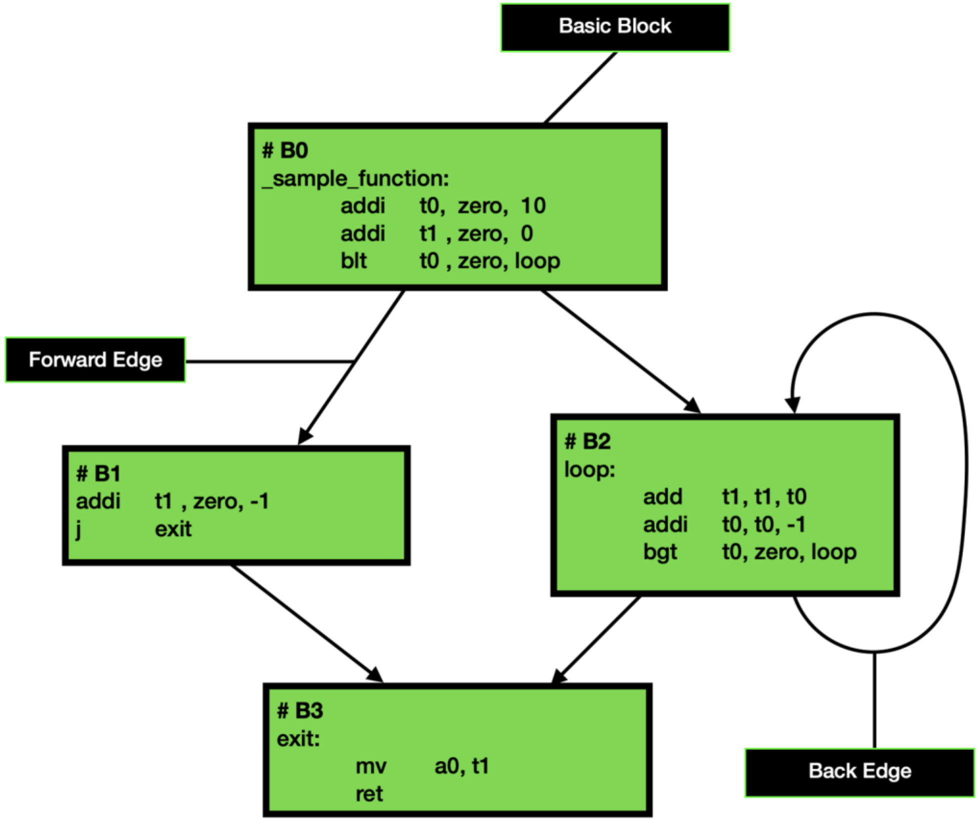
CMPUT 229

Lab #5: Control Flow Graph

What is a Control Flow Graph?

A Control Flow Graph (CFG) shows all the possible execution paths of a program.



Basic Blocks

The nodes of a CFG are basic blocks.

A basic block is a segment of code where instructions execute in order from start to finish.

Once execution enters a block, all instructions in the block must be executed.

Only the first instruction of a basic block can be the target for a branch or jump instruction.

```
sample_function:
        addi    t0, zero, 10
        addi    t1, zero, 0
        blt     t0, zero, loop
```

```
addi    t1, zero, -1
j       exit
```

```
loop:
        add     t1, t1, t0
        addi    t0, t0, -1
        bgt     t0, zero, loop
```

```
exit:
        mv      a0, t1
        ret
```

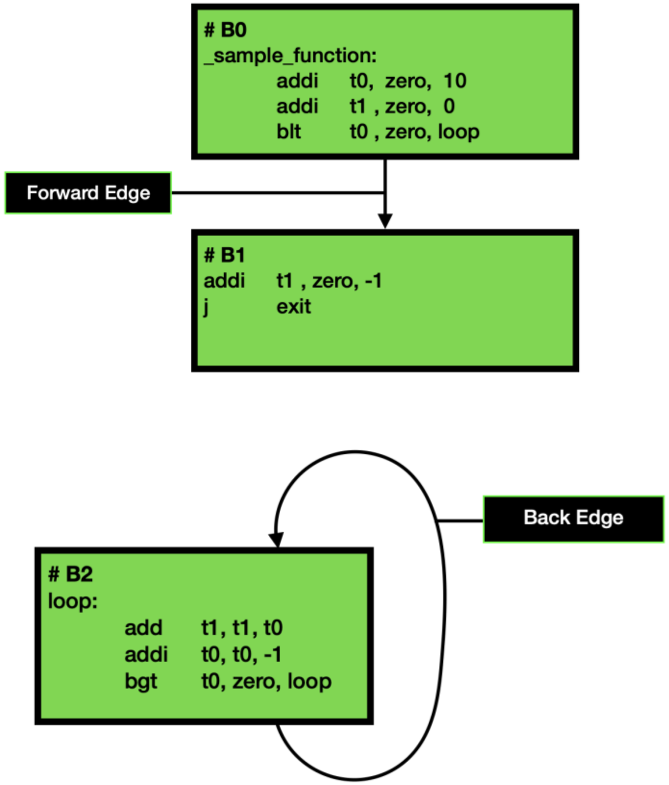
Edges

Edges denote directed control flow between basic blocks.

There are two types of edges:

- forward edge
- back edge.

Branches and jumps redirect control flow, creating multiple execution paths within a program.

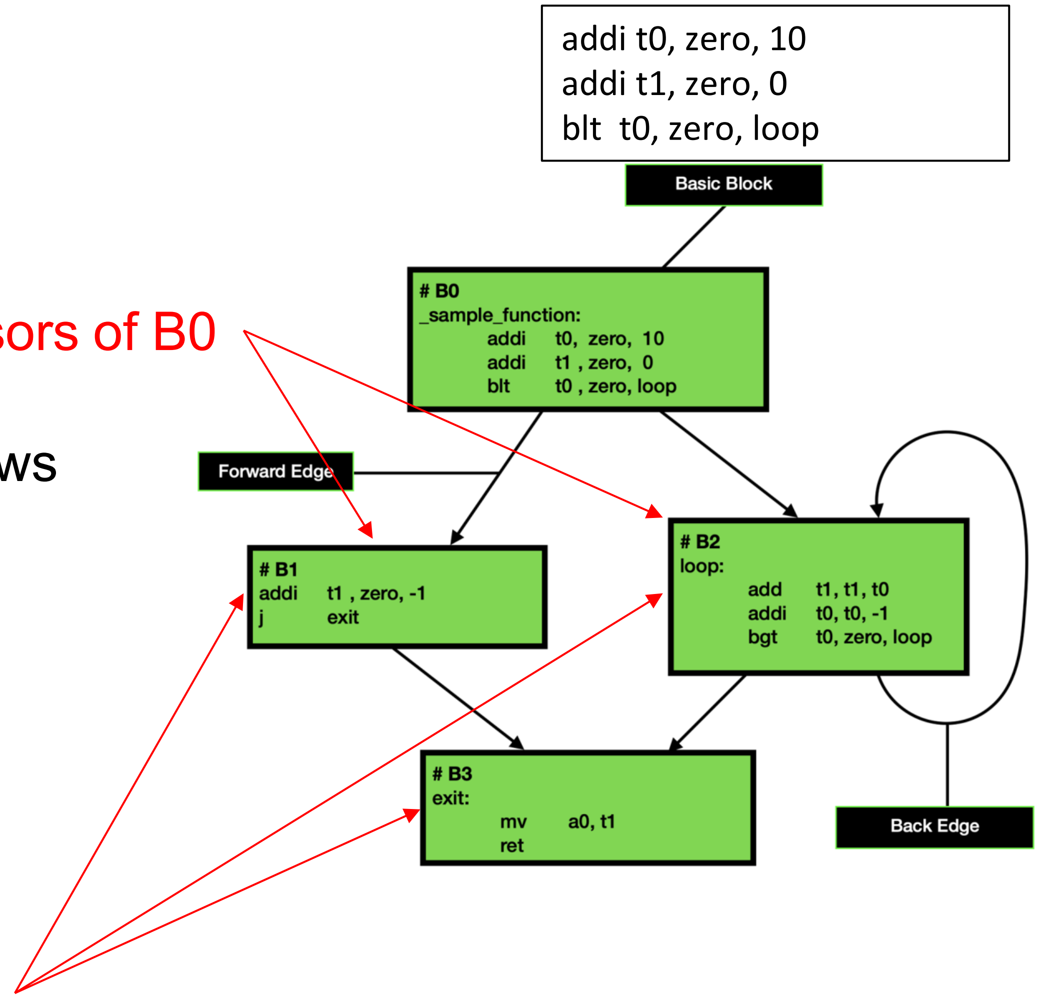


Successors & Predecessors

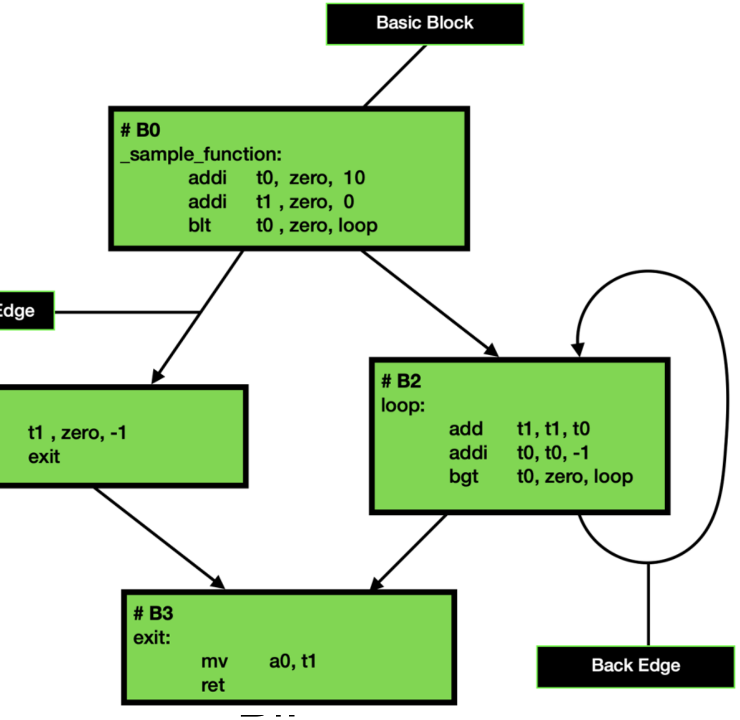
A successor of a block Bi is any block that can be reached from Bi.

An immediate successor of a block Bi directly follows Bi.

The concepts of predecessor and immediate predecessor are analogous.



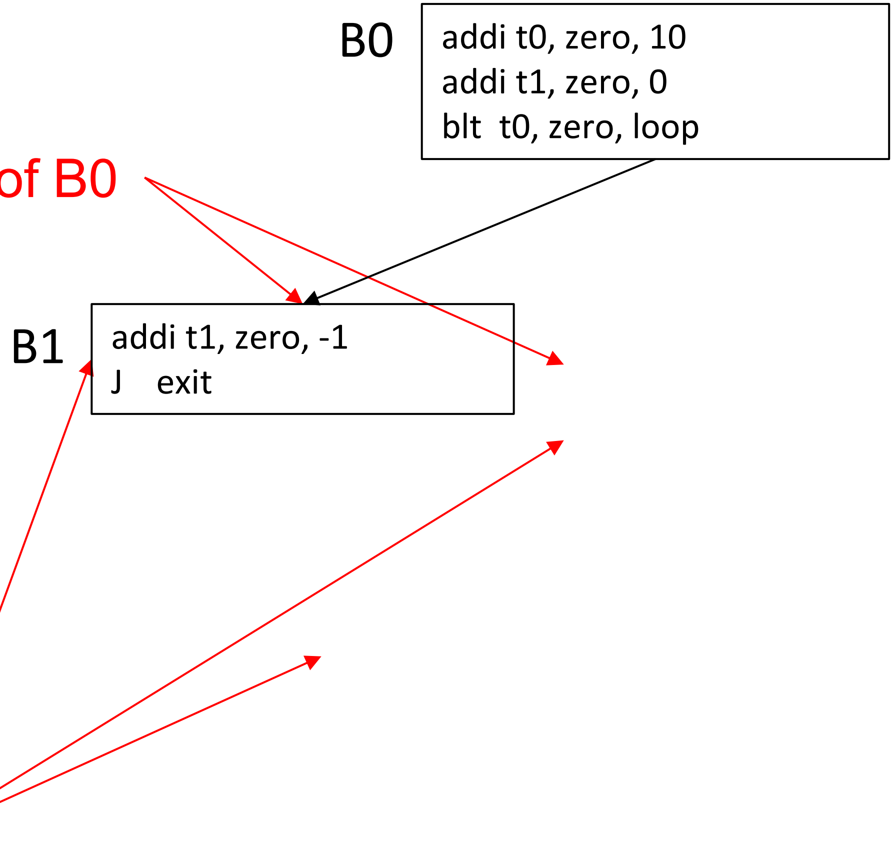
cessors & Predecessors



k Bi is any block that can be

Immediate successors of B0

sor of a block Bi directly follows



The concepts of predecessor and immediate predecessor are analogous.

Successors of B0

Assignment

Overview

Goal: to generate control flow information for a single function.

These functions will populate data structures that represent control flow constructs:

- basic blocks, edges, successors, and predecessors.

Each function’s implementation should accurately populate these custom data structures.

The **common.s** file initializes the data structures and runs tests on your implementation.
It also helps to parse the populated data structures.

Overview

- **Goal**: generate control flow information for a single function.
- **Task**: write functions that populate data structures that represent control flow constructs:
  - Basic blocks
  - Edges
  - Successors
  - Predecessors
- **Resources**: the **common.s** file:
  - initializes the data structures
  - runs tests on your implementation
  - helps to parse the populated data structures.

A Sample Input Function

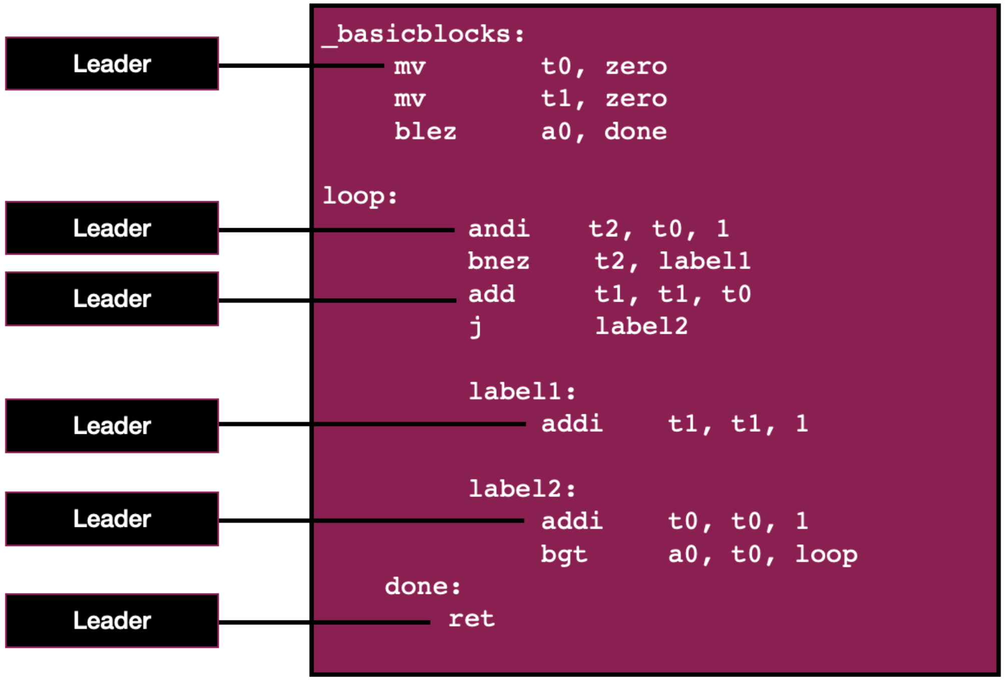
The **_basicblocks** function below serves as an example of an intended input for the lab. It will be used as a reference to explain the custom data structures.

```
_basicblocks:
    mv        t0, zero
    mv        t1, zero
    blez      a0, done

   loop:
        andi    t2, t0, 1
        bnez    t2, label1
        add     t1, t1, t0
        j       label2

        label1:
            addi    t1, t1, 1

        label2:
            addi    t0, t0, 1
            bgt     a0, t0, loop
   done:
       ret
```

Basic Block Leaders

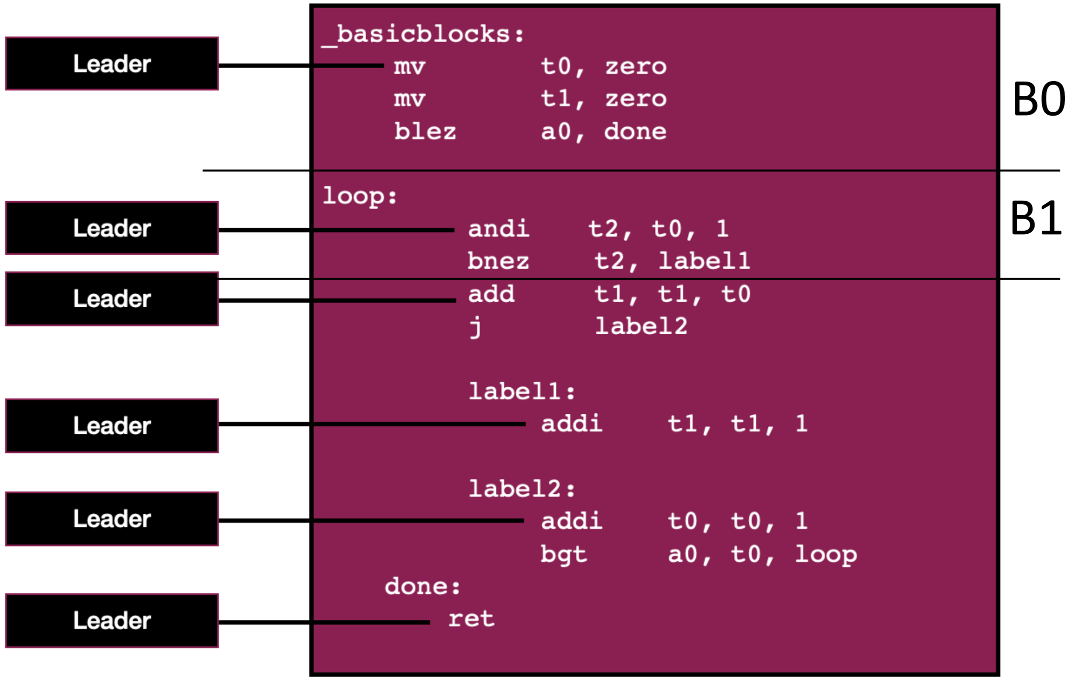
Basic-block leaders:

- The first statement in a function
- Any statement that is the target of a branch/jump
- Any statement that immediately follows a branch or jump



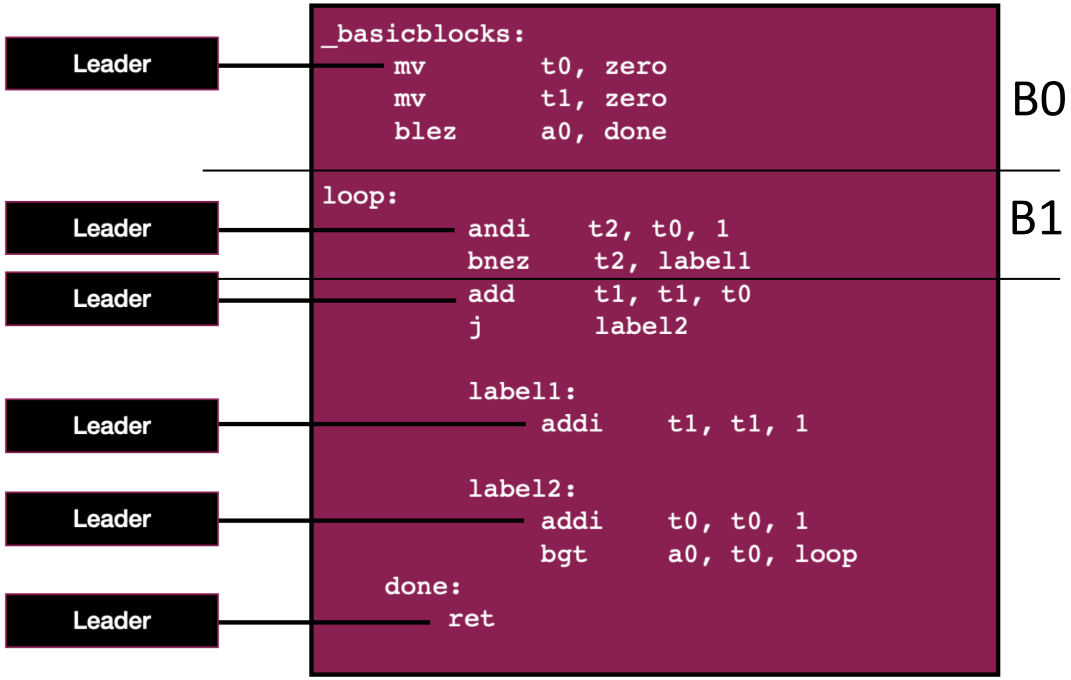
Forming Basic Blocks

A basic-block is a leader and all instructions after the leader up to, but not including the next leader.



Building the Control Flow Graph

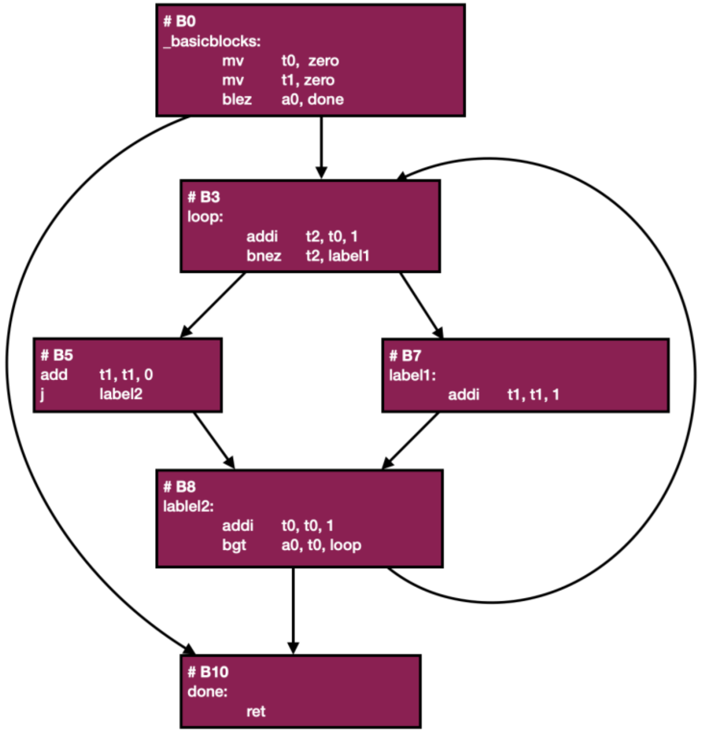
A basic-block is a leader and all instructions after the leader up to, but not including the next leader.



Basic Block Identifiers

To uniquely identify blocks, we use the offsets (in words) of their leaders from the function's base address in memory.

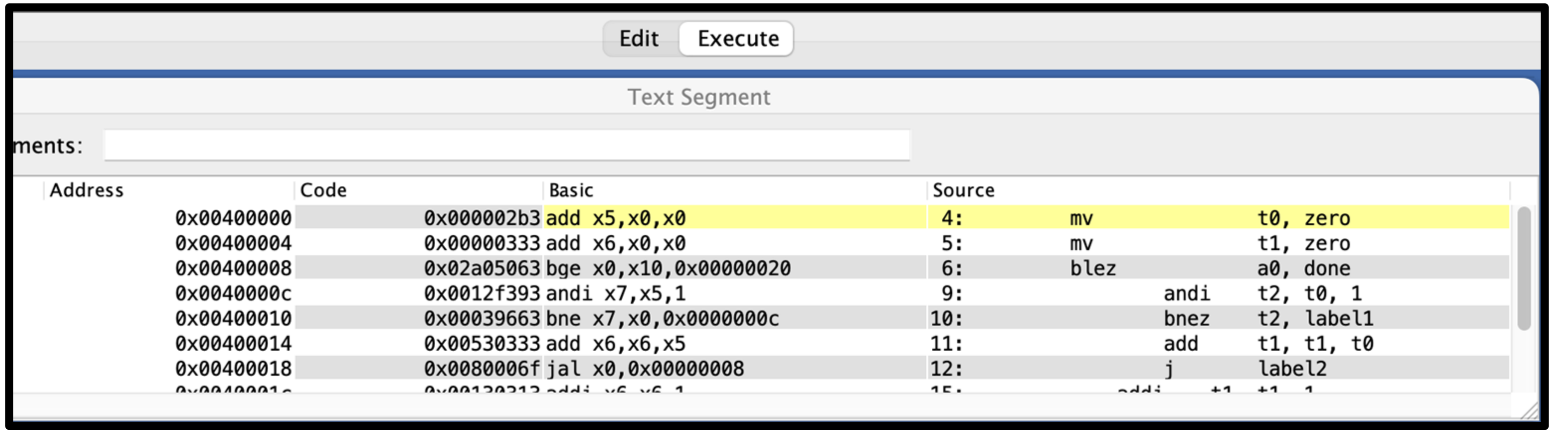
```
 Address        Hex              Instructions
00010000:  (000002b3)  add       t0, zero, zero
00010004:  (00000333)  add       t1, zero, zero
00010008:  (02a05063)  bge       zero, a0, 32

0001000c:  (0012f393)  andi      t2, t0, 1
00010010:  (00039663)  bne       t2, zero, 12

00010014:  (00530333)  add       t1, t1, t0
00010018:  (0080006f)  jal       zero, 8

0001001c:  (00130313)  addi      t1, t1, 1

00010020:  (00128293)  addi      t0, t0, 1
00010024:  (fea2c4e3)  blt       t0, a0, -24

00010028:  (00008067)  ret
```



E.g., The second block starts 3 words away from the function's base address so its ID is 3.

Pseudo Instructions and Labels

Pseudo-instructions in RISC-V simplify assembly code by representing higher-level operations.



For example, **'mv t0, zero'** translates to **'add t0, zero, zero'** during assembly, which copies the value 0 into the register **t0** as well.

Labels are symbolic pointers to specific instructions used for branch or jump offsets and do not appear in the executable code. They are also used to simplify assembly.

Decoding Instructions

RISC-V instruction formats are differentiated by their **opcode**, while **funct3** distinguishes instructions within each format.

31	30 25	24 21	20	19 15	14 12	11 8	7	6 0	
funct7		rs2		rs1	funct3	rd		opcode	R-type
imm[11:0]				rs1	funct3	rd		opcode	I-type
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode	S-type
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	J-type

Decoding the fourth instruction (0x0012f393) of the **_basicblocks** function as an example:

0x0012f393 = 0000 0000 0001 0010 1**111** 0011 1**001 0011**

opcode = **001 0011 (I-type** instruction)

funct3 = **111 (andi** instruction)

Lab #5: Control Flow Graph

Data Structures

instructionsArray

An array of words that contains the instructions of the input function.

```
_basicblocks:
    mv      t0, zero
    mv      t1, zero
    blez    a0, done

    loop:
        andi    t2, t0, 1
        bnez    t2, label1
        add     t1, t1, t0
        j       label2

        label1:
            addi    t1, t1, 1

        label2:
            addi    t0, t0, 1
            bgt     a0, t0, loop
    done:
        ret
```

```
 Address      Hex           Instructions
00010000:  (000002b3)  add     t0, zero, zero
00010004:  (00000333)  add     t1, zero, zero
00010008:  (02a05063)  bge     zero, a0, 32

0001000c:  (0012f393)  andi    t2, t0, 1
00010010:  (00039663)  bne     t2, zero, 12

00010014:  (00530333)  add     t1, t1, t0
00010018:  (0080006f)  jal     zero, 8

0001001c:  (00130313)  addi    t1, t1, 1

00010020:  (00128293)  addi    t0, t0, 1
00010024:  (fea2c4e3)  blt     t0, a0, -24

00010028:  (00008067)  ret
```

0x000002b3
0x00000333
0x02a05063
0x0012f393
0x00039663
0x00530333
0x0080006f
0x00130313
0x00128293
0xFEA2C4E3
0x00008067
0xFFFFFFFF

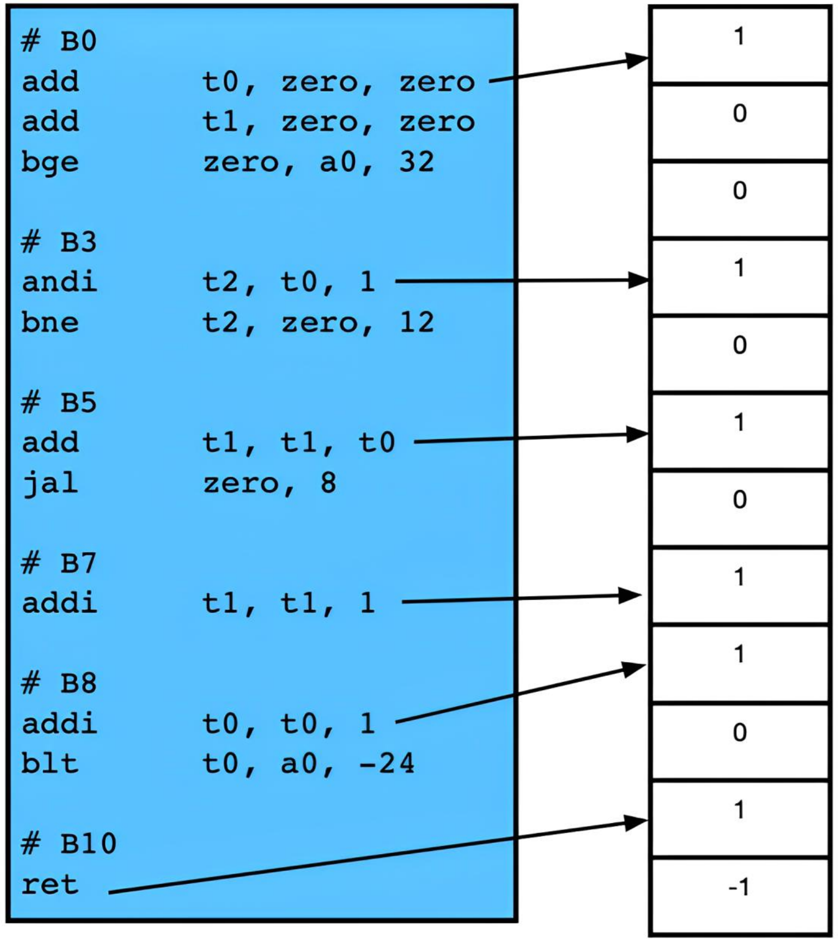
Ends with a sentinel value of -1.

leadersArray

An array of bytes representing the leadership status of each instruction.

Each byte corresponds directly to an instruction in the **instructionsArray**.

If an instruction is a leader, its corresponding byte will be 1; otherwise, it will be 0.

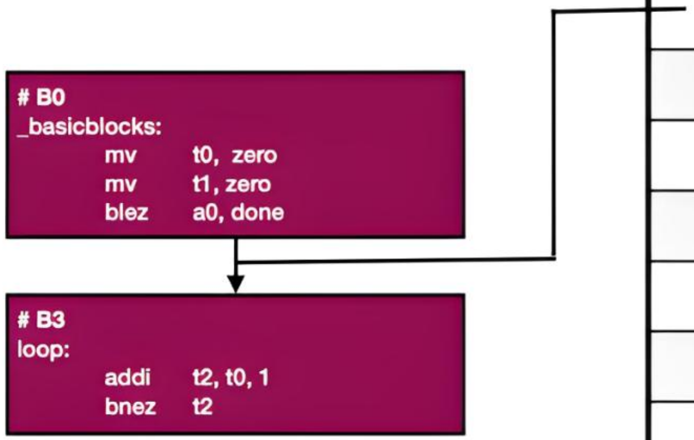


Ends with a sentinel value of -1.

edgesList

A 2D byte array which stores all the edges within a function's CFG.

In an entry: the first byte indicates the source, and the second byte indicates the target.



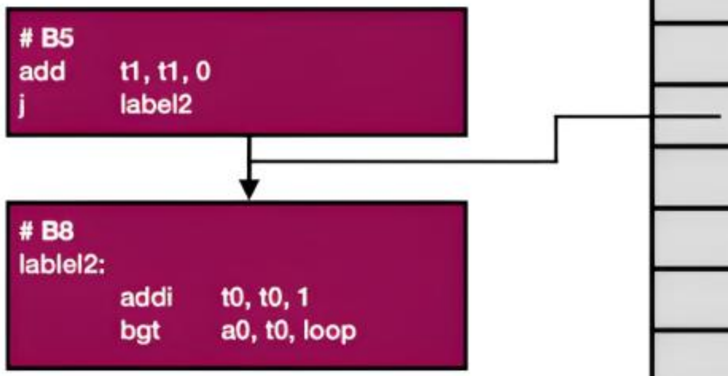
Source	Target
0	3
0	10
3	5
3	7
5	8
7	8
8	3
8	10
-1	-1

Ends with a sentinel value of -1 in both bytes.

successorsTable

A 2D byte array storing immediate successors for each basic block in a function's CFG.

In an entry: the first byte indicates the block, followed by 2 bytes for its **successorsList**.



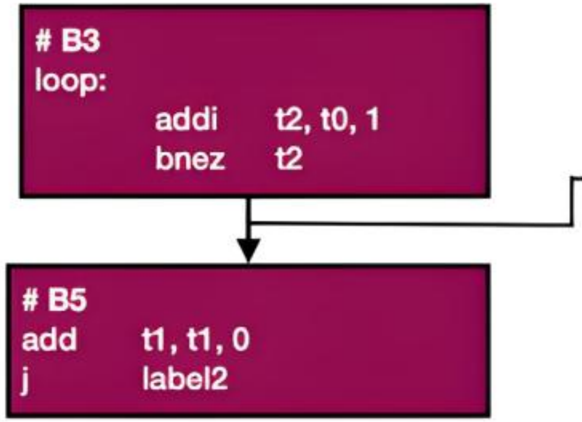
Basic Block	Successors		
0	3	10	-1
3	5	7	-1
5	8	-1	-1
7	8	-1	-1
8	3	10	-1
10	-1	-1	-1
-2	-1	-1	-1

Each **successorsList** ends with -1, while the basic blocks column ends with -2 (indicating the table’s end).

predecessorsTable

A 2D byte array storing immediate predecessors for each basic block in a function's CFG.

In an entry: the first byte indicates the block, followed by 4 bytes for its **predecessorsList**.



Basic Block	Predecessors				
0	-1	-1	-1	-1	-1
3	0	8	-1	-1	-1
5	3	-1	-1	-1	-1
7	3	-1	-1	-1	-1
8	5	7	-1	-1	-1
10	0	8	-1	-1	-1
-2	-1	-1	-1	-1	-1

Each **predecessorsList** ends with -1, while the basic blocks column ends with -2 (indicating the table's end).

Function Signatures

getControlFlowGraph

Description:

The main function called from **common.s** to retrieve information about a function's control flow graph.

Arguments:

a0: a pointer to the **instructionsArray**

a1: a pointer to the **leadersArray**

a2: a pointer to the **edgesList**

a3: a pointer to the **successorsTable**

a4: a pointer to the **predecessorsTable**

Returns:

None

getLeaders

Description:

Identifies the basic block leaders within a function's instructions.

Arguments:

a0: a pointer to the **instructionsArray**

a1: a pointer to the **leadersArray**

Returns:

None

1	0	0	1	0	1	0	1	1	0	1	-1

getEdges

Description:

Creates a list of edges between basic blocks within a function's control flow graph.

Arguments:

a0: a pointer to the **instructionsArray**

a1: a pointer to the **leadersArray**

a2: a pointer to the **edgesList**

Source	Target
0	3
0	10
3	5
3	7
5	8
7	8
8	3
8	10
-1	-1

Returns:

None

getSuccessors

Description:

Creates a list of immediate successors for each basic block within a function's control flow graph.

Arguments:

a0: a pointer to the **instructionsArray**

a1: a pointer to the **leadersArray**

a2: a pointer to the **successorsTable**

Returns:

None

Basic Block	Successors		
0	3	10	-1
3	5	7	-1
5	8	-1	-1
7	8	-1	-1
8	3	10	-1
10	-1	-1	-1
-2	-1	-1	-1

getPredecessors

Description:

Creates a list of immediate predecessors for each basic block within a function's control flow graph.

Arguments:

a0: a pointer to the **instructionsArray**

a1: a pointer to the **leadersArray**

a2: a pointer to the **predecessorsTable**

Returns:

None

Basic Block	Predecessors				
0	-1	-1	-1	-1	-1
3	0	8	-1	-1	-1
5	3	-1	-1	-1	-1
7	3	-1	-1	-1	-1
8	5	7	-1	-1	-1
10	0	8	-1	-1	-1
-2	-1	-1	-1	-1	-1

getBranchImm (helper)

Description:

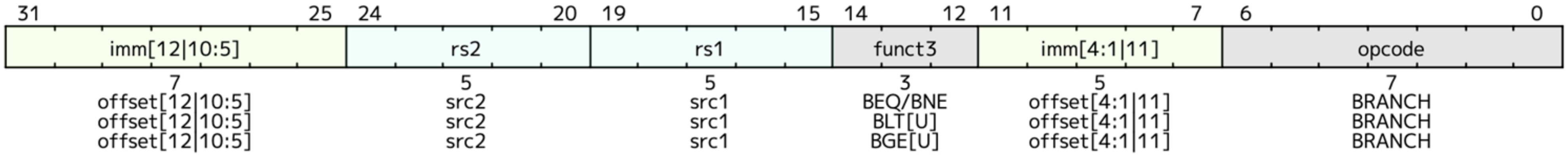
Takes a branch instruction (**SB-type**) and extracts the sign-extended immediate

Arguments:

a0: a **branch** instruction word

Returns:

a0: a sign extended immediate



getJalImm (optional helper)

Description:

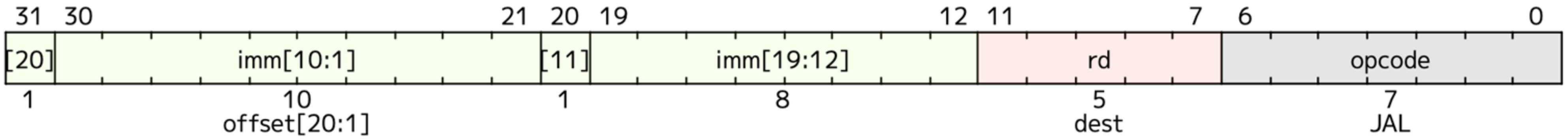
Takes a jump instruction (**UJ-type**) and extracts the sign-extended immediate

Arguments:

a0: a **jal** instruction word

Returns:

a0: a sign extended immediate



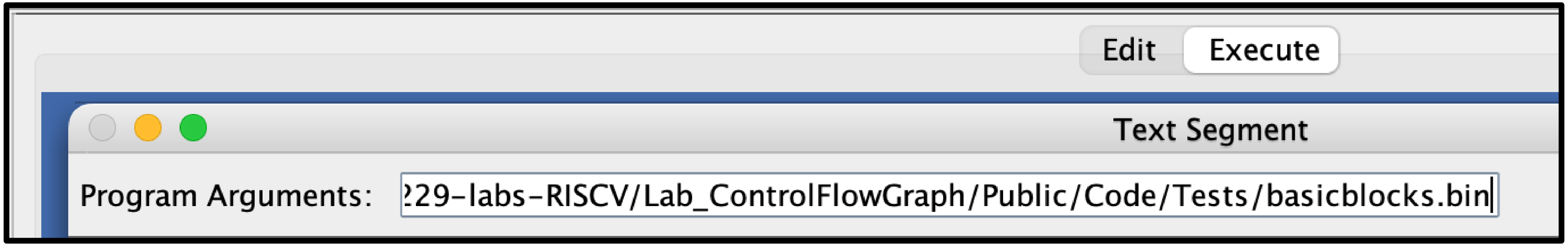
Testing

Program Arguments

We have provided some test inputs and expected outputs in the **Tests** folder.

One argument required for **controlFlowGraph.s**: the full path to the binary file of an assembled RISC-V function.

Ensure there are no quotation marks or spaces in the path.



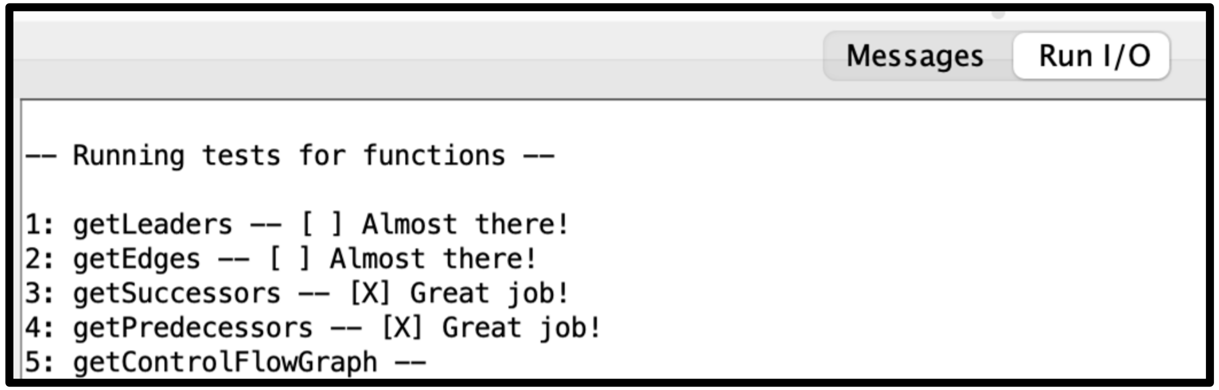
Unit Tests

The **common.s** file will run unit tests on the functions in **controlFlowGraph.s**.

Tests are hardcoded and do not use the file set as the program argument.

Check out the **.data** section in the **common.s** file to see how they are set up.

You can view the results in the “Run I/O” panel of RARS.



Parsed Control Flow Graph

The **common.s** file also parses the data structures populated by **controlFlowGraph.s**.

All structures must end with a sentinel value to be parsed properly.

You can view the results in the “Run I/O” panel of RARS following the unit tests.

Messages | Run I/O

```
Control Flow Graph Infromation:

6 basic block(s) found.

Note: Basic Block IDs are determined by the index of their leader in the instructionsArray.
For example, if a block's first instruction is the 4th instruction in the instructionsArray,
--------------------------------

Basic Block ID: 0

Instructions:
0x00000293
0x00000313
0x02A05063

Successors: 3, 10
Predecessors: None
--------------------------------

Basic Block ID: 3

Instructions:
0x0012F393
0x00039663

Successors: 5, 7
Predecessors: 0, 8
```

```
--------------------------------

Basic Block ID: 8

Instructions:
0x00128293
0xFEA2C4E3

Successors: 3, 10
Predecessors: 5, 7
--------------------------------

Basic Block ID: 10

Instructions:
0x00008067

Successors: None
Predecessors: 0, 8
--------------------------------

Edges List:

0 --> 3
0 --> 10
3 --> 5
3 --> 7
5 --> 8
7 --> 8
8 --> 3
8 --> 10
--------------------------------
```

Tests Folder

There are three tests: basicblocks, nestedloop, singleinstruction.

Each test has the following:

- A **.s** file containing the assembly code of the input function.
- A **.bin** (binary) file containing the assembled function (program argument).
- A **.out** file containing the correct output for the test.

Creating Tests

Unit tests and provided tests are not extensive.

Create your own tests to ensure your lab handles corner cases as well.

To create a test, do the following:

- Write the test function in assembly file (e.g., test.s).
- Create the binary file for your function i.e., execute the command "**rars a dump .text Binary test.bin test.s**".
- You can now execute **controlFlowGraph.s** with the program argument set to the full path of **test.bin**.

Disassembler

What is a Disassembler?

RARS is a RISC-V Assembler that translates RISC-V instructions to executable binary.

A Disassembler does the opposite.

```
0x000002b3
0x00000333
0x02a05063
0x0012f393
0x00039663
0x00530333
0x0080006f
0x00130313
0x00128293
0xfea2c4e3
0x00008067
```

```
mv      t0, zero
mv      t1, zero
blez    a0, 32
andi    t2, t0, 1
bnez    t2, 12
add     t1, t1, t0
j       8
addi    t1, t1, 1
addi    t0, t0, 1
bgt     a0, t0,-24
ret
```

For this lab, we will use an Open-Source RISC-V Disassembler
(https://github.com/michaeljclark/riscv-disassembler)

How to Use the Disassembler

There is a **Disassembly** folder in the **Code** folder.

There is a file called **“print-instructions.c”** in this folder that prints the equivalent instructions for hexadecimal words.

First, compile **“print-instructions.c”** i.e., execute the command **“gcc print-instructions.c”**

Next, create a text file containing hexadecimal instructions. The file should look like this:

example.txt

Lab_ControlFlowGraph > Public > Disassembly > example.txt

```
0x00100413
0x02850433
0xFFF50513
0x00050463
0xFF5FF06F
0x00800533
0x00008067
```

How to Use the Disassembler (cont.)

Once you have an executable for print-instructions.c (a.out) and a text file with hexadecimal instructions (example.txt), execute **“./a.out example.txt”**

Here is the disassembled instructions from **example.txt**:

```
 ./a.out example.txt
0000000000010000:  00100413         addi          s0,zero,1
0000000000010004:  02850433         mul           s0,a0,s0
0000000000010008:  fff50513         addi          a0,a0,-1
000000000001000c:  00050463         beqz          a0,8                              # 0x10014
0000000000010010:  ff5ff06f         j             -12                               # 0x10004
0000000000010014:  00800533         add           a0,zero,s0
0000000000010018:  00008067         ret
```

Notes on the Disassembler

The disassembler translates **addi t0, t0, 0** to **mv t0, zero**. Keep this in mind to avoid confusion with **addi** instructions.

The disassembler will not translate a sentinel value (0xFFFFFFFF) as expected.

What to Submit?

A single file, called **controlFlowGraph.s**.

Keep the file in the **Code** folder of the git repository.

Do not modify the name of any function.

Do not remove the CMPUT 229 Student Submission License.

Do not modify the line **.include "common.s"**.

Do not modify the **common.s** file.

Push your repository to GitHub before the deadline.

Good Luck!