CMPUT 229

Lab #6: Dead Code Elimination

What is Dead Code?

Dead Code refers to any line or block of code that is either *unreachable* (no execution path leads to that code) or it is *redundant* i.e. if it is executed the code has no visible effect on the output of the program.

Example of a Function with Dead Code

```
function_with_dead_code:
      addi    sp, sp, 4
      sw      s0, 0(sp)
      li      t0, 10              # t0 = 10
      add     t1, a0, t0          # t1 = a0 + t0
      sub     s0, a0, t0          # s0 = a0 - t0

      li      t2, 20              # t2 = 20 (dead, all uses of the value in t2 are dead)
      addi    t3, t2, 5           # t3 = t2 + 5 (dead, value in t3 never used)

      li      t4, 0               # t4 (loop counter)
      li      t5, 5               # t5 (loop end trip count)

      loop_start:
            bge     t4, t5, loop_end

            slli    t6, t4, 1           # t6 = t4 * 2 (dead, value in t6 never used)

            add     t1, t1, t4          # t1 += t4
            sub     s0, s0, t4          # s0 -= t4
            addi    t4, t4, 1           # t4 += 1
            j       loop_start

      loop_end:
            blez    a1, else_branch
            add     a0, t1, a2          # a0 = t1 + a2
            j       end_if

      else_branch
            sub     a0, t1, a2          # a0 = t1 - a2
            addi    zero, t1, 1         # (dead, defines the zero register)

      end_if:
            mul     s3, t0, t2          # t3 = t0 * t2 (dead, value in s3 never used)
            addi    a1, s0, 4           # a1 = s0 + 4 (dead, value in a1 never used)

            add     a0, a0, t0          # a0 += t0 (return)
            mv      a1, s0              # a1 = s0 (return)

            lw      s0, 0(sp)
            addi    sp, sp, 4
            ret
```

What is Dead Code?

A Closer Look

Dead Code refers to any line or block of code that is either *unreachable* (no execution path leads to that code) or it is *redundant* i.e. if it is executed the code has no visible effect on the output of the program.

Let's study an example from the sample function

Instruction: li t2, 20

- **Dead** because it only feeds into a subsequent dead instruction.

Instruction: addi t3, t2, 5

- **Dead** as its result (t3) is never used later in the program.

```
function_with_dead_code:
        addi    sp, sp, 4
        sw      s0, 0(sp)
        li      t0, 10              # t0 = 10
        add     t1, a0, t0          # t1 = a0 + t0
        sub     s0, a0, t0          # s0 = a0 - t0

        li      t2, 20              # t2 = 20 (dead, all uses of the value in t2 are dead)
        addi    t3, t2, 5           # t3 = t2 + 5 (dead, value in t3 never used)

        li      t4, 0               # t4 (loop counter)
        li      t5, 5               # t5 (loop end trip count)
```

What is Dead Code?
A Closer Look

Dead Code refers to any line or block of code that is either *unreachable* (no execution path leads to that code) or it is *redundant* i.e. if it is executed the code has no visible effect on the output of the program.

Example:

Instruction: addi zero t1, 1

- **Dead** because it is trying to set the zero register which is immutable.

Instruction: mul s3, t0, t2

- **Dead** as its result (s3) is never used

Instruction: addi a1, s0, 4

- **Dead** because the result (a1) Is discarded immediately

```
else_branch
      sub     a0, t1, a2          # a0 = t1 - a2
      addi    zero, t1, 1         # (dead, defines the zero register)

end_if:
      mul     s3, t0, t2          # t3 = t0 * t2 (dead, value in s3 never used)
      addi    a1, s0, 4           # a1 = s0 + 4 (dead, value in a1 never used)

      add     a0, a0, t0          # a0 += t0 (return)
      mv      a1, s0              # a1 = s0 (return)

      lw      s0, 0(sp)
      addi    sp, sp, 4
      ret
```

Liveness Analysis

A variable is said to be *live* at a particular point in a program if its current value might be used later meaning it hasn’t been overwritten or discarded yet.

Liveness Analysis

Two Levels of Liveness Analysis

- **Block-Level Analysis**
  - ► Identifies live variables at the **entry and exit** of each basic block.

- **Instruction-Level Analysis**
  - ► Tracks liveness at the **granularity of individual instructions**.
  - ► Especially useful for catching **transitively dead code** that block-level analysis may overlook.

Block-Level Liveness Analysis

Block-level liveness analysis uses a **fixed-point algorithm** to figure out which variables are still needed at the end of each basic block in a program's control flow.

The algorithm iteratively computes the sets of variables that are live at the entry and exit of each basic block in the CFG. It continues until a fixed point is reached, where further iterations do not change the sets of live variables.

Pseudo Code for Fixed-Point Iteration (Block-Level Liveness Analysis)

```
liveness_analysis:

for each block in cfg.blocks:
block.live_in = empty_set
block.live_out = empty_set

intialize worklist

worklist.add(exit_block)

while worklist not empty:
block = worklist.pop()

new_live_out = empty_set

if block == exit_block:
    new_live_out = function_live_out

for each succ in block.successors:
    new_live_out = new_live_out ∪ succ.live_in

new_live_in = block.gen ∪ (new_live_out - block.kill)

if new_live_out != block.live_out or new_live_in != block.live_in:
block.live_out = new_live_out
block.live_in = new_live_in

for each pred in block.predecessors
if pred not in worklist:
worklist = worklist.add(pred)
```

Initially, all sets are empty.

Then:

- Start by adding the exit block to the worklist.
- For each block, calculate its new `live out` set by combining the `live in` sets of its successors.
- Compute the new `live in` set using the formula: `live_in = gen ∪ (live_out - kill)`
- If either set changes, update them and re-add the block's predecessors to the worklist.

This continues until no further changes occur in any set, indicating fixed-point convergence.

Instruction-Level Liveness Analysis

After reaching fixed-point convergence at the block level, we can perform a finer, **instruction-level** liveness analysis to catch more precise instances of dead code.

Block-level analysis may miss definitions that are never used before being redefined within the same block, these are effectively dead but still appear live at the block level.

Instruction-level analysis resolves this by examining each instruction in reverse (from last to first) within a block. This reverse pass catches redefinitions that hide unused values, allowing us to more accurately identify dead code.

Pseudo Code for Dead Code Identification (Instruction-Level Liveness Analysis)

```
find_dead_code:

for each block in cfg.blocks:
current_live_out = block.live_out

for each instruction in block (iterate from end to start):

if instruction is already marked as dead:
        continue to next instruction

    new_live_in = instruction.gen ∪ (current_live_out - kill)

    if instruction.kill:
if instruction.kill not in current_live_out or zero_register in instruction.kill:
    mark instruction as dead

current_live_out = new_live_in
```

The algorithm initializes `current_live_out` with the block's `live_out` set.

It then walks through each instruction in reverse. For each instruction:

- If it's already marked dead, skip it.
- Compute `live_in = gen ∪ (live_out - kill)`.
- If it defines a register that isn't live out (and isn't `zero`), mark it as dead.
- Update `current_live_out` to reflect the new liveness state.

Lab #6: Dead Code Elimination

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.

genArray

- Imagine some RISC-V Assembly function with basic blocks B0, B3, B10, B12. Then consider the genArray for foo, an array of words, where each word-entry corresponds to a block in the CFG for foo and each such word is a bitvector which represents the gen-set for that block.
- Every bit that is set (i.e. the bits that are 1) represents the registers that are generated or defined within a block before any of them are killed by another instruction in the same block.
- Example: if bit 23 is set in the third element of the genArray, that means register x23 is part of the genset for the 3rd block (B10 in the picture)

killArray

- An array of words representing the kill sets for each basic block in the input function's CFG.
- Each word is a bit vector where each bit corresponds to a register that is killed (overwritten or redefined) in that block.
- Ends with a sentinel value of -1 (0xFFFFFFFF).

liveInArray

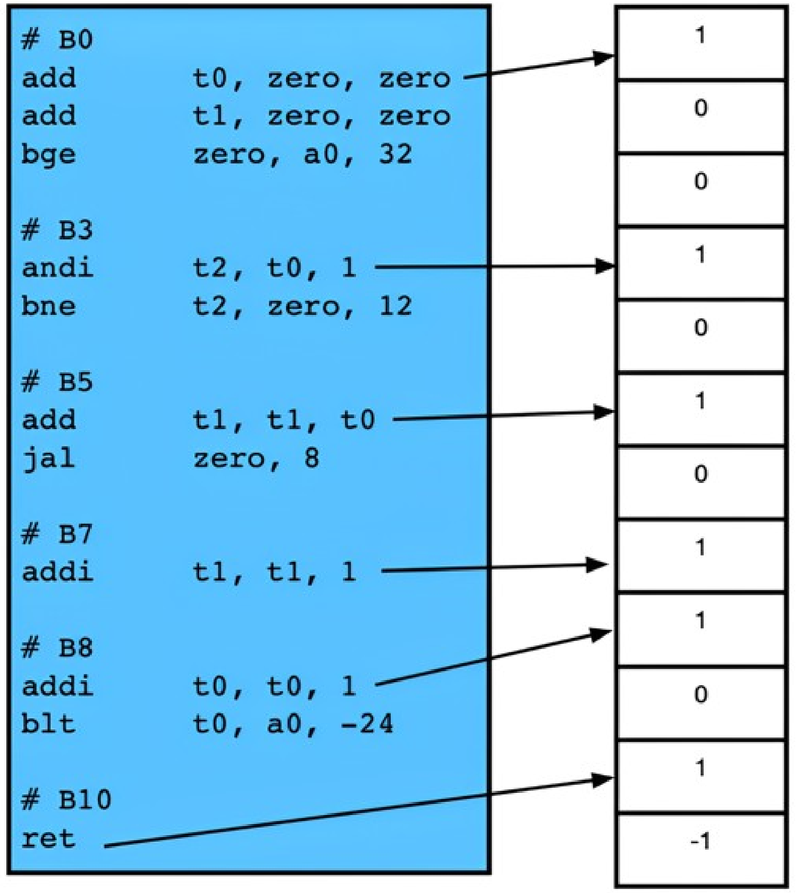
- An array of words representing the live-in sets for each basic block in the input function's CFG.
- Each word is a bit vector where each bit corresponds to a register that is in the live-in of a given block.
- Ends with a sentinel value of -1 (0xFFFFFFFF).

liveOutArray

- An array of words representing the live-out sets for each basic block in the input function's CFG.
- Each word is a bit vector where each bit corresponds to a register that is in the live-out of a given block.
- Ends with a sentinel value of -1 (0xFFFFFFFF).

deadCodeArray

- An array of bytes representing the dead code status of each instruction in the input function.
- Each byte corresponds directly to an instruction.
- If the instruction is identified as dead code, its corresponding byte will be 1; if it is not dead code, the byte will be 0.



Ends with a sentinel value of -1.

Example may not be representative of an actual function with dead code, only for illustration

workList

worklist:

The workList is a circular queue used to manage a list of items to be processed. It operates with a fixed size and maintains two indices:

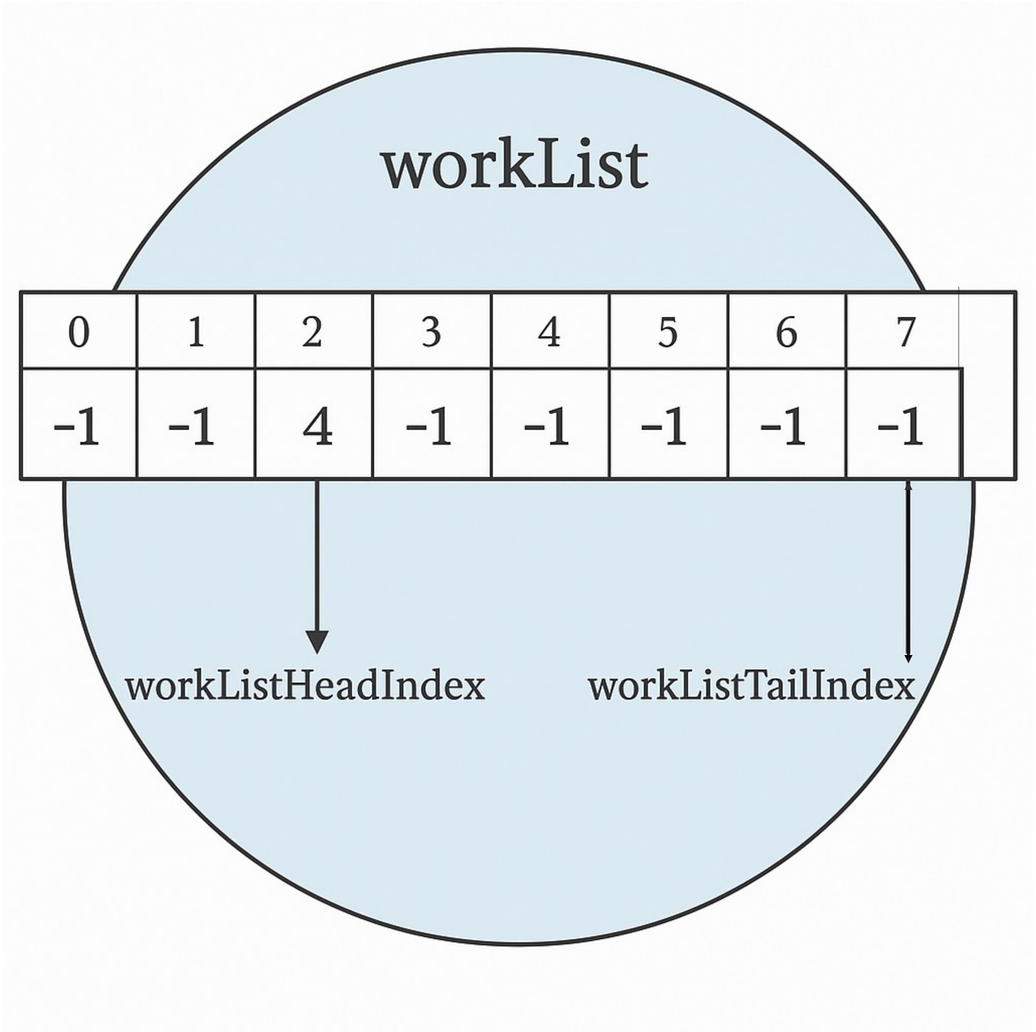
- workListHeadIndex: Points to the position in the list where the next item will be removed.
- workListTailIndex: Points to the position in the list where the next item will be added.

The workList uses an array where:

- An entry of -1 (0xFF) indicates an empty slot.
- When the list is full, adding a new item will result in an error if there is no space.
- When the list is empty, removing an item will return an error if there are no items to remove.

The workList is implemented as a circular queue, meaning that when the tail index reaches the end of the array, it wraps around to the beginning, and similarly for the head index.

workList


workList
0
1
2
3
4
5
6
7
-1
-1
4
-1
-1
-1
-1
-1
workListHeadIndex
workListTailIndex

inWorkListArray

- An array of bytes corresponding to each block in the input function's CFG.
- Each byte represents whether the block is currently in the worklist.
- If the block is in the worklist, the byte will be set to 1; otherwise, it will be set to 0.
- Ends with a sentinel value of -1 (0xFF).

Lab #6: Dead Code Elimination

Function Signatures

deadCodeElimination

Performs an iterative analysis to remove dead code from the input. This is the main entry point of the solution. Follow the recommended flow given in the lab description.

Arguments:
- a0: Pointer to the instructionsArray.
- a1: Pointer to the genArray.
- a2: Pointer to the killArray.
- a3: Pointer to the liveInArray.
- a4: Pointer to the liveOutArray.
- a5: Input function's live-out set.
- a6: Pointer to the deadCodeArray.
- a7: Pointer to the refinedInstructionsArray.

Returns:
- None.

getGenKillSets

Processes each basic block to compute GEN and KILL sets for later use.
GEN sets mark registers read before defined in the block.
KILL sets mark registers defined within the block.

Instructions marked dead are skipped.
For each instruction in a block:
- Used registers that have not already been killed are added to GEN.
- Defined registers are added to KILL.

The computed GEN and KILL sets for each block are stored in the provided arrays.

Arguments:
a0: Pointer to the instructionsArray.
a1: Pointer to the deadCodeArray.
a2: Pointer to the genArray.
a3: Pointer to the killArray.

Returns:
None.

getLiveSets

Iteratively computes live-in and live-out sets for each basic block
until they reach a fixed point. Refer to the lab description for details on the algorithm.
The live-in and live-out sets are stored back in the provided arrays.

Arguments:
a0: Pointer to the genArray.
a1: Pointer to the killArray.
a2: Pointer to the liveInArray.
a3: Pointer to the liveOutArray.
a4: Function's live-out set.

Returns:
None.

markDeadCode

Performs dead code identification analysis by iterating backwards through each basic block's instructions.
For each instruction (starting from the block's last instruction), marks each instruction as 0 or 1 (alive or dead) and returns 1 if dead code marked, 0 otherwise.

Arguments:
a0: Pointer to the instructionsArray.
a1: Pointer to the liveOutArray.
a2: Pointer to the deadCodeArray.

Returns:
a0: Dead code status (1 if dead code found, 0 otherwise).

fixTargets

Description:
Adjusts branch and jump instruction targets based on the presence of dead code.

Arguments:
a0: Pointer to the instructionsArray.
a1: Pointer to the deadCodeArray.

Returns:
None.

removeDeadCode

Description:

Removes dead code from the input function by filling the refinedInstructionsArray with only the instructions marked as not dead in the deadCodeArray.

Arguments:

a0: Pointer to the instructionsArray.
a1: Pointer to the deadCodeArray.
a2: Pointer to the refinedInstructionsArray.

Returns:

None.

decodeInstruction

Description:
Decodes a given instruction to determine its defined and used registers and returns them as bit vectors.

Arguments:
a0: An instruction word.

Returns:
a0: Bit vector of the defined register.
a1: Bit vector of the used registers.

For this lab, only the following opcodes are relevant:
I-type (load): 0000011
I-type (arithmetic with immediate): 0010011
U-type (load upper immediate): 0110111
S-type (store): 0100011
SB-type (branch): 1100011
R-type (arithmetic): 0110011
Jump instructions are not considered to define or use any registers in this lab.

workList Functions

initliazeWorkList:
Initializes the head and tail indices of the workList to zero.
Sets up the workList to be empty and ready for new entries.

Arguments:
None.

Returns:
None.

popFromWorkList:
Pops an item from the workList.

Arguments:
None.

Returns:
a0: Item from workList or -1 (0xFF) if workList is empty.

addToWorkList:
Adds an item to the workList. Returns an error if the list is full.

Arguments:
a0: Item to add to workList.

Returns:
a0: 0 on success, -1 (0xFF) if workList is full.

adjustBranchImm (optional helper)

Description:

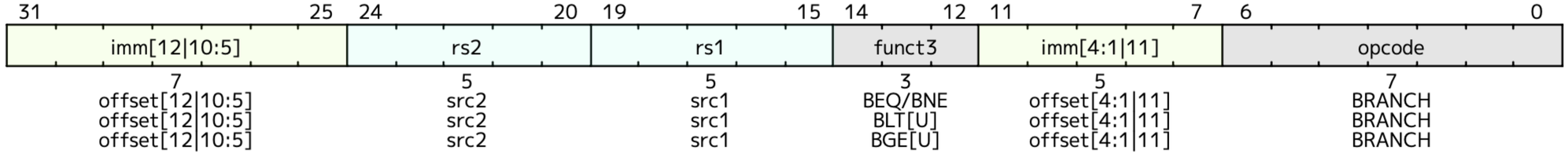
Takes a branch instruction (**SB-type**) and extracts the sign-extended immediate

Arguments:

a0: A branch instruction.
a1: New immediate value.

Returns:

a0: Branch instruction with updated immediate.



adjustJalImm (optional helper)

Description:

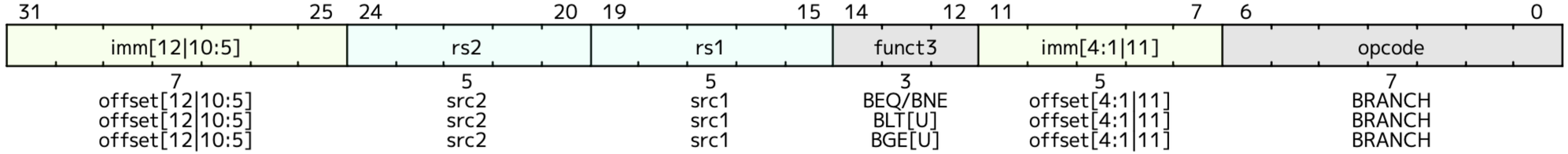
Updates the immediate value of a jump instruction (jal) (UJ type).

Arguments:

a0: A jal instruction.
a1: New immediate value.

Returns:

a0: Jump instruction with updated immediate.



Flow of Implementation

Recommended Program Flow

1. Call getControlFlowGraph to populate the CFG data structures.

2. Initialize the deadCodeArray with all instructions set as unmarked.

3. Perform the iterative analysis as follows:

- Call genKillSets to compute the generation and killing sets.
- Call getLiveSets to determine the live-in and live-out sets for each basic block.
- Call markDeadCode to identify and mark the dead code based on the live sets.
- If markDeadCode indicates that new dead code was found, return to step 3.

4. Once the loop exits (when no new dead code is found), call fixTargets to adjust any targets affected by the dead code removal.

5. Finally, call removeDeadCode to eliminate the marked dead code from the program.

Lab #6: Dead Code Elimination

Testing

CFG_simulator tool

In the Code/test_case_validator folder you will find an executable called cfg_verifier.

To create a test case, write a RISC-V function (e.g., INPUT_FUNCTION.s) in an assembly file to begin with. Then, type in the following command in the terminal: rars a dump .text Binary <INPUT_FUNCTION.bin> <INPUT_FUNCTION.s>.

The binary file generated by this command can serve as a program argument to the CFG_simulator tool included with this lab. The cfg_verifier tool will parse the file and output another binary file (the cfg.bin) which can be used as an argument to the deadCodeElimination.s file.

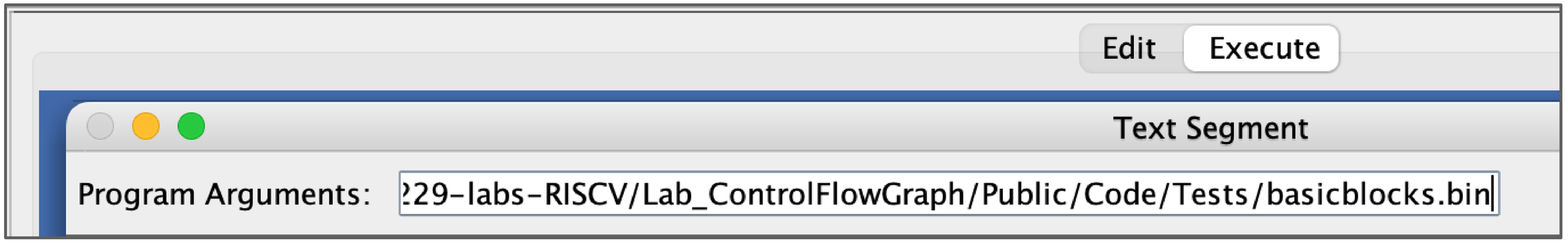
Run as ./cfg_verifier <INPUT_FUNCTION.bin>

Program Arguments

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

Two arguments required for deadCodeElimination.s: the full path to the binary file cfg.bin (output from the cfg_verifier tool) and the final live-out array of the function.

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



Tests Folder

There are three tests: basicblocks, nestedloop, singleinstruction.

Each test has the following:

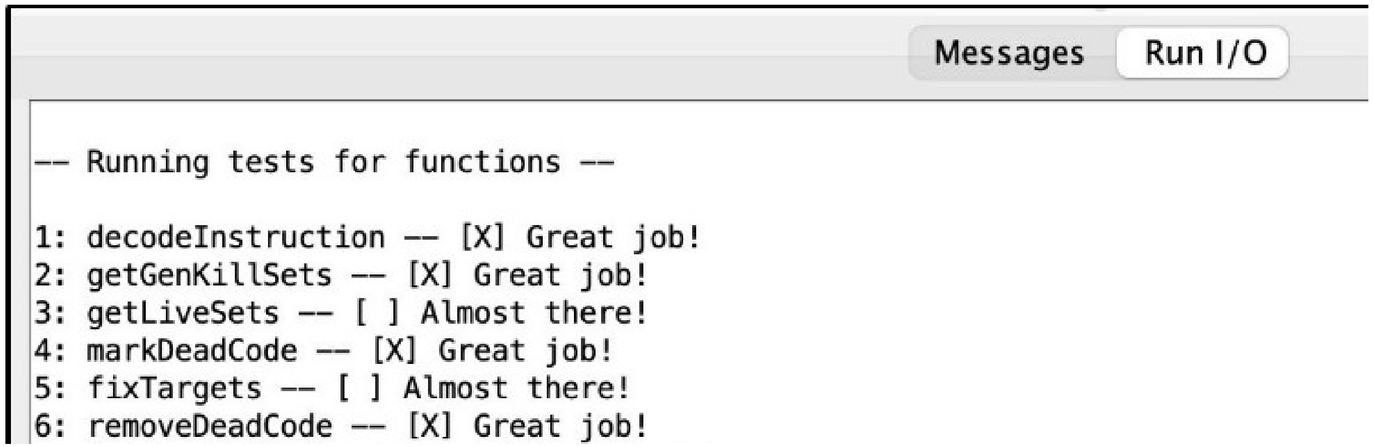
- A **.s** file containing the assembly code of the input function.
- A **.bin** (binary) file containing the assembled function (program argument).
- A **.txt** file containing the correct output for the test.

Unit Tests

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



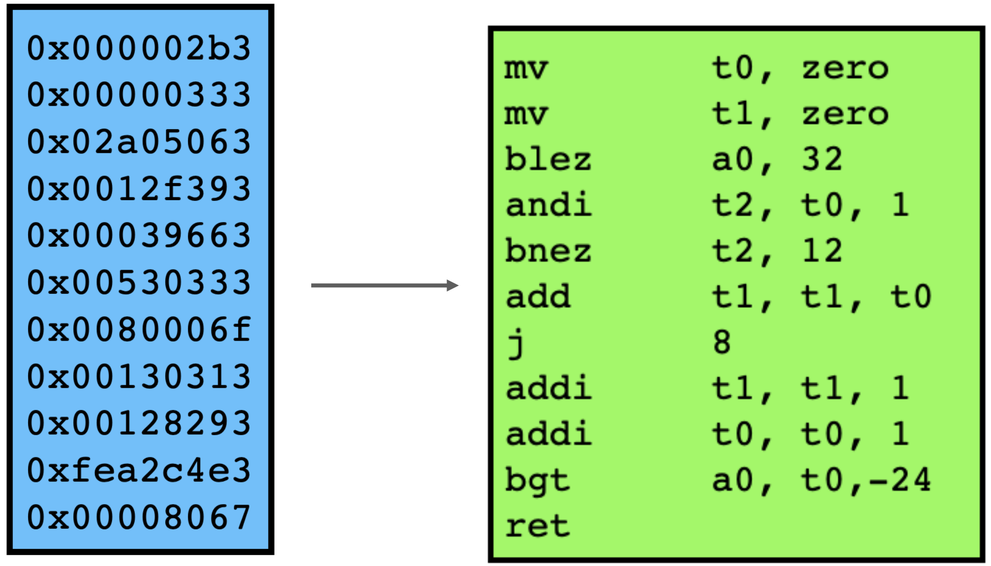
Lab #6: Dead Code Elimination

Disassembler

What is a Disassembler?

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

A Disassembler does the opposite.



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

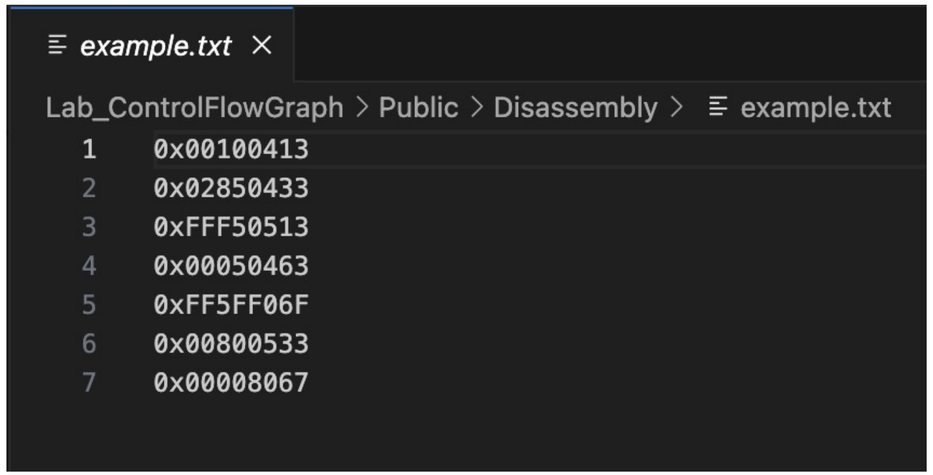
How to Use the Disassembler

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”** by running e.g. **“gcc print-instructions.c”**

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



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.

Lab #6: Dead Code Elimination

Good Luck!