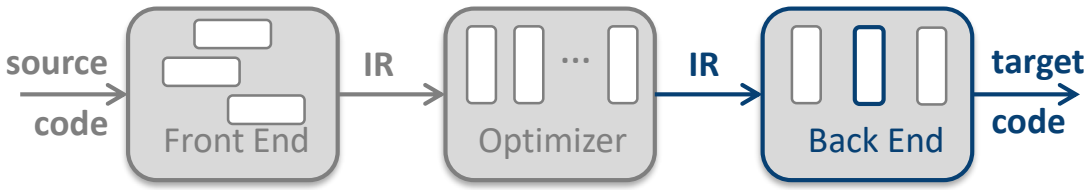
RICE

COMP 412
FALL 2018

Lab 3: Improving the Quality of Schedules

Comp 412



Material from the Performance Tutorial

Lab 3 Schedule

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
Oct 21	22	23	**24**	25	26 **Docs Available** **Lab Lecture**	27 Start work
28	29	30	31	1 **You are here**	2 Tutorial 4PM	3 Complete & Correct Dependence Graph
4	5	6	7	8	9	10 Correct Schedule
11	12 **The goal is here**	13	14	15	16	17 Improve Schedules
18	19	20 **Lab 3 Due Date**	21 **No Class (Walk)** Late day	22 Not a late day	23 **Thanksgiving Break** Not a late day	24 Not a late day

Performance: What does it mean?

In Lab 3, 50% of the code grade is based on the speed of the code that your scheduler produces

- We will measure performance using the **Lab 3 ILOC Simulator** [1]
  - Take the original code & run it
  - Take the scheduled code & run it
  - Make sure that the answers are identical
  - Difference between the two runs is the improvement due to scheduling
- The goal of Lab 3 is to maximize the improvement due to scheduling
  - The 50% credit for performance is based on how your scheduler does relative to the schedulers of other students (& relative to the reference implementation — which can be beaten)
  - Attention to detail can improve your scheduler
  - **But**, the problem is NP-Complete, so you shouldn't expect to win every time

[1] The Lab 3 simulator differs from the Lab 2 simulator. Be sure to use the correct one, and to measure the code's speed with the correct interlock setting, "**-s 1**"

What Affects Schedule Quality?

What goes into creating a good schedule?

- A good priority function
  - Priorities determine which operation is chosen when > 1 is ready
  - There are many possibilities
- Tie-breaking
  - how does scheduler choose between two equal-priority ops?
  - Multiple levels of priority function, applied in a carefully chosen order or a weighted linear combination of multiple priority functions
- An accurate dependence graph
  - Need to represent every necessary dependence
  - Should avoid dependences that are not necessary
- Must it be complicated?
  - No. The reference scheduler uses a fairly simple set of heuristics.

What Affects Schedule Quality?

Priority Functions

- Priorities determine which operation is chosen when > 1 is ready
- Many priority functions are recommended in the literature
  - Latency-weighted depth
  - Most descendants in the graph
  - Breadth-first numbering
  - Depth-first numbering
  - Boost the priority of operations that are restricted to one functional unit
  - Boost the priority of loads over stores
- Cannot make up your mind?
  - Try a linear combination of two priority functions (α x $prio_1$ + β x $prio_2$)
  - You can then do α / β tuning:
    - → Restrict $\alpha + \beta = 1$ and perform a parameter sweep over values for α and β to find the best overall results from the schedule

Tie Breaking

Picking an operation from the *Ready* queue is critically important

Among the possible considerations:

- Operations restricted to one or the other of the functional units
  - Should it give priority to a restricted operation, or not?
- Overall progress toward the goal of scheduling all operations
- The impact of preferring one operation over another

Time tested (& sometimes contradictory) ideas

- Boost priority of multi-cycle operations (multiply, load & store)
- Boost priority of operations that have more successors
- Balance priority along all paths toward the leaves
  - Breadth first may create more **ILP** and, hence, fill more empty slots
- Boost priority of operations along the critical path
  - The only thing that matters is the critical path through the block

Classic reference is "Efficient Instruction Scheduling for a Pipelined Architecture," P.B. Gibbons and S.S. Muchnick, ACM SIGPLAN 86 Conference on Compiler Construction. URL on Lectures page.

Tie Breaking

Picking an operation from the *Ready* queue is critically important

Among the possible considerations:

- Operations restricted to one or the other of the functional units
  - Should it give priority to a restricted operation, or not?
- Overall progress toward the goal of scheduling all operations
- The impact of preferring one operation over another

Encode tie breaking directly into your priority function

- Go back to the **α**-**β** tuning idea and use **β** for your tie-breaker
- For tie breaking, you could bias strongly toward one metric
  - e.g., **α** = 0.95 and **β** = 0.05 (might become **α**, **β**, **γ** tuning)
  - Result is a number that breaks ties in $prio_1$ with values from $prio_2$
  - Simple arithmetic replacing complicated nests of **if-then-else** statements

Tie Breaking

Picking an operation from the *Ready* queue is critically important

Among the possible considerations:

- Operations restricted to one or the other of its
  - Should it give priority to a restricted
- Overall progress toward the
- The impact of pref

Encod ity function

- Go
- Set
  - R mber that breaks ties in $prio_1$ with values from $prio_2$
  - Sim arithmetic replacing complicated nests of **if-then-else** statements

Yes. These are contradictory suggestions.
The problem is **NP**-Complete for a reason.
There is no easy answer.

What Affects Schedule Quality?

What goes into creating a good schedule?

- A good priority function
  - Priorities determine which operation is chosen when > 1 is ready
  - There are many possibilities
- Tie-breaking
  - how does scheduler choose between two equal-priority ops?
  - Multiple levels of priority function, applied in a carefully chosen order or a weighted linear combination of multiple priority functions
- An accurate dependence graph
  - Need to represent every necessary dependence
  - Should avoid dependences that are not necessary
- Must it be complicated?
  - No. The reference scheduler uses a fairly simple set of heuristics.

From local scheduling lecture

Instruction Scheduling (Operational View)

2. Build a dependence graph to capture critical relationships in the code

Create an empty map, *M*
walk the block, top to bottom
  at each operation *o* that defines $\mathbf{VR}_i$:
    create a node for *o* [1]
    set *M*($\mathbf{VR}_i$) to *o*
    for each $\mathbf{VR}_j$ used in *o*, add an edge from *o* to the node in M($\mathbf{VR}_j$)
    if *o* is a **load**, **store**, or **output** operation, add edges to ensure serialization of memory ops [2]

O($n + m^2$) where m is |memory ops|

Building the Graph

Explanatory Notes

1. '*o*' refers to both the operation in the original block and the node that represents it in the graph.

   The meaning should be clear by context.

2. **At a minimum, in the absence of other information:**
   - **load** & **output** need an edge to the most recent store *(full latency)*
   - **output** needs an edge to the most recent **output** (*serialization*)
   - **store** needs an edge to the most recent **store** & **output** , as well as each previous **load** (*serialization*)

   If your lab tries to simplify the graph, it may need more edges for **store** & **output** nodes

From Q & A Session

Dependence Among Memory Operations

The key point is simple:

The original code is the specification for the computation

- The order of operations in the original code is correct
- Any legal schedule must produce equivalent results

Consequences

- Loads & outputs that precede a store in the original code must precede it in the scheduled code
  - Unless the scheduler can **prove** that the operations are disjoint
    - → *Store never touches the same memory location as the load or output*
- Stores that precede a load or output in the original code must precede them in the scheduled code
  - Unless the scheduler can **prove** that the operations are disjoint
- Outputs must be strictly ordered, according to the original code
  - The order of execution of outputs is critical to correctness

From Q & A Session

Dependence Among Memory Operations

The key point is simple:

The original code is the specification for the computation

- The order of operations in the original code is correct
- Any legal schedule must produce equivalent results

Consequences

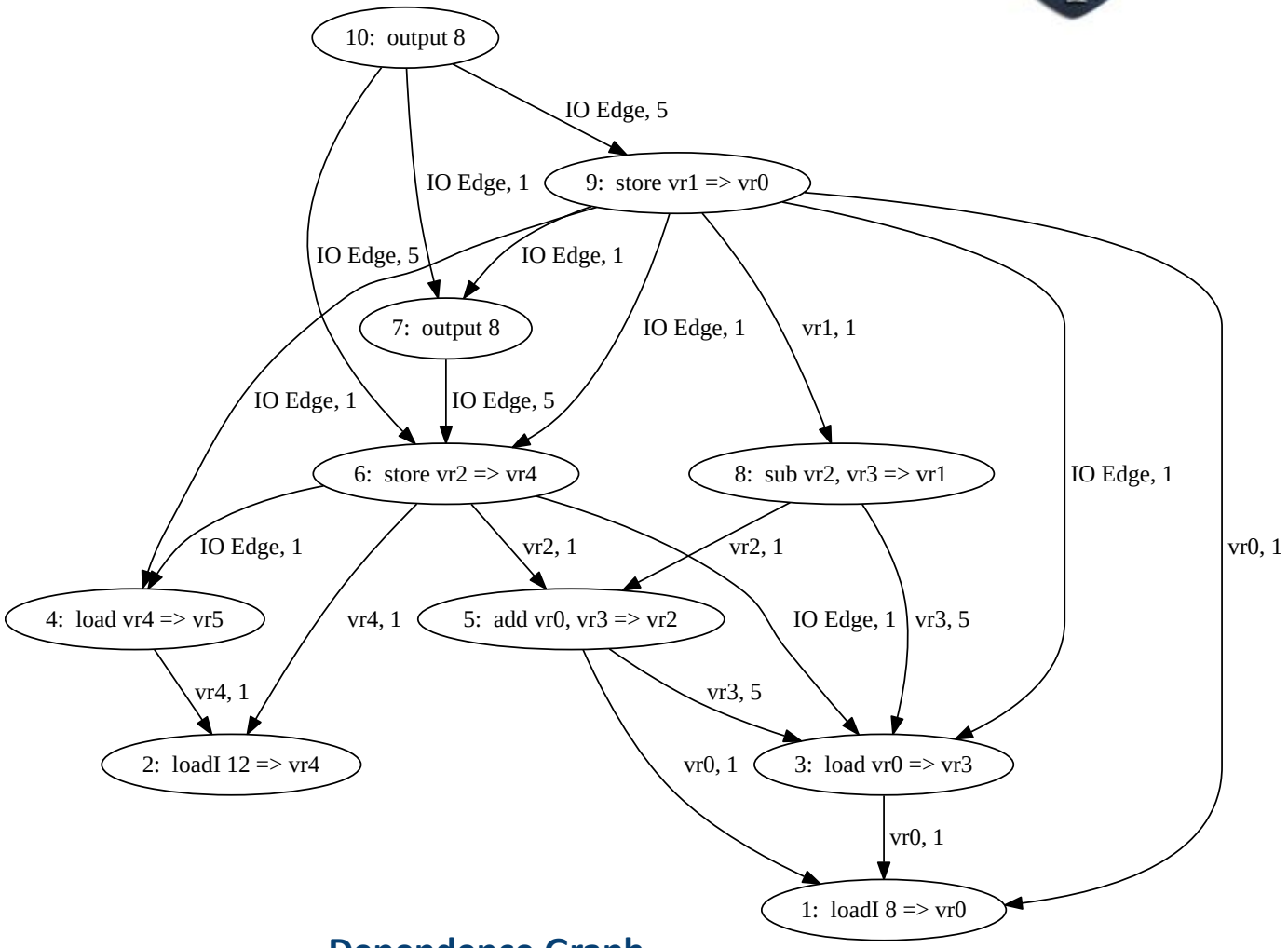
- To preserve these relationships, you will add a lot of edges to the graph
  - Those edges constrain the schedule, of course …
  - If they constrain the schedule needlessly, that hurts performance
- Many students implement some kind of graph simplification phase
  - Limits on what you can do (arbitrary, dictatorial limits)
  - Limits on what the analysis can detect

Dependence Among Memory Operations

Example

```
1    loadI 8 => r1
2    loadI 12 => r2
3    load  r1 => r3
4    load  r2 => r4
5    add   r1, r3 => r5
6    store r5 => r2
7    output 8
8    sub   r5, r3 => r6
9    store r6 => r1
10   output 8
```

Original Code



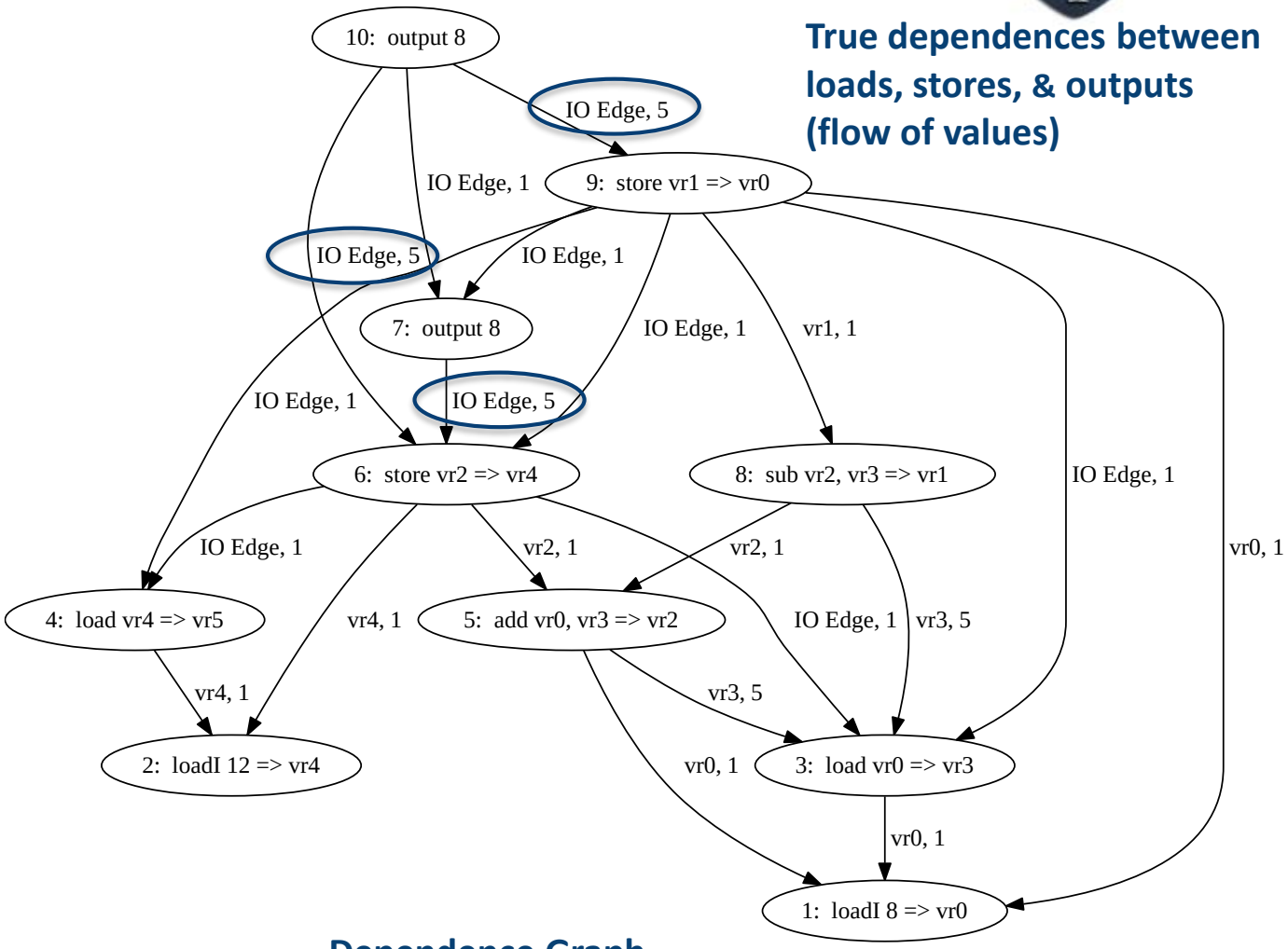
Dependence Graph

Dependence Among Memory Operations

Example

```
1   loadI 8 => r1
2   loadI 12 => r2
3   load  r1 => r3
4   load  r2 => r4
5   add   r1, r3 => r5
6   store r5 => r2
7   output 8
8   sub   r5, r3 => r6
9   store r6 => r1
10  output 8
```

True dependences between loads, stores, & outputs (flow of values)



Original Code

Dependence Graph

Dependence Among Memory Operations

Example

```
1   loadI 8 => r1
2   loadI 12 => r2
3   load  r1 => r3
4   load  r2 => r4
5   add   r1, r3 => r5
6   store r5 => r2
7   output 8
8   sub   r5, r3 => r6
9   store r6 => r1
10  output 8
```

Original Code

```
ILOC Simulator, Version 412-2015-2
Interlock settings: memory registers branches

0:   [loadI 8 => r1 (8)]
1:   [loadI 12 => r2 (12)]
2:   [load r1 (addr: 8) => r3 (20)]
3:   [load r2 (addr: 12) => r4 (30)]
4:   [ stall ]
5:   [ stall ]
6:   [ stall ] *2
7:   [add r1 (8), r3 (20) => r5 (28)] *3
8:   [store r5 (28) => r2 (addr: 12)]
9:   [output 8 (20)]
output generates => 20
10:  [sub r5 (28), r3 (20) => r6 (8)]
11:  [store r6 (8) => r1 (addr: 8)]
12:  [ stall ] *8
13:  [ stall ]
14:  [ stall ]
15:  [ stall ] *11
16:  [output 8 (8)]
output generates => 8

Executed 10 instructions and 10 operations in
17 cycles.
```

-s 3 can tell that this output is independent of the prior store because r2 (12) ≠ 8

Dependence Among Memory Operations

Example

```
1   loadI 8 => r1
2   loadI 12 => r2
3   load  r1 => r3
4   load  r2 => r4
5   add   r1, r3 => r5
6   store r5 => r2
7   output 8
8   sub   r5, r3 => r6
9   store r6 => r1
10  output 8
```

Original Code

```
ILOC Simulator, Version 412-2015-2
Interlock settings: memory registers branches

0:   [loadI 8 => r1 (8)]
1:   [loadI 12 => r2 (12)]
2:   [load r1 (addr: 8) => r3 (20)]
3:   [load r2 (addr: 12) => r4 (30)]
4:   [ stall ]
5:   [ stall ]
6:   [ stall ] *2
7:   [add r1 (8), r3 (20) => r5 (28)] *3
8:   [store r5 (28)
9:   [output 8 (20)]
output generates => 20
10:  [sub r5 (28), r3 (20) => r6 (8)]
11:  [store r6 (8) => r1 (addr: 8)]
12:  [ stall ] *8
13:  [ stall ]
14:  [ stall ]
15:  [ stall ] *11
16:  [output 8 (8)]
output generates => 8

Executed 10 instructions and 10 operations in
17 cycles.
```

-s 3 can tell that this output is dependent on the prior store because r1 (8) = 8

Dependence Among Memory Operations

Example

```
1   loadI 8 => r1
2   loadI 12 => r2
3   load  r1 => r3
4   load  r2 => r4
5   add   r1, r3
6   store r5 =>
7   output 8
8   sub   r5, r3 =>
9   store r6 => r1
10  output 8
```

Original Code

```
ILOC Simulator, Version 412-2015-2
Interlock settings: memory registers branches

0:   [loadI 8 => r1 (8)]
1:   [loadI 12 => r2 (12)]
2:   [load r1 (addr: 8)
3:   [load r2
4:   [
                                    => r6 (8)]
                    => r1 (addr: 8)]
          ] *8
     [ stall ]
14:  [ stall ]
15:  [ stall ] *11
16:  [output 8 (8)]
output generates => 8

Executed 10 instructions and 10 operations in
17 cycles.
```

-s 3 option on the simulator disambiguates the memory operations at runtime, when the values are known.

Your scheduler cannot, in general, do as well.

It can still do pretty well.

Dependence Among Memory Operations

Example

```
1   loadI 8 => r1
2   loadI 12 => r2
3   load  r1 => r3
4   load  r2 => r4
5   add   r1, r3 => r5
6   store r5 => r2
7   output 8
8   sub   r5, r3 => r6
9   store r6 => r1
10  output 8
```

Original Code

```
diana% ../412sched -n -s Lab3Perf-2.i
[loadI 8 => r0; loadI 12 => r4]
[load r0 => r3; nop]
[load r4 => r5; nop]
[nop; nop]
[nop; nop]
[nop; nop]
[add r0, r3 => r2; nop]
[store r2 => r4; sub r2, r3 => r1]
[nop; nop]
[nop; nop]
[nop; nop]
[nop; nop]
[output 8; nop]
[store r1 => r0; nop]
[nop; nop]
[nop; nop]
[nop; nop]
[nop; nop]
[output 8; nop]
diana%
```

Scheduler cannot tell whether or not r4 = 8, so it must insert **nop**s.

Dependence Among Memory Operations

Example

```
1   loadI 8 => r1
2   loadI 12 => r2
3   load  r1 => r3
4   load  r2 => r4
5   add   r1, r3 => r5
6   store r5 => r2
7   output 8
8   sub   r5, r3 => r6
9   store r6 => r1
10  output 8
```

```
diana% ../412sched -n -s Lab3Perf-2.i
[loadI 8 => r0; loadI 12 => r4]
[load r0 => r3; nop]
[load r4 => r5; nop]
[nop; nop]
[nop; nop]
[nop; nop]
[add r0, r3 => r2; nop]
[store r2 => r4; sub r2, r3 => r1]
[nop; nop]
[nop; nop]
[nop; nop]
[nop; nop]
[output 8; nop]
[store r1 => r0; nop]
[nop; nop]
[nop; nop]
[nop; nop]
[nop; nop]
[output 8; nop]
diana%
```

r0 = 8, whether or not the scheduler knows, so it must insert **nop**s.

Execution takes 19 cycles rather than 17 cycles

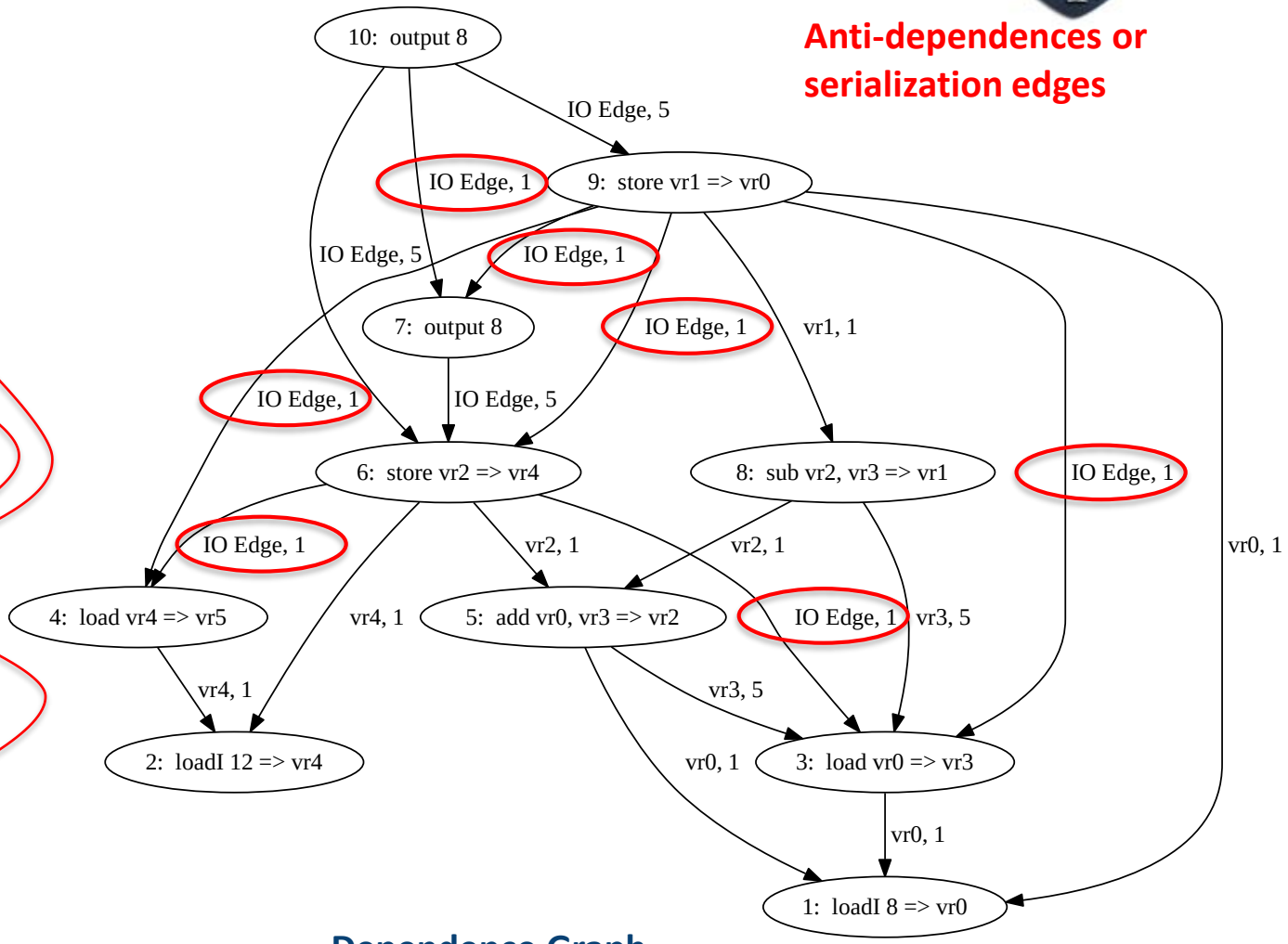
(**lab3_ref** added 4 **nop**s, but scheduled 2 ops in cycles 0 and 7)

Dependence Among Memory Operations

Example

```
1    loadI 8 => r1
2    loadI 12 => r2
3    load  r1 => r3
4    load  r2 => r4
5    add   r1, r3 => r5
6    store r5 => r2
7    output 8
8    sub   r5, r3 => r6
9    store r6 => r1
10   output 8
```

Anti-dependences or serialization edges



Original Code

Dependence Graph

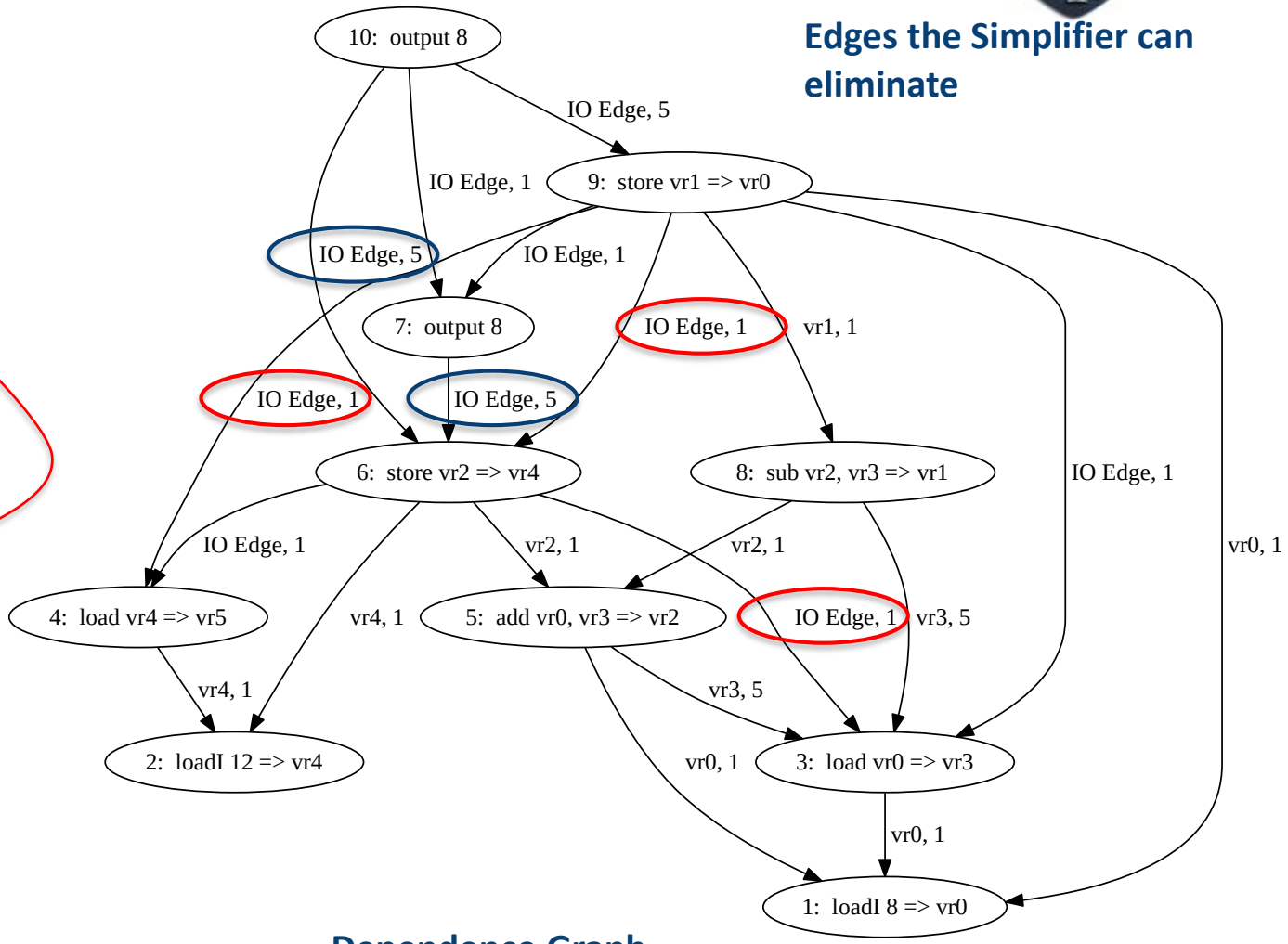
Dependence Among Memory Operations

Example

```
1   loadI 8 => r1
2   loadI 12 => r2
3   load  r1 => r3
4   load  r2 => r4
5   add   r1, r3 => r5
6   store r5 => r2
7   output 8
8   sub   r5, r3 => r6
9   store r6 => r1
10  output 8
```

Original Code

Edges the Simplifier can eliminate



Dependence Graph

Notice the edge from 7 to 6

Dependence Among Memory Operations

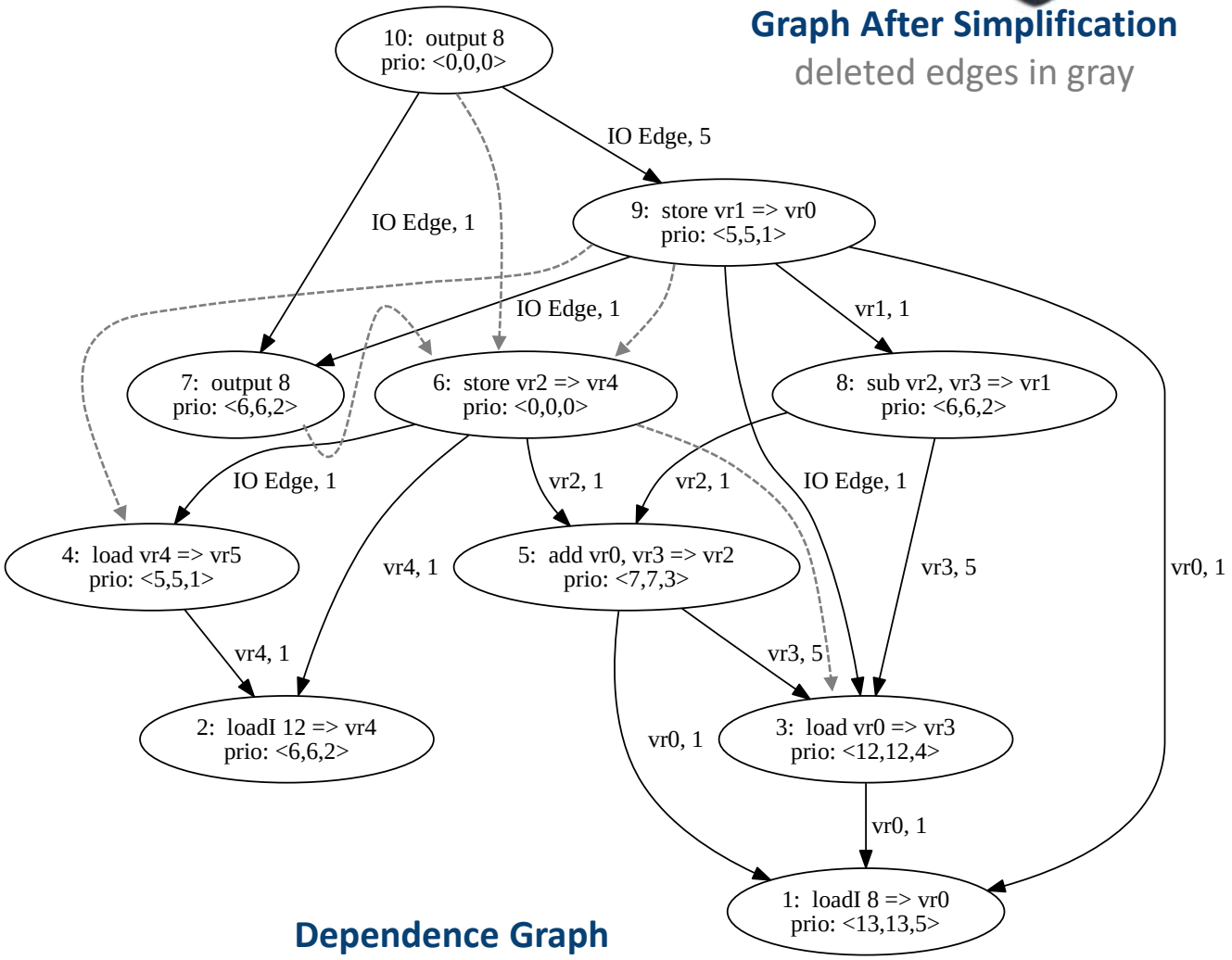
Example

```
1    loadI 8 => r1
2    loadI 12 => r2
3    load  r1 => r3
4    load  r2 => r4
5    add   r1, r3 => r5
6    store r5 => r2
7    output 8
8    sub   r5, r3 => r6
9    store r6 => r1
10   output 8
```

Original Code

Graph After Simplification

deleted edges in gray



Dependence Graph

The edge (7,6) is gone, so nodes moved.

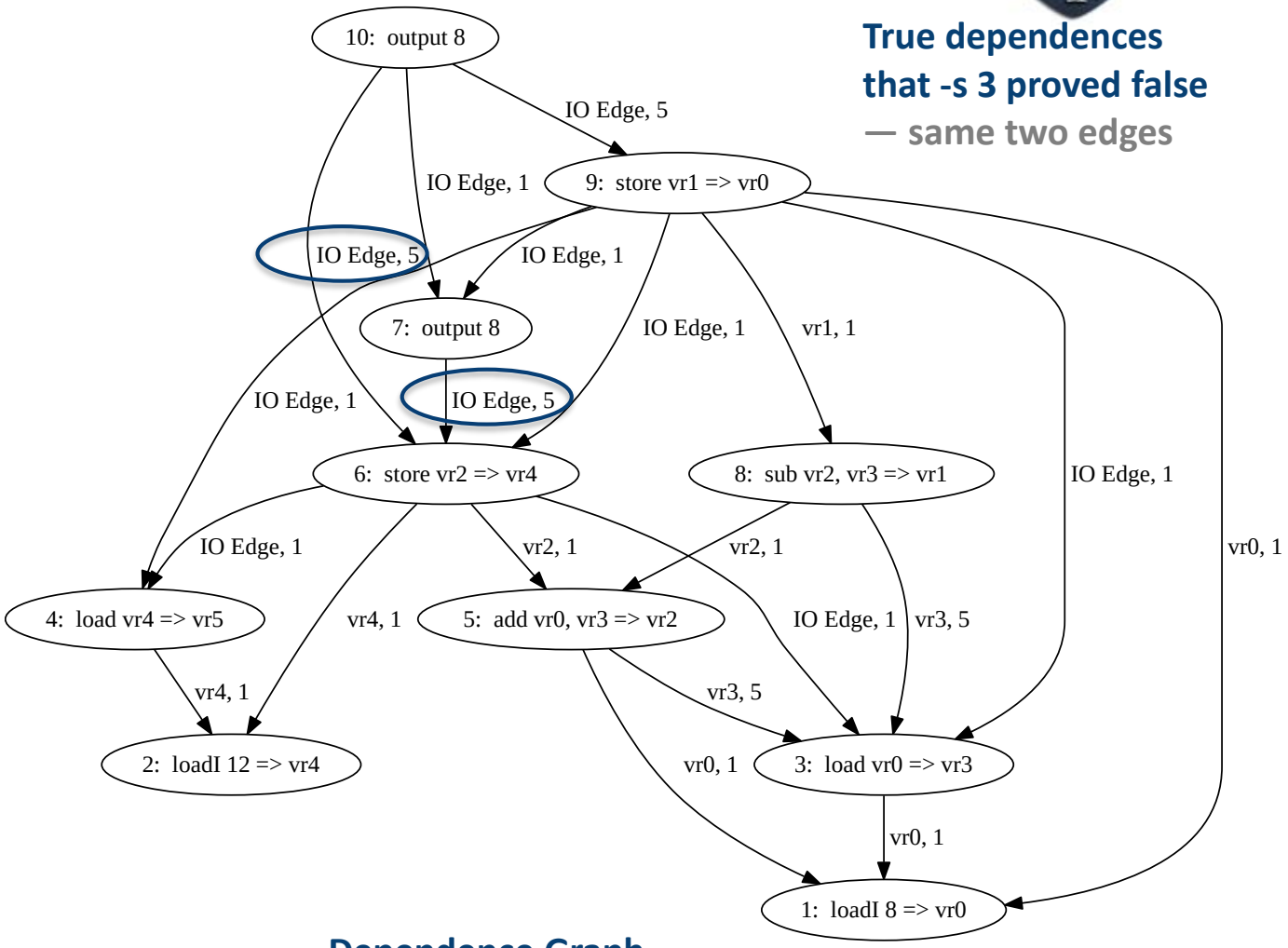
Dependence Among Memory Operations

Example

```
1    loadI 8 => r1
2    loadI 12 => r2
3    load  r1 => r3
4    load  r2 => r4
5    add   r1, r3 => r5
6    store r5 => r2
7    output 8
8    sub   r5, r3 => r6
9    store r6 => r1
10   output 8
```

Original Code

True dependences that -s 3 proved false — same two edges



Dependence Graph

Dependence Among Memory Operations

Example

```
1   loadI 8 => r1
2   loadI 12 => r2
3   load  r1 => r3
4   load  r2 => r4
5   add   r1, r3 => r5
6   store r5 => r2
7   output 8
8   sub   r5, r3 => r6
9   store r6 => r1
10  output 8
```

Original Code

```
diana% ../412sched -g Lab3Perf-2.i
[loadI 8 => r0; output 8]
[load r0 => r3; loadI 12 => r4]
[load r4 => r5; nop]
[nop; nop]
[nop; nop]
[nop; nop]
[add r0, r3 => r2; nop]
[store r2 => r4; sub r2, r3 => r1]
[store r1 => r0; nop]
[nop; nop]
[nop; nop]
[nop; nop]
[nop; nop]
[output 8; nop]
diana%
```

Changed these op's positions & timing (nops)

Could not eliminate these nops

In this example, the removed anti-dependence edges did not improve the speed of the code. In some cases, they will.

Dependence Among Memory Operations

Example

```
1   loadI 8 => r1
2   loadI 12 => r2
3   load  r1 => r3
4   load  r2 => r4
5   add   r1, r3 => r5
6   store r5 => r2
7   output 8
8   sub   r5, r3 => r6
9   store r6 => r1
10  output 8
```

Original Code

```
ILOC Simulator, Version 412-2015-2
Interlock settings: branches

0:   [loadI 8 => r0 (8); output 8 (20)]
output generates => 20
1:   [load r0 (addr: 8) => r3 (20); loadI 12 => r4 (12)]
2:   [load r4 (addr: 12) => r5 (30); nop ]
3:   [nop ; nop ]
4:   [nop ; nop ]
5:   [nop ; nop ] *1
6:   [add r0 (8), r3 (20) => r2 (28); nop ] *2
7:   [store r2 (28) => r4 (addr: 12);
      sub r2 (28), r3 (20) => r1 (8)]
8:   [store r1 (8) => r0 (addr: 8); nop ]
9:   [nop ; nop ]
10:  [nop ; nop ]
11:  [nop ; nop ] *7
12:  [nop ; nop ] *8
13:  [output 8 (8); nop ]
output generates => 8

Executed 14 instructions and 28 operations in 14 cycles.
```

Scary admonition

Warning: Your scheduler must not perform optimizations other than instruction scheduling and register renaming. It must not remove useless (dead) instructions; it must not fold constants. Any instruction left in the code by the optimizer is presumed to have a purpose. Assume that the optimizer had a firm basis for not performing the optimization. Your lab should schedule the instructions, not search for other opportunities for improvement. *This lab is an investigation of instruction scheduling, not a project in general optimization algorithms.*

Additionally, your scheduler must not rely on input constants specified in comments. When the TAs grade your scheduler, they are not required to use the input specified in the comments. Your scheduler is expected to generate correct code regardless of whether or not the TAs use the input specified in the comments.

If the result of an op is not used, you cannot delete the op.

Simplifying the Dependence Graph

A simplifier does not need to be complex

for each IOEdge E = <src,sink>
* if op(src) is independent of op(sink)*
* then delete edge E*

The Reference Scheduler's Algorithm

Proving independence is the hard part

- Simple comparison of **loadI** gets many of the test cases
- You may do arithmetic on the values from **loadI**s
- You may do algebra on expressions involving **loadI**s
  - E.g., @a + 4 ≠ @a + 12
- You may not rely on unknown values
  - **MEM[12]** may or may not be identical to **MEM[32]**
- You may not fold constants in the code (*see prev slide*)

Algebra Example

```
loadI 1024 => r0
  …
addI r0,4 => r1
store r0 => r1
  …
addI r0,12 => r2
load r2 => r3
  …
```

Are the load and the store independent?