

# Lab Find Live

CMPUT 229

# Control Flow Graphs

# Control Flow Graph Example

Loop:

```
lw    $t0 0($t7)
beq   $t0 $zero Done
```

```
addi  $t0 $t0 -1
addi  $t7 $t7 4
j     Loop
```

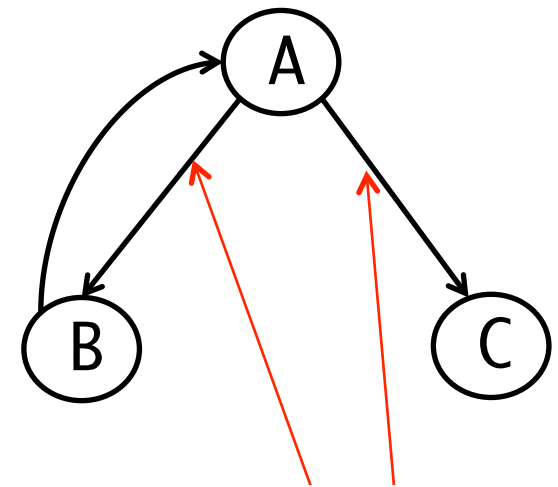
Done:

```
addi  $v0 $zero 1
syscall
jr    $ra
```

A

B

C



Do not know  
the value of `$t0`

Do not know  
the path

Static Analysis

# How to find control flow instructions?

Instruction	Binary
bgez \$s, offset	0000 01ss sss0 0001 iiiii iiiii iiiii iiiii
bltz \$s, offset	0000 01ss sss0 0000 iiiii iiiii iiiii iiiii
beq \$s, \$t, offset	0001 00ss ssst tttt iiiii iiiii iiiii iiiii
bne \$s, \$t, offset	0001 01ss ssst tttt iiiii iiiii iiiii iiiii
blez \$s, offset	0001 10ss sss0 0000 iiiii iiiii iiiii iiiii
bgtz \$s, offset	0001 11ss sss0 0000 iiiii iiiii iiiii iiiii
j target	0000 10aa aaaa aaaa aaaa aaaa aaaa aaaa
jal target	0000 11aa aaaa aaaa aaaa aaaa aaaa aaaa
jr \$s	0000 00ss sss0 0000 0000 0000 0000 1000

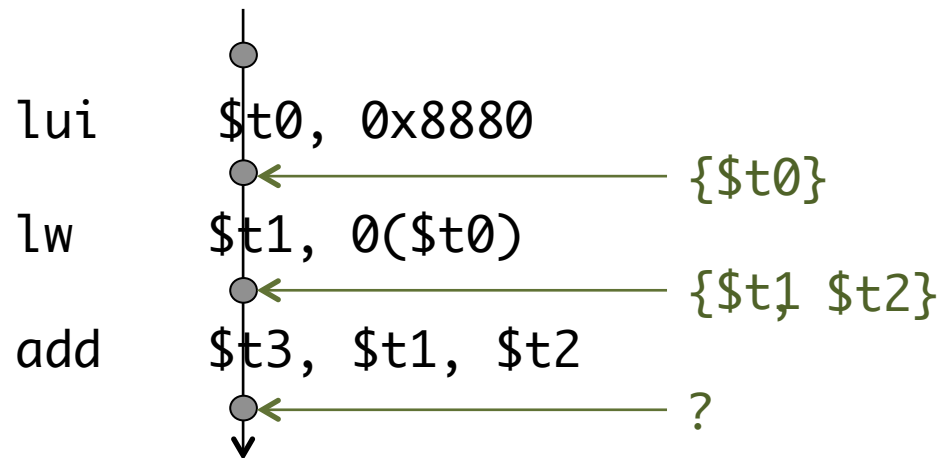
Check the opcodes. To find jr \$ra check both the opcode and the register value in the appropriate bitfield.

Liveness

# Liveness Definition

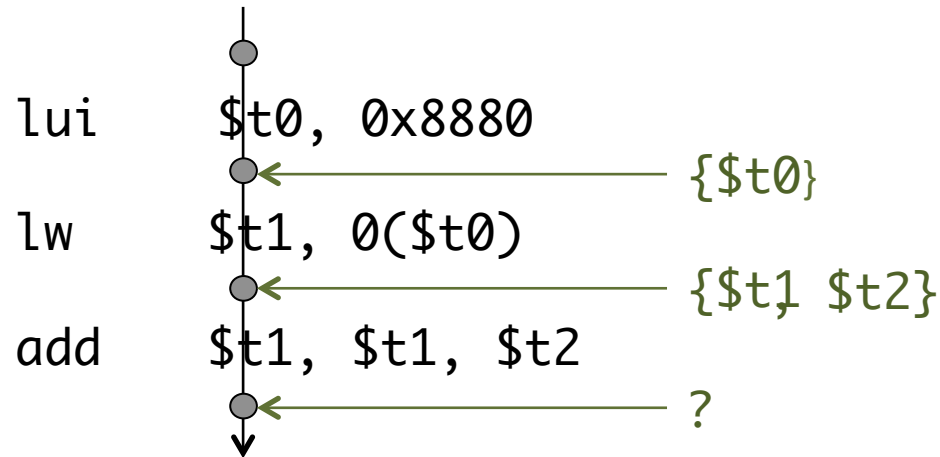
A register contains a value that is live if that value may be used by another instruction.

Example:

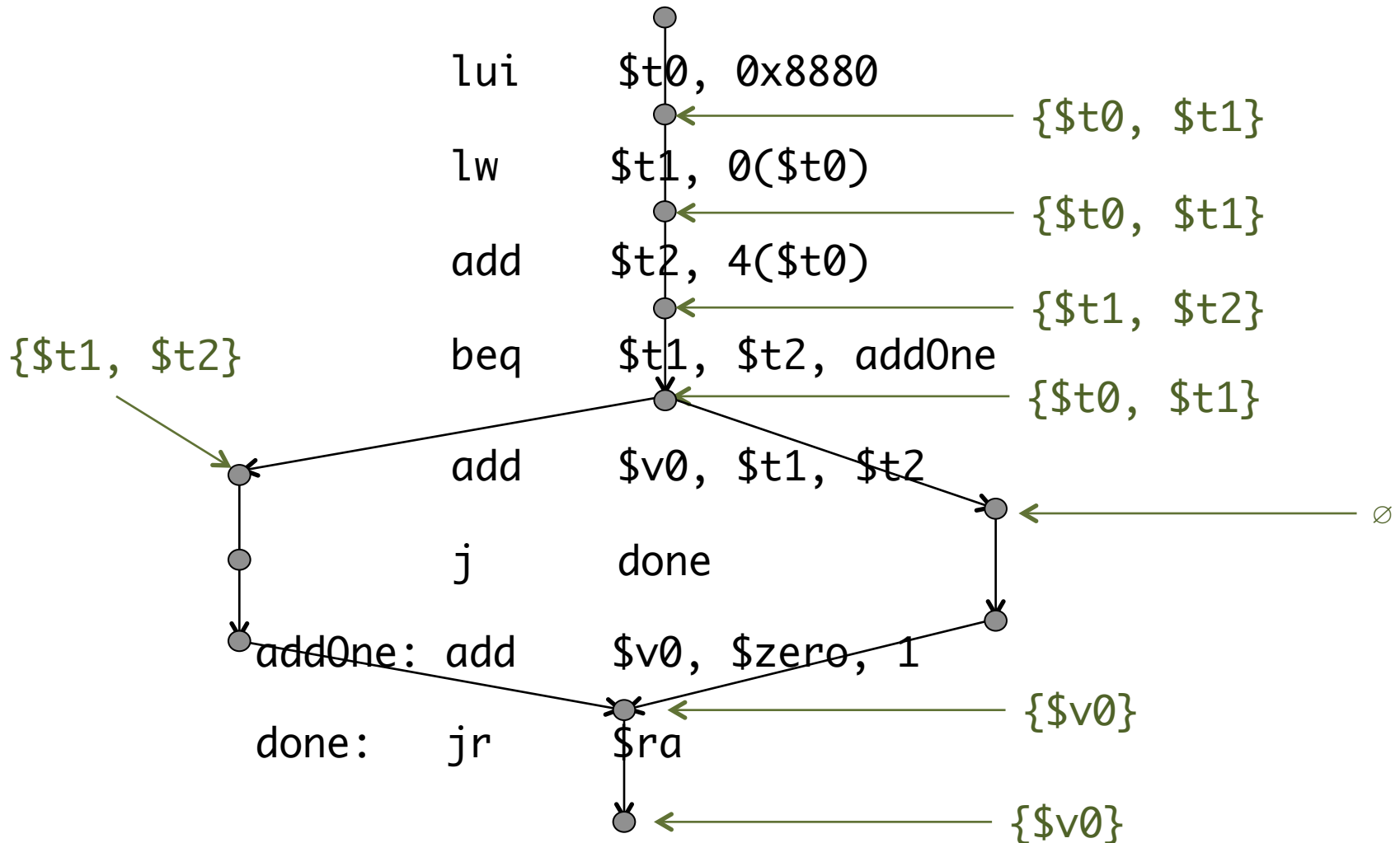


# Liveness Examples

Another Example:



# Liveness Examples (Control Flow)





# Liveness

\$t0, \$t1 and \$t2 are used as source/target registers after the start point

\$t3 is assigned a value, even if we found it as a source later, it is 'dead' at the instruction immediately following the function call

```
.text
main:

    addi    St0 $zero 1
    addi    St1 $zero 0
    addi    St2 $zero 10
    addi    St3 $zero 7

    jal     Function

loop:
    beq     St1 St2 done
    add     St1 St1 St0
    j       loop

done:
    addi    St3 $zero 1
    jr      Sra

Function:

    addi    Ss0 $zero 1
    addi    Ss1 $zero 2

    jr      Sra
```

# Liveness

\$t1 is both the source and the destination in the add instruction.

We can think about the order of operations that the instruction breaks down into on a lower level and note that the register is used before it has a value assigned to it.

Therefore we can think of it as “updated” not “overwritten” and see that it is live.

```
.text
main:

    addi    St0 $zero 1
    addi    St1 $zero 0
    addi    St2 $zero 10
    addi    St3 $zero 7

    jal     Function

loop:
    beq     St1 St2 done
    add     St1 St1 St0
    j       loop

done:
    addi    St3 $zero 1
    jr      Sra

Function:

    addi    Ss0 $zero 1
    addi    Ss1 $zero 2

    jr      Sra
```

# Instruction Parsing

In all branch instructions values are used from registers for comparison, so even though they are an I-type instruction they must be parsed differently than other I-type instructions.

beq \$t0 \$t7 someLabel -> \$t0 == \$t7 ?

vs.

addi \$t0 \$t7 100 -> \$t0 = \$t7 + 100

Some branch instructions use two registers and some only use one. While extracting registers to find the live/dead registers you must handle the different branch instructions appropriately.

Instruction	Binary
bgez \$s, offset	0000 01ss sss0 0001 iiii iiii iiii iiii
bltz \$s, offset	0000 01ss sss0 0000 iiii iiii iiii iiii
beq \$s, \$t, offset	0001 00ss ssst tttt iiii iiii iiii iiii
bne \$s, \$t, offset	0001 01ss ssst tttt iiii iiii iiii iiii
blez \$s, offset	0001 10ss sss0 0000 iiii iiii iiii iiii
bgtz \$s, offset	0001 11ss sss0 0000 iiii iiii iiii iiii
j target	0000 10aa aaaa aaaa aaaa aaaa aaaa aaaa
jal target	0000 11aa aaaa aaaa aaaa aaaa aaaa aaaa
jr \$s	0000 00ss sss0 0000 0000 0000 0000 1000

Also in all types of save instructions (sw, sh, sb) both registers are a source (rt is not a destination as it is in other I type instructions), so they also must be parsed differently than other I-type instructions.

`sw $t0 4($t2)` -> store value in \$t0 at  
(address in \$t2)+4

sw, sh and sb instructions can be detected by their opcode.

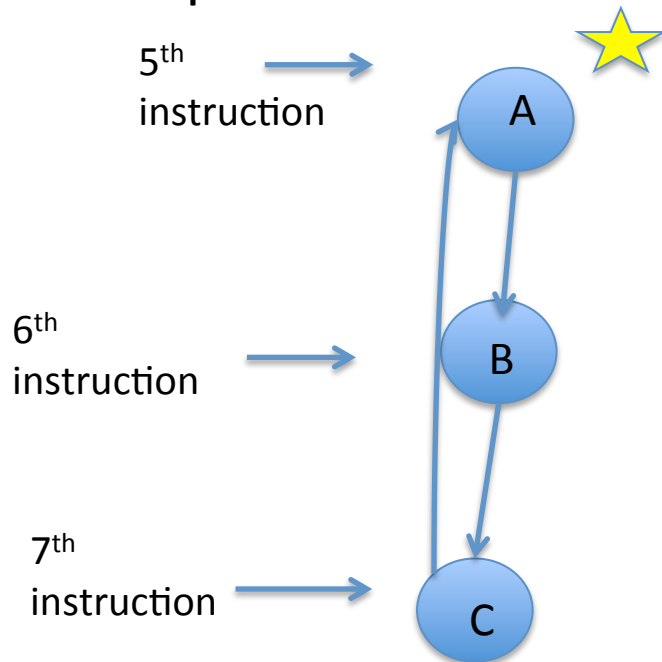
Address	Instruction
sw \$t, offset(\$s)	1010 11ss ssst tttt oooo oooo oooo oooo
sh \$t, offset(\$s)	1010 01ss ssst tttt oooo oooo oooo oooo
sb \$t, offset(\$s)	1010 00ss ssst tttt oooo oooo oooo oooo

# Algorithm Breakdown

# Tracing Loops

While tracing the execution of the program we need to avoid getting stuck in loops.

One way to do this is to indicate to ourselves where we have already been. This is what you can use the visited array for. You may index into this array based on the sequential position of the particular instruction in the program.



(all initialized to 0)

Indices = 0-3, 4, 5, 6, ect.

Visited = [ ... , \_ , \_ , \_ , ... ]

1

1

1

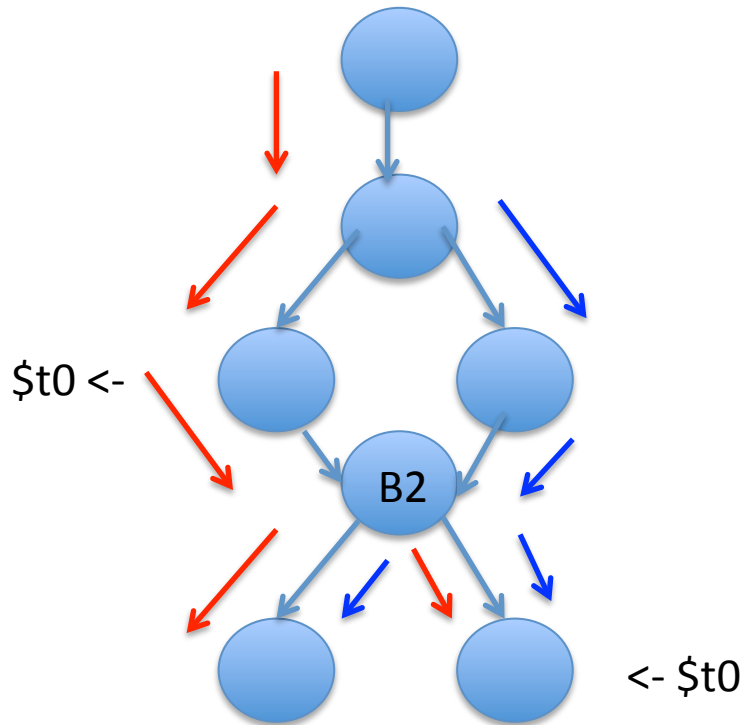
First instruction in the program is index 0 in visited.



# Loop exception

While you do not want to get stuck in a loop, you may actually want to visit the same instruction more than once, in a different path trace. This means that after you are done processing one path pass over that instruction, you will need to clear that bit that you set earlier in visited.

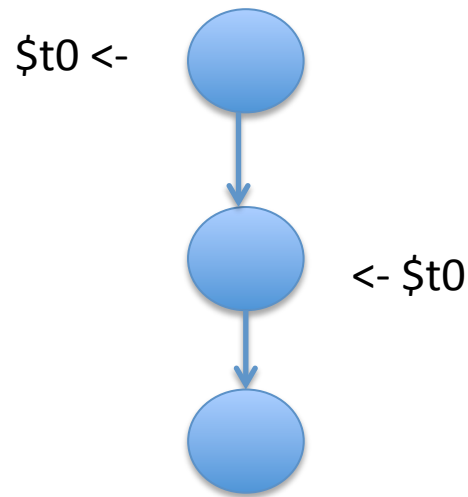
The following illustrates why:



\$t0 is live but if the blue path stopped at B2 because we had already visited B2 while on the red path then we would miss this important fact.

# Dead Registers

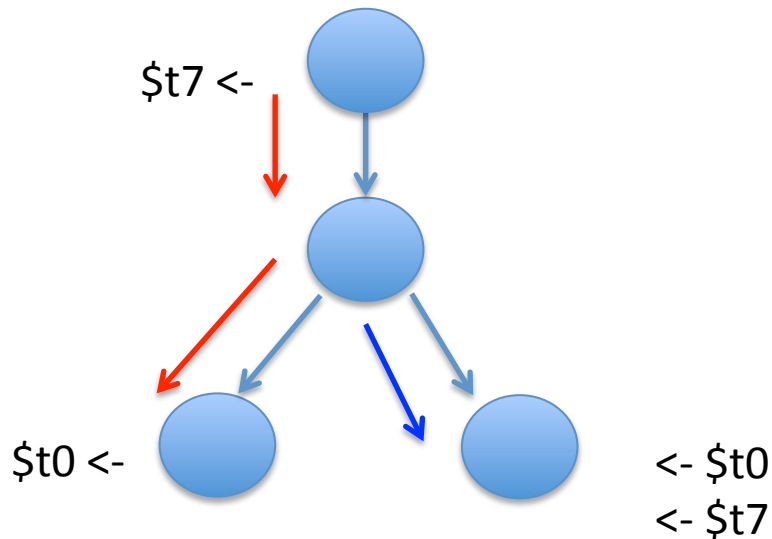
In the trivial case a register is dead if it is overwritten before it is used. If we see the register as a source after it is overwritten we don't care – it is still dead. We need some way to indicate this. You may use the `knownToBeDead` word to store this information.



`$t0` is dead in this example.

# Dead Registers continued

In the nontrivial case a register is dead on one path but live on another. Thus we cannot simply use the same `knownToBeDead` word for every path. We must save and restore the word appropriately so that we do not carry over information between two paths, but also do not lose relevant information from the path's past.



`$t7` is clearly dead. When we go down the blue path we want to keep this information, however we don't want the discovery of `$t0` as dead to carry over from the red path to the blue path.

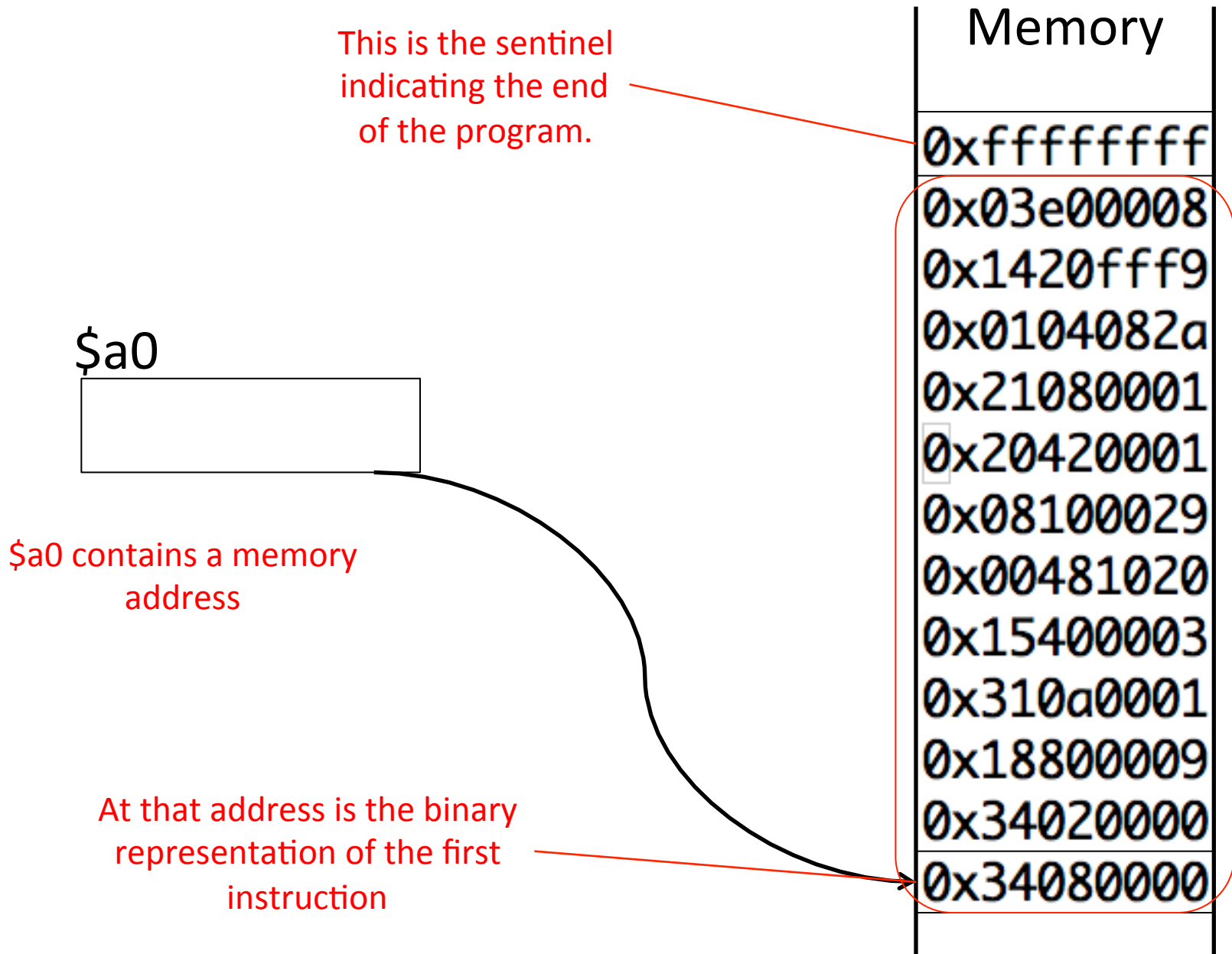
# Live vs Dead

You are encouraged to make control flow graphs on your own to explore the different types of scenarios that can arise when registers are dead on some paths and live on others. Try to include different combinations of branches and jumps to cover all of the different cases that you must handle.

# Tips

- If you choose to use the globals that are suggested in the algorithm keep in mind that the contents of these will not apply from one start point to the next.
- How you parse the registers from the instructions is completely up to you, as long as what you return fits the specified formats.
- Keep in mind you are gathering the live registers on paths each starting from the instruction immediately after a function call

# The Assignment ( findLive input)



# The Assignment (findLive output)

\$v0: An address of a list that contains live registers at each start point.

This list must contain the sentinel 0xFFFFFFFF as the last element.



# The Assignment (findLive output)

Expected \$v0 format:

Each element in the array that is specified by \$v0 should (possibly) have bits set in the positions of their value:

Example:

at the first start point, \$s0 and \$t8 are live:

Bit	31-26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7-0
Encodes For	NA	\$t9	\$t8	\$s7	\$s6	\$s5	\$s4	\$s3	\$s2	\$s1	\$s0	\$t7	\$t6	\$t5	\$t4	\$t3	\$t2	\$t1	\$t0	NA
																				
			*								*									

\* == Only these bits are set.



# Testing

- Test Cases
  - One test case is provided, under the link in the assignment specification
- Student-Generated Test Cases
  - Students will submit test cases
- Printing the Output of your solution
  - MIPS code provided for printing

# University of Alberta

## Code of Student Behavior

<http://www.governance.ualberta.ca/en/CodesofConductandResidenceCommunityStandards/CodeofStudentBehaviour.aspx>

### **30.3.2(1) Plagiarism**

No Student shall submit the words, ideas, images or data of another person as the Student's own in any academic writing, essay, thesis, project, assignment, presentation or poster in a course or program of study.

### **30.3.2(2) Cheating**

30.3.2(2) d No Student shall submit in any course or program of study, without the written approval of the course Instructor, all or a substantial portion of any academic writing, essay, thesis, research report, project, assignment, presentation or poster for which credit has previously been obtained by the Student or which has been or is being submitted by the Student in another course or program of study in the University or elsewhere