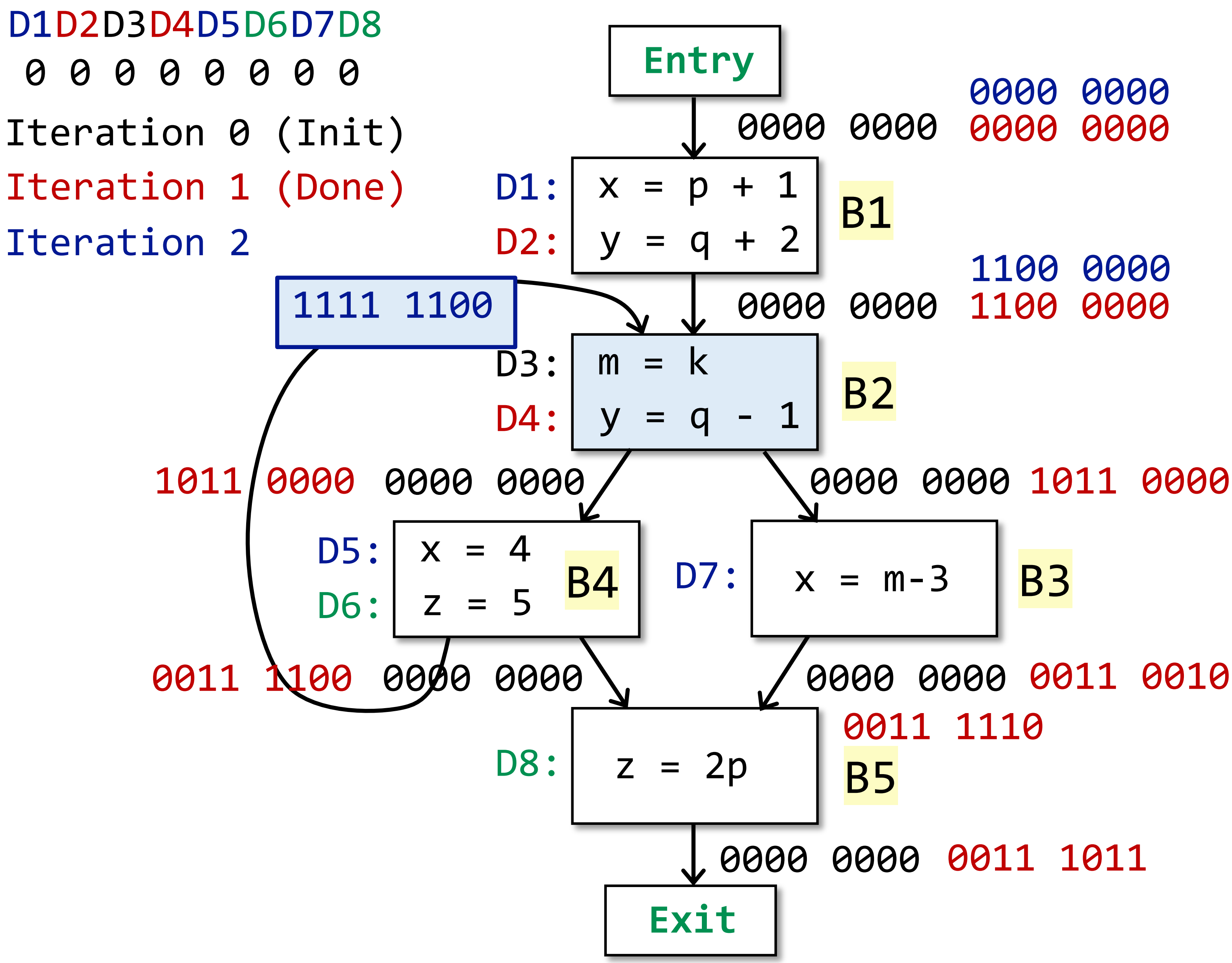
Iteration 1 (Done)

Iteration 2




D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2
Entry
0000 0000
0000 0000
0000 0000
D1: x = p + 1
D2: y = q + 2
B1
1100 0000
1100 0000
0000 0000
U
D3: m = k
D4: y = q - 1
B2
1011 0000
0000 0000
0000 0000
1011 0000
D5: x = 4
D6: z = 5
B4
D7: x = m-3
B3
0011 1100
0000 0000
0000 0000
0011 0010
0011 1110
D8: z = 2p
B5
0000 0000
0011 1011
Exit


D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2
Entry
0000 0000
0000 0000
0000 0000
D1: x = p + 1
D2: y = q + 2
B1
1100 0000
1100 0000
0000 0000
1111 1100
U
D3: m = k
D4: y = q - 1
B2
1011 0000
0000 0000
0000 0000
1011 0000
D5: x = 4
D6: z = 5
B4
D7: x = m-3
B3
0011 1100
0000 0000
0000 0000
0011 0010
0011 1110
D8: z = 2p
B5
0000 0000
0011 1011
Exit



D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2
Entry
0000 0000
0000 0000
0000 0000
D1: x = p + 1
D2: y = q + 2
B1
1100 0000
0000 0000
1100 0000
1111 1100
D3: m = k
D4: y = q - 1
B2
1011 0000
0000 0000
0000 0000
1011 0000
D5: x = 4
D6: z = 5
B4
D7: x = m-3
B3
0011 1100
0000 0000
0000 0000
0011 0010
0011 1110
D8: z = 2p
B5
0000 0000
0011 1011
Exit

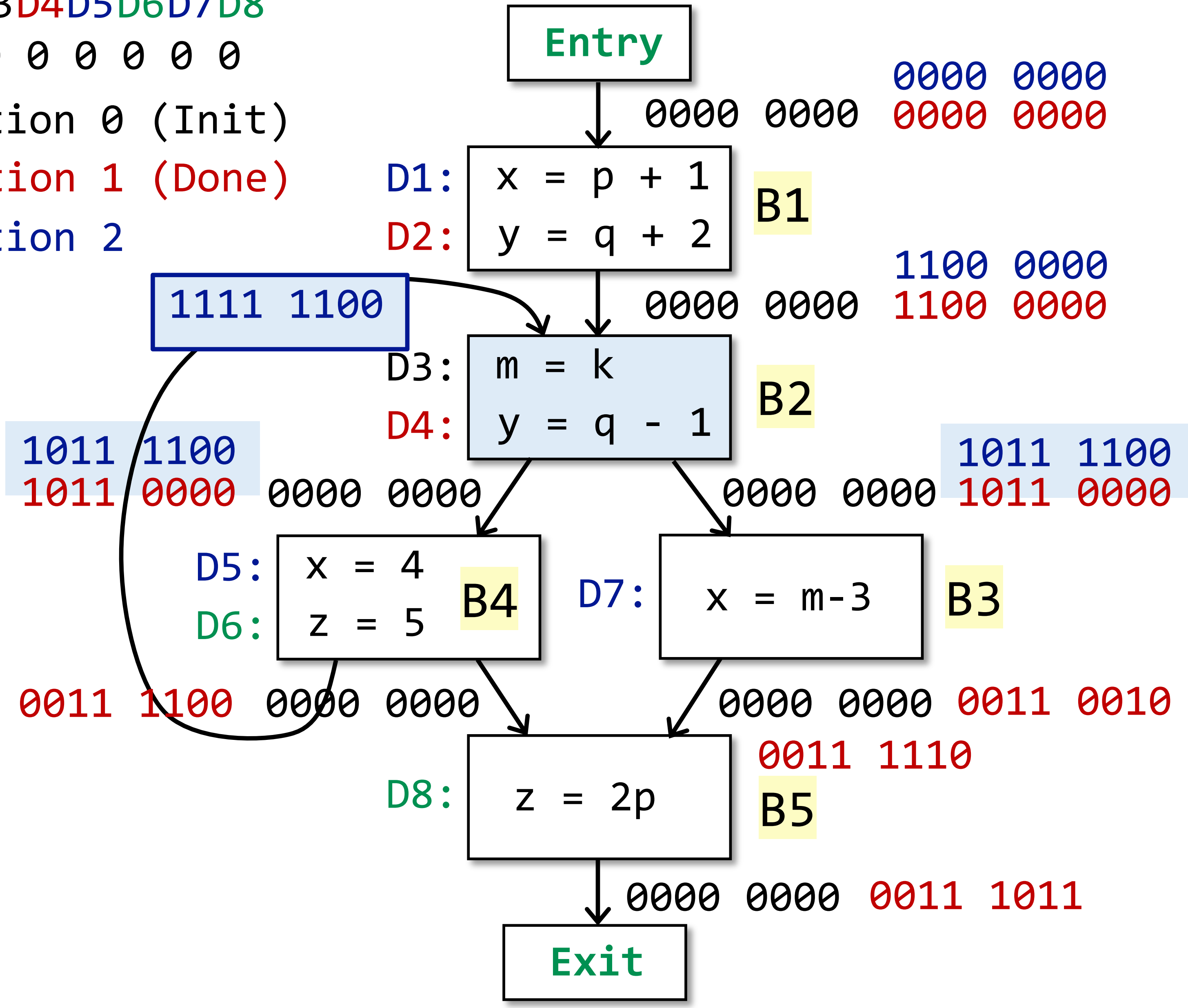
D1D2D3D4D5D6D7D8

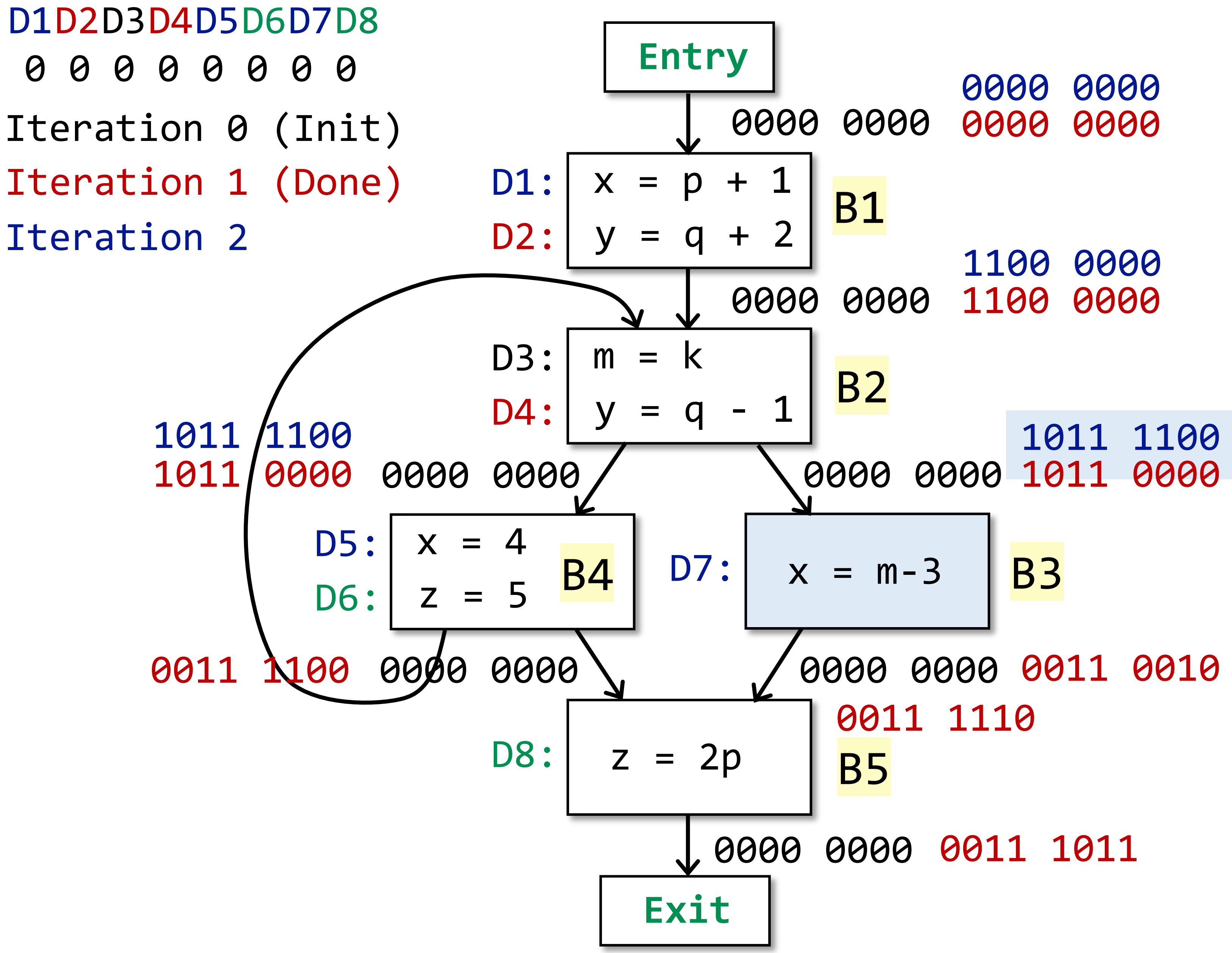
0 0 0 0 0 0 0 0

Iteration 0 (Init)

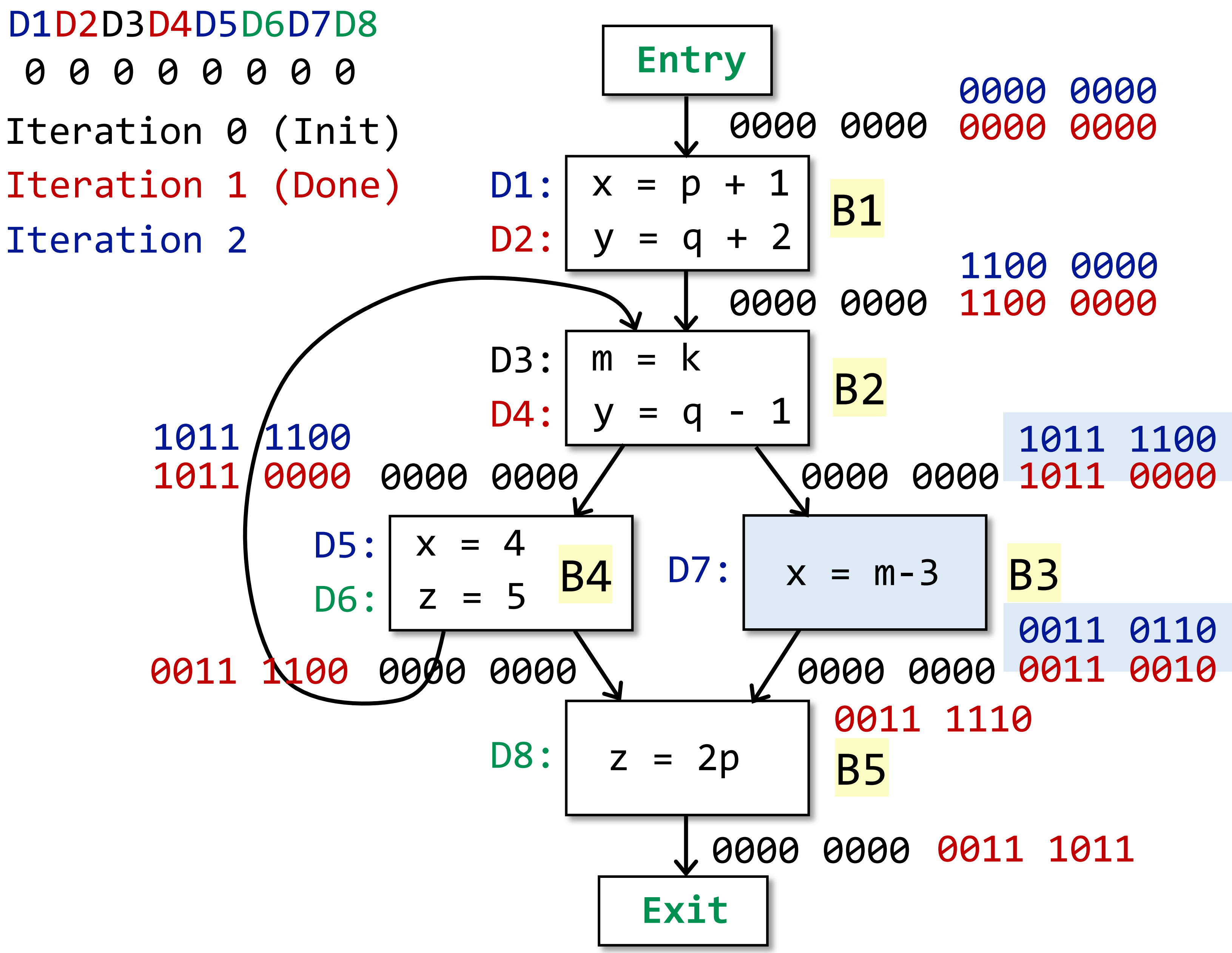
Iteration 1 (Done)

Iteration 2





D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2
Entry
0000 0000
0000 0000
0000 0000
D1: x = p + 1
D2: y = q + 2
B1
1100 0000
0000 0000
1100 0000
D3: m = k
D4: y = q - 1
B2
1011 1100
1011 0000
0000 0000
0000 0000
1011 1100
1011 0000
D5: x = 4
D6: z = 5
B4
D7: x = m-3
B3
0011 1100
0000 0000
0000 0000
0011 0010
0011 1110
D8: z = 2p
B5
0000 0000
0011 1011
Exit

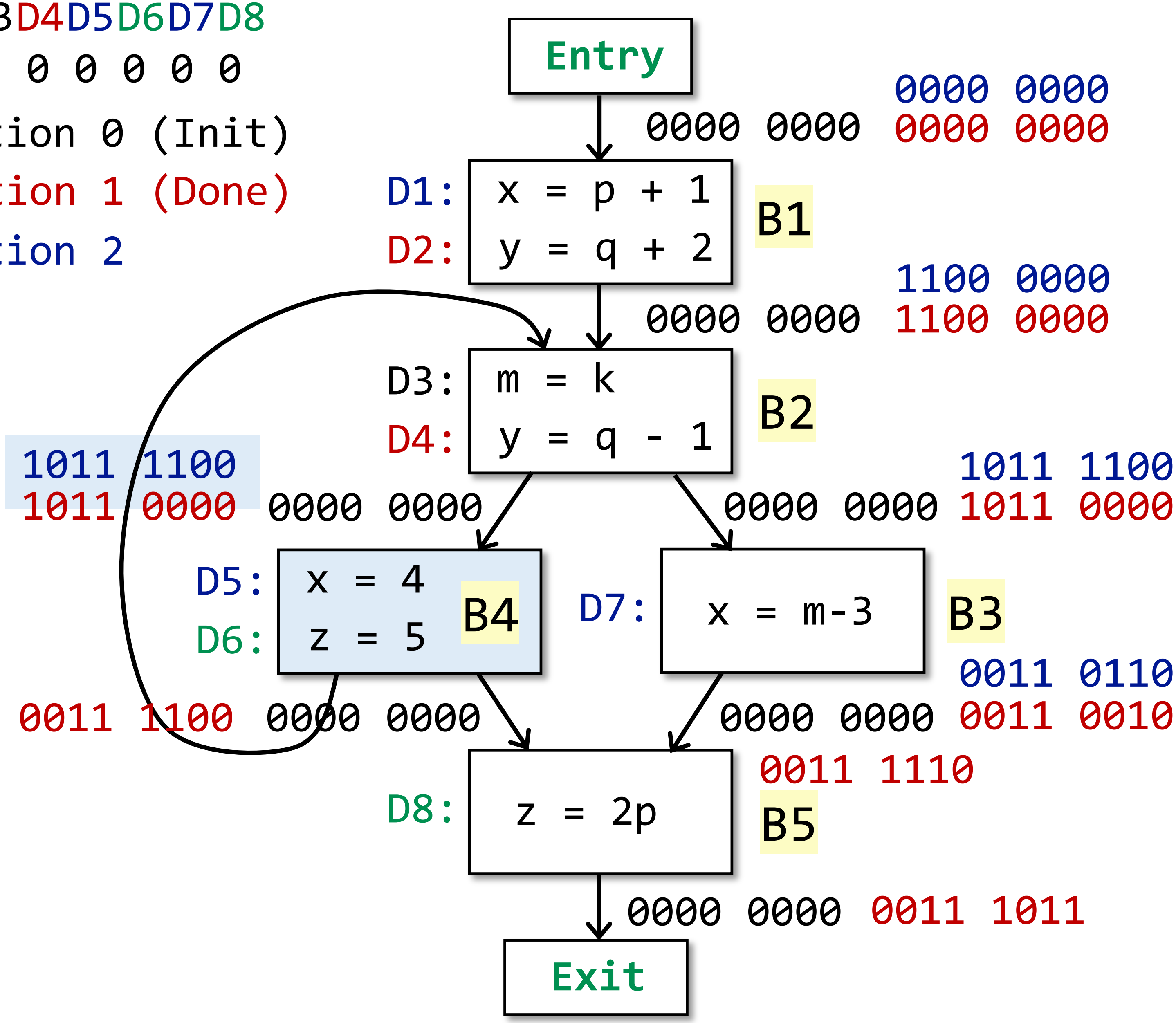

D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2
Entry
0000 0000
0000 0000
0000 0000
D1:
x = p + 1
D2:
y = q + 2
B1
1100 0000
0000 0000
1100 0000
D3:
m = k
D4:
y = q - 1
B2
1011 1100
1011 0000
0000 0000
0000 0000
1011 1100
1011 0000
D5:
x = 4
D6:
z = 5
B4
D7:
x = m-3
B3
0011 0110
0011 0010
0011 1100
0000 0000
0000 0000
0011 1110
D8:
z = 2p
B5
0000 0000
0011 1011
Exit

D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0

Iteration 0 (Init)

Iteration 1 (Done)

Iteration 2

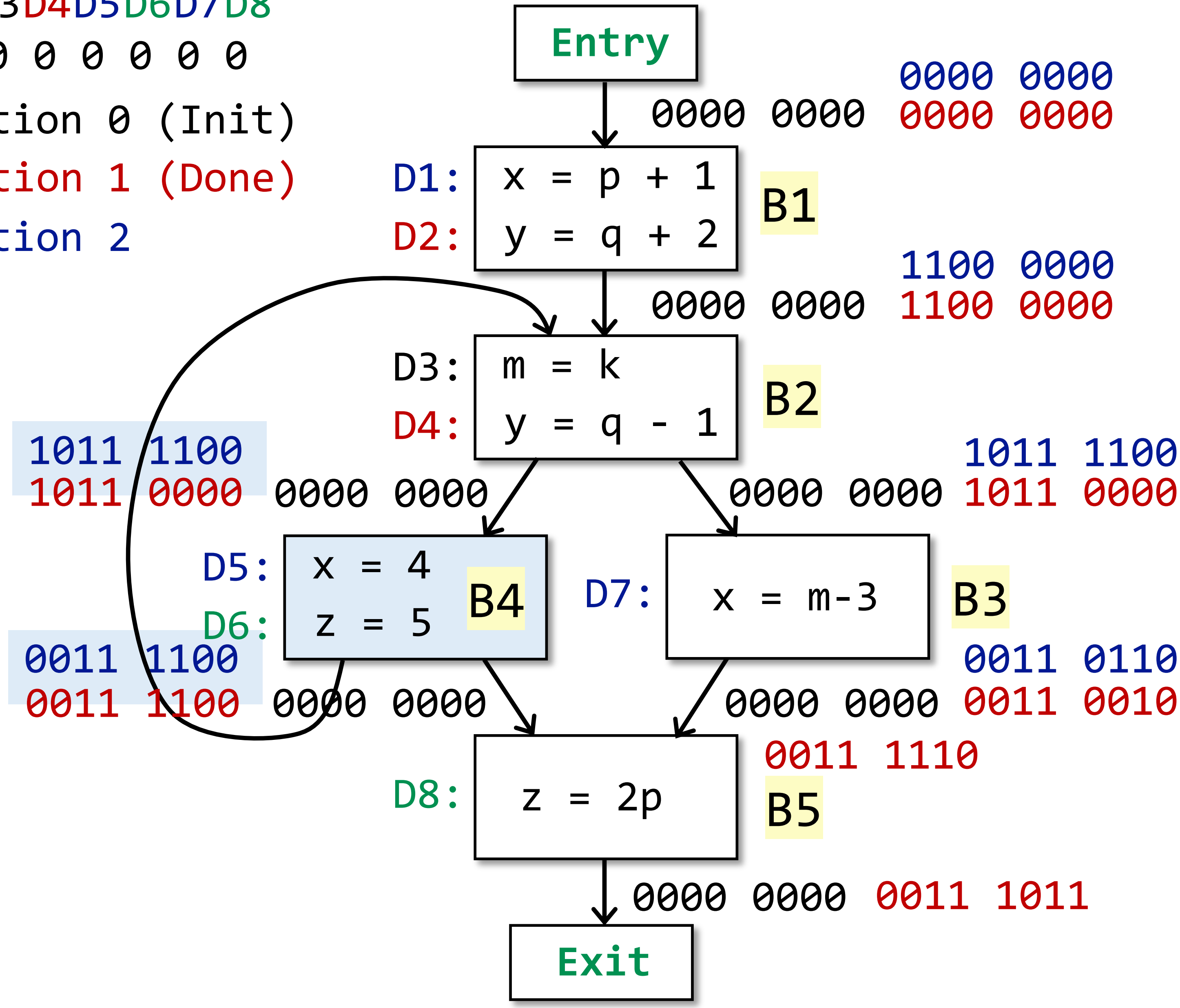


D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0

Iteration 0 (Init)

Iteration 1 (Done)

Iteration 2

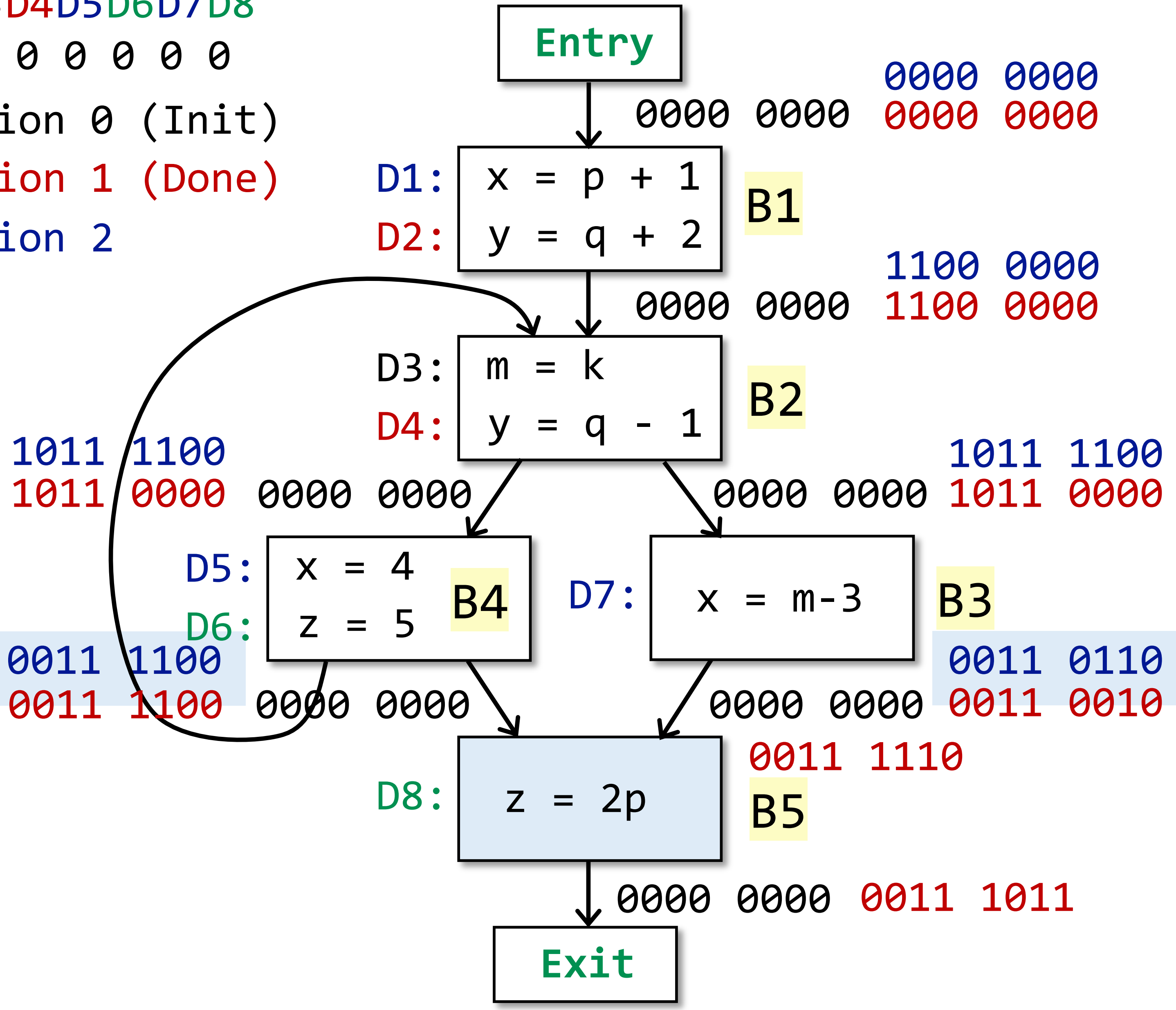


D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0

Iteration 0 (Init)

Iteration 1 (Done)

Iteration 2



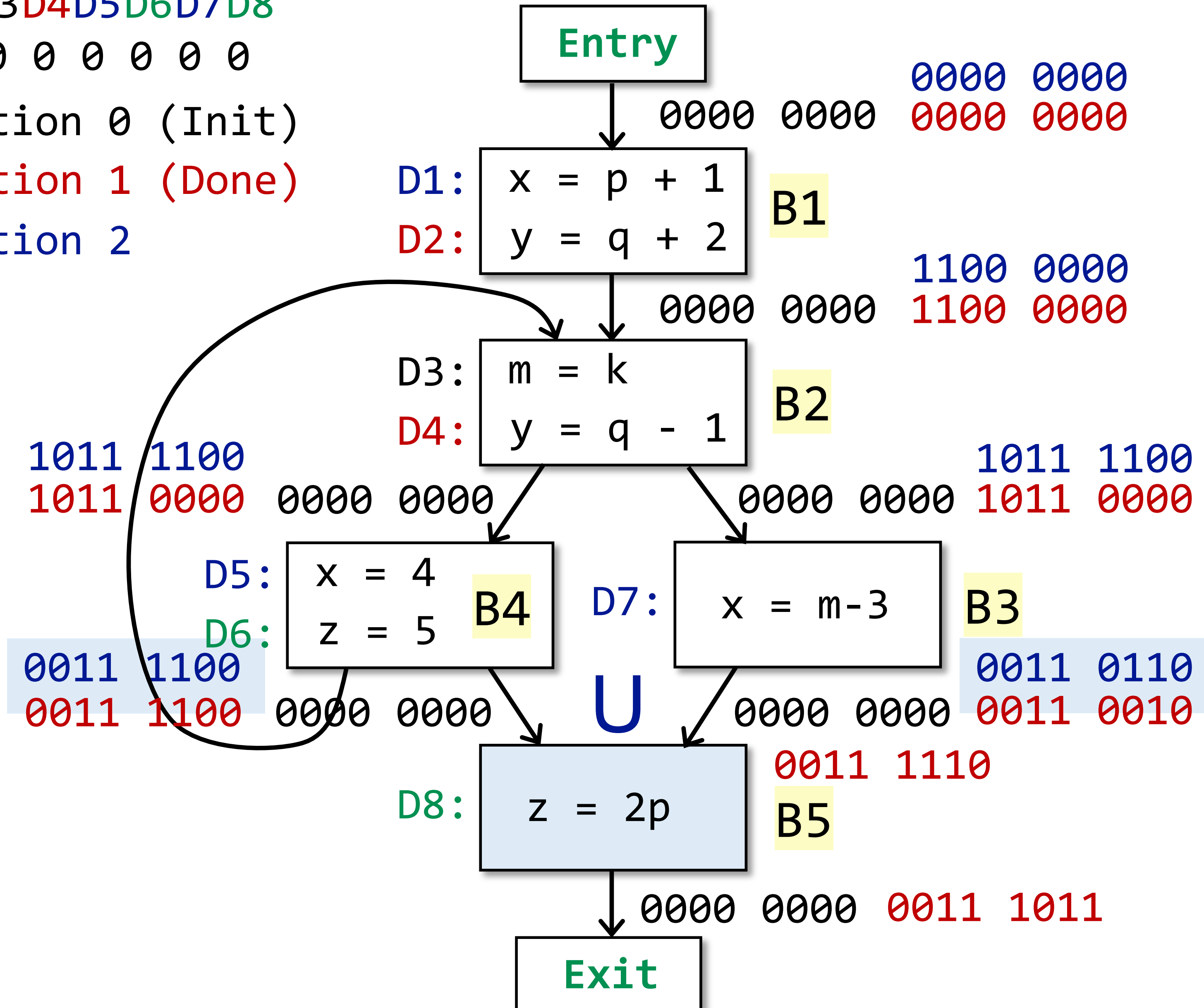
D1D2D3D4D5D6D7D8

0 0 0 0 0 0 0 0

Iteration 0 (Init)

Iteration 1 (Done)

Iteration 2

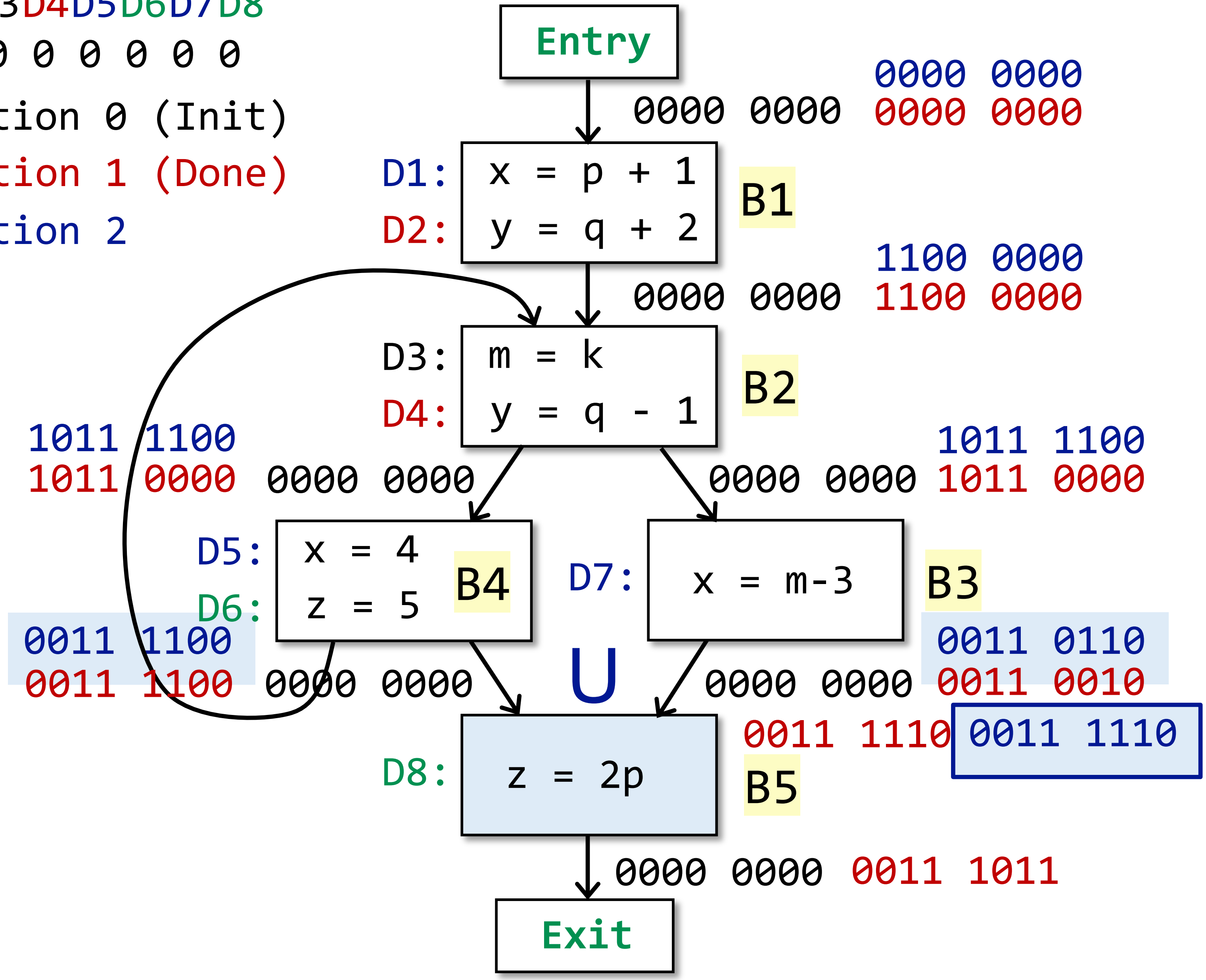


D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0

Iteration 0 (Init)

Iteration 1 (Done)

Iteration 2



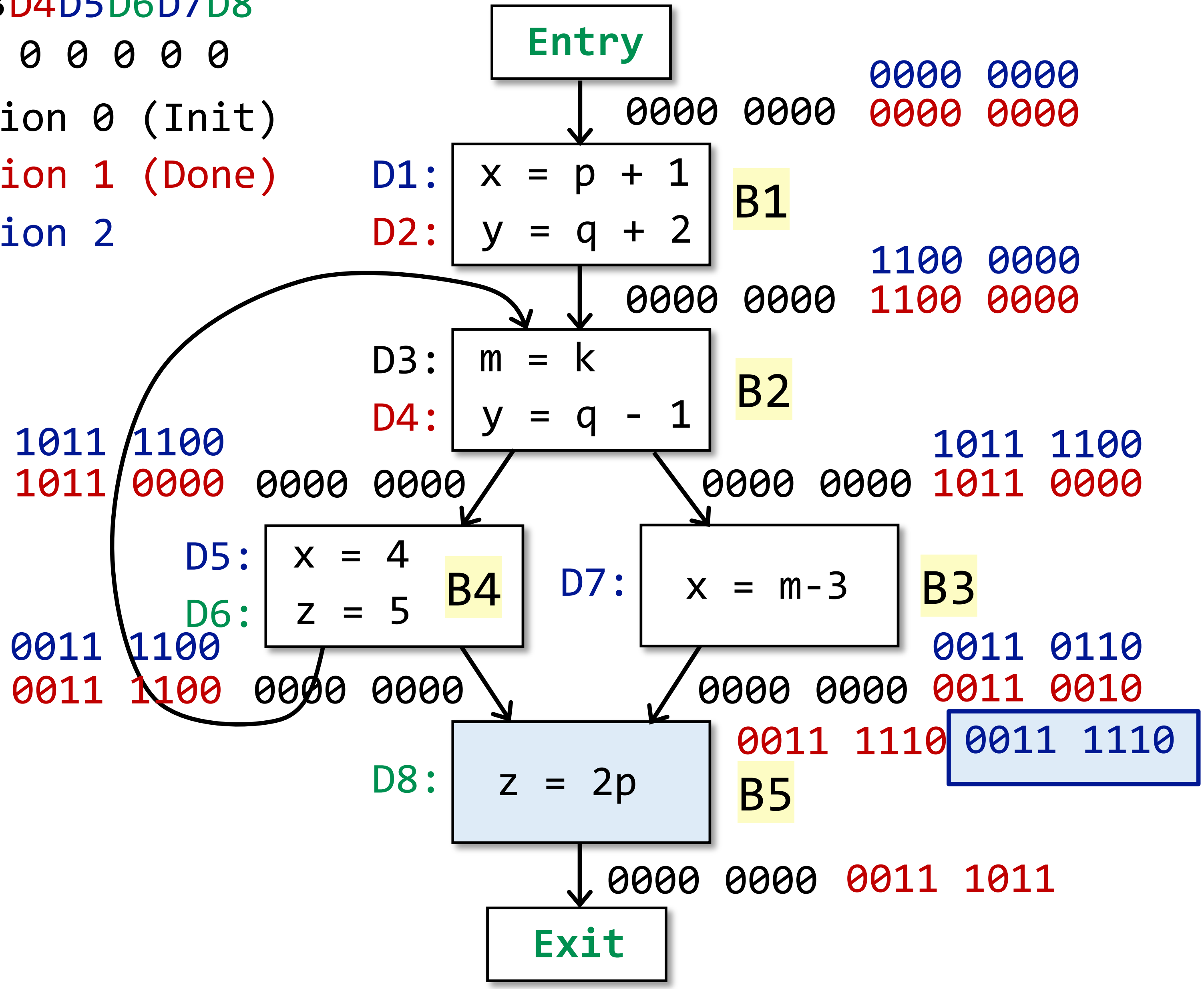
D1D2D3D4D5D6D7D8

0 0 0 0 0 0 0 0

Iteration 0 (Init)

Iteration 1 (Done)

Iteration 2

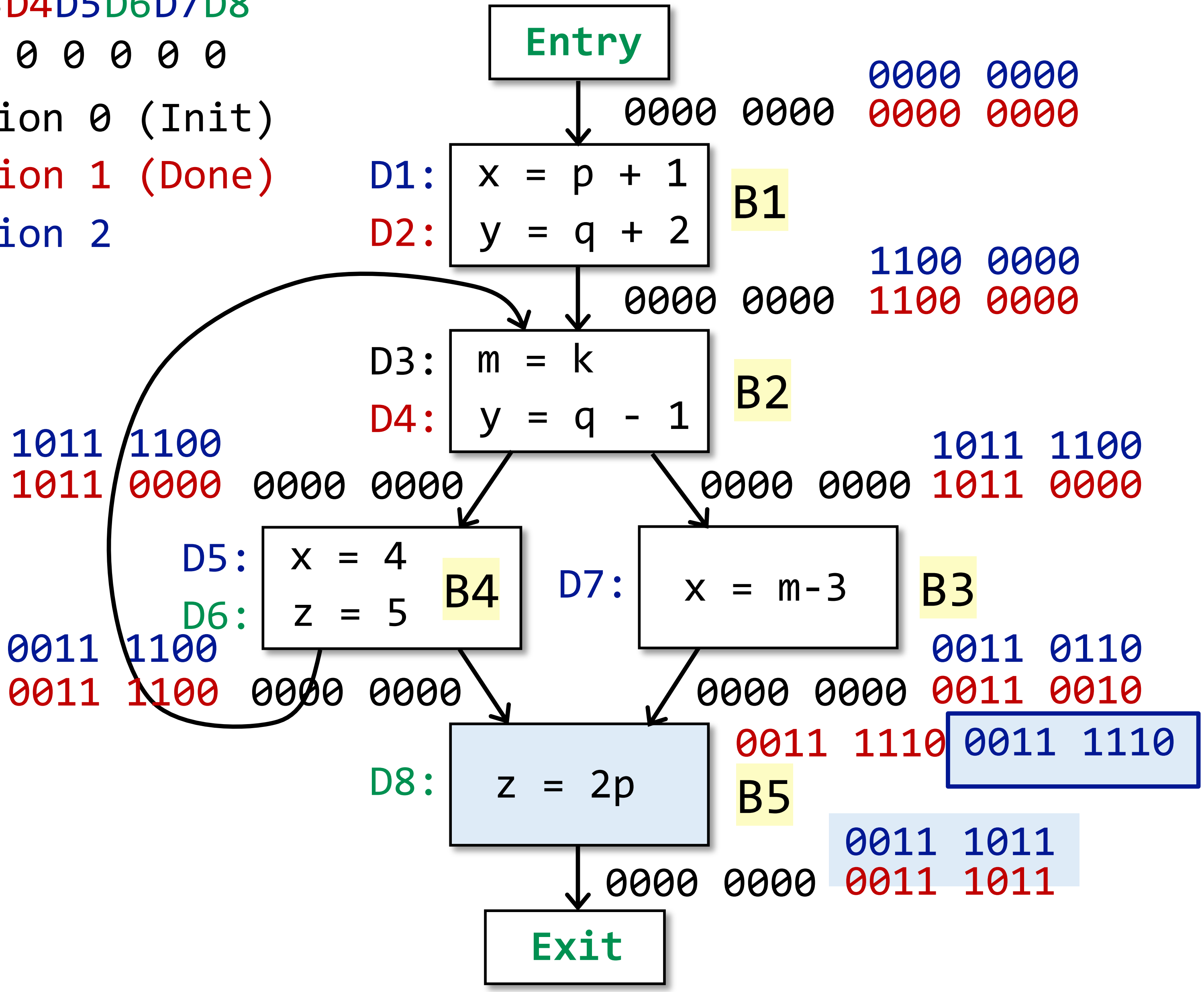


D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0

Iteration 0 (Init)

Iteration 1 (Done)

Iteration 2



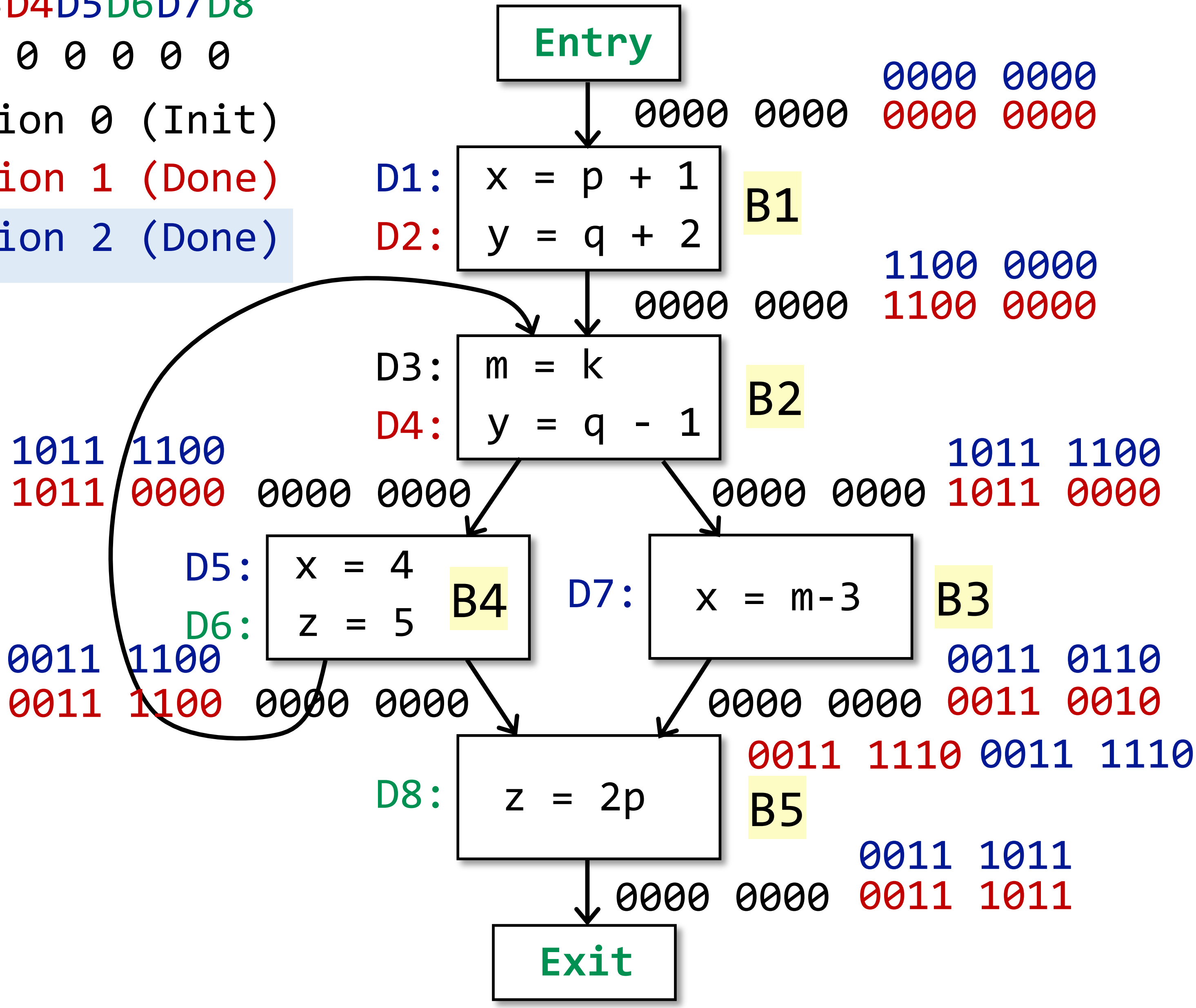
D1D2D3D4D5D6D7D8

0 0 0 0 0 0 0 0

Iteration 0 (Init)

Iteration 1 (Done)

Iteration 2 (Done)

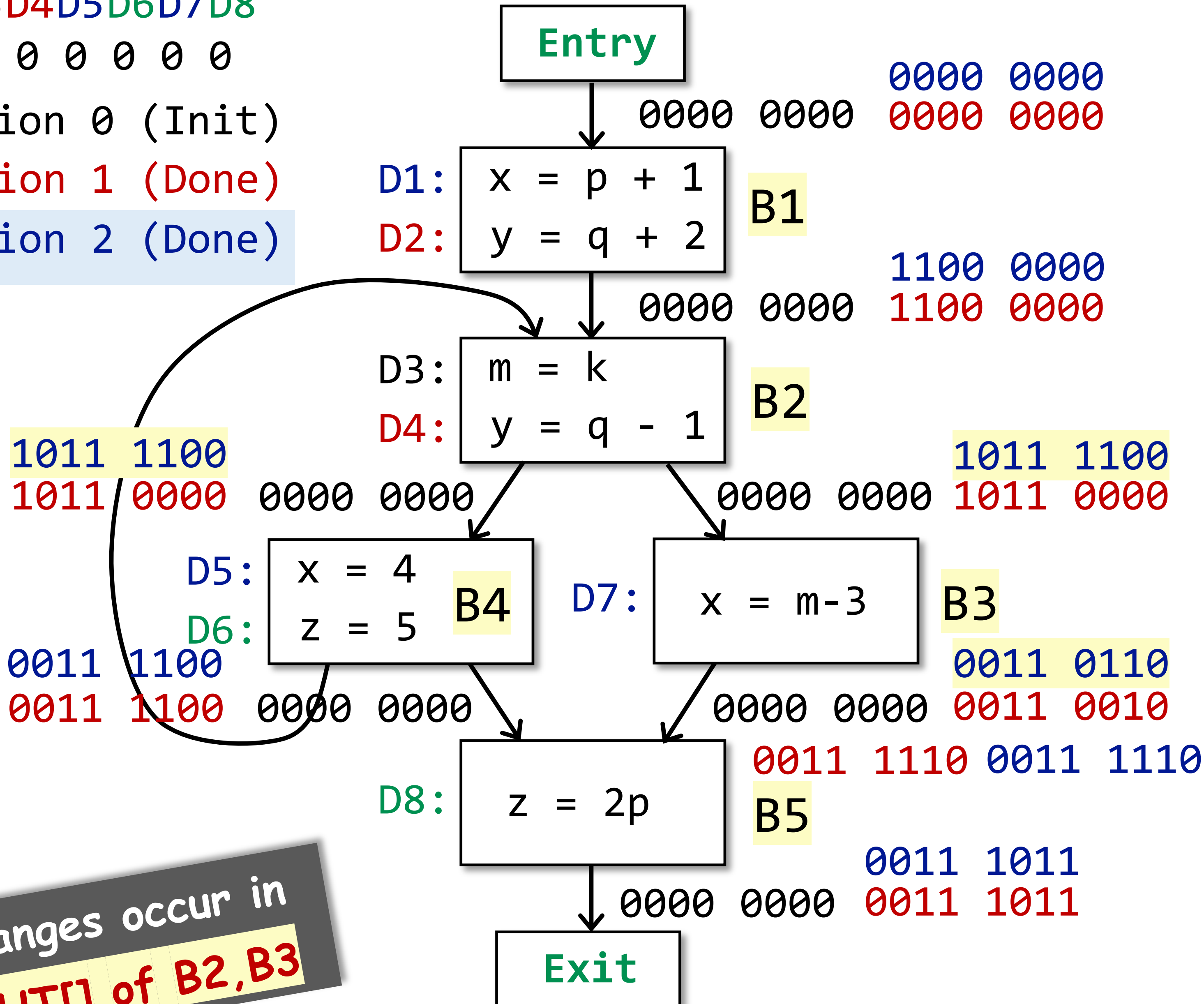


D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0

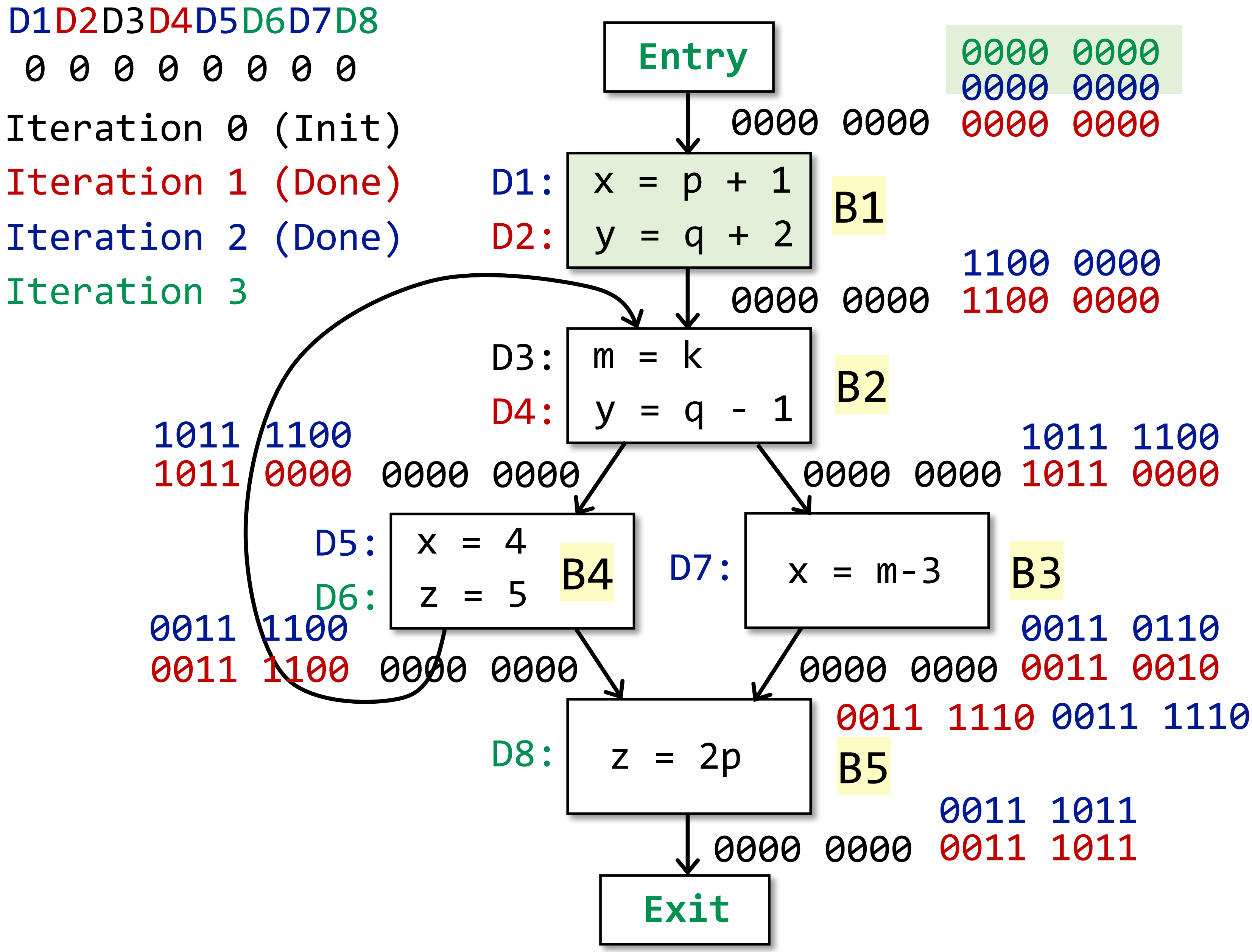
Iteration 0 (Init)

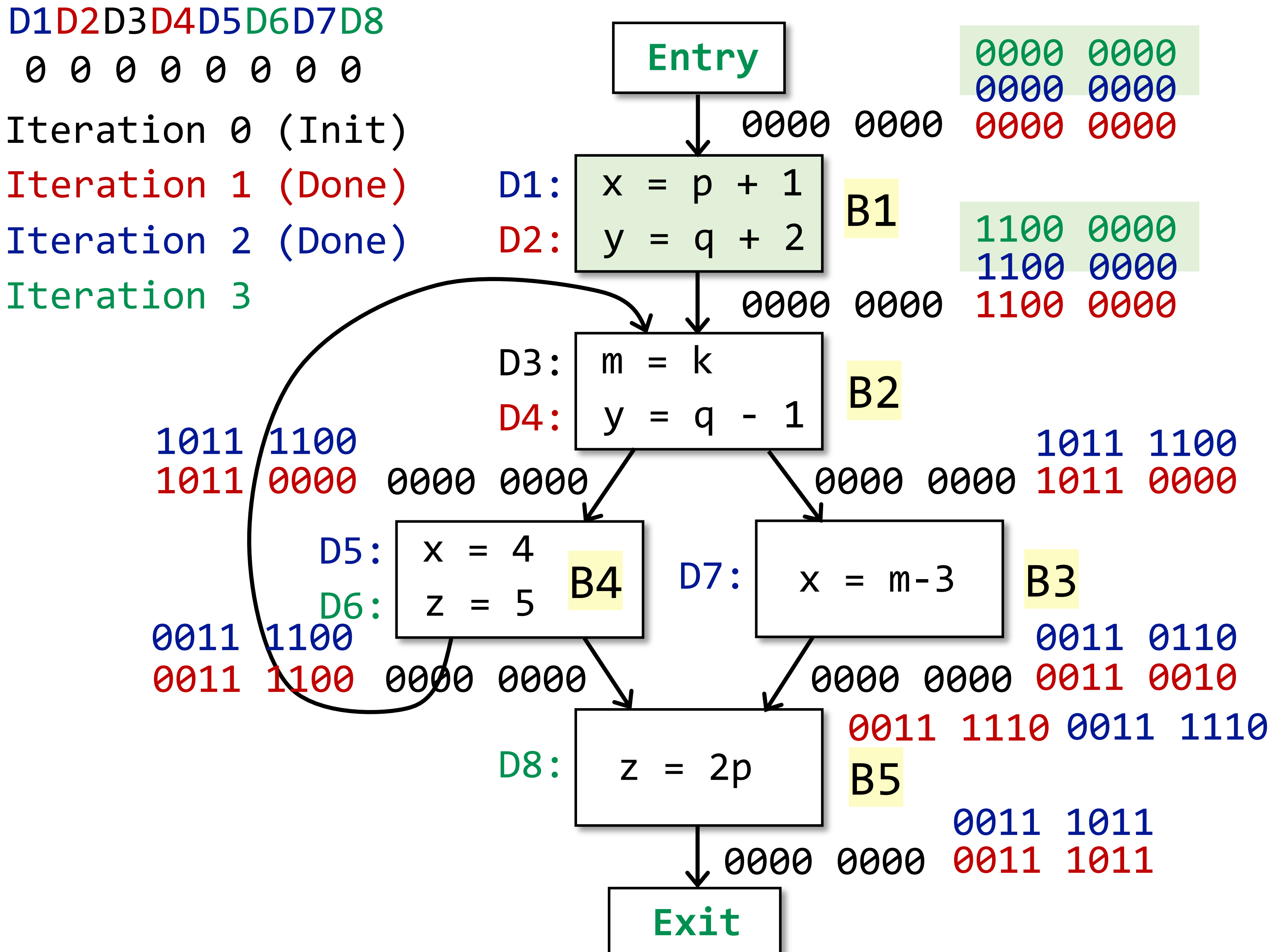
Iteration 1 (Done)

Iteration 2 (Done)



Changes occur in OUT[] of B2,B3


D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2 (Done)
Iteration 3
Entry
0000 0000
0000 0000
0000 0000
0000 0000
D1: x = p + 1
D2: y = q + 2
B1
1100 0000
0000 0000
1100 0000
D3: m = k
D4: y = q - 1
B2
1011 1100
1011 0000
0000 0000
1011 1100
0000 0000
1011 0000
D5: x = 4
D6: z = 5
B4
D7: x = m-3
B3
0011 1100
0011 1100
0000 0000
0011 0110
0000 0000
0011 0010
0011 1110
0011 1110
D8: z = 2p
B5
0011 1011
0000 0000
0011 1011
Exit



D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2 (Done)
Iteration 3
Entry
0000 0000
0000 0000
0000 0000 0000 0000
D1: x = p + 1
D2: y = q + 2
B1
1100 0000
1100 0000
0000 0000 1100 0000
D3: m = k
D4: y = q - 1
B2
1011 1100
1011 0000 0000 0000
1011 1100
0000 0000 1011 0000
D5: x = 4
D6: z = 5
B4
D7: x = m-3
B3
0011 1100
0011 1100 0000 0000
0011 0110
0000 0000 0011 0010
0011 1110 0011 1110
D8: z = 2p
B5
0011 1011
0000 0000 0011 1011
Exit

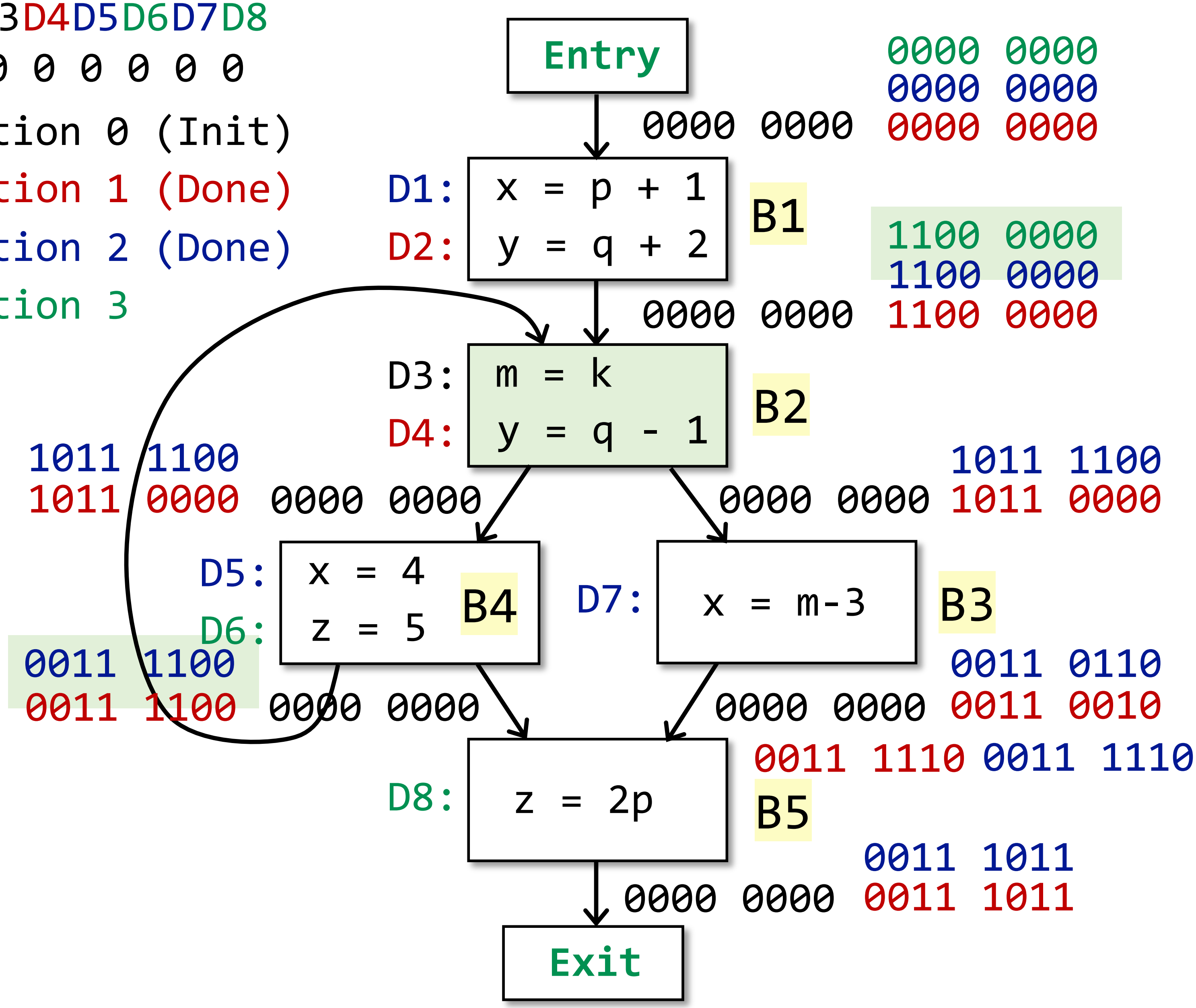
D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0

Iteration 0 (Init)

Iteration 1 (Done)

Iteration 2 (Done)

Iteration 3



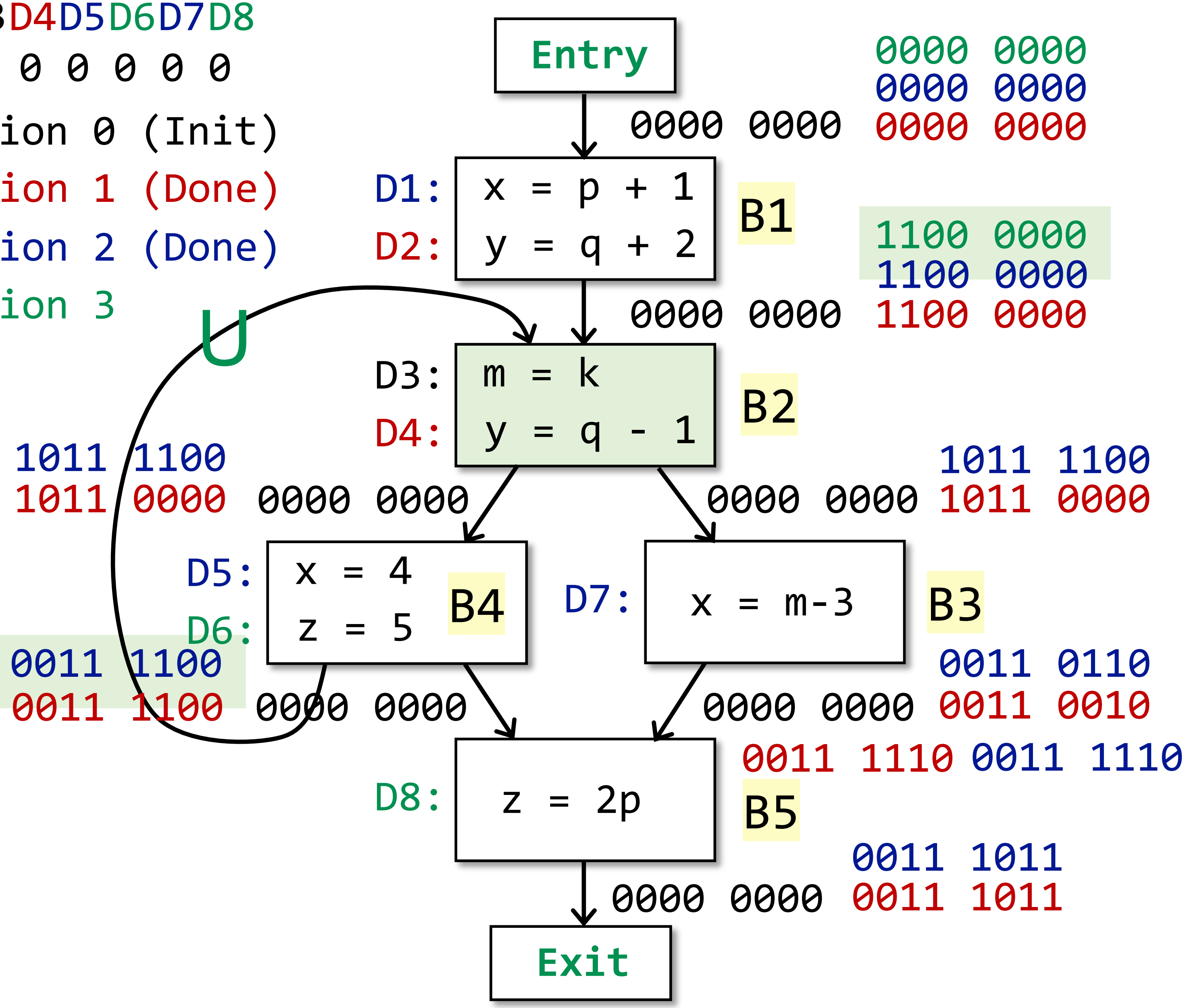
D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0

Iteration 0 (Init)

Iteration 1 (Done)

Iteration 2 (Done)

Iteration 3



D1D2D3D4D5D6D7D8

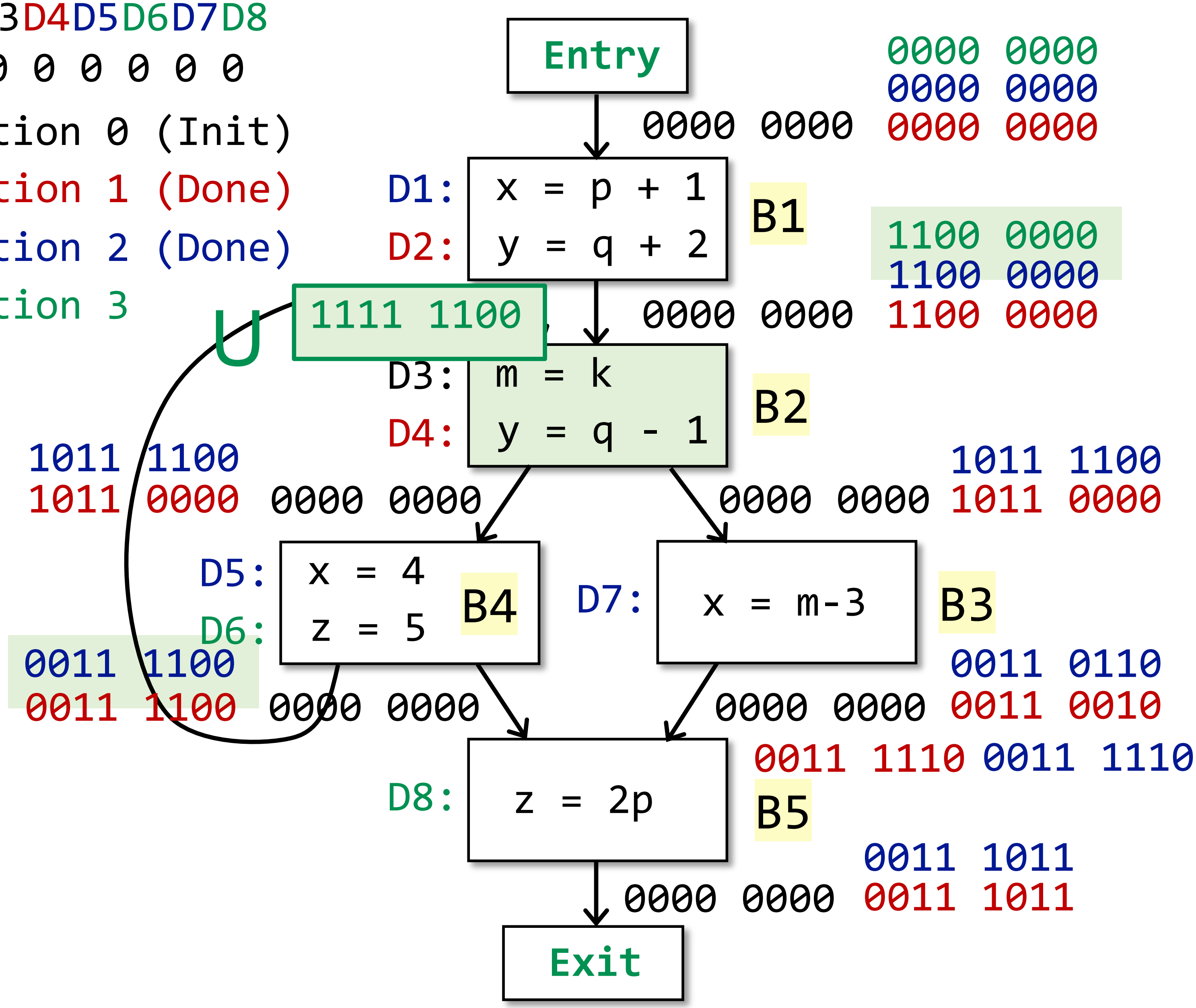
0 0 0 0 0 0 0 0

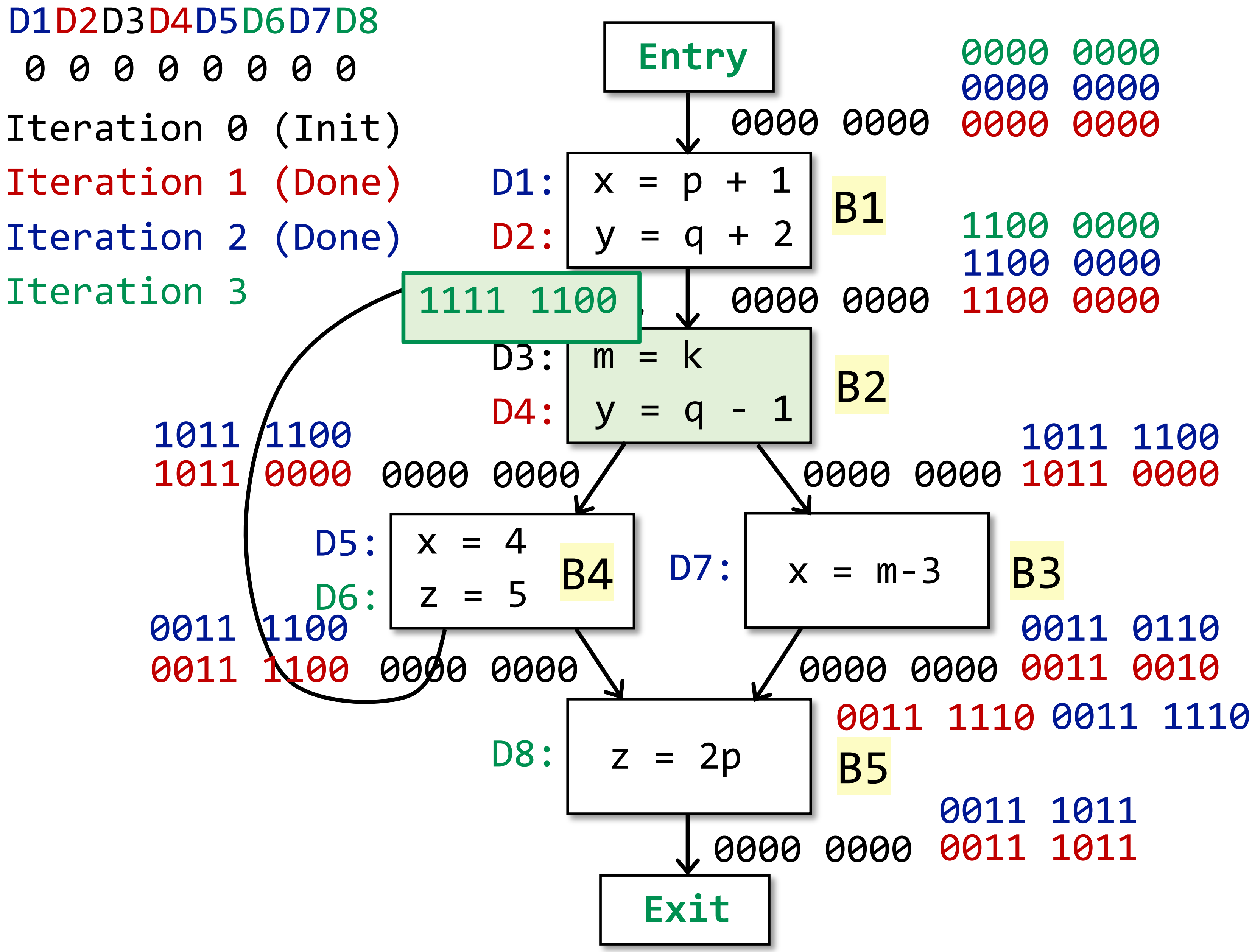
Iteration 0 (Init)

Iteration 1 (Done)

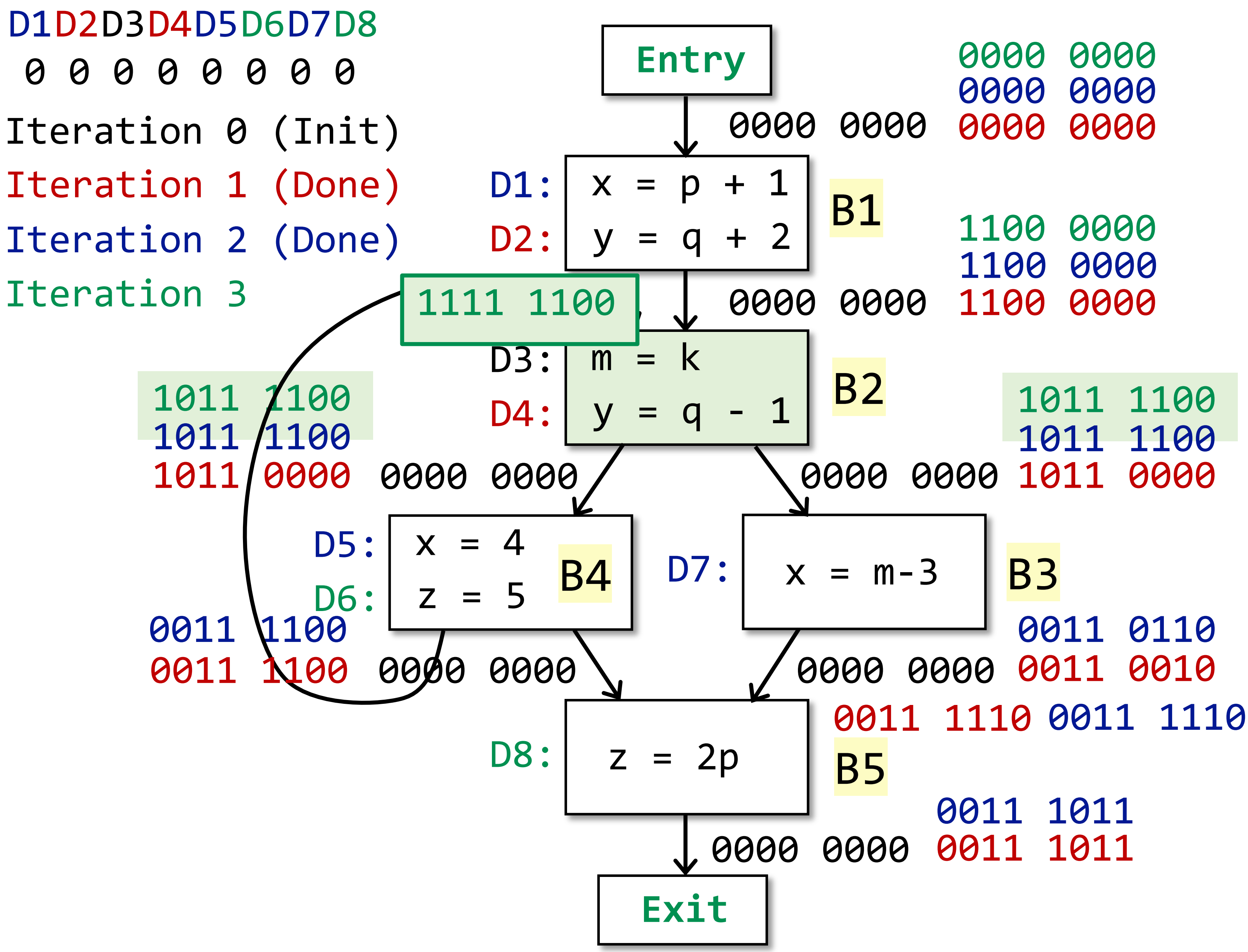
Iteration 2 (Done)

Iteration 3

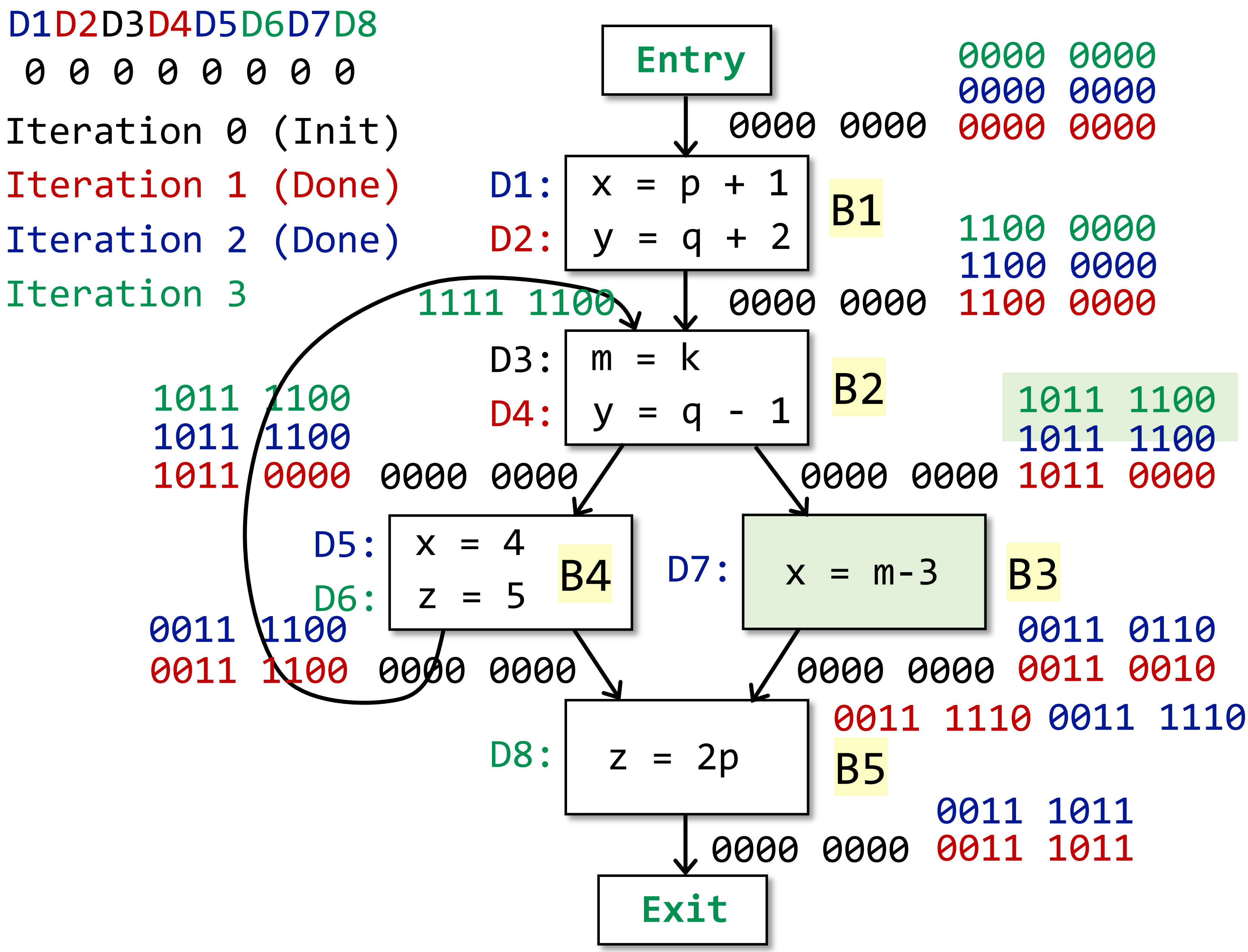




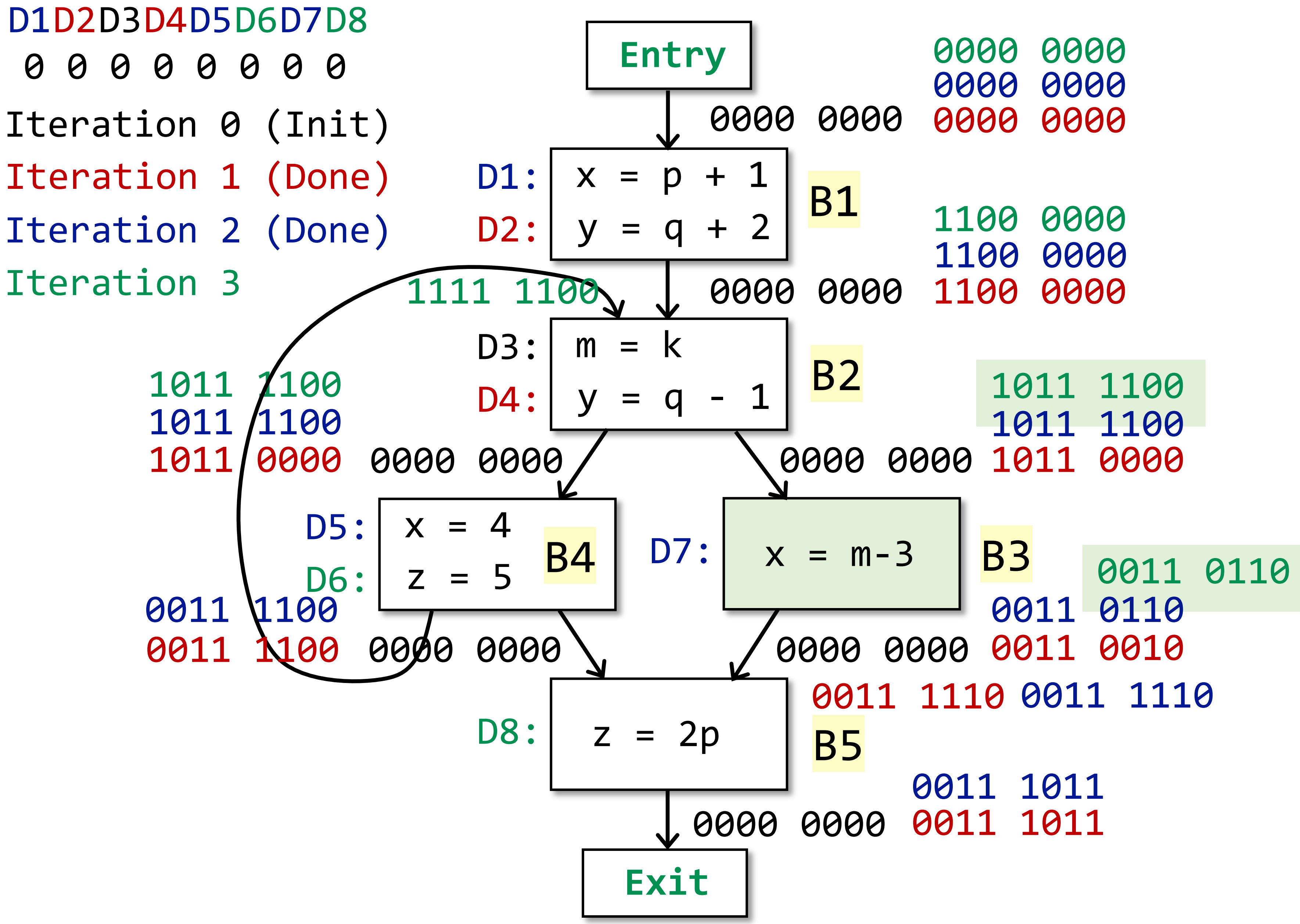
D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2 (Done)
Iteration 3
Entry
0000 0000
0000 0000
0000 0000
0000 0000
D1: x = p + 1
D2: y = q + 2
B1
1100 0000
1100 0000
0000 0000
1100 0000
1111 1100
D3: m = k
D4: y = q - 1
B2
1011 1100
1011 0000
0000 0000
0000 0000
1011 1100
1011 0000
D5: x = 4
D6: z = 5
B4
D7: x = m-3
B3
0011 1100
0011 1100
0000 0000
0000 0000
0011 0110
0011 0010
0011 1110 0011 1110
D8: z = 2p
B5
0011 1011
0000 0000
0011 1011
Exit



D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2 (Done)
Iteration 3
Entry
0000 0000
0000 0000
0000 0000
0000 0000
D1: x = p + 1
D2: y = q + 2
B1
1100 0000
1100 0000
1100 0000
0000 0000
1111 1100
D3: m = k
D4: y = q - 1
B2
1011 1100
1011 1100
1011 0000
0000 0000
1011 1100
1011 1100
0000 0000
1011 0000
D5: x = 4
D6: z = 5
B4
D7: x = m-3
B3
0011 1100
0011 1100
0000 0000
0011 0110
0000 0000
0011 0010
0011 1110
0011 1110
D8: z = 2p
B5
0011 1011
0000 0000
0011 1011
Exit



Yue Li @ Nanjing University



D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2 (Done)
Iteration 3
Entry
0000 0000
0000 0000
0000 0000
0000 0000
D1: x = p + 1
D2: y = q + 2
B1
1100 0000
1100 0000
0000 0000
1100 0000
1111 1100
D3: m = k
D4: y = q - 1
B2
1011 1100
1011 1100
1011 0000
0000 0000
1011 1100
1011 1100
0000 0000
1011 0000
D5: x = 4
D6: z = 5
B4
D7: x = m-3
B3
0011 0110
0011 1100
0011 1100
0000 0000
0011 0110
0000 0000
0011 0010
0011 1110
0011 1110
D8: z = 2p
B5
0011 1011
0000 0000
0011 1011
Exit

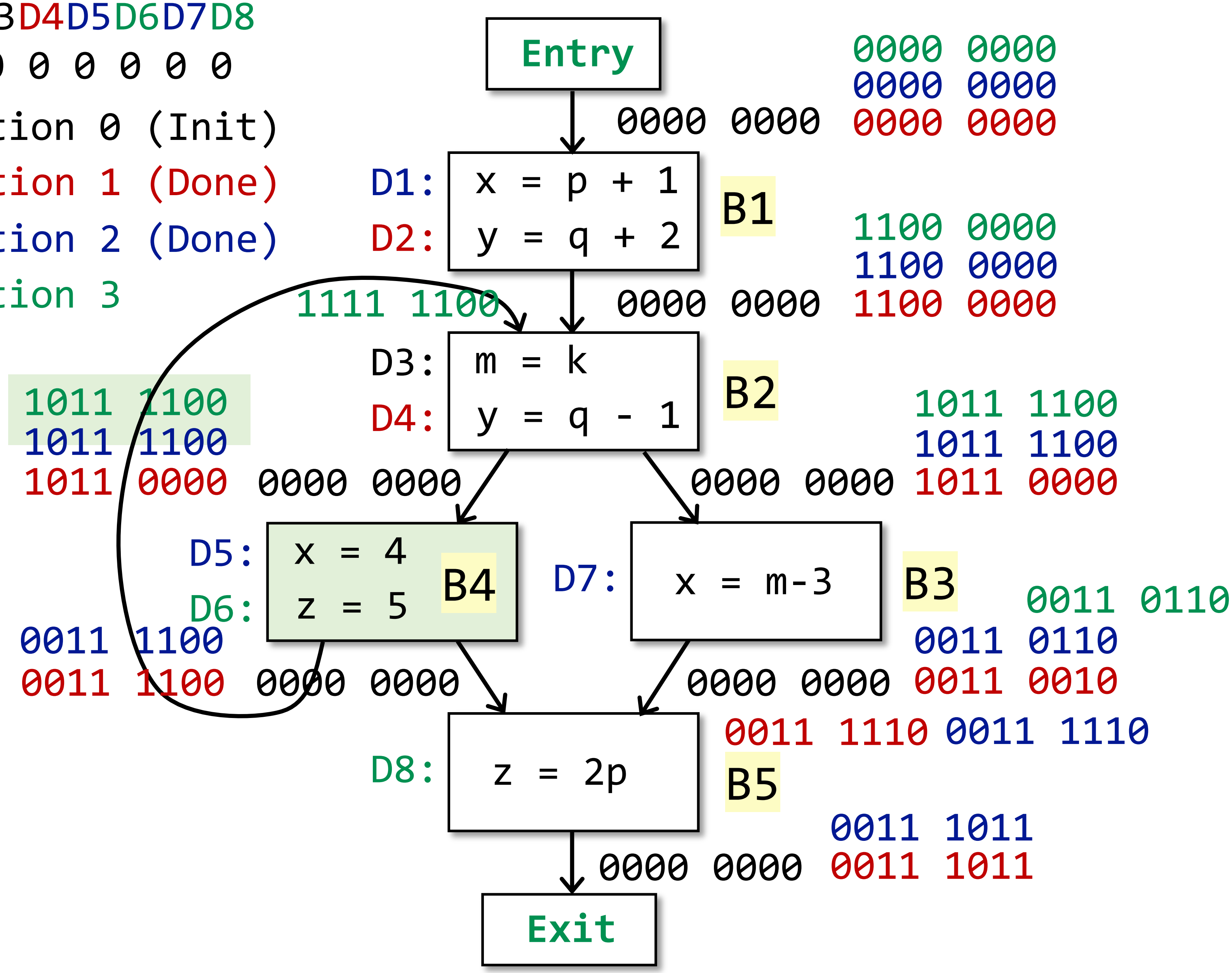
D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0

Iteration 0 (Init)

Iteration 1 (Done)

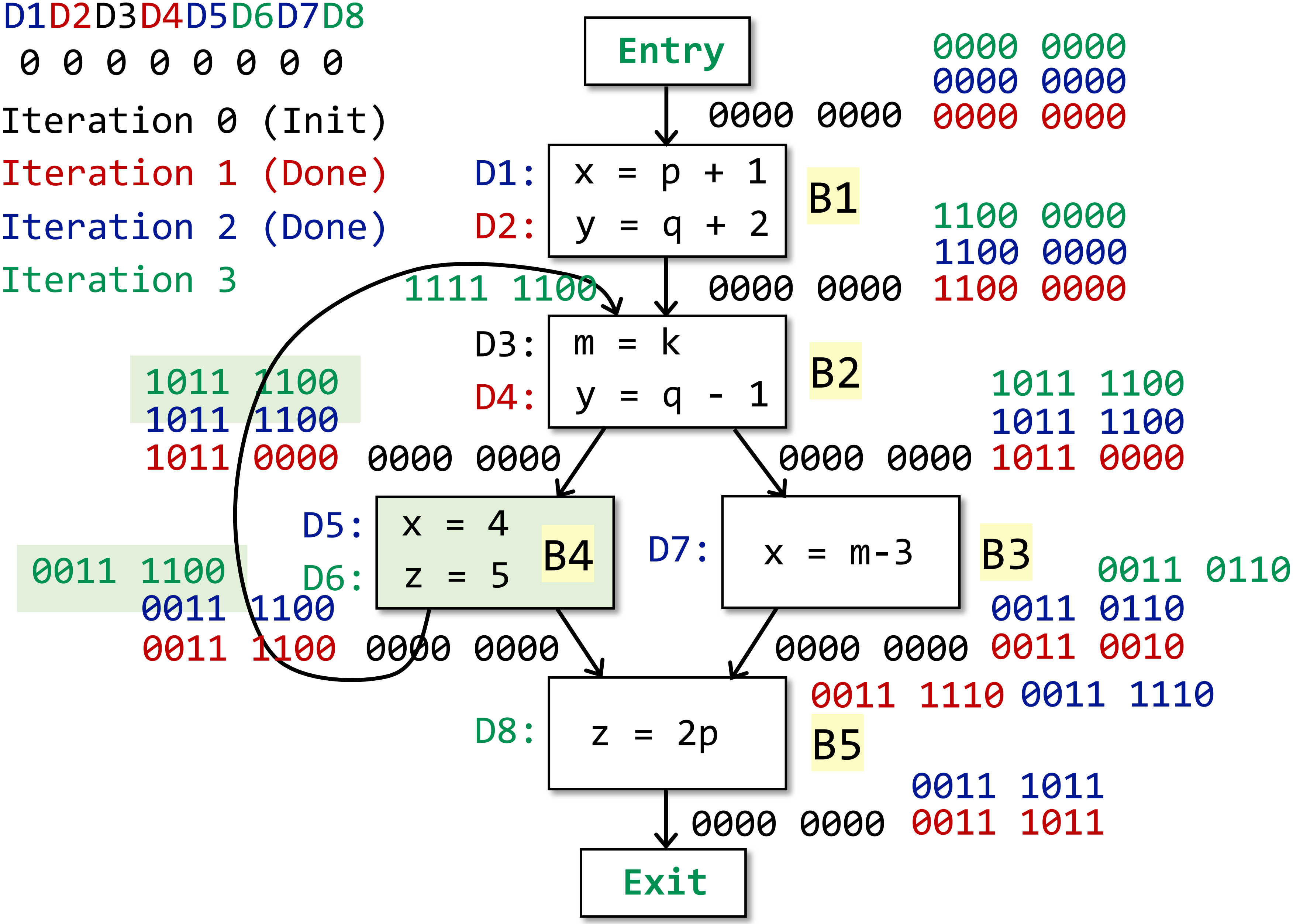
Iteration 2 (Done)

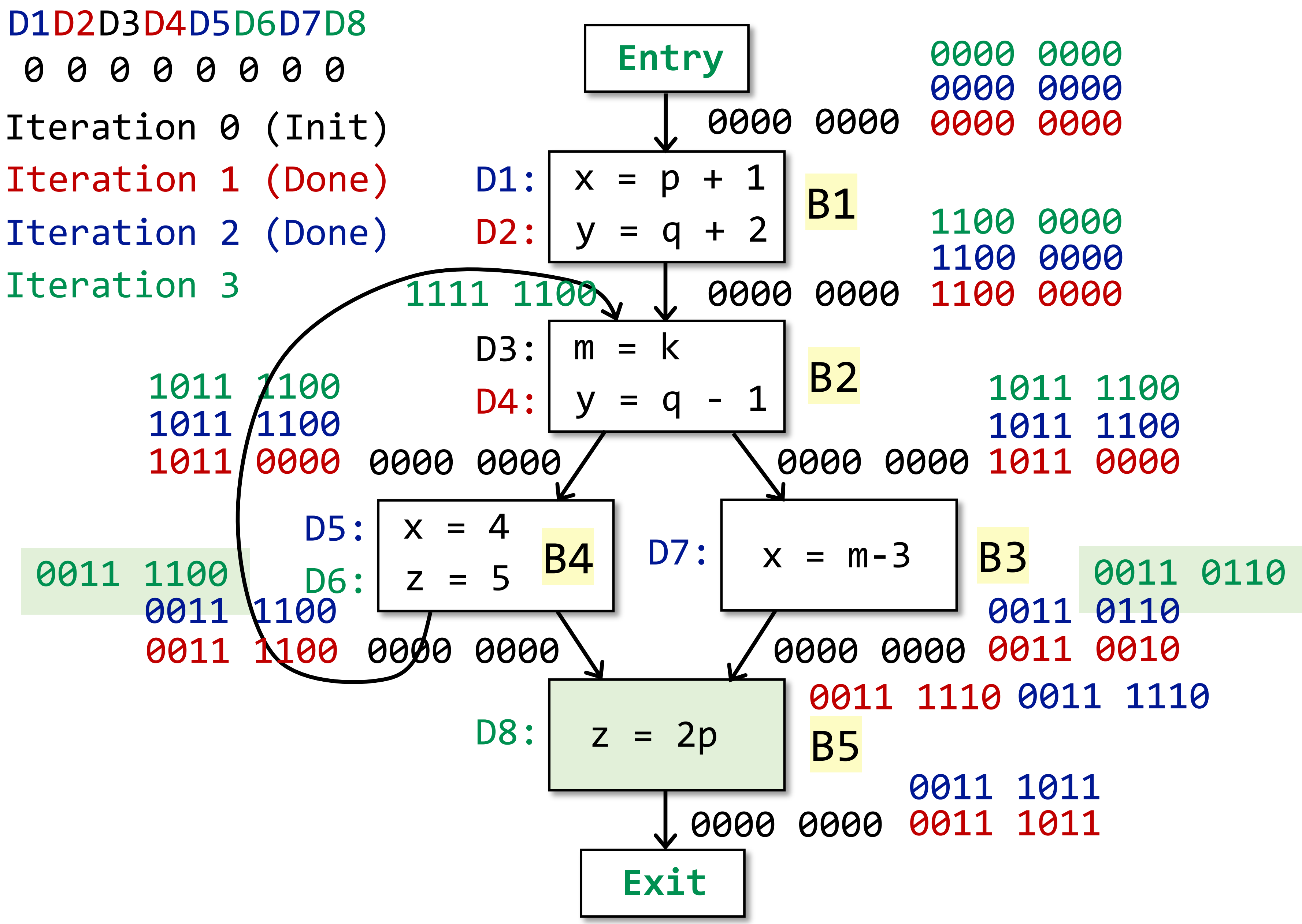
Iteration 3

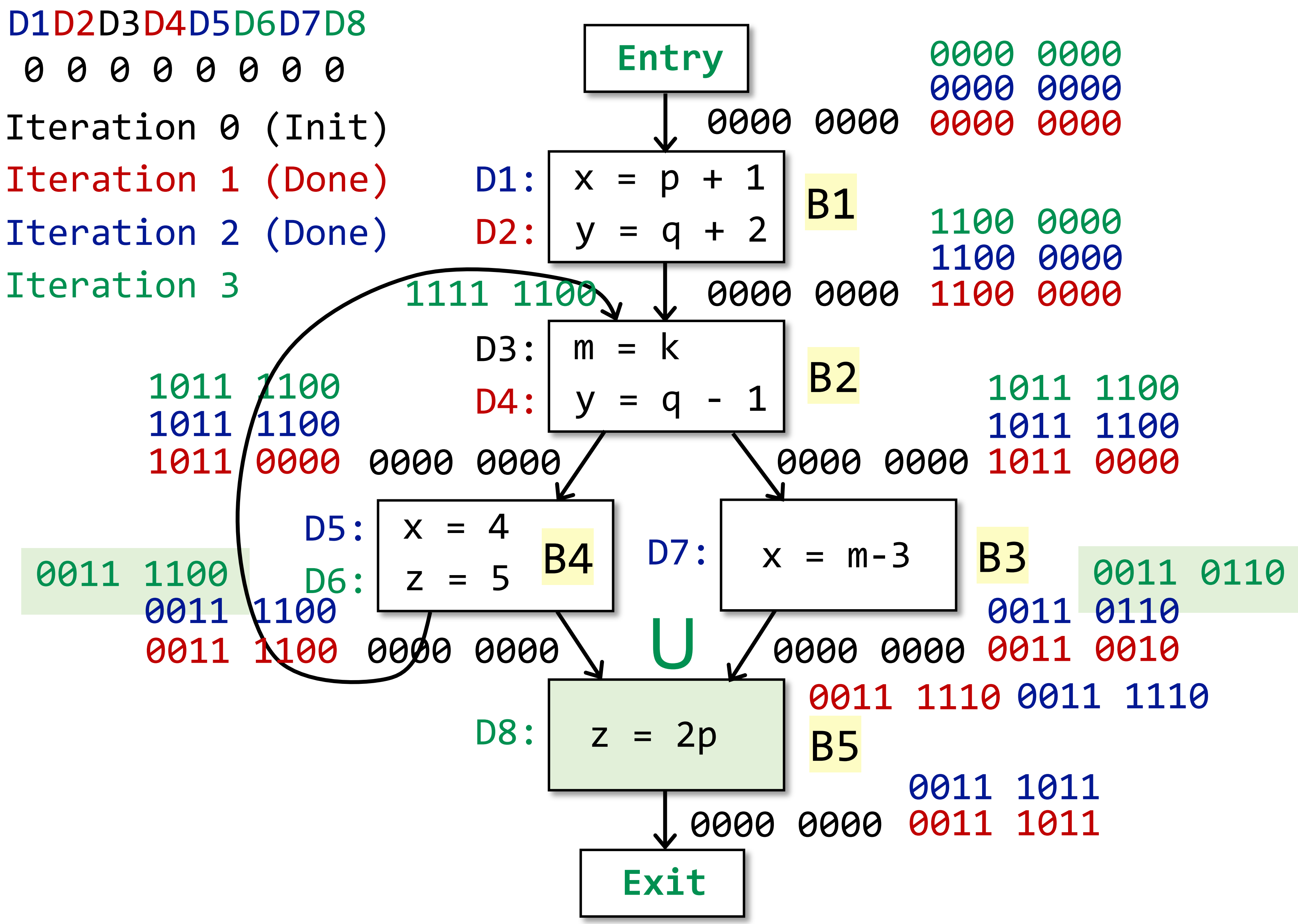


D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0

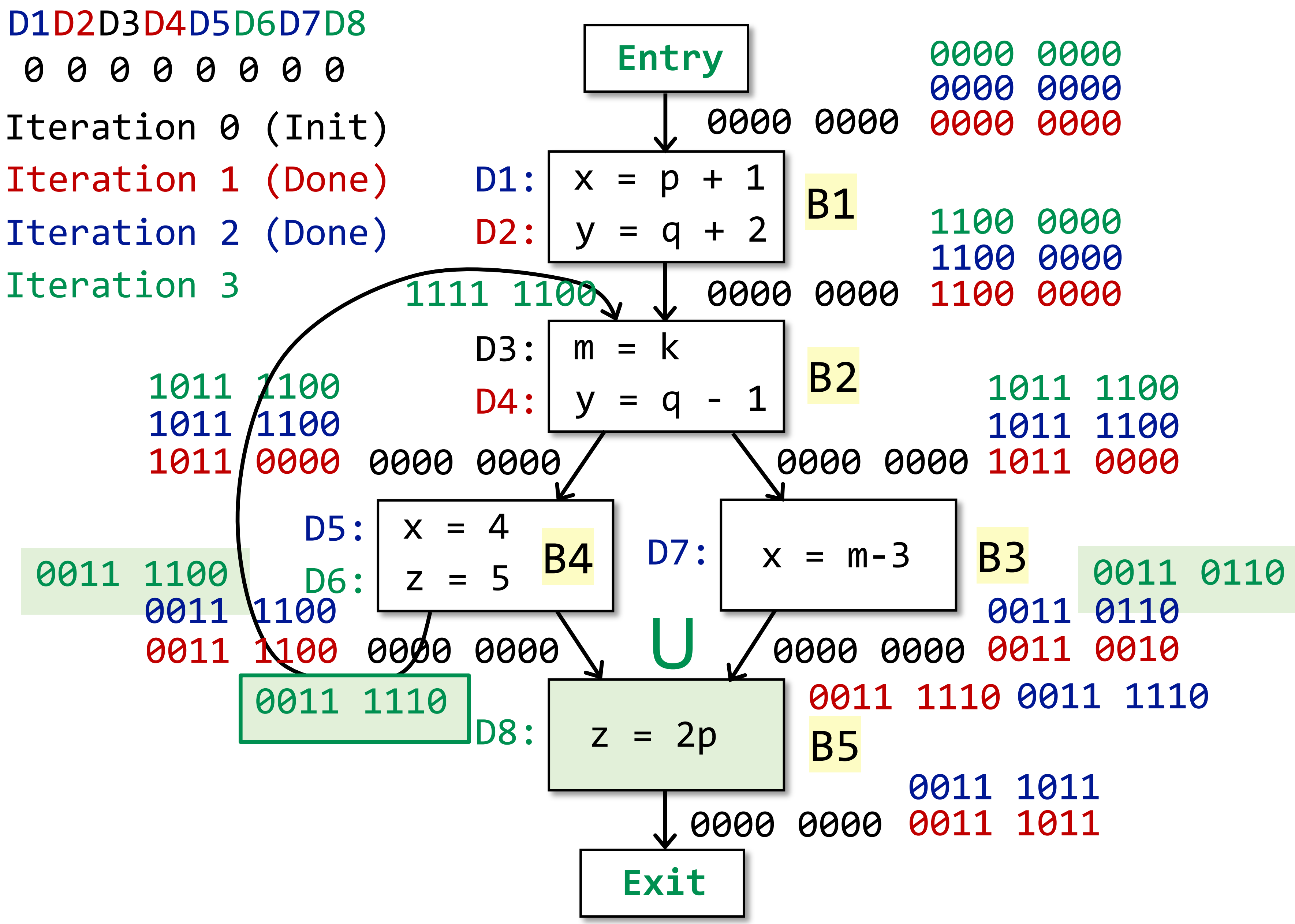
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2 (Done)
Iteration 3




D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2 (Done)
Iteration 3
Entry
0000 0000
0000 0000
0000 0000
0000 0000
D1: x = p + 1
D2: y = q + 2
B1
1100 0000
1100 0000
1100 0000
0000 0000
1111 1100
D3: m = k
D4: y = q - 1
B2
1011 1100
1011 1100
1011 0000
0000 0000
1011 1100
1011 1100
1011 0000
0000 0000
D5: x = 4
D6: z = 5
B4
D7: x = m-3
B3
0011 1100
0011 1100
0011 1100
0000 0000
0011 0110
0011 0110
0011 0010
0000 0000
D8: z = 2p
B5
0011 1110
0011 1110
0011 1011
0011 1011
0000 0000
Exit



D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2 (Done)
Iteration 3
Entry
0000 0000
0000 0000
0000 0000
0000 0000
D1: x = p + 1
D2: y = q + 2
B1
1100 0000
1100 0000
1100 0000
0000 0000
1111 1100
D3: m = k
D4: y = q - 1
B2
1011 1100
1011 1100
1011 0000
0000 0000
1011 1100
1011 1100
1011 0000
0000 0000
D5: x = 4
D6: z = 5
B4
0011 1100
0011 1100
0011 1100
0000 0000
D7: x = m-3
B3
0011 0110
0011 0110
0011 0010
0000 0000
U
0011 1110 0011 1110
D8: z = 2p
B5
0011 1011
0011 1011
0000 0000
Exit



D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2 (Done)
Iteration 3
Entry
0000 0000
0000 0000
0000 0000
0000 0000
D1: x = p + 1
D2: y = q + 2
B1
1100 0000
1100 0000
0000 0000
1100 0000
1111 1100
D3: m = k
D4: y = q - 1
B2
1011 1100
1011 1100
1011 0000
0000 0000
1011 1100
1011 1100
0000 0000
1011 0000
D5: x = 4
D6: z = 5
B4
D7: x = m-3
B3
0011 1100
0011 1100
0011 1100
0000 0000
U
0011 0110
0011 0110
0000 0000
0011 0010
0011 1110
D8: z = 2p
B5
0011 1110
0011 1110
0011 1011
0000 0000
0011 1011
Exit

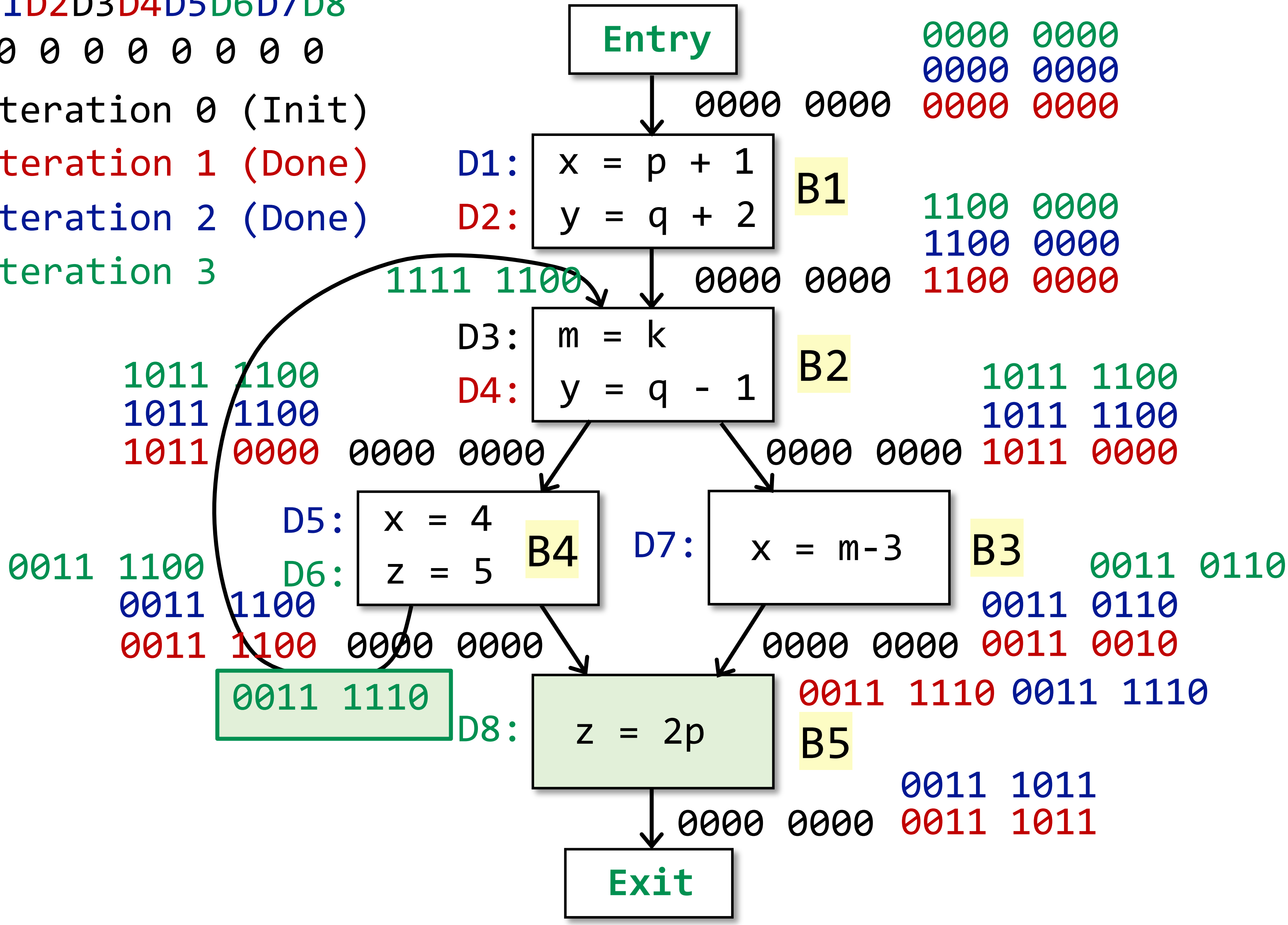
D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0

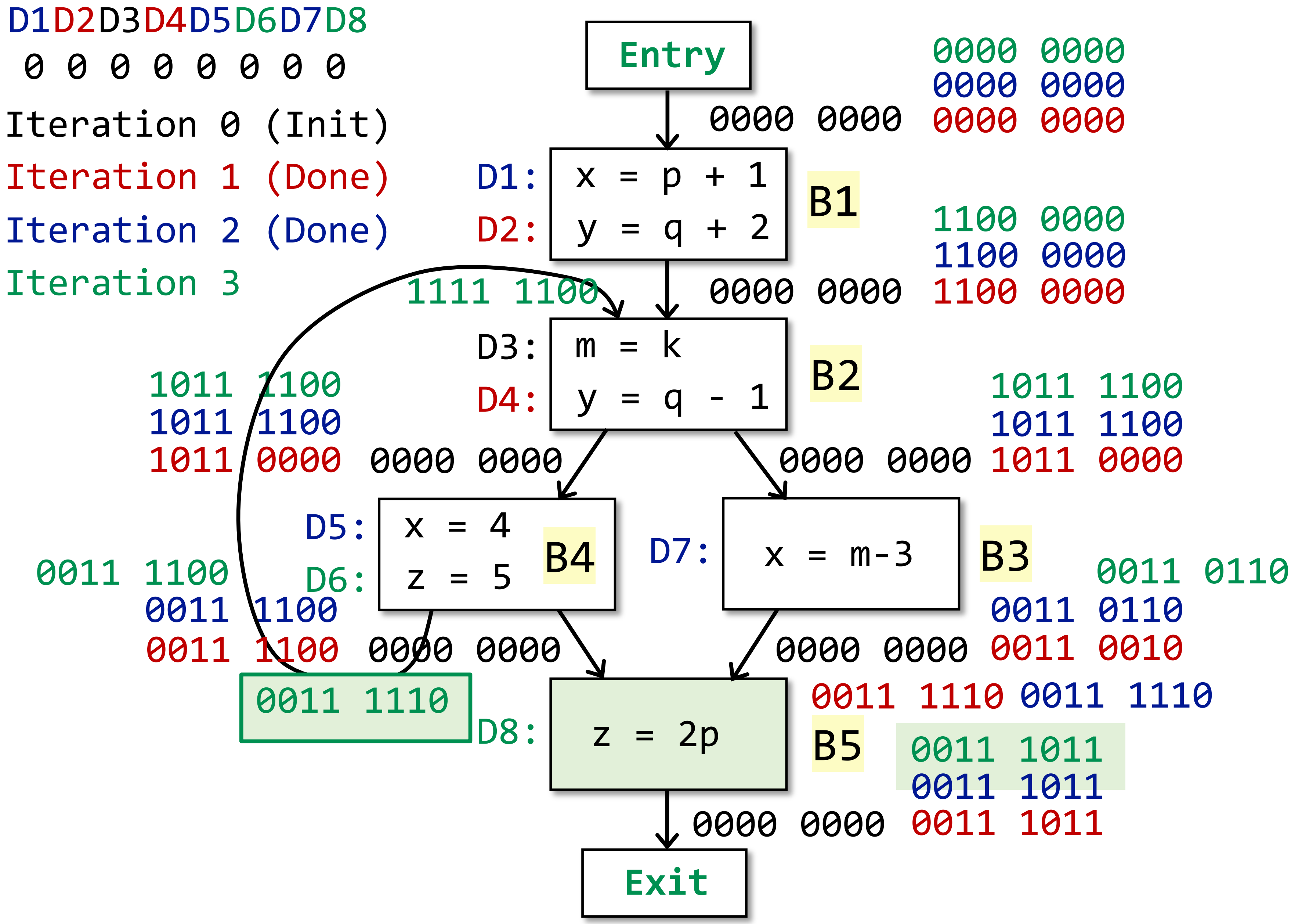
Iteration 0 (Init)

Iteration 1 (Done)

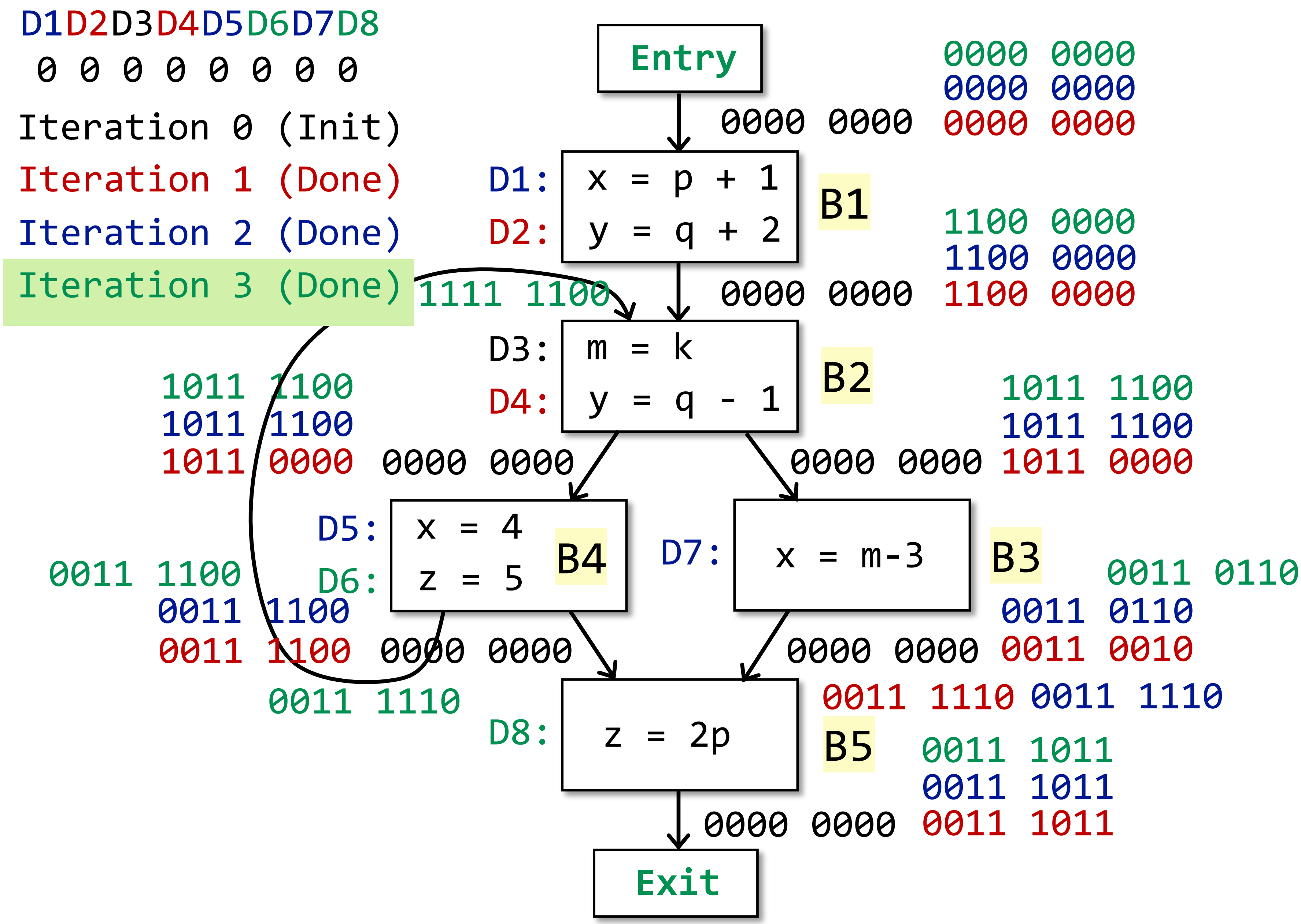
Iteration 2 (Done)

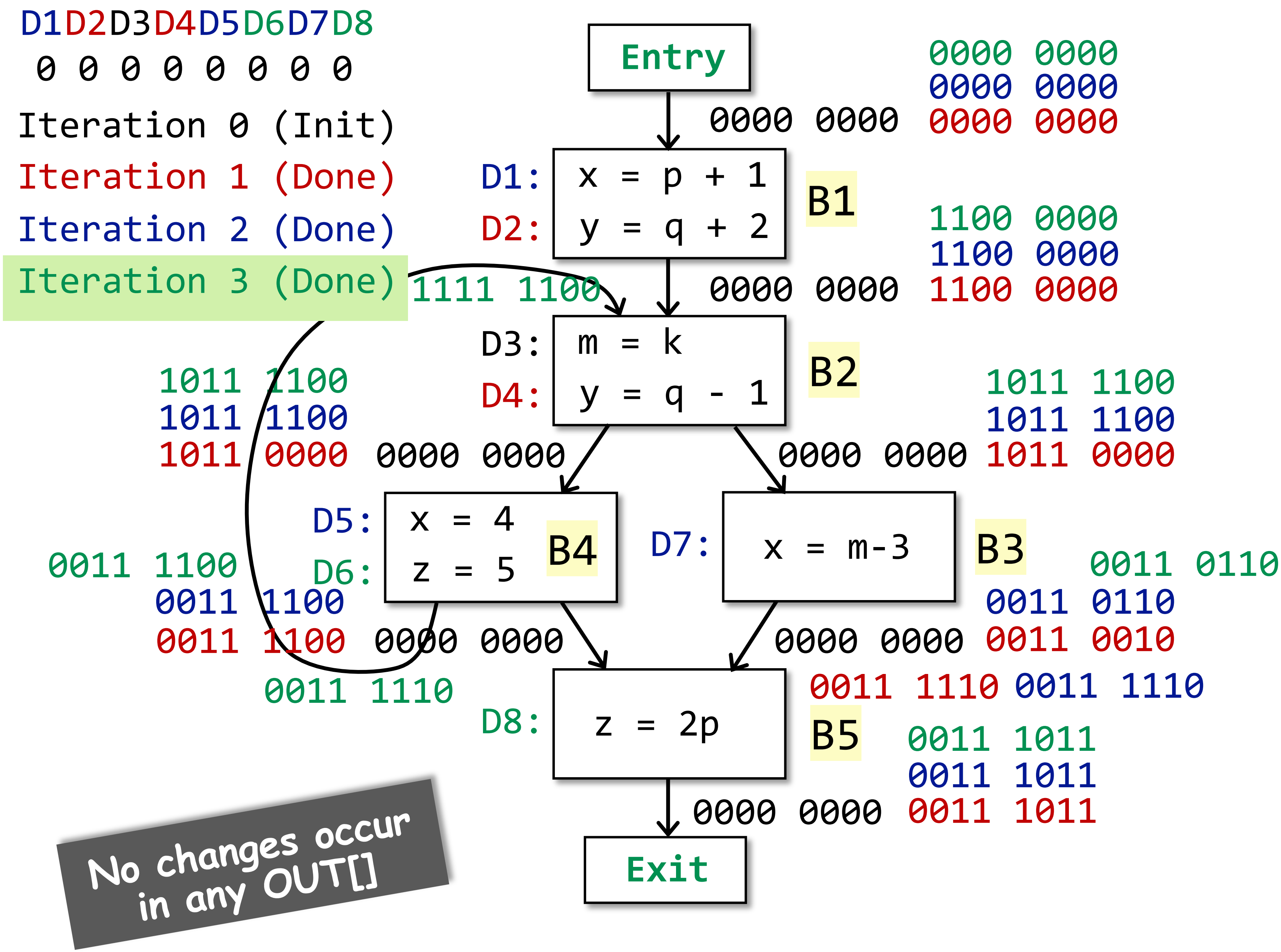
Iteration 3





D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2 (Done)
Iteration 3
Entry
0000 0000
0000 0000
0000 0000
0000 0000
D1: x = p + 1
D2: y = q + 2
B1
1100 0000
1100 0000
1100 0000
0000 0000
1111 1100
D3: m = k
D4: y = q - 1
B2
1011 1100
1011 1100
1011 0000
0000 0000
1011 1100
1011 1100
1011 0000
0000 0000
D5: x = 4
D6: z = 5
B4
0011 1100
0011 1100
0011 1100
0000 0000
D7: x = m-3
B3
0011 0110
0011 0110
0011 0010
0000 0000
0011 1110
D8: z = 2p
B5
0011 1110 0011 1110
0011 1011
0011 1011
0011 1011
0000 0000
Exit


D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2 (Done)
Iteration 3 (Done)
Entry
0000 0000
0000 0000
0000 0000
0000 0000
D1: x = p + 1
D2: y = q + 2
B1
1100 0000
1100 0000
1100 0000
0000 0000
1111 1100
D3: m = k
D4: y = q - 1
B2
1011 1100
1011 1100
1011 0000
0000 0000
1011 1100
1011 1100
1011 0000
0000 0000
D5: x = 4
D6: z = 5
B4
0011 1100
0011 1100
0011 1100
0000 0000
D7: x = m-3
B3
0011 0110
0011 0110
0011 0010
0000 0000
0011 1110
0011 1110
0011 1110
D8: z = 2p
B5
0011 1011
0011 1011
0011 1011
0000 0000
Exit


D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2 (Done)
Iteration 3 (Done)
Entry
0000 0000
0000 0000
0000 0000
0000 0000
D1: x = p + 1
D2: y = q + 2
B1
1100 0000
1100 0000
1100 0000
0000 0000
1111 1100
D3: m = k
D4: y = q - 1
B2
1011 1100
1011 1100
1011 0000
0000 0000
1011 1100
1011 1100
1011 0000
0000 0000
D5: x = 4
D6: z = 5
B4
D7: x = m-3
B3
0011 1100
0011 1100
0011 1100
0000 0000
0011 0110
0011 0110
0011 0010
0000 0000
0011 1110
0011 1110
0011 1110
D8: z = 2p
B5
0011 1011
0011 1011
0011 1011
0000 0000
Exit
No changes occur in any OUT[]

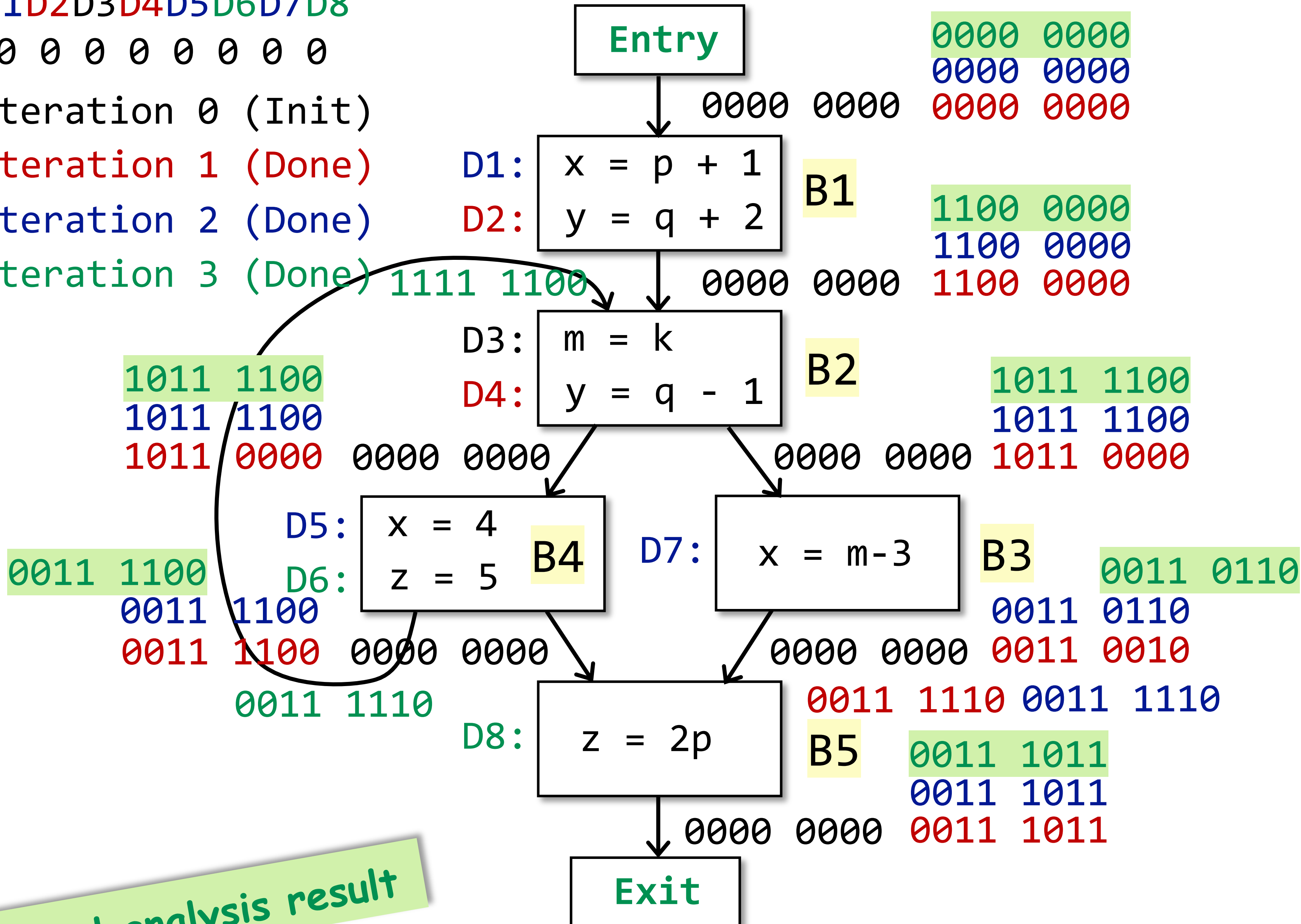
D1D2D3D4D5D6D7D8
0 0 0 0 0 0 0 0

Iteration 0 (Init)

Iteration 1 (Done)

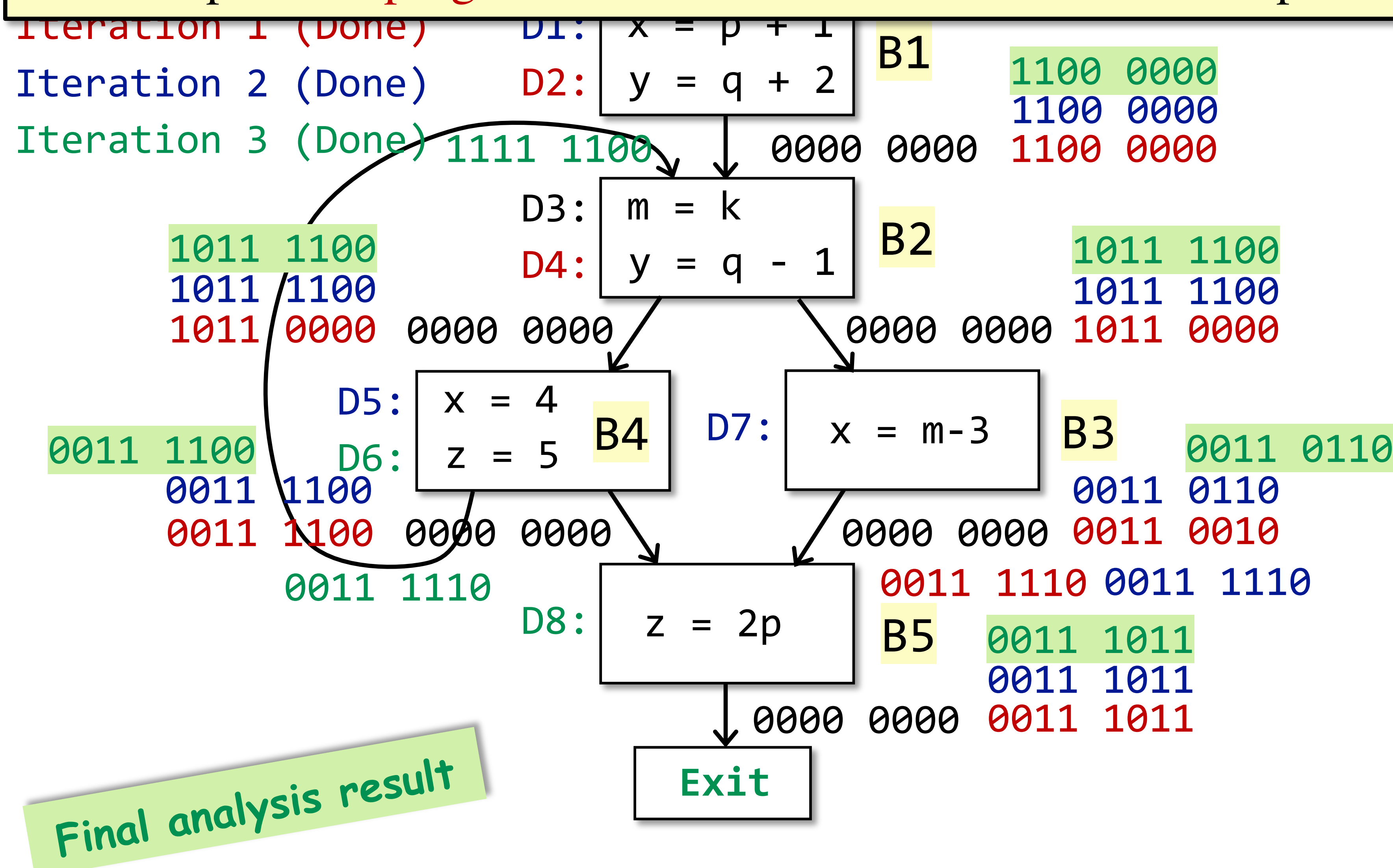
Iteration 2 (Done)

Iteration 3 (Done)

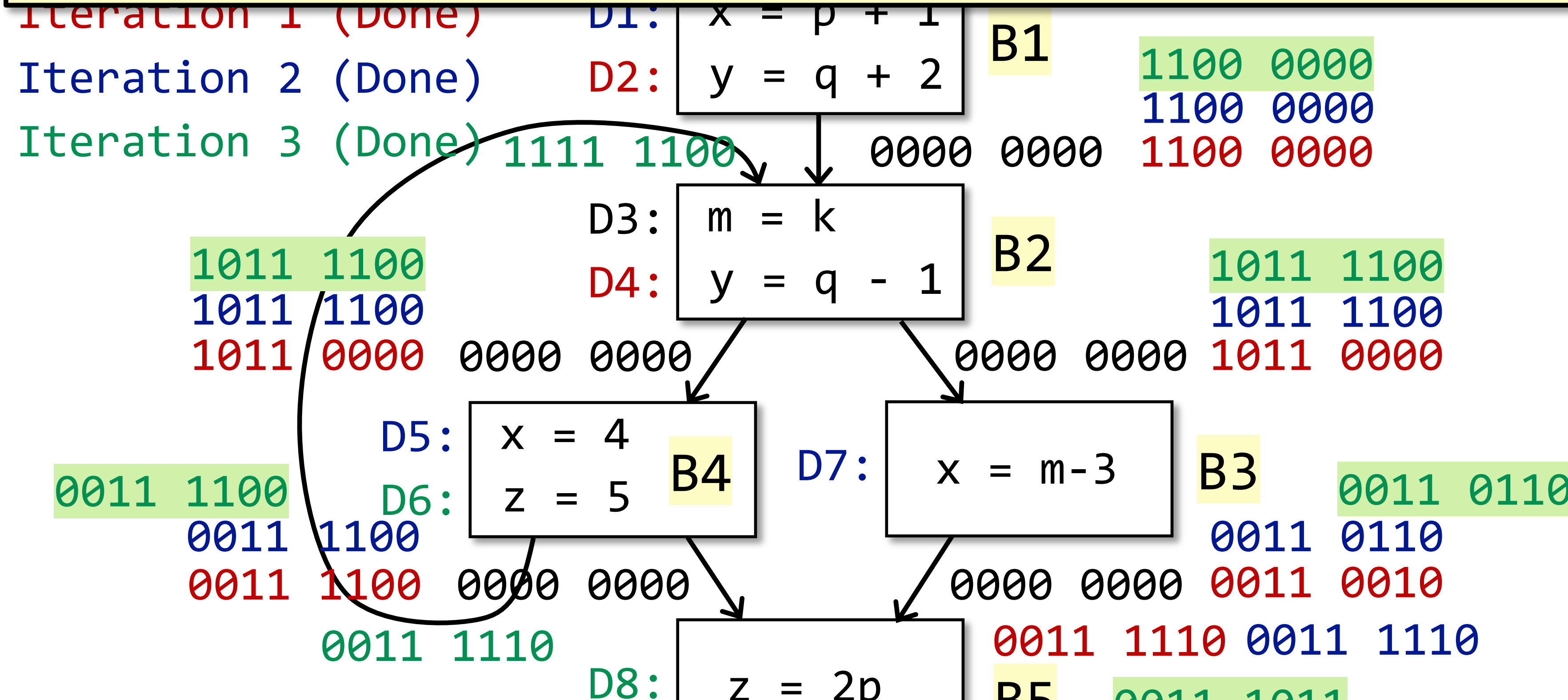


Final analysis result

In each data-flow analysis application, we associate with every program point a data-flow value that represents an *abstraction* of the set of all possible program states that can be observed for that point.



In each data-flow analysis application, we associate with every program point a data-flow value that represents an *abstraction* of the set of all possible program states that can be observed for that point.



Data-flow analysis is to find a solution to a set of *safe-approximation-directed constraints* on the IN[s]'s and OUT[s]'s, for all statements s.

- *constraints* based on semantics of statements (*transfer functions*)
- *constraints* based on the *flows of control*

Algorithm of Reaching Definitions Analysis

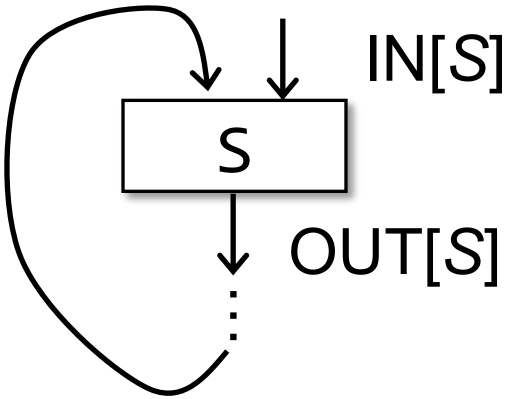
INPUT: CFG ($kill_B$ and gen_B computed for each basic block B)

OUTPUT: IN[B] and OUT[B] for each basic block B

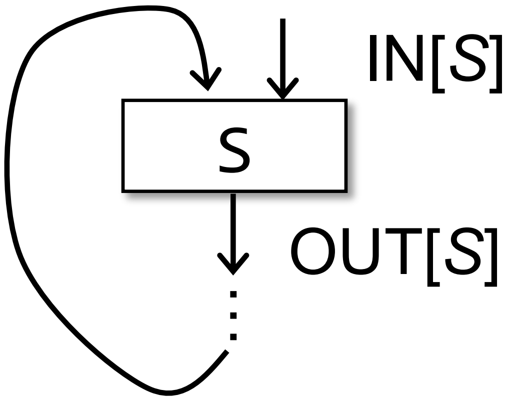
METHOD:

OUT[*entry*] = ∅;
for (each basic blo
    OUT[B] = ∅;
while (changes to any OUT occur)
    for (each basic block $B \setminus entry$) {
        $\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P]$;
        $\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B)$;
    }

Why this iterative algorithm can finally stop?

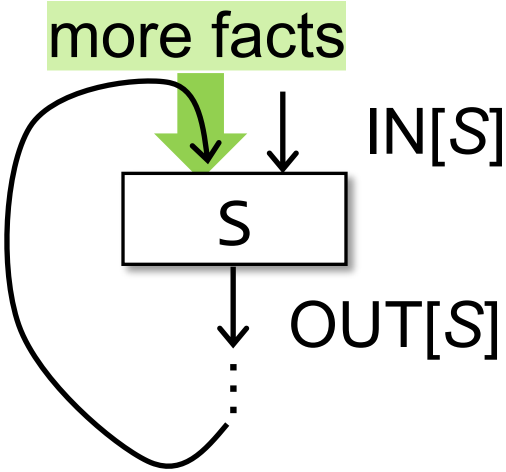


OUT[S] = gen_S U (IN[S] - $kill_S$);



$$\text{OUT}[S] = gen_S \cup (\text{IN}[S] - kill_S);$$

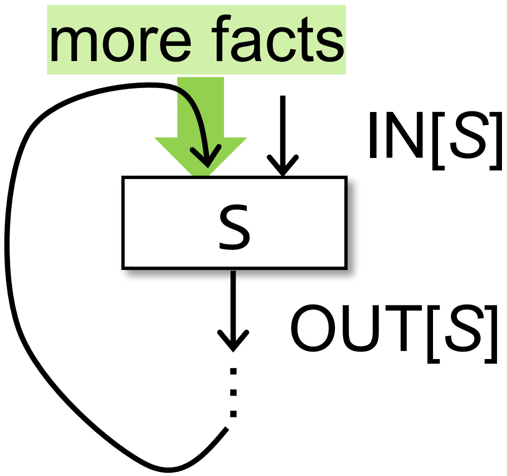
- gen_S and $kill_S$ remain unchanged



more facts

$$OUT[S] = gen_S \cup (IN[S] - kill_S);$$

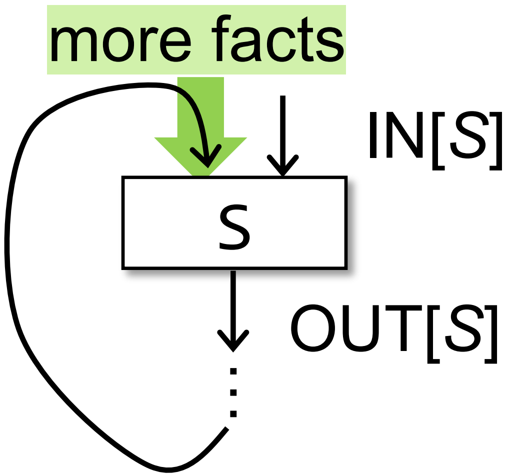
- gen_S and $kill_S$ remain unchanged
- When more facts flow in IN[*S*], the "more facts" either



more facts

$$OUT[S] = gen_S \cup \underbrace{(IN[S] - kill_S)}_{survivor_S};$$

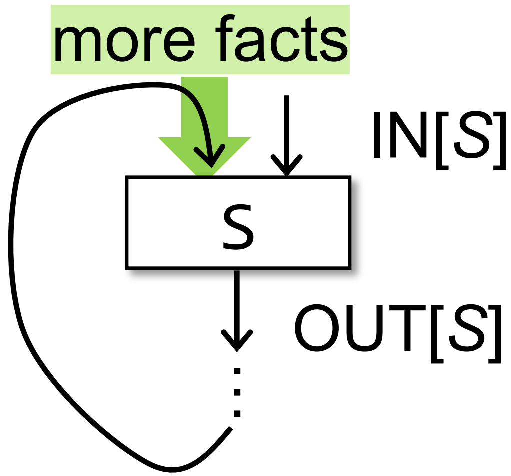
- gen_S and $kill_S$ remain unchanged
- When more facts flow in IN[*S*], the "more facts" either
  - is killed, or
  - flows to OUT[*S*] ($survivor_S$)



more facts

$$OUT[S] = gen_S \cup (\underbrace{IN[S] - kill_S}_{survivor_S});$$

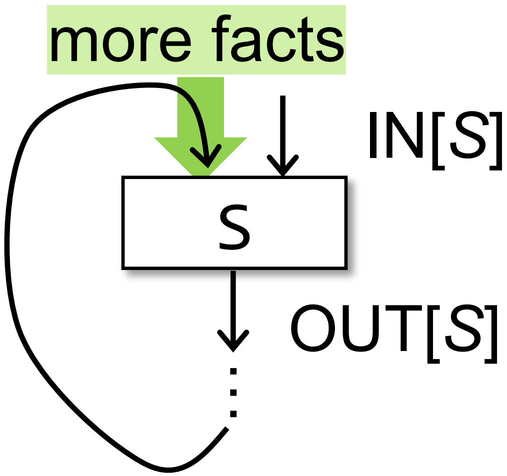
- gen_S and $kill_S$ remain unchanged
- When more facts flow in IN[S], the "more facts" either
  - is killed, or
  - flows to OUT[S] ($survivor_S$)
- When a fact is added to OUT[S], through either gen_S, or $survivor_S$, it stays there forever



more facts

$$OUT[S] = gen_S \cup (\underbrace{IN[S] - kill_S}_{survivor_S});$$

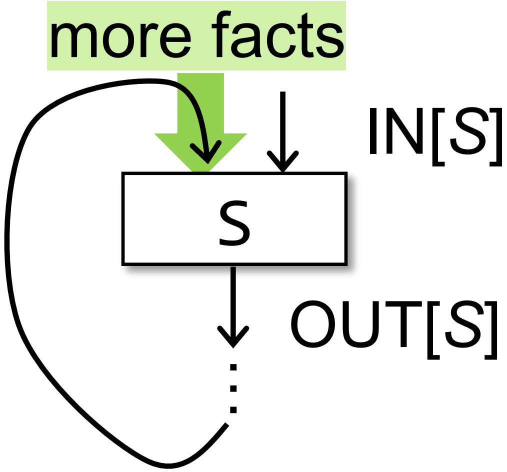
- gen_S and $kill_S$ remain unchanged
- When more facts flow in IN[S], the "more facts" either
  - is killed, or
  - flows to OUT[S] ($survivor_S$)
- When a fact is added to OUT[S], through either gen_S, or $survivor_S$, it stays there forever
- Thus OUT[S] never shrinks (e.g., 0→1, or 1→1)



more facts

$$OUT[S] = gen_S \cup (\underbrace{IN[S] - kill_S}_{survivor_S});$$

- gen_S and $kill_S$ remain unchanged
- When more facts flow in IN[*S*], the "more facts" either
  - is killed, or
  - flows to OUT[*S*] ($survivor_S$)
- When a fact is added to OUT[*S*], through either gen_S, or $survivor_S$, it stays there forever
- Thus OUT[*S*] never shrinks (e.g., 0→1, or 1→1)
- As the set of facts is finite (e.g., all definitions in the program),



more facts

$$\mathrm{OUT}[S] = gen_S \cup \underbrace{(\mathrm{IN}[S] - kill_S)}_{survivor_S};$$

- gen_S and $kill_S$ remain unchanged
- When more facts flow in IN[*S*], the "more facts" either
  - is killed, or
  - flows to OUT[*S*] ($survivor_S$)
- When a fact is added to OUT[*S*], through either gen_S, or $survivor_S$, it stays there forever
- Thus OUT[*S*] never shrinks (e.g., 0→1, or 1→1)
- As the set of facts is finite (e.g., all definitions in the program), there must exist a pass of iteration during which nothing is added to any OUT, and then the algorithm terminates

Algorithm of Reaching Definitions Analysis

INPUT: CFG ($kill_B$ and gen_B computed for each basic block B)

OUTPUT: IN[B] and OUT[B] for each basic block B

METHOD:

OUT[*entry*] = ∅;

for (each basic block *B*\e

  OUT[B] = ∅;

while (changes to any OUT occur)

  for (each basic block *B**entry*) {

    IN[B] = $\bigcup_{P \text{ a predecessor of } B}$ OUT[P];

    OUT[B] = $gen_B \cup (\text{IN}[B] - kill_B)$;

  }

Safe to terminate by this condition?

Algorithm of Reaching Definitions Analysis

INPUT: CFG ($kill_B$ and gen_B computed for each basic block B)

OUTPUT: IN[B] and OUT[B] for each basic block B

METHOD:

OUT[*entry*] = ∅;
for (each basic block B\e

Safe to terminate by this condition?

    OUT[B] = ∅;
while (changes to any OUT occur)
    for (each basic block B*entry*) {
        IN[B] = $\bigcup_{P \text{ a predecessor of } B}$ OUT[P];
        OUT[B] = gen_B U (IN[B] - $kill_B$);
    }

IN's will not change if OUT's do not change

Algorithm of Reaching Definitions Analysis

INPUT: CFG ($kill_B$ and gen_B computed for each basic block B)

OUTPUT: IN[B] and OUT[B] for each basic block B

METHOD:

OUT[*entry*] = ∅;

for (each basic block *B*\e

    OUT[*B*] = ∅;

Safe to terminate by this condition?

while (changes to any OUT occur)

    for (each basic block *B**entry*) {

        IN[B] = $\bigcup_{P \text{ a predecessor of } B}$ OUT[P];

        OUT[B] = gen_B U (IN[B] - $kill_B$);

    }

IN's will not change if OUT's do not change

OUT's will not change if IN's do not change

Algorithm of Reaching Definitions Analysis

INPUT: CFG ($kill_B$ and gen_B computed for each basic block B)

OUTPUT: IN[B] and OUT[B] for each basic block B

METHOD:

OUT[*entry*] = ∅;
for (each basic block $B \setminus entry$)
    OUT[B] = ∅;
while (changes to any OUT occur)
    for (each basic block $B \setminus entry$) {
        $IN[B] = \bigcup_{P \text{ a predecessor of } B} OUT[P]$;
        $OUT[B] = gen_B \cup (IN[B] - kill_B)$;
    }

Safe to terminate by this condition?

IN's will not change if OUT's do not change

OUT's will not change if IN's do not change

Reach a fixed point
Also related with monotonicity
(next lectures)

Data Flow Analysis Applications

(I) Reaching Definitions Analysis

(II) Live Variables Analysis

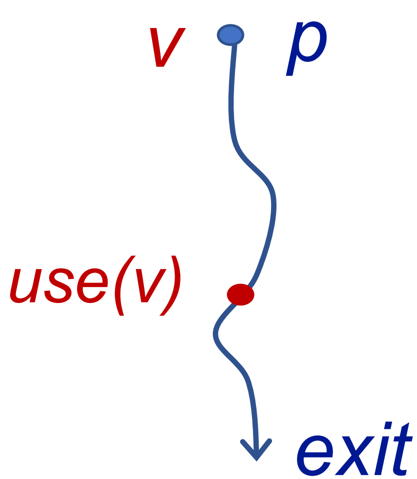
(III) Available Expressions Analysis

Live Variables Analysis

Live variables analysis tells whether the value of variable v at program point p could be used along some path in CFG starting at p. If so, v is live at p; otherwise, v is dead at p.

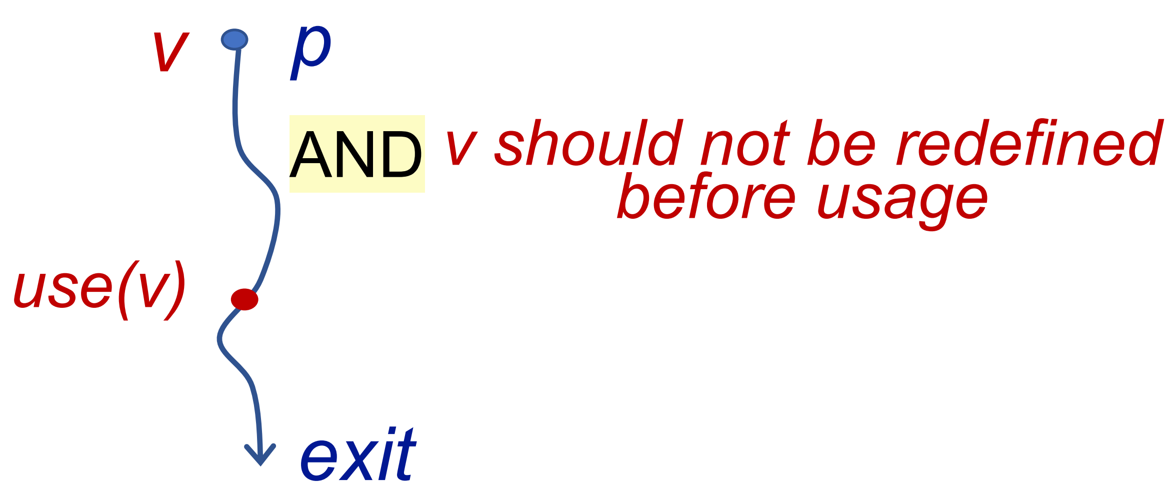
Live Variables Analysis

Live variables analysis tells whether the value of variable v at program point p could be used along some path in CFG starting at p. If so, v is live at p; otherwise, v is dead at p.



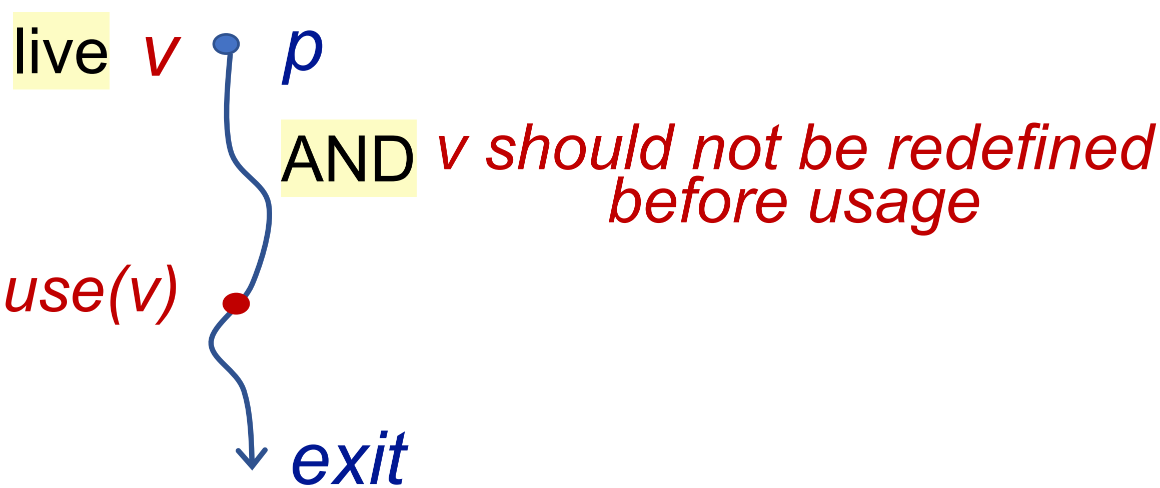
Live Variables Analysis

Live variables analysis tells whether the value of variable v at program point p could be used along some path in CFG starting at p. If so, v is live at p; otherwise, v is dead at p.



Live Variables Analysis

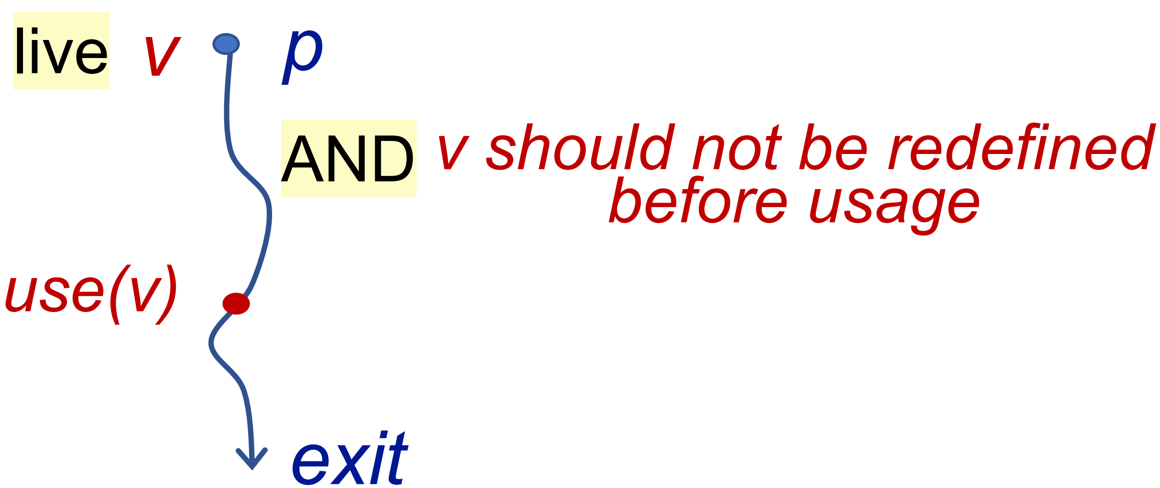
Live variables analysis tells whether the value of variable v at program point p could be used along some path in CFG starting at p. If so, v is live at p; otherwise, v is dead at p.



Live Variables Analysis

Live variables analysis tells whether the value of variable v at program point p could be used along some path in CFG starting at p. If so, v is live at p; otherwise, v is dead at p.

- Information of live variables can be used for register allocations. e.g., at some point all registers are full and we need to use one, then we should favor using a register with a dead value.



Understanding Live Variables Analysis

Abstraction

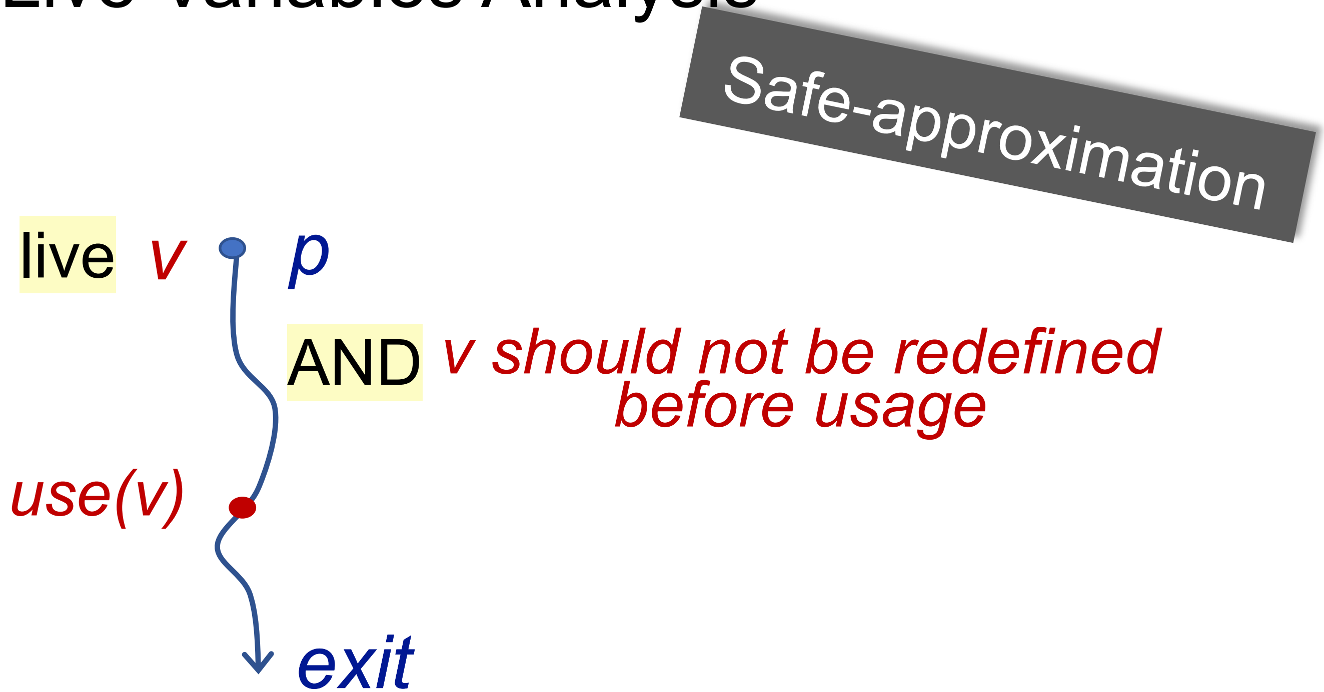
- Data Flow Values/Facts
  - All the variables in a program
  - Can be represented by bit vectors

e.g., V1, V2, V3, V4, …, V100 (100 variables)

$$\underbrace{00000\ldots0}_{\text{100 bits}}$$

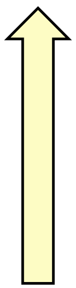
Bit i from the left represents variable Vi

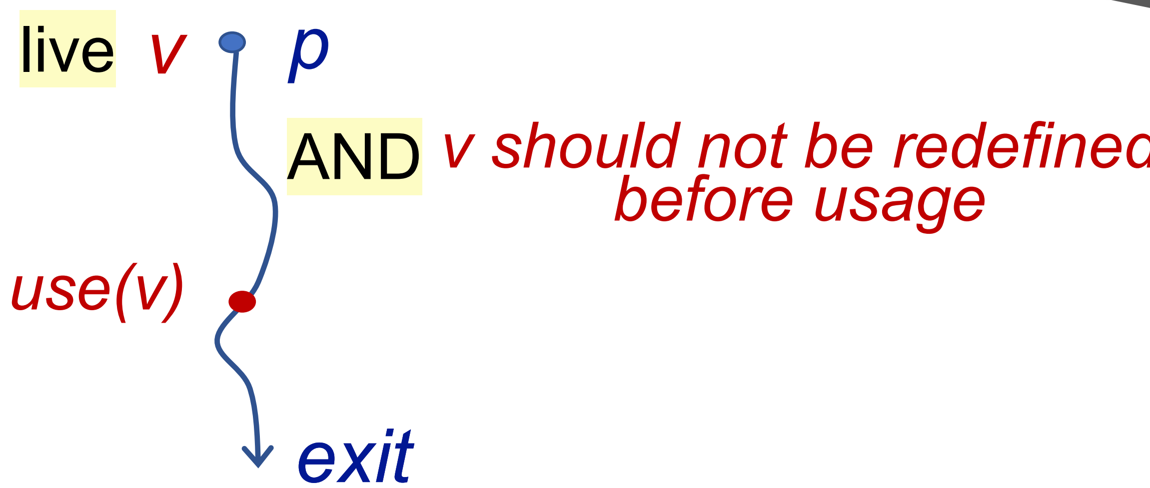
Understanding Live Variables Analysis



Understanding Live Variables Analysis

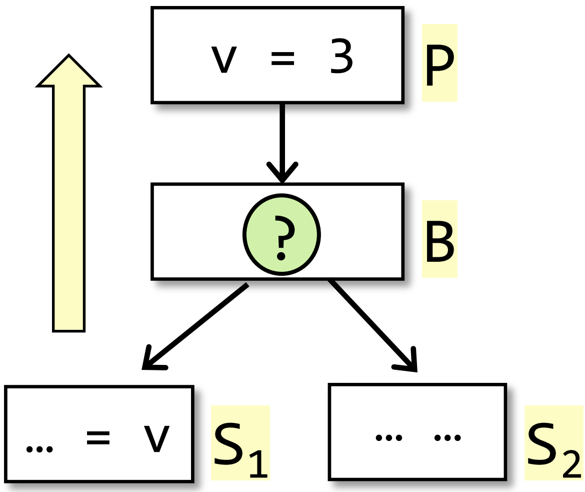
Safe-approximation



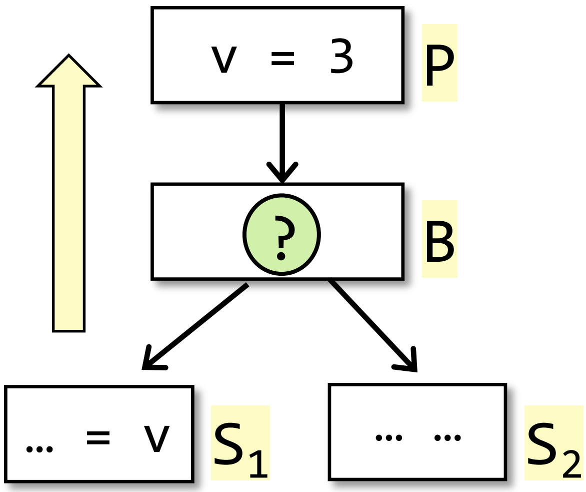


Understanding Live Variables Analysis

Safe-approximation

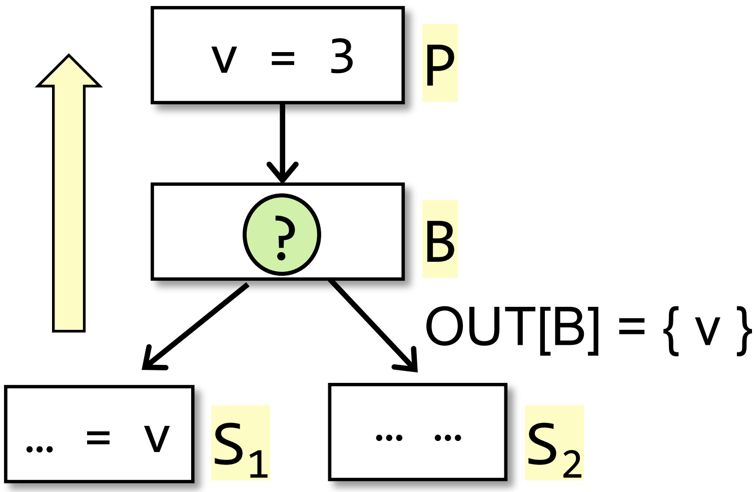


Understanding Live Variables Analysis



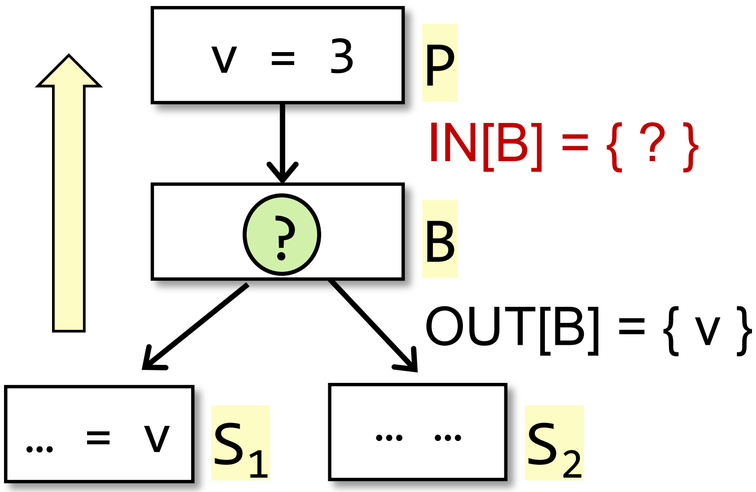
$$OUT[B] = \bigcup_{S \text{ a successor of } B} IN[S]$$

Understanding Live Variables Analysis



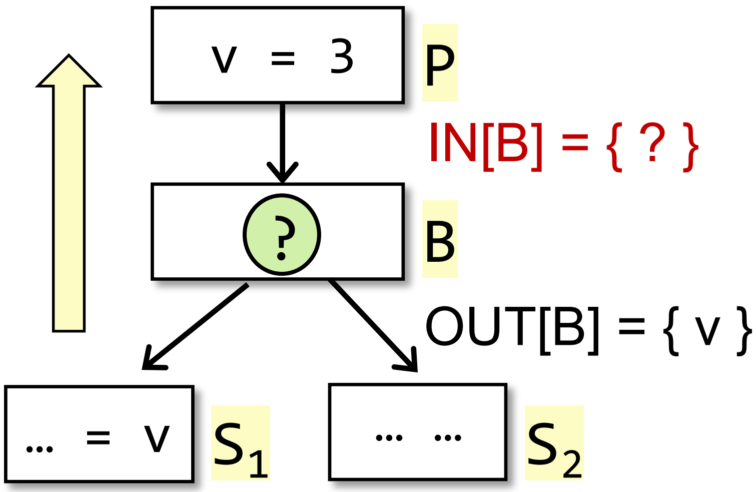
$$OUT[B] = \bigcup_{S \text{ a successor of } B} IN[S]$$

Understanding Live Variables Analysis



$$OUT[B] = \bigcup_{S \text{ a successor of } B} IN[S]$$

Understanding Live Variables Analysis

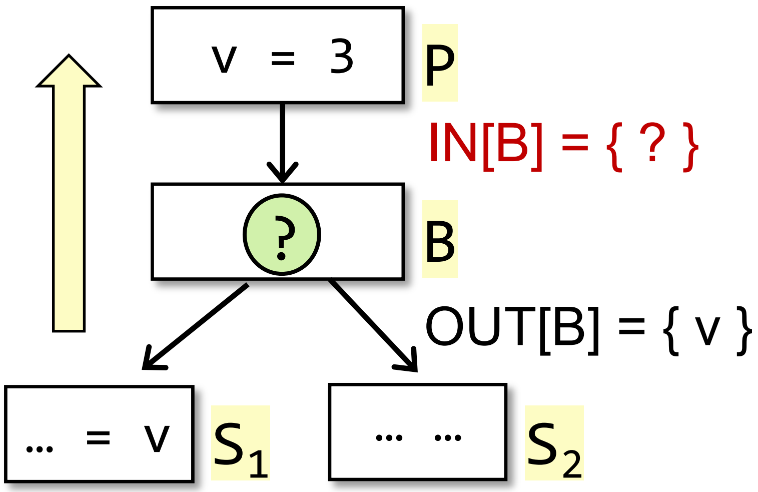


Tip: determine whether the variable v in some register R is live, or should we delete the value 3 of v in R, at the point of IN[B]?

Yes: IN[B] = { v } No: IN[B] = { }

$$OUT[B] = \bigcup_{S \text{ a successor of } B} IN[S]$$

Understanding Live Variables Analysis



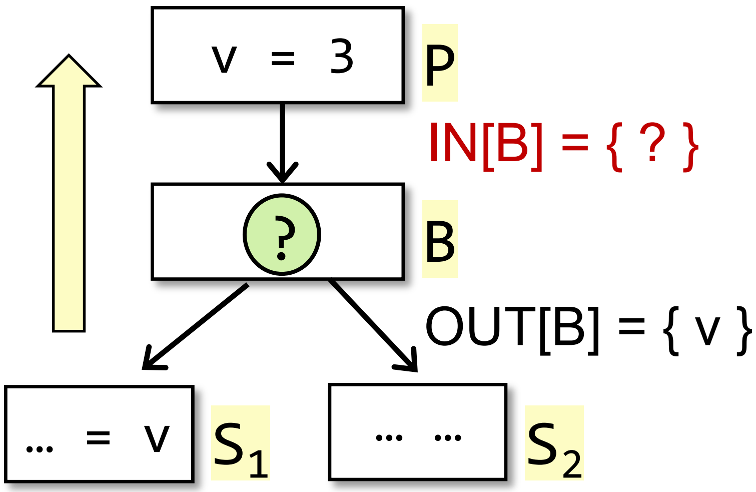
Tip: determine whether the variable v in some register R is live, or should we delete the value 3 of v in R, at the point of IN[B]?

Yes: IN[B] = { v } No: IN[B] = { }

1 k = n

$$OUT[B] = \bigcup_{S\ a\ successor\ of\ B} IN[S]$$

Understanding Live Variables Analysis



Tip: determine whether the variable v in some register R is live, or should we delete the value 3 of v in R, at the point of IN[B]?

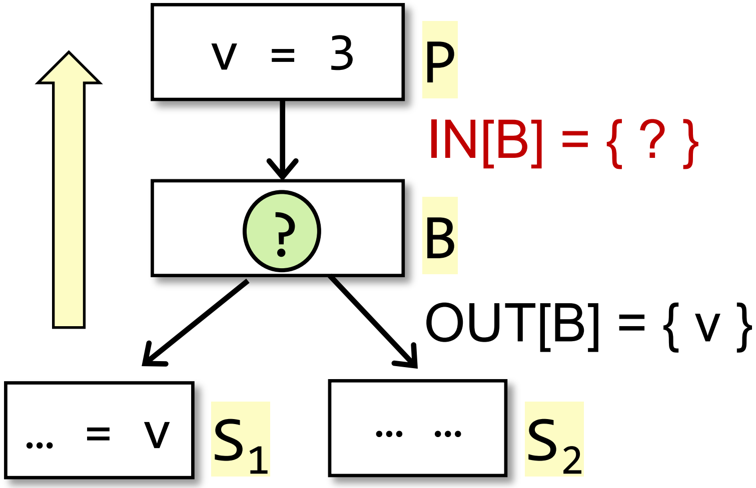
Yes: IN[B] = { v } No: IN[B] = { }

1 k = n

IN[B] = { v }

$$OUT[B] = \bigcup_{S \text{ a successor of } B} IN[S]$$

Understanding Live Variables Analysis



Tip: determine whether the variable v in some register R is live, or should we delete the value 3 of v in R, at the point of IN[B]?

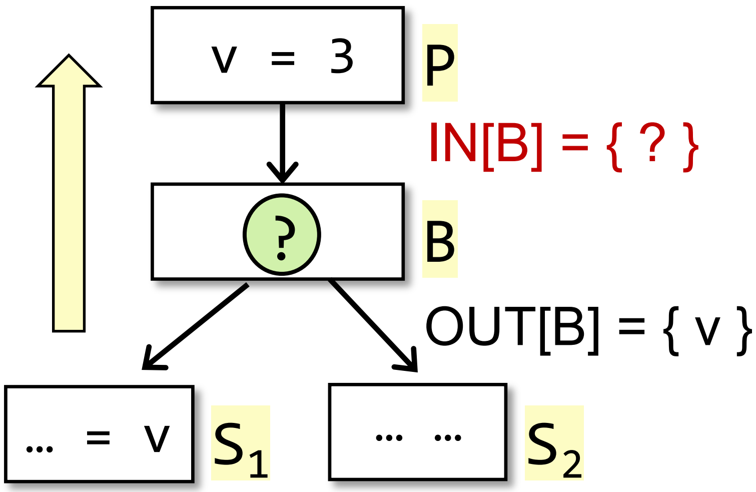
Yes: IN[B] = { v } No: IN[B] = { }

1 k = n 2 k = v

IN[B] = { v }

$$OUT[B] = \bigcup_{S\ a\ successor\ of\ B} IN[S]$$

Understanding Live Variables Analysis



Tip: determine whether the variable v in some register R is live, or should we delete the value 3 of v in R, at the point of IN[B]?

Yes: IN[B] = { v } No: IN[B] = { }

1 k = n

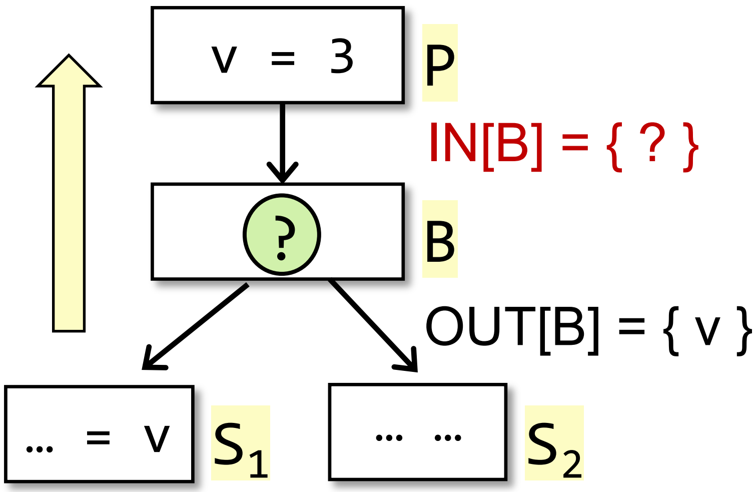
IN[B] = { v }

2 k = v

IN[B] = { v }

$$OUT[B] = \bigcup_{S \text{ a successor of } B} IN[S]$$

Understanding Live Variables Analysis



Tip: determine whether the variable v in some register R is live, or should we delete the value 3 of v in R, at the point of IN[B]?

Yes: IN[B] = { v } No: IN[B] = { }

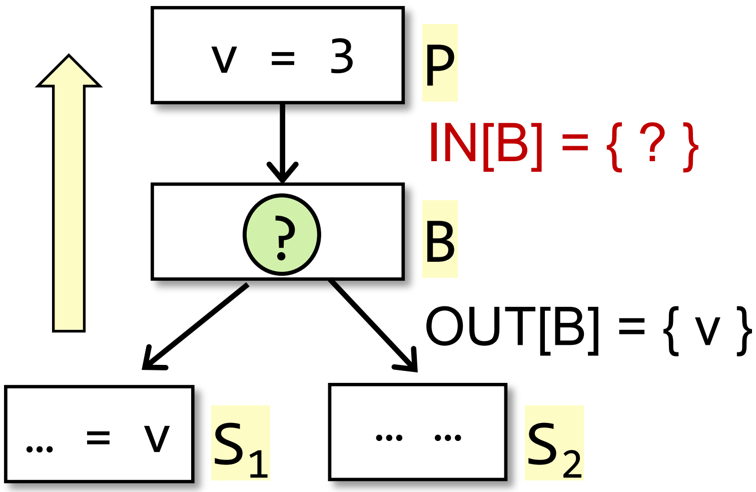
1 k = n
IN[B] = { v }

2 k = v
IN[B] = { v }

3 v = 2

$$\mathrm{OUT}[B] = \bigcup_{S \text{ a successor of } B} \mathrm{IN}[S]$$

Understanding Live Variables Analysis



Tip: determine whether the variable v in some register R is live, or should we delete the value 3 of v in R, at the point of IN[B]?

Yes: IN[B] = { v } No: IN[B] = { }

1 k = n

IN[B] = { v }

2 k = v

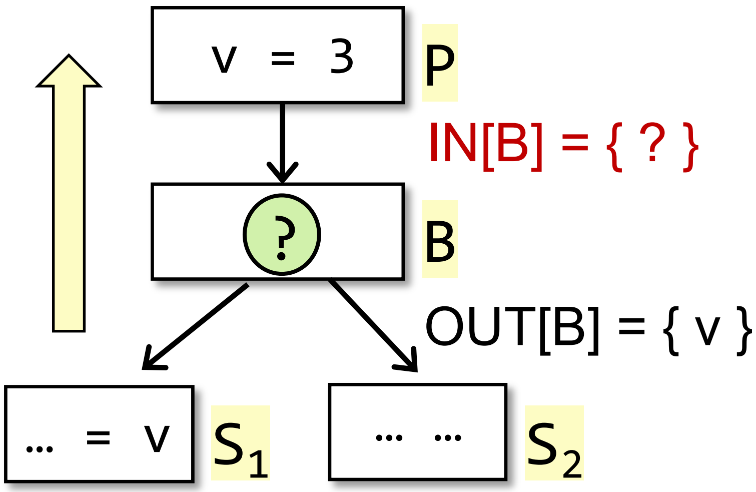
IN[B] = { v }

3 v = 2

IN[B] = { }

$$\text{OUT}[B] = \bigcup_{S \text{ a successor of } B} \text{IN}[S]$$

Understanding Live Variables Analysis



Tip: determine whether the variable v in some register R is live, or should we delete the value 3 of v in R, at the point of IN[B]?

Yes: IN[B] = { v } No: IN[B] = { }

1 k = n
IN[B] = { v }

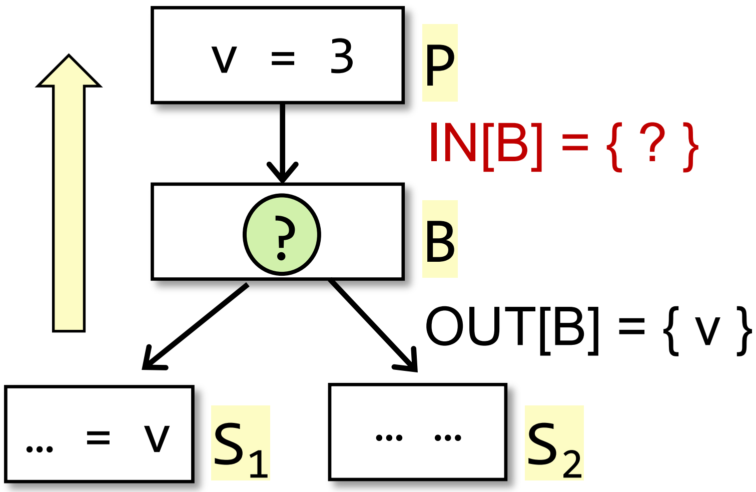
2 k = v
IN[B] = { v }

3 v = 2
IN[B] = { }

4 v = v-1

$$\text{OUT}[B] = \bigcup_{S \text{ a successor of } B} \text{IN}[S]$$

Understanding Live Variables Analysis



Tip: determine whether the variable v in some register R is live, or should we delete the value 3 of v in R, at the point of IN[B]?

Yes: IN[B] = { v } No: IN[B] = { }

1 `k = n`
IN[B] = { v }

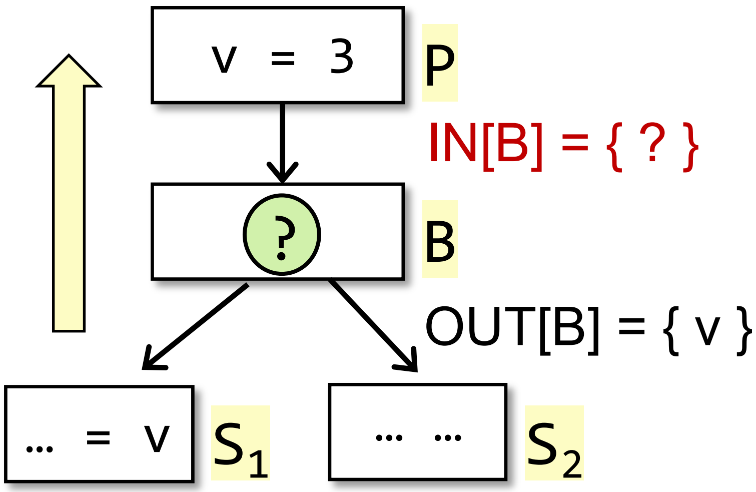
2 `k = v`
IN[B] = { v }

3 `v = 2`
IN[B] = { }

4 `v = v-1`
IN[B] = { v }

$$OUT[B] = \bigcup_{S \text{ a successor of } B} IN[S]$$

Understanding Live Variables Analysis



Tip: determine whether the variable v in some register R is live, or should we delete the value 3 of v in R, at the point of IN[B]?

Yes: IN[B] = { v } No: IN[B] = { }

1 `k = n`
IN[B] = { v }

2 `k = v`
IN[B] = { v }

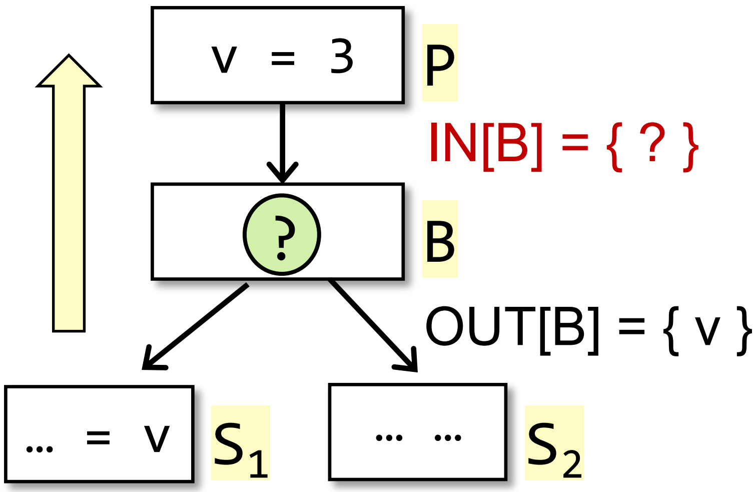
3 `v = 2`
IN[B] = { }

4 `v = v-1`
IN[B] = { v }

5
```
v = 2
k = v
```

$$OUT[B] = \bigcup_{S\ a\ successor\ of\ B} IN[S]$$

Understanding Live Variables Analysis



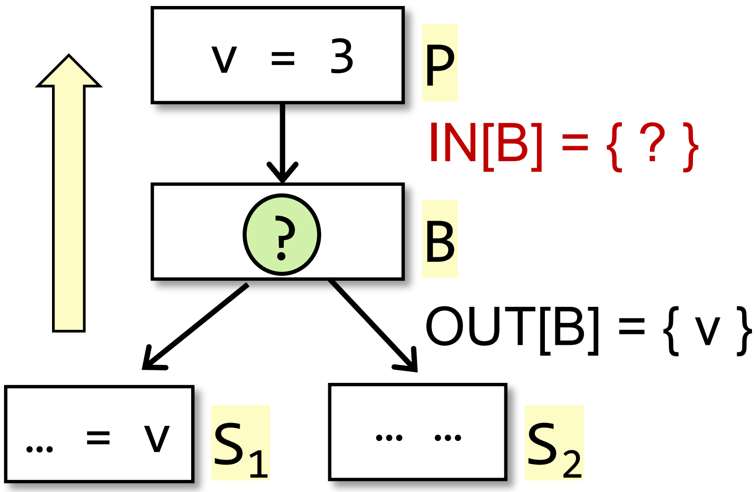
Tip: determine whether the variable v in some register R is live, or should we delete the value 3 of v in R, at the point of IN[B]?

Yes: IN[B] = { v } No: IN[B] = { }

1. k = n
   IN[B] = { v }
2. k = v
   IN[B] = { v }
3. v = 2
   IN[B] = { }
4. v = v-1
   IN[B] = { v }
5. v = 2
   k = v
   IN[B] = { }

$$OUT[B] = \bigcup_{S \text{ a successor of } B} IN[S]$$

Understanding Live Variables Analysis



Tip: determine whether the variable v in some register R is live, or should we delete the value 3 of v in R, at the point of IN[B]?

Yes: IN[B] = { v } No: IN[B] = { }

1 `k = n`
IN[B] = { v }

2 `k = v`
IN[B] = { v }

3 `v = 2`
IN[B] = { }

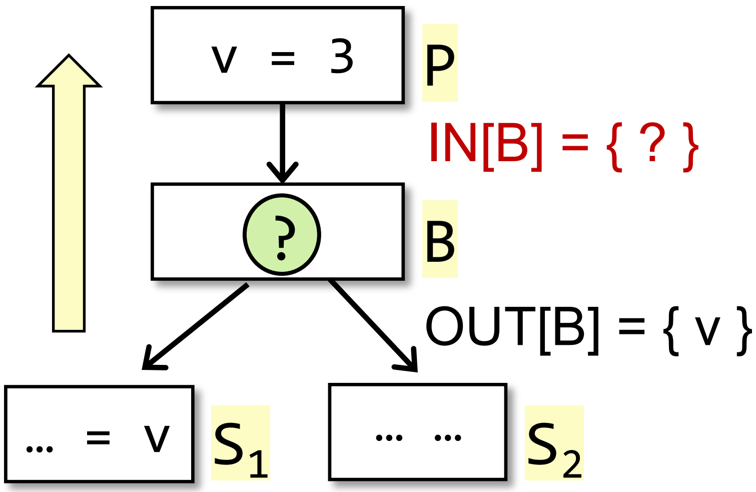
4 `v = v-1`
IN[B] = { v }

5 `v = 2`
`k = v`
IN[B] = { }

6 `k = v`
`v = 2`

$$OUT[B] = \bigcup_{S \text{ a successor of } B} IN[S]$$

Understanding Live Variables Analysis



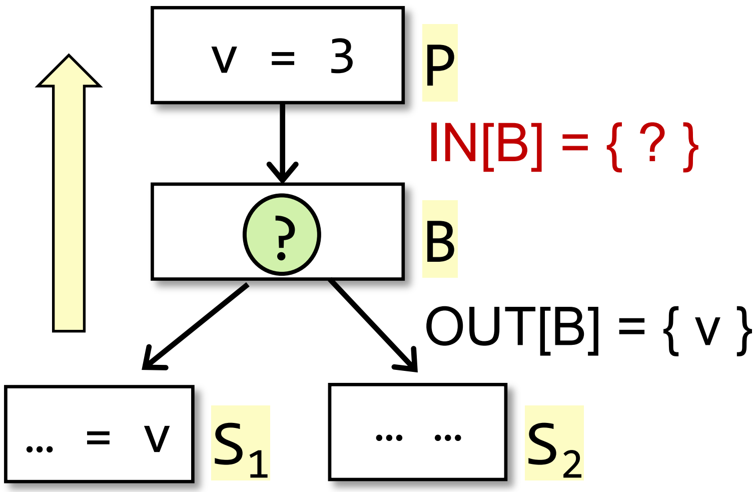
Tip: determine whether the variable v in some register R is live, or should we delete the value 3 of v in R, at the point of IN[B]?

Yes: IN[B] = { v } No: IN[B] = { }

1. `k = n`
   IN[B] = { v }
2. `k = v`
   IN[B] = { v }
3. `v = 2`
   IN[B] = { }
4. `v = v-1`
   IN[B] = { v }
5. `v = 2`
   `k = v`
   IN[B] = { }
6. `k = v`
   `v = 2`
   IN[B] = { v }

$$OUT[B] = \bigcup_{S \text{ a successor of } B} IN[S]$$

Understanding Live Variables Analysis



Tip: determine whether the variable v in some register R is live, or should we delete the value 3 of v in R, at the point of IN[B]?

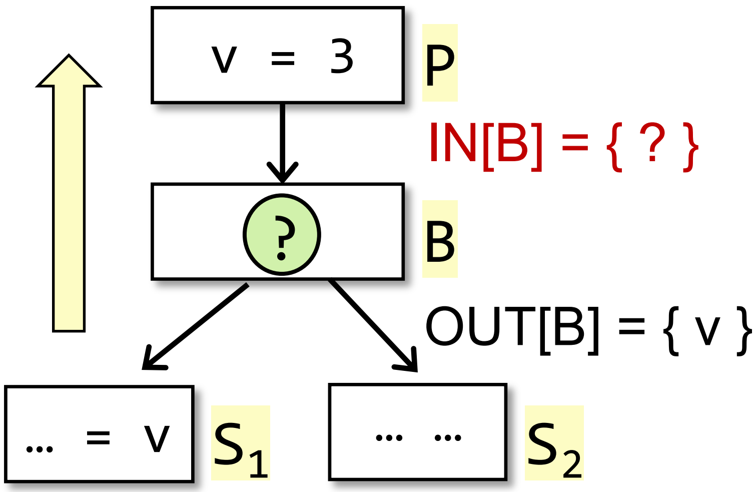
Yes: IN[B] = { v } No: IN[B] = { }

1. k = n
   IN[B] = { v }
2. k = v
   IN[B] = { v }
3. v = 2
   IN[B] = { }
4. v = v-1
   IN[B] = { v }
5. v = 2
   k = v
   IN[B] = { }
6. k = v
   v = 2
   IN[B] = { v }

$$OUT[B] = \bigcup_{S \text{ a successor of } B} IN[S]$$

$$IN[B] = use_B \cup (OUT[B] - def_B)$$

Understanding Live Variables Analysis



Tip: determine whether the variable v in some register R is live, or should we delete the value 3 of v in R, at the point of IN[B]?

Yes: IN[B] = { v } No: IN[B] = { }

1 `k = n`
IN[B] = { v }

2 `k = v`
IN[B] = { v }

3 `v = 2`
IN[B] = { }

4 `v = v-1`
IN[B] = { v }

5 `v = 2`
`k = v`
IN[B] = { }

6 `k = v`
`v = 2`
IN[B] = { v }

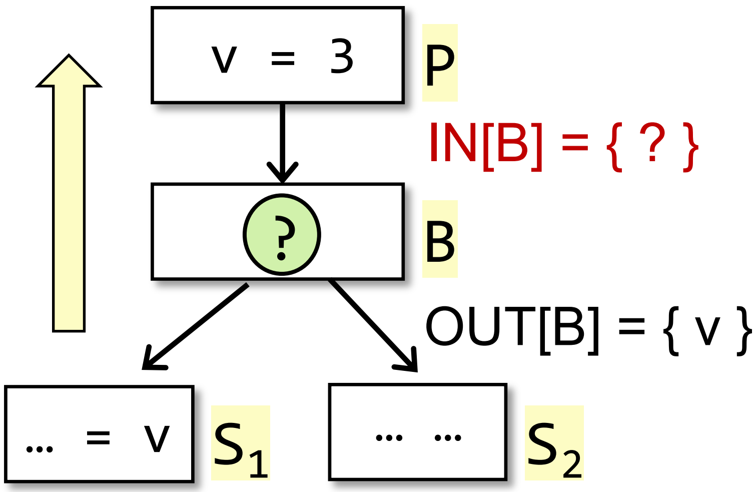
$$\text{OUT}[B] = \bigcup_{S \text{ a successor of } B} \text{IN}[S]$$

$$\text{IN}[B] = use_B \cup (\text{OUT}[B] - def_B)$$

It is redefined in B

3, 4, 5, 6

Understanding Live Variables Analysis



Tip: determine whether the variable v in some register R is live, or should we delete the value 3 of v in R, at the point of IN[B]?

Yes: IN[B] = { v } No: IN[B] = { }

1. k = n
   IN[B] = { v }
2. k = v
   IN[B] = { v }
3. v = 2
   IN[B] = { }
4. v = v-1
   IN[B] = { v }
5. v = 2
   k = v
   IN[B] = { }
6. k = v
   v = 2
   IN[B] = { v }

$$OUT[B] = \bigcup_{S \text{ a successor of } B} IN[S]$$

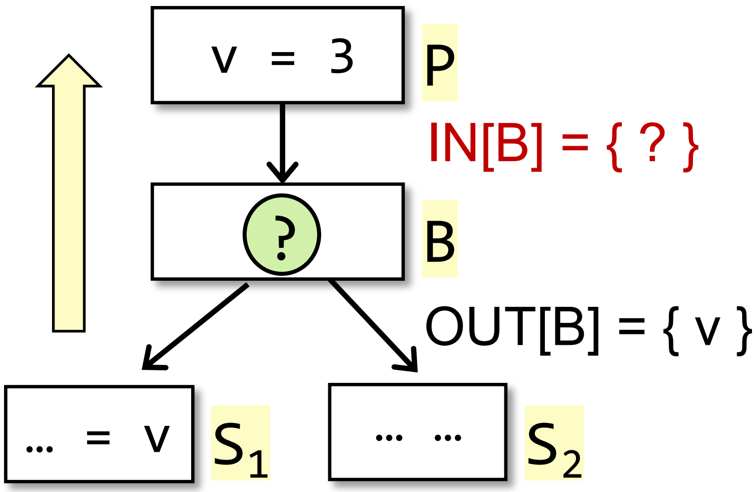
$$IN[B] = use_B \cup (OUT[B] - def_B)$$

It is redefined in B

3, 4, 5, 6

It is live coming out of B and is not redefined in B

Understanding Live Variables Analysis



Tip: determine whether the variable v in some register R is live, or should we delete the value 3 of v in R, at the point of IN[B]?

Yes: IN[B] = { v } No: IN[B] = { }

1. `k = n`
   IN[B] = { v }
2. `k = v`
   IN[B] = { v }
3. `v = 2`
   IN[B] = { }
4. `v = v-1`
   IN[B] = { v }
5. `v = 2`
   `k = v`
   IN[B] = { }
6. `k = v`
   `v = 2`
   IN[B] = { v }

$$\text{OUT}[B] = \bigcup_{S\ a\ successor\ of\ B} \text{IN}[S]$$

$$\text{IN}[B] = use_B \cup (\text{OUT}[B] - def_B)$$

It is used before redefinition in B

4, 6

It is live coming out of B and is not redefined in B

It is redefined in B

3, 4, 5, 6

Algorithm of Live Variables Analysis

INPUT: CFG (def_B and use_B computed for each basic block B)

OUTPUT: IN[B] and OUT[B] for each basic block B

METHOD:

IN[*exit*]= ∅;
for (each basic block *B\exit*)
    IN[B] = ∅;
while (changes to any IN occur)
    for (each basic block *B\exit*) {
        $\text{OUT}[B] = \bigcup_{S \text{ a successor of } B} \text{IN}[S];$
            $\text{IN}[B] = use_B \cup (\text{OUT}[B] - def_B);$
    }

Algorithm of Live Variables Analysis

INPUT: CFG (def_B and use_B computed for each basic block B)

OUTPUT: IN[B] and OUT[B] for each basic block B

METHOD:

IN[*exit*] = ∅;
for (each basic block $B \backslash exit$)
    IN[B] = ∅;
while (changes to any IN occur)
    for (each basic block $B \backslash exit$) {
        $\text{OUT}[B] = \bigcup_{S \text{ a successor of } B} \text{IN}[S]$;
            $\text{IN}[B] = use_B \cup (\text{OUT}[B] - def_B)$;
    }

Algorithm of Live Variables Analysis

INPUT: CFG (def_B and use_B computed for each basic block *B*)

OUTPUT: IN[*B*] and OUT[*B*] for each basic block *B*

METHOD:

IN[*exit*] = ∅;
for (each basic block *B\exit*)
    IN[*B*] = ∅;
while (changes to any IN occur)
    for (each basic block *B\exit*) {
        $\text{OUT}[B] = \bigcup_{S \text{ a successor of } B} \text{IN}[S]$;
        $\text{IN}[B] = use_B \cup (\text{OUT}[B] - def_B)$;
    }

Algorithm of Live Variables Analysis

INPUT: CFG (def_B and use_B computed for each basic block B)

OUTPUT: IN[B] and OUT[B] for each basic block B

METHOD:

IN[*exit*] = ∅;
for (each basic block $B \backslash exit$)
    IN[B] = ∅;
while (changes to any IN occur)
    for (each basic block $B \backslash exit$) {
        $\text{OUT}[B] = \bigcup_{S \text{ a successor of } B} \text{IN}[S]$;
            $\text{IN}[B] = use_B \cup (\text{OUT}[B] - def_B)$;
    }

Algorithm of Live Variables Analysis

INPUT: CFG (def_B and use_B computed for each basic block B)

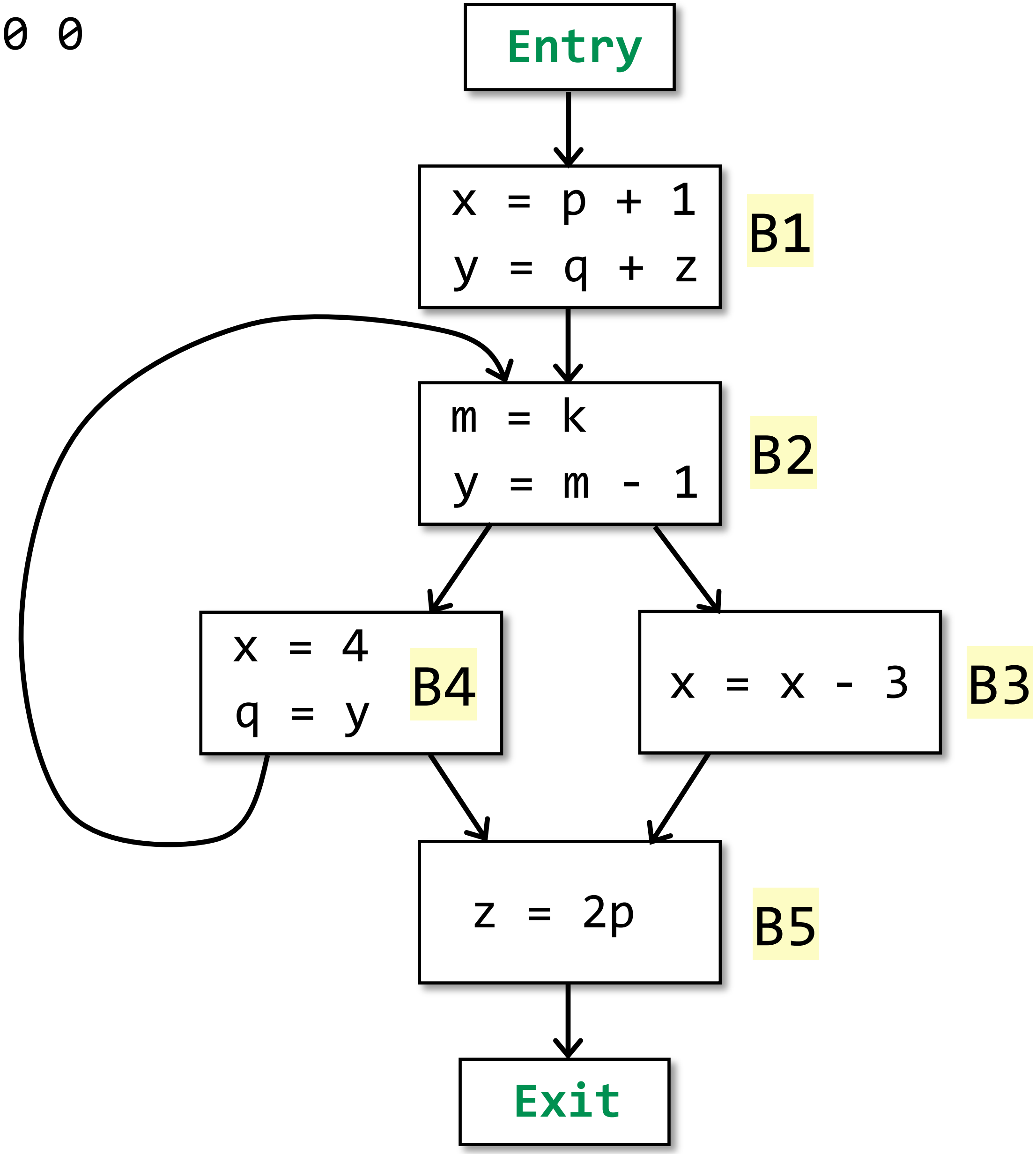
OUTPUT: IN[B] and OUT[B] for each basic block B

METHOD:

IN[*exit*] = Ø;
for (each basic block $B\backslash exit$)
    IN[B] = Ø;
while (changes to any IN occur)
    for (each basic block $B\backslash exit$) {
        $\text{OUT}[B] = \bigcup_{S \text{ a successor of } B} \text{IN}[S]$;
        $\text{IN}[B] = use_B \cup (\text{OUT}[B] - def_B)$;
    }

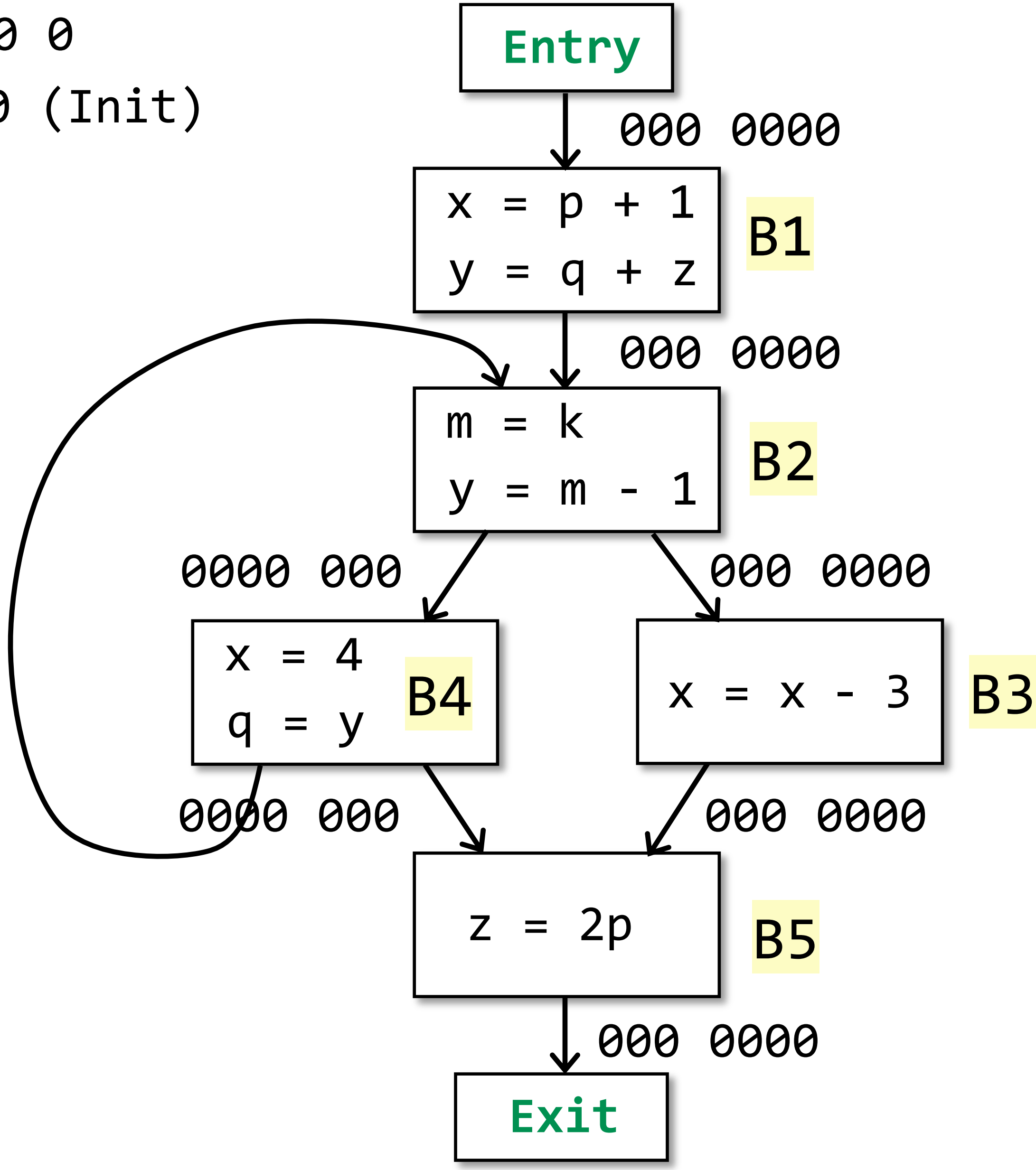
x y z p q m k

0 0 0 0 0 0 0



x	y	z	p	q	m	k
0	0	0	0	0	0	0

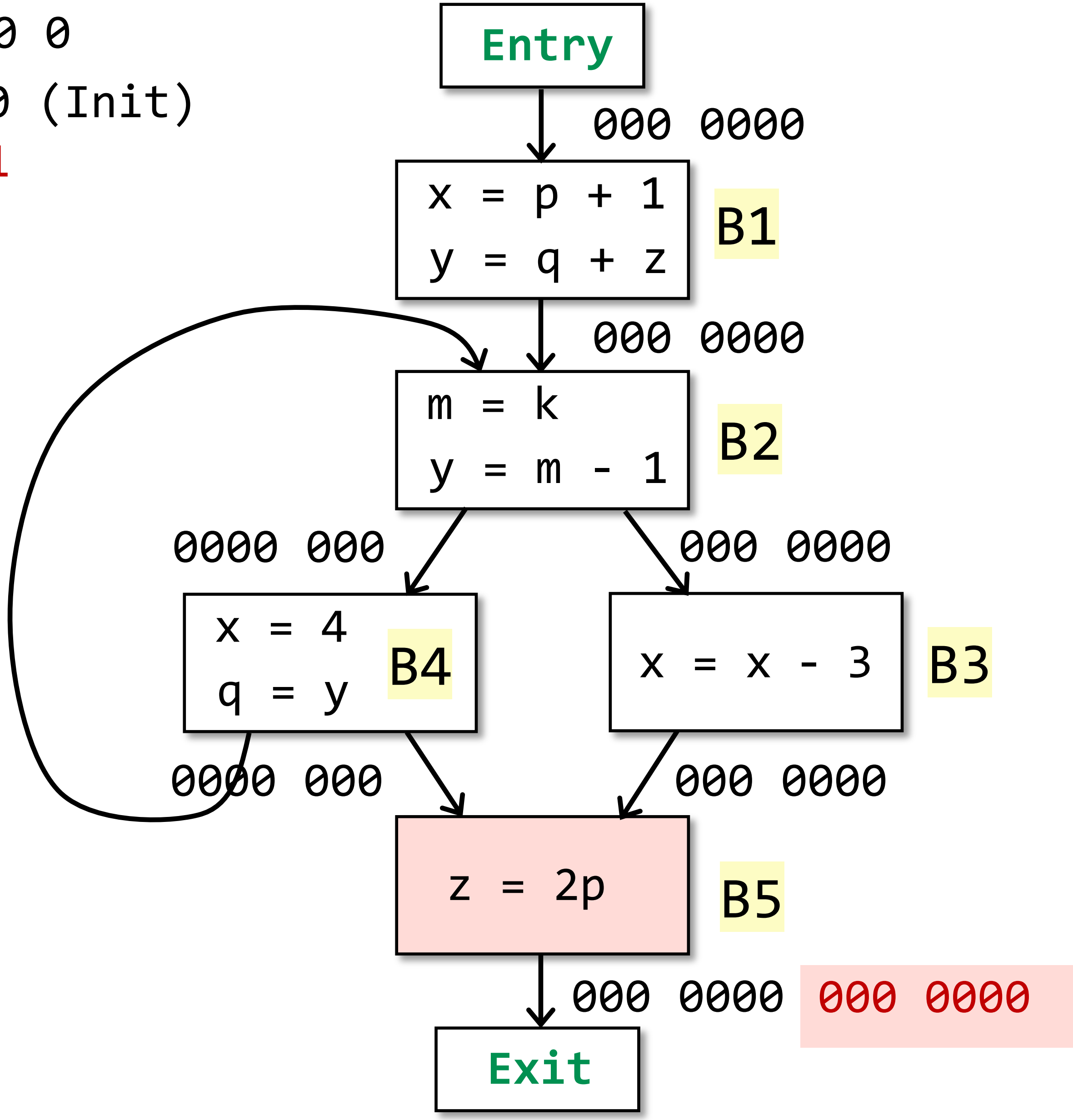
Iteration 0 (Init)



x y z p q m k
0 0 0 0 0 0 0

Iteration 0 (Init)

Iteration 1



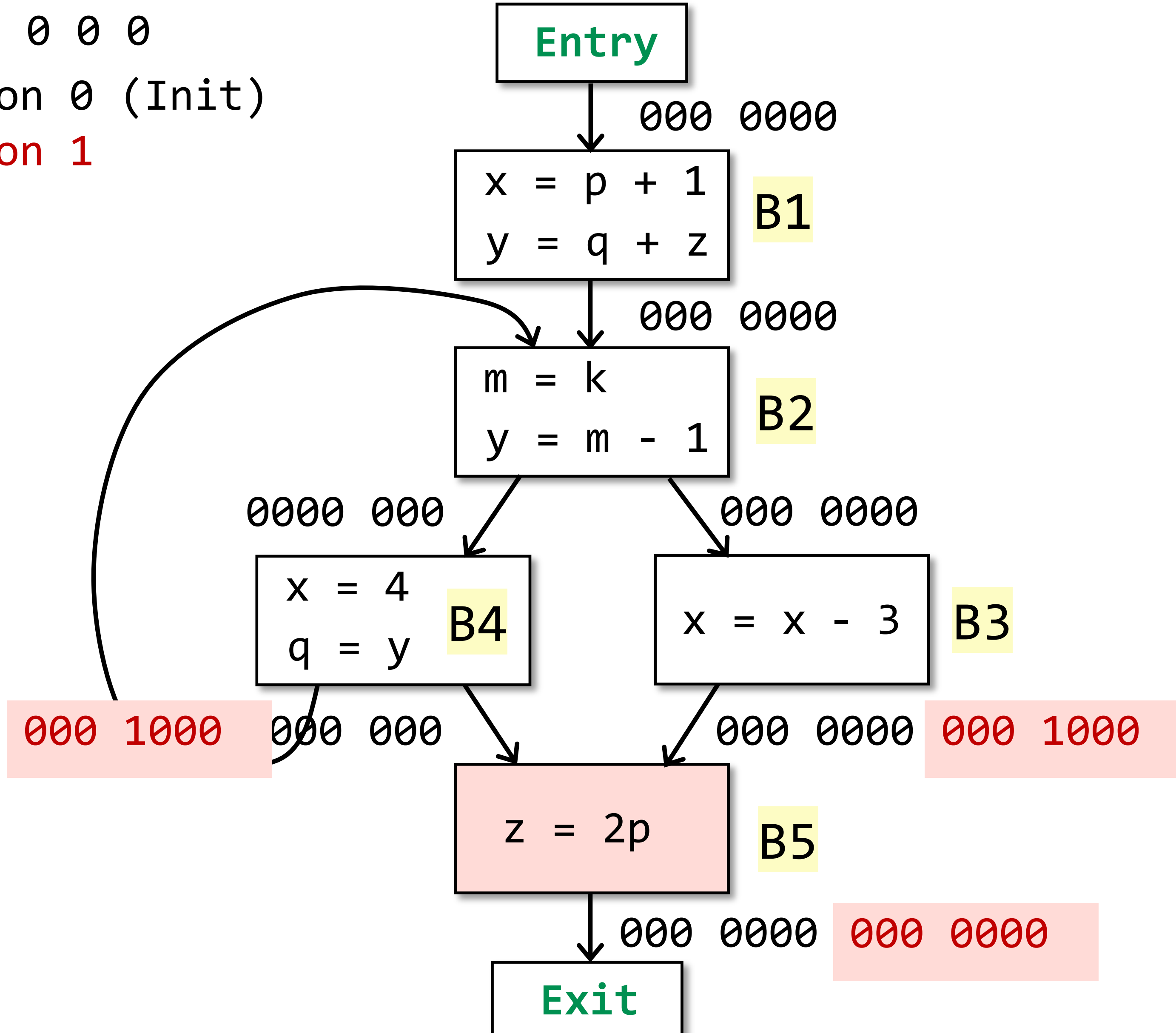
$IN[B] = use_B \cup (OUT[B] - def_B)$

x y z p q m k

0 0 0 0 0 0 0

Iteration 0 (Init)

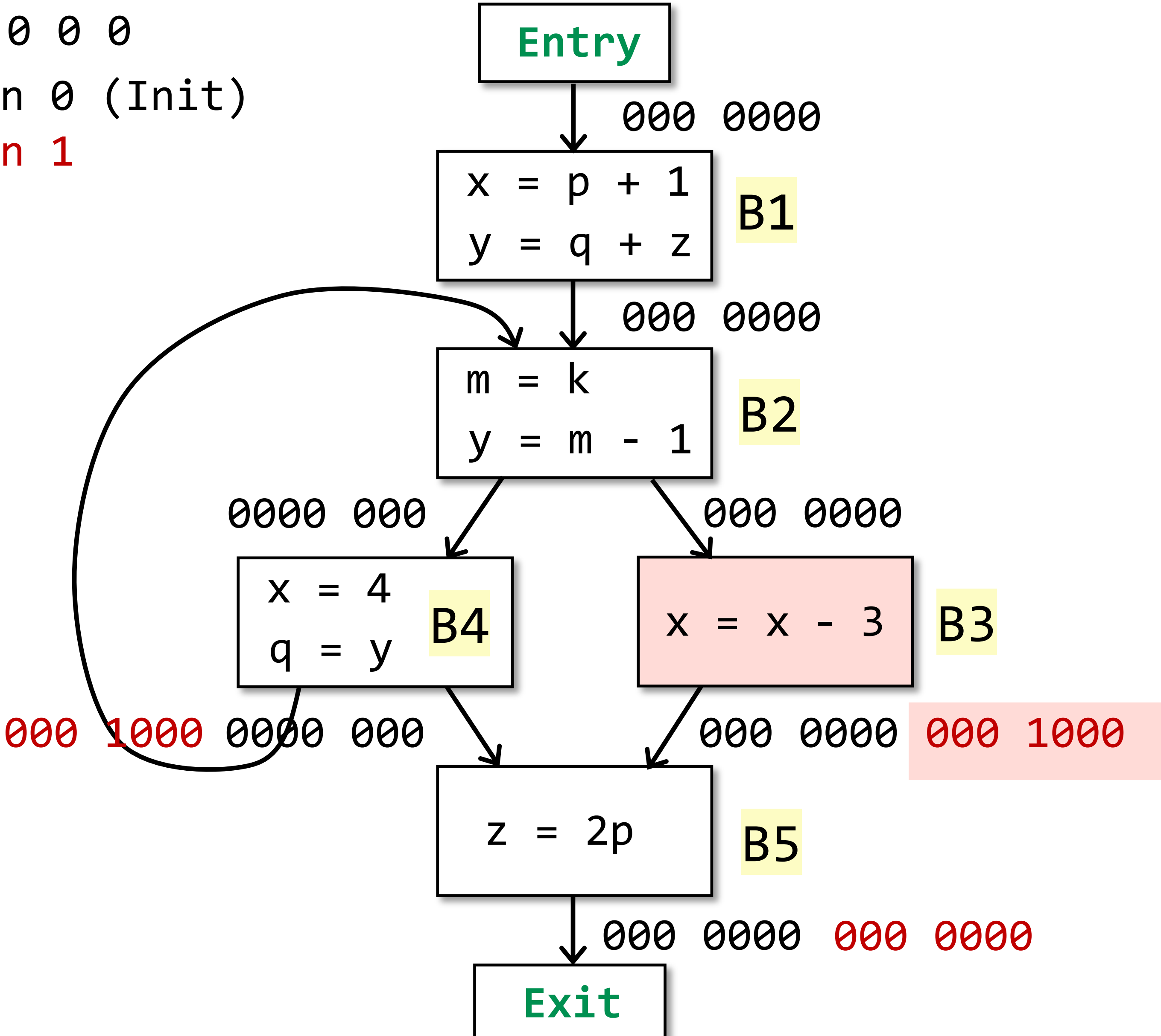
Iteration 1



x y z p q m k
0 0 0 0 0 0 0

Iteration 0 (Init)

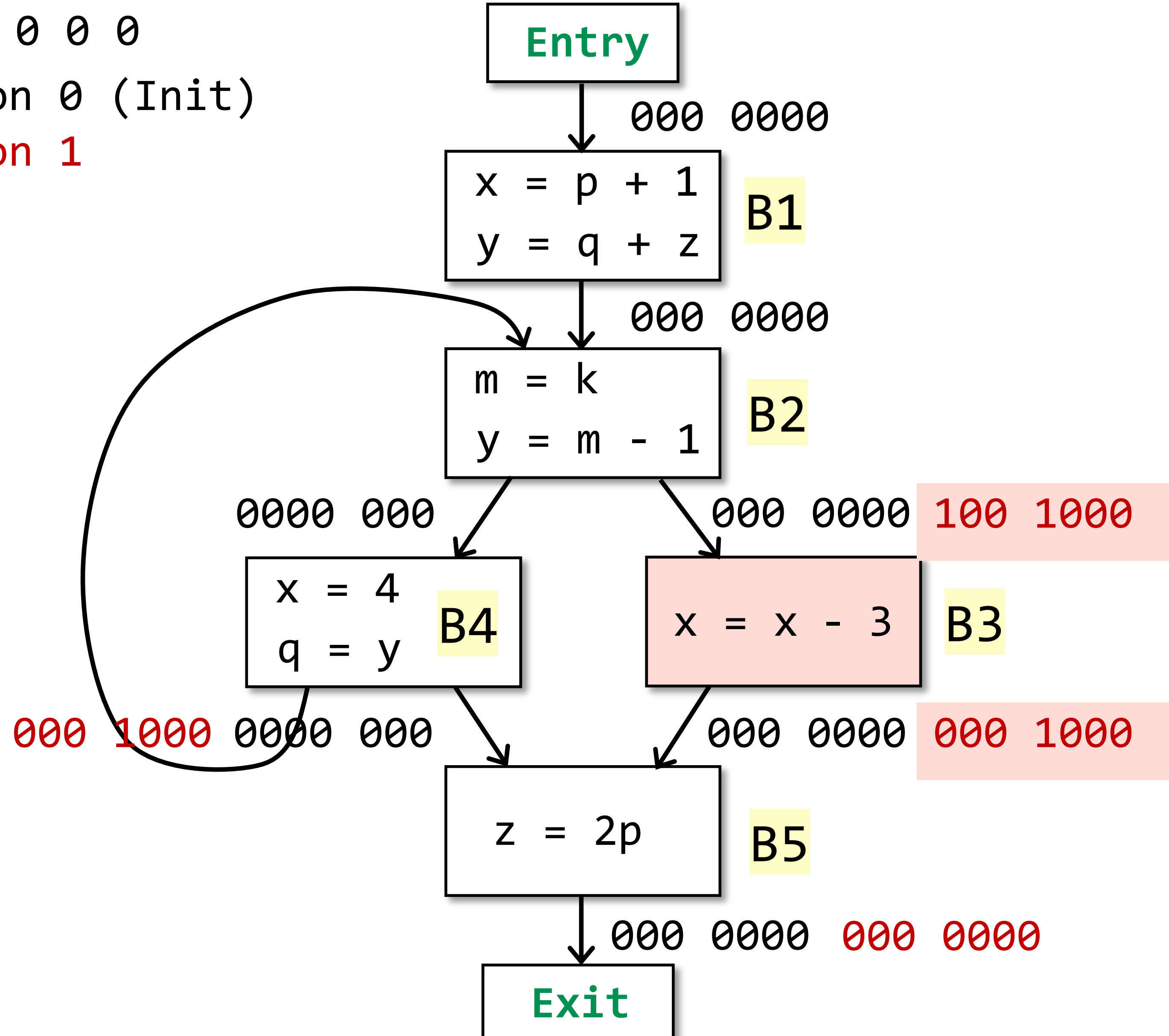
Iteration 1



x y z p q m k
0 0 0 0 0 0 0

Iteration 0 (Init)

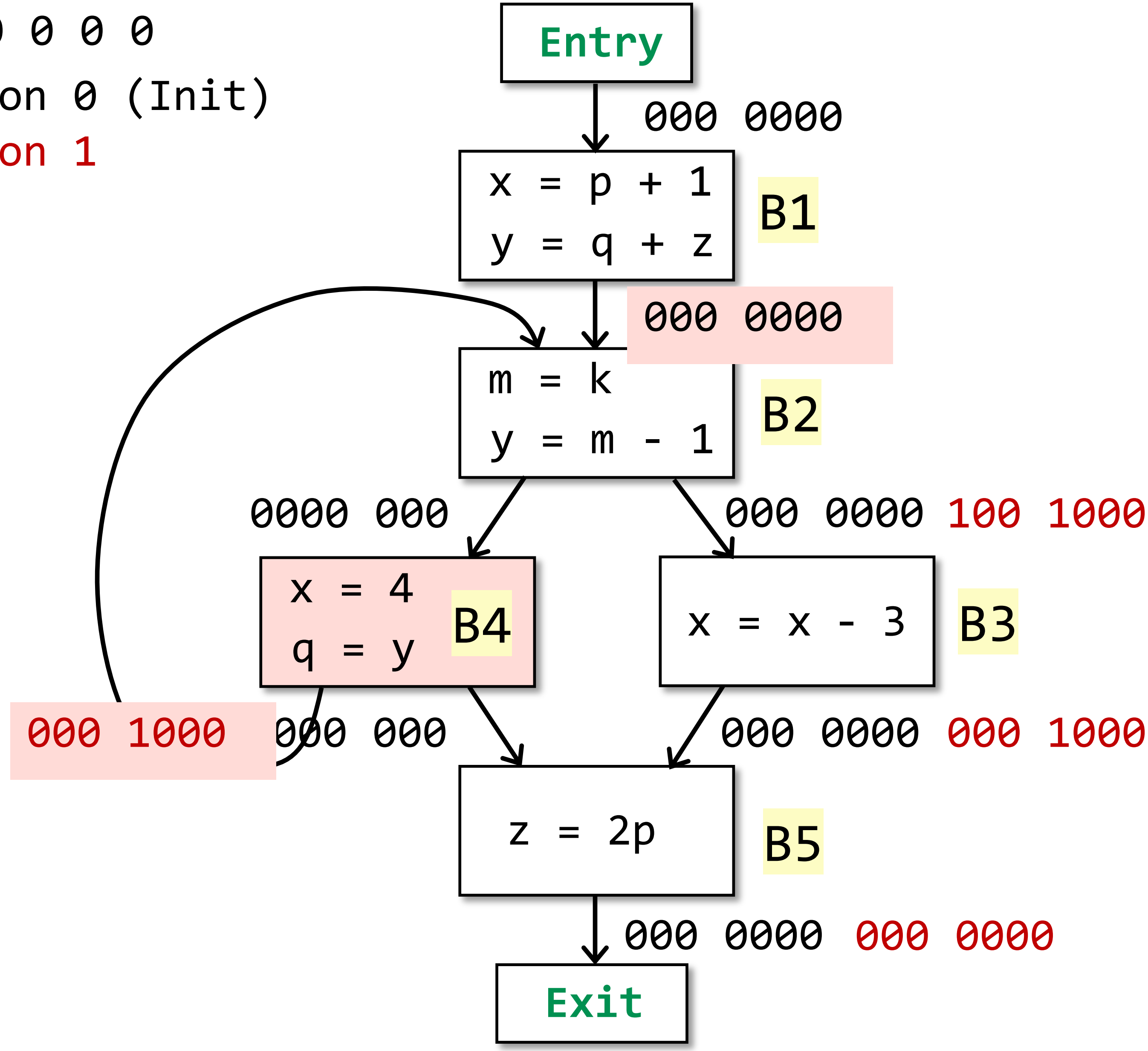
Iteration 1



x y z p q m k
0 0 0 0 0 0 0

Iteration 0 (Init)

Iteration 1



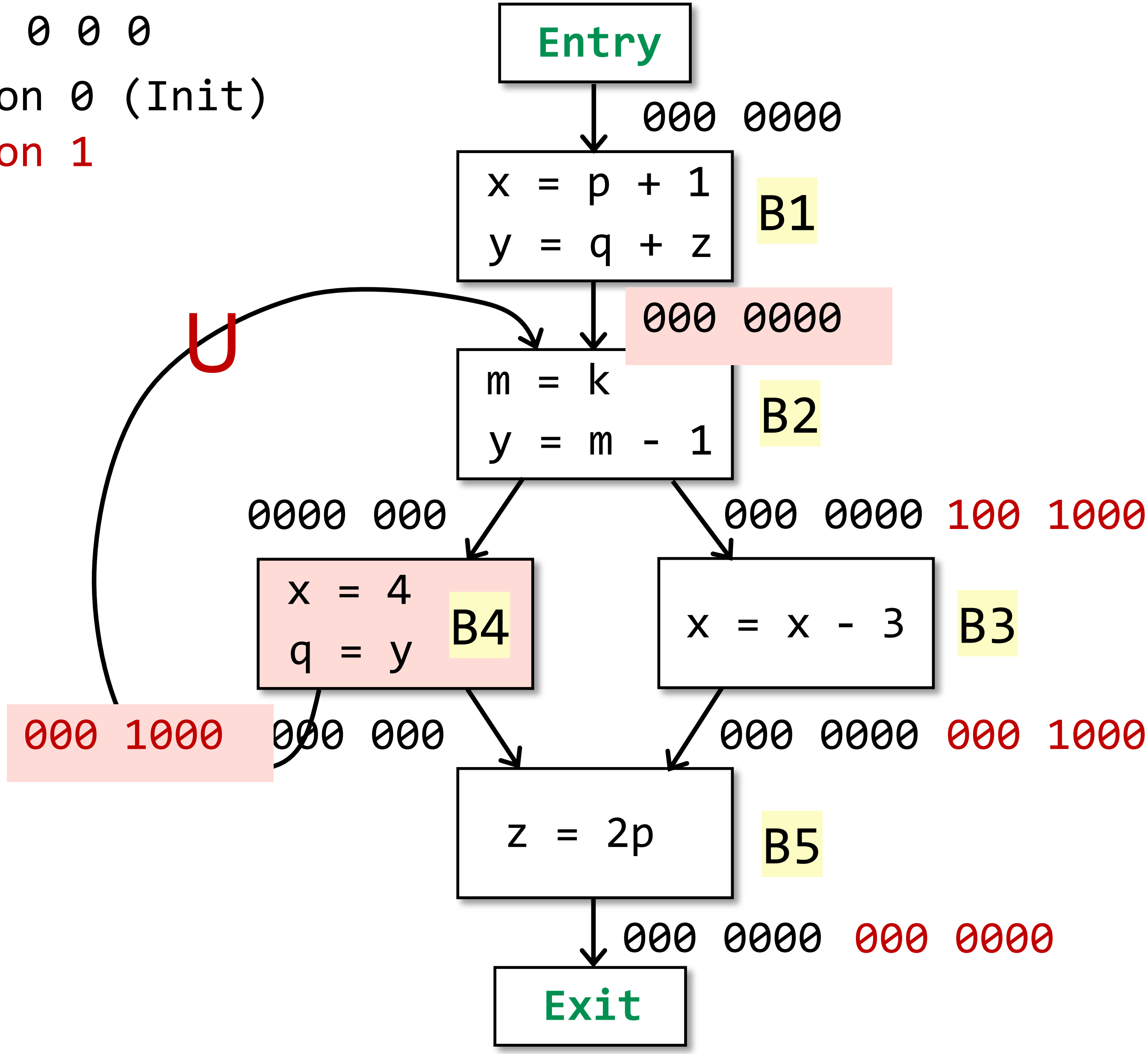
$$OUT[B] = \bigcup_{S \text{ a successor of } B} IN[B]$$

x y z p q m k

0 0 0 0 0 0 0

Iteration 0 (Init)

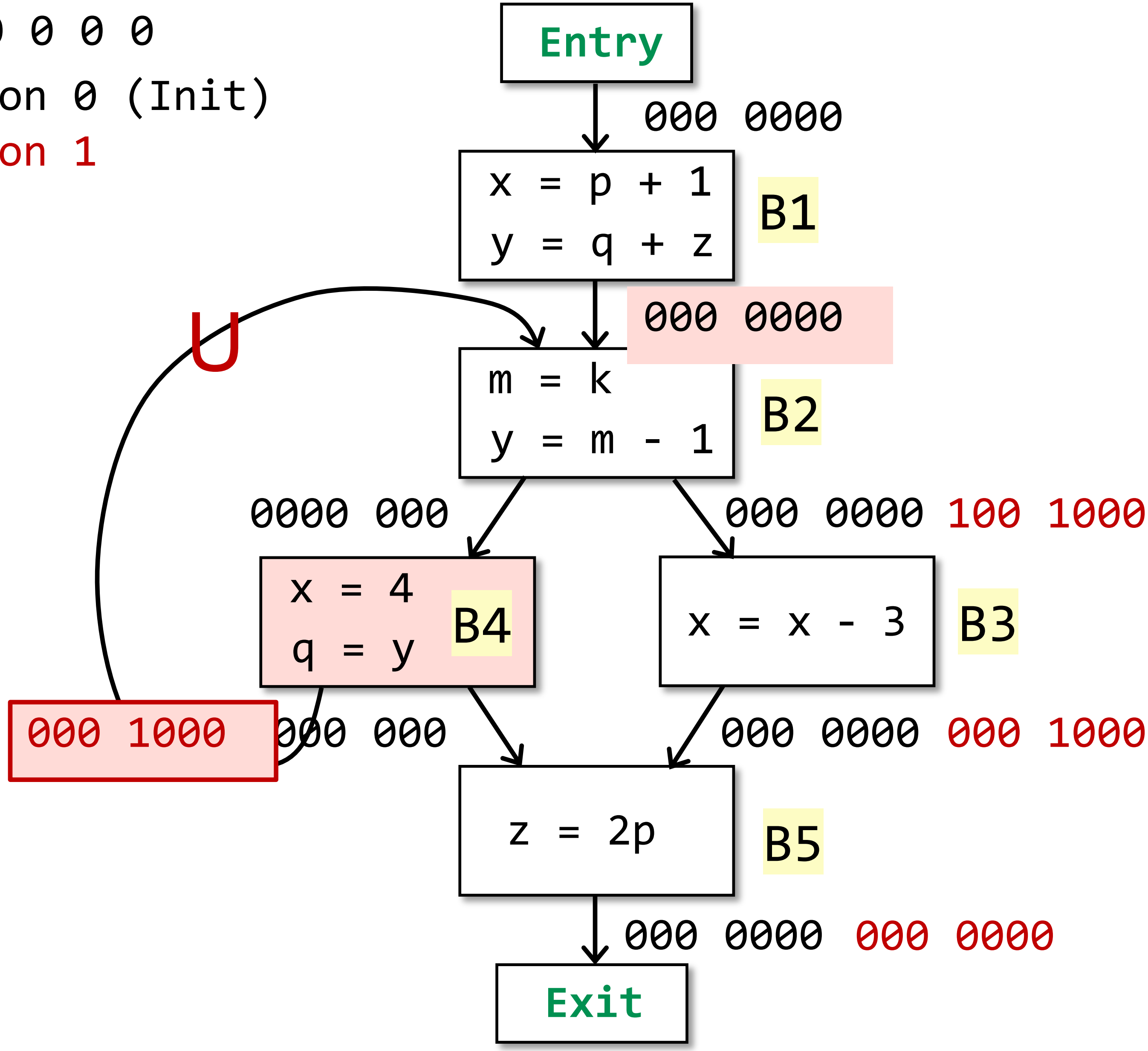
Iteration 1



x	y	z	p	q	m	k
0	0	0	0	0	0	0

Iteration 0 (Init)

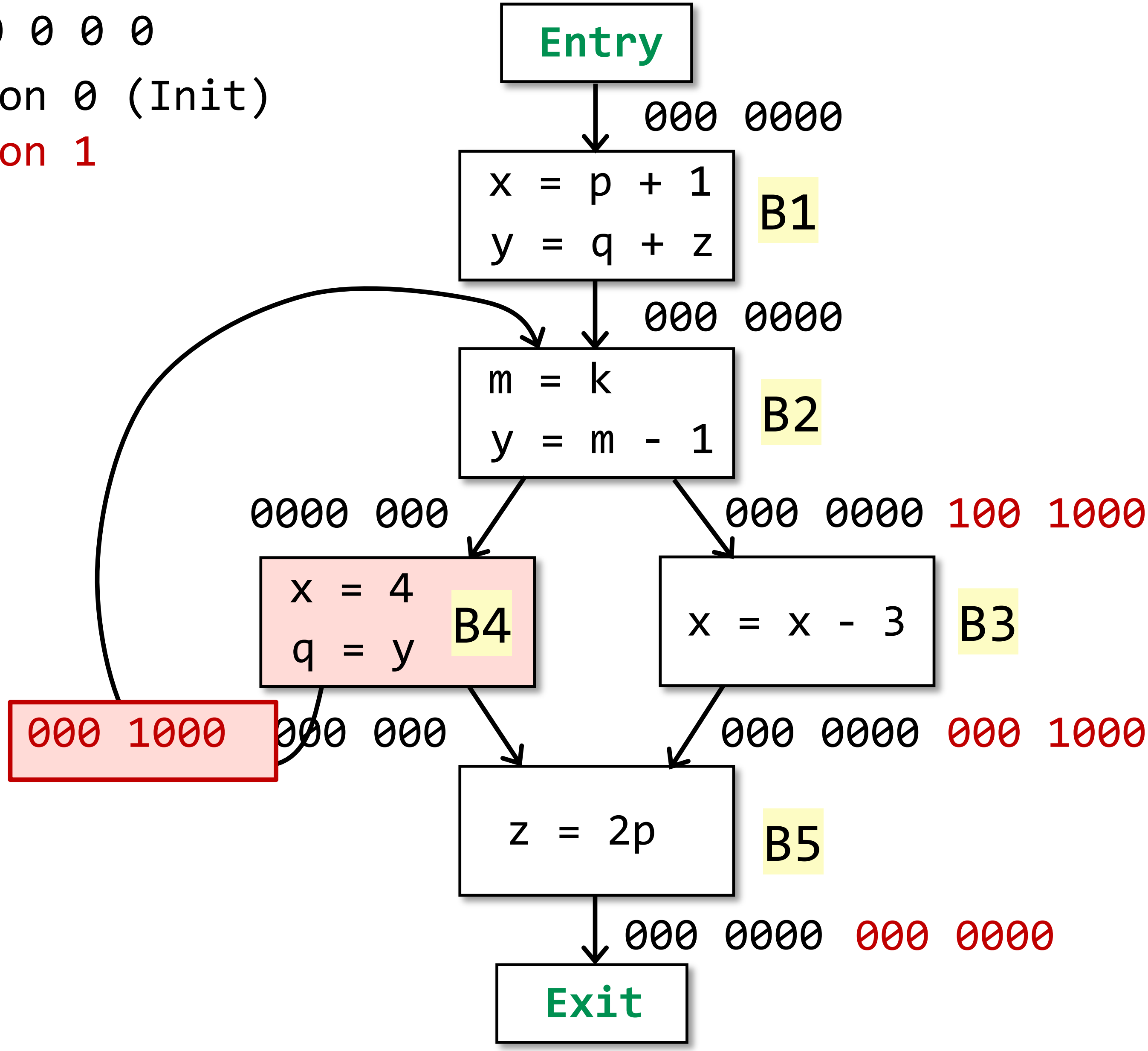
Iteration 1



x y z p q m k
0 0 0 0 0 0 0

Iteration 0 (Init)

Iteration 1

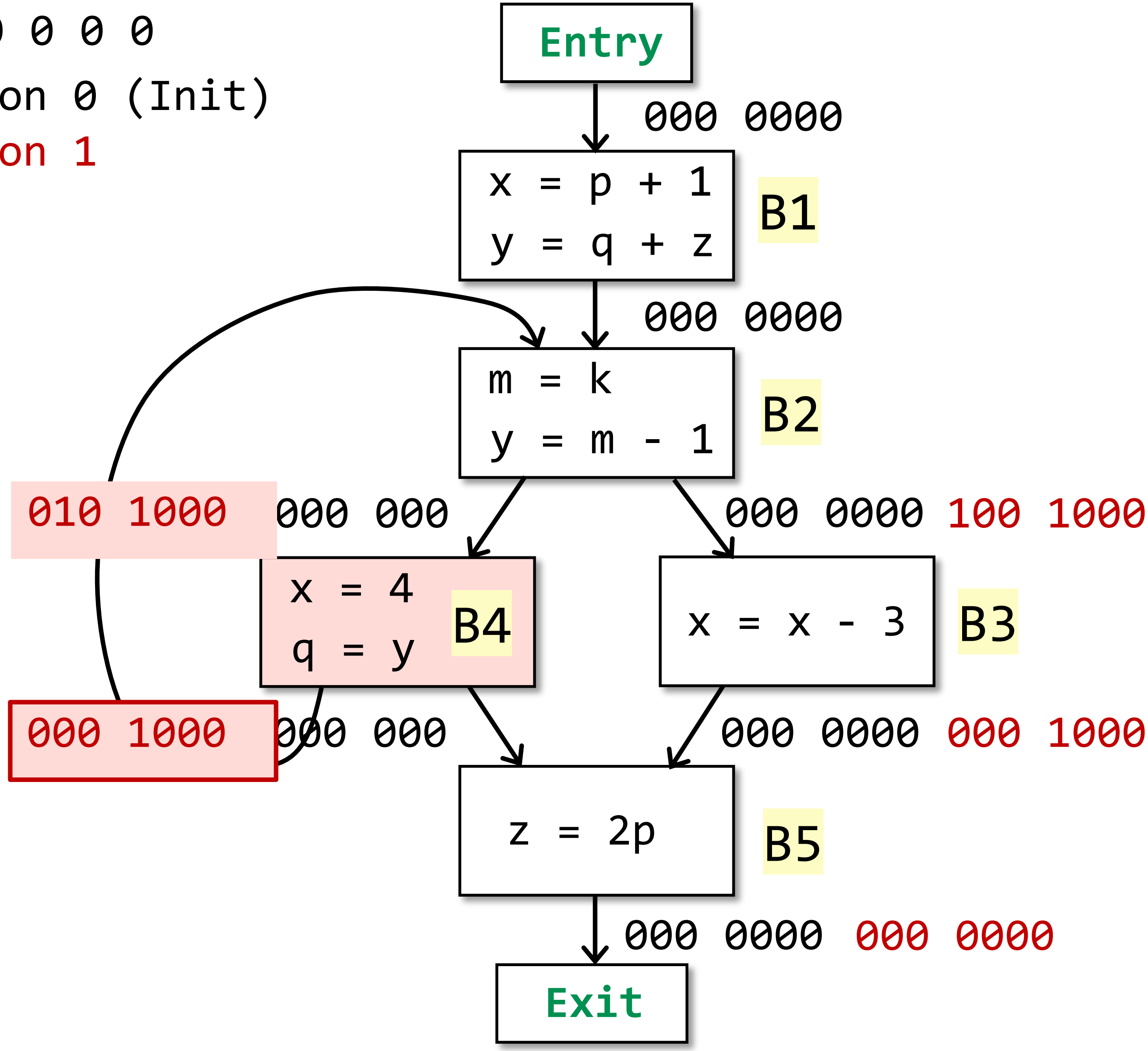


x y z p q m k

0 0 0 0 0 0 0

Iteration 0 (Init)

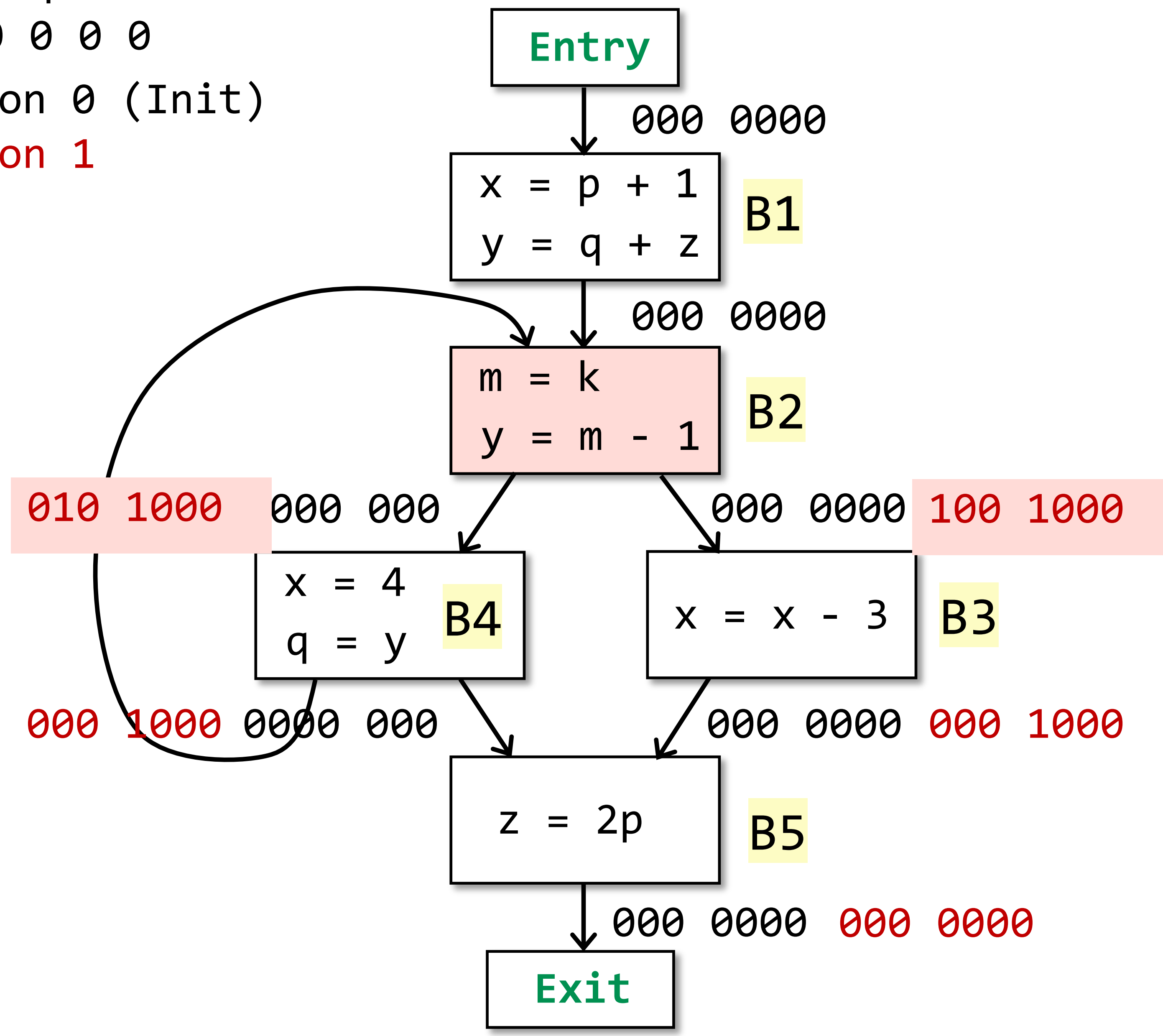
Iteration 1



x y z p q m k
0 0 0 0 0 0 0

Iteration 0 (Init)

Iteration 1

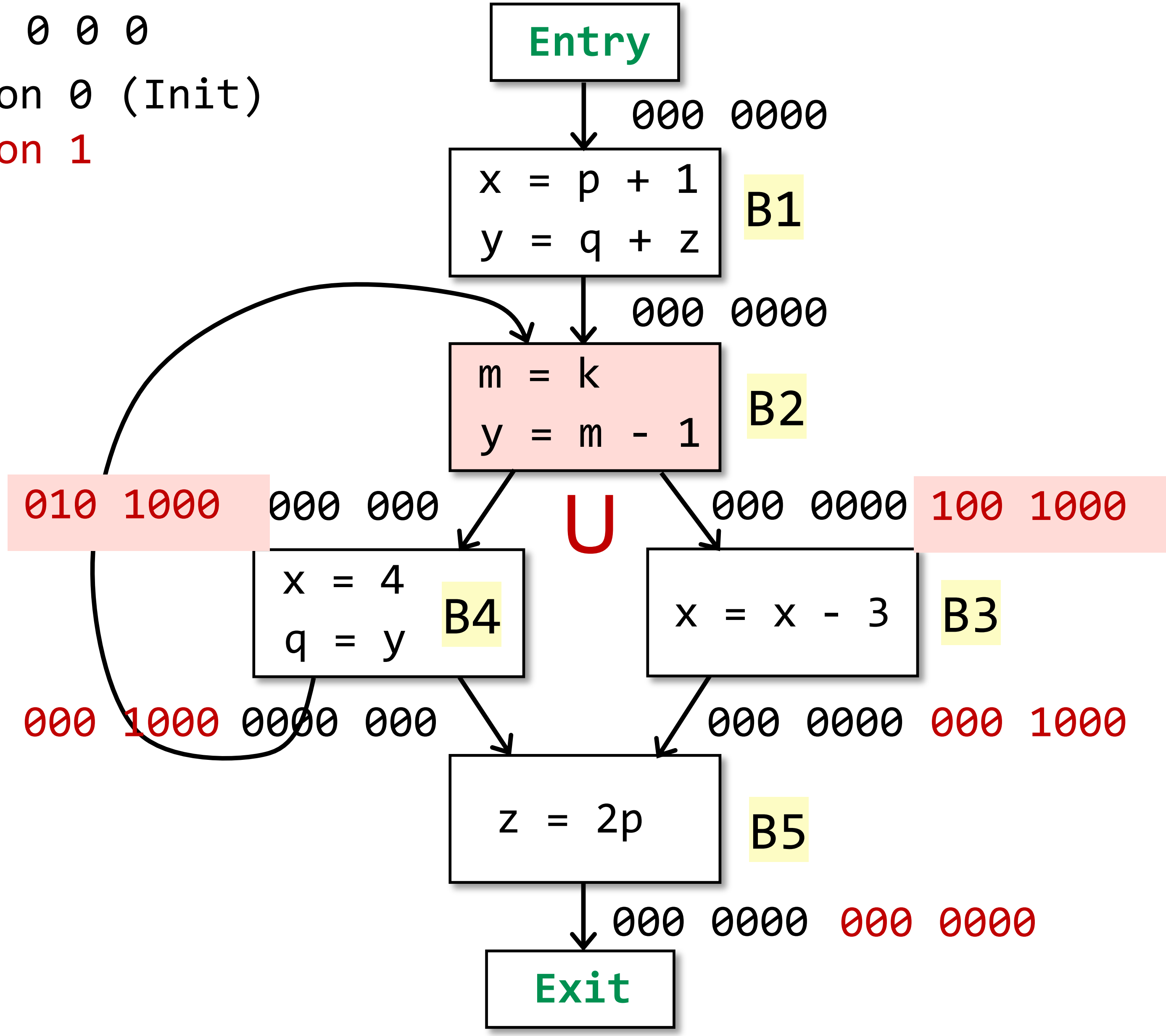


x y z p q m k

0 0 0 0 0 0 0

Iteration 0 (Init)

Iteration 1

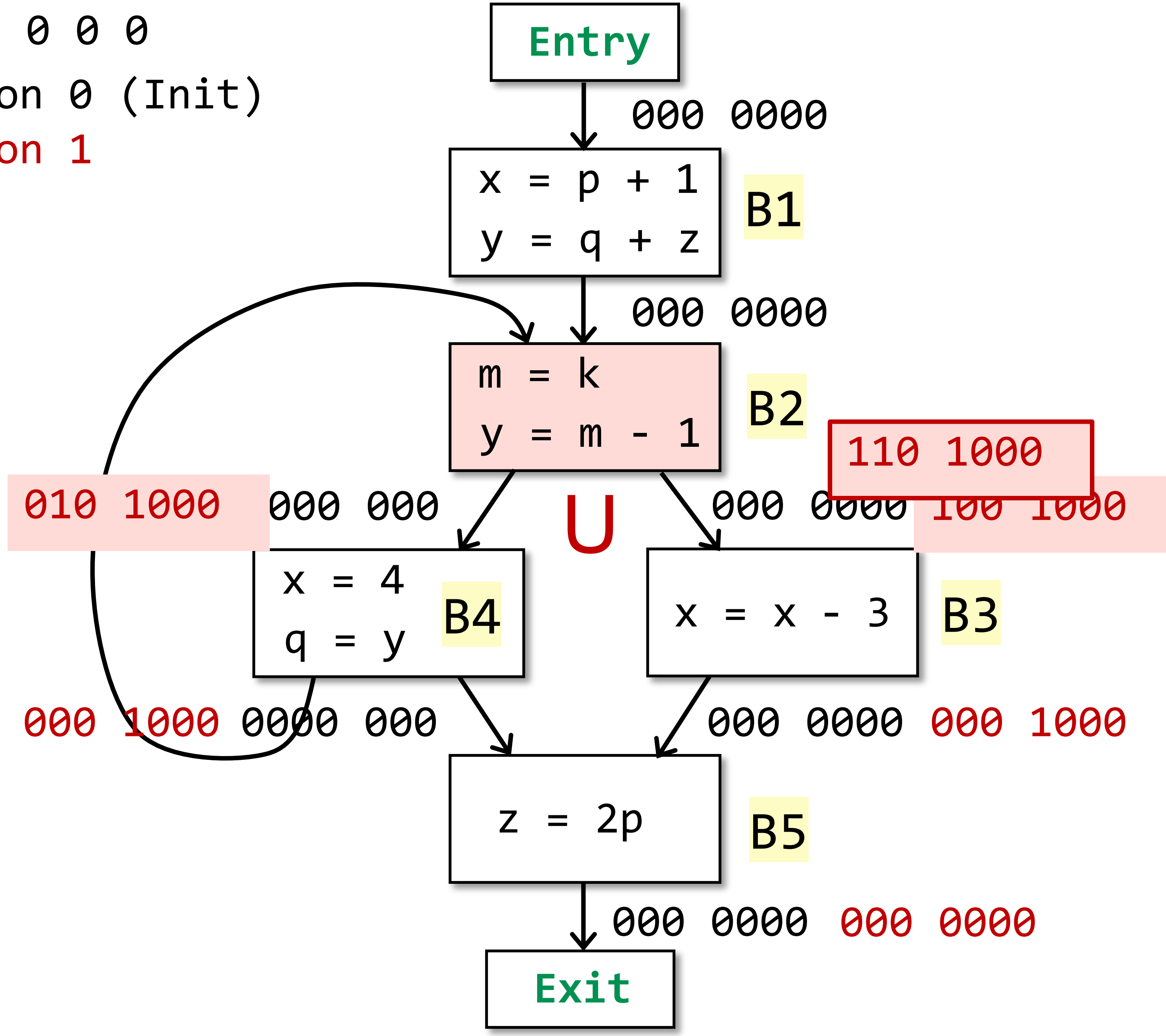


x y z p q m k

0 0 0 0 0 0 0

Iteration 0 (Init)

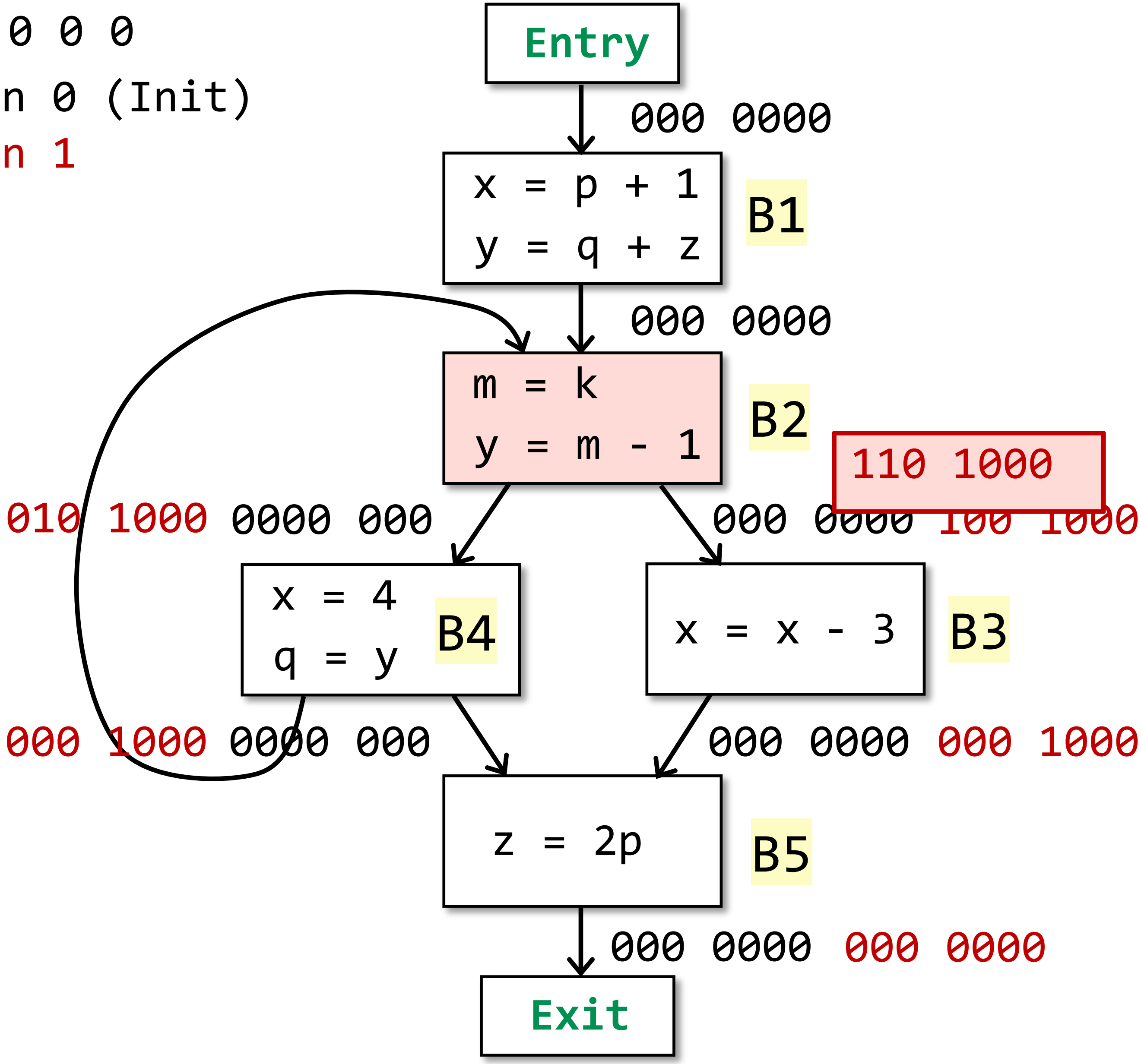
Iteration 1



x y z p q m k
0 0 0 0 0 0 0

Iteration 0 (Init)

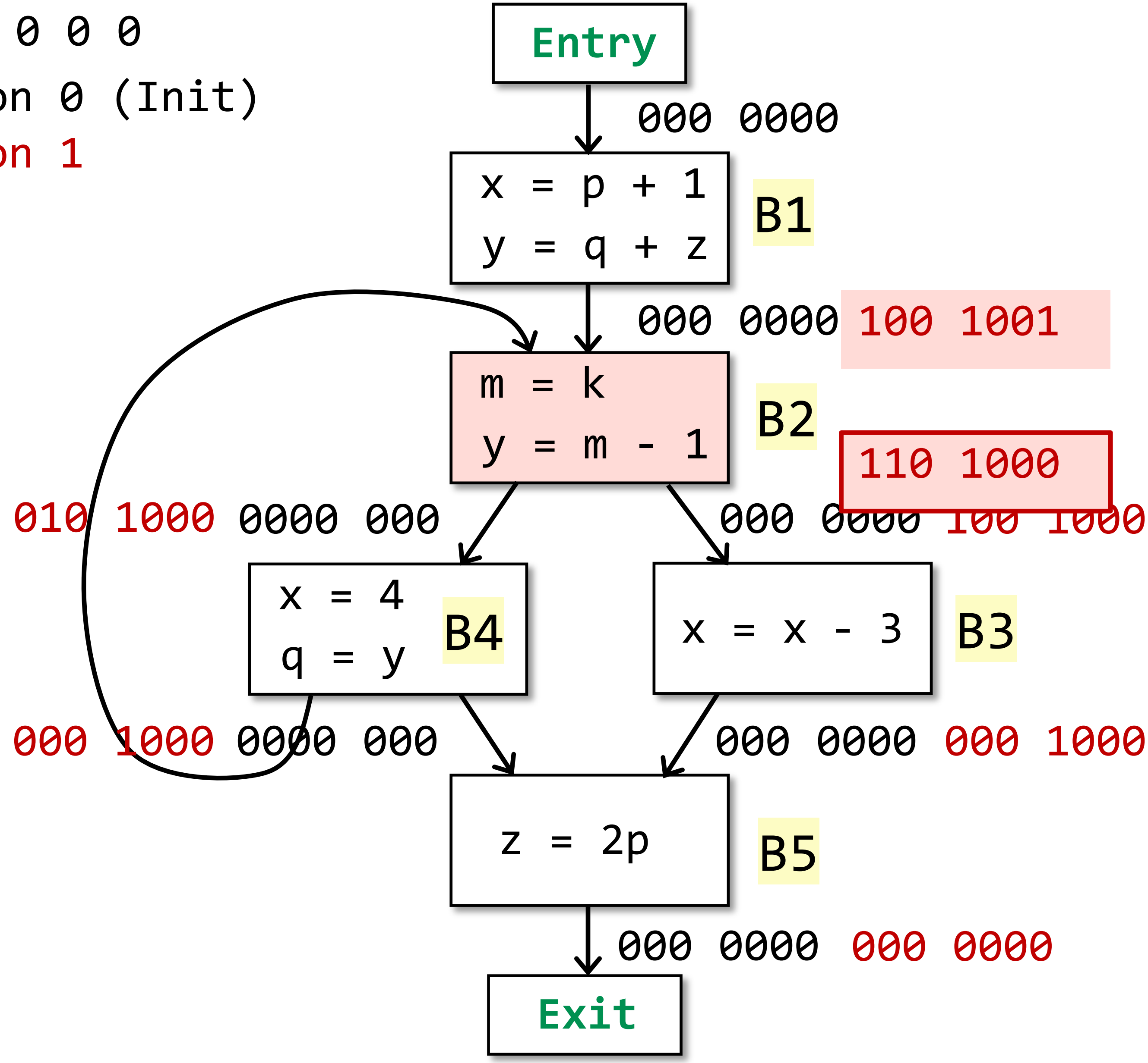
Iteration 1



x y z p q m k
0 0 0 0 0 0 0

Iteration 0 (Init)

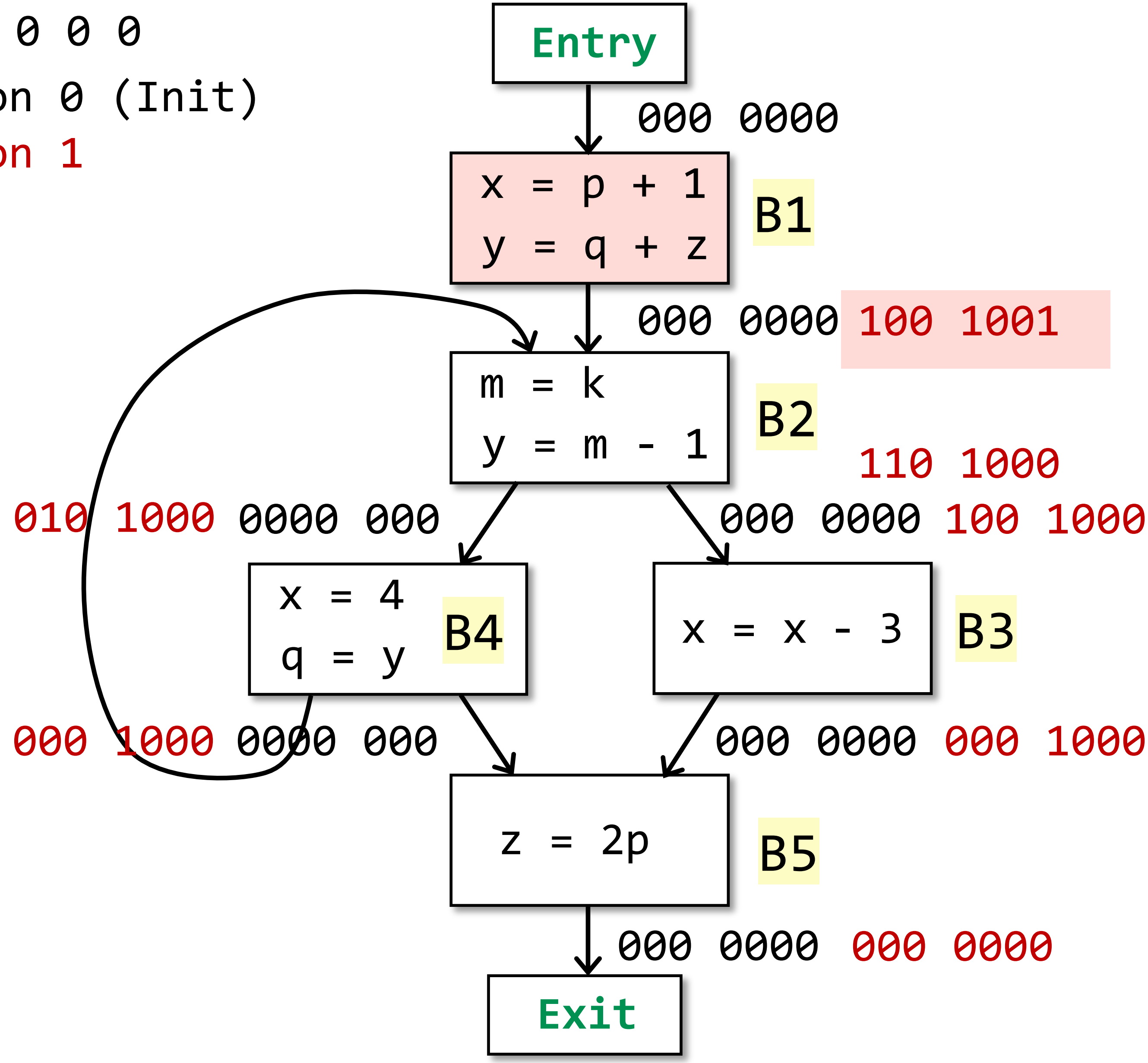
Iteration 1



x y z p q m k
0 0 0 0 0 0 0

Iteration 0 (Init)

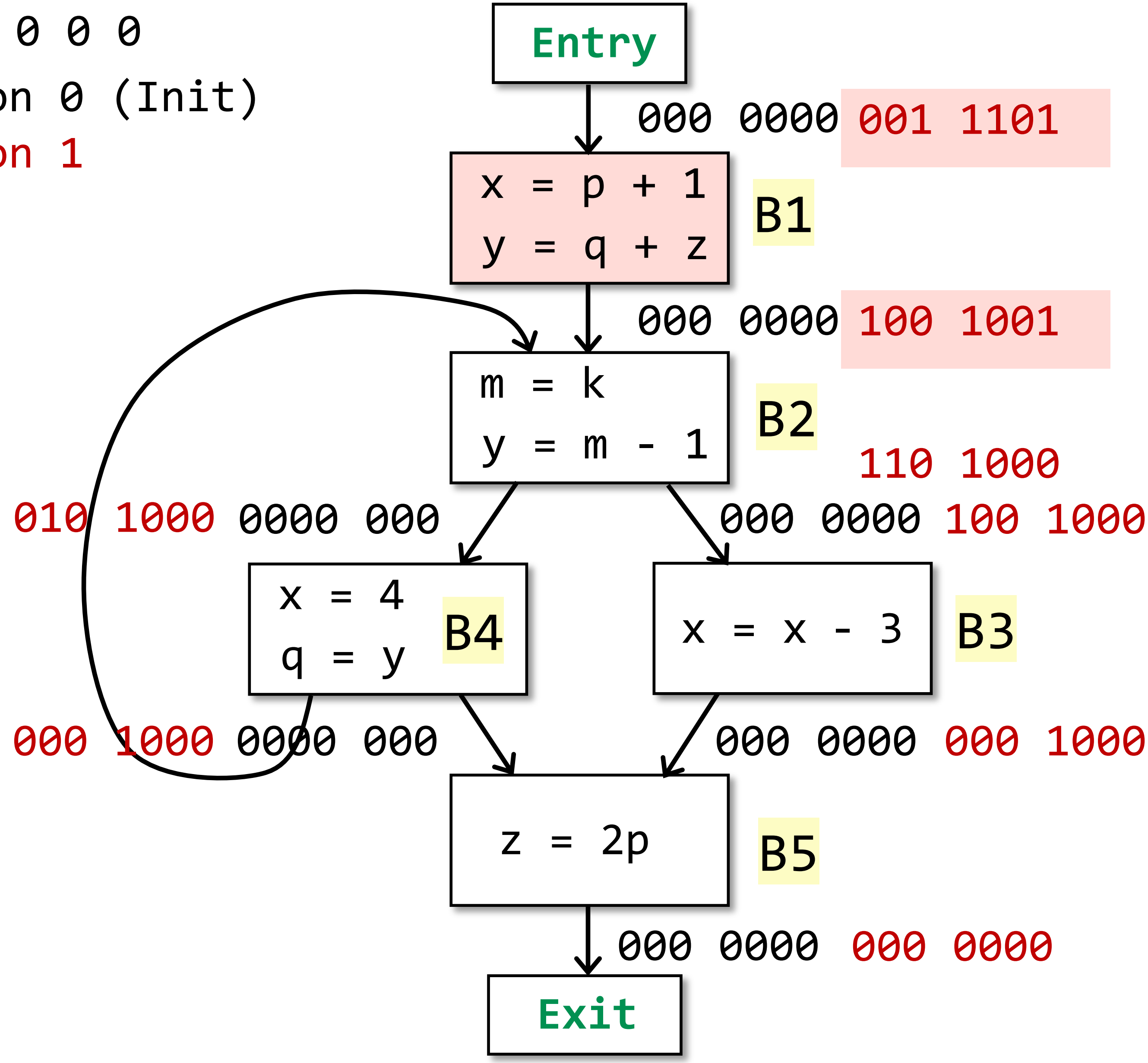
Iteration 1



x y z p q m k
0 0 0 0 0 0 0

Iteration 0 (Init)

Iteration 1

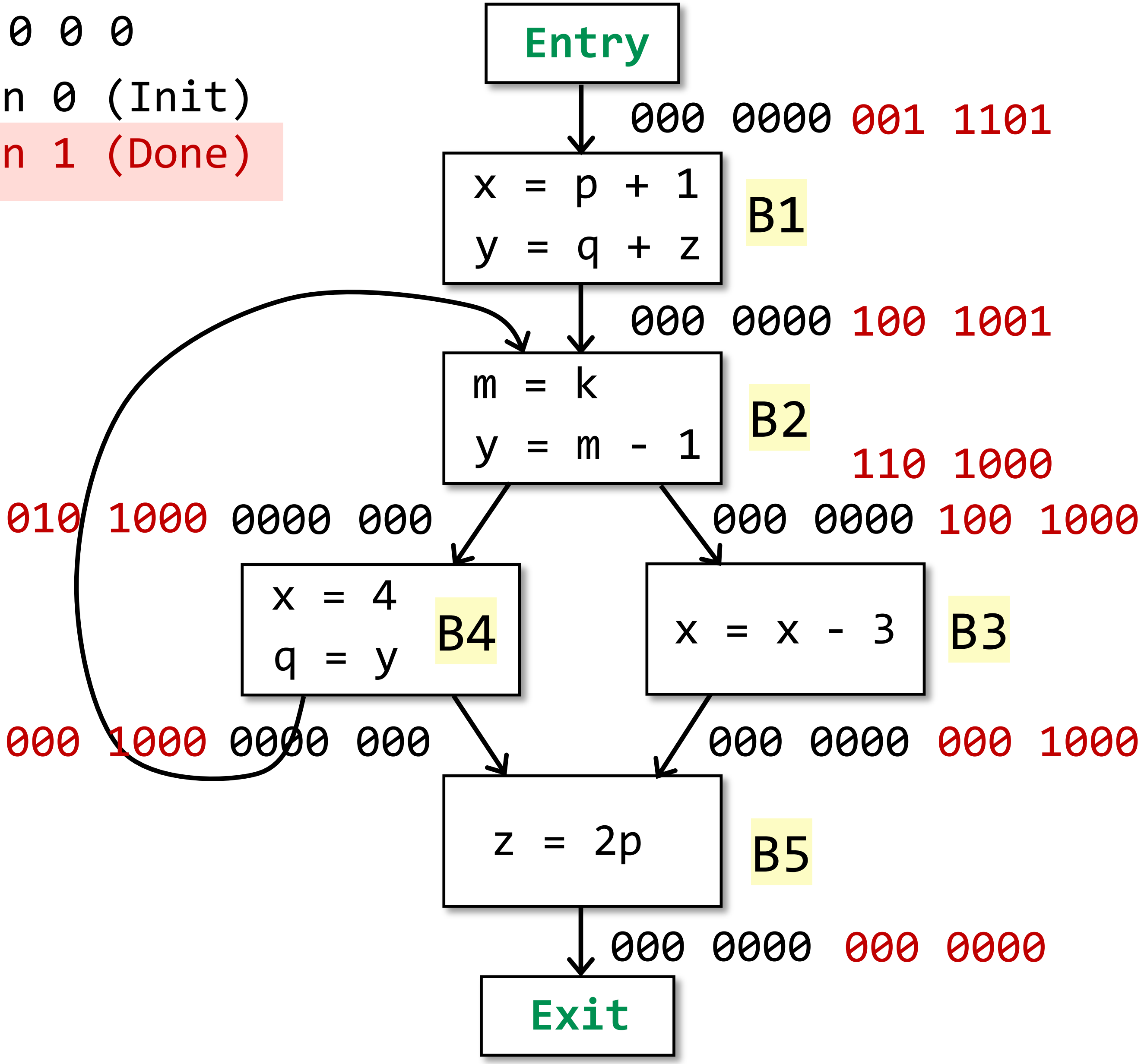


x y z p q m k

0 0 0 0 0 0 0

Iteration 0 (Init)

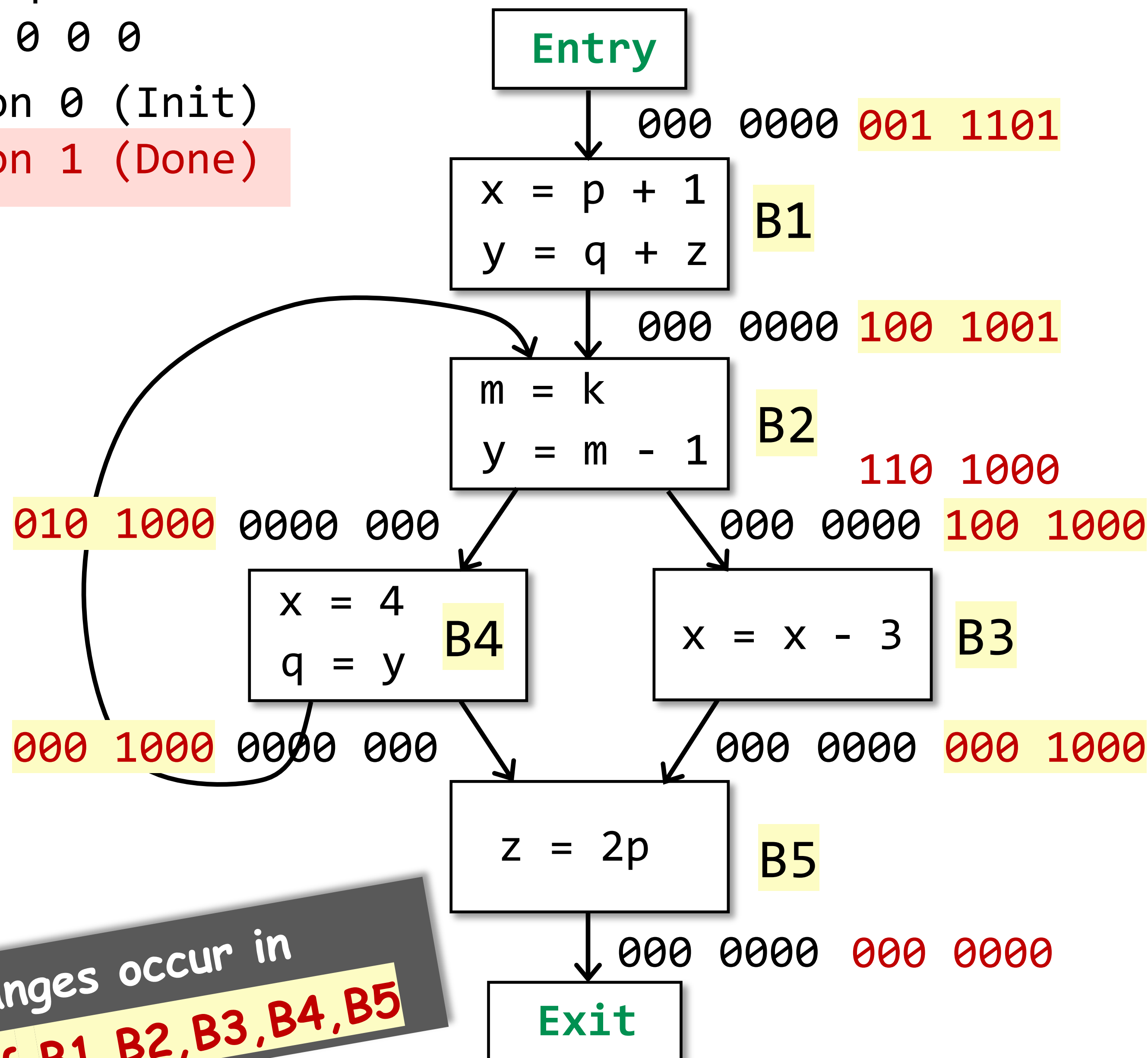
Iteration 1 (Done)



x y z p q m k
0 0 0 0 0 0 0

Iteration 0 (Init)

Iteration 1 (Done)



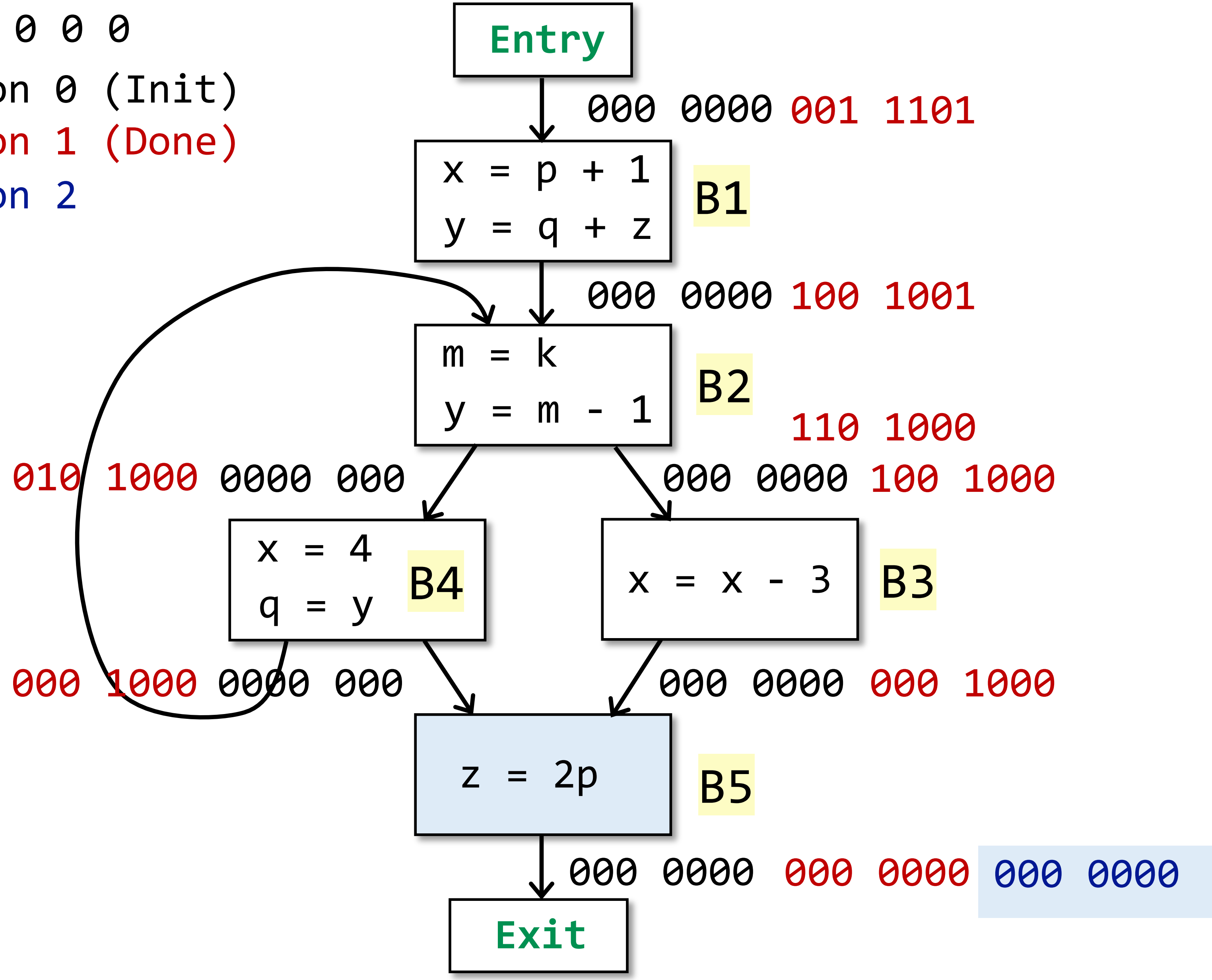
Changes occur in
IN[] of B1,B2,B3,B4,B5

x y z p q m k
0 0 0 0 0 0 0

Iteration 0 (Init)

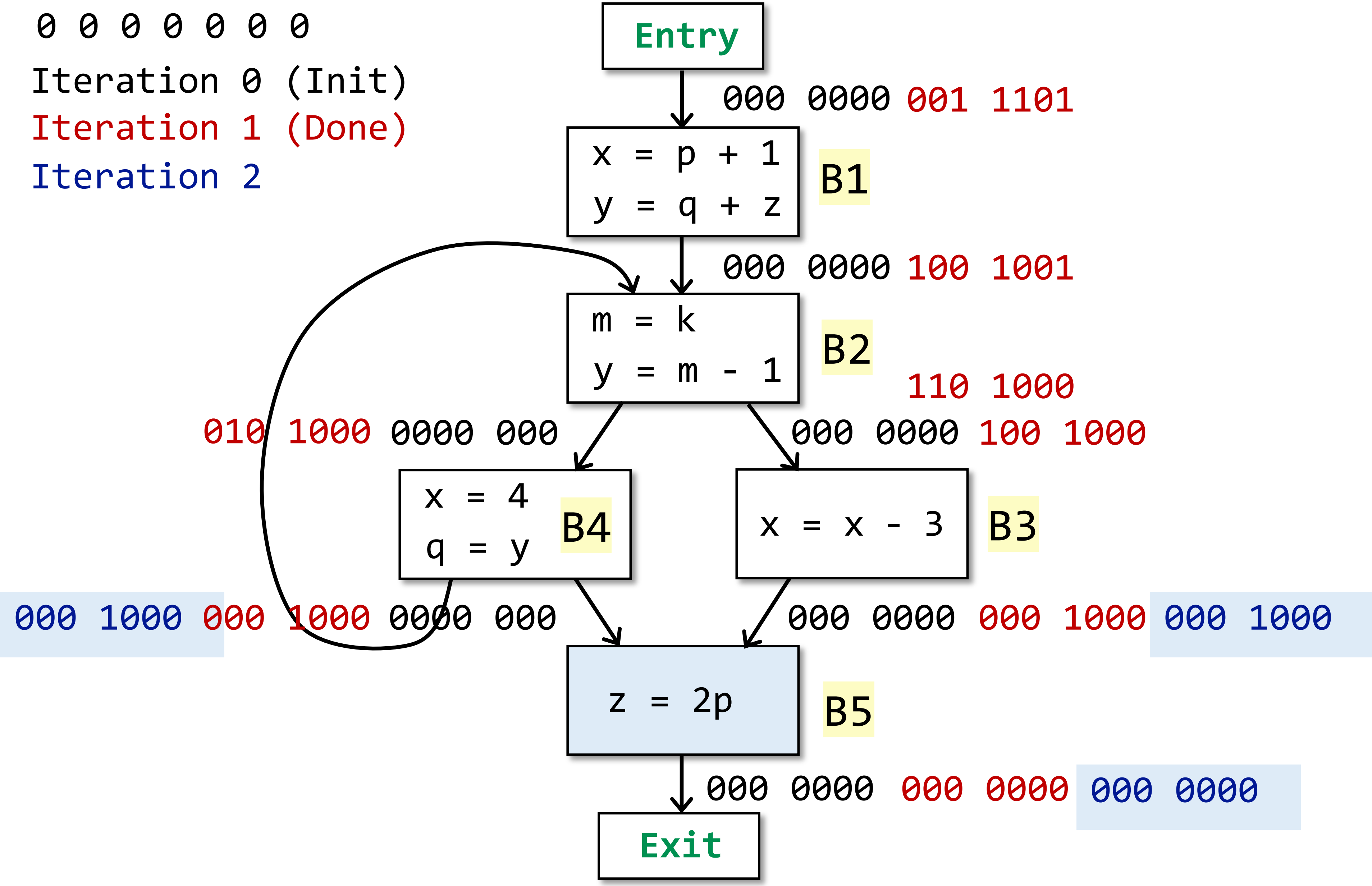
Iteration 1 (Done)

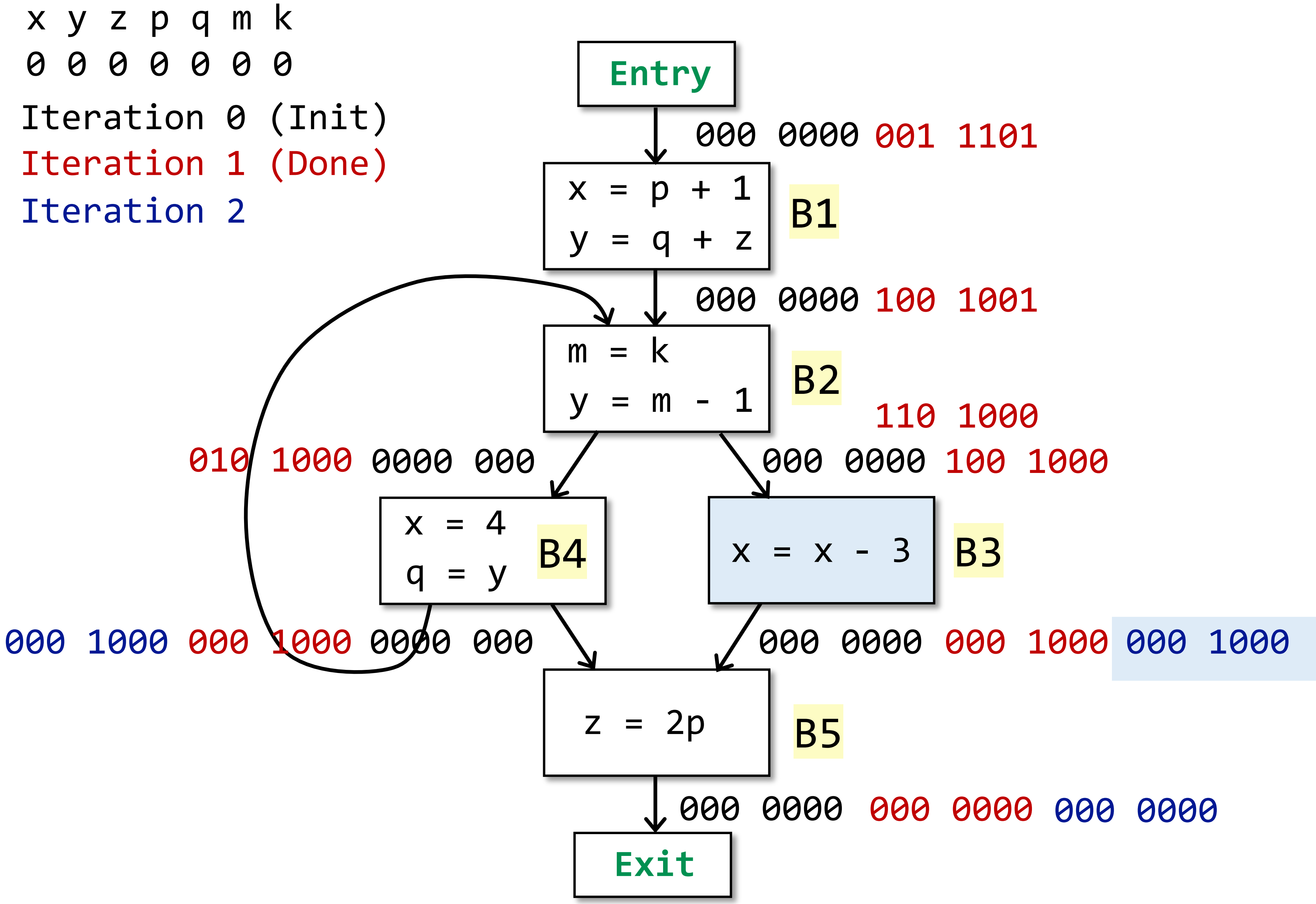
Iteration 2

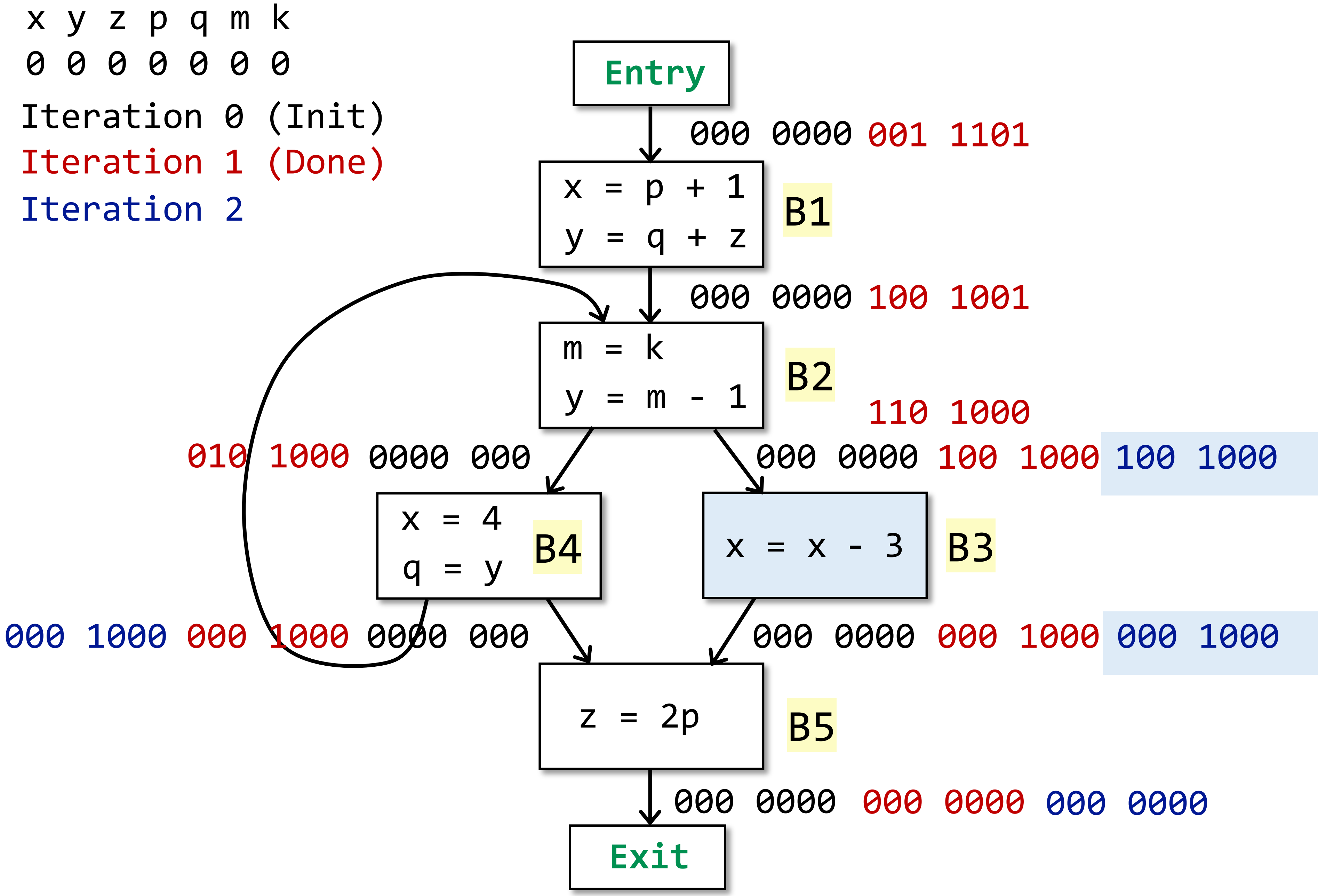


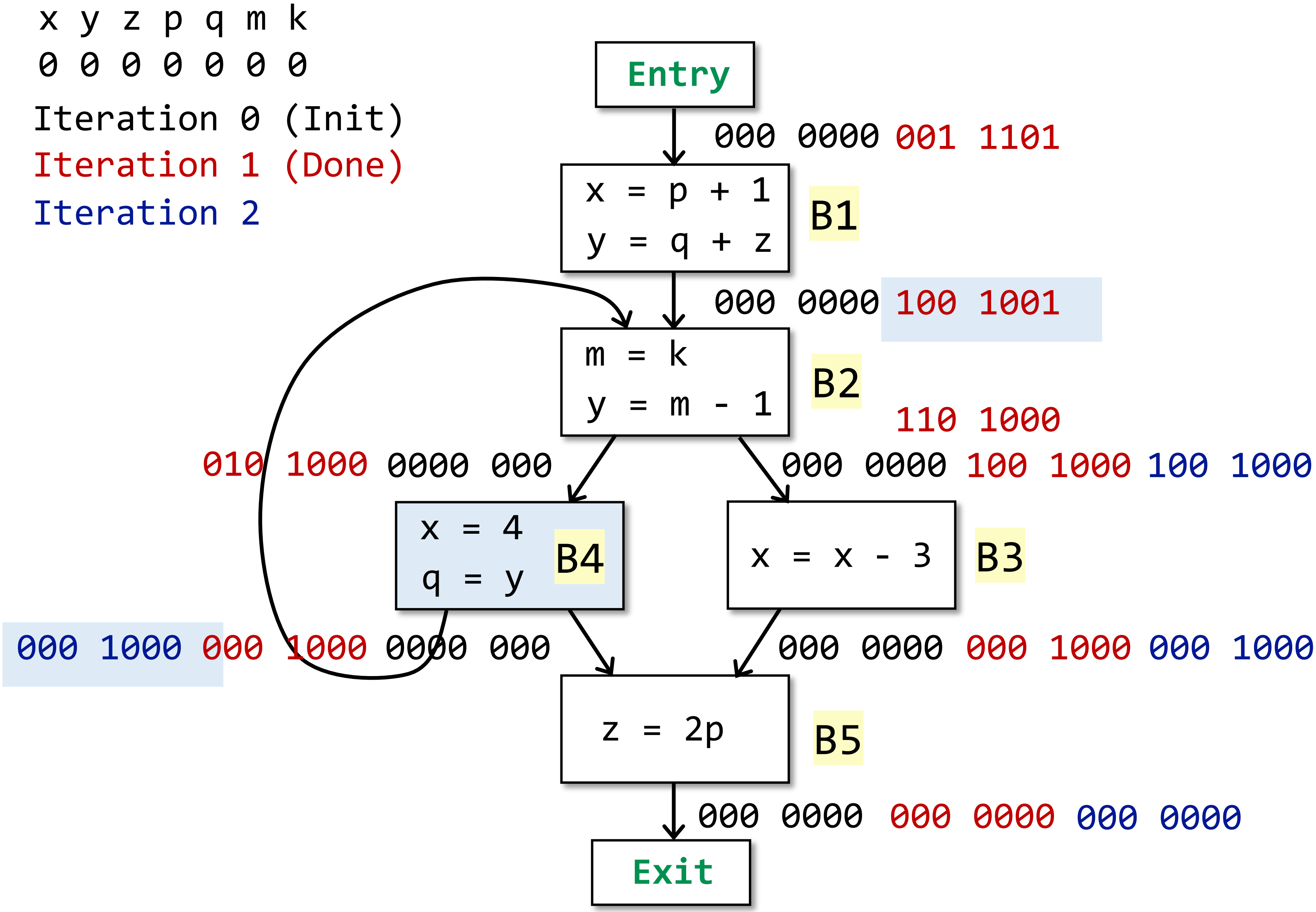
x y z p q m k
0 0 0 0 0 0 0

Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2




x y z p q m k
0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2
Entry
000 0000 001 1101
x = p + 1
y = q + z
B1
000 0000 100 1001
m = k
y = m - 1
B2
110 1000
010 1000 0000 000
000 0000 100 1000
x = 4
q = y
B4
x = x - 3
B3
000 1000 000 1000 0000 000
000 0000 000 1000 000 1000
z = 2p
B5
000 0000 000 0000 000 0000
Exit

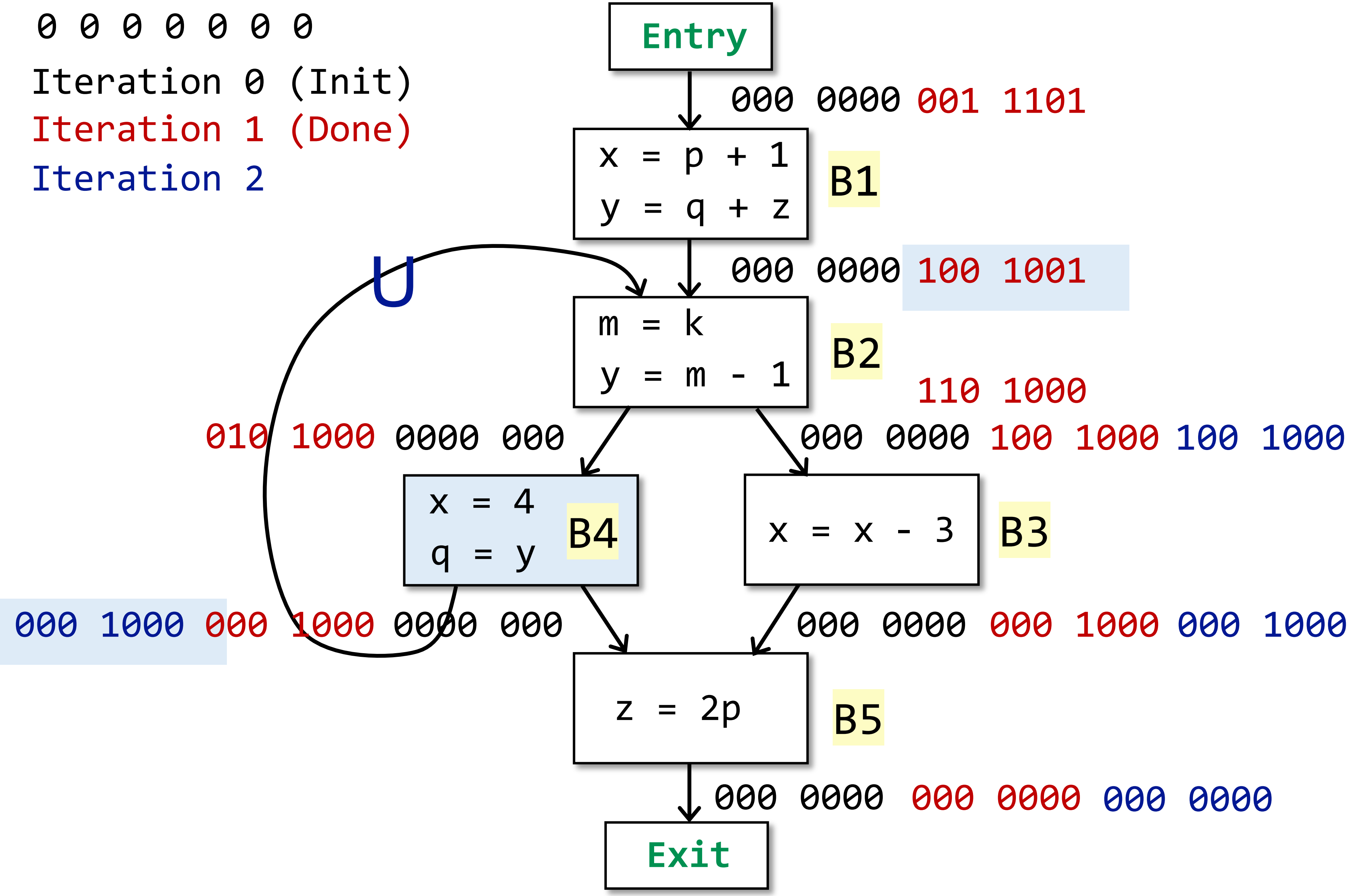

x y z p q m k
0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2
Entry
000 0000 001 1101
x = p + 1
y = q + z
B1
000 0000 100 1001
m = k
y = m - 1
B2
110 1000
010 1000 0000 000
000 0000 100 1000 100 1000
x = 4
q = y
B4
x = x - 3
B3
000 1000 000 1000 0000 000
000 0000 000 1000 000 1000
z = 2p
B5
000 0000 000 0000 000 0000
Exit

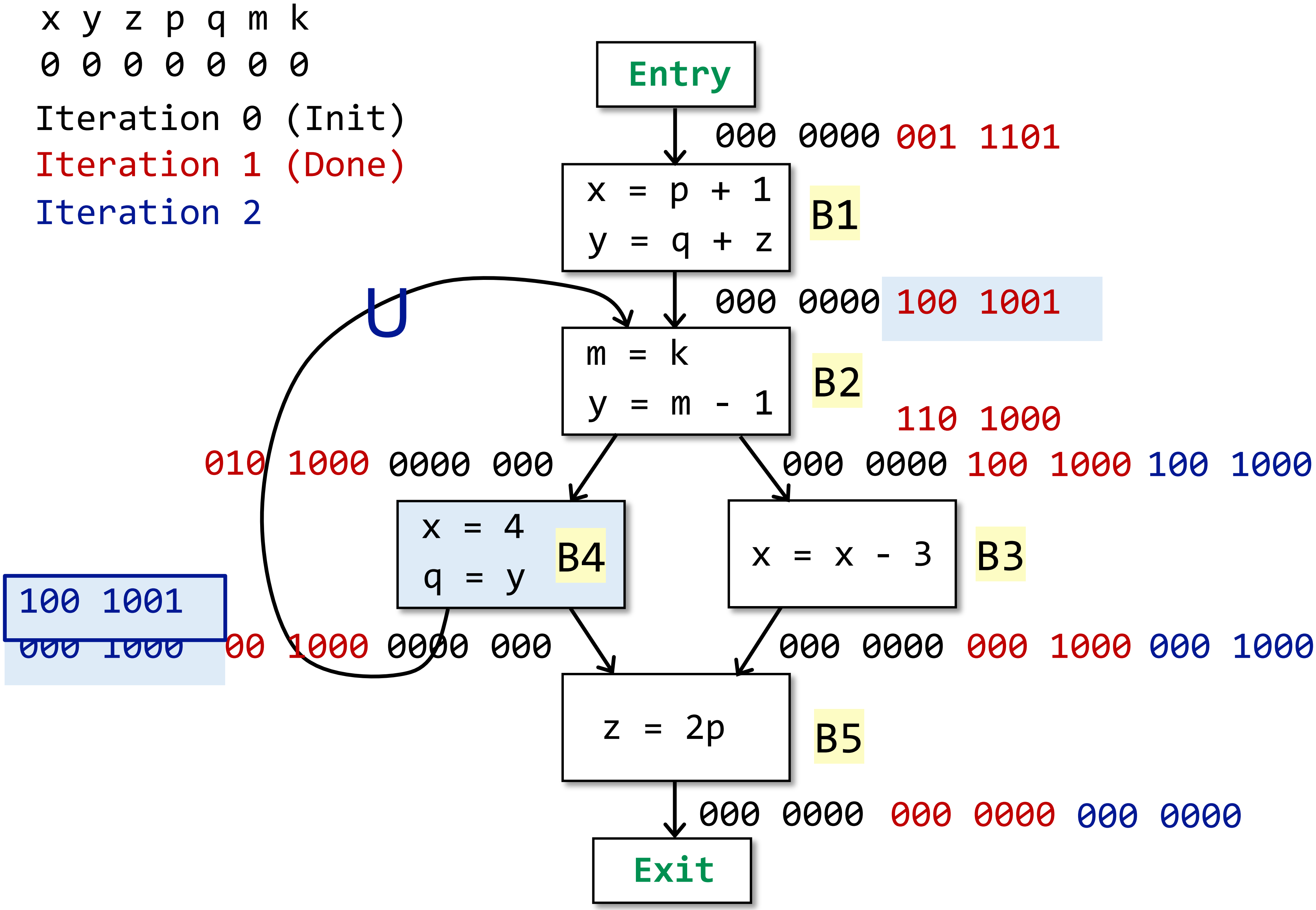


x y z p q m k
0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2
Entry
000 0000 001 1101
x = p + 1
y = q + z
B1
000 0000 100 1001
m = k
y = m - 1
B2
110 1000
010 1000 0000 000
000 0000 100 1000 100 1000
x = 4
q = y
B4
x = x - 3
B3
000 1000 000 1000 0000 000
000 0000 000 1000 000 1000
z = 2p
B5
000 0000 000 0000 000 0000
Exit

x y z p q m k
0 0 0 0 0 0 0

Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2

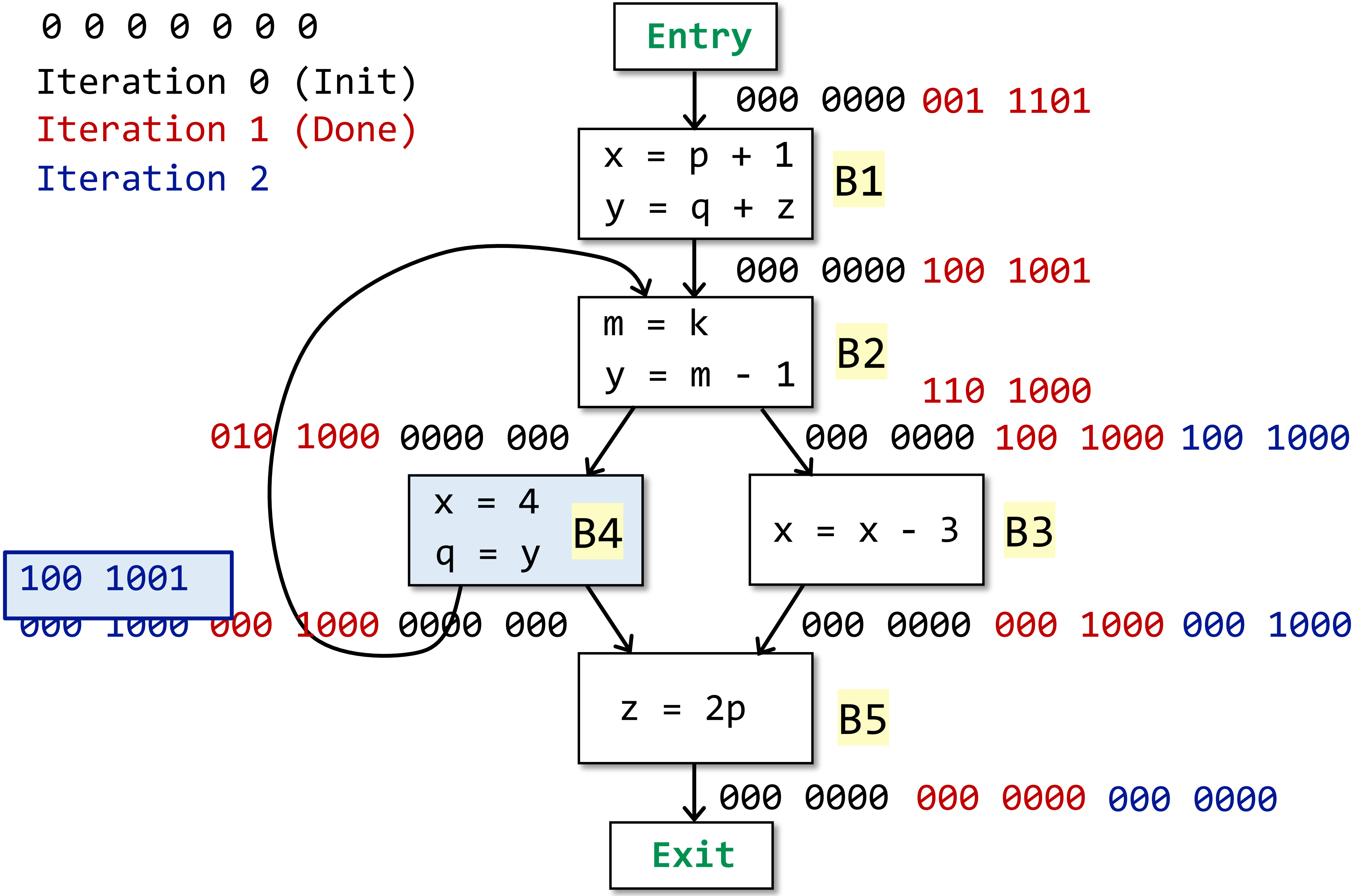




x y z p q m k
0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2
Entry
000 0000 001 1101
x = p + 1
y = q + z
B1
000 0000 100 1001
U
m = k
y = m - 1
B2
110 1000
010 1000 0000 000
000 0000 100 1000 100 1000
x = 4
q = y
B4
x = x - 3
B3
100 1001
000 1000
00 1000 0000 000
000 0000 000 1000 000 1000
z = 2p
B5
000 0000 000 0000 000 0000
Exit

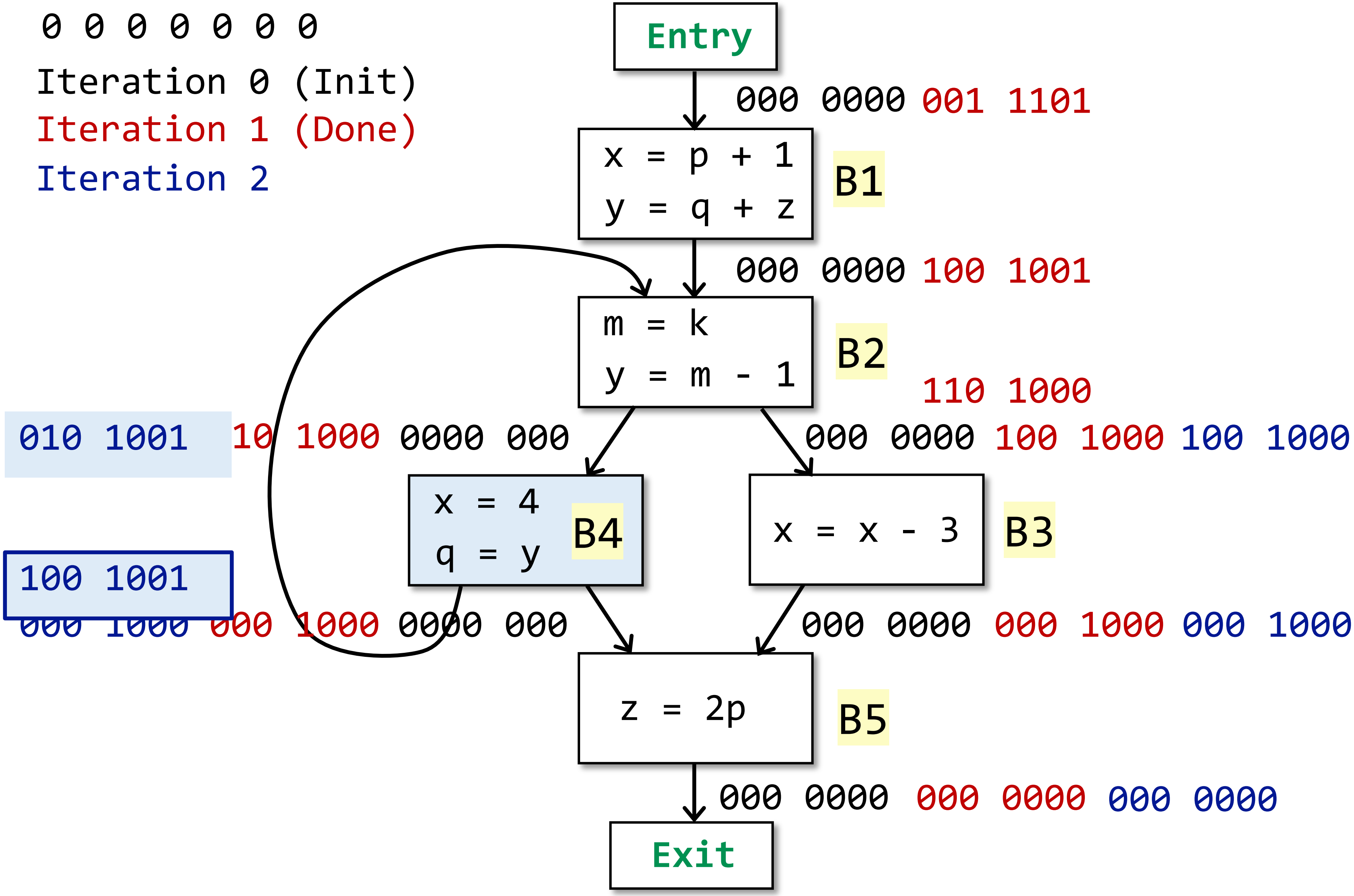
x y z p q m k
0 0 0 0 0 0 0

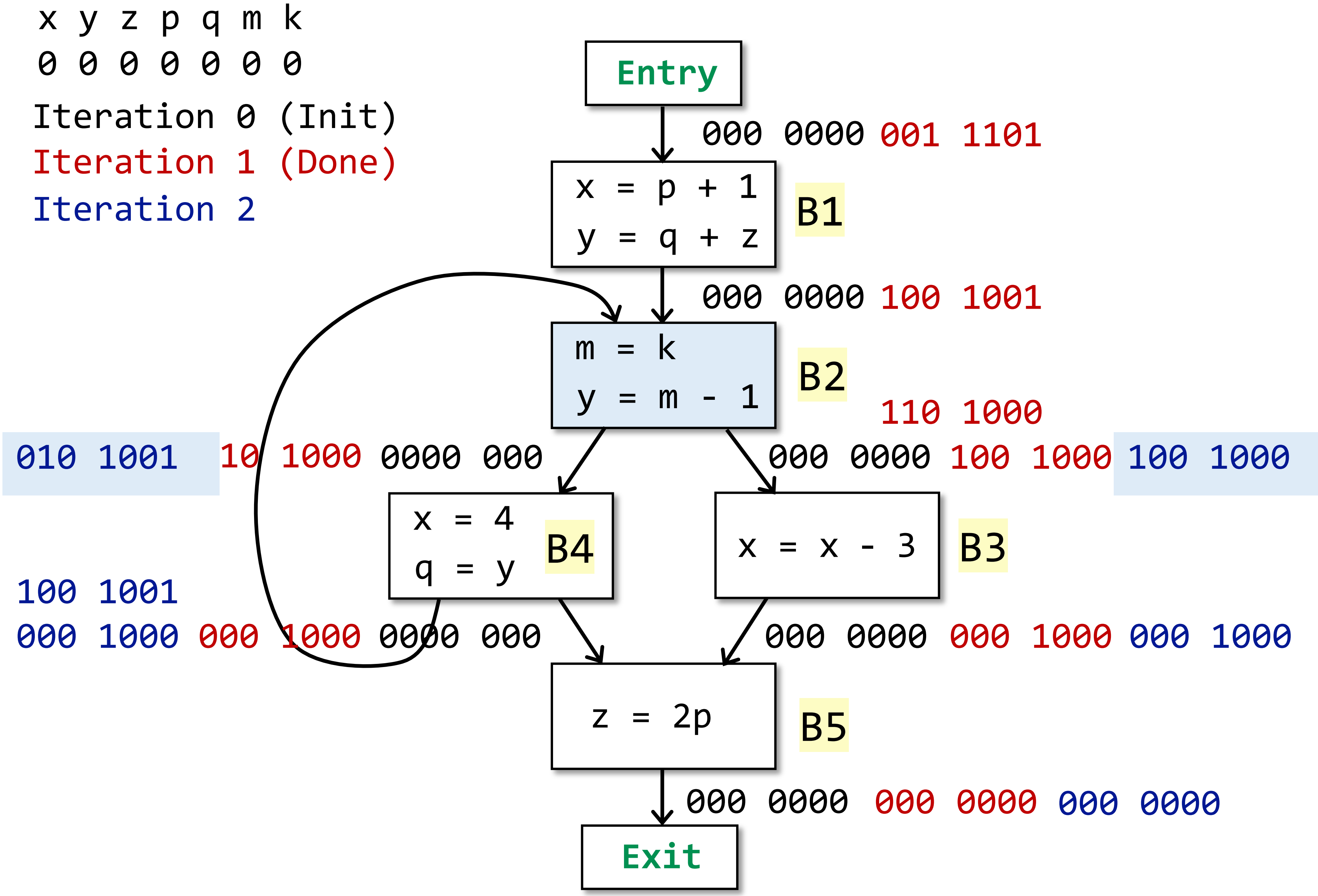
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2

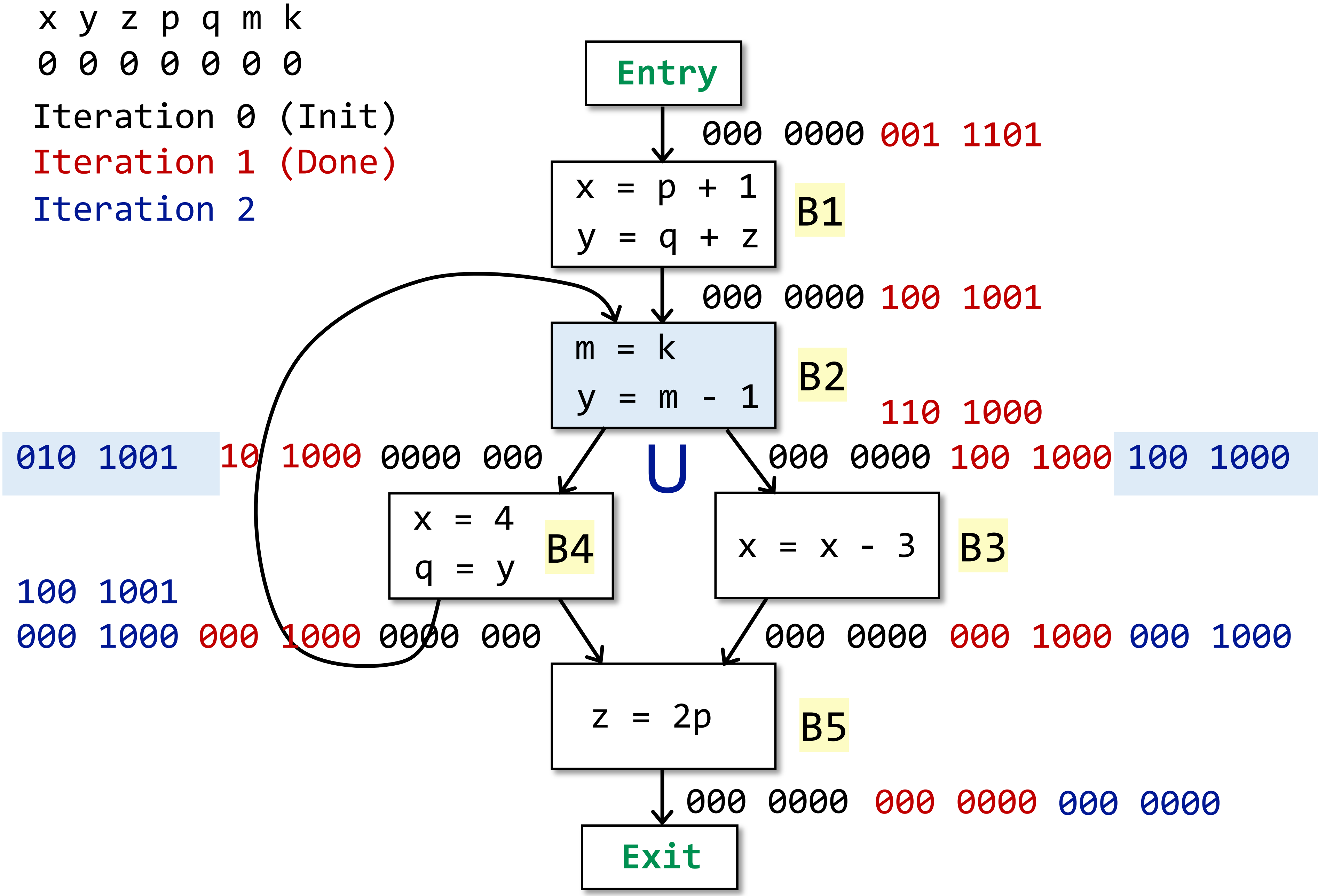


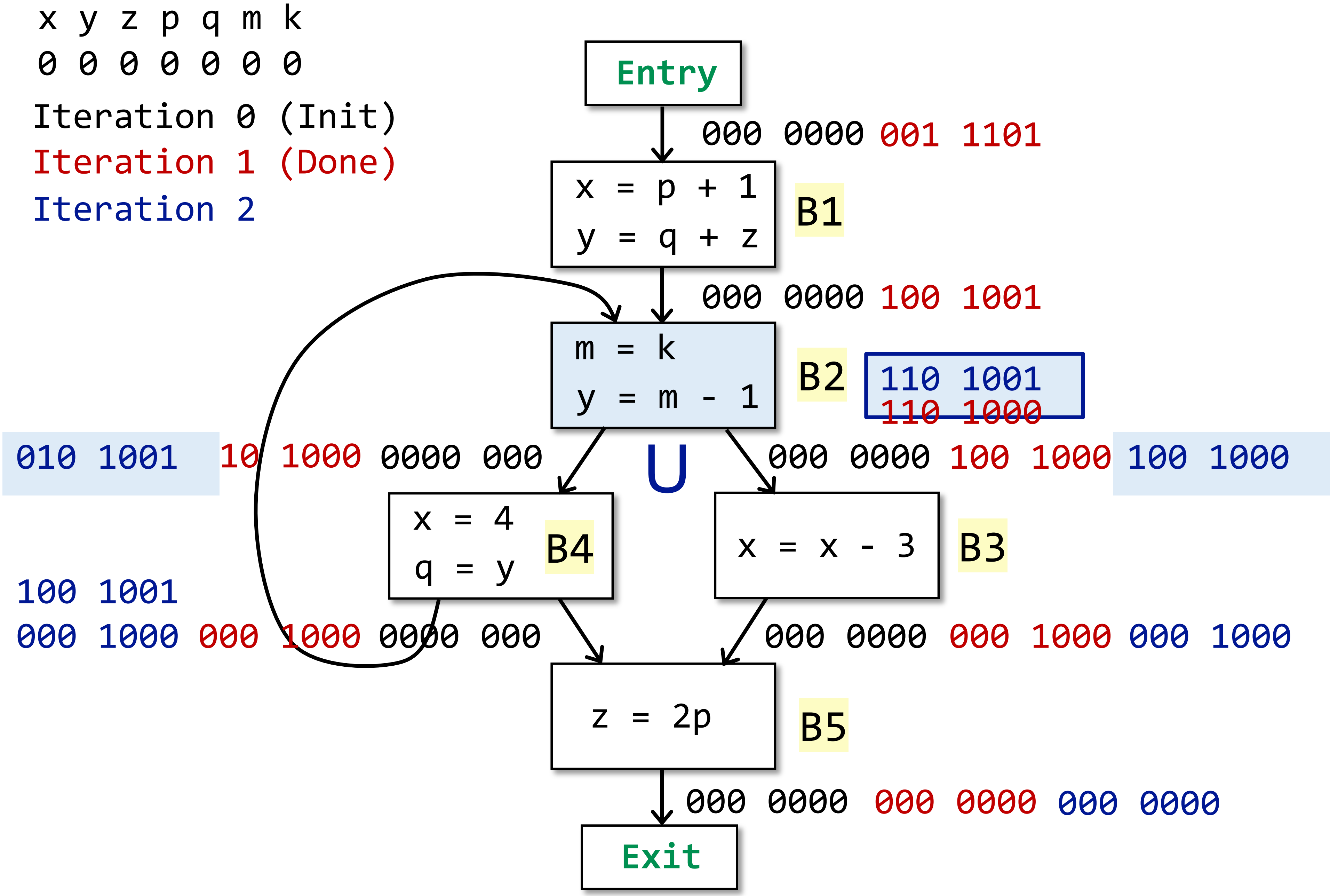
x y z p q m k
0 0 0 0 0 0 0

Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2

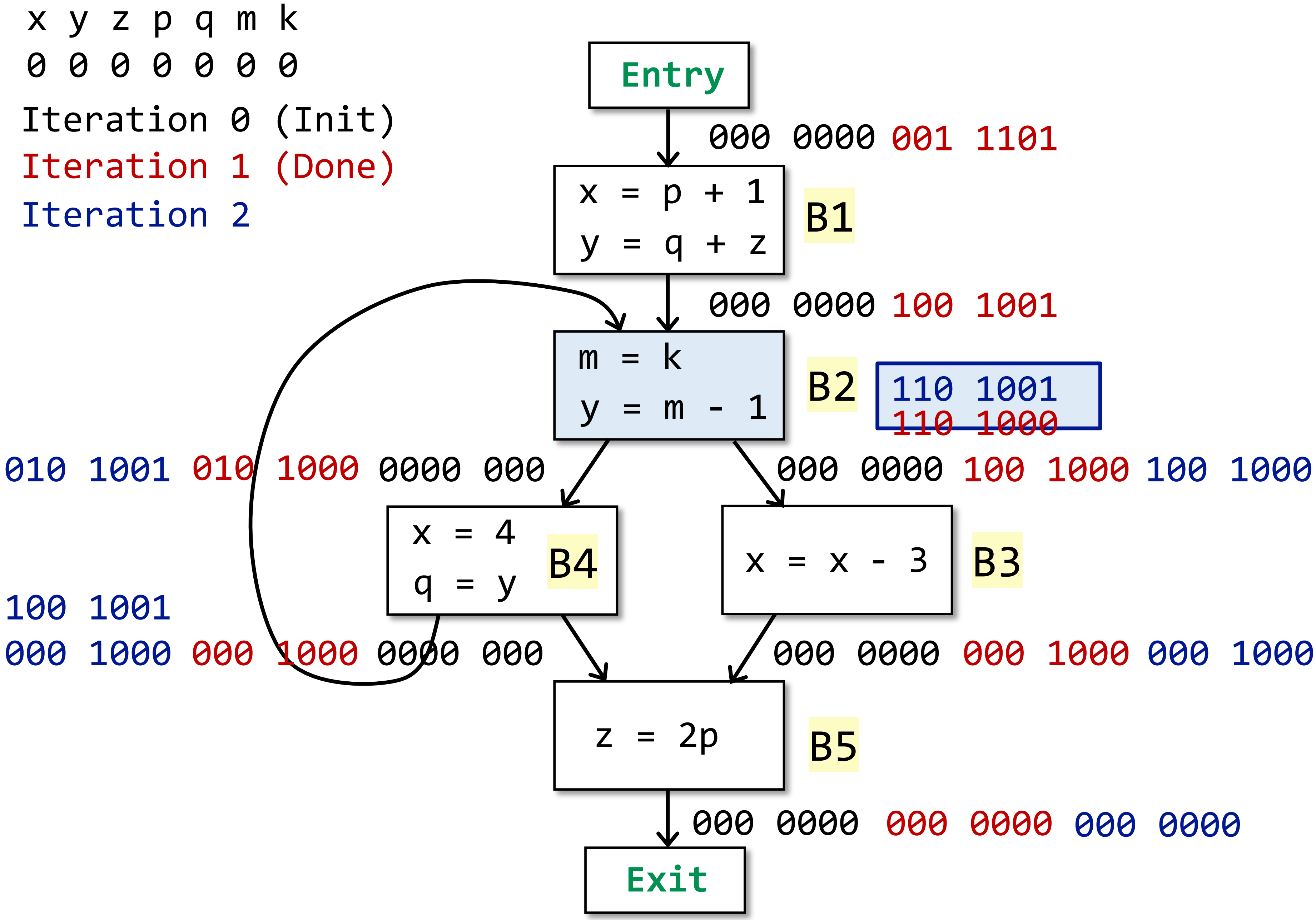



x y z p q m k
0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2
Entry
000 0000 001 1101
x = p + 1
y = q + z
B1
000 0000 100 1001
m = k
y = m - 1
B2
110 1000
010 1001
10 1000 0000 000
000 0000 100 1000 100 1000
x = 4
q = y
B4
x = x - 3
B3
100 1001
000 1000 000 1000 0000 000
000 0000 000 1000 000 1000
z = 2p
B5
000 0000 000 0000 000 0000
Exit


x y z p q m k
0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2
Entry
000 0000 001 1101
x = p + 1
y = q + z
B1
000 0000 100 1001
m = k
y = m - 1
B2
110 1000
010 1001
10 1000 0000 000
U
000 0000 100 1000 100 1000
x = 4
q = y
B4
x = x - 3
B3
100 1001
000 1000 000 1000 0000 000
000 0000 000 1000 000 1000
z = 2p
B5
000 0000 000 0000 000 0000
Exit

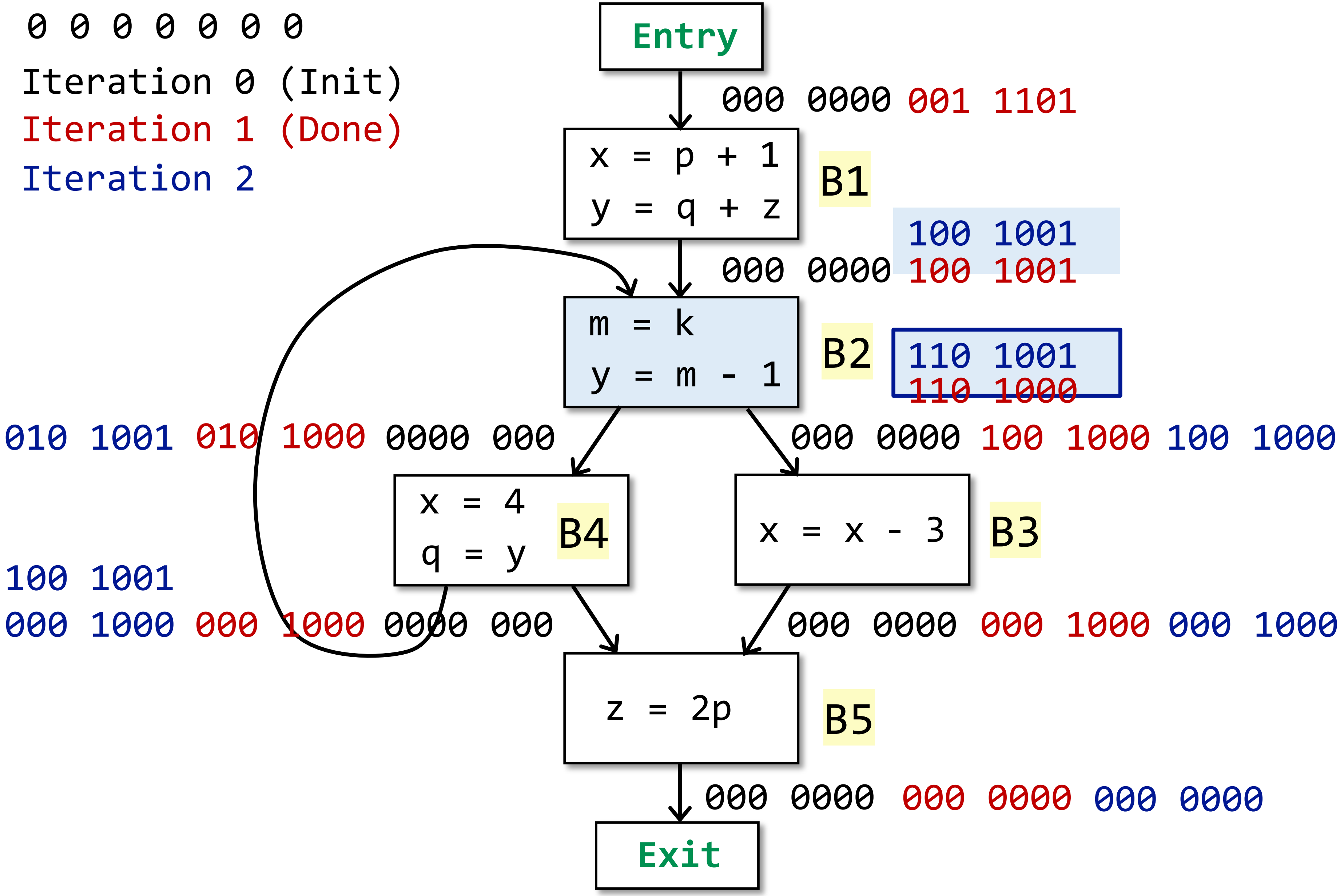


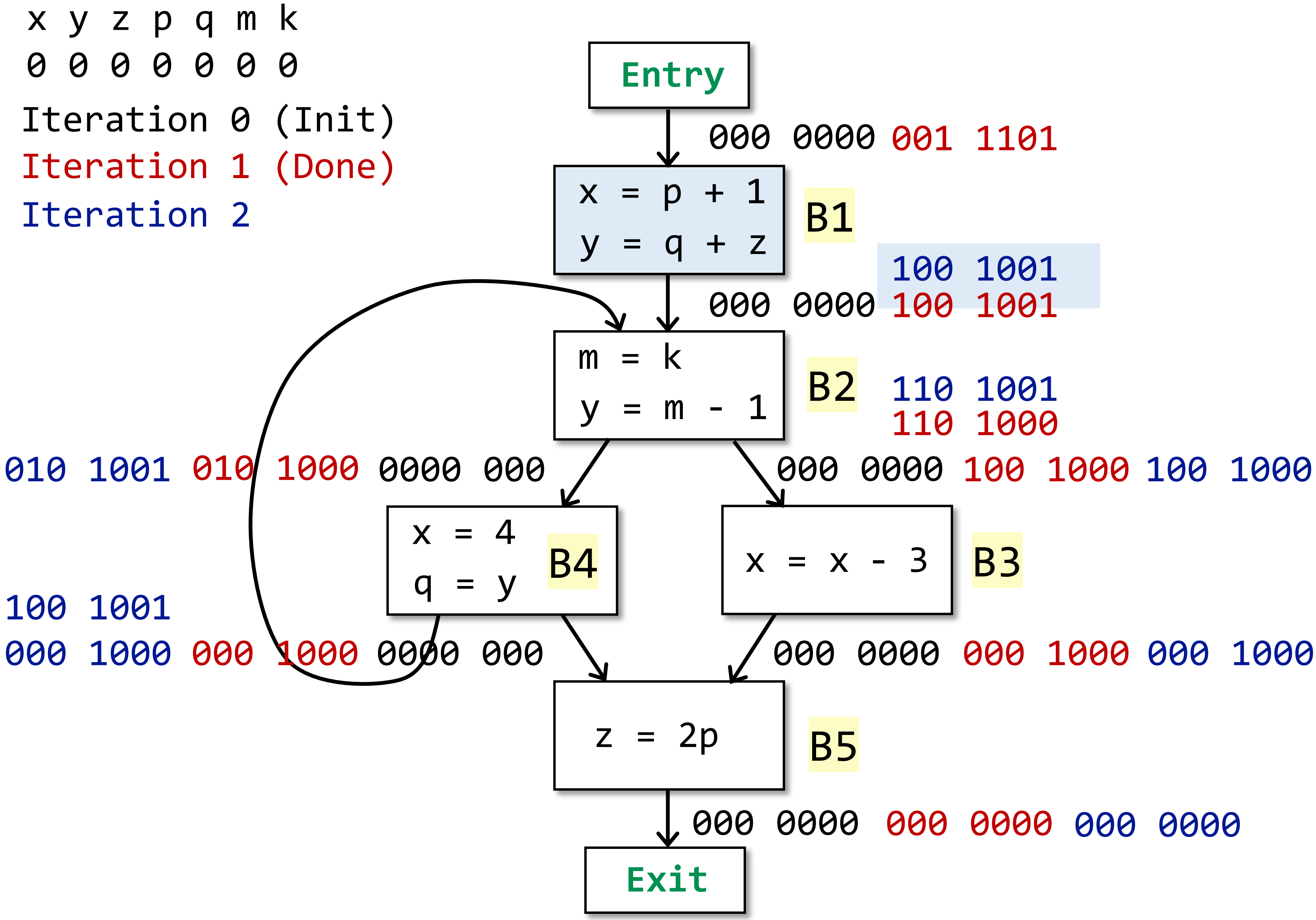
x y z p q m k
0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2
Entry
000 0000 001 1101
x = p + 1
y = q + z
B1
000 0000 100 1001
m = k
y = m - 1
B2
110 1001
~~110 1000~~
010 1001
10 1000 0000 000
U
000 0000 100 1000 100 1000
x = 4
q = y
B4
x = x - 3
B3
100 1001
000 1000 000 1000 0000 000
000 0000 000 1000 000 1000
z = 2p
B5
000 0000 000 0000 000 0000
Exit


x y z p q m k
0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2
Entry
000 0000 001 1101
x = p + 1
y = q + z
B1
000 0000 100 1001
m = k
y = m - 1
B2
110 1001
~~110 1000~~
010 1001 010 1000 0000 000
000 0000 100 1000 100 1000
x = 4
q = y
B4
x = x - 3
B3
100 1001
000 1000 000 1000 0000 000
000 0000 000 1000 000 1000
z = 2p
B5
000 0000 000 0000 000 0000
Exit

x y z p q m k
0 0 0 0 0 0 0

Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2

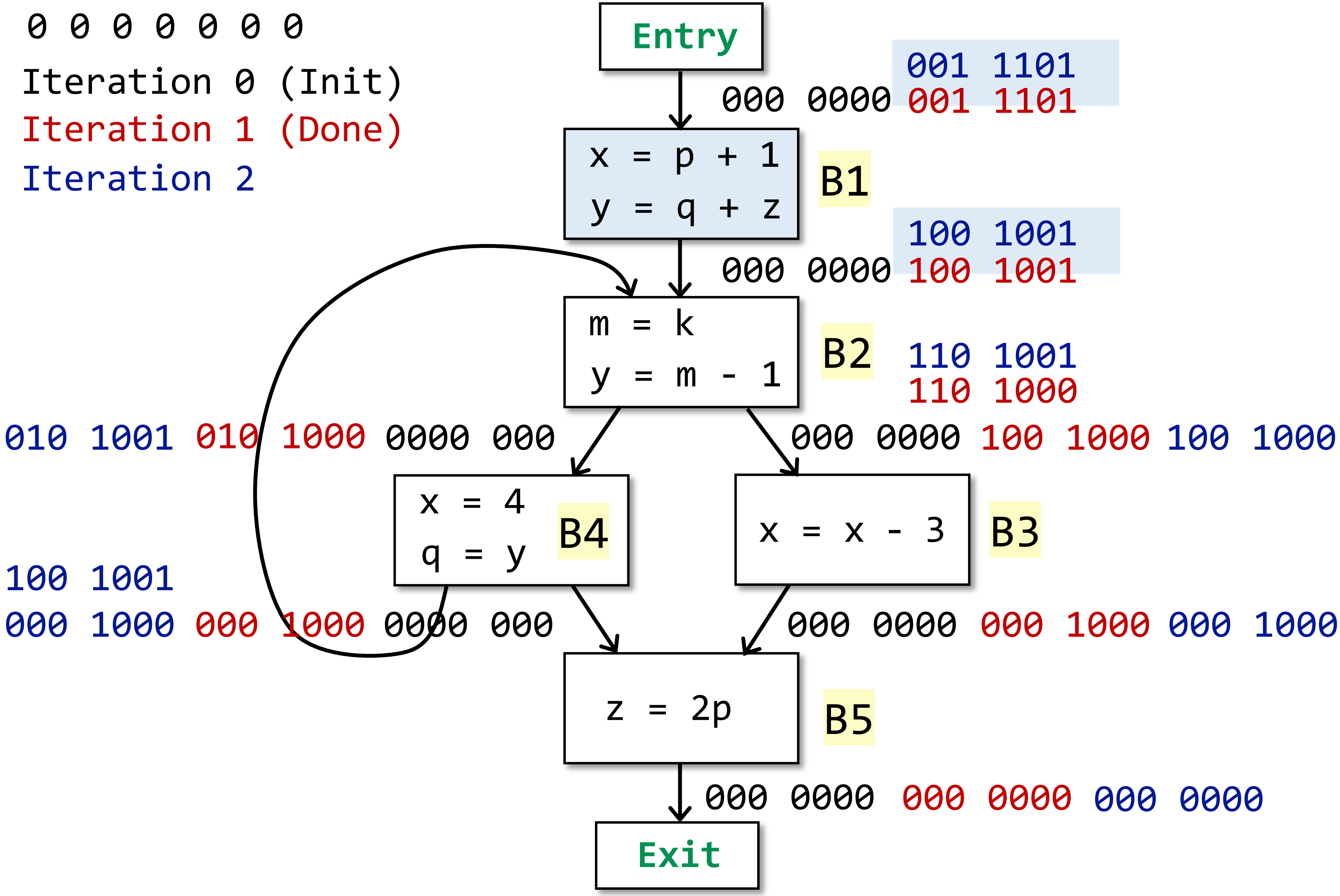


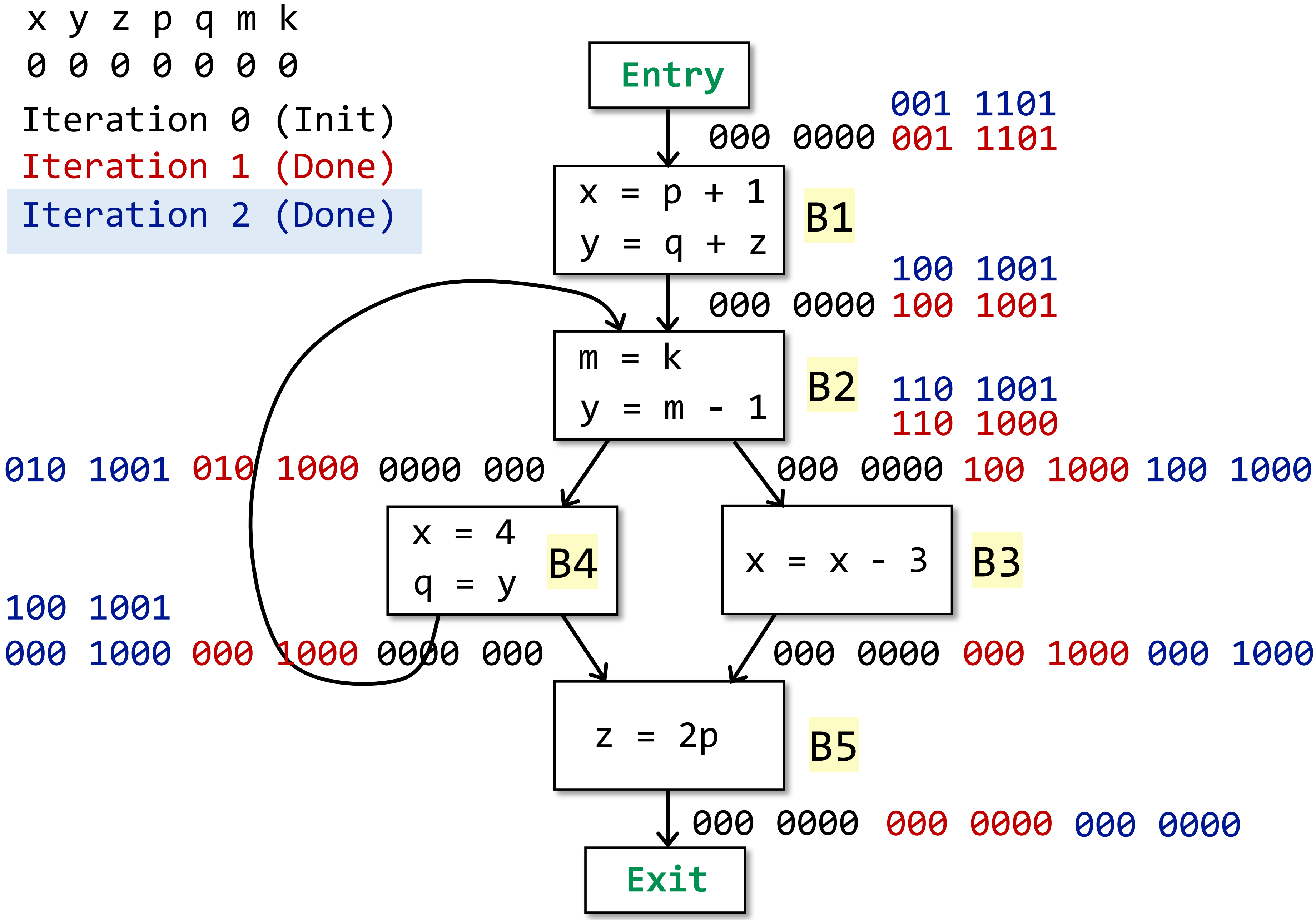


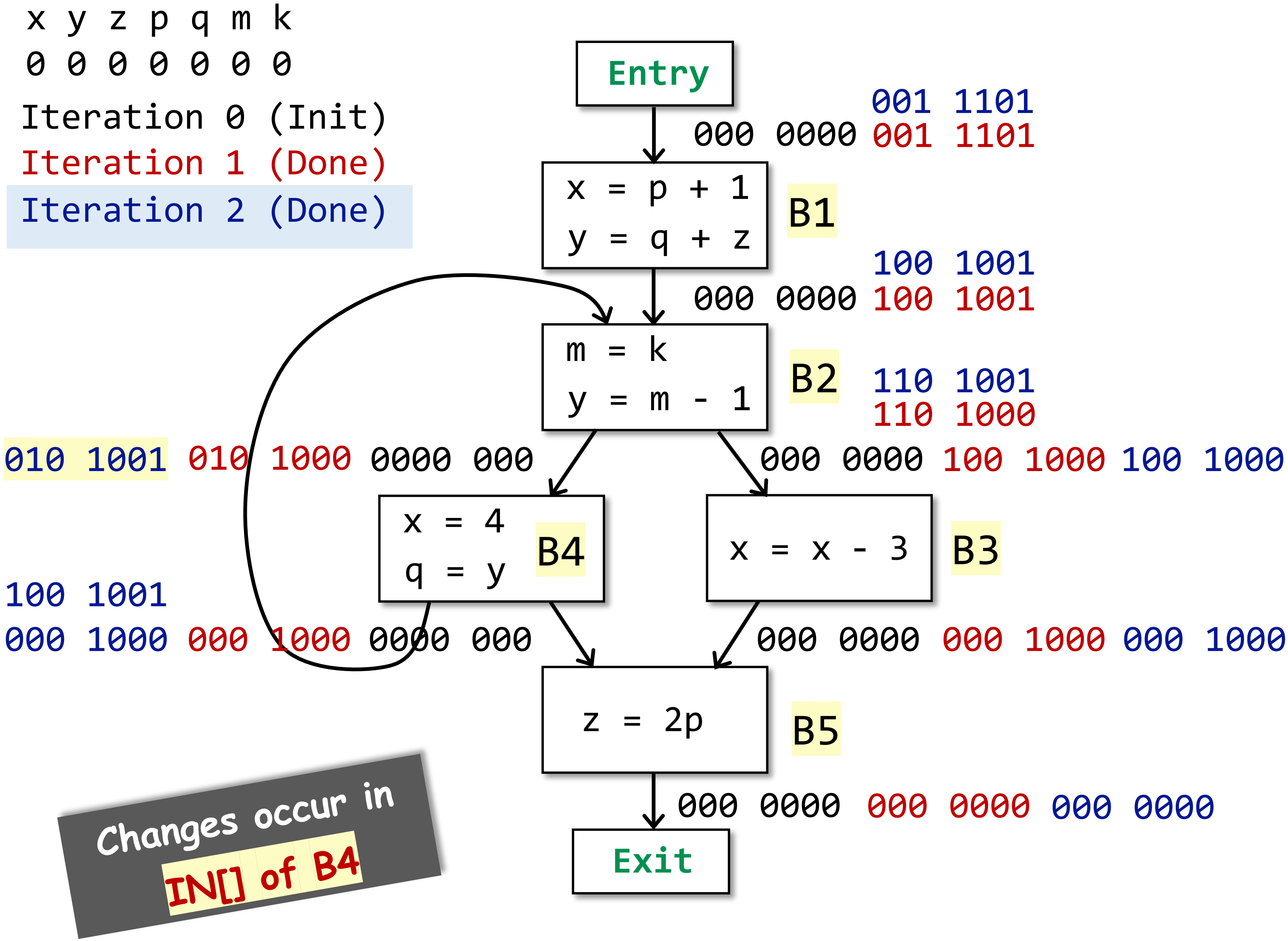
x y z p q m k
0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2
Entry
000 0000 001 1101
x = p + 1
y = q + z
B1
100 1001
000 0000 100 1001
m = k
y = m - 1
B2
110 1001
110 1000
010 1001 010 1000 0000 000
000 0000 100 1000 100 1000
x = 4
q = y
B4
x = x - 3
B3
100 1001
000 1000 000 1000 0000 000
000 0000 000 1000 000 1000
z = 2p
B5
000 0000 000 0000 000 0000
Exit

x y z p q m k
0 0 0 0 0 0 0

Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2

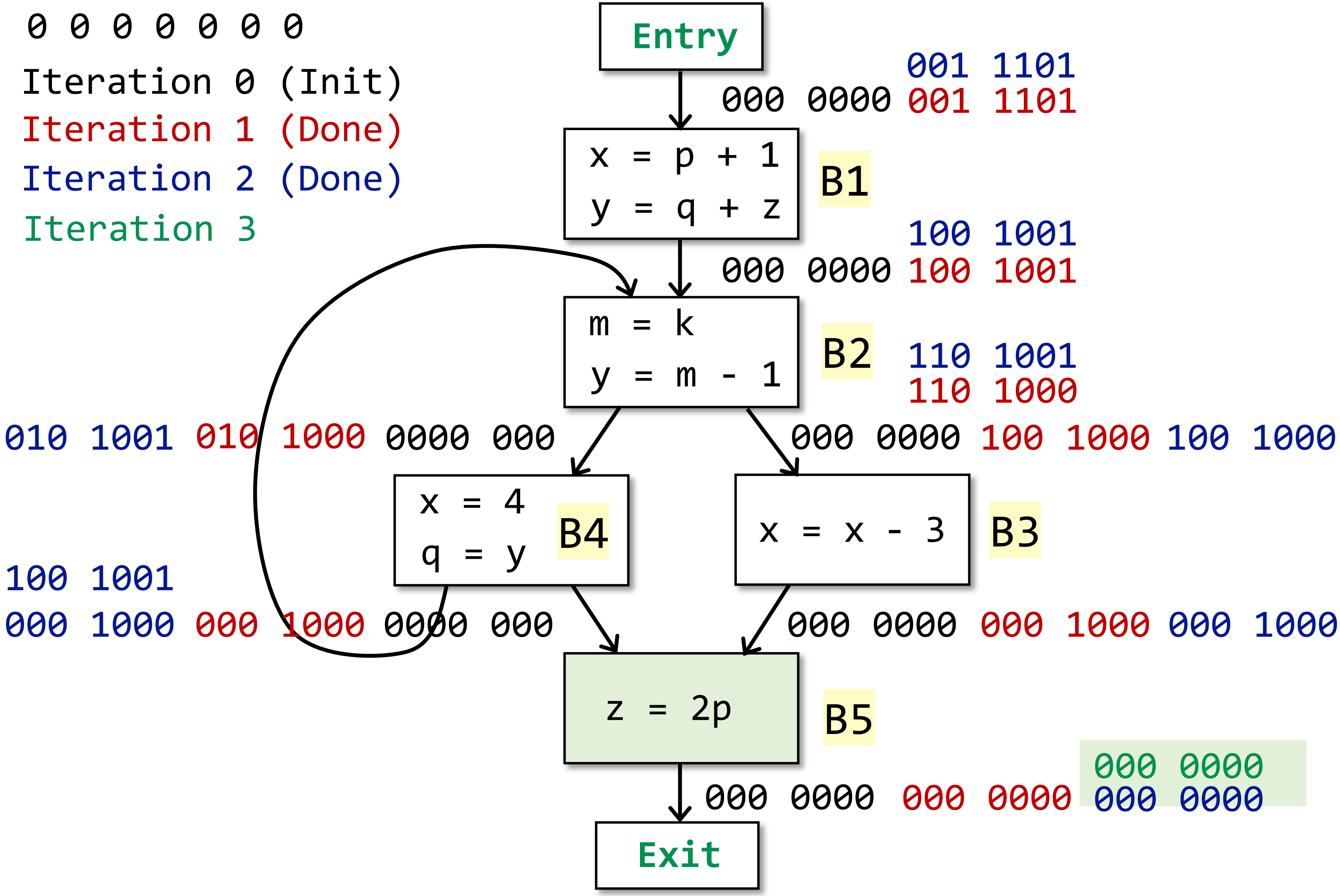



x y z p q m k
0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2 (Done)
Entry
001 1101
000 0000 001 1101
x = p + 1
y = q + z
B1
100 1001
000 0000 100 1001
m = k
y = m - 1
B2
110 1001
110 1000
010 1001 010 1000 0000 000
000 0000 100 1000 100 1000
x = 4
q = y
B4
x = x - 3
B3
100 1001
000 1000 000 1000 0000 000
000 0000 000 1000 000 1000
z = 2p
B5
000 0000 000 0000 000 0000
Exit


x y z p q m k
0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2 (Done)
Entry
001 1101
000 0000 001 1101
x = p + 1
y = q + z
B1
100 1001
000 0000 100 1001
m = k
y = m - 1
B2
110 1001
110 1000
010 1001 010 1000 0000 000
000 0000 100 1000 100 1000
x = 4
q = y
B4
x = x - 3
B3
100 1001
000 1000 000 1000 0000 000
000 0000 000 1000 000 1000
z = 2p
B5
000 0000 000 0000 000 0000
Exit
Changes occur in
IN[] of B4

x y z p q m k
0 0 0 0 0 0 0

Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2 (Done)
Iteration 3



x y z p q m k

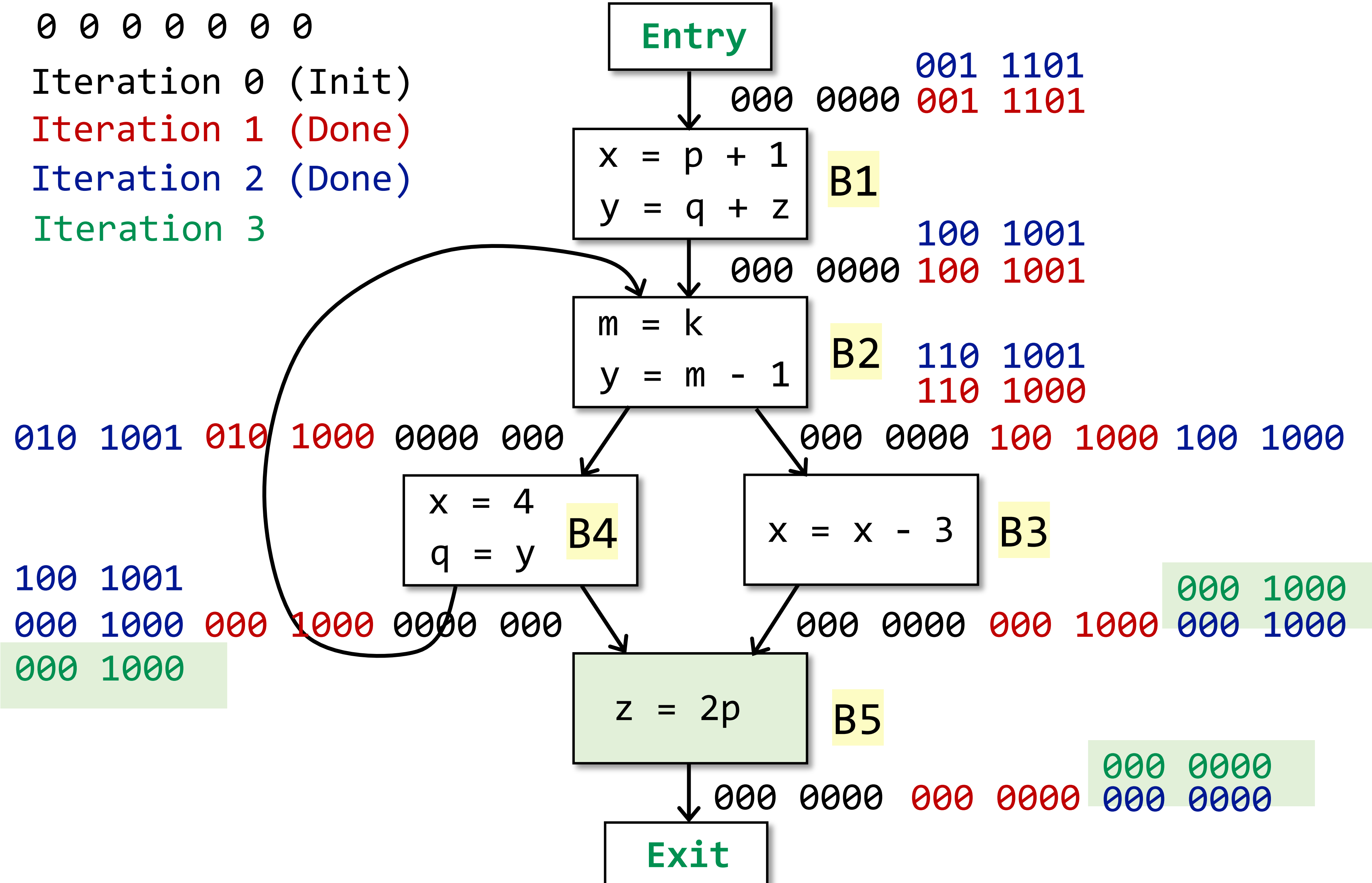
0 0 0 0 0 0 0

Iteration 0 (Init)

Iteration 1 (Done)

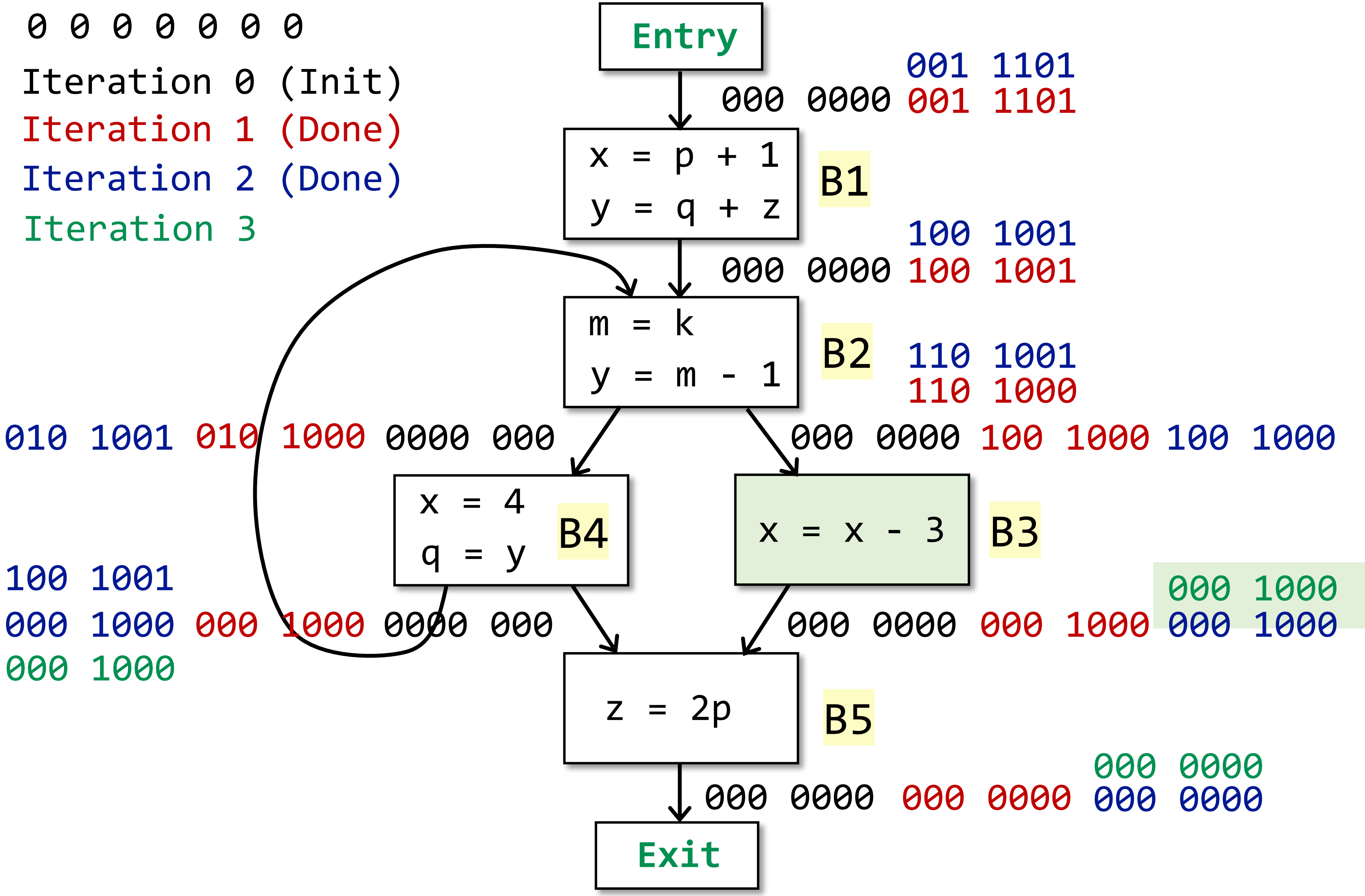
Iteration 2 (Done)

Iteration 3



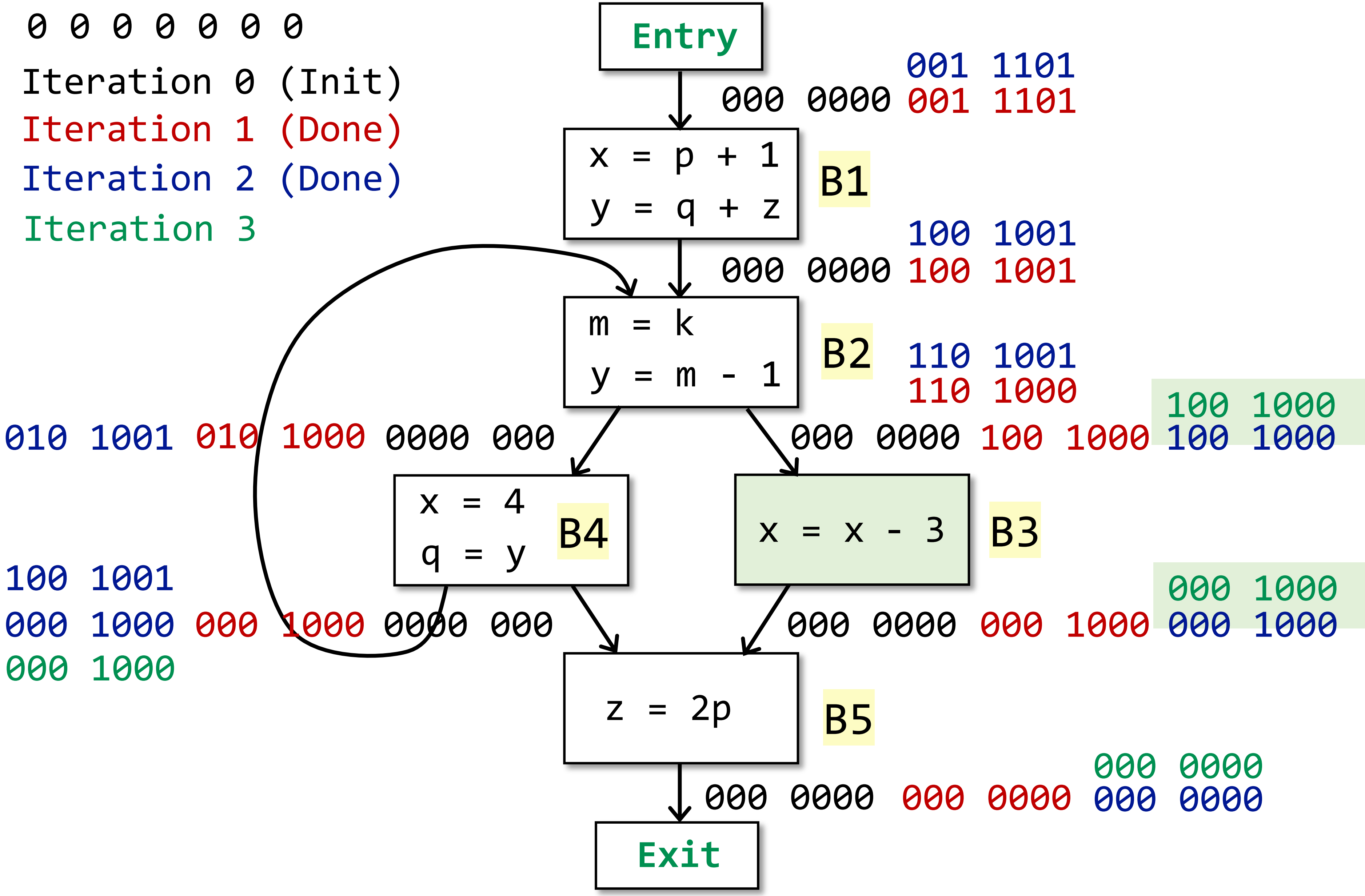
x y z p q m k
0 0 0 0 0 0 0

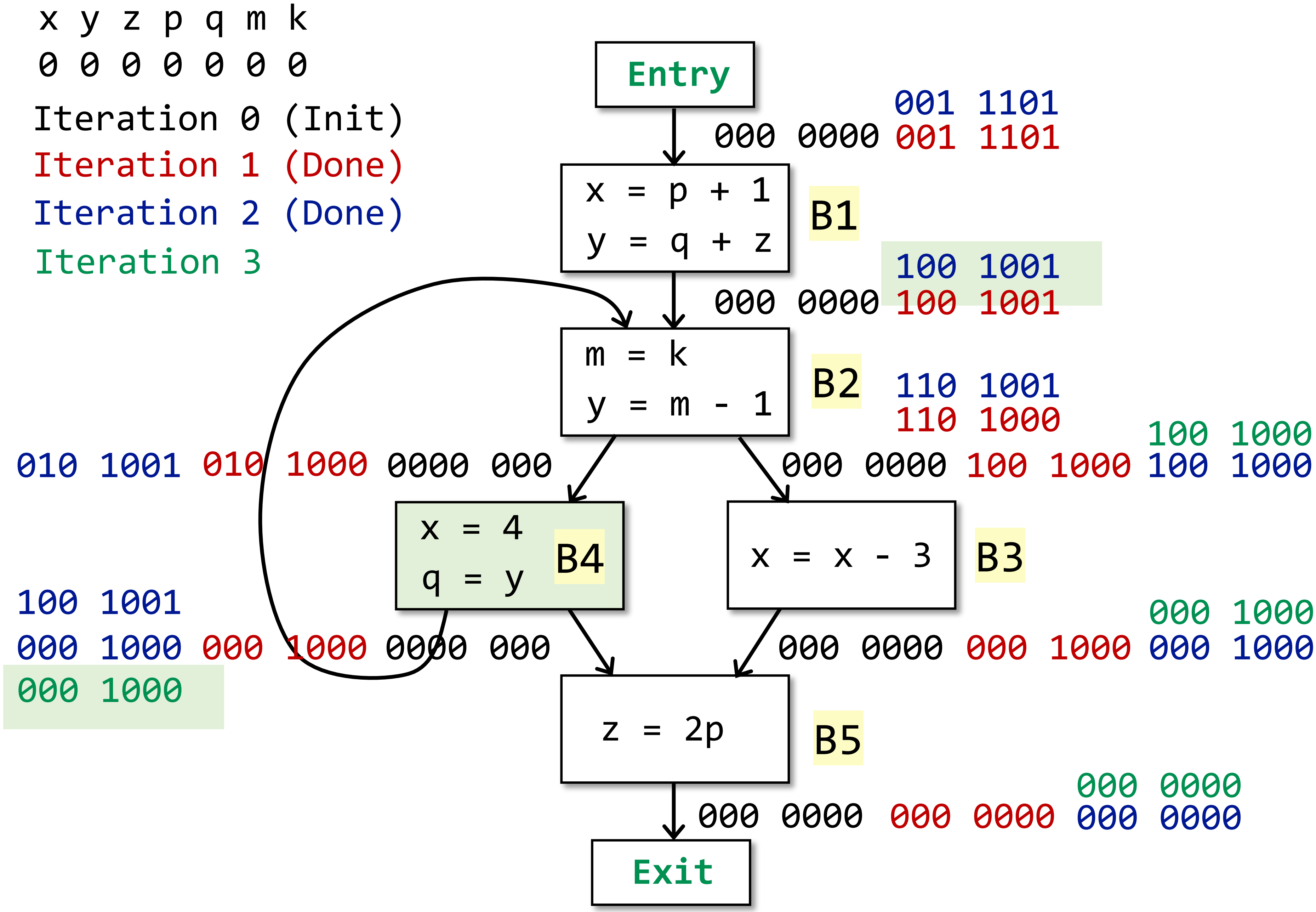
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2 (Done)
Iteration 3



x y z p q m k
0 0 0 0 0 0 0

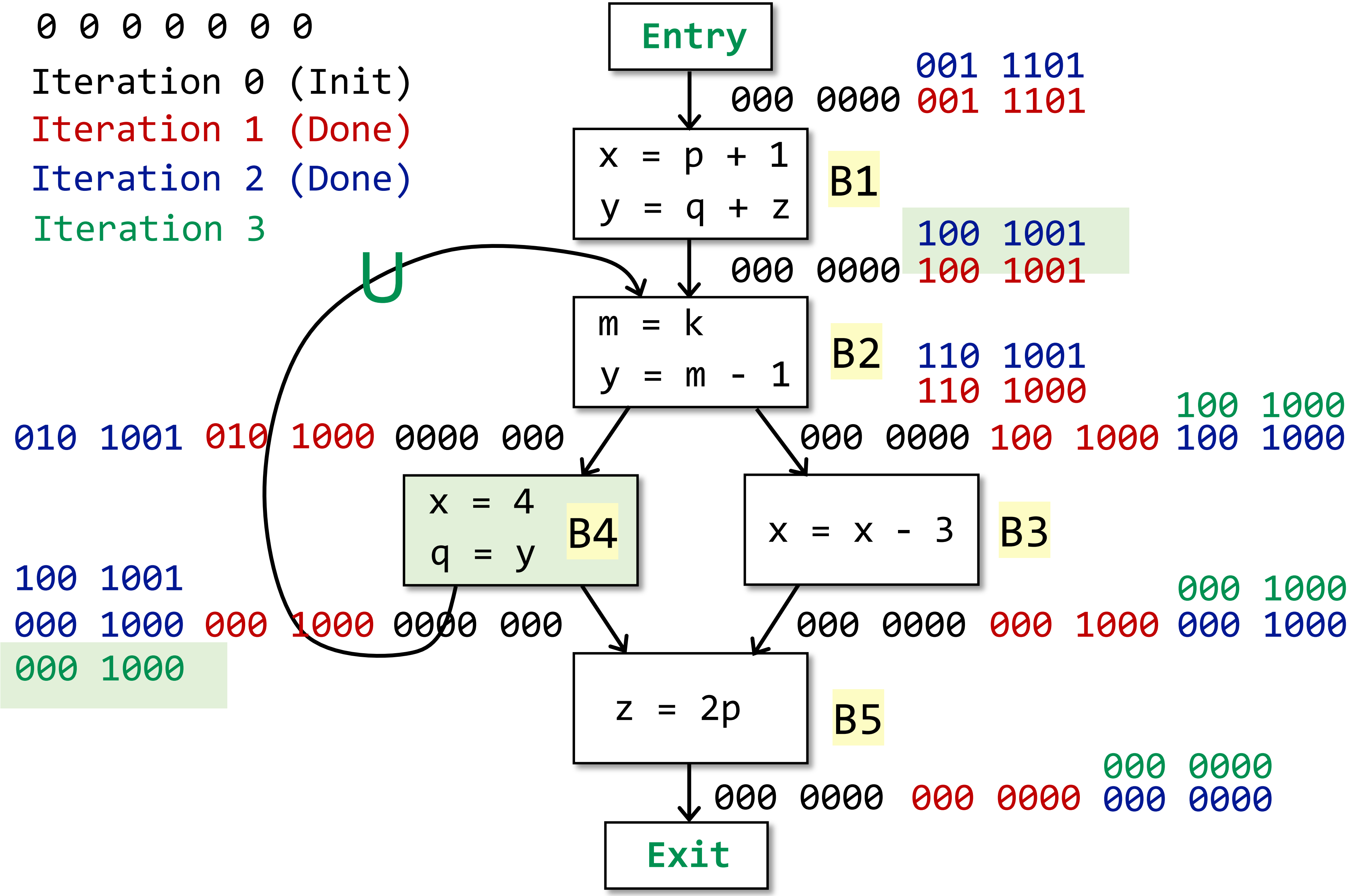
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2 (Done)
Iteration 3

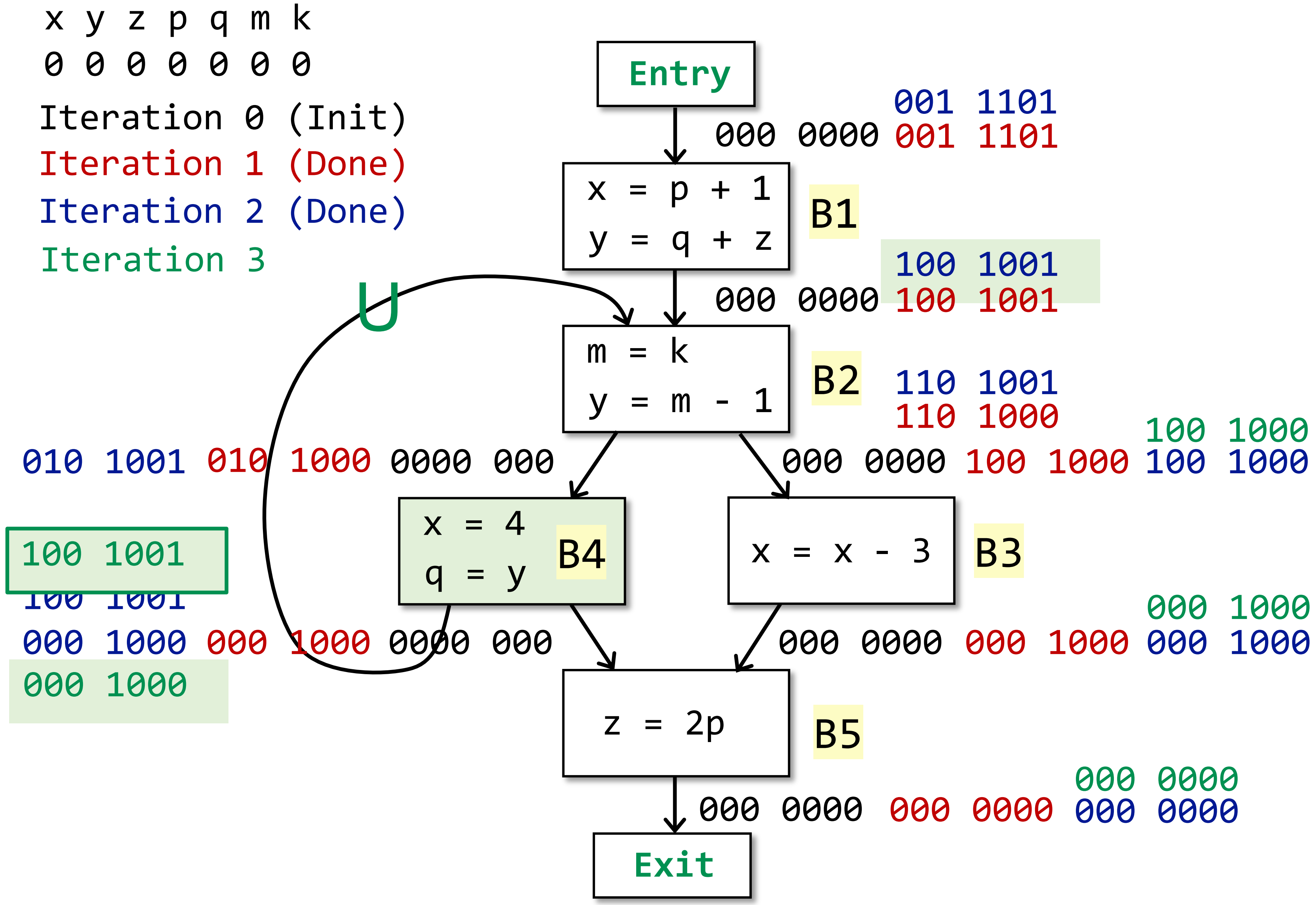


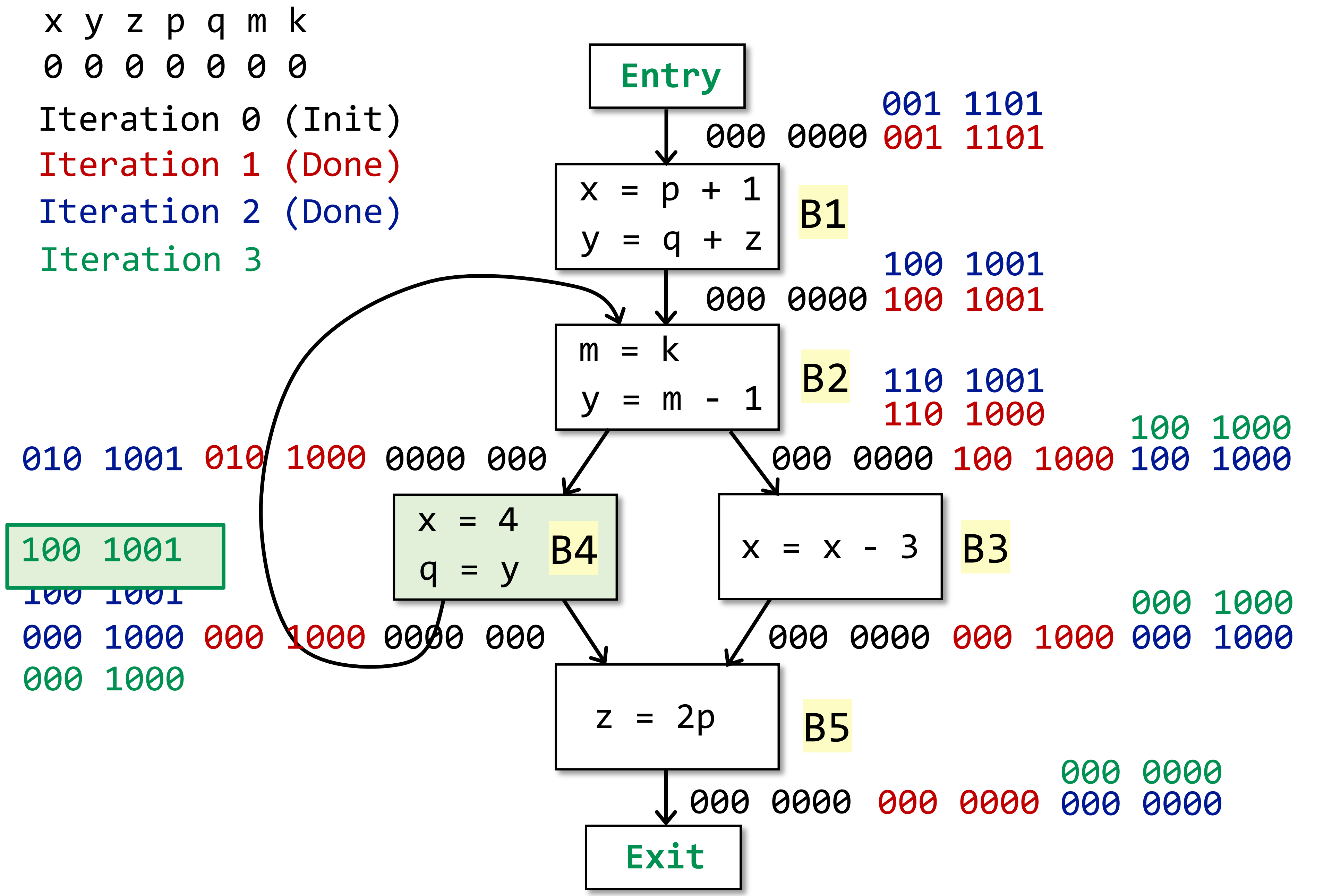

x y z p q m k
0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2 (Done)
Iteration 3
Entry
001 1101
000 0000 001 1101
x = p + 1
y = q + z
B1
100 1001
000 0000 100 1001
m = k
y = m - 1
B2
110 1001
110 1000
100 1000
000 0000 100 1000 100 1000
010 1001 010 1000 0000 000
x = 4
q = y
B4
x = x - 3
B3
100 1001
000 1000 000 1000 0000 000
000 1000
000 1000
000 0000 000 1000 000 1000
z = 2p
B5
000 0000
000 0000 000 0000 000 0000
Exit

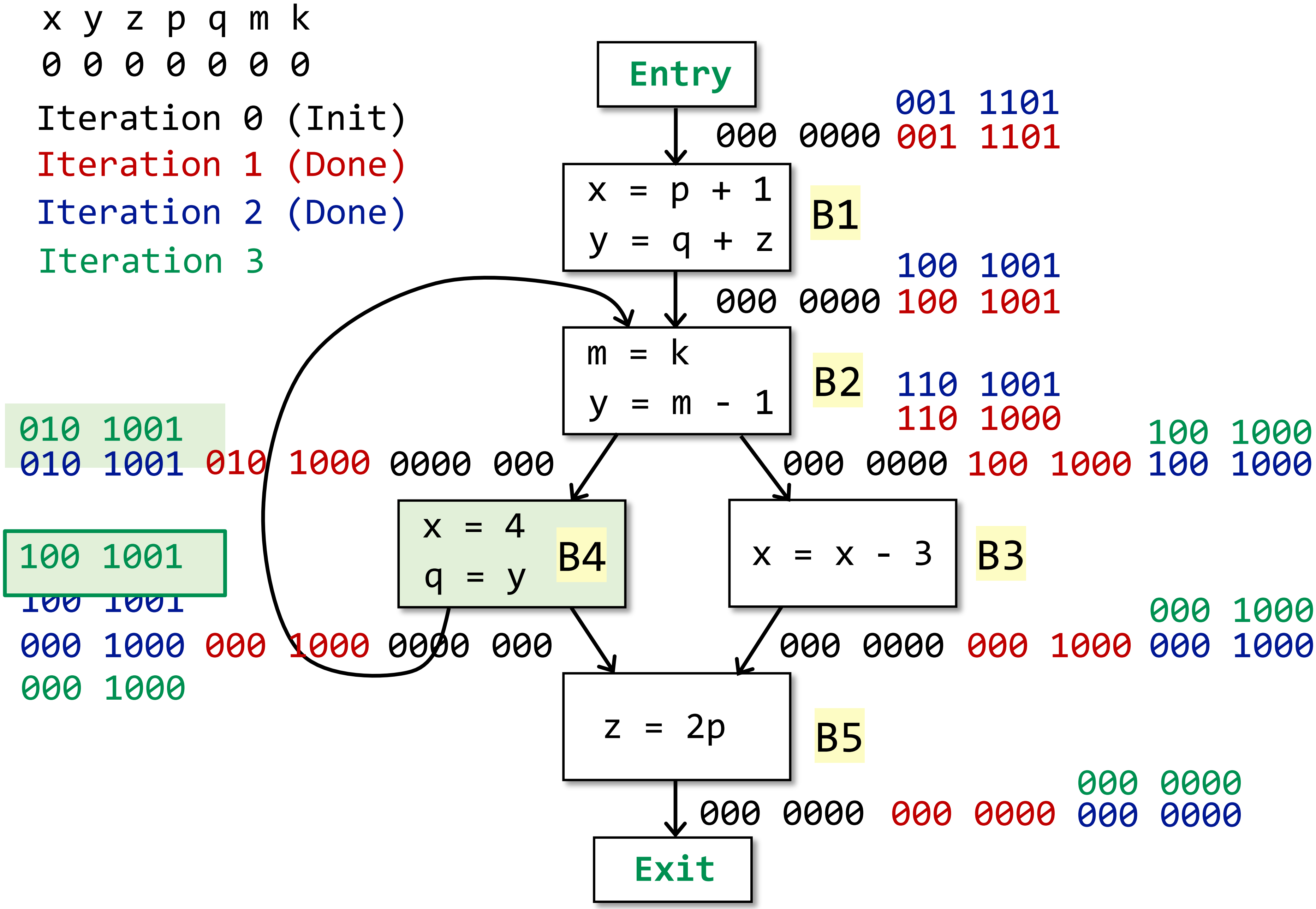
x y z p q m k
0 0 0 0 0 0 0

Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2 (Done)
Iteration 3

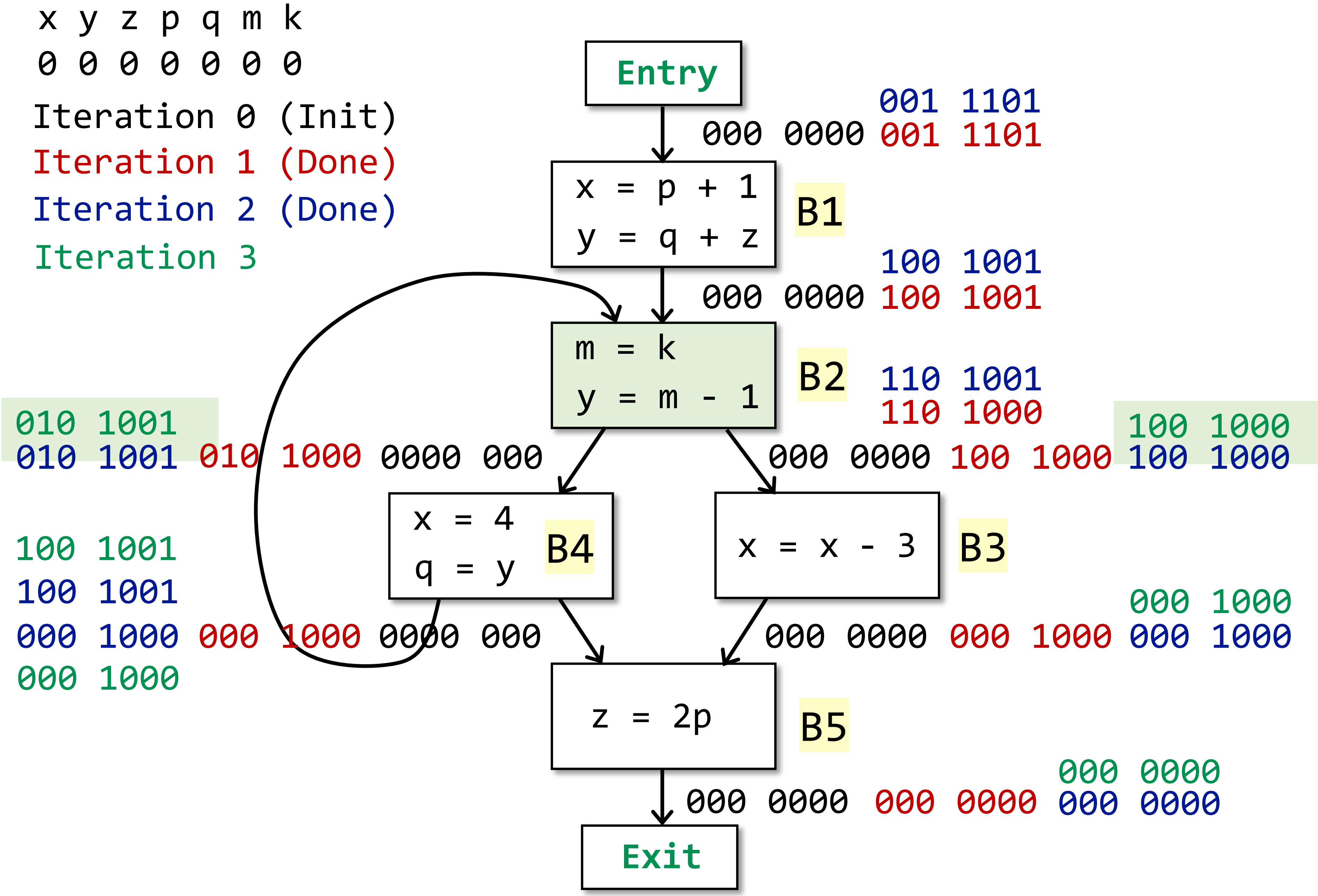


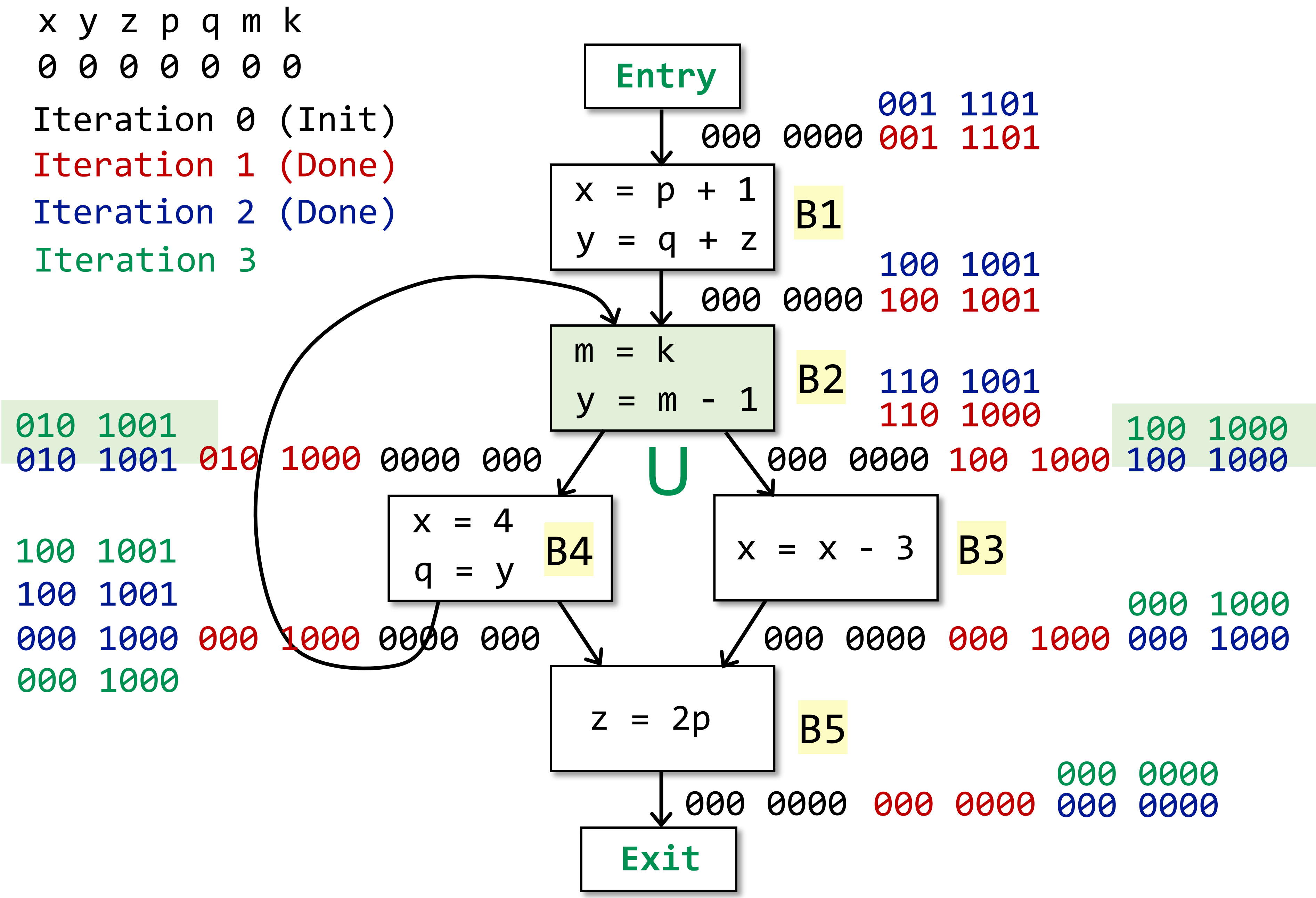

x y z p q m k
0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2 (Done)
Iteration 3
Entry
001 1101
000 0000 001 1101
x = p + 1
y = q + z
B1
100 1001
000 0000 100 1001
∪
m = k
y = m - 1
B2
110 1001
110 1000
100 1000
010 1001 010 1000 0000 000
000 0000 100 1000 100 1000
x = 4
q = y
B4
x = x - 3
B3
100 1001
100 1001
000 1000
000 1000 000 1000 0000 000
000 0000 000 1000 000 1000
000 1000
z = 2p
B5
000 0000
000 0000 000 0000 000 0000
Exit

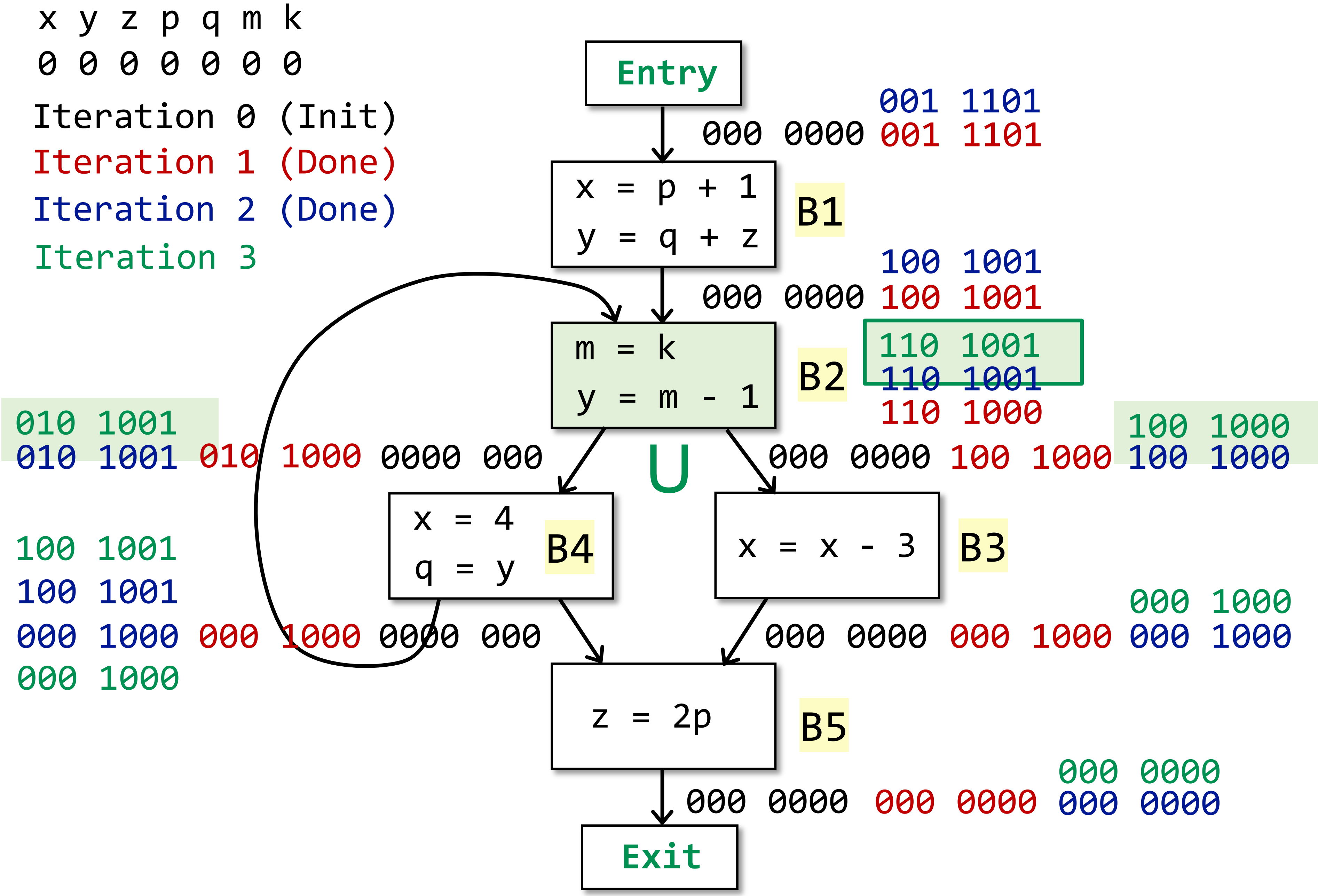

x y z p q m k
0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2 (Done)
Iteration 3
Entry
001 1101
000 0000 001 1101
x = p + 1
y = q + z
B1
100 1001
000 0000 100 1001
m = k
y = m - 1
B2
110 1001
110 1000
100 1000
010 1001 010 1000 0000 000
000 0000 100 1000 100 1000
x = 4
q = y
B4
x = x - 3
B3
100 1001
100 1001
000 1000
000 1000 000 1000 0000 000
000 0000 000 1000 000 1000
000 1000
z = 2p
B5
000 0000
000 0000 000 0000 000 0000
Exit

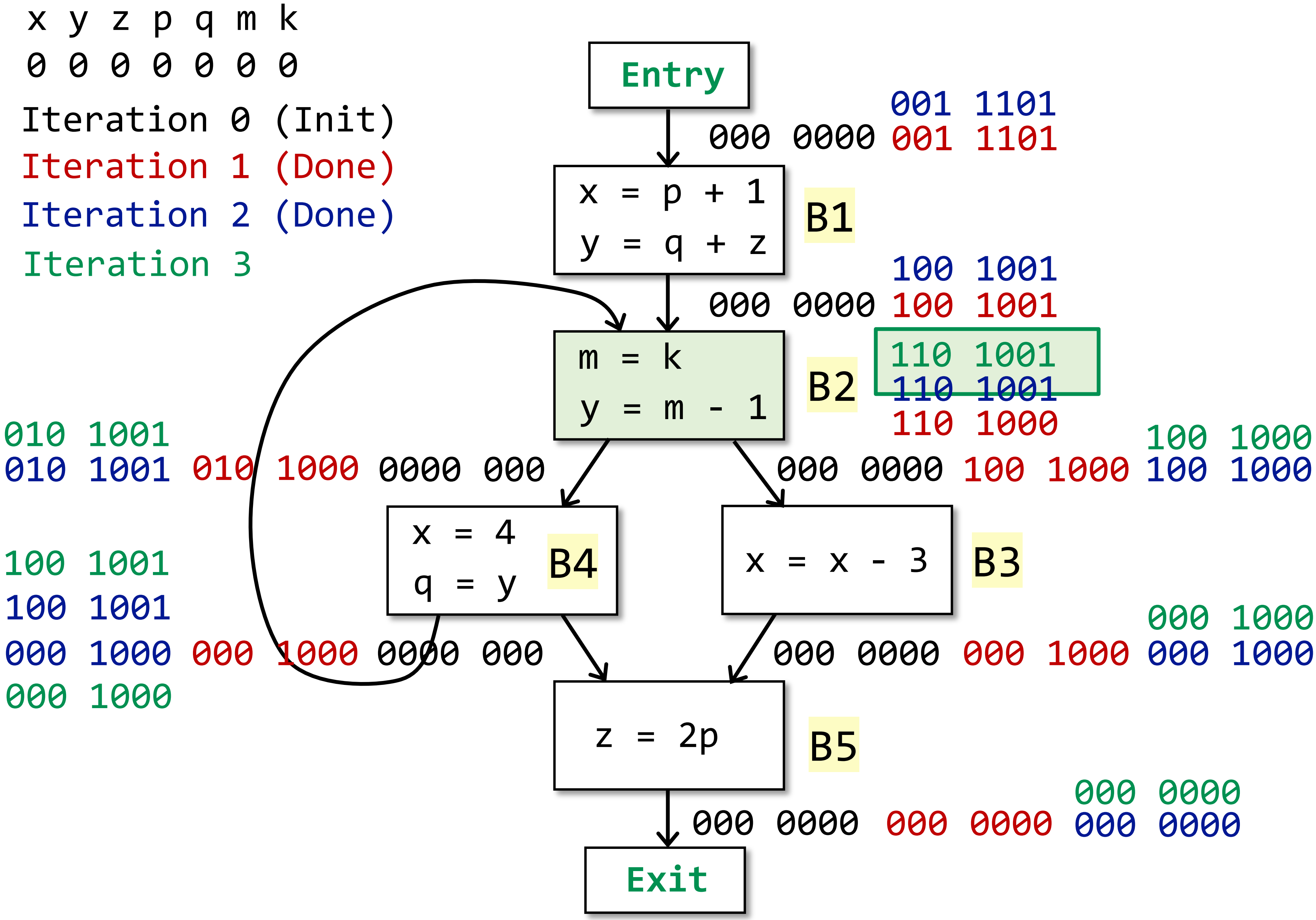


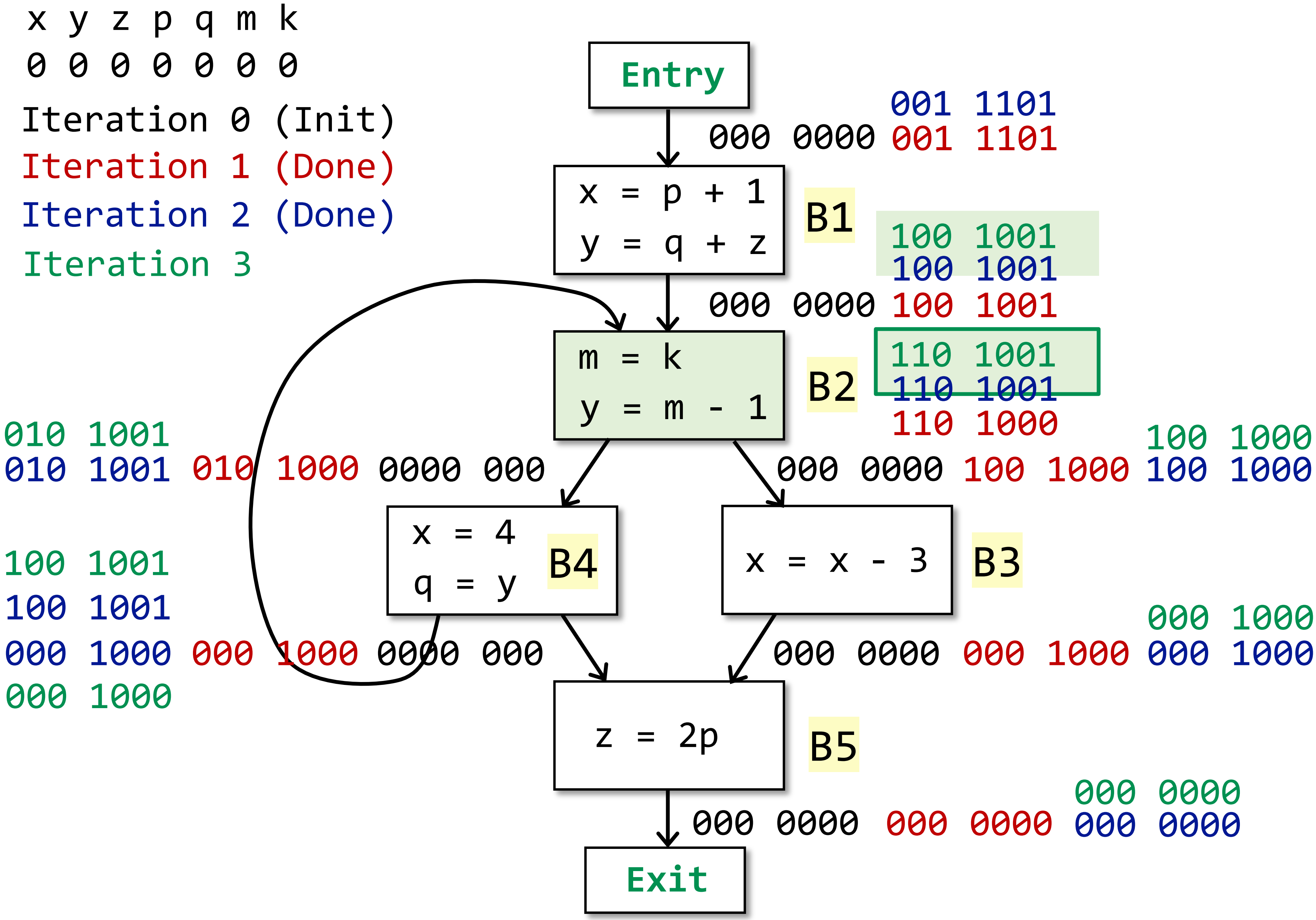
x y z p q m k
0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2 (Done)
Iteration 3
Entry
001 1101
000 0000 001 1101
x = p + 1
y = q + z
B1
100 1001
000 0000 100 1001
m = k
y = m - 1
B2
110 1001
110 1000
010 1001
010 1001 010 1000 0000 000
100 1000
000 0000 100 1000 100 1000
x = 4
q = y
B4
x = x - 3
B3
100 1001
100 1001
000 1000 000 1000 0000 000
000 1000
000 1000
000 0000 000 1000 000 1000
000 1000
z = 2p
B5
000 0000
000 0000 000 0000 000 0000
Exit


x y z p q m k
0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2 (Done)
Iteration 3
Entry
000 0000 001 1101 001 1101
x = p + 1
y = q + z
B1
000 0000 100 1001 100 1001
m = k
y = m - 1
B2
110 1000 110 1001
010 1001 010 1001 010 1000 0000 000
000 0000 100 1000 100 1000 100 1000
x = 4
q = y
B4
x = x - 3
B3
100 1001
100 1001
000 1000 000 1000 0000 000
000 1000
000 0000 000 1000 000 1000 000 1000
z = 2p
B5
000 0000 000 0000 000 0000 000 0000
Exit

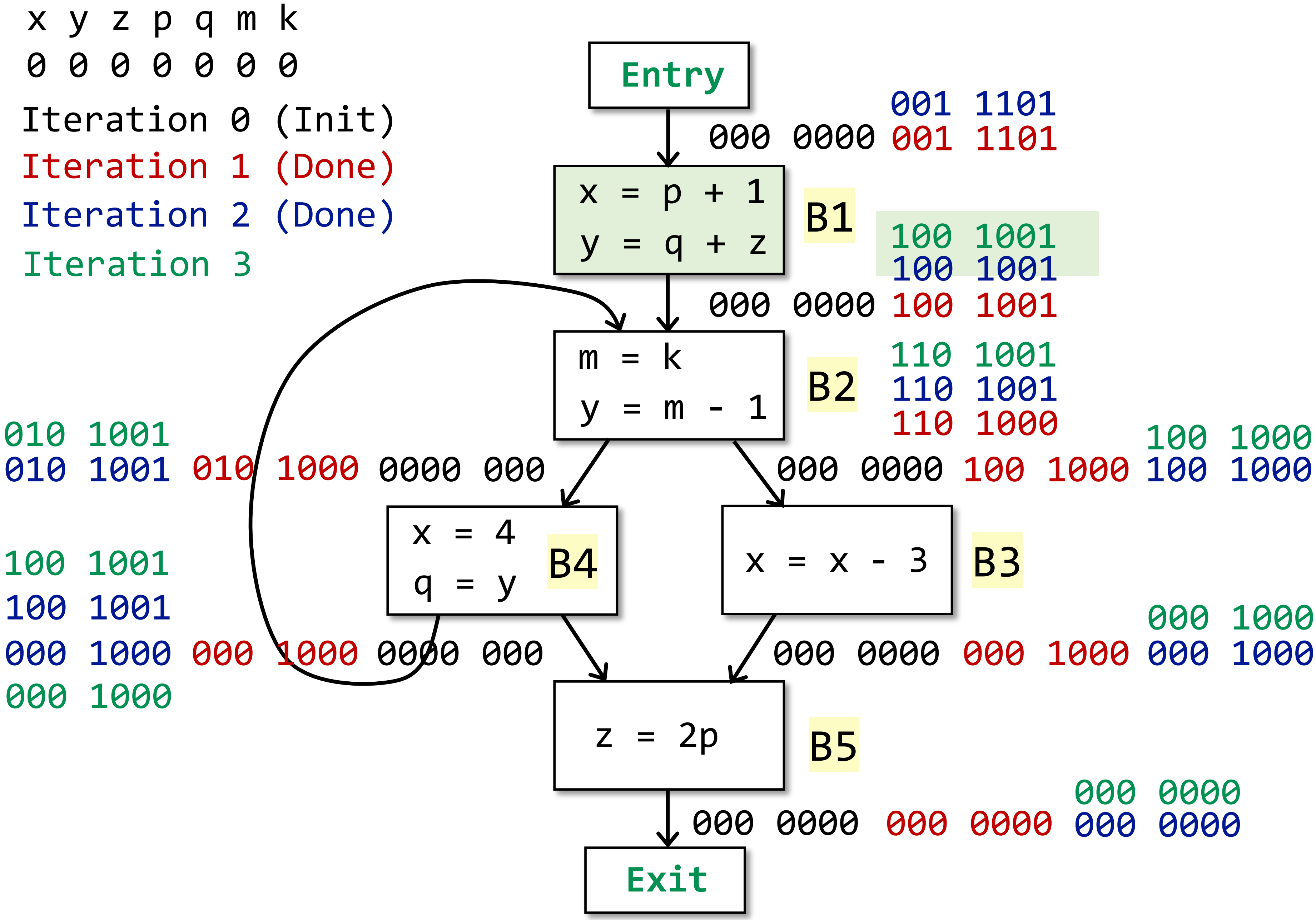

x y z p q m k
0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2 (Done)
Iteration 3
Entry
001 1101
000 0000 001 1101
x = p + 1
y = q + z
B1
100 1001
000 0000 100 1001
m = k
y = m - 1
B2
110 1001
110 1000
U
010 1001
010 1001 010 1000 0000 000
000 0000 100 1000 100 1000
100 1000
x = 4
q = y
B4
x = x - 3
B3
100 1001
100 1001
000 1000 000 1000 0000 000
000 1000
000 1000
000 0000 000 1000 000 1000
z = 2p
B5
000 0000
000 0000 000 0000 000 0000
Exit

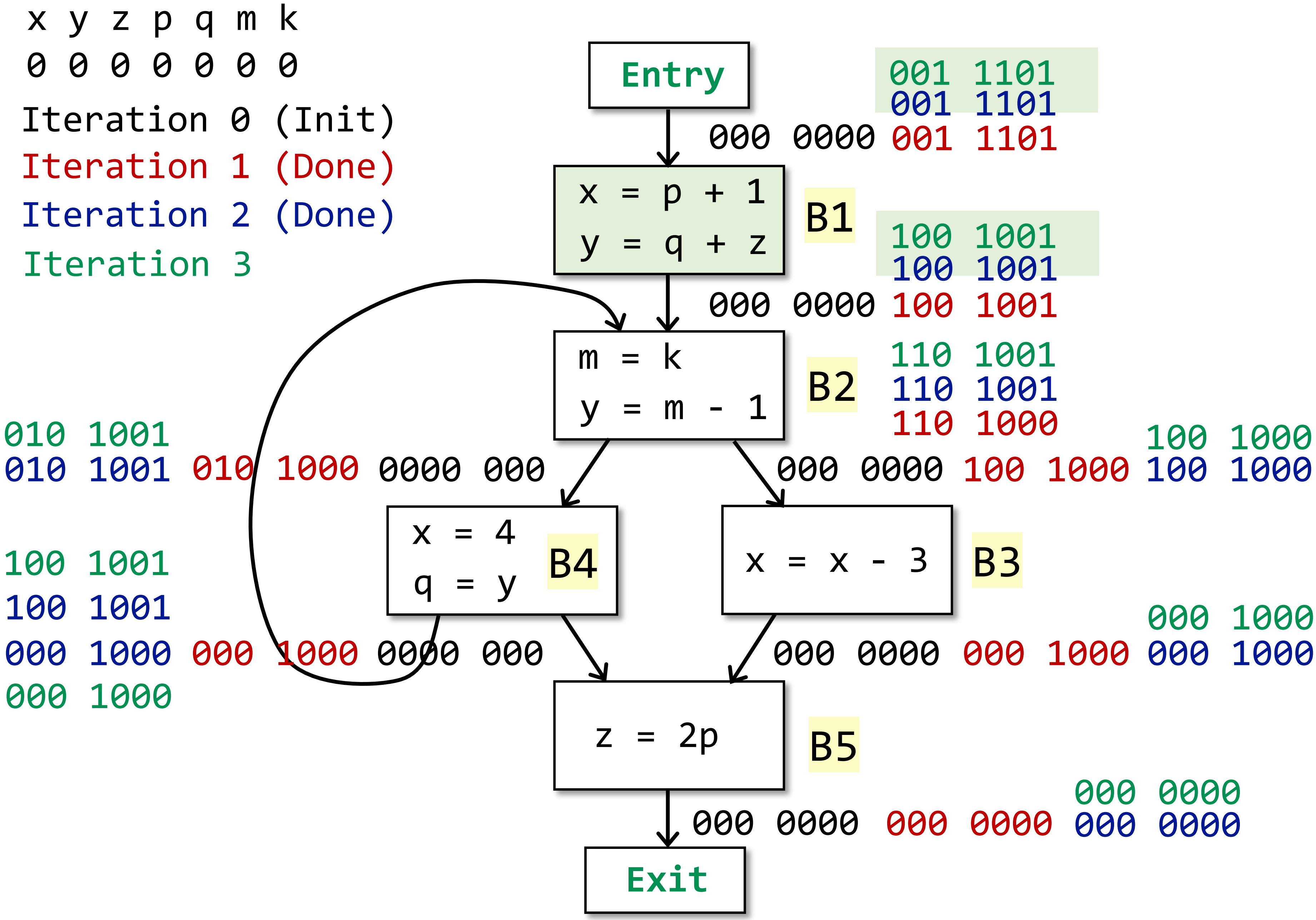

x y z p q m k
0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2 (Done)
Iteration 3
Entry
000 0000 001 1101
001 1101
x = p + 1
y = q + z
B1
000 0000 100 1001
100 1001
m = k
y = m - 1
B2
110 1001
110 1001
110 1000
U
010 1001
010 1001 010 1000 0000 000
000 0000 100 1000 100 1000
100 1000
x = 4
q = y
B4
x = x - 3
B3
100 1001
100 1001
000 1000 000 1000 0000 000
000 1000
000 1000
000 0000 000 1000 000 1000
z = 2p
B5
000 0000 000 0000 000 0000
000 0000
Exit

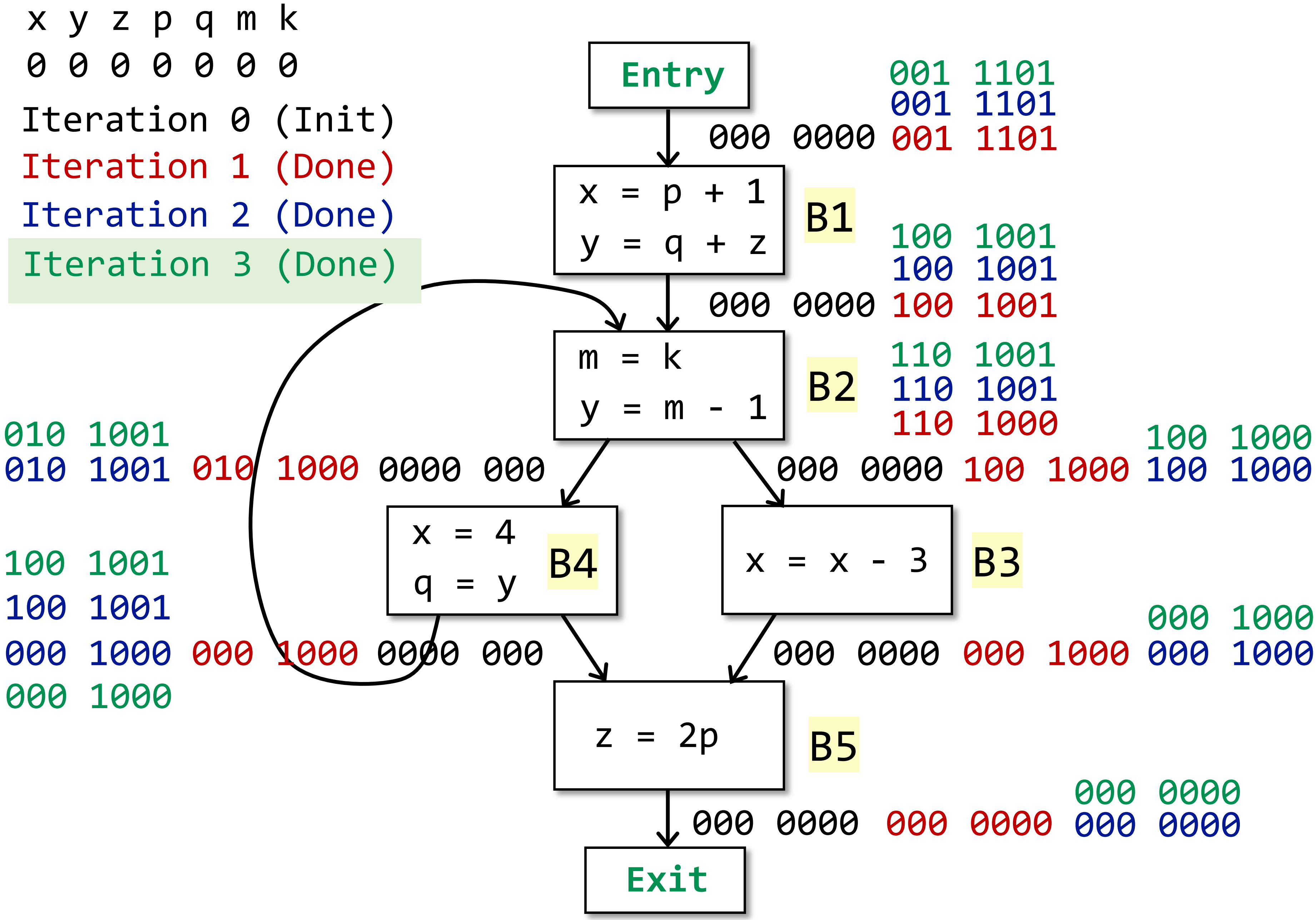

x y z p q m k
0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2 (Done)
Iteration 3
Entry
000 0000 001 1101
001 1101
x = p + 1
y = q + z
B1
100 1001
000 0000 100 1001
m = k
y = m - 1
B2
110 1001
110 1001
110 1000
010 1001
010 1001 010 1000 0000 000
000 0000 100 1000 100 1000
100 1000
x = 4
q = y
B4
x = x - 3
B3
100 1001
100 1001
000 1000 000 1000 0000 000
000 1000
000 1000
000 0000 000 1000 000 1000
z = 2p
B5
000 0000
000 0000 000 0000 000 0000
Exit

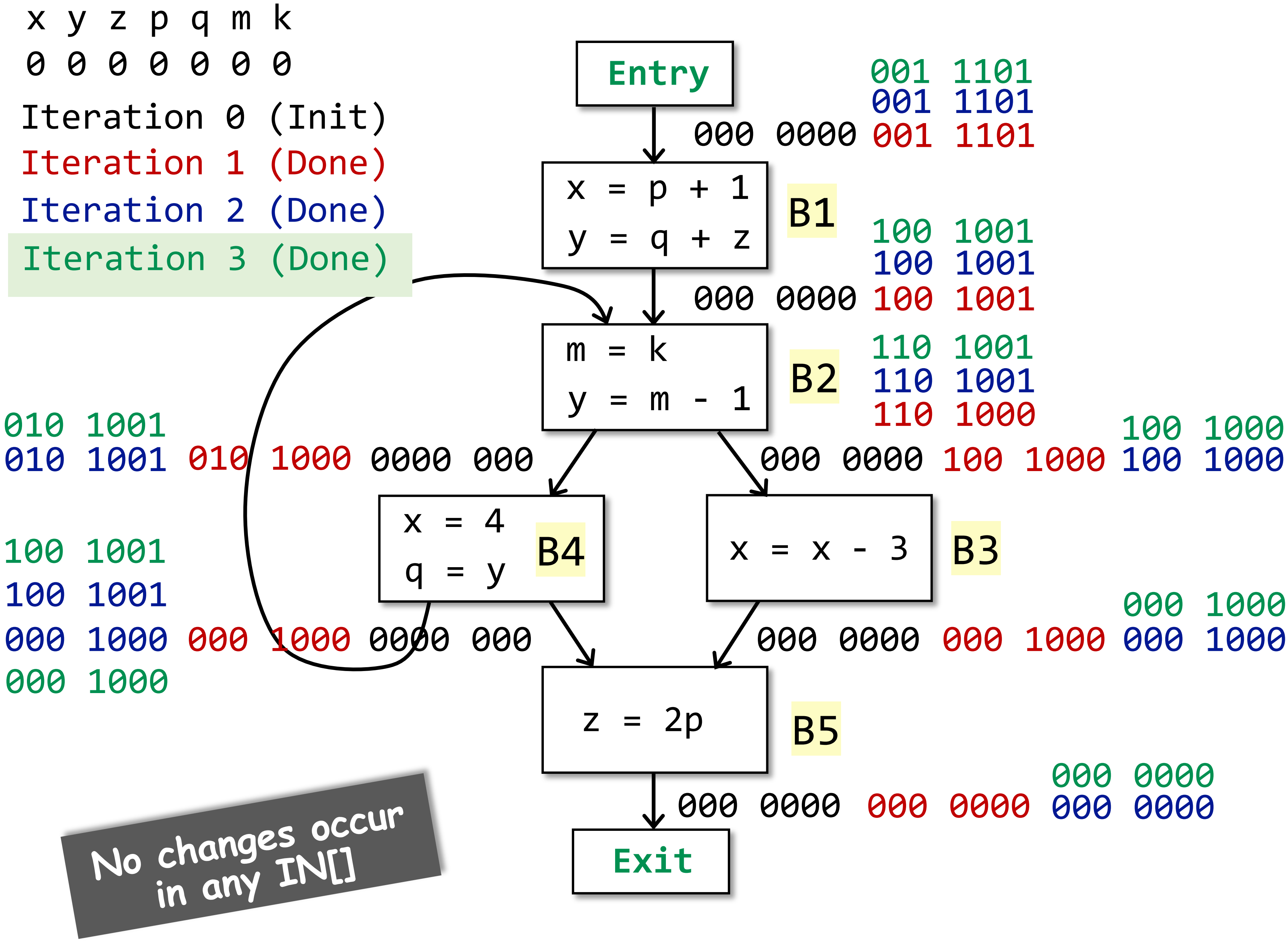


x y z p q m k
0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2 (Done)
Iteration 3
Entry
000 0000 001 1101
001 1101
x = p + 1
y = q + z
B1
100 1001
100 1001
000 0000 100 1001
m = k
y = m - 1
B2
110 1001
110 1001
110 1000
010 1001
010 1001 010 1000 0000 000
000 0000 100 1000 100 1000
100 1000
x = 4
q = y
B4
x = x - 3
B3
100 1001
100 1001
000 1000 000 1000 0000 000
000 1000
000 1000
000 0000 000 1000 000 1000
z = 2p
B5
000 0000 000 0000 000 0000
000 0000
Exit

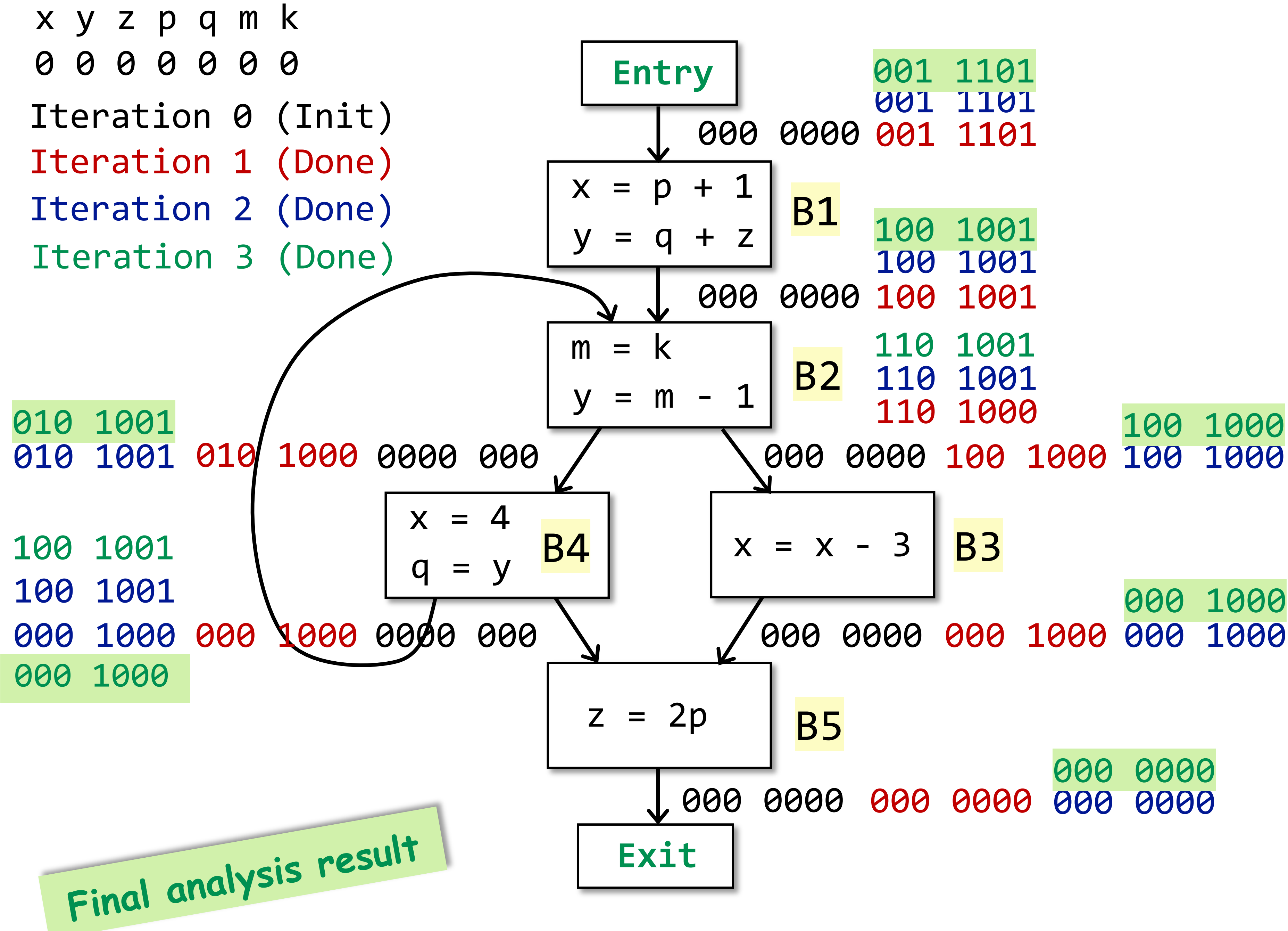

x y z p q m k
0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2 (Done)
Iteration 3
Entry
000 0000
001 1101
001 1101
x = p + 1
y = q + z
B1
100 1001
100 1001
000 0000 100 1001
m = k
y = m - 1
B2
110 1001
110 1001
110 1000
010 1001
010 1001 010 1000 0000 000
000 0000 100 1000
100 1000
100 1000
x = 4
q = y
B4
x = x - 3
B3
100 1001
100 1001
000 1000 000 1000 0000 000
000 1000
000 1000
000 0000 000 1000 000 1000
z = 2p
B5
000 0000
000 0000 000 0000 000 0000
Exit


x y z p q m k
0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2 (Done)
Iteration 3
Entry
001 1101
001 1101
000 0000 001 1101
x = p + 1
y = q + z
B1
100 1001
100 1001
000 0000 100 1001
m = k
y = m - 1
B2
110 1001
110 1001
110 1000
010 1001
010 1001 010 1000 0000 000
100 1000
000 0000 100 1000 100 1000
x = 4
q = y
B4
x = x - 3
B3
100 1001
100 1001
000 1000 000 1000 0000 000
000 1000
000 1000
000 0000 000 1000 000 1000
z = 2p
B5
000 0000
000 0000 000 0000 000 0000
Exit


x y z p q m k
0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2 (Done)
Iteration 3 (Done)
Entry
001 1101
001 1101
000 0000 001 1101
x = p + 1
y = q + z
B1
100 1001
100 1001
000 0000 100 1001
m = k
y = m - 1
B2
110 1001
110 1001
110 1000
010 1001
010 1001 010 1000 0000 000
100 1000
000 0000 100 1000 100 1000
x = 4
q = y
B4
x = x - 3
B3
100 1001
100 1001
000 1000 000 1000 0000 000
000 1000
000 1000
000 0000 000 1000 000 1000
z = 2p
B5
000 0000
000 0000 000 0000 000 0000
Exit



x y z p q m k
0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2 (Done)
Iteration 3 (Done)
Entry
001 1101
001 1101
000 0000 001 1101
x = p + 1
y = q + z
B1
100 1001
100 1001
000 0000 100 1001
m = k
y = m - 1
B2
110 1001
110 1001
110 1000
010 1001
010 1001 010 1000 0000 000
100 1000
000 0000 100 1000 100 1000
x = 4
q = y
B4
x = x - 3
B3
100 1001
100 1001
000 1000 000 1000 0000 000
000 1000
000 1000
000 0000 000 1000 000 1000
z = 2p
B5
000 0000
000 0000 000 0000 000 0000
Exit
No changes occur in any IN[]


x y z p q m k
0 0 0 0 0 0 0
Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2 (Done)
Iteration 3 (Done)
Entry
001 1101
001 1101
000 0000 001 1101
x = p + 1
y = q + z
B1
100 1001
100 1001
000 0000 100 1001
m = k
y = m - 1
B2
110 1001
110 1001
110 1000
010 1001
010 1001 010 1000 0000 000
100 1000
000 0000 100 1000 100 1000
x = 4
q = y
B4
x = x - 3
B3
100 1001
100 1001
000 1000 000 1000 0000 000
000 1000
000 1000
000 0000 000 1000 000 1000
z = 2p
B5
000 0000
000 0000 000 0000 000 0000
Exit
Final analysis result

Data Flow Analysis Applications

(I) Reaching Definitions Analysis

(II) Live Variables Analysis

(III) Available Expressions Analysis

Available Expressions Analysis

An expression x *op* y is available at program point p if (1) **all** paths from the entry to p **must** pass through the evaluation of x *op* y, and (2) after the last evaluation of x *op* y, there is no redefinition of x or y

Available Expressions Analysis

An expression x *op* y is available at program point p if (1) **all** paths from the entry to p **must** pass through the evaluation of x *op* y, and (2) after the last evaluation of x *op* y, there is no redefinition of x or y

- This definition means at program p, we can replace expression x *op* y by the result of its last evaluation
- The information of available expressions can be used for detecting global common subexpressions.

Understanding Available Expressions Analysis

Abstraction

- Data Flow Values/Facts
  - All the expressions in a program
  - Can be represented by bit vectors

e.g., E1, E2, E3, E4, …, E100 (100 expressions)

$$\underbrace{00000\ldots0}_{\text{100 bits}}$$

Bit i from the left represents expression Ei

Understanding Available Expressions Analysis

Safe-approximation

IN = { a + b }

a = x *op* y

Understanding Available Expressions Analysis

Safe-approximation

IN = { a + b }

```
a = x op y
```

- Add to OUT the expression x *op* y (gen)
- Delete from IN any expression involving variable a (kill)

Understanding Available Expressions Analysis

Safe-approximation

IN = { a + b }

a = x *op* y

OUT = { x *op* y }

- Add to OUT the expression x *op* y (gen)
- Delete from IN any expression involving variable a (kill)

Understanding Available Expressions Analysis

Safe-approximation

IN = { a + b }

a = x *op* y

OUT = { x *op* y }

- Add to OUT the expression x *op* y (gen)
- Delete from IN any expression involving variable a (kill)

$$OUT[B] = gen_B \cup (IN[B] - kill_B)$$

Understanding Available Expressions Analysis

Safe-approximation

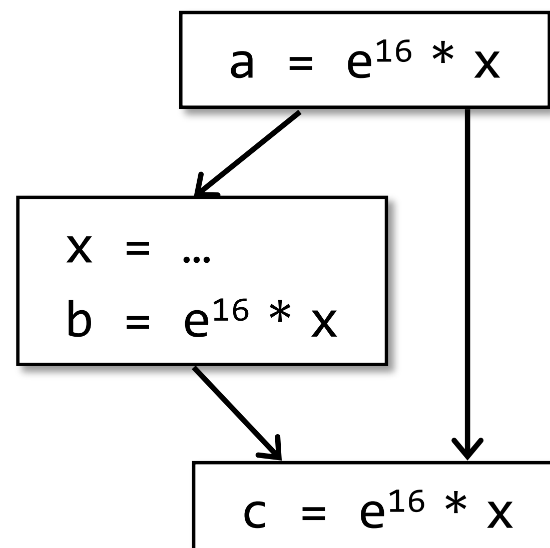
IN = { a + b }

a = x *op* y

OUT = { x *op* y }

- Add to OUT the expression x *op* y (gen)
- Delete from IN any expression involving variable a (kill)

$$OUT[B] = gen_B \cup (IN[B] - kill_B)$$



Understanding Available Expressions Analysis

Safe-approximation

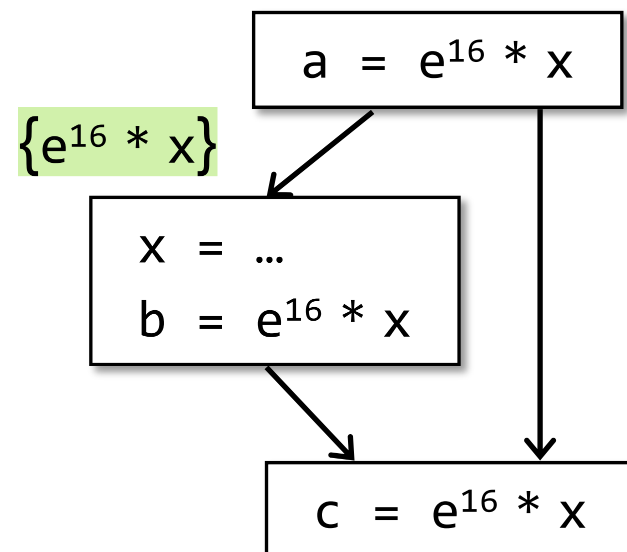
IN = { a + b }

a = x *op* y

OUT = { x *op* y }

- Add to OUT the expression x *op* y (gen)
- Delete from IN any expression involving variable a (kill)

$$OUT[B] = gen_B \cup (IN[B] - kill_B)$$



Understanding Available Expressions Analysis

Safe-approximation

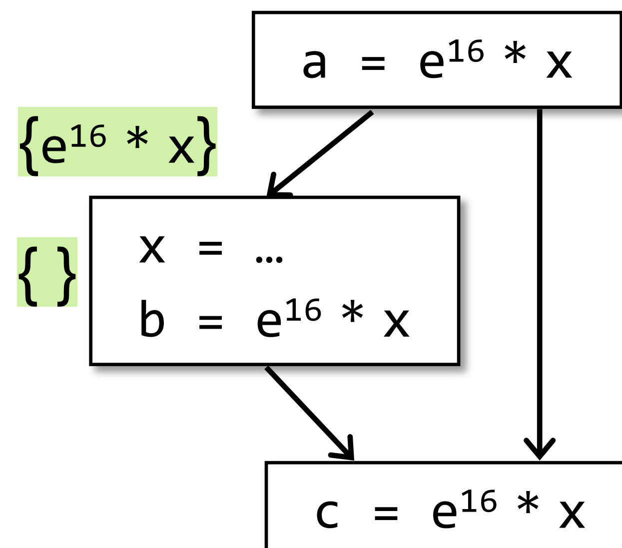
IN = { a + b }

a = x *op* y

OUT = { x *op* y }

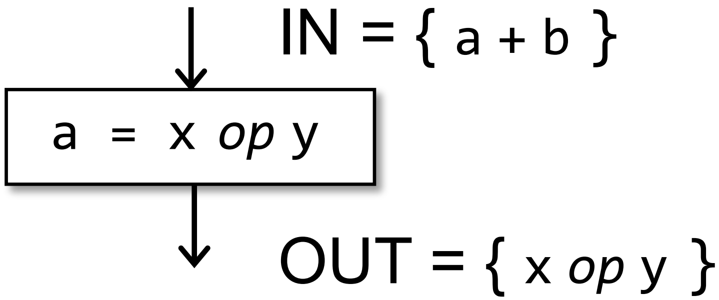
- Add to OUT the expression x *op* y (gen)
- Delete from IN any expression involving variable a (kill)

$$OUT[B] = gen_B \cup (IN[B] - kill_B)$$



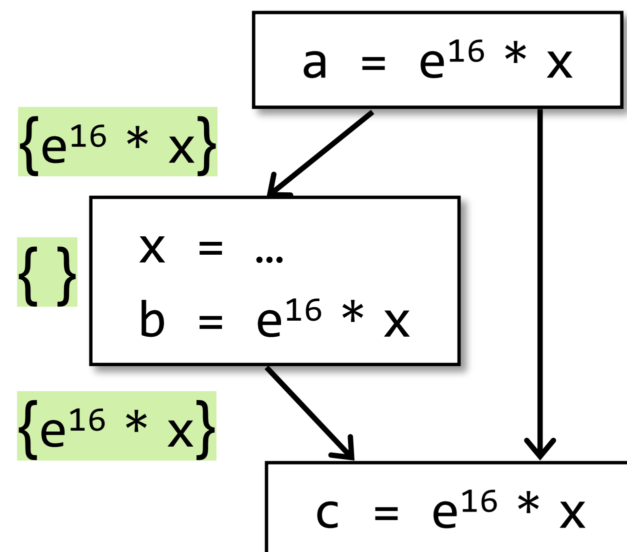
Understanding Available Expressions Analysis

Safe-approximation



- Add to OUT the expression x *op* y (gen)
- Delete from IN any expression involving variable a (kill)

$\mathrm{OUT}[B] = gen_B \cup (\mathrm{IN}[B] - kill_B)$



Understanding Available Expressions Analysis

Safe-approximation

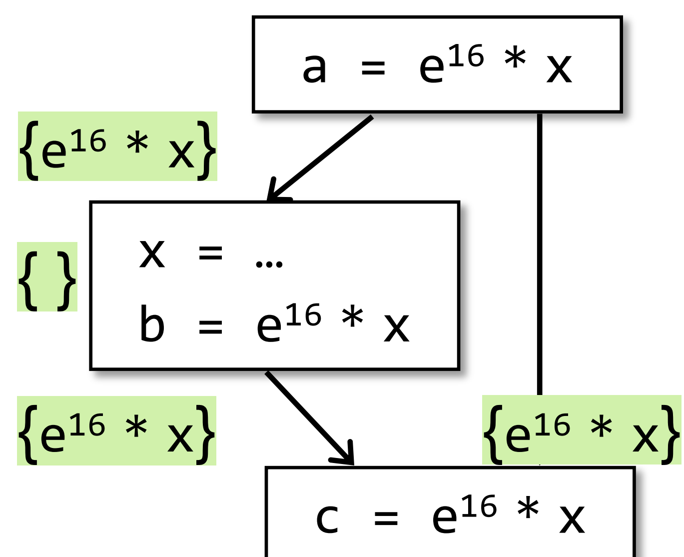
IN = { a + b }

$a = x \; op \; y$

OUT = { x *op* y }

- Add to OUT the expression x *op* y (gen)
- Delete from IN any expression involving variable a (kill)

$$OUT[B] = gen_B \cup (IN[B] - kill_B)$$



Understanding Available Expressions Analysis

Safe-approximation

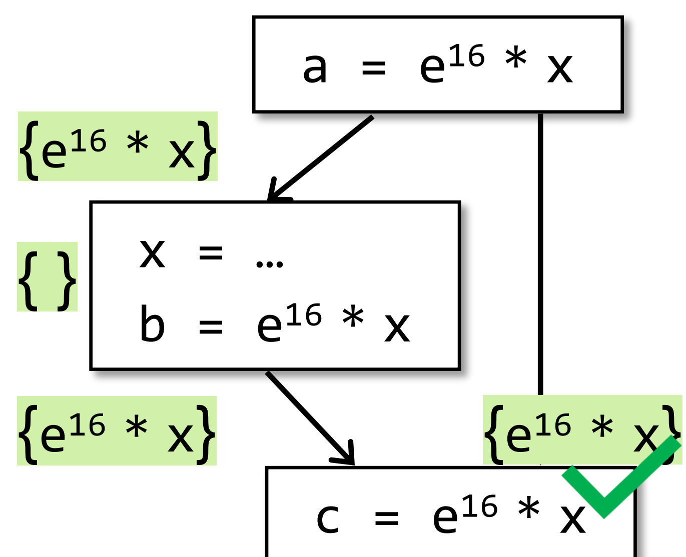
IN = { a + b }

a = x *op* y

OUT = { x *op* y }

- Add to OUT the expression x *op* y (gen)
- Delete from IN any expression involving variable a (kill)

$$\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B)$$



Understanding Available Expressions Analysis

Safe-approximation

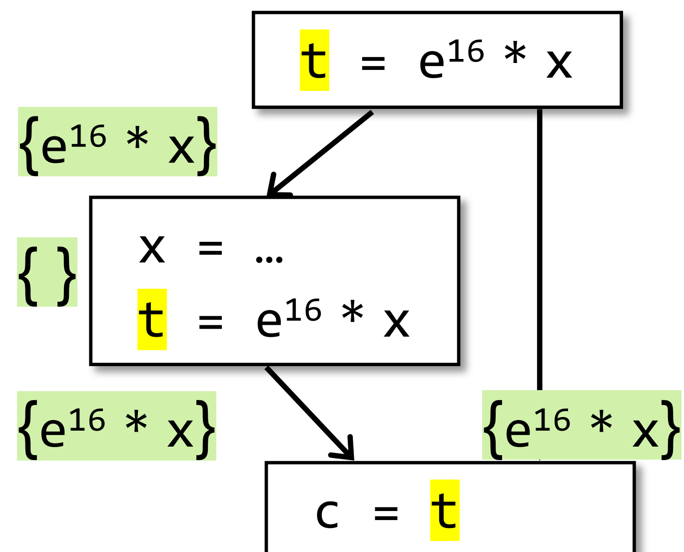
IN = { a + b }

a = x *op* y

OUT = { x *op* y }

- Add to OUT the expression x *op* y (gen)
- Delete from IN any expression involving variable a (kill)

$$OUT[B] = gen_B \cup (IN[B] - kill_B)$$



Understanding Available Expressions Analysis

Safe-approximation

IN = { a + b }

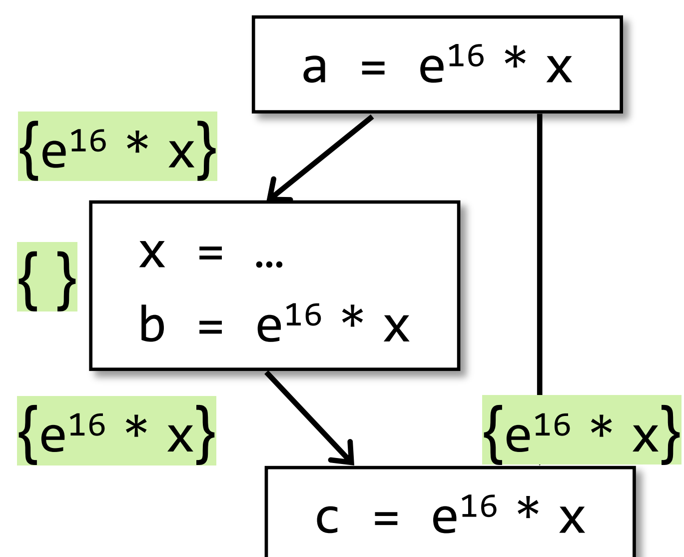
a = x *op* y

OUT = { x *op* y }

- Add to OUT the expression x *op* y (gen)
- Delete from IN any expression involving variable a (kill)

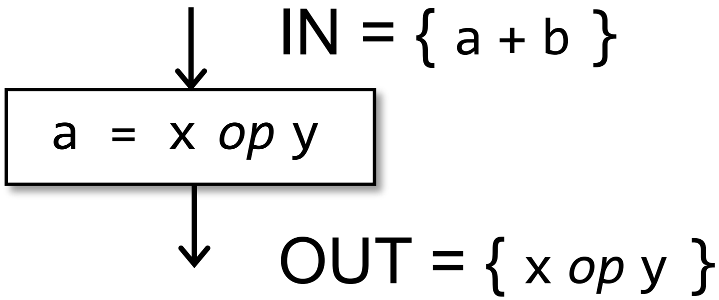
$$OUT[B] = gen_B \cup (IN[B] - kill_B)$$

$$IN[B] = \bigcap_{P \text{ a predecessor of } B} OUT[P]$$



Understanding Available Expressions Analysis

Safe-approximation

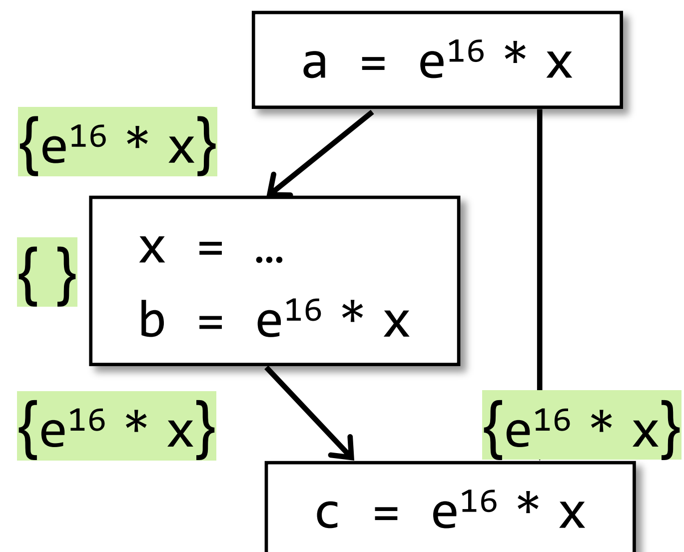


- Add to OUT the expression x *op* y (gen)
- Delete from IN any expression involving variable a (kill)

$\mathrm{OUT}[B] = gen_B \cup (\mathrm{IN}[B] - kill_B)$

$\mathrm{IN}[B] = \bigcap_{P \text{ a predecessor of } B} \mathrm{OUT}[P]$

All paths from entry to point p must pass through the evaluation of x *op* y



Understanding Available Expressions Analysis

Safe-approximation

IN = { a + b }

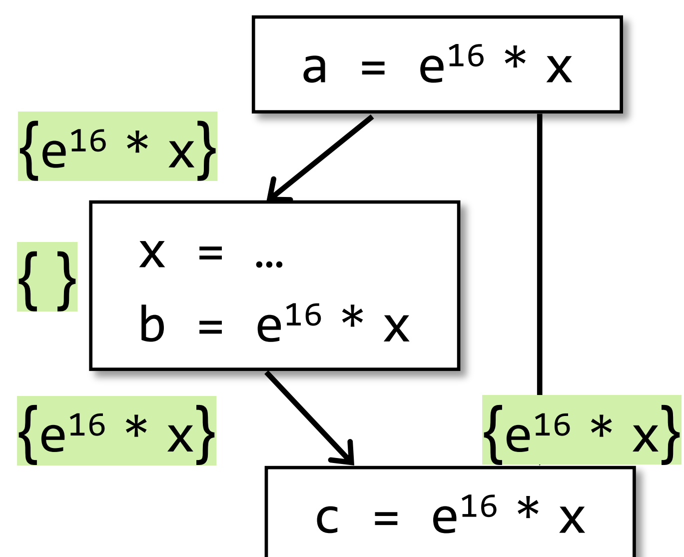
For safety of the analysis, it may report an expression as unavailable even if it is truly available (must analysis -> under-approximation)

- Add to OUT the expression x *op* y (gen)
- Delete from IN any expression involving variable a (kill)

$$OUT[B] = gen_B \cup (IN[B] - kill_B)$$

$$IN[B] = \bigcap_{P \text{ a predecessor of } B} OUT[P]$$

All paths from entry to point p must pass through the evaluation of x *op* y



Understanding Available Expressions Analysis

Safe-approximation

IN = { a + b }

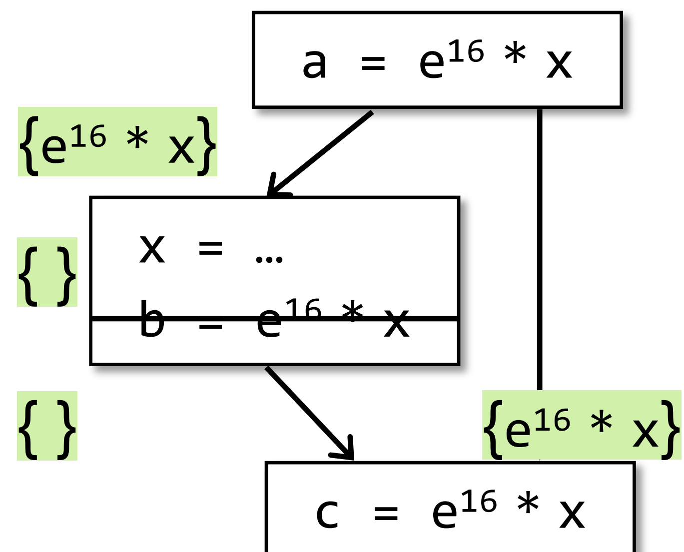
For safety of the analysis, it may report an expression as unavailable even if it is truly available (must analysis -> under-approximation)

- Add to OUT the expression x *op* y (gen)
- Delete from IN any expression involving variable a (kill)

$$OUT[B] = gen_B \cup (IN[B] - kill_B)$$

$$IN[B] = \bigcap_{P \text{ a predecessor of } B} OUT[P]$$

All paths from entry to point p must pass through the evaluation of x *op* y



Understanding Available Expressions Analysis

Safe-approximation

IN = { a + b }

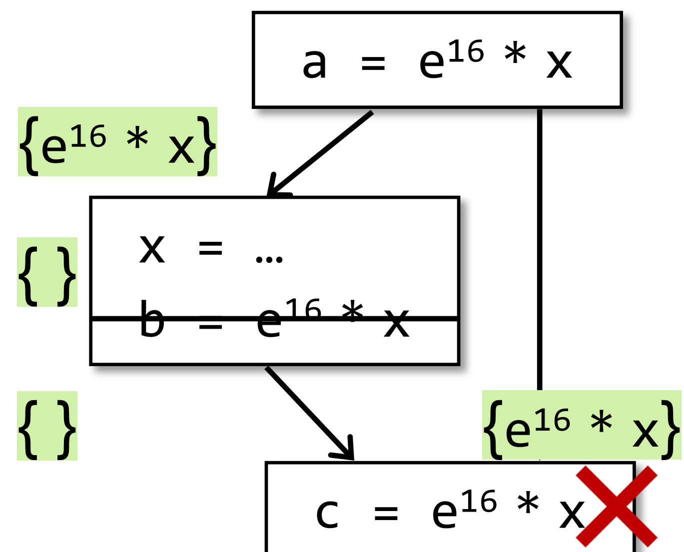
For safety of the analysis, it may report an expression as unavailable even if it is truly available (must analysis -> under-approximation)

- Add to OUT the expression x *op* y (gen)
- Delete from IN any expression involving variable a (kill)

$OUT[B] = gen_B \cup (IN[B] - kill_B)$

$IN[B] = \bigcap_{P\ a\ predecessor\ of\ B} OUT[P]$

All paths from entry to point p must pass through the evaluation of x *op* y



Understanding Available Expressions Analysis

Safe-approximation

IN = { a + b }

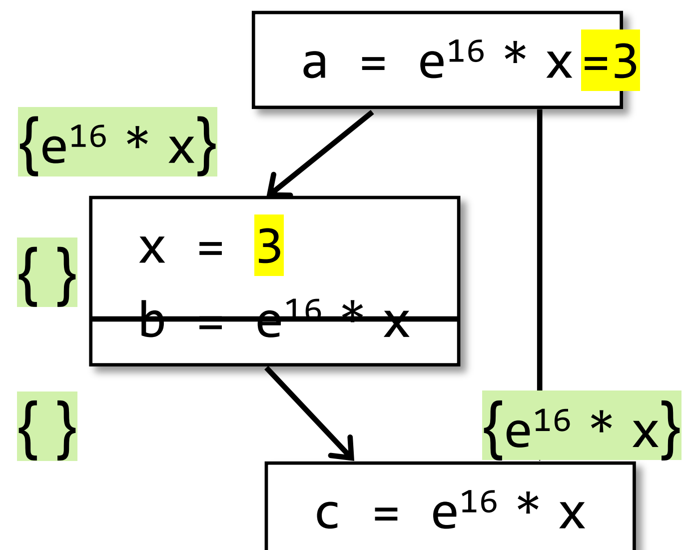
For safety of the analysis, it may report an expression as unavailable even if it is truly available (must analysis -> under-approximation)

- Add to OUT the expression x *op* y (gen)
- Delete from IN any expression involving variable a (kill)

$OUT[B] = gen_B \cup (IN[B] - kill_B)$

$IN[B] = \bigcap_{P \text{ a predecessor of } B} OUT[P]$

All paths from entry to point p must pass through the evaluation of x *op* y



Understanding Available Expressions Analysis

Safe-approximation

IN = { a + b }

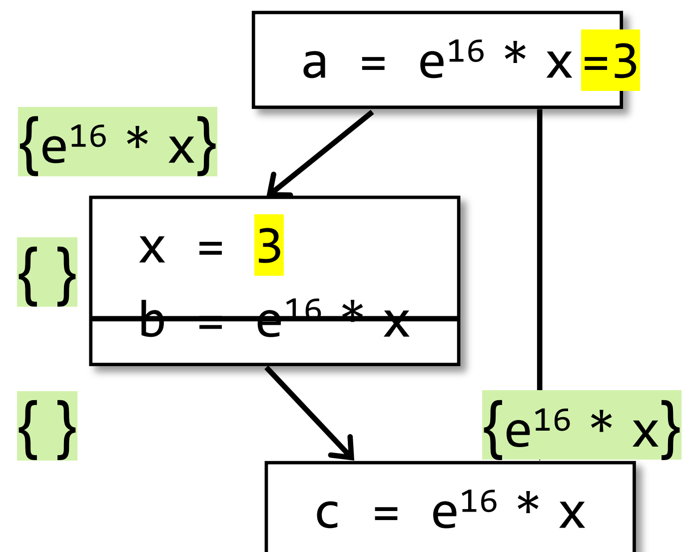
For safety of the analysis, it may report an expression as unavailable even if it is truly available (must analysis -> under-approximation)

- Add to OUT the expression x *op* y (gen)
- Delete from IN any expression involving variable a (kill)

$OUT[B] = gen_B \cup (IN[B] - kill_B)$

$IN[B] = \bigcap_{P \text{ a predecessor of } B} OUT[P]$

All paths from entry to point p must pass through the evaluation of x *op* y



Algorithm of Available Expressions Analysis

INPUT: CFG ($kill_B$ and gen_B computed for each basic block B)

OUTPUT: IN[B] and OUT[B] for each basic block B

METHOD:

```
OUT[entry] = ∅;
for (each basic block B\entry)
    OUT[B] = U;
while (changes to any OUT occur)
    for (each basic block B\entry) {
        IN[B] = ⋂_{P a predecessor of B} OUT[P];
        OUT[B] = gen_B U (IN[B] - kill_B);
    }
```

Algorithm of Available Expressions Analysis

INPUT: CFG ($kill_B$ and gen_B computed for each basic block B)

OUTPUT: IN[B] and OUT[B] for each basic block B

METHOD:

```
OUT[entry] = ∅;
for (each basic block B\entry)
    OUT[B] = U;
while (changes to any OUT occur)
    for (each basic block B\entry) {
        IN[B] = ∩_{P a predecessor of B} OUT[P];
        OUT[B] = gen_B U (IN[B] - kill_B);
    }
```

Algorithm of Available Expressions Analysis

INPUT: CFG ($kill_B$ and gen_B computed for each basic block B)

OUTPUT: IN[B] and OUT[B] for each basic block B

METHOD:

```
OUT[entry] = ∅;
for (each basic block B\entry)
    OUT[B] = U;
while (changes to any OUT occur)
    for (each basic block B\entry) {
        IN[B] = ∩_{P a predecessor of B} OUT[P];
        OUT[B] = gen_B U (IN[B] - kill_B);
    }
```

$$\text{IN}[B] = \bigcap_{P \text{ a predecessor of } B} \text{OUT}[P];$$

$$\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B);$$

Algorithm of Available Expressions Analysis

INPUT: CFG ($kill_B$ and gen_B computed for each basic block B)

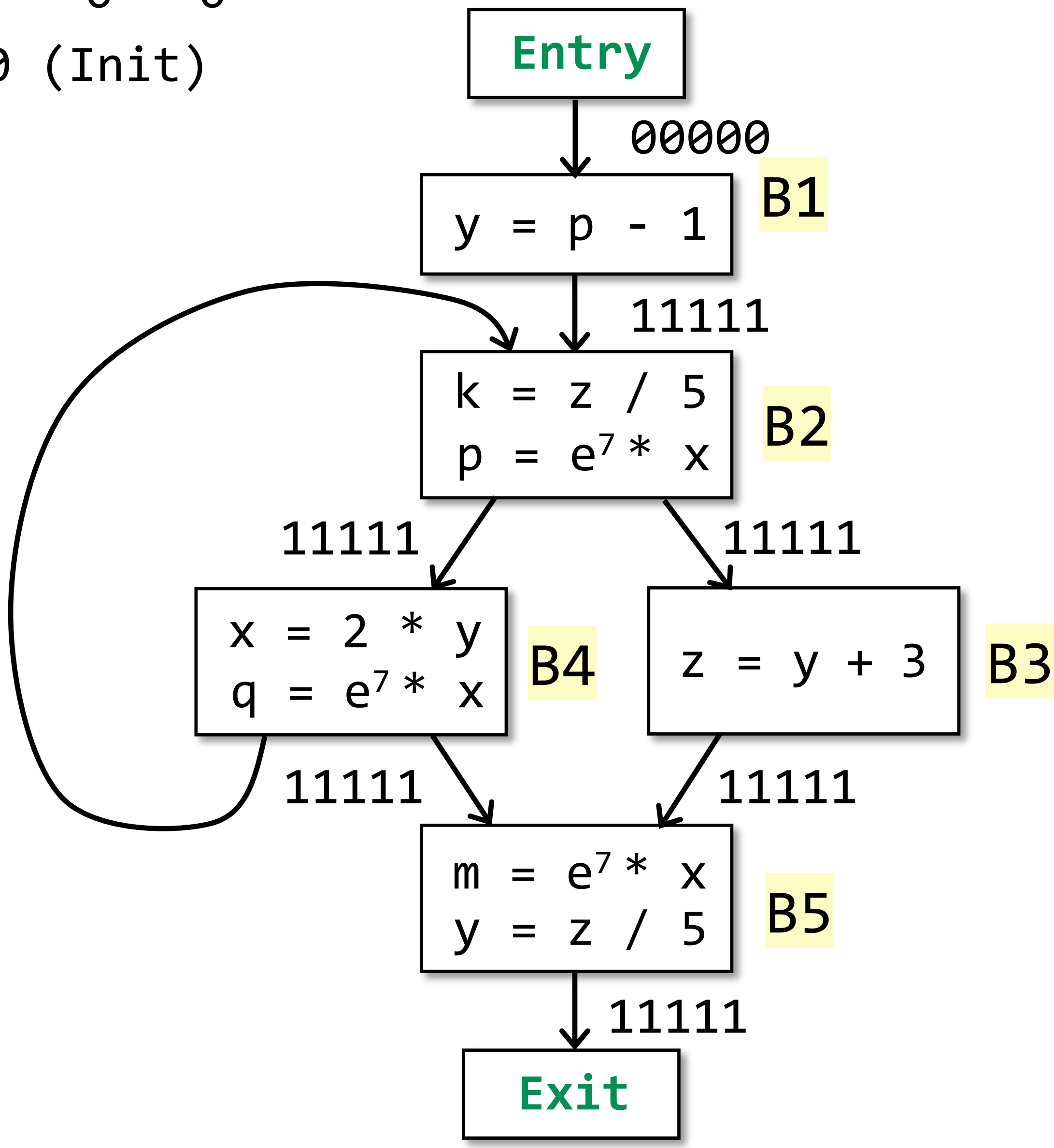
OUTPUT: IN[B] and OUT[B] for each basic block B

METHOD:

OUT[*entry*] = $\emptyset$;
for (each basic block $B\backslash entry$)
    OUT[B] = U;
while (changes to any OUT occur)
    for (each basic block $B\backslash entry$) {
        IN[B] = $\bigcap_{P \text{ a predecessor of } B}$ OUT[P];
        OUT[B] = gen_B U (IN[B] - $kill_B$);
    }

p-1	z/5	2*y	e^7*x	y+3
0	0	0	0	0

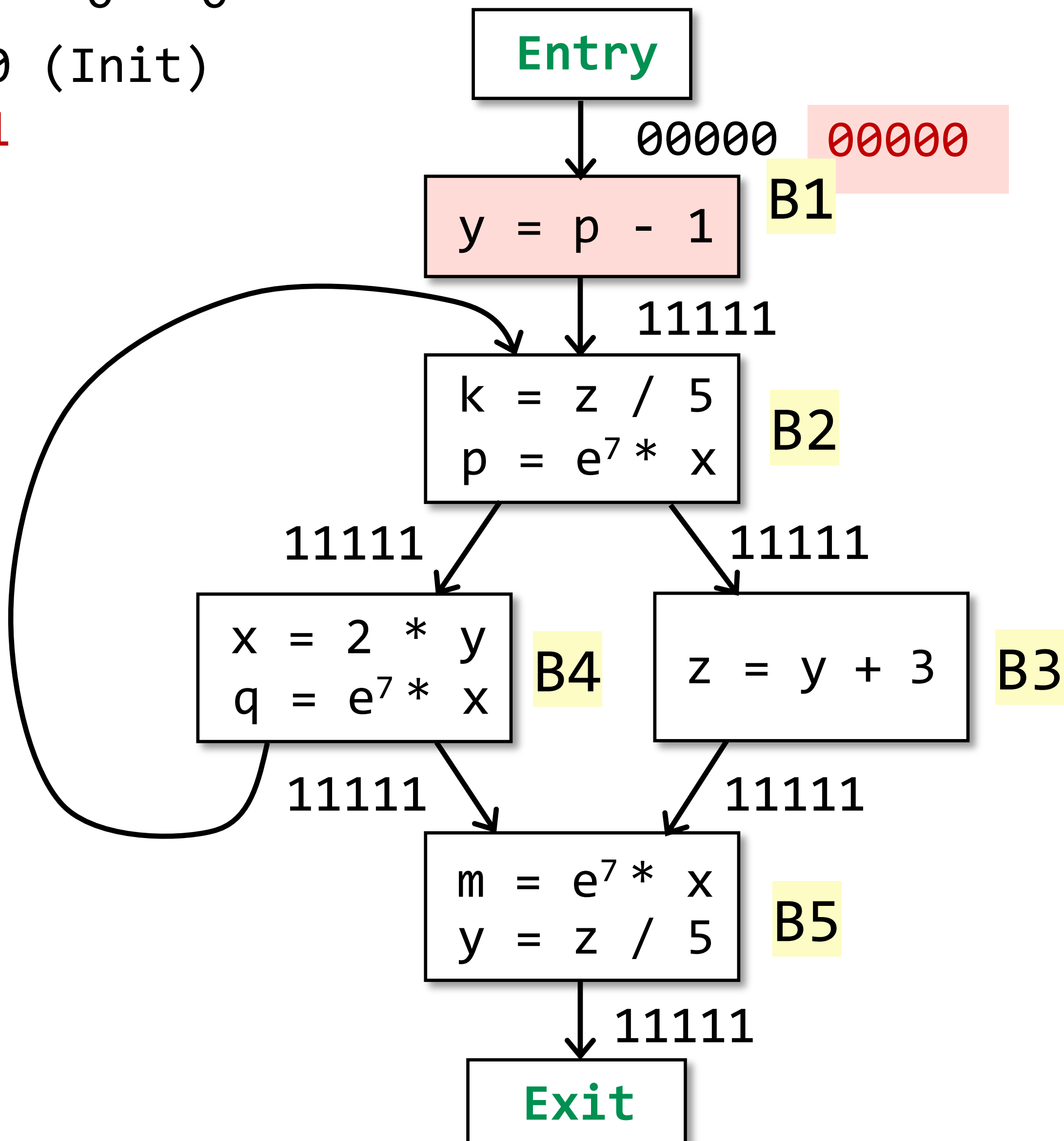
Iteration 0 (Init)



p-1	z/5	2*y	e^7*x	y+3
0	0	0	0	0

Iteration 0 (Init)

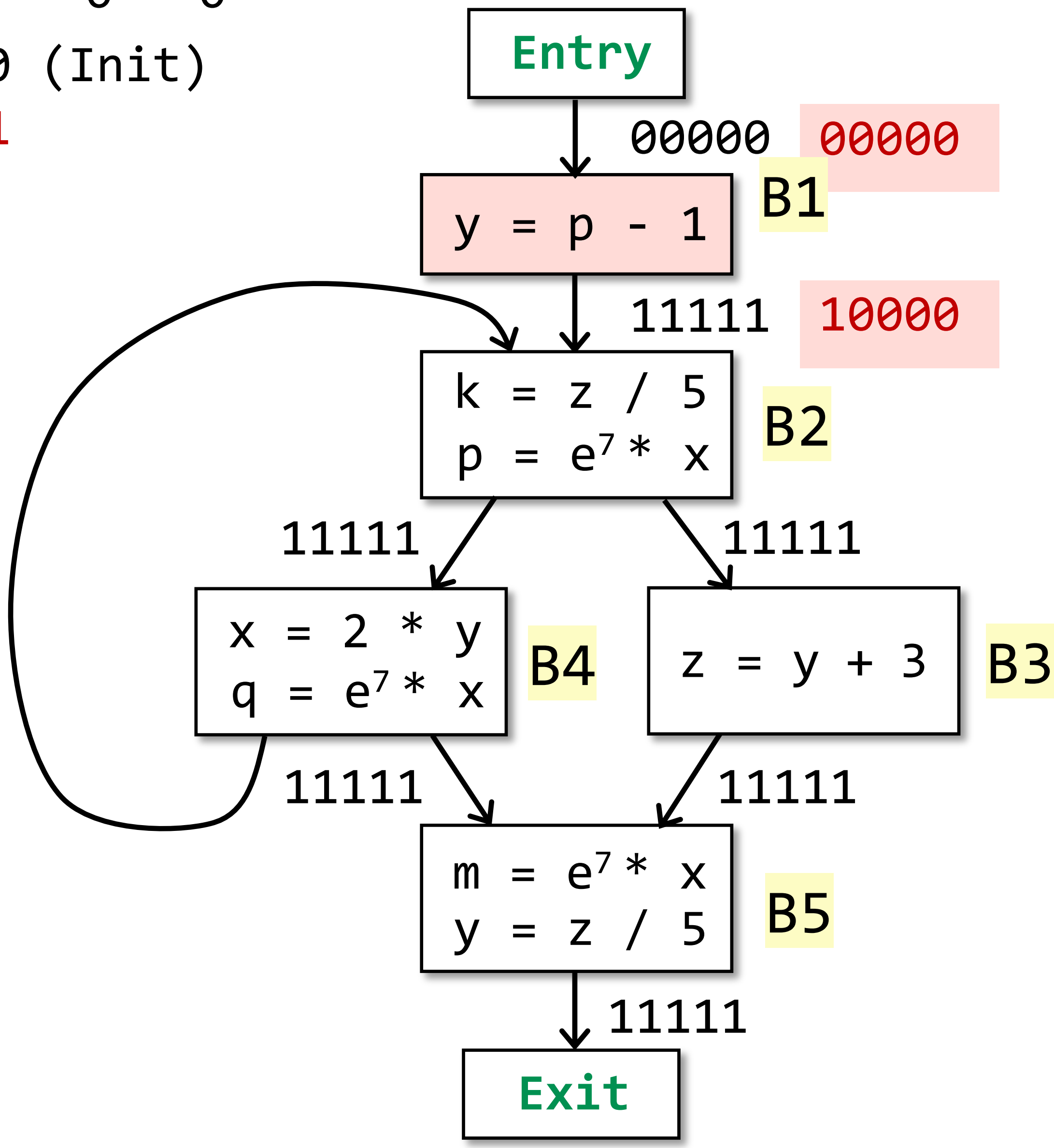
Iteration 1



p-1	z/5	2*y	e^7*x	y+3
0	0	0	0	0

Iteration 0 (Init)

Iteration 1

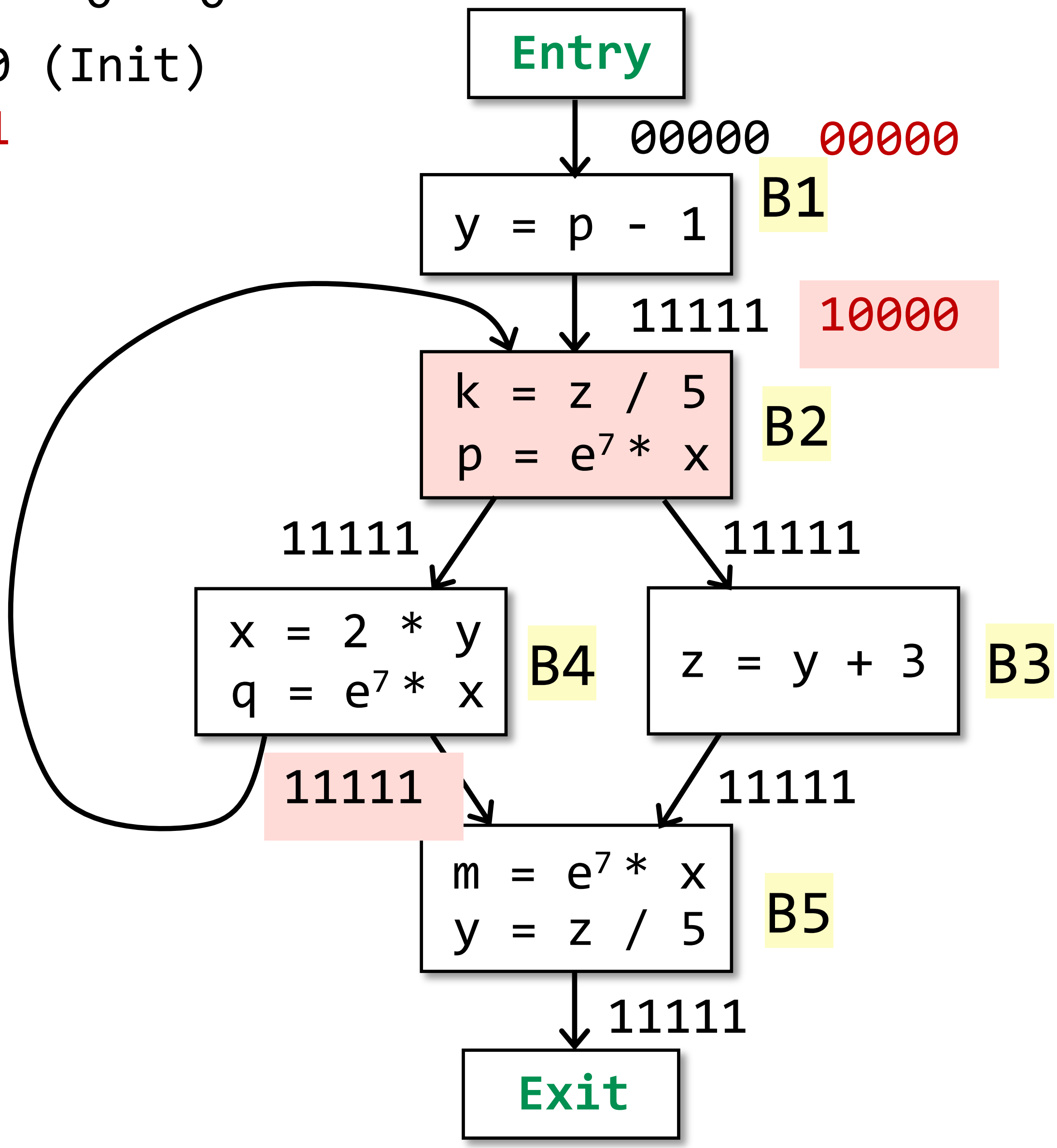


p-1 z/5 2*y e^7*x y+3

0 0 0 0 0

Iteration 0 (Init)

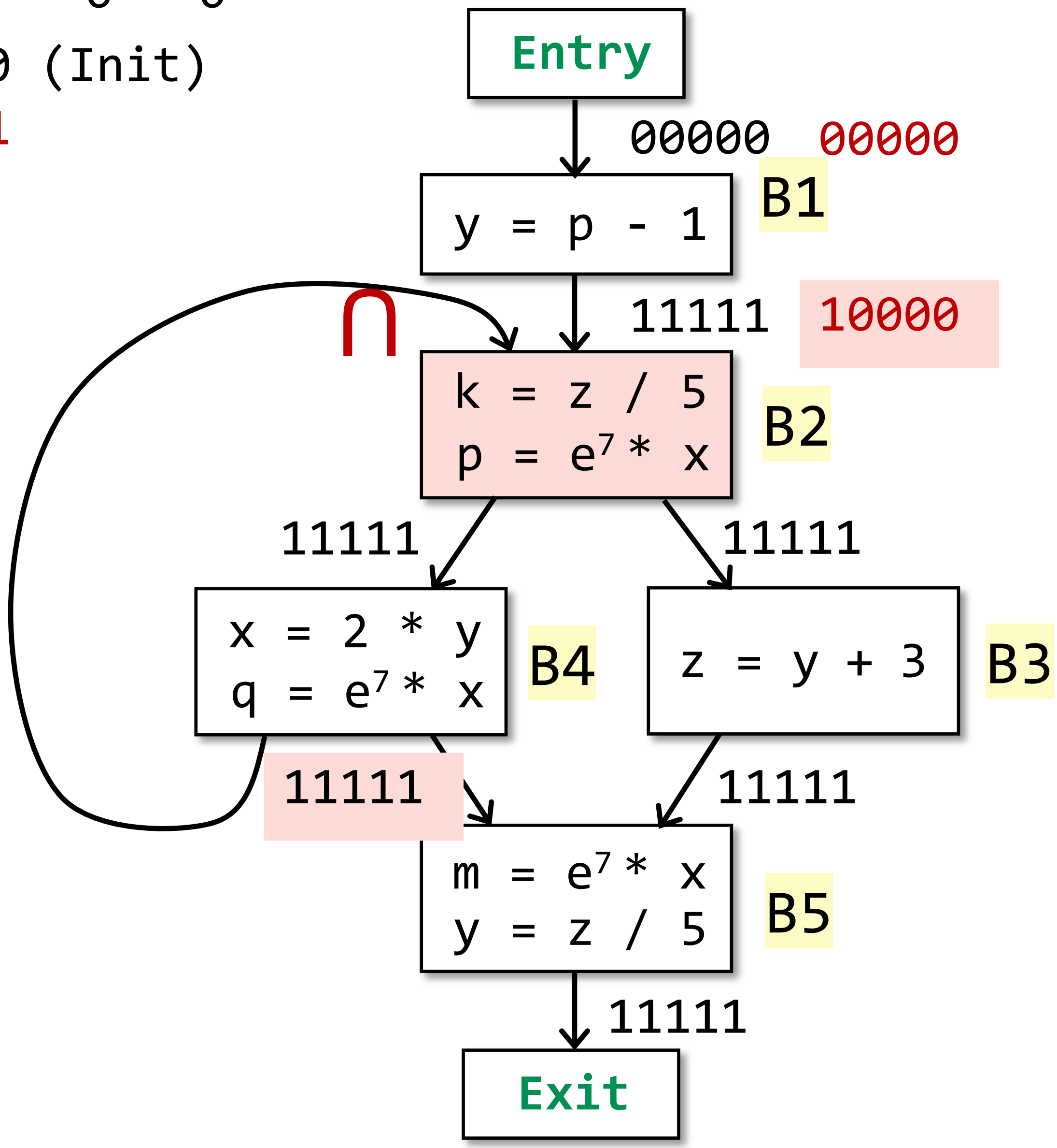
Iteration 1



p-1 z/5 2*y e⁷*x y+3
0 0 0 0 0

Iteration 0 (Init)

Iteration 1

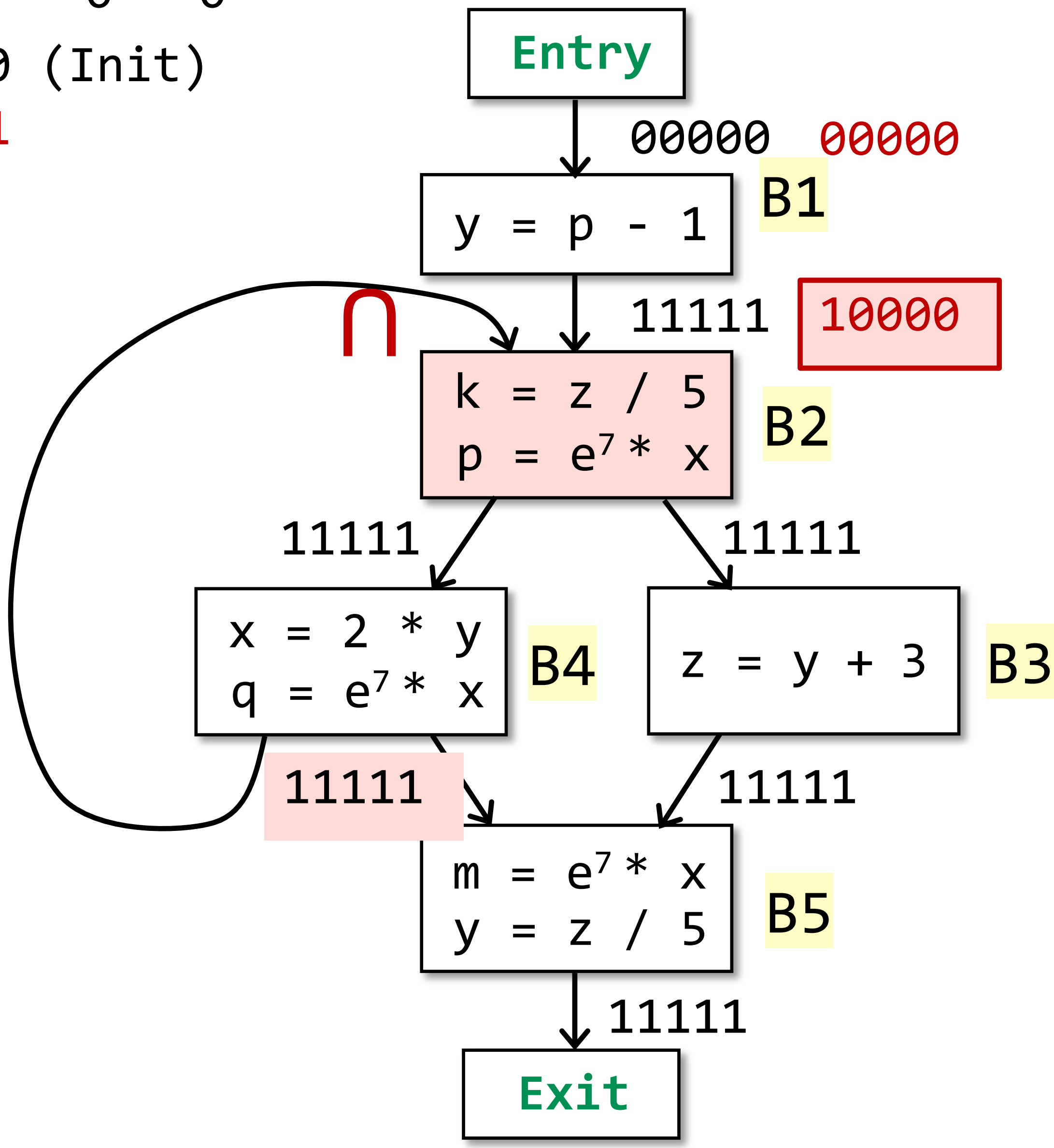


p-1 z/5 2*y e^7*x y+3

0 0 0 0 0

Iteration 0 (Init)

Iteration 1

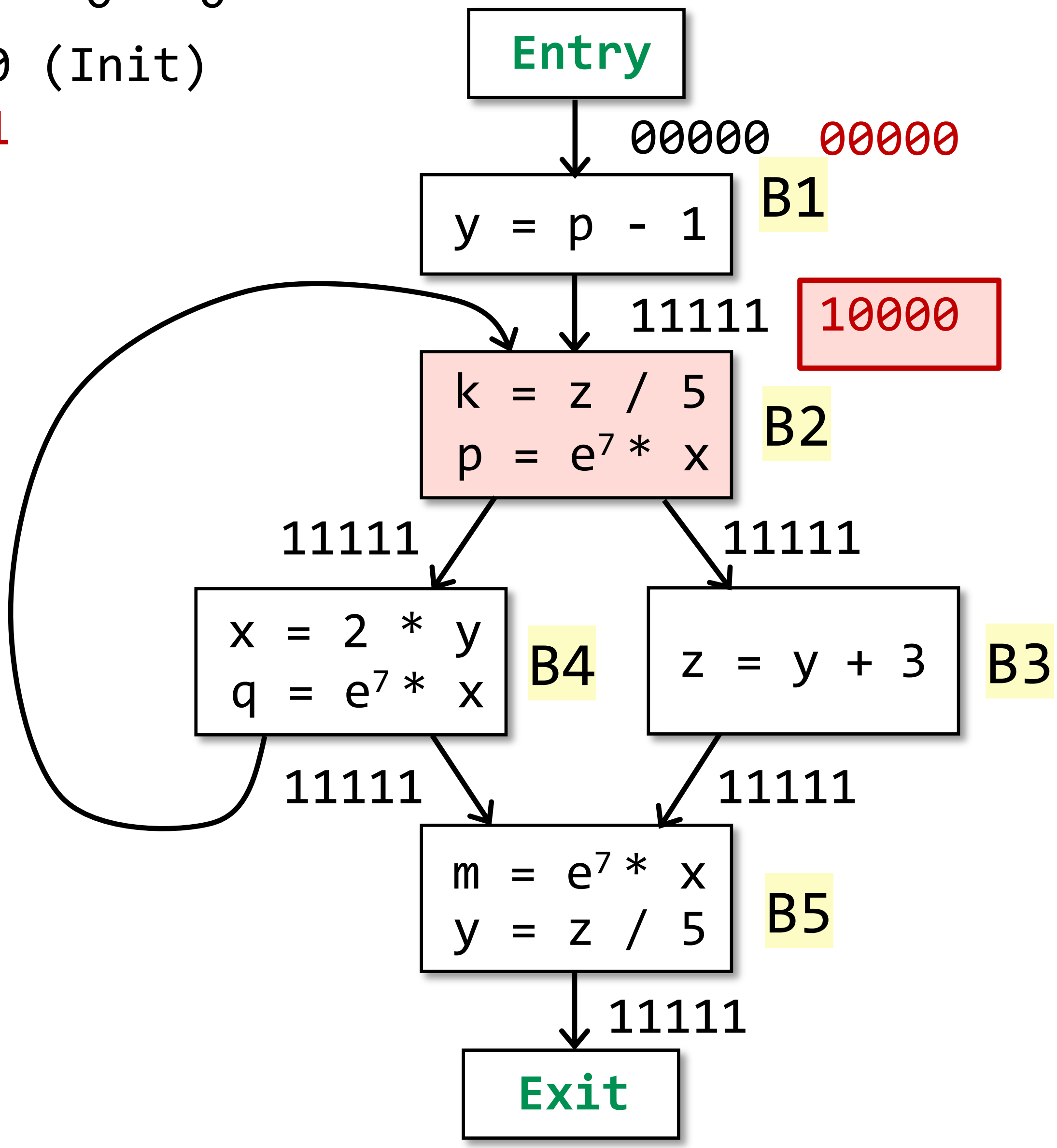


p-1 z/5 2*y e^7*x y+3

0 0 0 0 0

Iteration 0 (Init)

Iteration 1

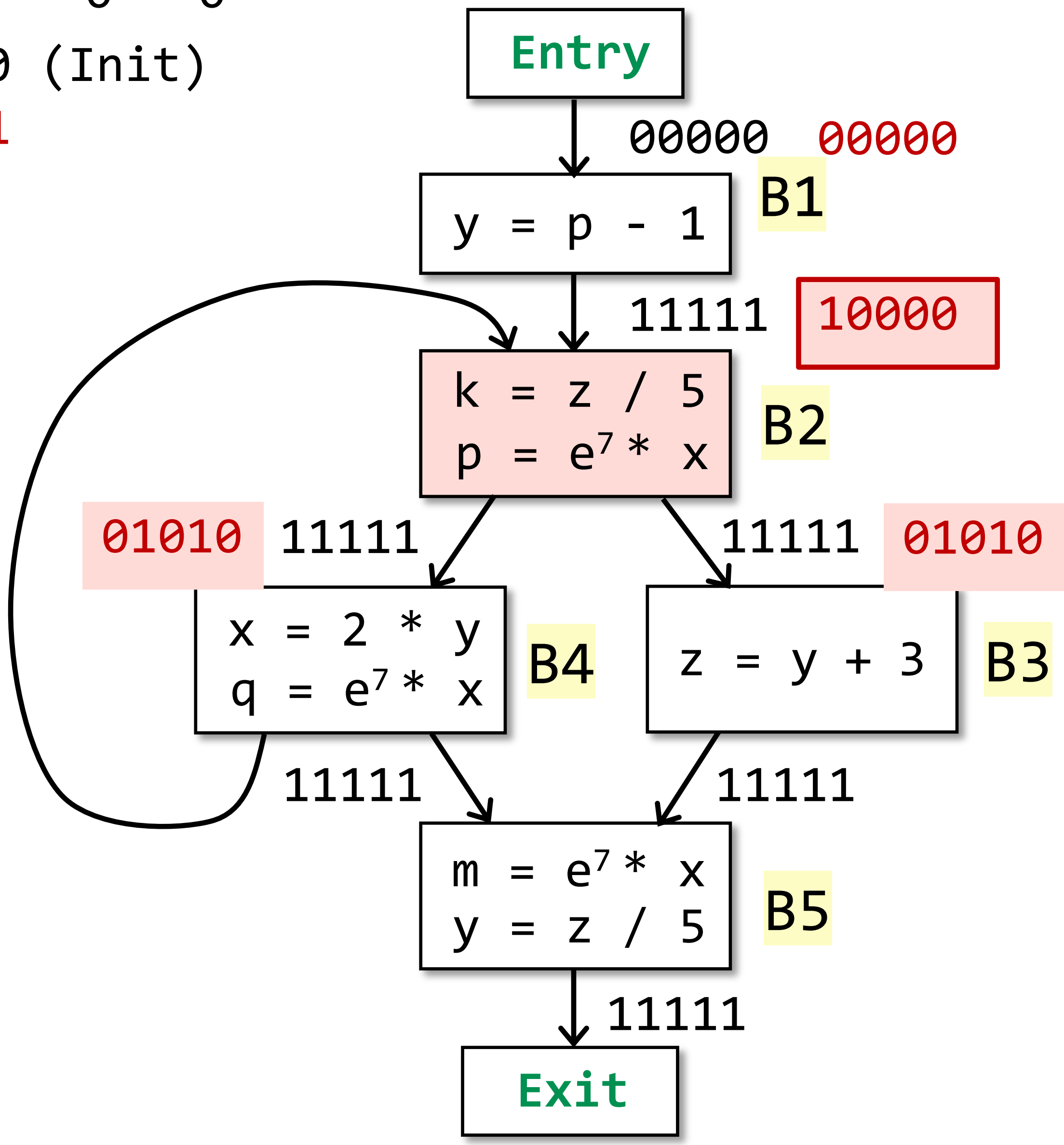


p-1 z/5 2*y e^7*x y+3

0 0 0 0 0

Iteration 0 (Init)

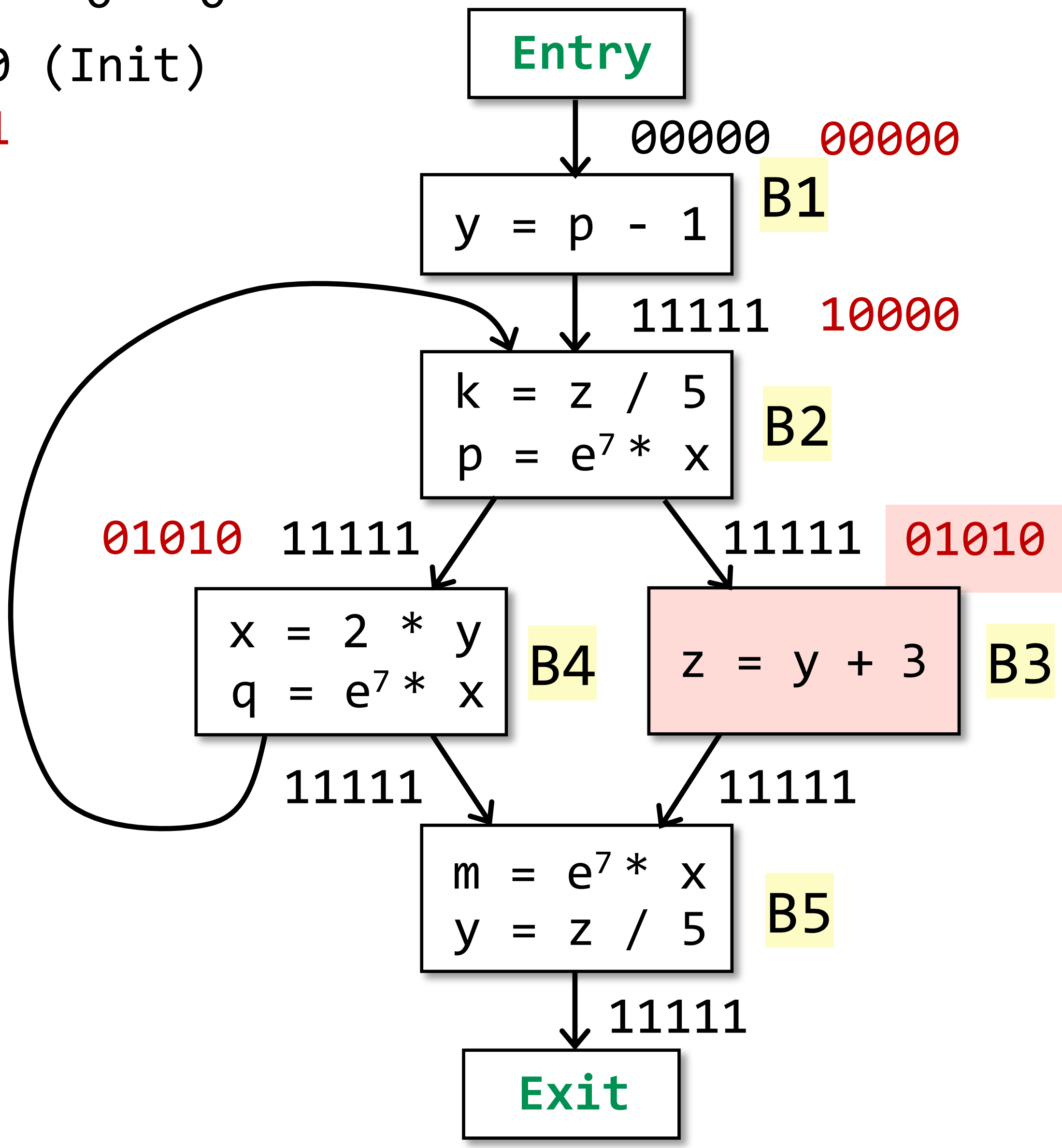
Iteration 1



p-1	z/5	2*y	e^7*x	y+3
0	0	0	0	0

Iteration 0 (Init)

Iteration 1

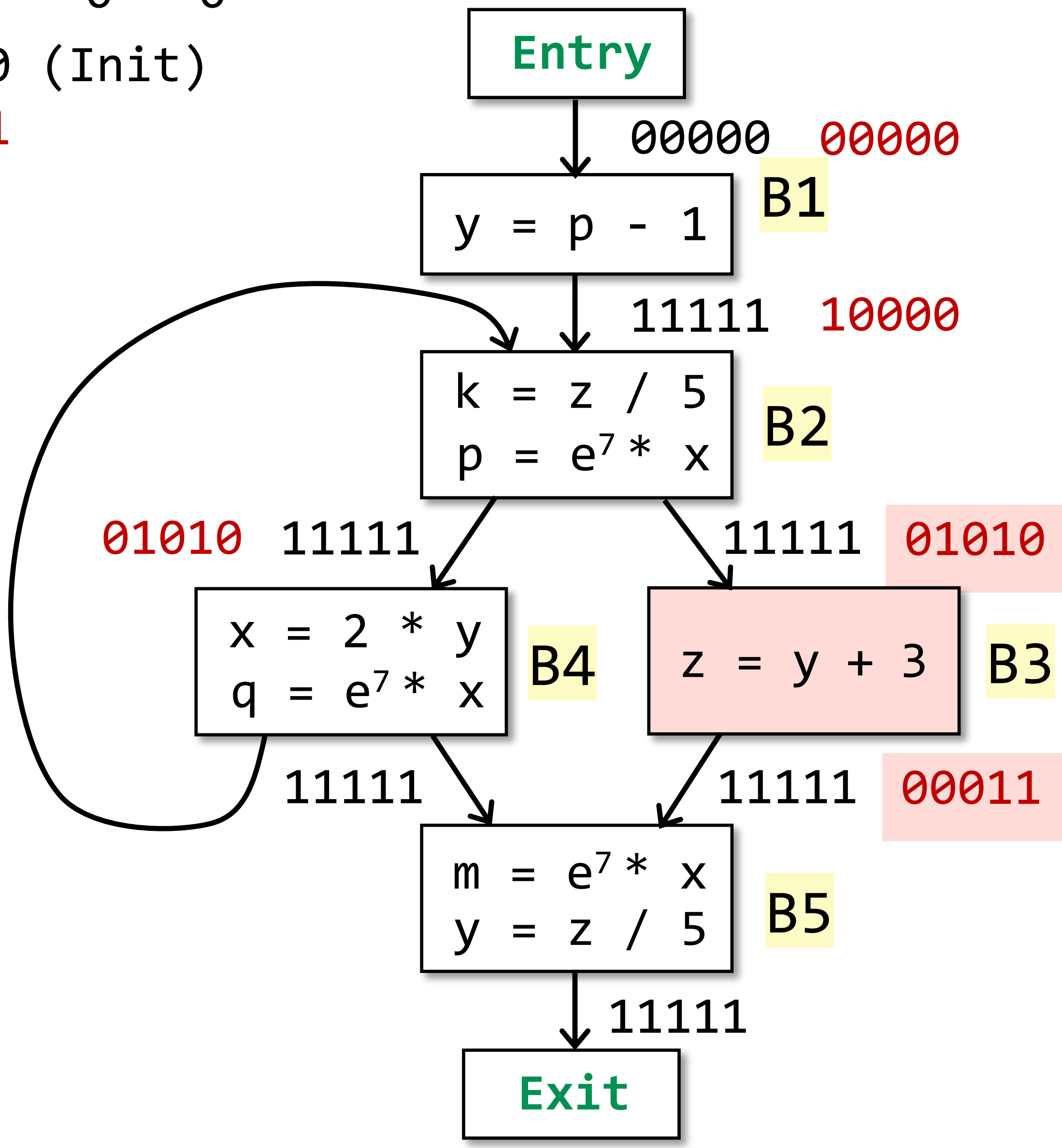


p-1 z/5 2*y e^7*x y+3

0 0 0 0 0

Iteration 0 (Init)

Iteration 1

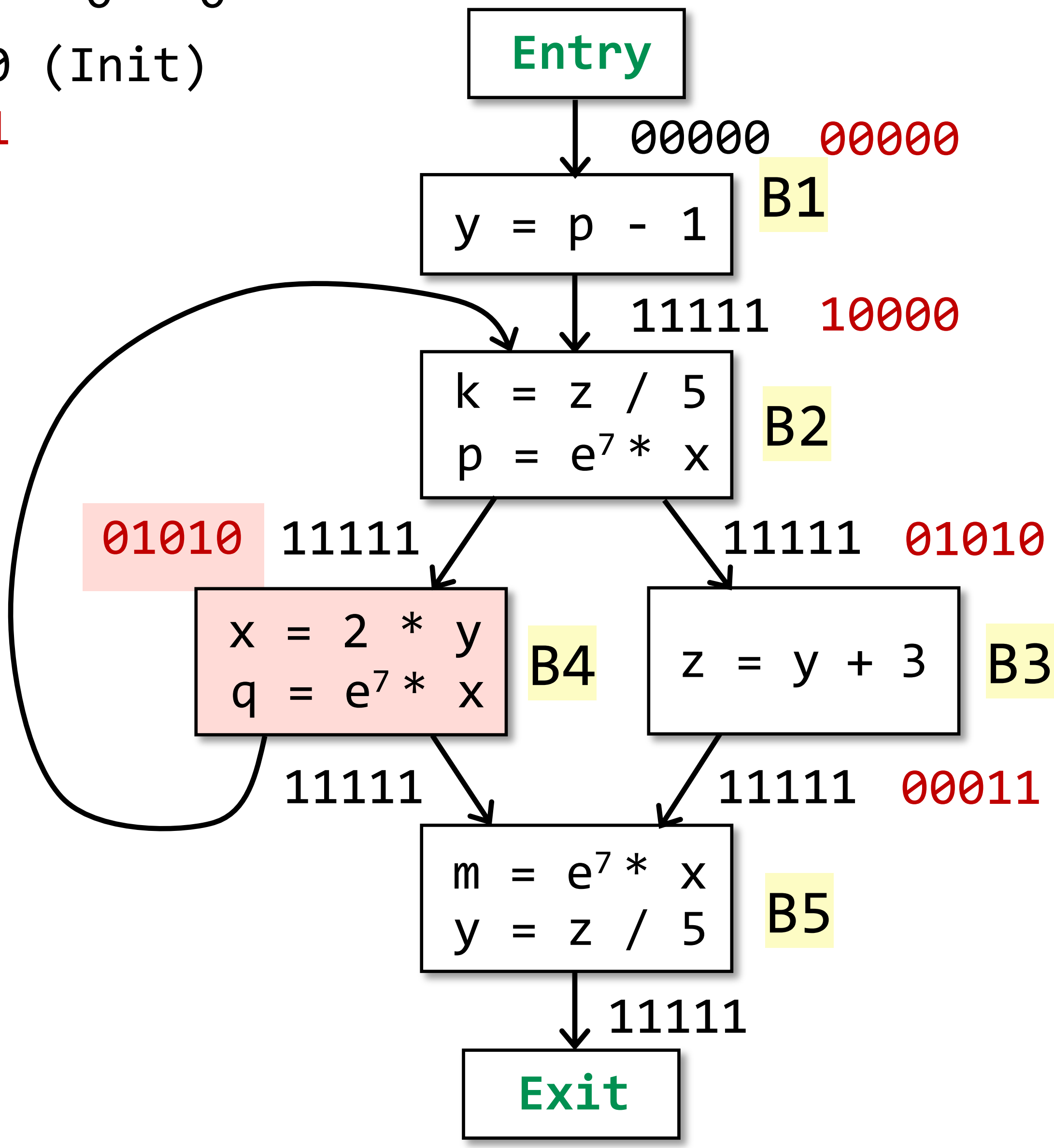


p-1 z/5 2*y e^7*x y+3

0 0 0 0 0

Iteration 0 (Init)

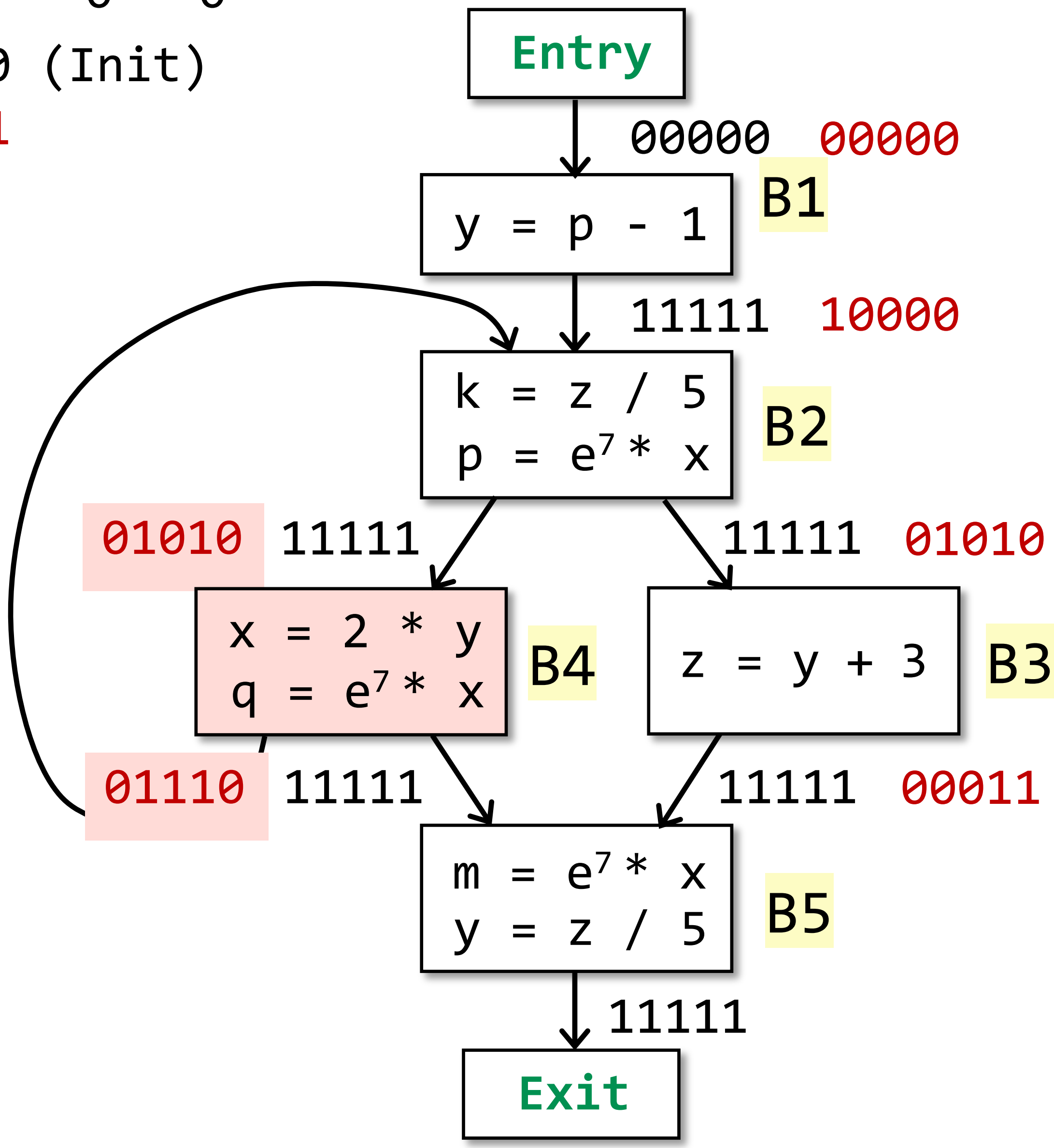
Iteration 1



p-1	z/5	2*y	e^7*x	y+3
0	0	0	0	0

Iteration 0 (Init)

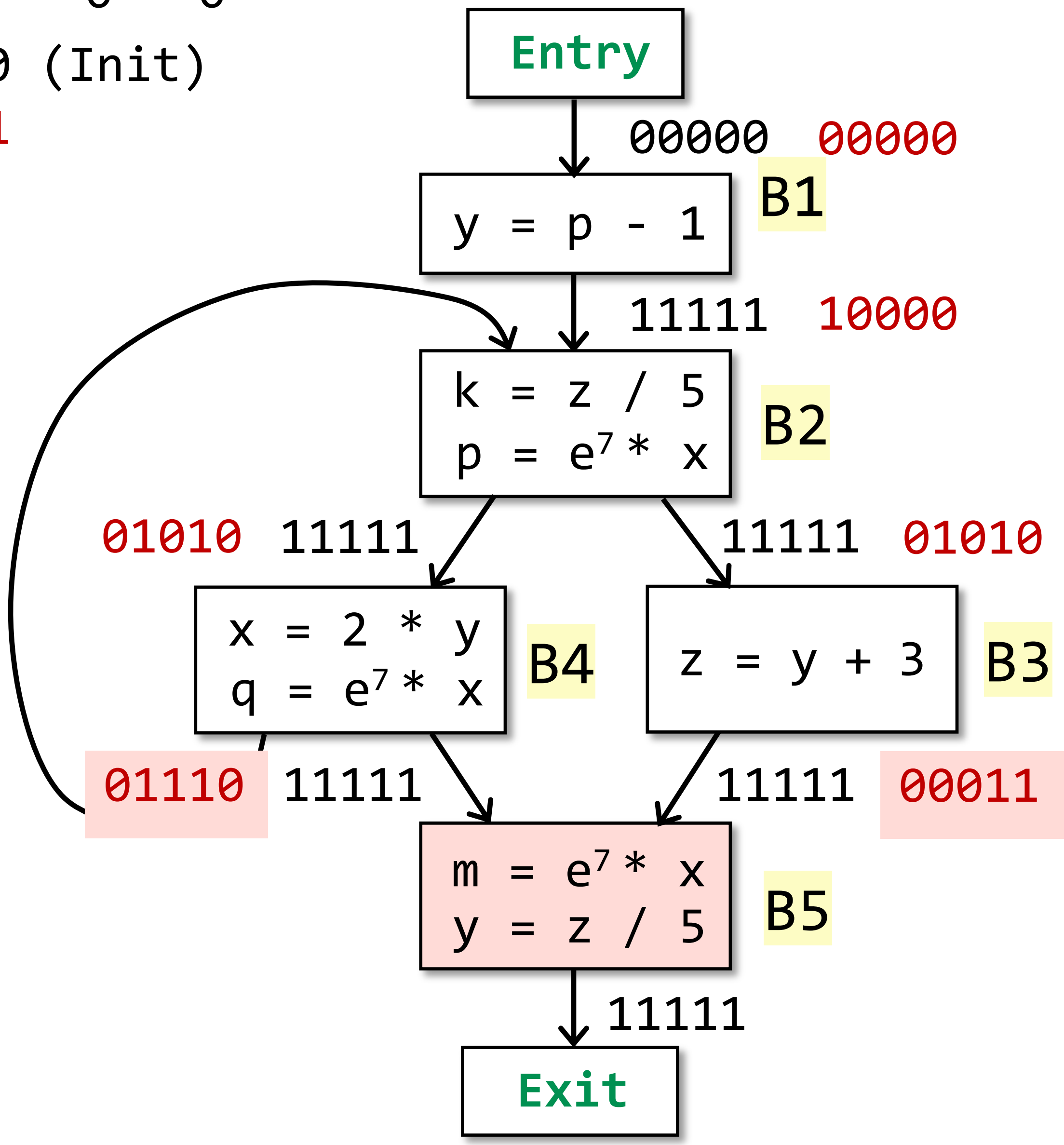
Iteration 1



p-1 z/5 2*y e^7*x y+3
0 0 0 0 0

Iteration 0 (Init)

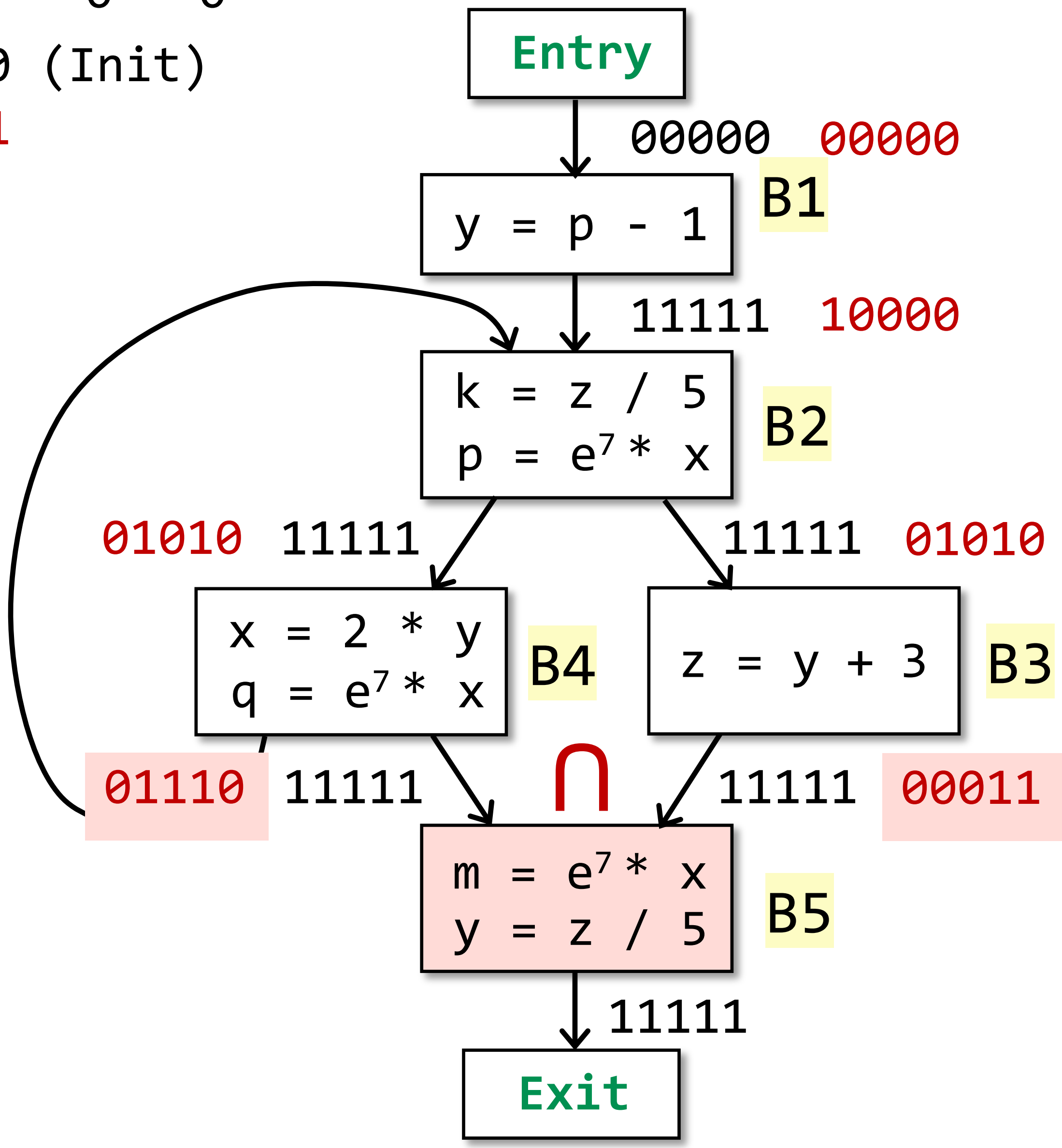
Iteration 1



p-1 z/5 2*y e^7*x y+3
0 0 0 0 0

Iteration 0 (Init)

Iteration 1

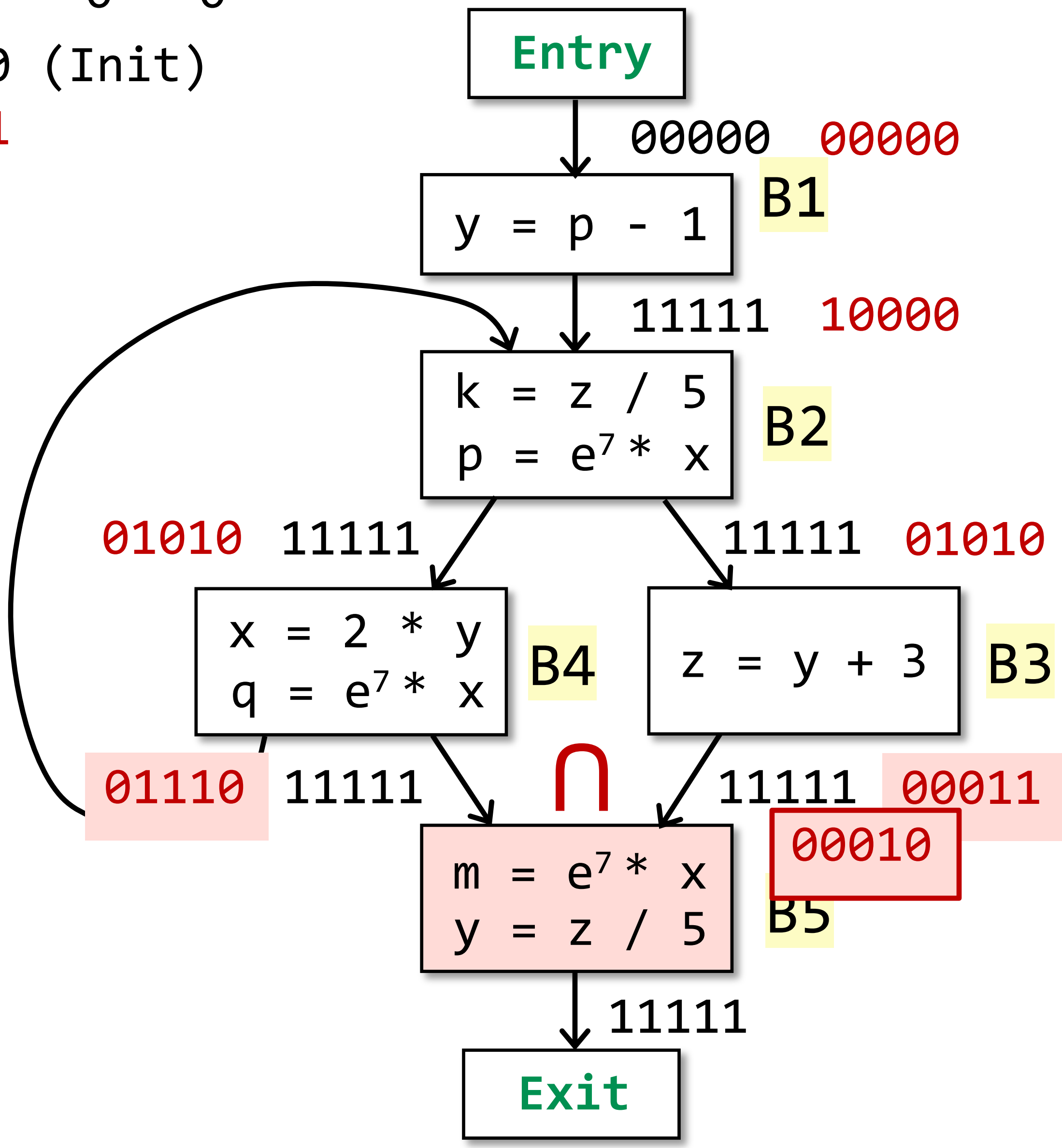


p-1 z/5 2*y e⁷*x y+3

0 0 0 0 0

Iteration 0 (Init)

Iteration 1

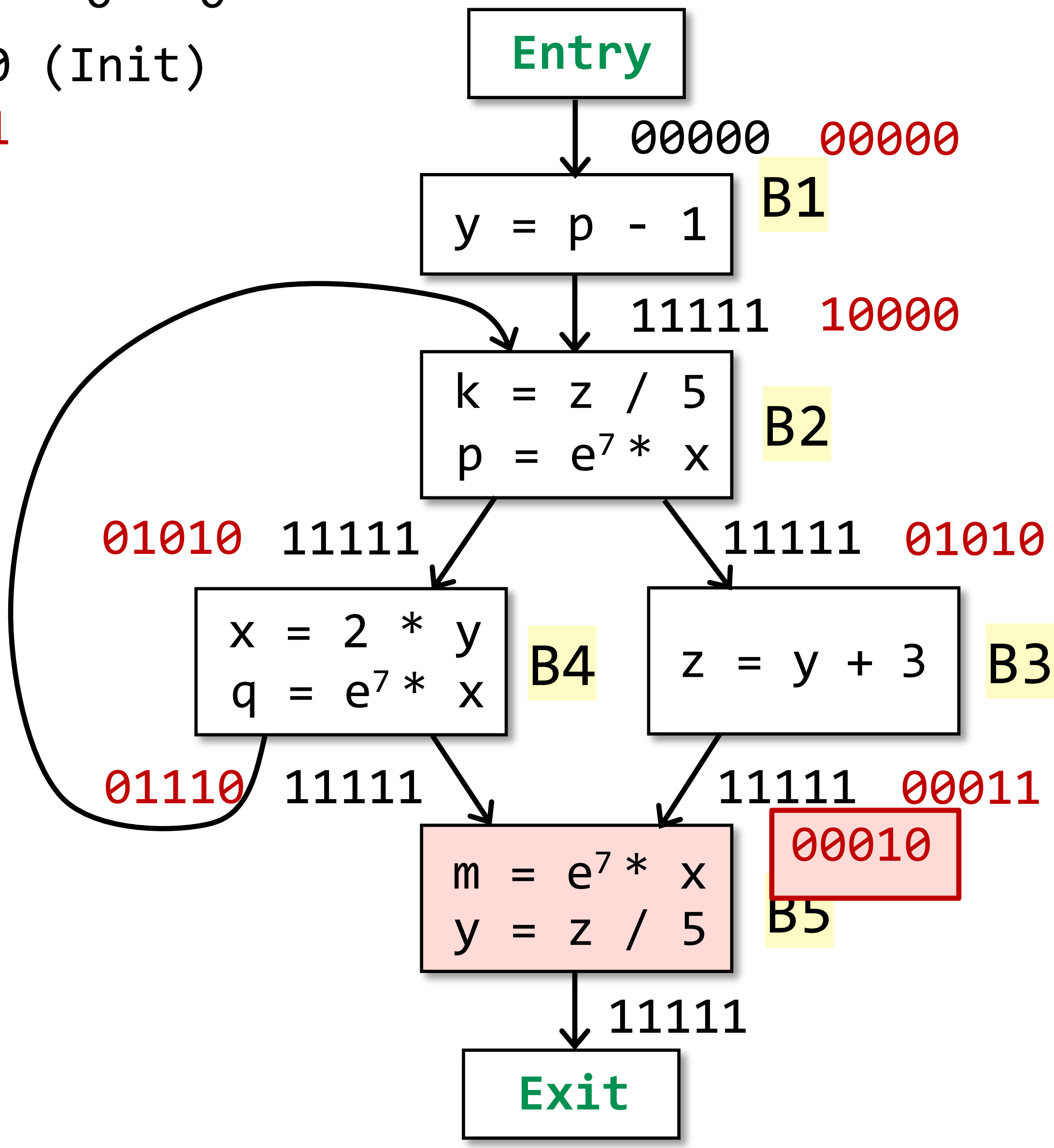


p-1 z/5 2*y e^7*x y+3

0 0 0 0 0

Iteration 0 (Init)

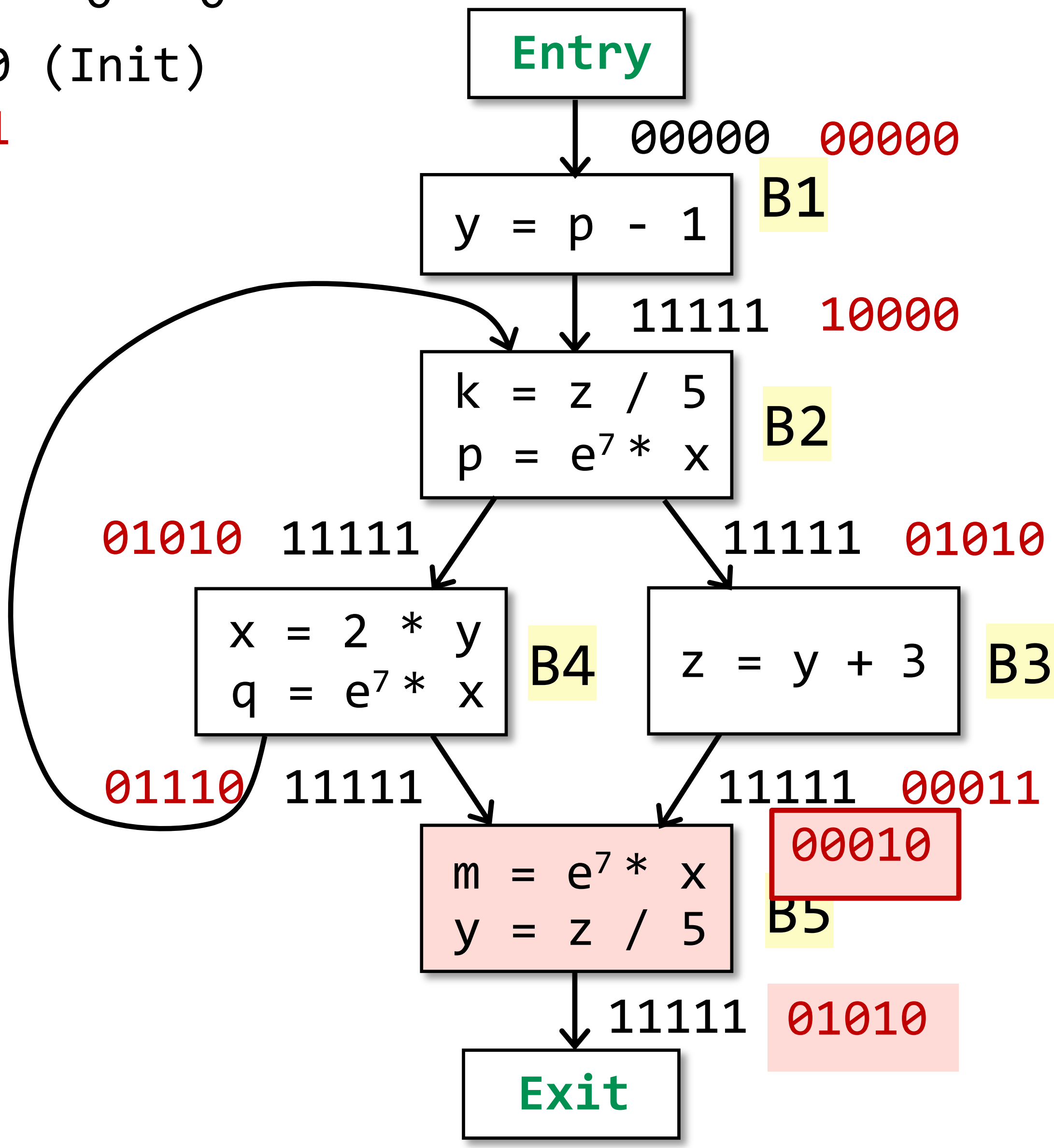
Iteration 1



p-1 z/5 2*y e^7*x y+3
0 0 0 0 0

Iteration 0 (Init)

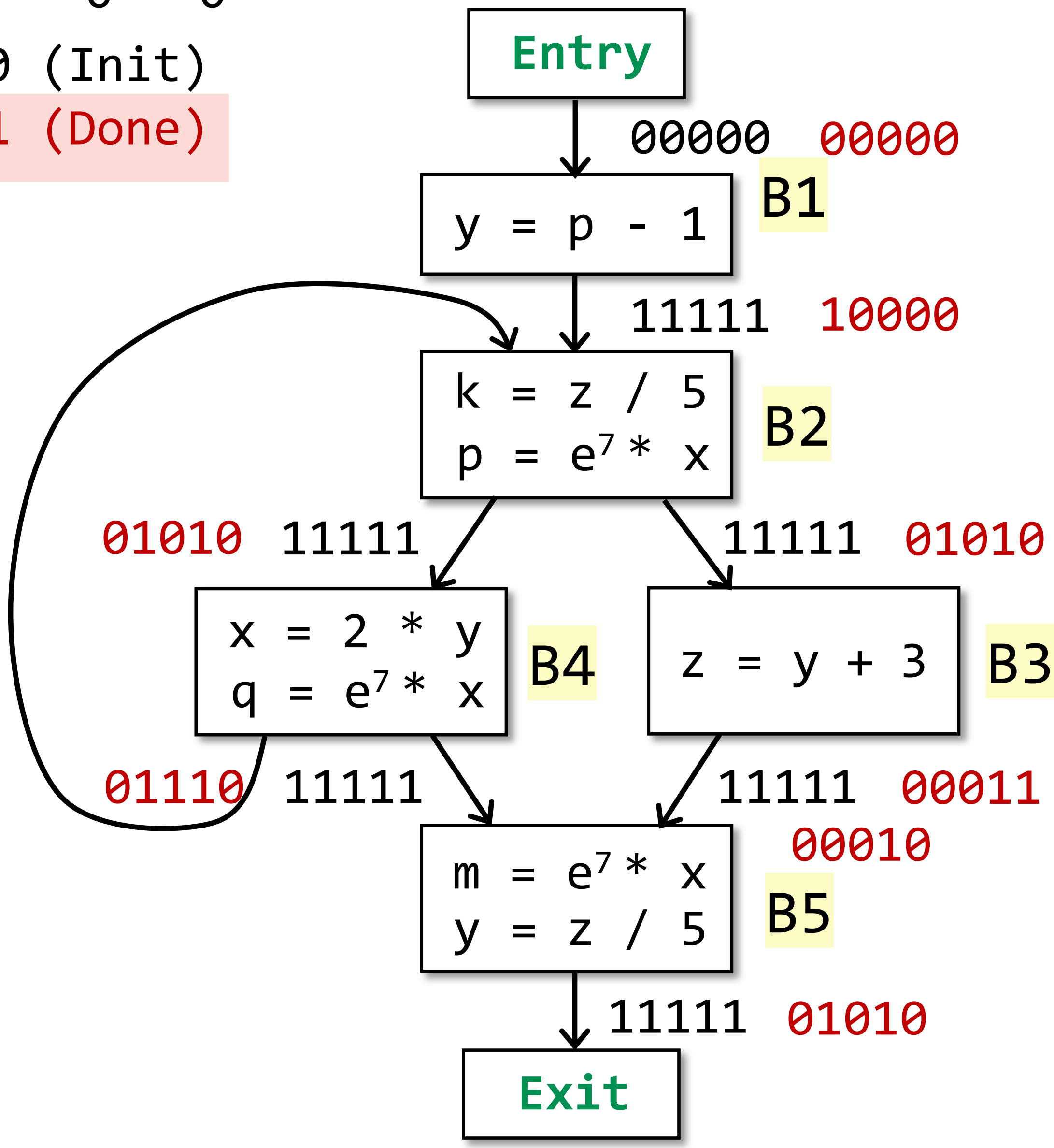
Iteration 1



p-1	z/5	2*y	e^7*x	y+3
0	0	0	0	0

Iteration 0 (Init)

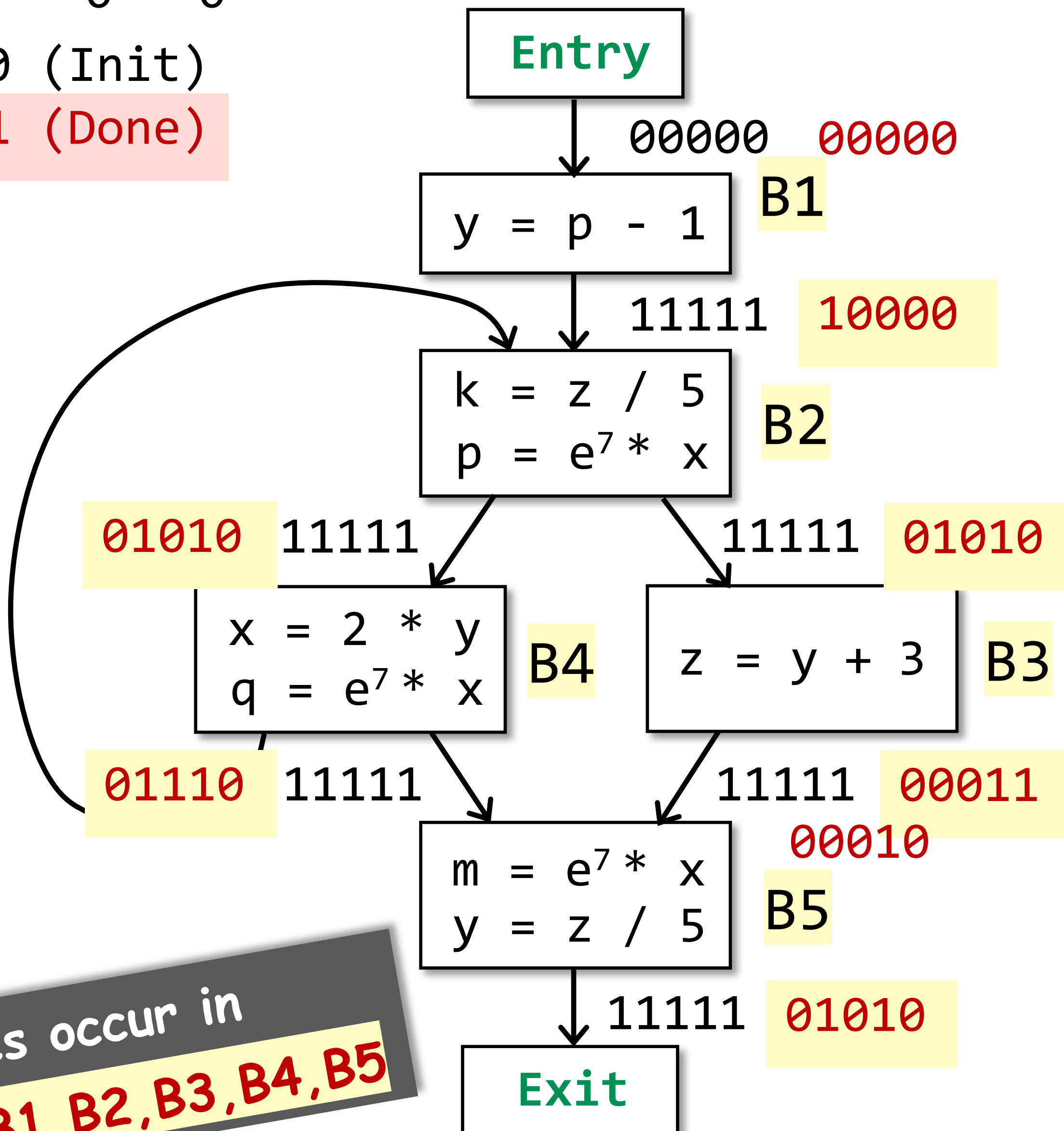
Iteration 1 (Done)



p-1	z/5	2*y	e^7*x	y+3
0	0	0	0	0

Iteration 0 (Init)

Iteration 1 (Done)



Changes occur in OUT[] of B1,B2,B3,B4,B5

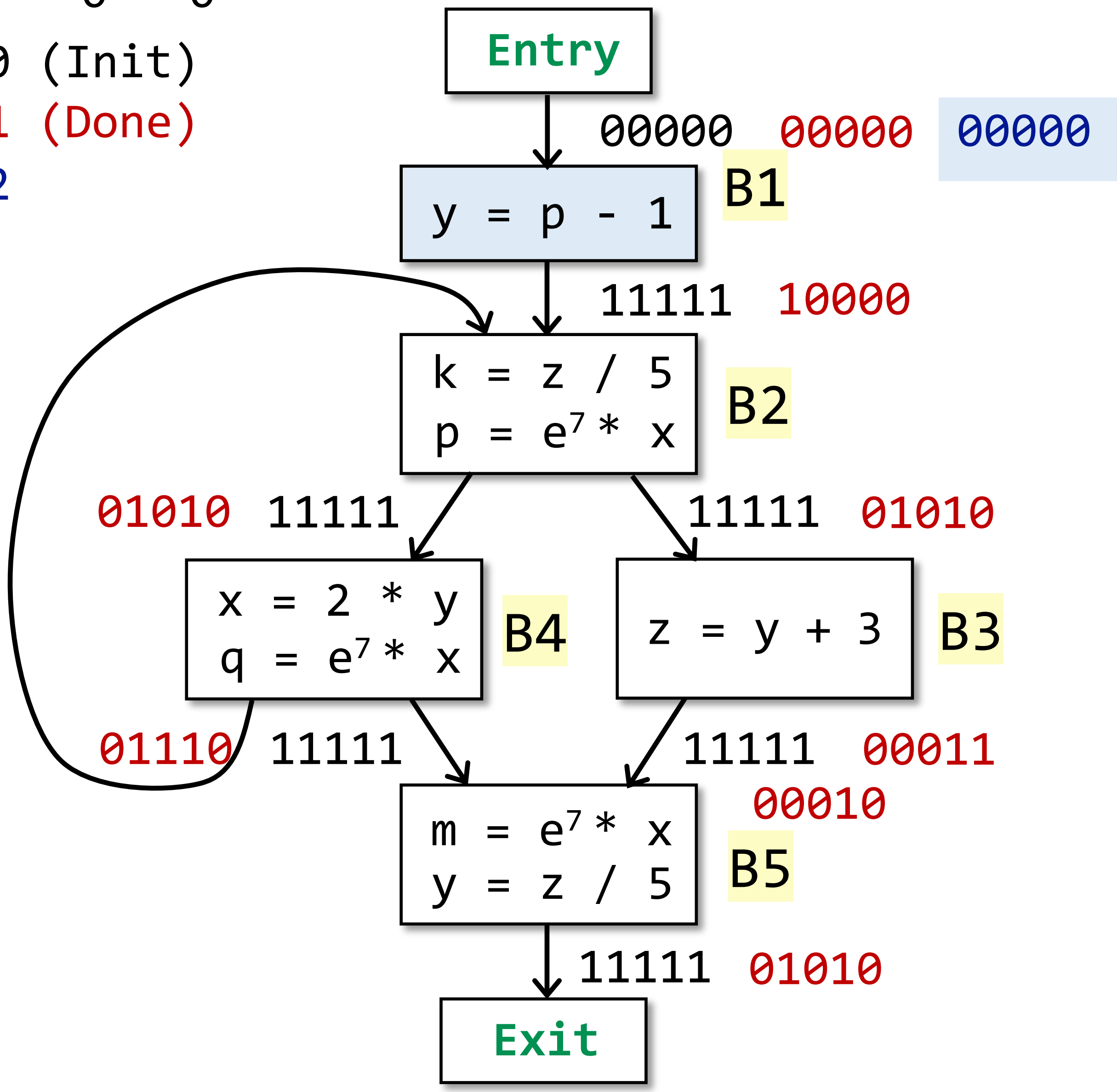
p-1 z/5 2*y e^7*x y+3

0 0 0 0 0

Iteration 0 (Init)

Iteration 1 (Done)

Iteration 2

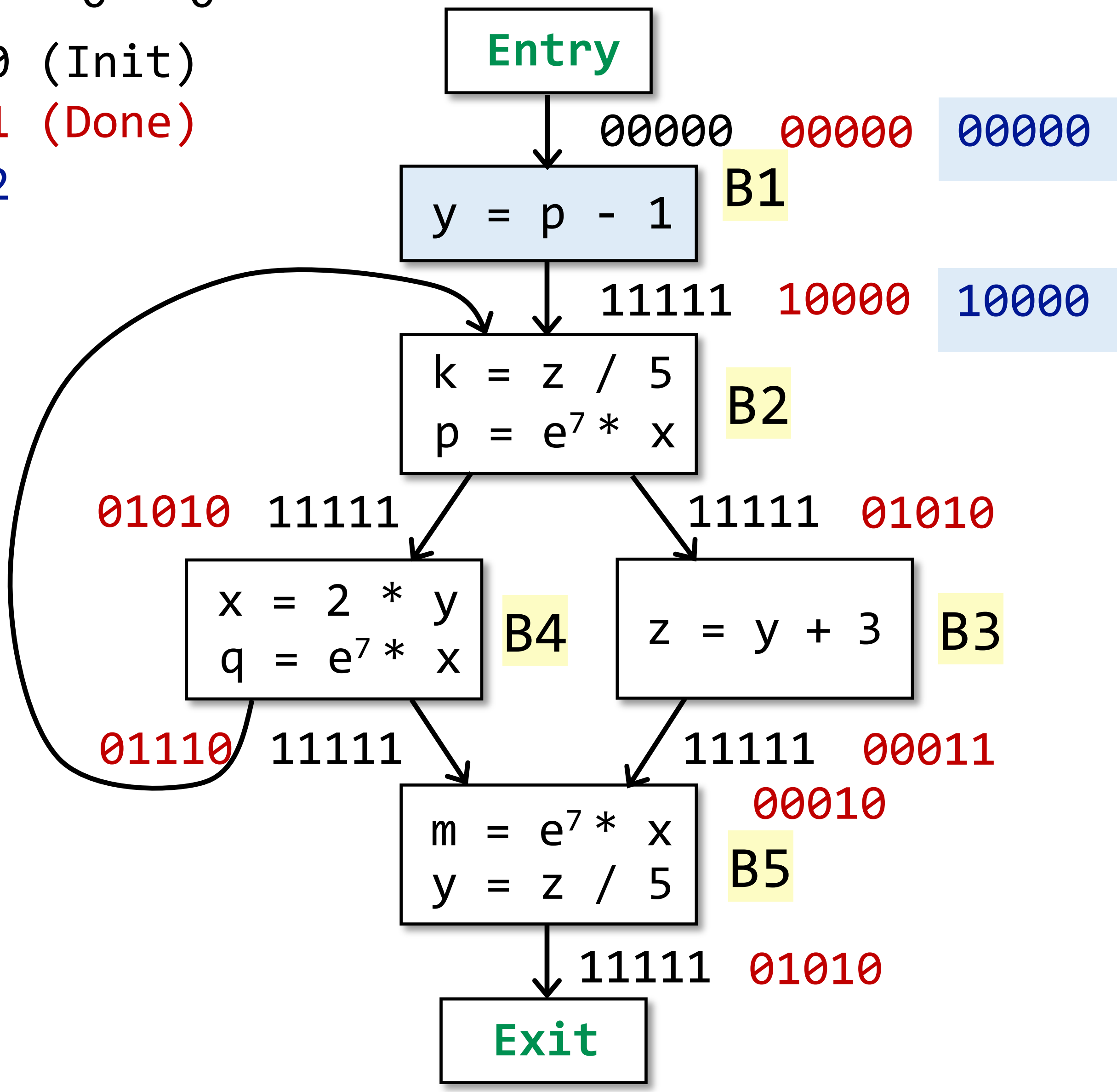


p-1	z/5	2*y	e^7*x	y+3
0	0	0	0	0

Iteration 0 (Init)

Iteration 1 (Done)

Iteration 2

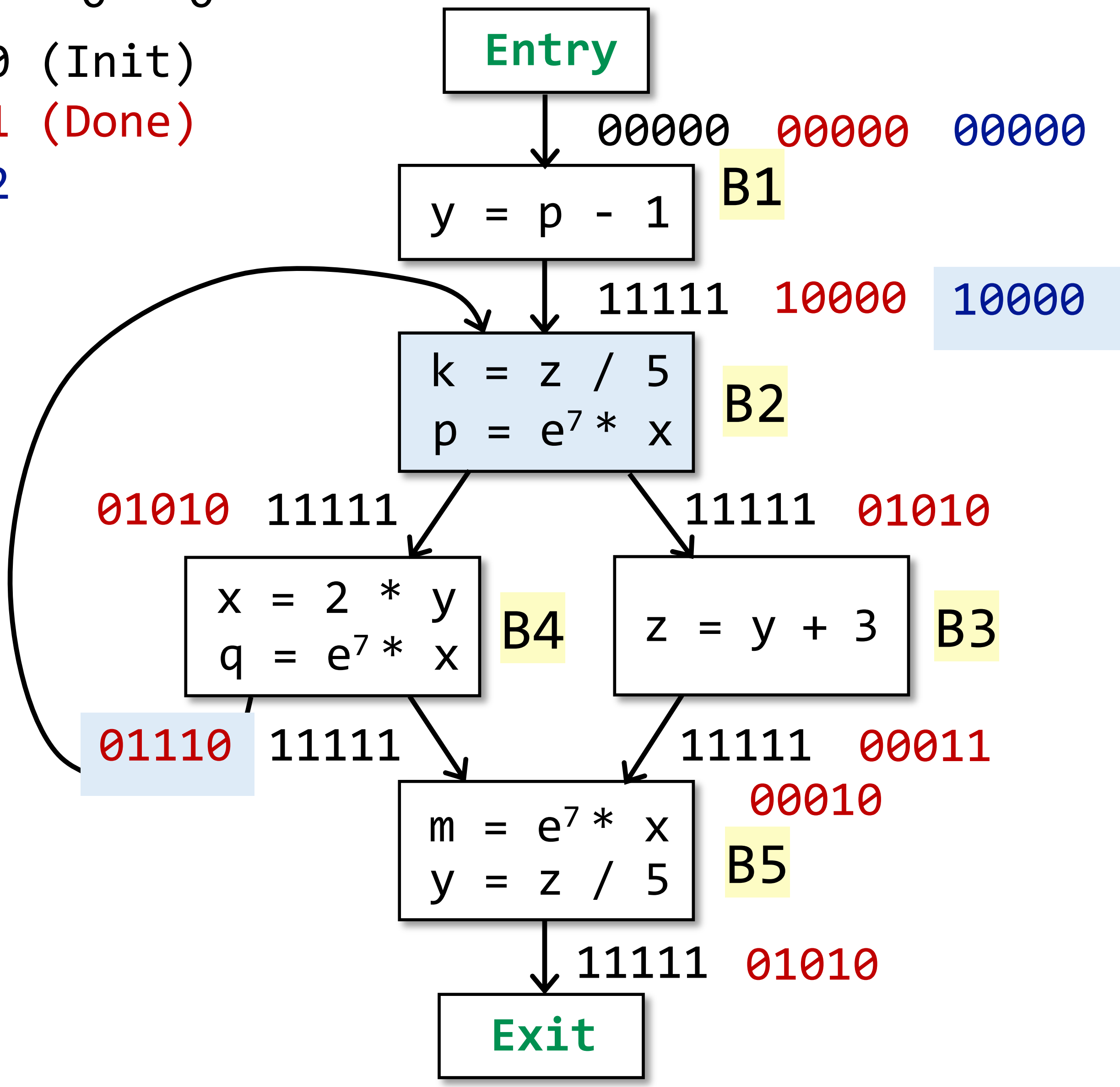


```
p-1 z/5 2*y e7*x y+3
 0   0   0   0   0
```

Iteration 0 (Init)

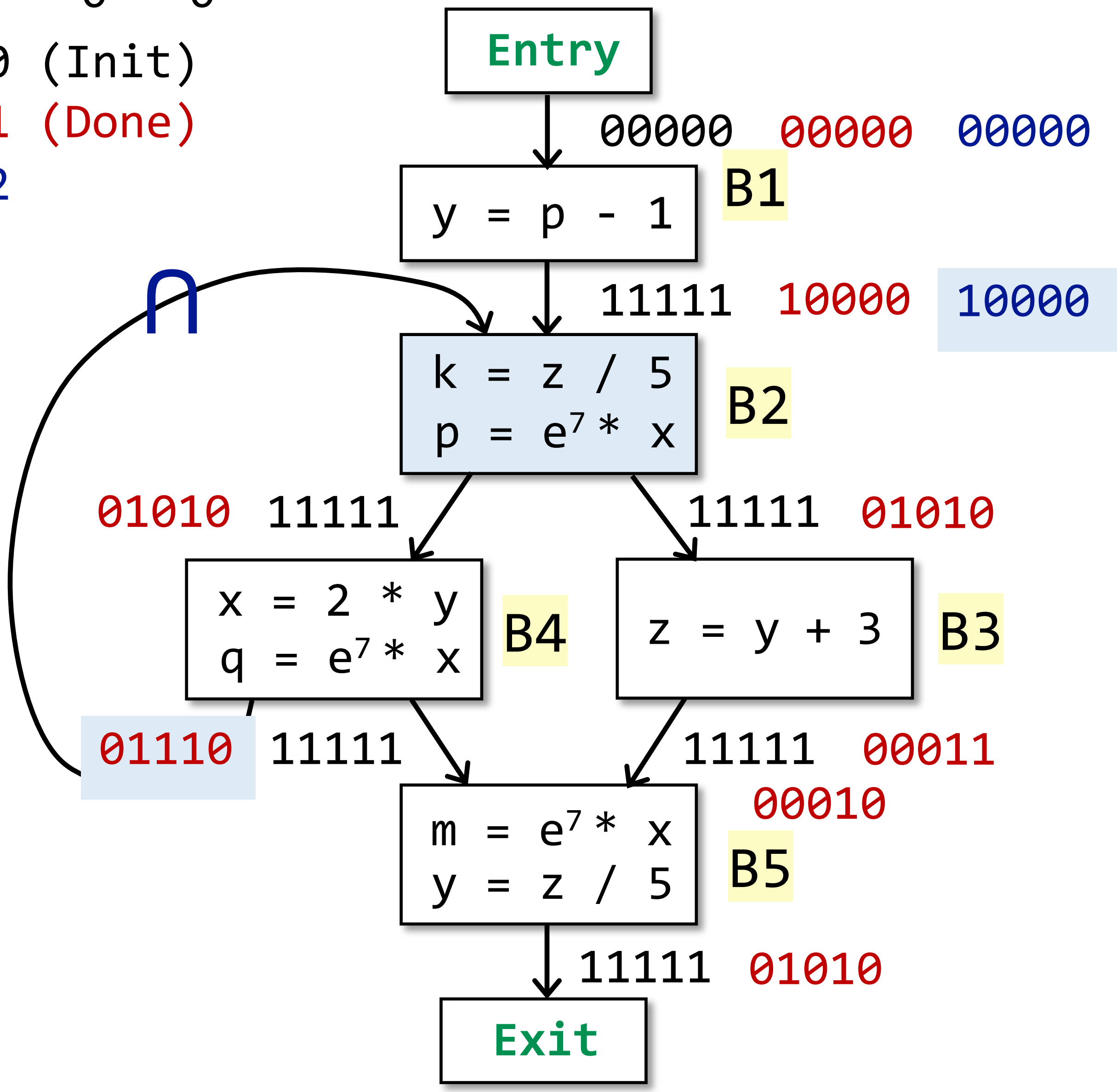
Iteration 1 (Done)

Iteration 2



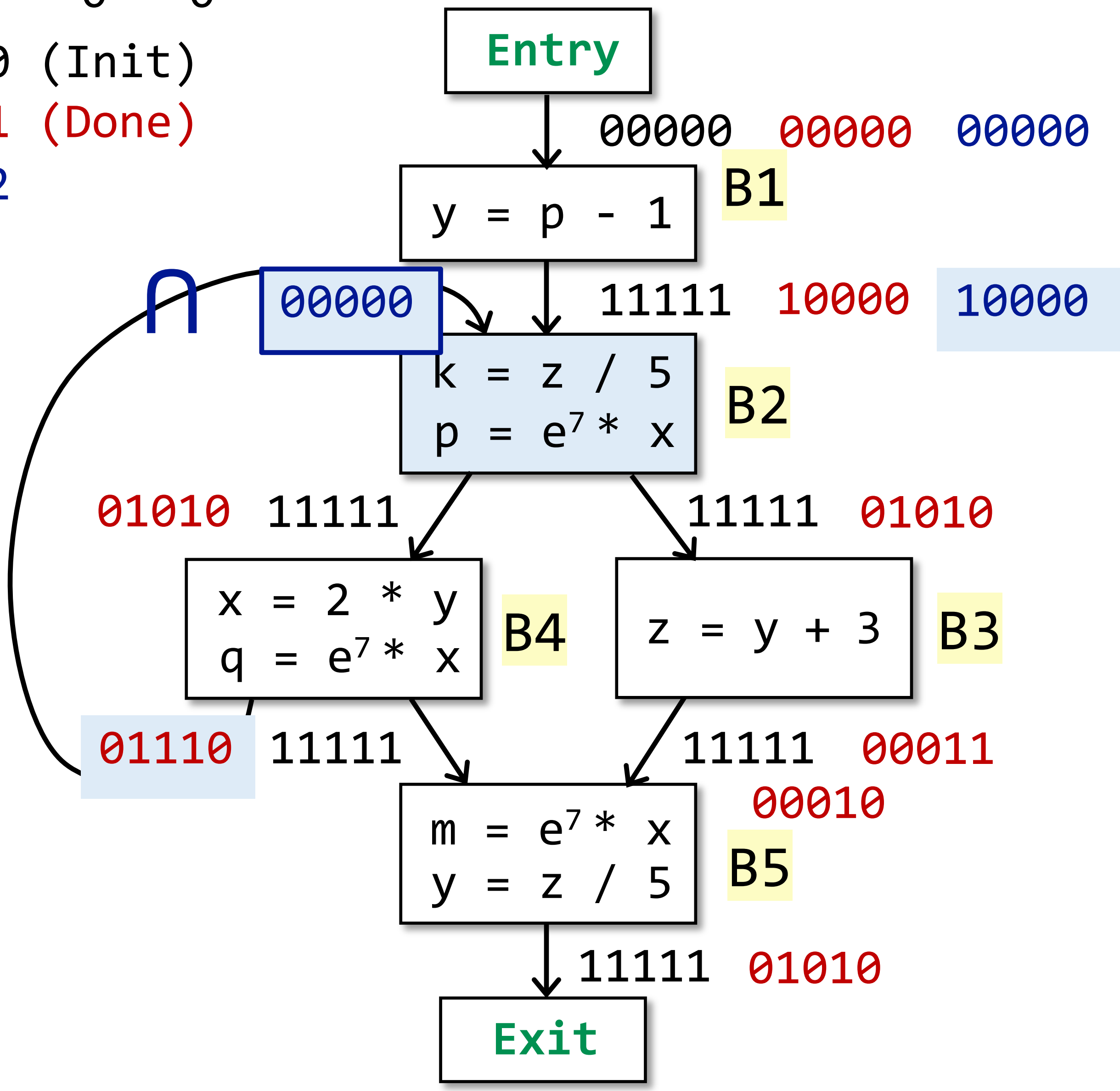
p-1 z/5 2*y e^7*x y+3
0 0 0 0 0

Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2



p-1 z/5 2*y e^7*x y+3
0 0 0 0 0

Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2

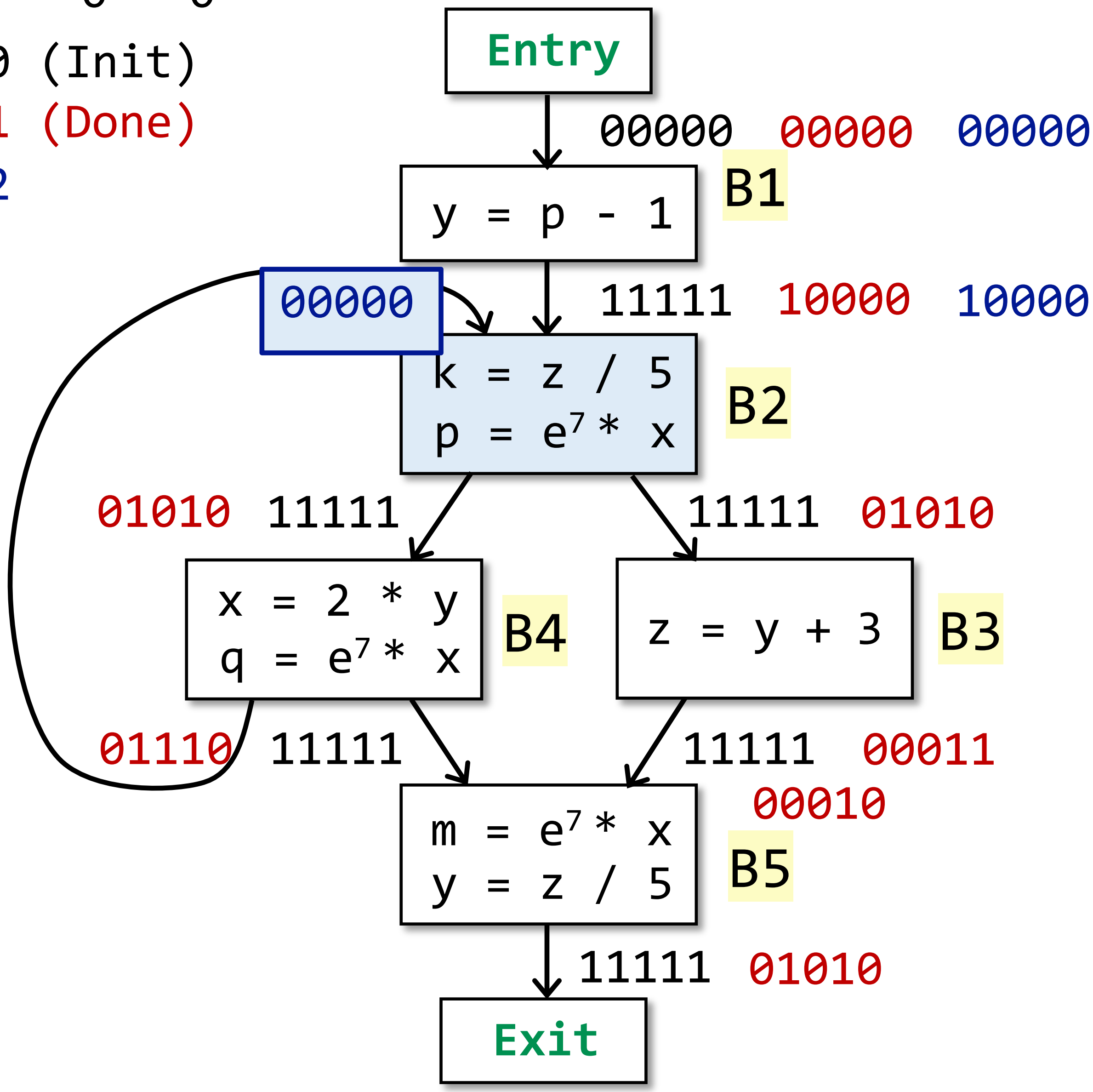


p-1	z/5	2*y	e^7*x	y+3
0	0	0	0	0

Iteration 0 (Init)

Iteration 1 (Done)

Iteration 2



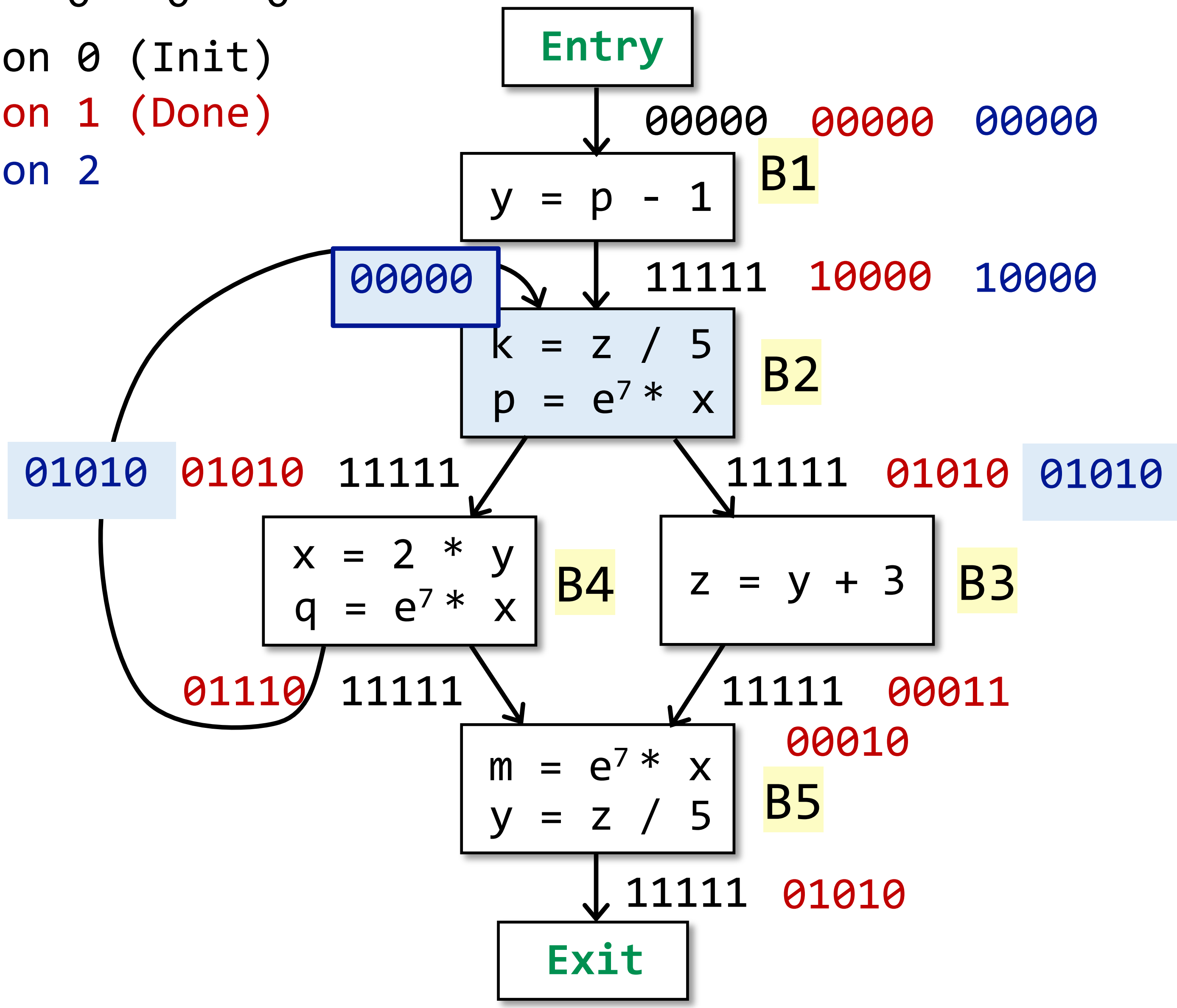
p-1 z/5 2*y e^7*x y+3

0 0 0 0 0

Iteration 0 (Init)

Iteration 1 (Done)

Iteration 2



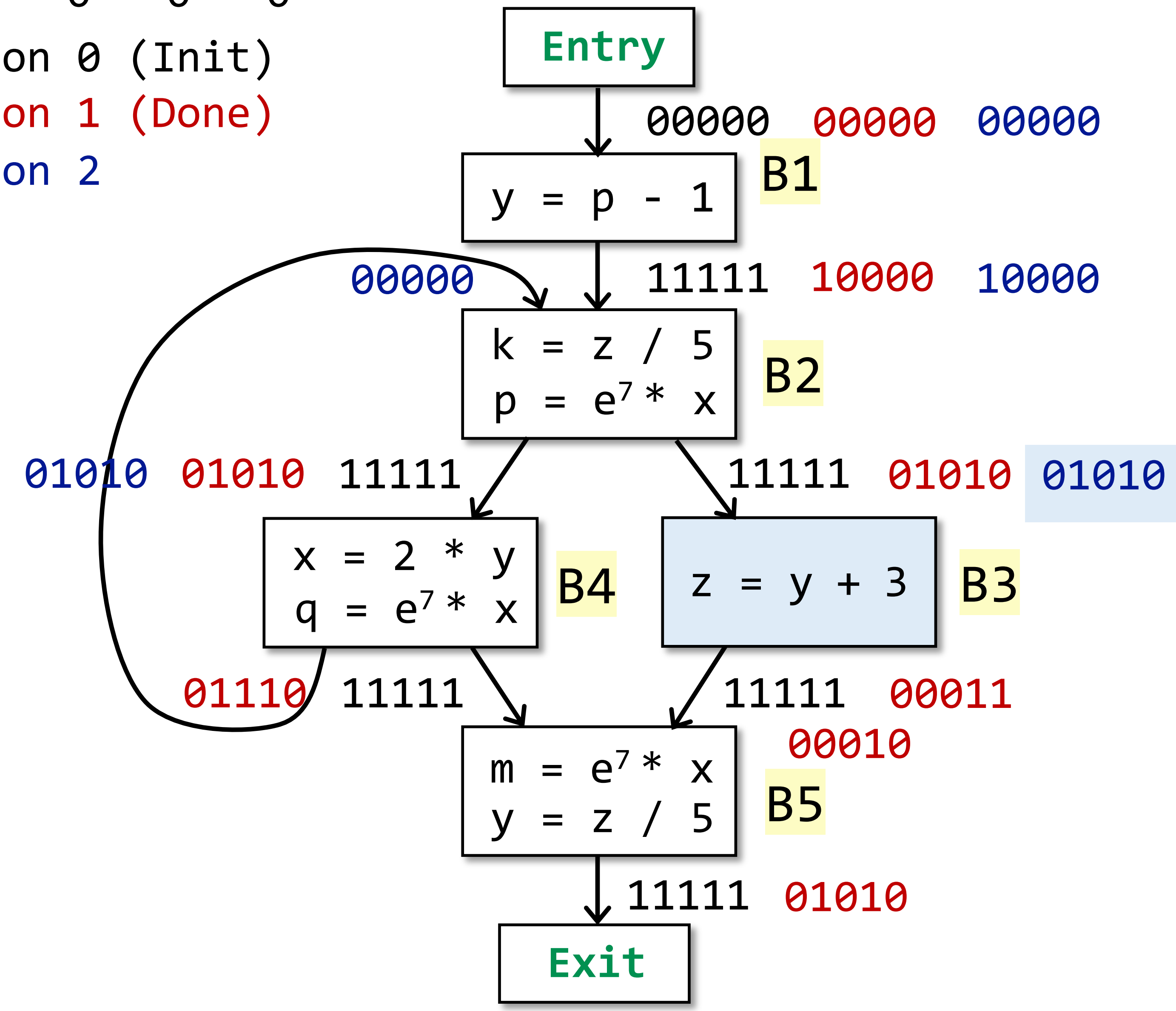
p-1 z/5 2*y e7*x y+3

0 0 0 0 0

Iteration 0 (Init)

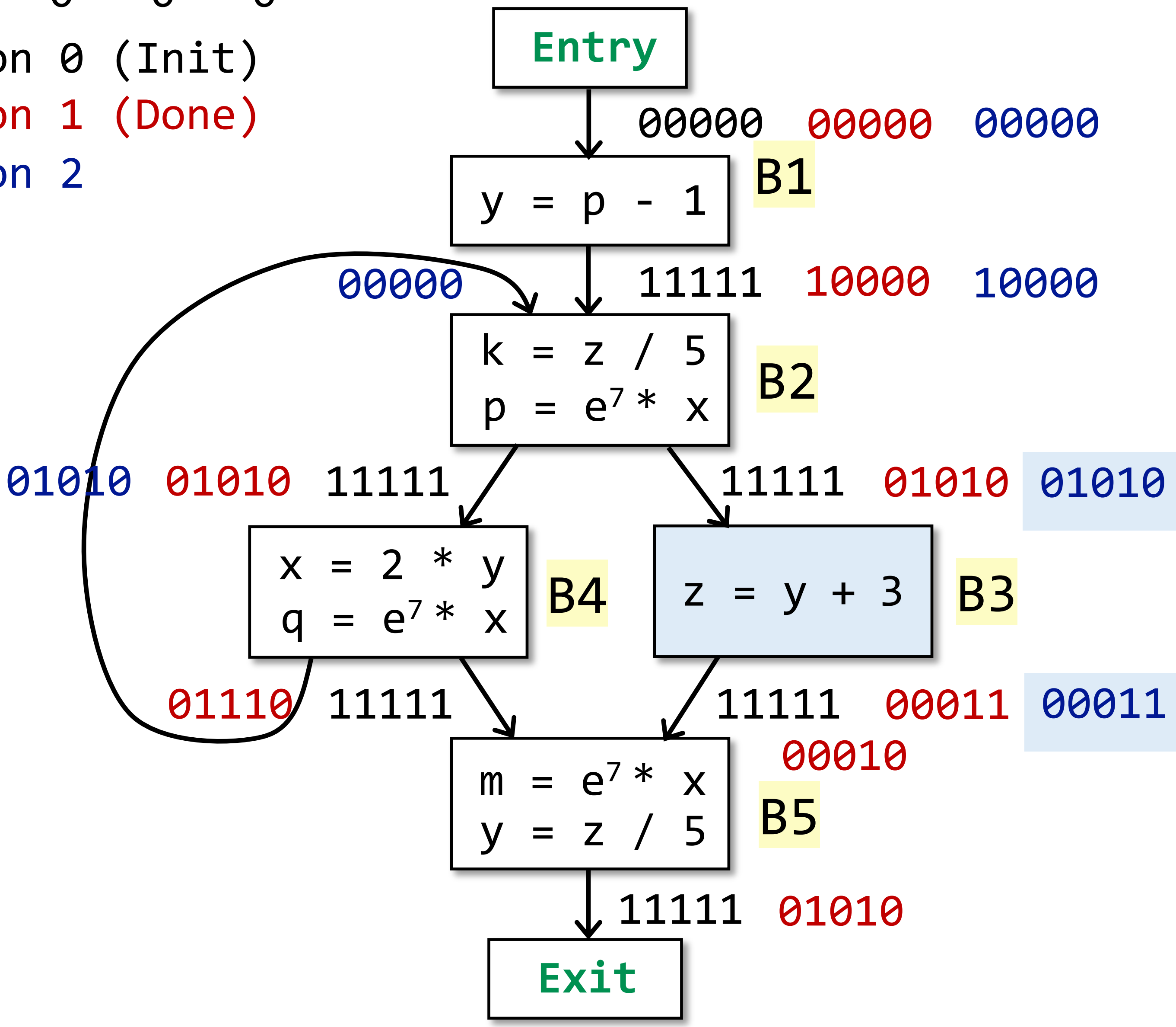
Iteration 1 (Done)

Iteration 2



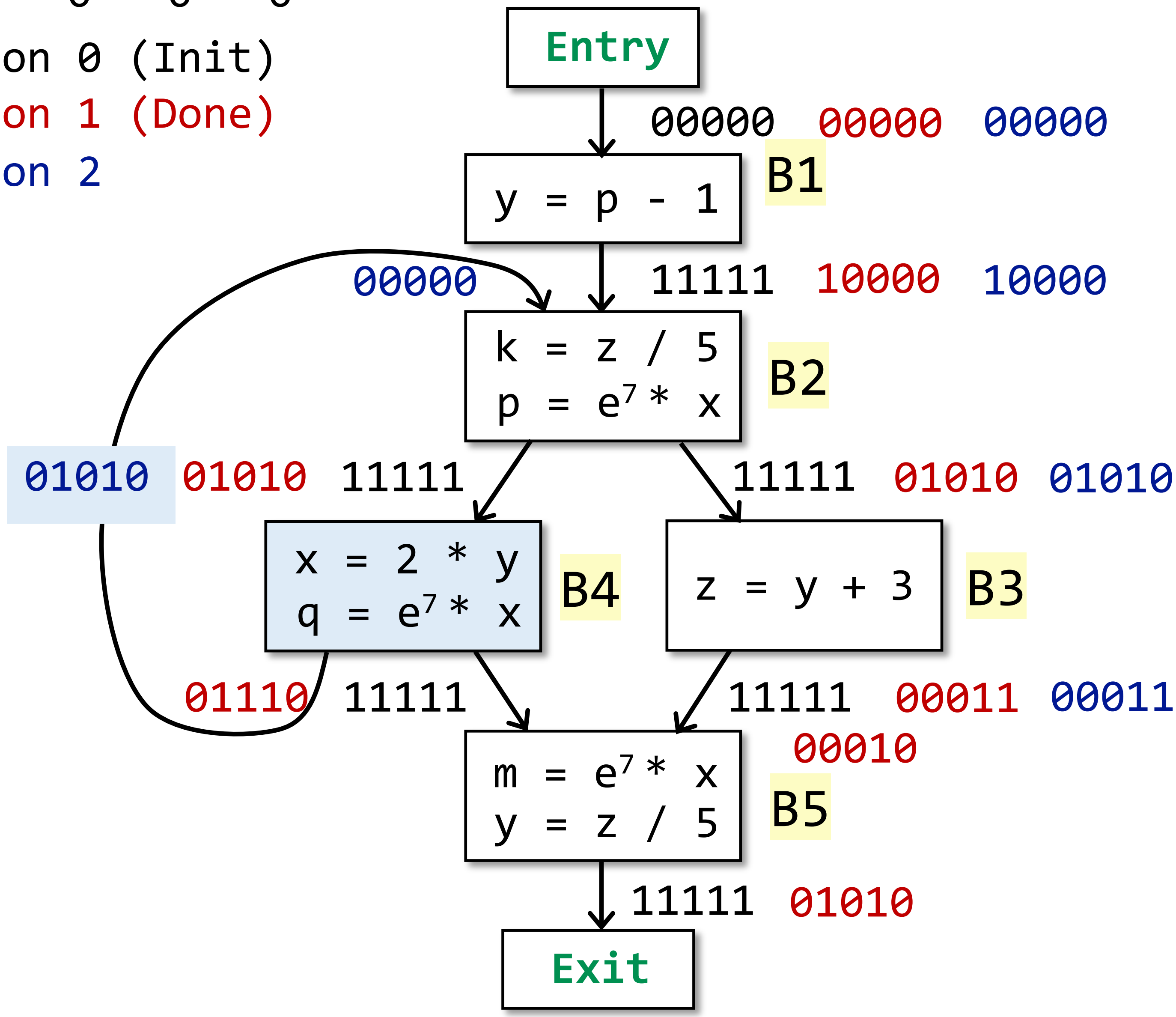
p-1	z/5	2*y	e^7*x	y+3
0	0	0	0	0

Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2



p-1 z/5 2*y e^7*x y+3
0 0 0 0 0

Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2



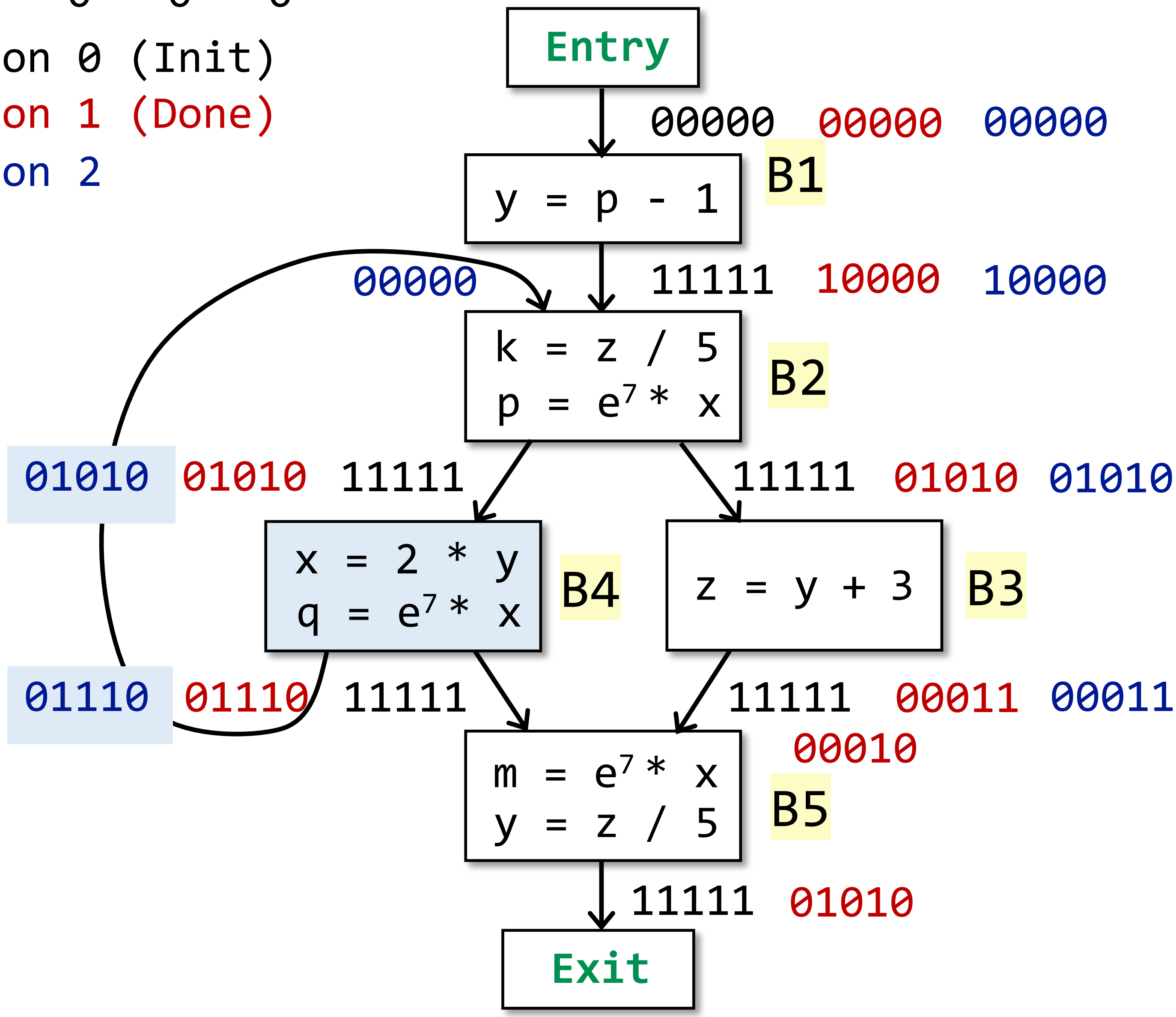
p-1 z/5 2*y e^7*x y+3

0 0 0 0 0

Iteration 0 (Init)

Iteration 1 (Done)

Iteration 2



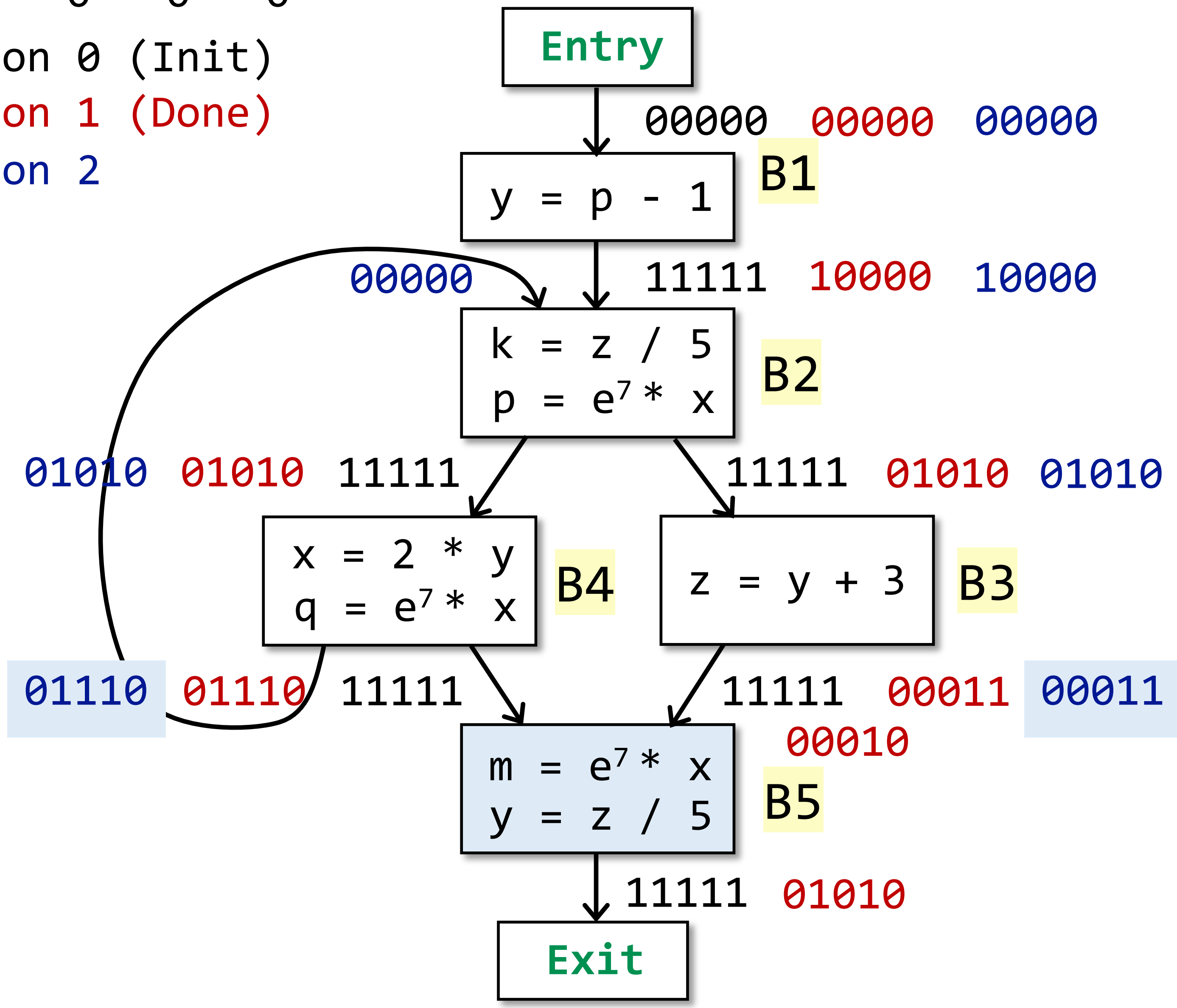
p-1 z/5 2*y e^7*x y+3

0 0 0 0 0

Iteration 0 (Init)

Iteration 1 (Done)

Iteration 2

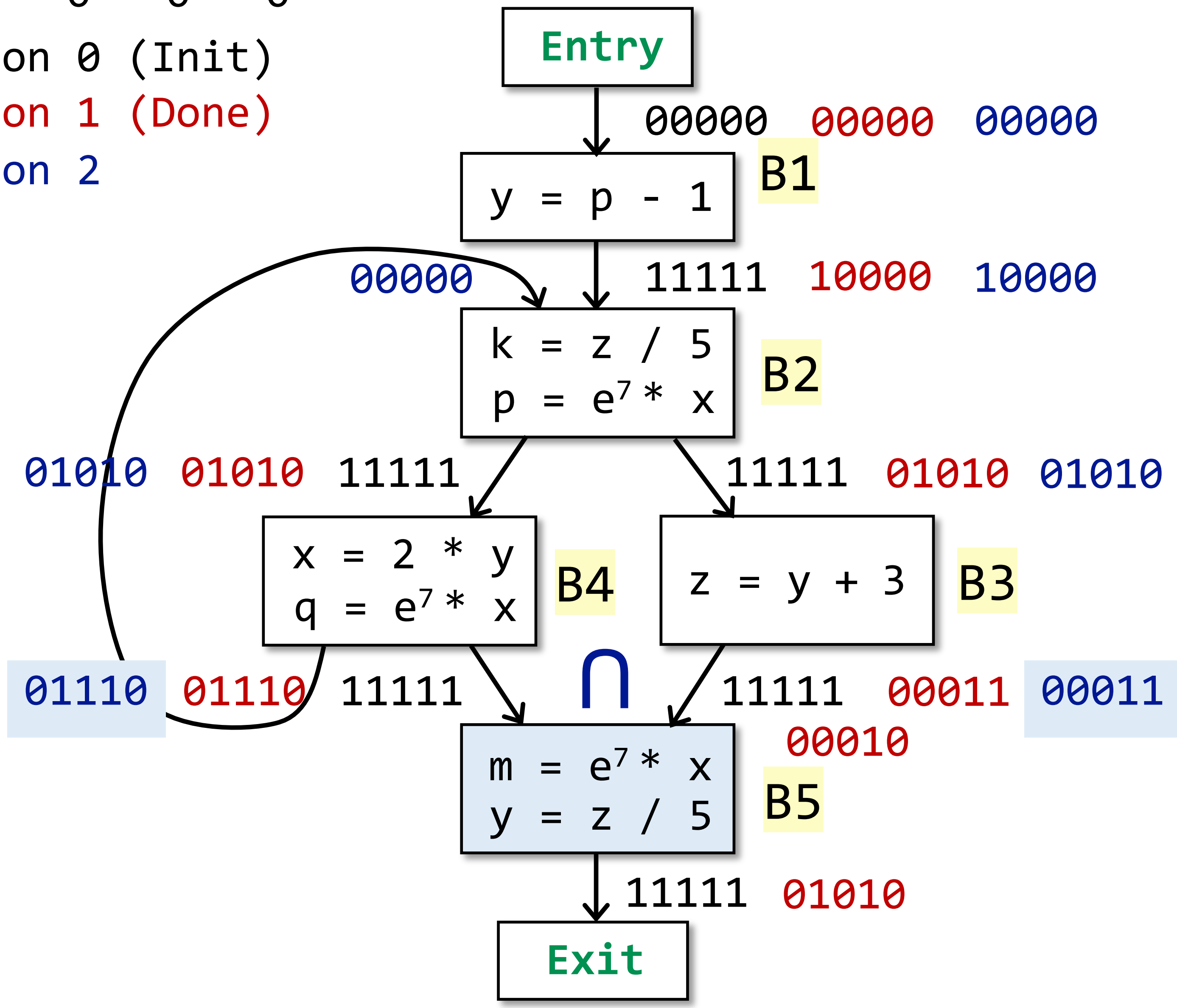


p-1 z/5 2*y e^7*x y+3
0 0 0 0 0

Iteration 0 (Init)

Iteration 1 (Done)

Iteration 2



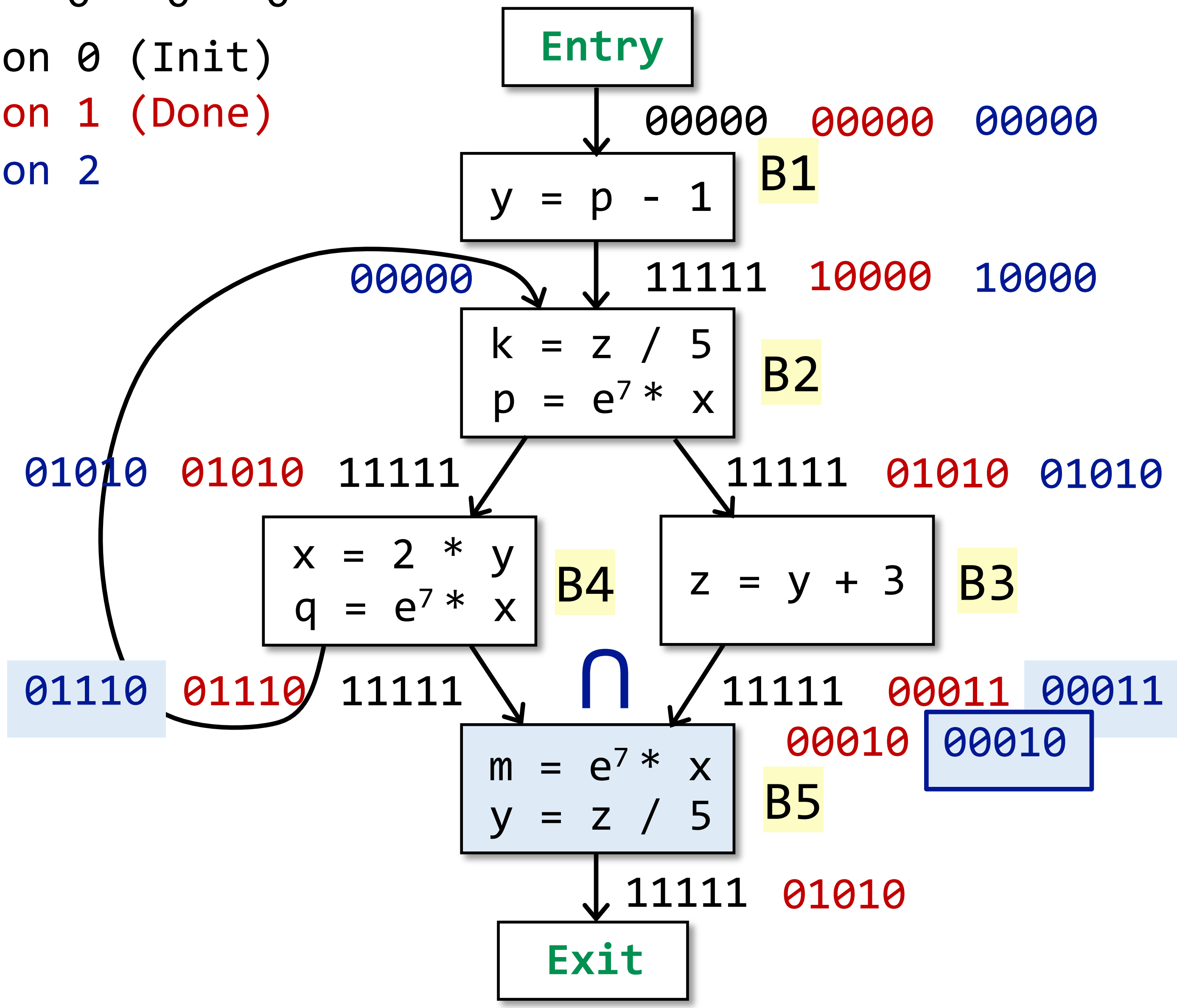
p-1 z/5 2*y e^7*x y+3

0 0 0 0 0

Iteration 0 (Init)

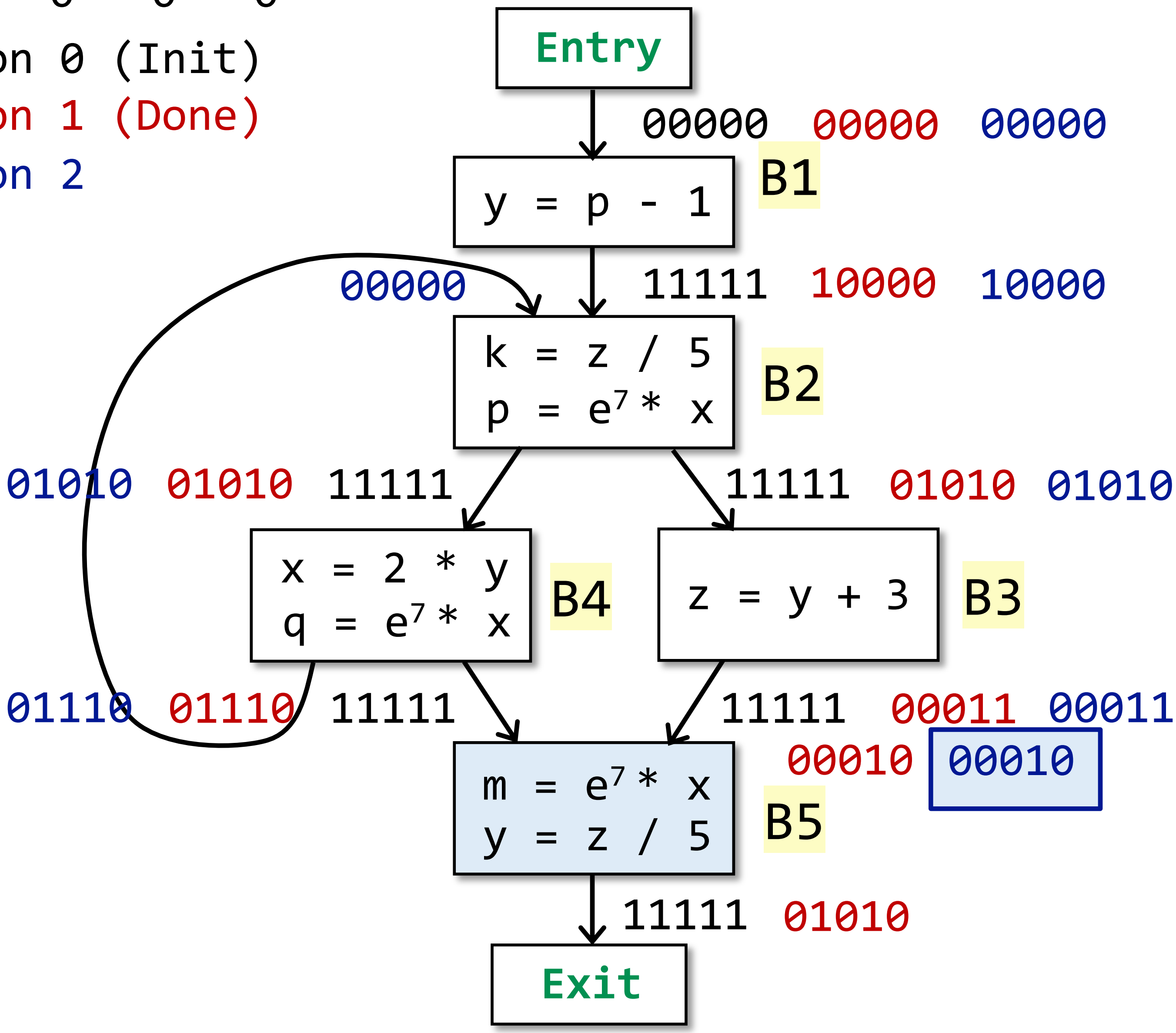
Iteration 1 (Done)

Iteration 2



p-1 z/5 2*y e^7*x y+3
0 0 0 0 0

Iteration 0 (Init)
Iteration 1 (Done)
Iteration 2

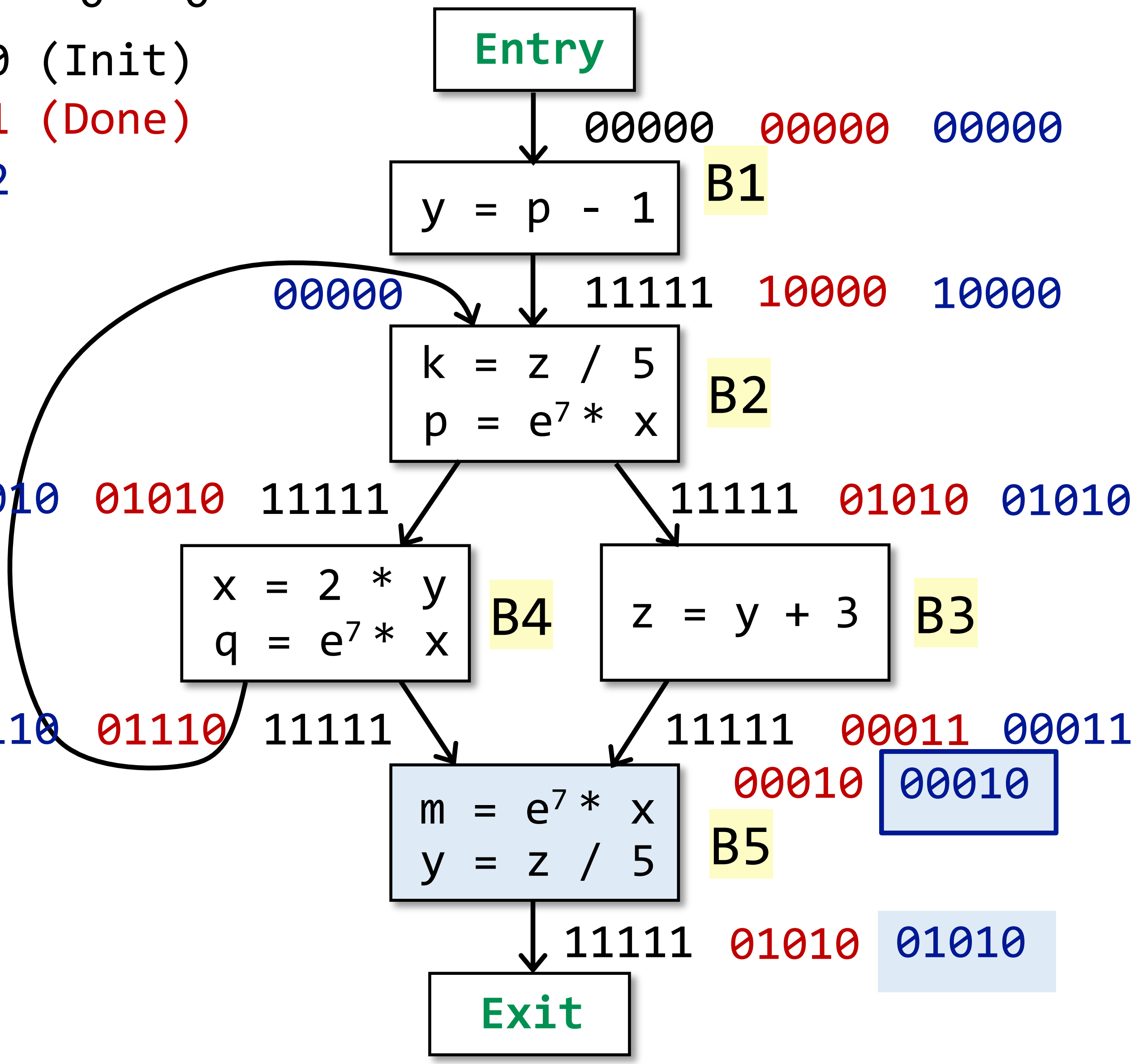


p-1 z/5 2*y e^7*x y+3
0 0 0 0 0

Iteration 0 (Init)

Iteration 1 (Done)

Iteration 2



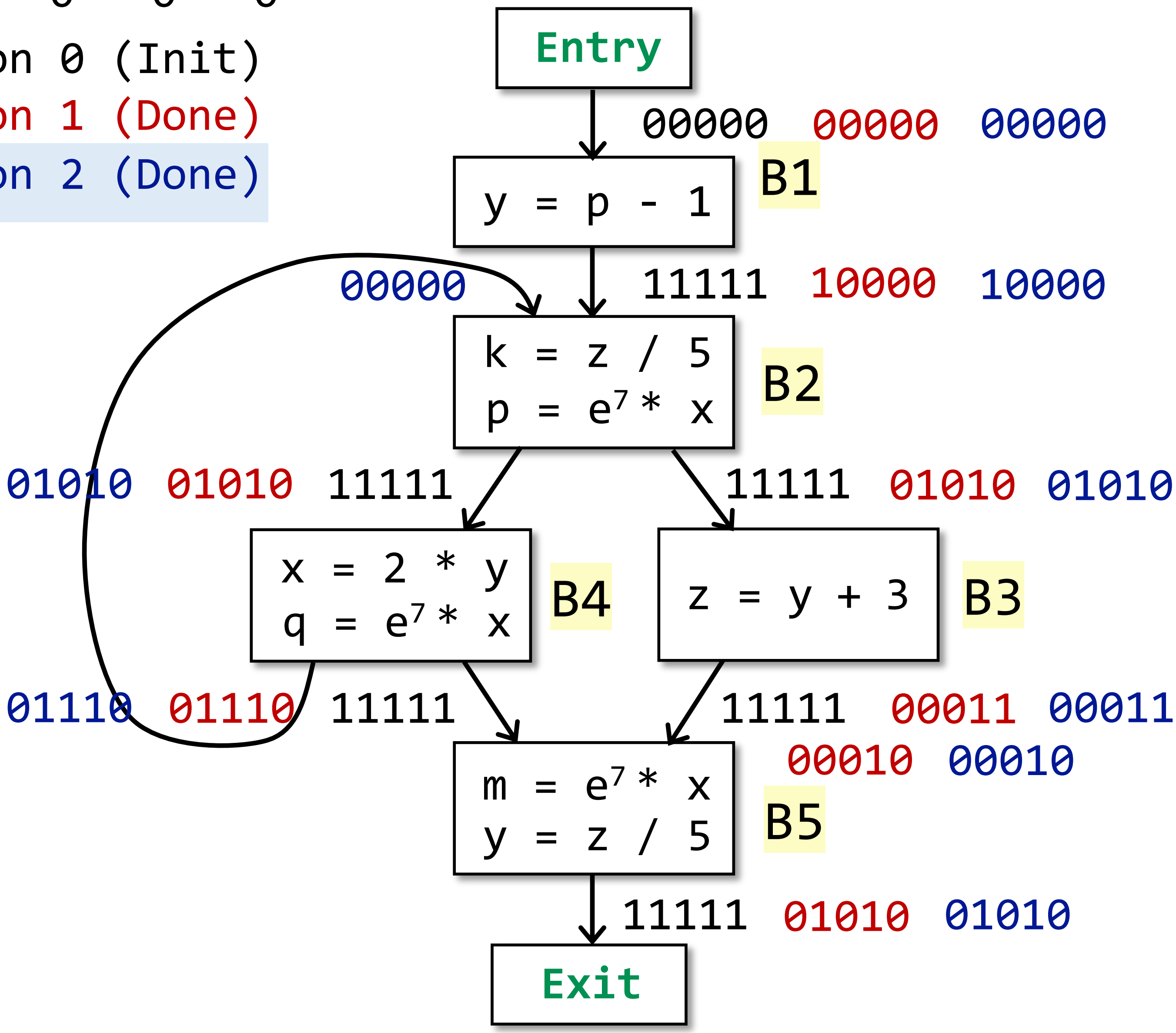
p-1 z/5 2*y e⁷*x y+3

0 0 0 0 0

Iteration 0 (Init)

Iteration 1 (Done)

Iteration 2 (Done)



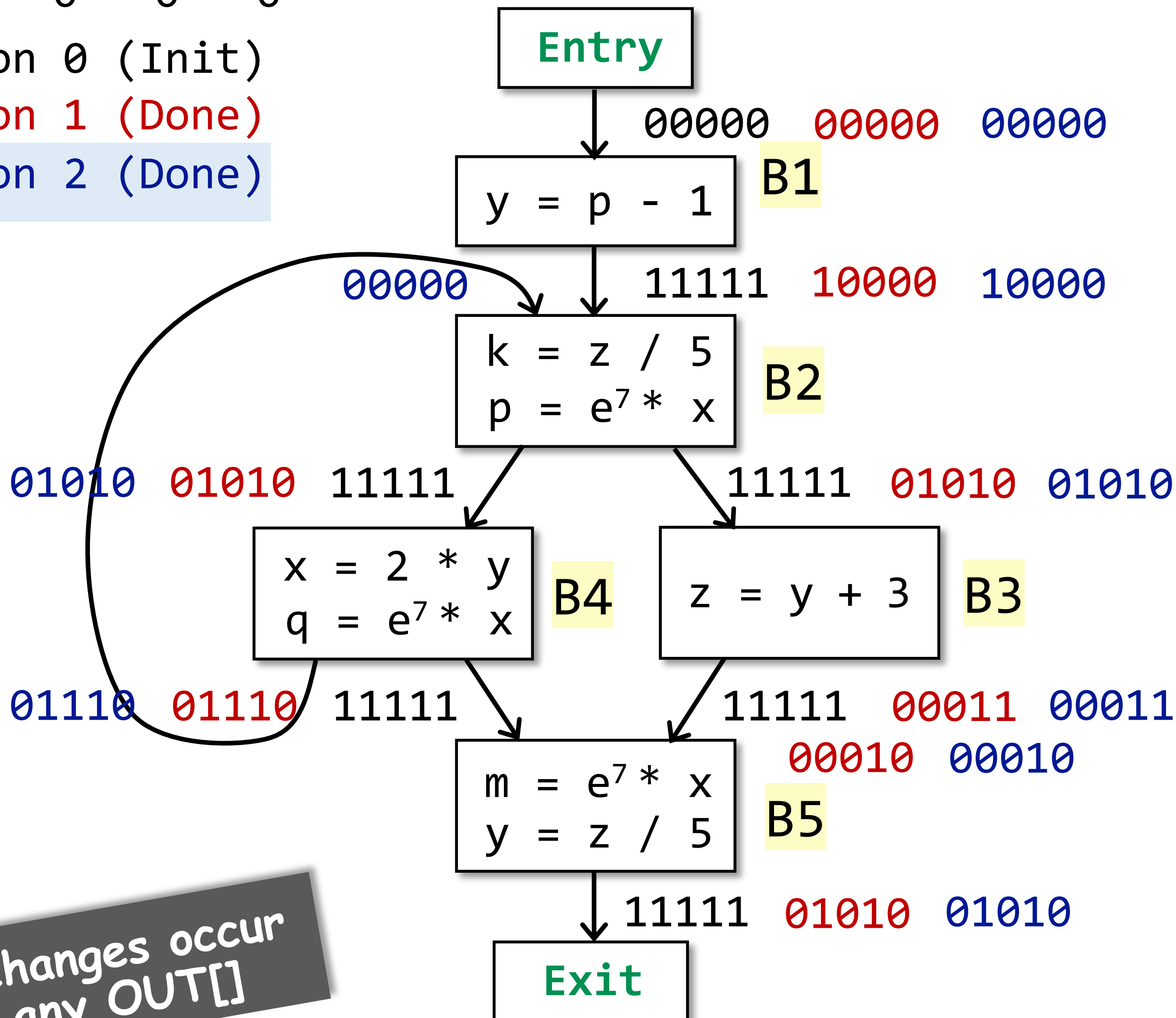
p-1 z/5 2*y e7*x y+3

0 0 0 0 0

Iteration 0 (Init)

Iteration 1 (Done)

Iteration 2 (Done)



No changes occur in any OUT[]

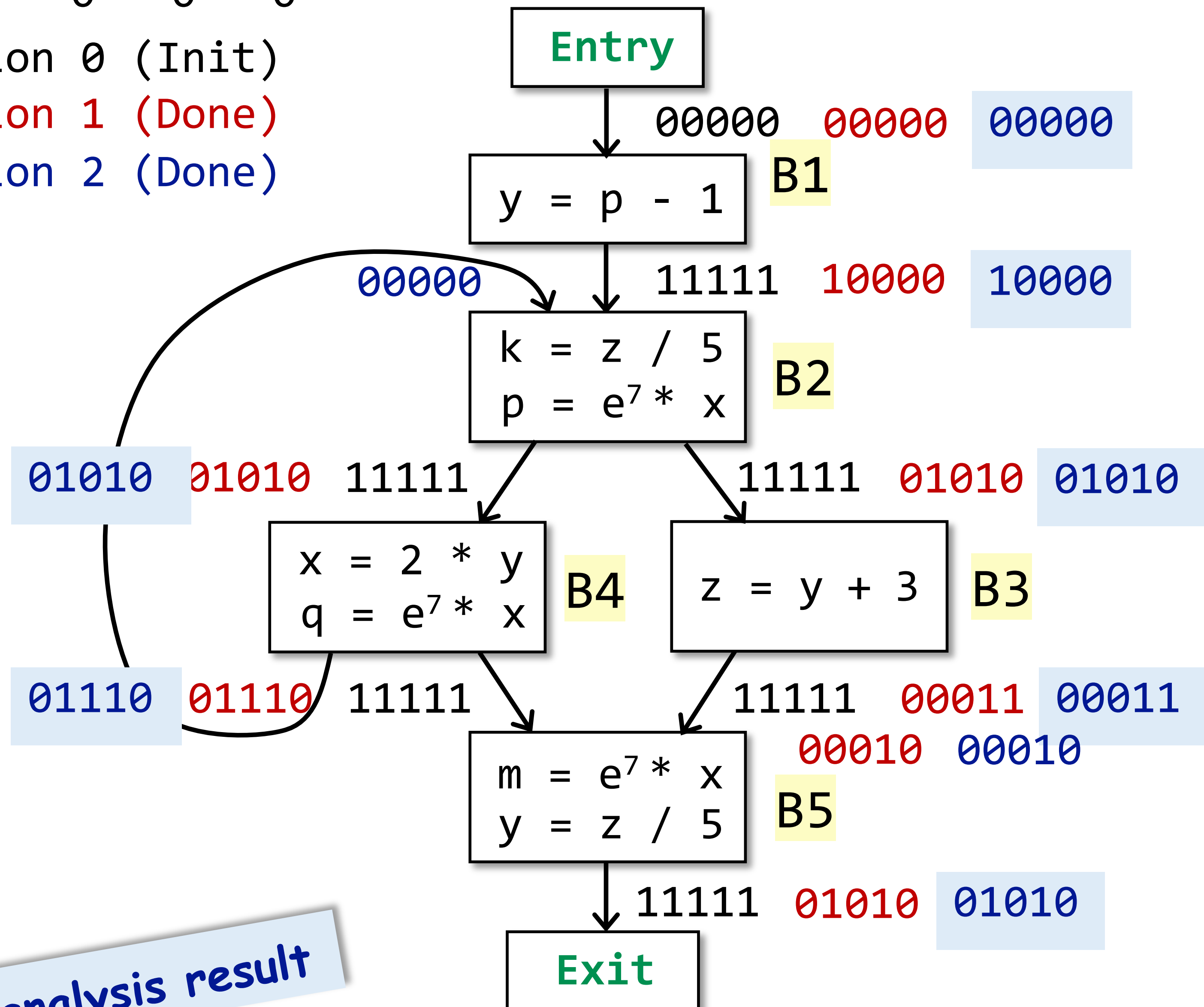
p-1 z/5 2*y e^7*x y+3

0 0 0 0 0

Iteration 0 (Init)

Iteration 1 (Done)

Iteration 2 (Done)



Final analysis result

Analysis Comparison

	Reaching Definitions	Live Variables	Available Expressions
Domain			
Direction			
May/Must			
Boundary			
Initialization			
Transfer function			
Meet			

Analysis Comparison

	Reaching Definitions	Live Variables	Available Expressions
Domain	Set of definitions	Set of variables	Set of expressions
Direction			
May/Must			
Boundary			
Initialization			
Transfer function			
Meet			

Analysis Comparison

	Reaching Definitions	Live Variables	Available Expressions
Domain	Set of definitions	Set of variables	Set of expressions
Direction	Forwards	Backwards	Forwards
May/Must			
Boundary			
Initialization			
Transfer function			
Meet			

Analysis Comparison

	Reaching Definitions	Live Variables	Available Expressions
Domain	Set of definitions	Set of variables	Set of expressions
Direction	Forwards	Backwards	Forwards
May/Must	May	May	Must
Boundary			
Initialization			
Transfer function			
Meet			

Analysis Comparison

	Reaching Definitions	Live Variables	Available Expressions
Domain	Set of definitions	Set of variables	Set of expressions
Direction	Forwards	Backwards	Forwards
May/Must	May	May	Must
Boundary	OUT[entry] = ∅	IN[exit] = ∅	OUT[entry] = ∅
Initialization			
Transfer function			
Meet			

Analysis Comparison

	Reaching Definitions	Live Variables	Available Expressions
Domain	Set of definitions	Set of variables	Set of expressions
Direction	Forwards	Backwards	Forwards
May/Must	May	May	Must
Boundary	OUT[entry] = ∅	IN[exit] = ∅	OUT[entry] = ∅
Initialization	OUT[B] = ∅	IN[B] = ∅	OUT[B] = U
Transfer function			
Meet			

Analysis Comparison

	Reaching Definitions	Live Variables	Available Expressions
Domain	Set of definitions	Set of variables	Set of expressions
Direction	Forwards	Backwards	Forwards
May/Must	May	May	Must
Boundary	OUT[entry] = ∅	IN[exit] = ∅	OUT[entry] = ∅
Initialization	OUT[B] = ∅	IN[B] = ∅	OUT[B] = U
Transfer function	OUT = gen U (IN - kill)		
Meet			

Analysis Comparison

	Reaching Definitions	Live Variables	Available Expressions
Domain	Set of definitions	Set of variables	Set of expressions
Direction	Forwards	Backwards	Forwards
May/Must	May	May	Must
Boundary	OUT[entry] = ∅	IN[exit] = ∅	OUT[entry] = ∅
Initialization	OUT[B] = ∅	IN[B] = ∅	OUT[B] = U
Transfer function	OUT = gen U (IN - kill)		
Meet	U	U	∩

According to the meaning of the analysis

Analysis Comparison

	Reaching Definitions	Live Variables	Available Expressions
Domain	Set of definitions	Set of variables	Set of expressions
Direction	Forwards	Backwards	Forwards
May/Must	May	May	Must
Boundary	OUT[entry] = ∅	IN[exit] = ∅	OUT[entry] = ∅
Initialization	OUT[B] = ∅	IN[B] = ∅	OUT[B] = U
Transfer function	OUT = gen U (IN - kill)		
Meet	∪	∪	∩

Analysis Comparison

According to the meaning of the analysis
We'll draw a theoretical framework to systematically explain them in next lectures

	Reaching Definitions	Live Variables	Available Expressions
Domain	Set of definitions	Set of variables	Set of expressions
Direction	Forwards	Backwards	Forwards
May/Must	May	May	Must
Boundary	OUT[entry] = ∅	IN[exit] = ∅	OUT[entry] = ∅
Initialization	OUT[B] = ∅	IN[B] = ∅	OUT[B] = U
Transfer function	OUT = gen U (IN - kill)		
Meet	⋃	⋃	⋂

Summary

1. Overview of Data Flow Analysis
2. Preliminaries of Data Flow Analysis
3. Reaching Definitions Analysis
4. Live Variables Analysis
5. Available Expressions Analysis

The X You Need To Understand in This Lecture

- Understand the three data flow analyses:
  - reaching definitions
  - live variables
  - available expressions
- Can tell the differences and similarities of the three data flow analyses
- Understand the iterative algorithm and can tell why it is able to terminate

软件分析

南京大学
计算机科学与技术系
程序设计语言与静态分析研究组
李樾 谭添