Assignment 2
ECE/CS 5544 Spring 2023
Group : Shambhavi Kuthe, Rohit Mehta
PID : shambhavianil, rohitnm

Part 1: Dataflow Analysis

2.1 Iterative Framework

Our implementation of the iterative framework which takes input parameters of a pass:

- Domain - set of expressions/ variables/ definitions
- Domain size - Size of the domain
- Direction - FORWARD/BACKWARD
- Transfer function - OUT = f(IN) or IN = f(OUT)
- Meet Operation - UNION/INTERSECTION
- Boundary Conditions - e.g. null
- Initial Conditions - e.g. null

With the help of the above pass specific data, we iteratively perform the analysis by checking if the previous output matches the current output i.e. until the analysis converges

The following functions are used to successfully implement the analysis:

meetFn() - Stores output/input of predecessor/successor blocks and applies a meet operator on them. Uses OR operator for UNION and AND operator for INTERSECTION
extractPredSuccBB() - Extracts predecessor and successor blocks of a given basic block
BBinit() - Initializes the attributes of basic blocks
flowOrder() - Stores the basic blocks in separate vectors for forwards and backwards propagation
dataflowAnalysis() - Function that performs dataflow analysis iteratively until the output converges and keeps count of the number of iterations

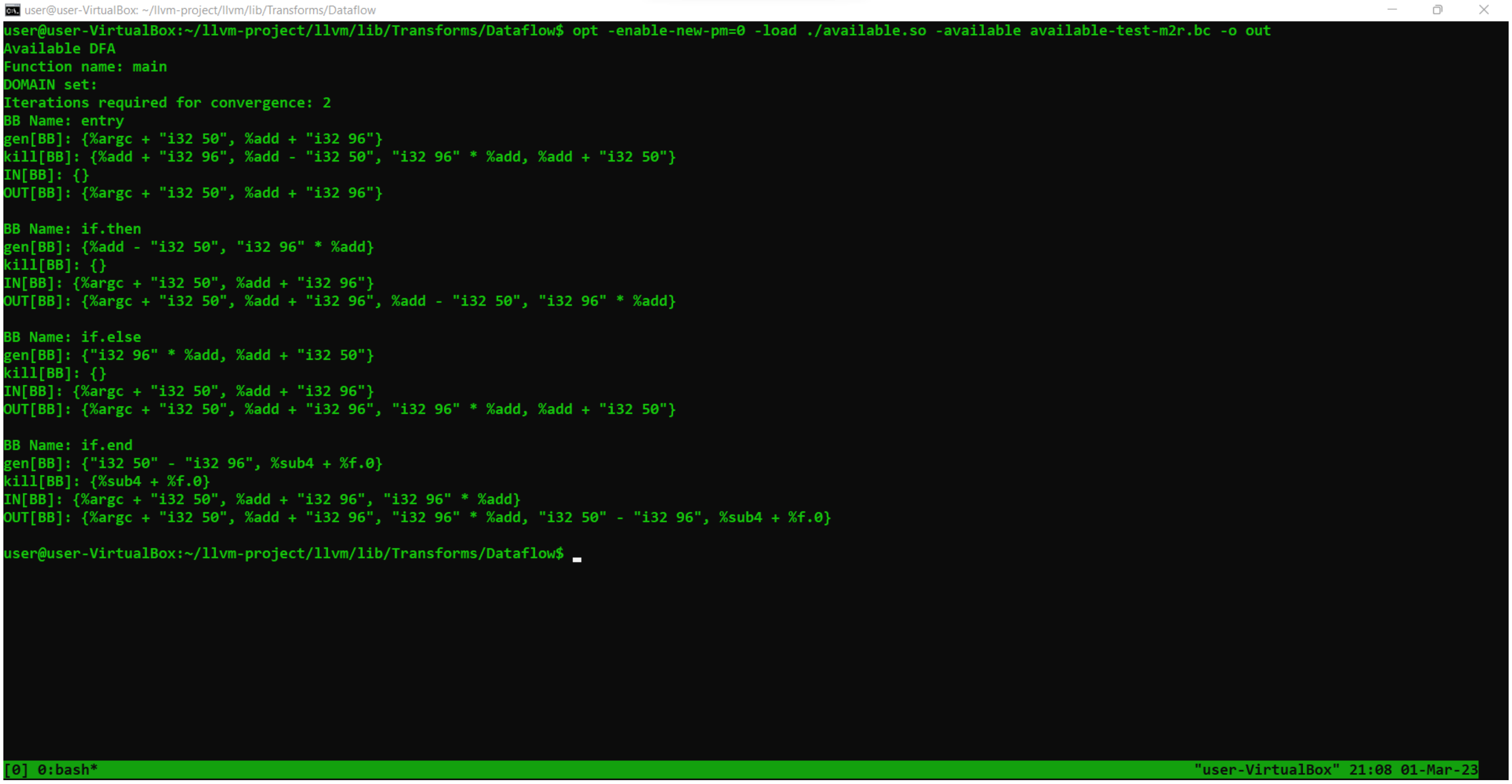
2.2 Analysis Pass
2.2.1 Available Expressions

Domain	Expressions
Direction	FORWARD
Transfer function	Gen **U** (IN - Kill)
Boundary condition	OUT[entry] = null
Initial condition	OUT[B] = **U**

Meet Operator	∩
OUT[BB]	OUT[B] = **fn**(IN[B])
IN[BB]	∩(Predecessor outputs)

We populate a domain vector with all the expressions in a basic block, evaluate boundary and initial conditions and then perform the analysis by passing the appropriate parameters to the dataflow analysis function. We store the result after each iteration and then display the IN, OUT, KILL and GEN sets of the final output. BitVectors have been used to keep a track of all the above mentioned sets. The dataflow analysis framework incorporates the transfer function passed to it and gives the final output.

Output:


user@user-VirtualBox: ~/llvm-project/llvm/lib/Transforms/Dataflow

```
user@user-VirtualBox:~/llvm-project/llvm/lib/Transforms/Dataflow$ opt -enable-new-pm=0 -load ./available.so -available available-test-m2r.bc -o out
Available DFA
Function name: main
DOMAIN set:
Iterations required for convergence: 2
BB Name: entry
gen[BB]: {%argc + "i32 50", %add + "i32 96"}
kill[BB]: {%add + "i32 96", %add - "i32 50", "i32 96" * %add, %add + "i32 50"}
IN[BB]: {}
OUT[BB]: {%argc + "i32 50", %add + "i32 96"}

BB Name: if.then
gen[BB]: {%add - "i32 50", "i32 96" * %add}
kill[BB]: {}
IN[BB]: {%argc + "i32 50", %add + "i32 96"}
OUT[BB]: {%argc + "i32 50", %add + "i32 96", %add - "i32 50", "i32 96" * %add}

BB Name: if.else
gen[BB]: {"i32 96" * %add, %add + "i32 50"}
kill[BB]: {}
IN[BB]: {%argc + "i32 50", %add + "i32 96"}
OUT[BB]: {%argc + "i32 50", %add + "i32 96", "i32 96" * %add, %add + "i32 50"}

BB Name: if.end
gen[BB]: {"i32 50" - "i32 96", %sub4 + %f.0}
kill[BB]: {%sub4 + %f.0}
IN[BB]: {%argc + "i32 50", %add + "i32 96", "i32 96" * %add}
OUT[BB]: {%argc + "i32 50", %add + "i32 96", "i32 96" * %add, "i32 50" - "i32 96", %sub4 + %f.0}

user@user-VirtualBox:~/llvm-project/llvm/lib/Transforms/Dataflow$ 
```

```
[0] 0:bash*                                                                                                              "user-VirtualBox" 21:08 01-Mar-23
```

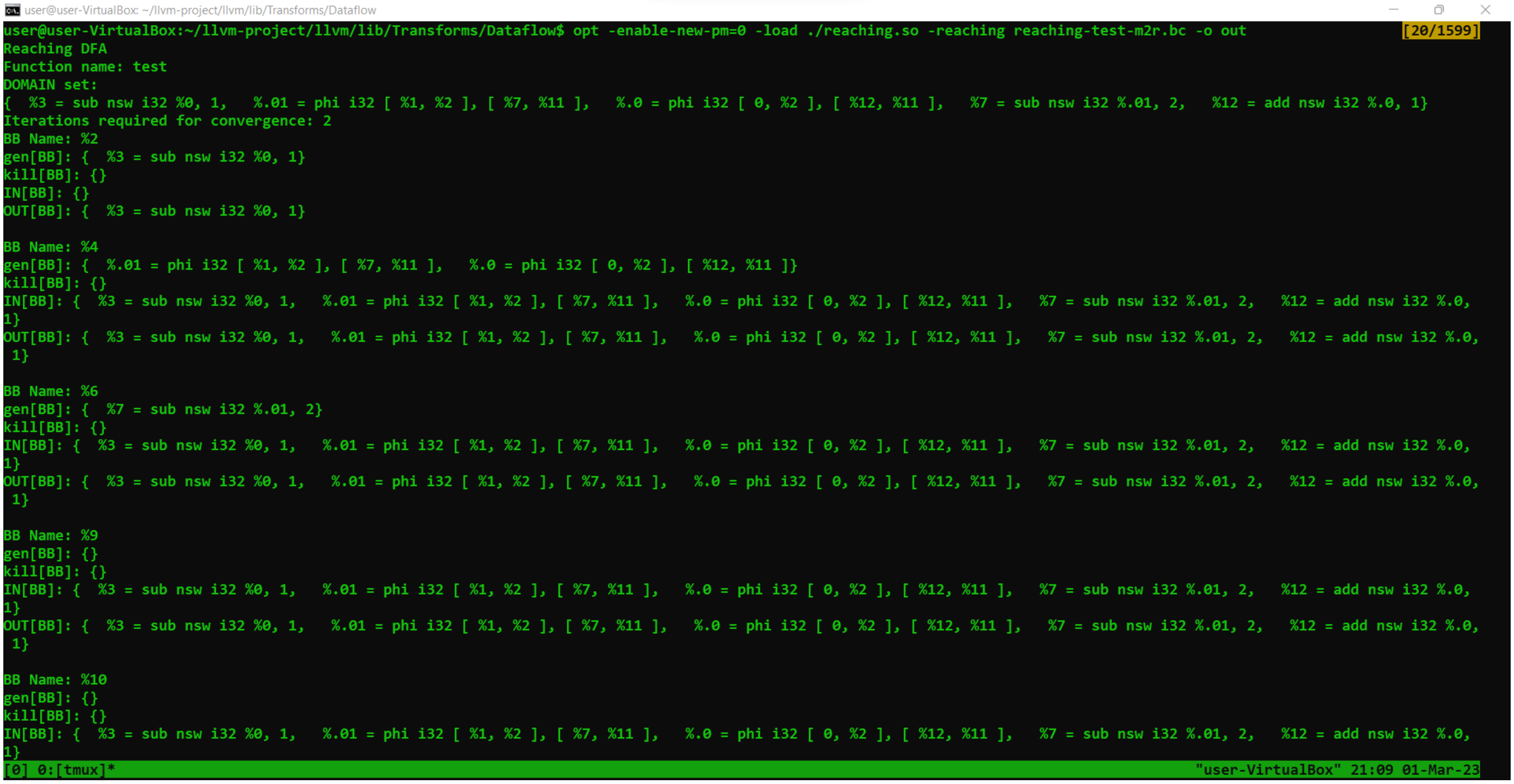
2.2.2 Reaching Definitions

Domain	Definitions

Direction	FORWARD
Transfer function	Gen **U** (IN - Kill)
Boundary condition	OUT[Entry] = null
Initial condition	OUT[B] = **U**
Meet Operator	∩
OUT[BB]	OUT[B] = **fn**(IN[B])
IN[BB]	∩(Predecessor outputs)

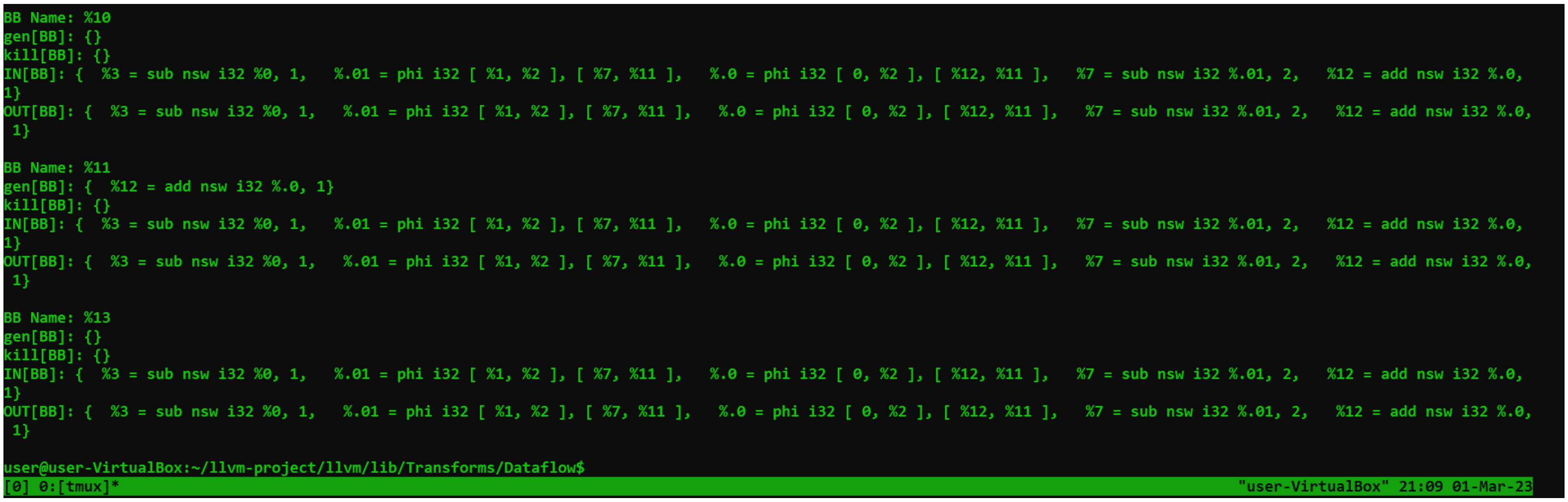
Reaching definitions are the definitions that reach a point p if there exists at least one path from definition to point p. We populate a domain vector with all the instructions of a basic block, evaluate boundary and initial conditions and then perform the analysis by passing the appropriate parameters to the dataflow analysis function. We store the result after each iteration and then display the IN, OUT, KILL and GEN sets of the final output. BitVectors have been used to keep a track of all the above mentioned sets. The dataflow analysis framework incorporates the transfer function passed to it and gives the final output. Output in SSA form is taken into consideration here.

Output



user@user-VirtualBox: ~/llvm-project/llvm/lib/Transforms/Dataflow

```
user@user-VirtualBox:~/llvm-project/llvm/lib/Transforms/Dataflow$ opt -enable-new-pm=0 -load ./reaching.so -reaching reaching-test-m2r.bc -o out          [20/1599]
Reaching DFA
Function name: test
DOMAIN set:
{  %3 = sub nsw i32 %0, 1,   %.01 = phi i32 [ %1, %2 ], [ %7, %11 ],   %.0 = phi i32 [ 0, %2 ], [ %12, %11 ],   %7 = sub nsw i32 %.01, 2,   %12 = add nsw i32 %.0, 1}
Iterations required for convergence: 2
BB Name: %2
gen[BB]: {  %3 = sub nsw i32 %0, 1}
kill[BB]: {}
IN[BB]: {}
OUT[BB]: {  %3 = sub nsw i32 %0, 1}

BB Name: %4
gen[BB]: {  %.01 = phi i32 [ %1, %2 ], [ %7, %11 ],   %.0 = phi i32 [ 0, %2 ], [ %12, %11 ]}
kill[BB]: {}
IN[BB]: {  %3 = sub nsw i32 %0, 1,   %.01 = phi i32 [ %1, %2 ], [ %7, %11 ],   %.0 = phi i32 [ 0, %2 ], [ %12, %11 ],   %7 = sub nsw i32 %.01, 2,   %12 = add nsw i32 %.0,
1}
OUT[BB]: {  %3 = sub nsw i32 %0, 1,   %.01 = phi i32 [ %1, %2 ], [ %7, %11 ],   %.0 = phi i32 [ 0, %2 ], [ %12, %11 ],   %7 = sub nsw i32 %.01, 2,   %12 = add nsw i32 %.0,
 1}

BB Name: %6
gen[BB]: {  %7 = sub nsw i32 %.01, 2}
kill[BB]: {}
IN[BB]: {  %3 = sub nsw i32 %0, 1,   %.01 = phi i32 [ %1, %2 ], [ %7, %11 ],   %.0 = phi i32 [ 0, %2 ], [ %12, %11 ],   %7 = sub nsw i32 %.01, 2,   %12 = add nsw i32 %.0,
1}
OUT[BB]: {  %3 = sub nsw i32 %0, 1,   %.01 = phi i32 [ %1, %2 ], [ %7, %11 ],   %.0 = phi i32 [ 0, %2 ], [ %12, %11 ],   %7 = sub nsw i32 %.01, 2,   %12 = add nsw i32 %.0,
 1}

BB Name: %9
gen[BB]: {}
kill[BB]: {}
IN[BB]: {  %3 = sub nsw i32 %0, 1,   %.01 = phi i32 [ %1, %2 ], [ %7, %11 ],   %.0 = phi i32 [ 0, %2 ], [ %12, %11 ],   %7 = sub nsw i32 %.01, 2,   %12 = add nsw i32 %.0,
1}
OUT[BB]: {  %3 = sub nsw i32 %0, 1,   %.01 = phi i32 [ %1, %2 ], [ %7, %11 ],   %.0 = phi i32 [ 0, %2 ], [ %12, %11 ],   %7 = sub nsw i32 %.01, 2,   %12 = add nsw i32 %.0,
 1}

BB Name: %10
gen[BB]: {}
kill[BB]: {}
IN[BB]: {  %3 = sub nsw i32 %0, 1,   %.01 = phi i32 [ %1, %2 ], [ %7, %11 ],   %.0 = phi i32 [ 0, %2 ], [ %12, %11 ],   %7 = sub nsw i32 %.01, 2,   %12 = add nsw i32 %.0,
1}
[0] 0:[tmux]*                                                                                                  "user-VirtualBox" 21:09 01-Mar-23
```



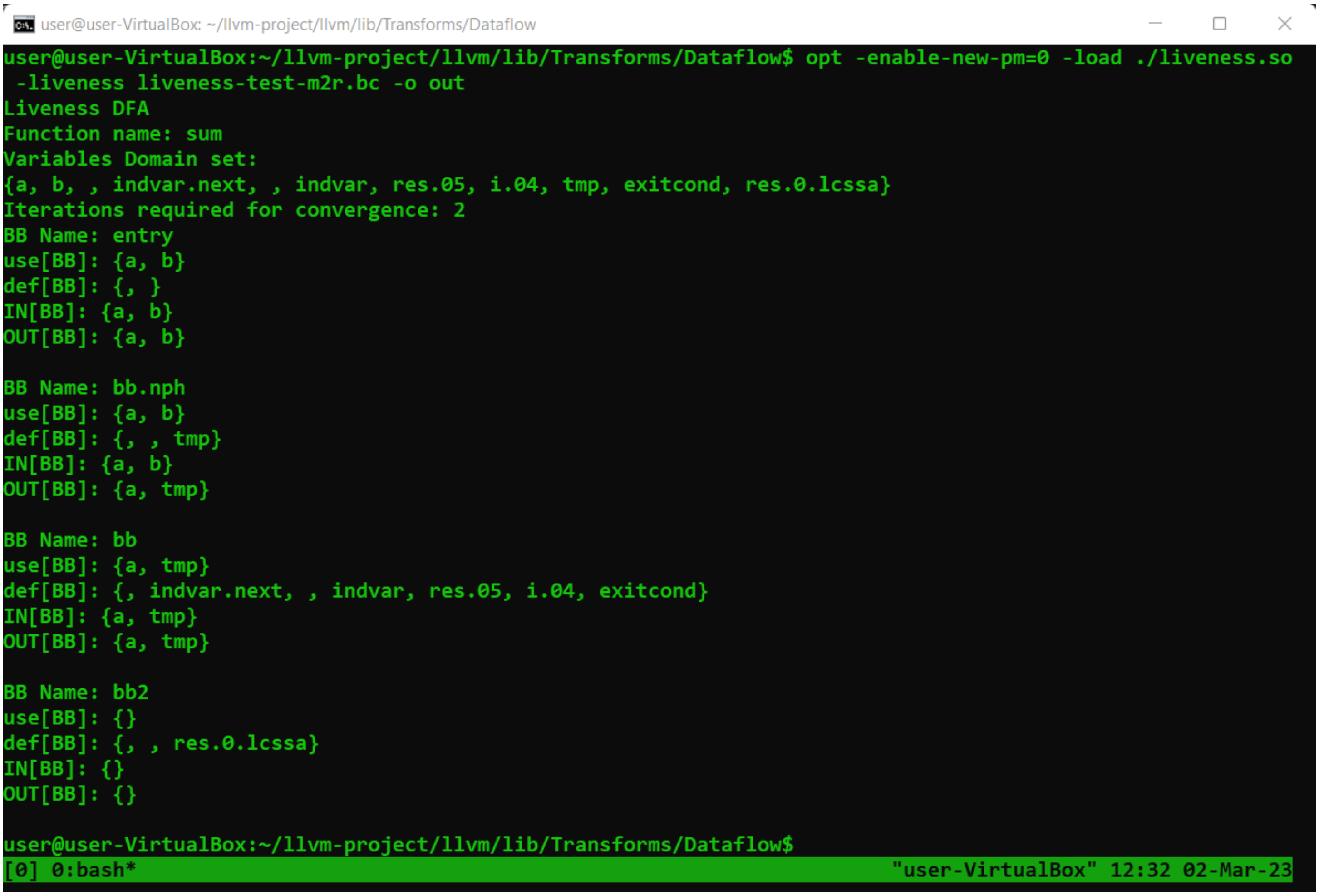
```
BB Name: %10
gen[BB]: {}
kill[BB]: {}
IN[BB]: {  %3 = sub nsw i32 %0, 1,   %.01 = phi i32 [ %1, %2 ], [ %7, %11 ],   %.0 = phi i32 [ 0, %2 ], [ %12, %11 ],   %7 = sub nsw i32 %.01, 2,   %12 = add nsw i32 %.0,
1}
OUT[BB]: {  %3 = sub nsw i32 %0, 1,   %.01 = phi i32 [ %1, %2 ], [ %7, %11 ],   %.0 = phi i32 [ 0, %2 ], [ %12, %11 ],   %7 = sub nsw i32 %.01, 2,   %12 = add nsw i32 %.0,
 1}

BB Name: %11
gen[BB]: {  %12 = add nsw i32 %.0, 1}
kill[BB]: {}
IN[BB]: {  %3 = sub nsw i32 %0, 1,   %.01 = phi i32 [ %1, %2 ], [ %7, %11 ],   %.0 = phi i32 [ 0, %2 ], [ %12, %11 ],   %7 = sub nsw i32 %.01, 2,   %12 = add nsw i32 %.0,
1}
OUT[BB]: {  %3 = sub nsw i32 %0, 1,   %.01 = phi i32 [ %1, %2 ], [ %7, %11 ],   %.0 = phi i32 [ 0, %2 ], [ %12, %11 ],   %7 = sub nsw i32 %.01, 2,   %12 = add nsw i32 %.0,
 1}

BB Name: %13
gen[BB]: {}
kill[BB]: {}
IN[BB]: {  %3 = sub nsw i32 %0, 1,   %.01 = phi i32 [ %1, %2 ], [ %7, %11 ],   %.0 = phi i32 [ 0, %2 ], [ %12, %11 ],   %7 = sub nsw i32 %.01, 2,   %12 = add nsw i32 %.0,
1}
OUT[BB]: {  %3 = sub nsw i32 %0, 1,   %.01 = phi i32 [ %1, %2 ], [ %7, %11 ],   %.0 = phi i32 [ 0, %2 ], [ %12, %11 ],   %7 = sub nsw i32 %.01, 2,   %12 = add nsw i32 %.0,
 1}

user@user-VirtualBox:~/llvm-project/llvm/lib/Transforms/Dataflow$
[0] 0:[tmux]*                                                                                                                    "user-VirtualBox" 21:09 01-Mar-23
```

2.2.3 Liveness Analysis

Domain	Variables
Direction	BACKWARD
Transfer function	Use **U** (IN - Def)
Boundary condition	IN[Exit] = null
Initial condition	IN[B] = null
Meet Operator	**U**
OUT[BB]	**U**(Successor INs)
IN[BB]	null

A variable is live when it is used and is dead when it is defined. We populate a domain vector with all the arguments of a basic block including the phi nodes, evaluate boundary and initial conditions and then perform the analysis in the backwards direction by passing the appropriate parameters to the dataflow analysis function. We store the result after each iteration and then display the IN, OUT, USE and DEF sets of the final output. BitVectors have been used to keep a track of all the above mentioned sets. The dataflow analysis framework incorporates the transfer function passed to it and gives the final output. We have incorporated a value to the integer map to keep track of the instructions. For convenience, we print the domain set in string form.

Output


user@user-VirtualBox: ~/llvm-project/llvm/lib/Transforms/Dataflow

```
user@user-VirtualBox:~/llvm-project/llvm/lib/Transforms/Dataflow$ opt -enable-new-pm=0 -load ./liveness.so
 -liveness liveness-test-m2r.bc -o out
Liveness DFA
Function name: sum
Variables Domain set:
{a, b, , indvar.next, , indvar, res.05, i.04, tmp, exitcond, res.0.lcssa}
Iterations required for convergence: 2
BB Name: entry
use[BB]: {a, b}
def[BB]: {, }
IN[BB]: {a, b}
OUT[BB]: {a, b}

BB Name: bb.nph
use[BB]: {a, b}
def[BB]: {, , tmp}
IN[BB]: {a, b}
OUT[BB]: {a, tmp}

BB Name: bb
use[BB]: {a, tmp}
def[BB]: {, indvar.next, , indvar, res.05, i.04, exitcond}
IN[BB]: {a, tmp}
OUT[BB]: {a, tmp}

BB Name: bb2
use[BB]: {}
def[BB]: {, , res.0.lcssa}
IN[BB]: {}
OUT[BB]: {}

user@user-VirtualBox:~/llvm-project/llvm/lib/Transforms/Dataflow$
[0] 0:bash*                                                           "user-VirtualBox" 12:32 02-Mar-23
```