INSTRUCTION-LEVEL PARALLELISM

PROGRAM ANALYSIS AND OPTIMIZATION – DCC888

Fernando Magno Quintão Pereira

fernando@dcc.ufmg.br

The material in these slides have been taken from Chapter 10 of the ”Dragon Book – Second Edition", by Aho, Lam, Sethi and Ullman.

Introduction

- Instruction Level Parallelism (ILP)♤ is a measure of how many operations in a computer program can be executed simultaneously.
  - ILP delivers parallelism even to inherently sequential programs.
- There are a few basic ways to get good ILP:
  - Instruction Pipelining: issue one instruction at a time, but overlap the execution of multiple instructions.
  - Superscalar Execution: issue multiple instructions at a time to multiple execution units to be executed in parallel.

♤: There is war going on for the ILP acronym. On one side, we have the Integer Linear Programming crew. On the other, the Instruction Level Parallelism crowd. So far, there seems to be no clear victor.

Warm up Example♧

```
void swap(int v[], int k) {
    int temp;
    temp = v[k];
    v[k] = v[k + 1];
    v[k + 1] = temp;
}
```

Below we have two different ways to translate the swap procedure on the left. Which way is better, the **red** or the **blue**?

```
add $t1, $a1, a1
add $t1, t$1, t1
add $t1, $a0, $t1

lw  $t0, 0($t1)
lw  $t2, 4($t1)
sw  $t2, 0($t1)
sw  $t0, 4($t1)
```

```
# reg $t1 = k * 2
# reg $t1 = k * 4
# reg $t1 = v + (k * 4)
# reg $t1 has address of v[k]
# reg $t0 (temp) = v[k]
# reg $t2 = v[k + 1]
# v[k] = reg $t2
# v[k + 1] = reg $t0 (temp)
```

```
add $t1, $a1, a1
add $t1, t$1, t1
add $t1, $a0, $t

lw  $t0, 0($t1)
lw  $t2, 4($t1)
sw  $t0, 4($t1)
sw  $t2, 0($t1)
```

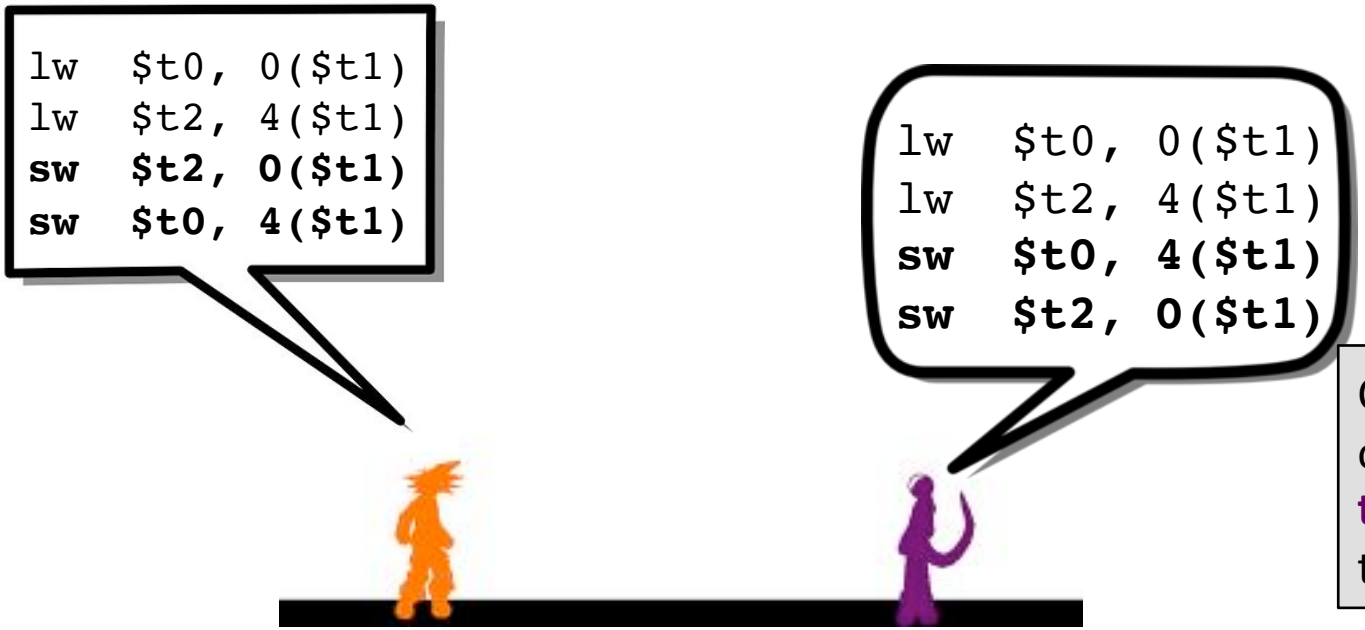
♧: taken from Computer Organization and Design, Second Edition, by Hennessy and Patterson, page 447/448

Pipelining

Each of these code snippets would be equally fast, if each instruction took exactly one cycle of processing time to execute. Yet, on a modern architecture, we must worry about pipelining. Usually an instruction is executed in small steps, called *stages*. At each stage we can, for instance: fetch an instruction from memory; read registers; perform an arithmetic computation; read memory or write registers. These operations can be performed in parallel whenever dependences do not prevent this kind of execution.

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

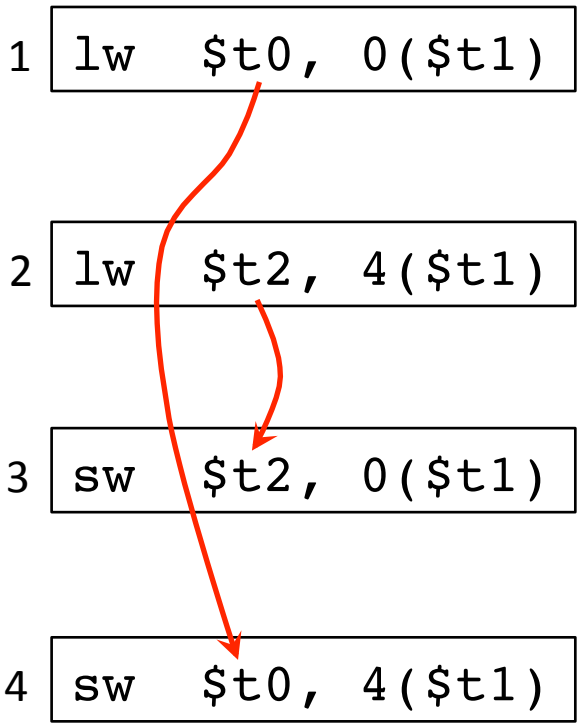
Figure taken from wikipedia commons.



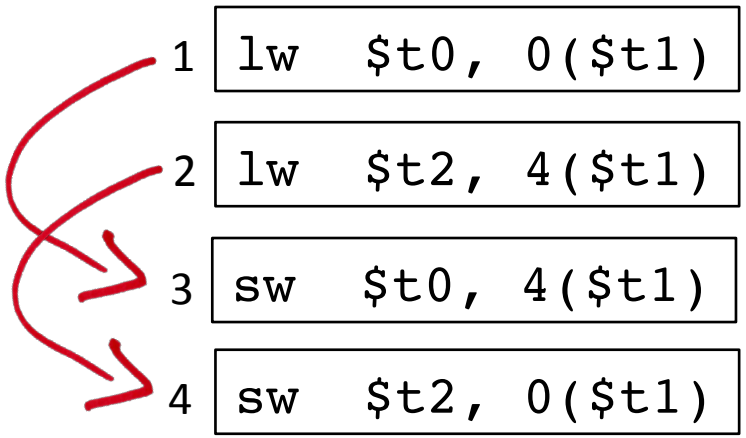
Can you see which kind of dependence makes **this** approach better than **this** one?

Data Hazards

Data hazards are situations that force the architecture to stall the pipelined execution of instructions due to data dependences. In the example on the left, we can safely execute instruction 4 after 1, because there is some processing distance between them. However, we would have to stall the pipeline after instruction 2, so that instruction 3 has all the data that it needs to execute.

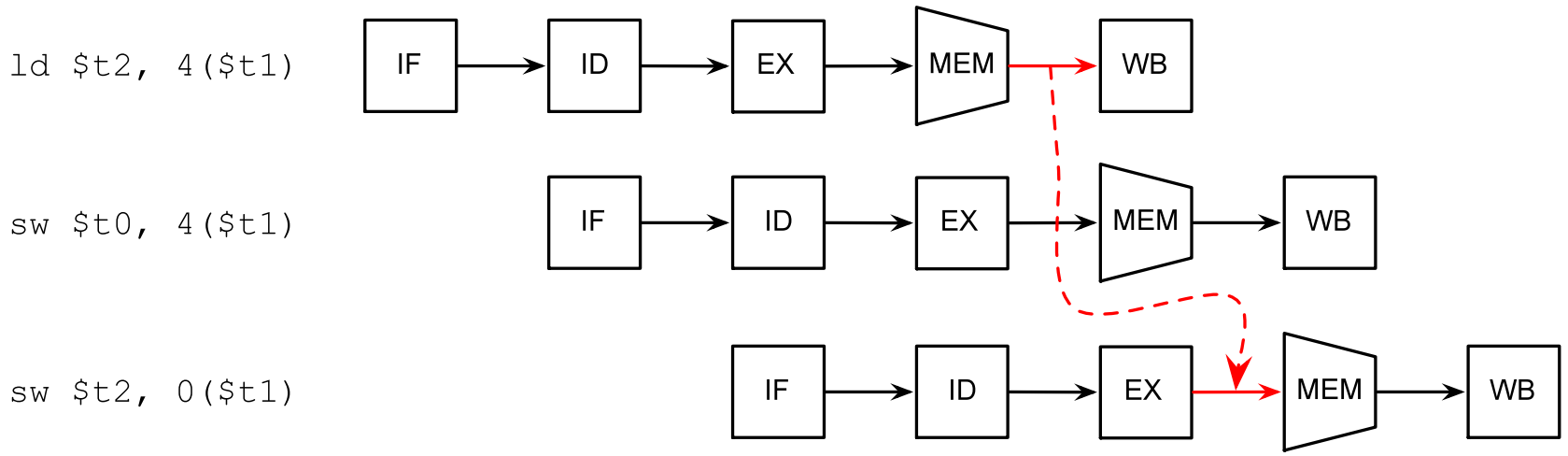


In the second example the data hazard does not exist, because each instruction has one cycle interposed between itself and the next instruction that needs its results.



Data Forwarding

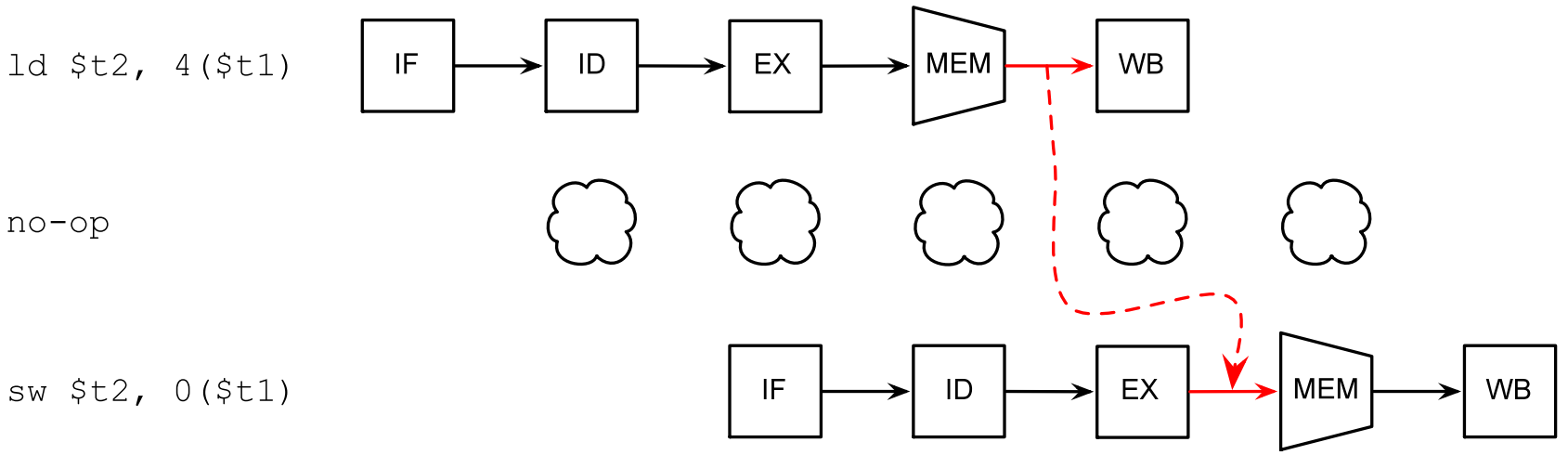
Computer architectures have different ways to mitigate the problem of data hazards. One of these ways is data forwarding: data can be sent from one instruction directly to the next instruction, before being written in registers:



Would forwarding solve our problem, if we had to process
`lw $t2, 0($t1);`
`sw $t2, 0($t1)`?

Stalls

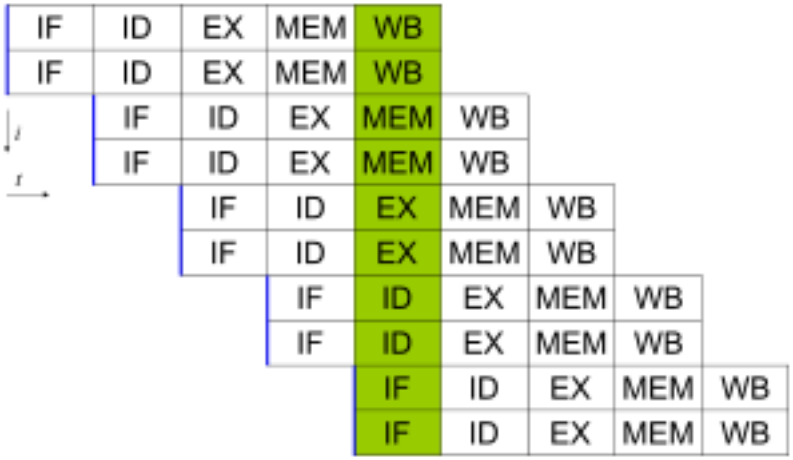
If the hardware is not able to solve dependences, be it through out-of-order execution, be it through data forwarding, then a *stall* happens. In this case, the hardware must fill up the pipeline with no-ops, e.g., instructions that will not cause any change of state in the current program execution.



Therefore, the compiler should strive to organize instructions in such a way to minimize the amount of stalls that happen during a typical execution of a program.

Superscalar Architectures

- A superscalar CPU architecture executes more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to redundant functional units on the processor♡.
- Superscalar architectures can also be pipelined.
- Some architectures use **V**ery **L**ong **I**nstruction **W**ords (VLIW), which are instructions with several operands.
- The compiler must be aware of multiple instances of a given resource.



♡: Taken from wikipedia [[w:Superscalar]]

Registers vs Parallelism

- Registers might generate dependences between instructions.

```
lw   $t0, 0($t1)
st   $t0, 4($t1)
```

```
st   $t0, 4($t1)
lw   $t0, 0($t1)
```

```
lw   $t0, 0($t1)
lw   $t0, 4($t1)
```

We cannot exchange the order within any of these groups of instructions. Why?

Registers vs Parallelism

- Registers might generate dependences between instructions.
  - True dependences: read after write.

```
lw   $t0, 0($t1)
st   $t0, 4($t1)
```

This sequence compiles v[4] = v[0]. Register $t0 creates a true dependence between the first and the second instructions.

  - Antidependence: write after read.

```
st   $t0, 4($t1)
lw   $t0, 0($t1)
```

This sequence compiles v[4] = t0; t0 = v[0]. We cannot exchange them, or we would write a different value into v[4].

  - Output dependence: write after write.

```
lw   $t0, 0($t1)
lw   $t0, 4($t1)
```

This sequence compiles t0 = v[0]; t0 = v[4]. Again, we cannot exchange them, or we would pass ahead a wrong value in t0.

Registers vs Parallelism

- Antidependences and output dependences can be minimized as long as the register allocator assigns unrelated variables to different registers.

Antidependence:

```
st   $t0, 4($t1)        st   $r0, 4($t1)
lw   $t0, 0($t1)   →     lw   $r1, 0($t1)
```

Output dependence:

```
lw   $t0, 0($t1)        lw   $r0, 0($t1)
lw   $t0, 4($t1)   →     lw   $r1, 4($t1)
```

But what is the problem of assigning these unrelated variables to different registers?

Register Allocation vs Parallelism

- The register allocator usually wants to minimize the number of registers used in the program.
  - Different variables might be allocated to the same register, as long as the live ranges of these variables do not overlap.
- Yet, if we minimize the number of registers, we might introduce dependences in the program.
  - A register, as a physical resource, forces dependences between different instructions.

```
LD t1, a♧
ST b, t1♡
LD t2, c
ST d, t2
```

If a, b, c and d are different memory addresses, what is the best allocation for this program?

♧: this is the same as t1 = *a

♡: this is the same as *b = t1

Register Allocation vs Parallelism

- The register allocator usually wants to minimize the number of registers used in the program.
  - Different variables might be allocated to the same register, as long as the live ranges of these variables do not overlap.
- Yet, if we minimize the number of registers, we might introduce dependences in the program.
  - A register, as a physical resource, forces dependences between different instructions.

```
LD t1, a          LD r0, a
ST b, t1          ST b, r0
LD t2, c          LD r0, c
ST d, t2          ST d, r0
```

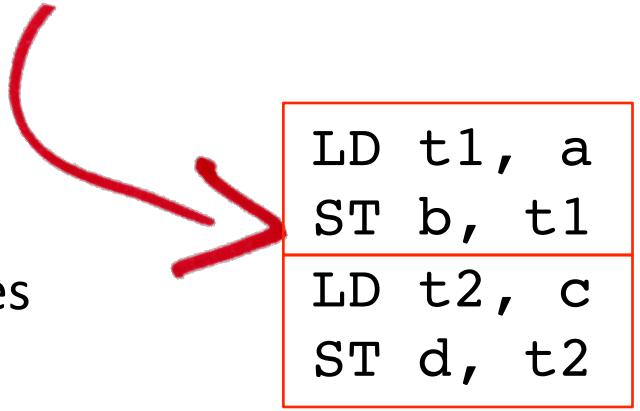
Has this allocation introduced new dependences in the program?

Register Dependences

```
1: LD r0, a
2: ST b, r0
3: LD r0, c
4: ST d, r0
```

We have two true dependences, between instructions 1 & 2, and between instructions 3 & 4. We could not avoid these dependences, because they already existed in the original program.

However, the register allocator has introduced other dependences in this program, which did not existed in the original code. We call these dependences *artificial*. They are anti-, and output dependences.



Which anti- and output dependences can you find in the allocated program?

Artificial Dependences

```
1: LD r0, a
2: ST b, r0
3: LD r0, c
4: ST d, r0
```

There exists an anti-dependence between instructions 2 and 3. Instruction 2 reads r0, and instruction 3 writes it. If we were to invert these two instructions, then we would end up reading a wrong value in instruction 2.

There exists also an output dependence between instructions 1 and 3. Both write on r0. If we were to switch them over, then r0 would end up receiving the wrong value after these two instructions had executed.

```
1: LD r0, a
2: ST b, r0
3: LD r0, c
4: ST d, r0
```

```
LD t1, a
ST b, t1
LD t2, c
ST d, t2
```

What about the **original code**? Can we run any operation in parallel?

So, in the end, are there instructions, in the **allocated code,** that we can run in parallel?

Register Allocation vs Parallelism

```
1: LD t1, a
2: ST b, t1
3: LD t2, c
4: ST d, t2
```


These two blocks of instructions, 1 & 2 and 3 & 4, do not depend on each other. We can safely run them in parallel:

```
1: LD t1, a      3: LD t2, c
2: ST b, t1      4: ST d, t2
```

We could avoid the insertion of such dependences, had we been less spartan with the use of our registers. In particular, had we assigned t1 and t2 different registers, we would not insert any artificial dependences in our final assembly code.

```
1: LD r0, a
2: ST b, r0
3: LD r1, c
4: ST d, r1
```

How to reconcile register allocation and instruction scheduling remains an important open problem in compiler design.

Register Allocation vs Parallelism

- As a more elaborate example, consider the code used to compile the expression `(a + b) + c + (d + e)`:

```
LD  r1, a
LD  r2, b
ADD r1, r1, r2
LD  r2, c
ADD r1, r1, r2
LD  r2, d
LD  r3, e
ADD r2, r2, r3
ADD r1, r1, r2
```

Which instructions can be executed in parallel in this code?

Register Allocation vs Parallelism

- As a more elaborate example, consider the code used to compile the expression `(a + b) + c + (d + e)`:

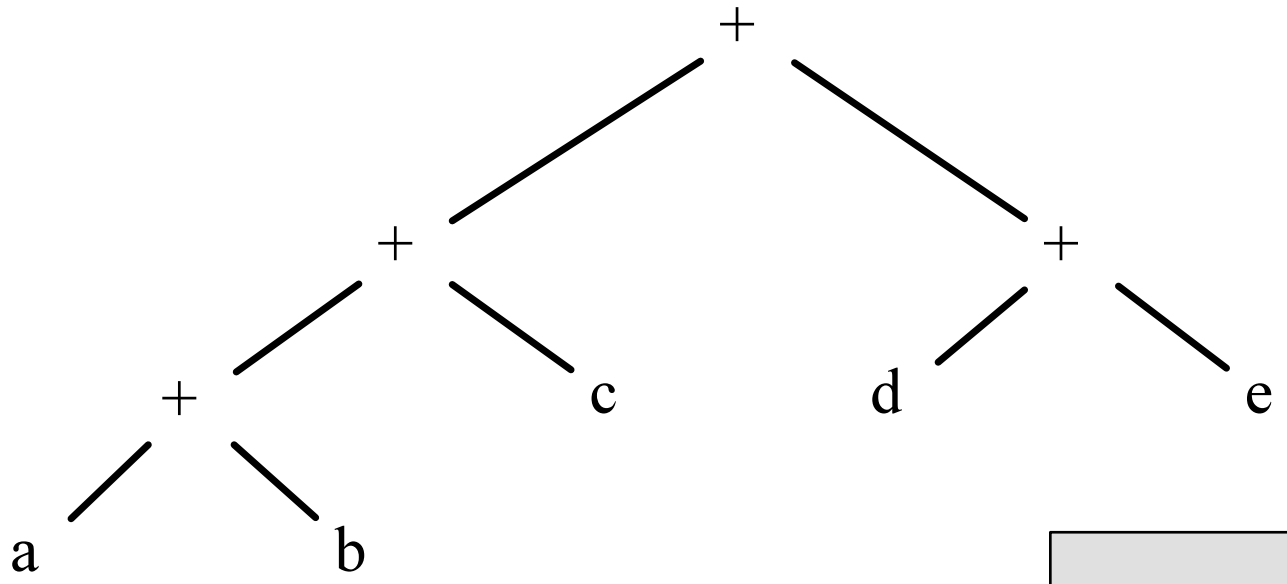
```
LD   r1, a          LD   r2, b
ADD  r1, r1, r2
LD   r2, c
ADD  r1, r1, r2
LD   r2, d          LD   r3, e
ADD  r2, r2, r3
ADD  r1, r1, r2
```

1) But, how parallel is this computation?

2) Can you conceive a way to infer this amount of parallelism?

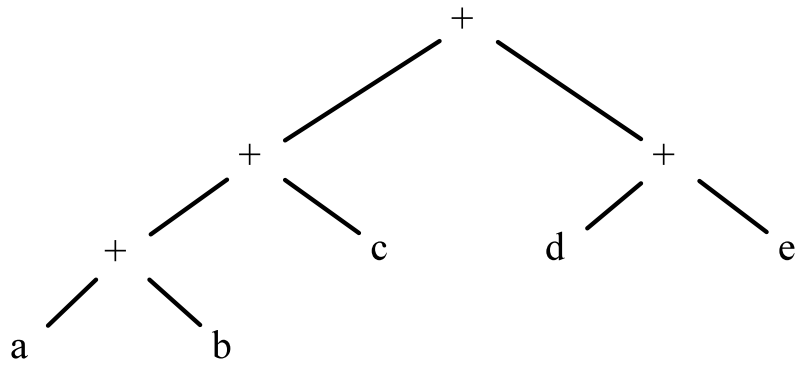
Expression Trees

Arithmetic expressions can be easily represented as trees. The expression (a + b) + c + (d + e) has the following tree representation:



Can you use this representation to infer how parallel is the computation?

The Amount of Parallelism



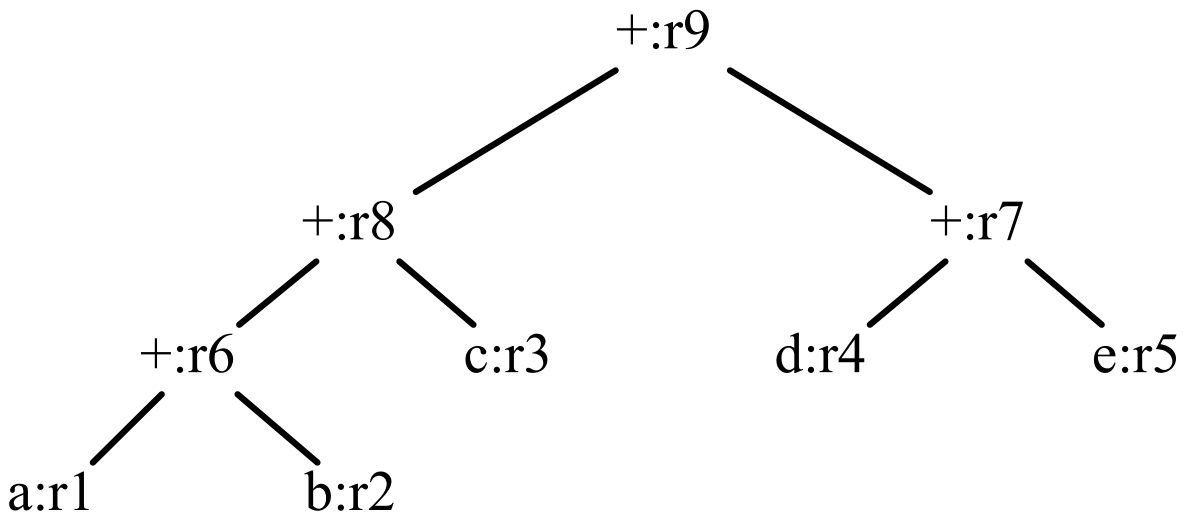
If we assign different nodes of the tree to different registers, then we remove artificial dependences, and can run more computation in parallel.

```
LD  r1, a       LD  r2, b       LD r3, c   LD r4, d   LD r5, e
ADD r6, r1, r2  ADD r7, r4, r5
ADD r8, r6, r3
ADD r9, r8, r7
```

1) What is the register assignment that we have used in this example?

2) How many instances of the same resources do we need to run this program?

The Amount of Parallelism



Could we still save registers, and be that parallel?

`LD r1, a`
`ADD r6, r1, r2`
`ADD r8, r6, r3`
`ADD r9, r8, r7`
	`LD r2, b`
	`ADD r7, r4, r5`
		`LD r3, c`	`LD r4, d`	`LD r5, e`

Compilers Laboratory

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSIDADE FEDERAL DE MINAS GERAIS
FEDERAL UNIVERSITY OF MINAS GERAIS, BRAZIL

BASIC BLOCK SCHEDULING

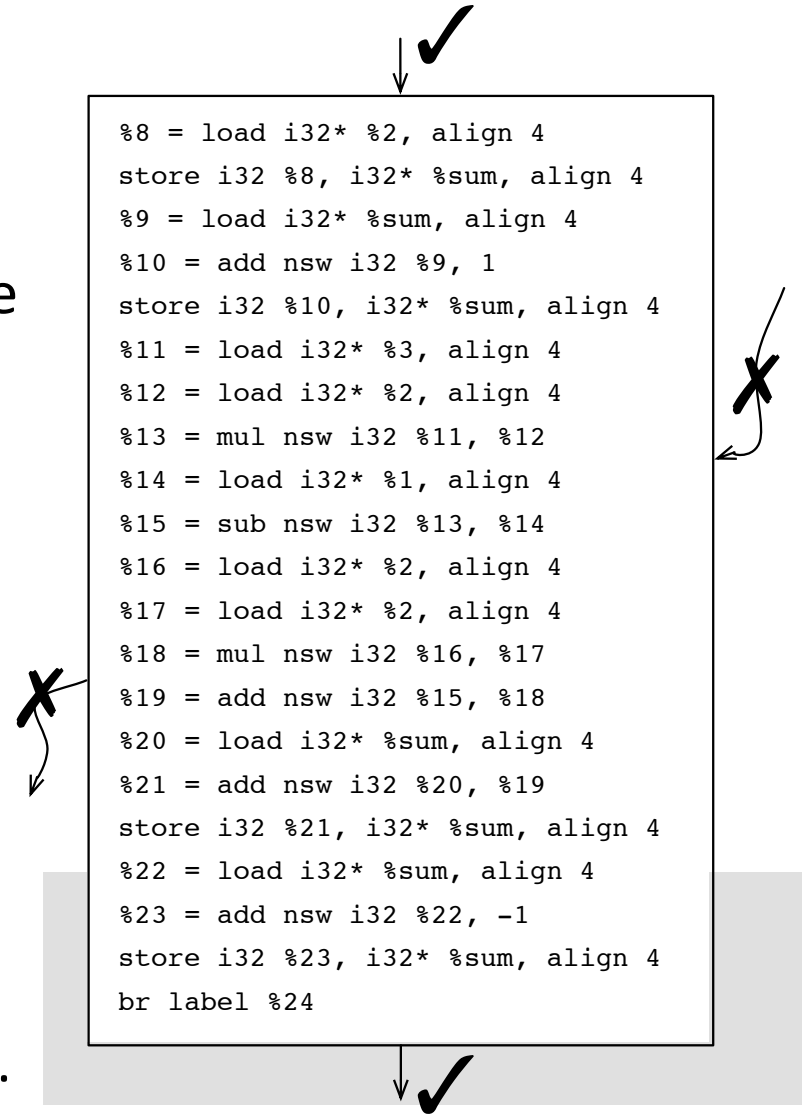
DCC 888

The Instruction Dependence Graph

- One of the main tools that helps us to schedule instructions within a basic block is the instruction dependence graph.
- This *directed* graph has one vertex for each operation in the program.
- There exist an edge from instruction i_0 to instruction i_1 if instruction i_1 uses an operand that i_0 generates.
- Each edge $e = (i_0, i_1)$ is labeled with a delay, which determines the minimum number of cycles that the hardware must wait so that the data produced by i_0 is available to i_1.

Recap: Basic Blocks

- Basic blocks are maximal sequences of consecutive instructions so that:
  - Control flow can only enter the basic block through the first instruction in the block.
  - Control will leave the block without halting or branching, except possibly at the last instruction in the block.



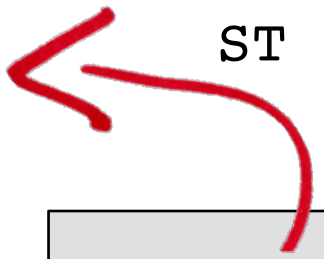
The algorithm that we describe in this section works for basic blocks only. That is why we call it Local Scheduling.

The Instruction Dependence Graph

Consider the following program, in a machine model with two types of instructions:

- Memory accesses (LD/ST), which produce operands after five cycles.
  - Unless a store to location m follows a load from location m. In this case, there is only three cycles of delay.
- Arithmetic operations (ADD, SUB, MUL, etc), which produce operands after two cycles.

```
LD   R2, 0(R1)
ST   4(R1), R2
LD   R3, 8(R1)
ADD  R3, R3, R3
ADD  R4, R3, R2
ST   12(R1), R4
ST   0(R7), R7
```

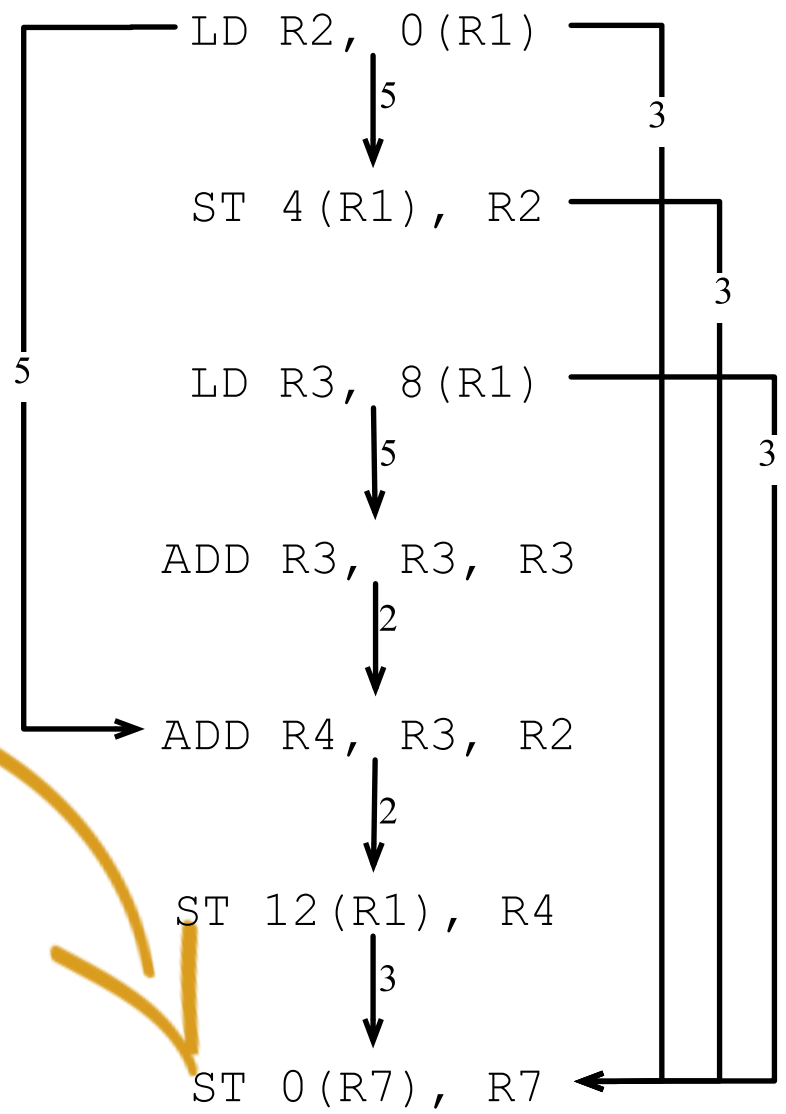


1) Why is **that** so? Which kind of feature are we modeling?

2) What is the longest delay in the program above?

The Instruction Dependence Graph

This is the graph that we obtain for our running example. The longest path, from `LD R3, 8(R1)` to `ST 0(R7),R7` has delay twelve. Thus, this is the minimum amount of time necessary to carry on this computation.

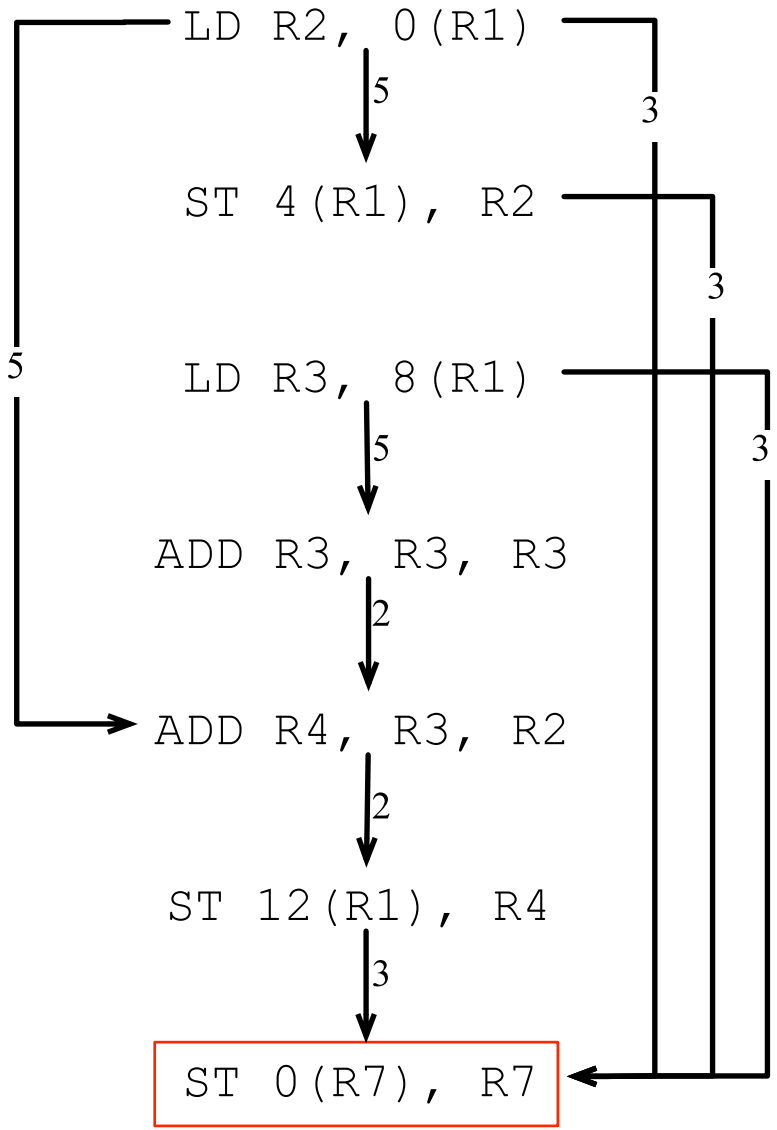


Why do we have all these dependences to the last instruction? Hint: it has to do with aliasing.

Dependences and Pointers

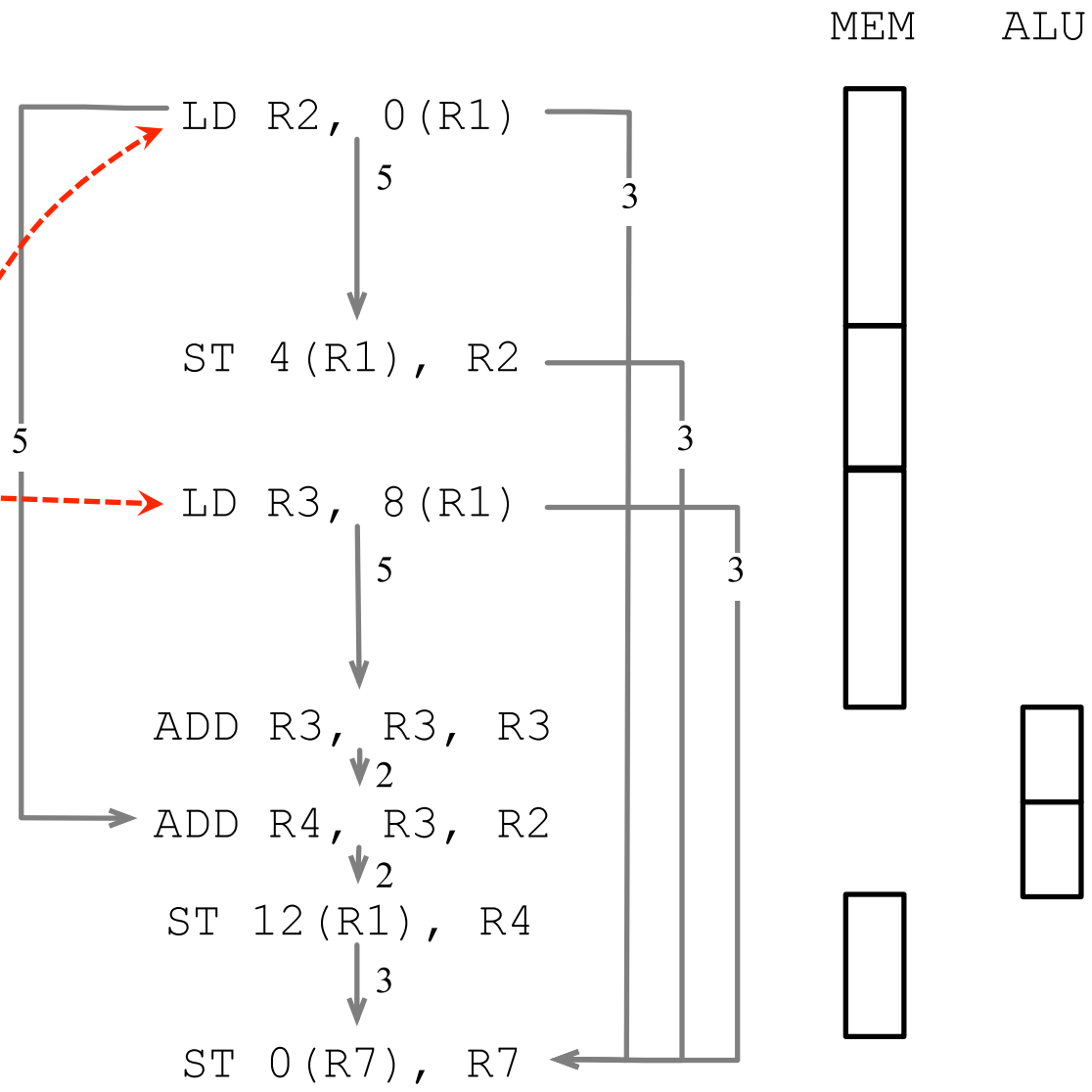
The thing is this: 0(R7) can, quite as well, be any of 0(R1), 4(R1), 8(R1) or 12(R1). We simply do not know it without alias analysis. Hence, we must assume the worst, and add dependences whenever this possibility of aliasing is true.

Easy question: why is it not necessary to force dependences between memory locations 0(R1), 4(R1), 8(R1) and 12(R1)?



Searching for a Schedule

Consider the figure on the right. It shows an scheduling of instructions in a machine where we have independent memory and arithmetic units.

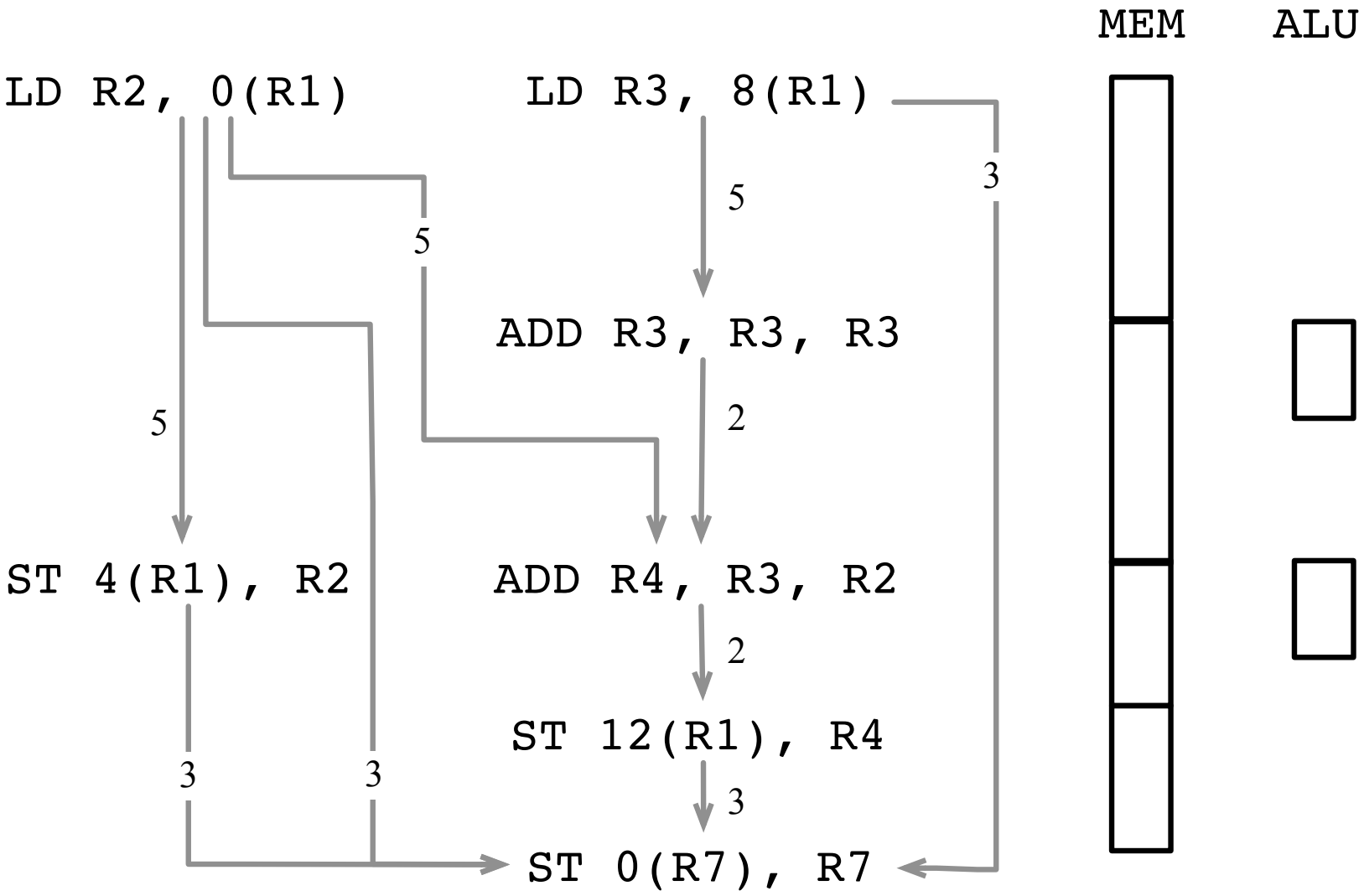


1) **These two** instructions are independent, and yet we cannot run them in parallel. Why?

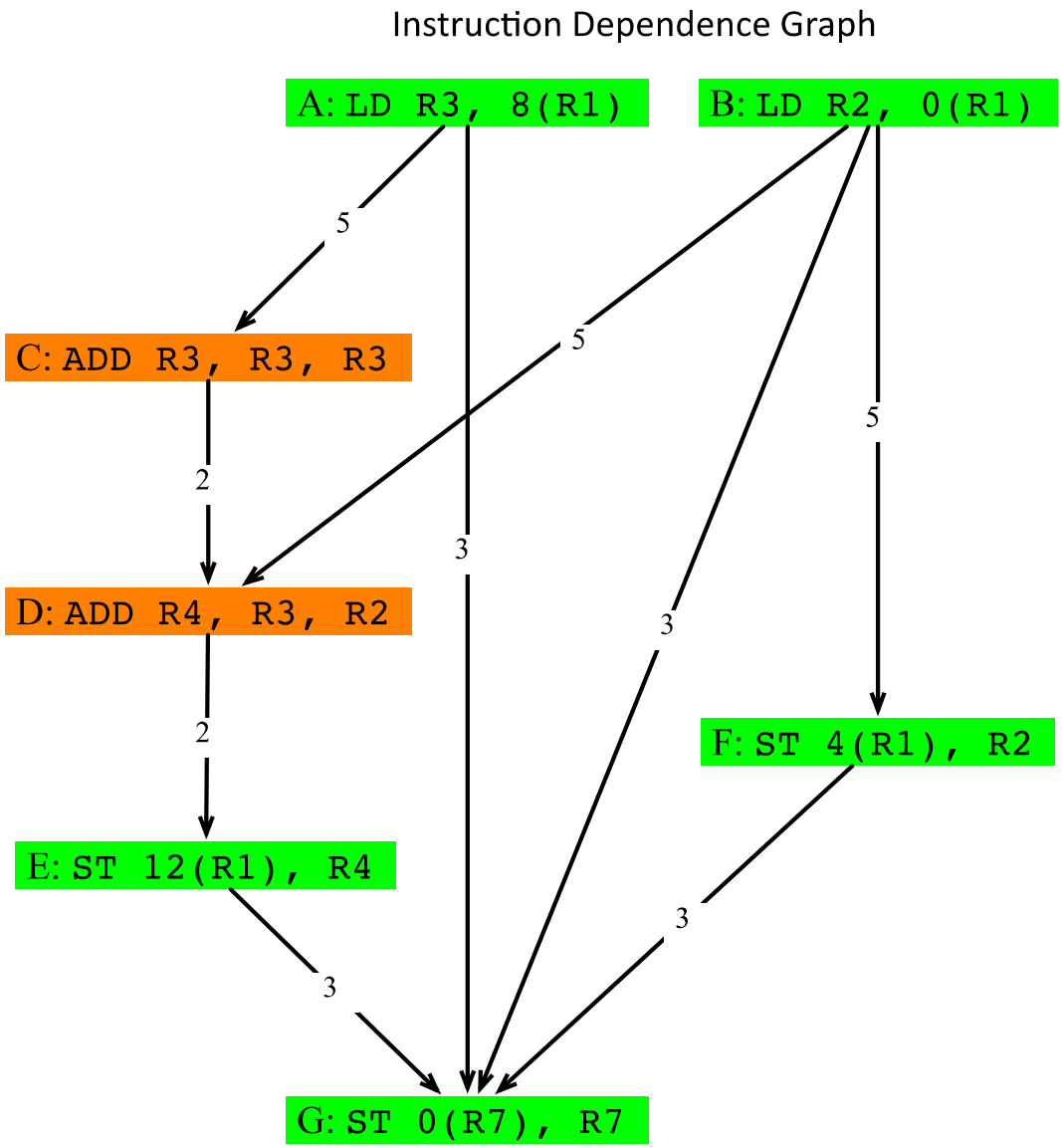
2) Do you think it is possible to find a better scheduling than this one?

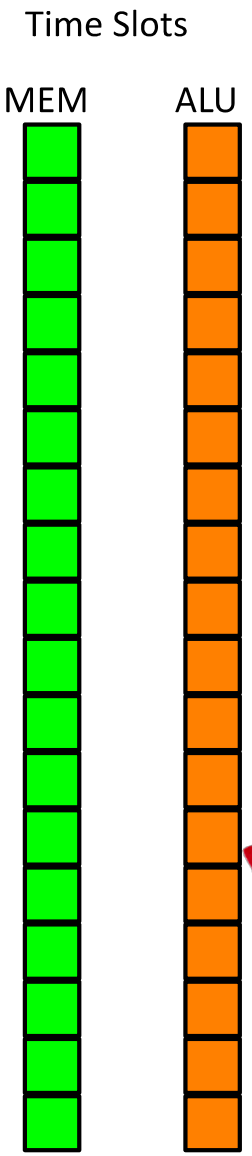
A Better Schedule

This schedule is a bit better than the previous one. The longest path here is 17 cycles long, whereas in our previous example we had a path 21 cycles long.



A Working Model





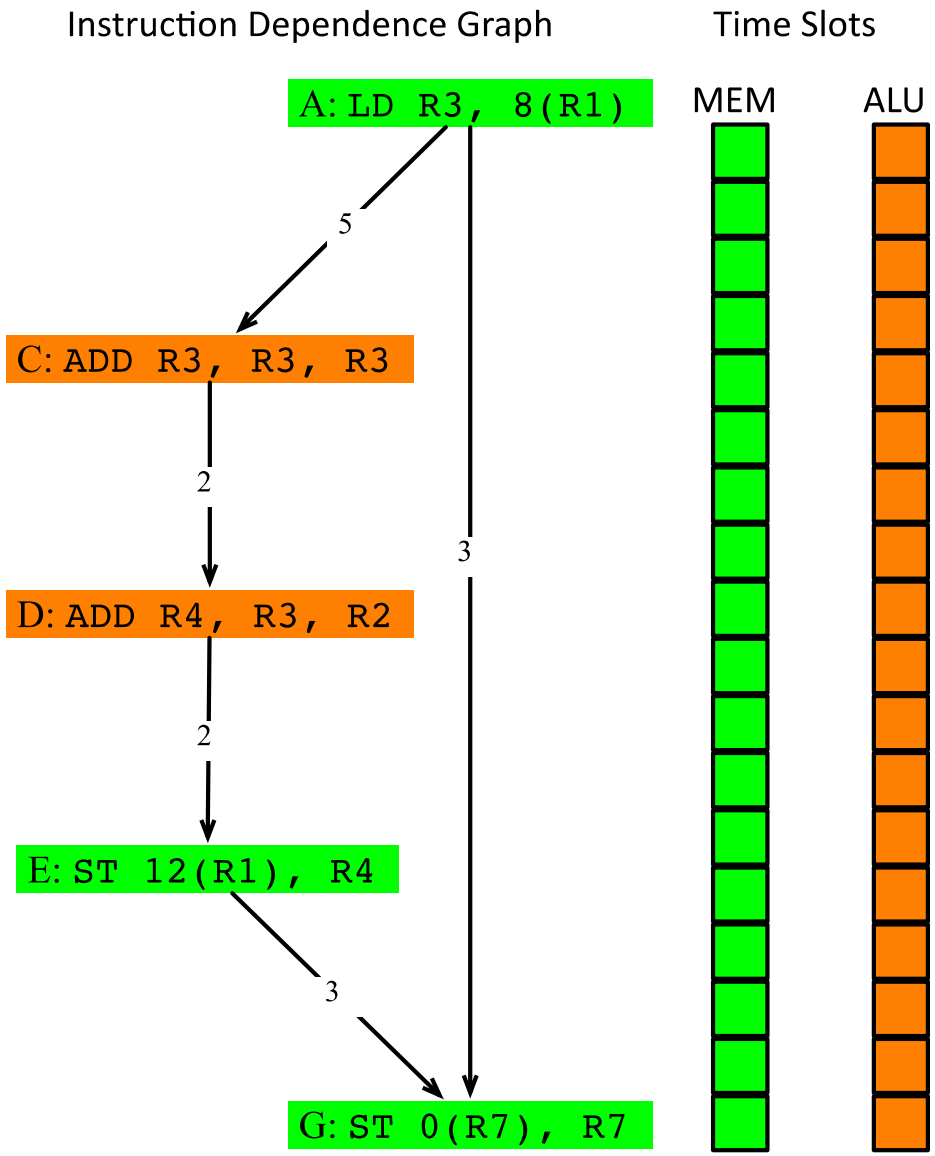
1) Can you think about a description for the scheduling problem?

2) Which information do the colors represent?

3) What are these "time slots"?

4) What is a valid solution to the scheduling problem?

A Working Model



- The colors represent the resource that each instruction uses.

- The time slots represent computation cycles.

- Our goal is to place the instructions in the time slots of same color, in such a way that Waiting times are respected.

Can you think about an algorithm that does this?

Straight-line code scheduling

RT = *empty reservation table*

foreach $v_n \in V$ *in* ***some*** *topological order*:

$s = MAX \{S(p) + d_e \mid e = p \rightarrow n \in E\}$

let $i = MIN \{x \mid RT(n, j, x) \in RT\}$ **in**

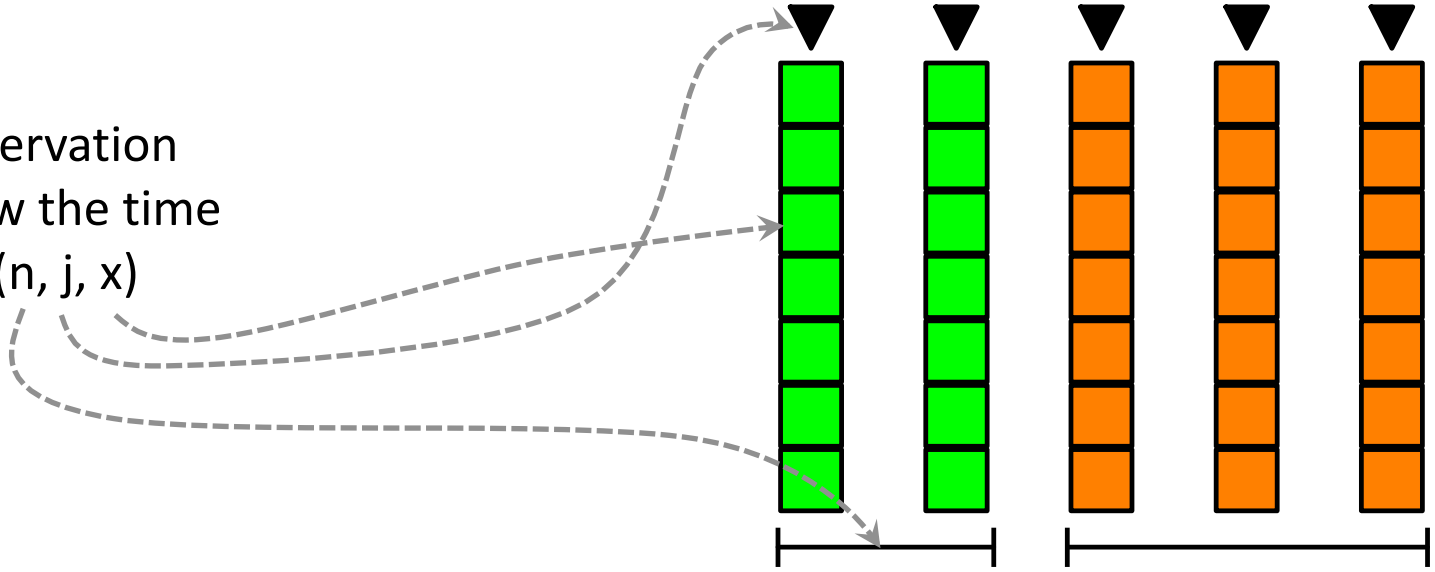
$s' = s + MAX(0, i - s)$

$RT(n, j, s') = v_n$

The next instruction to execute is the earliest available, given all the dependencies in the code.

Once an instruction has been found, we need to wait until a free resource is available to it.

The resource reservation table defines how the time slots are used. R(n, j, x)



Dependency Constraints

RT = *empty reservation table*

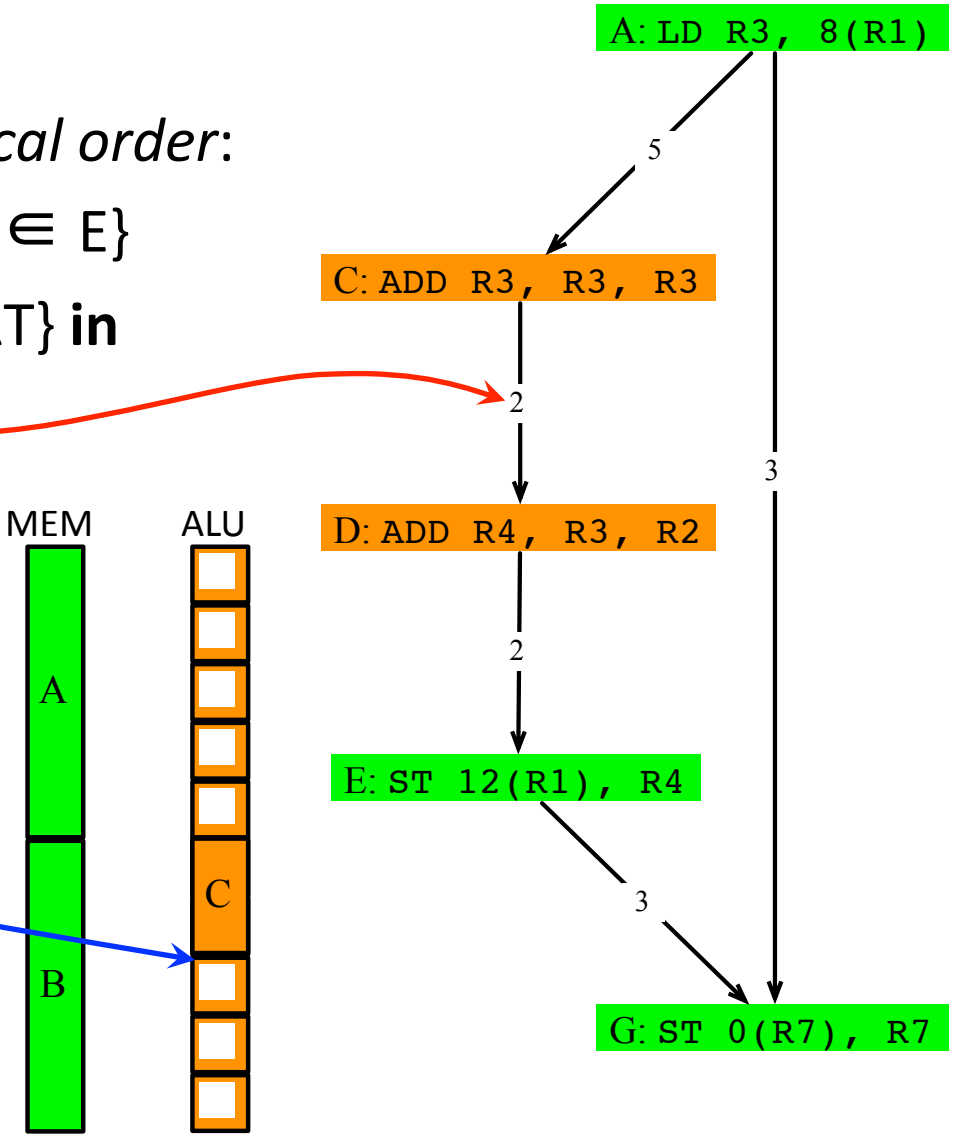
foreach $v_n \in V$ *in* ***some*** *topological order*:

$s = MAX \{S(p) + d_e \mid e = p \rightarrow n \in E\}$

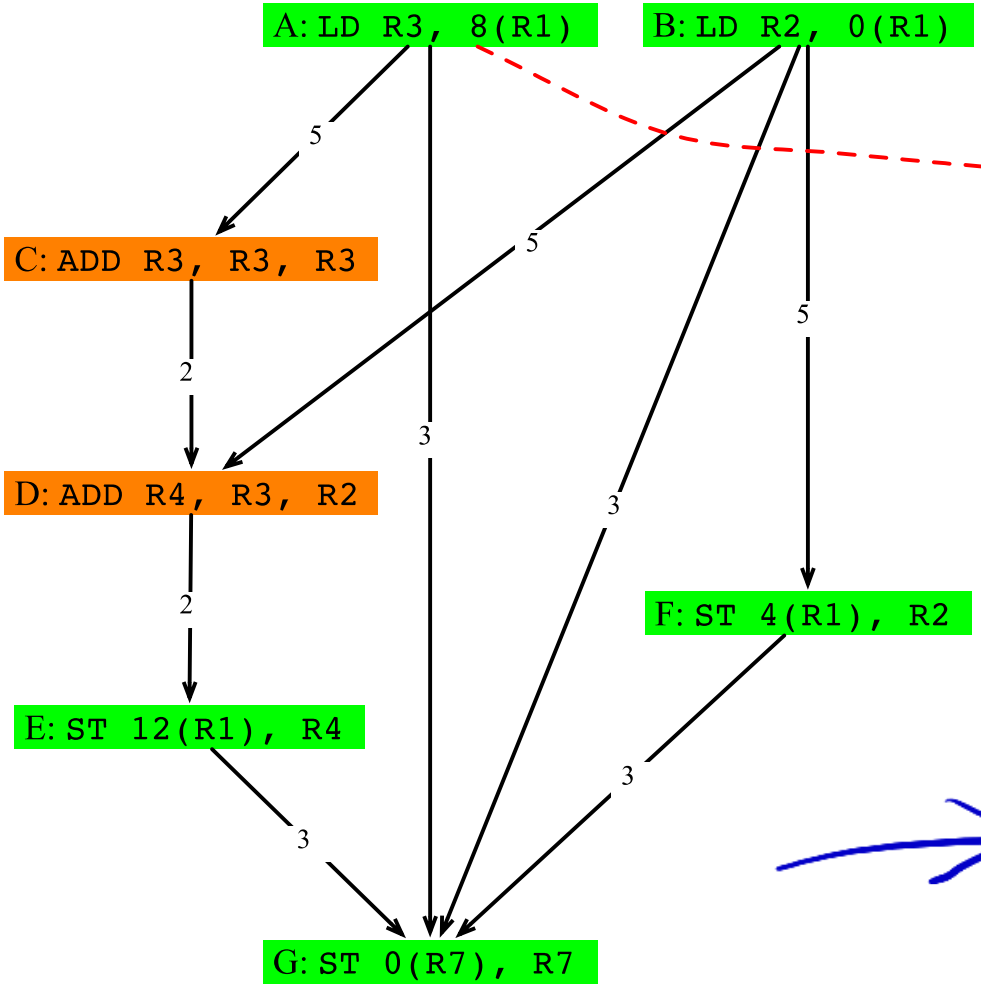
let $i = MIN \{x \mid RT(n, j, x) \in RT\}$ **in**

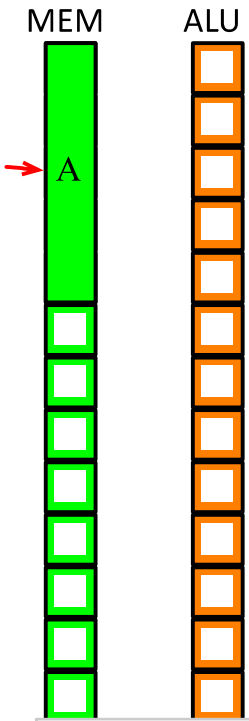
$s' = s + MAX(0, i - s)$

$RT(n, j, s') = v_n$



Basic Block Scheduling – Example (1)





We can start our scheduling with either instruction A or instruction B, because they have no predecessors. There are ways to break ties. Here, let's choose A first, because its distance to the end of the program is larger than B's.

RT = *empty reservation table*

foreach $v_n \in V$ *in* ***some*** *topological order*:

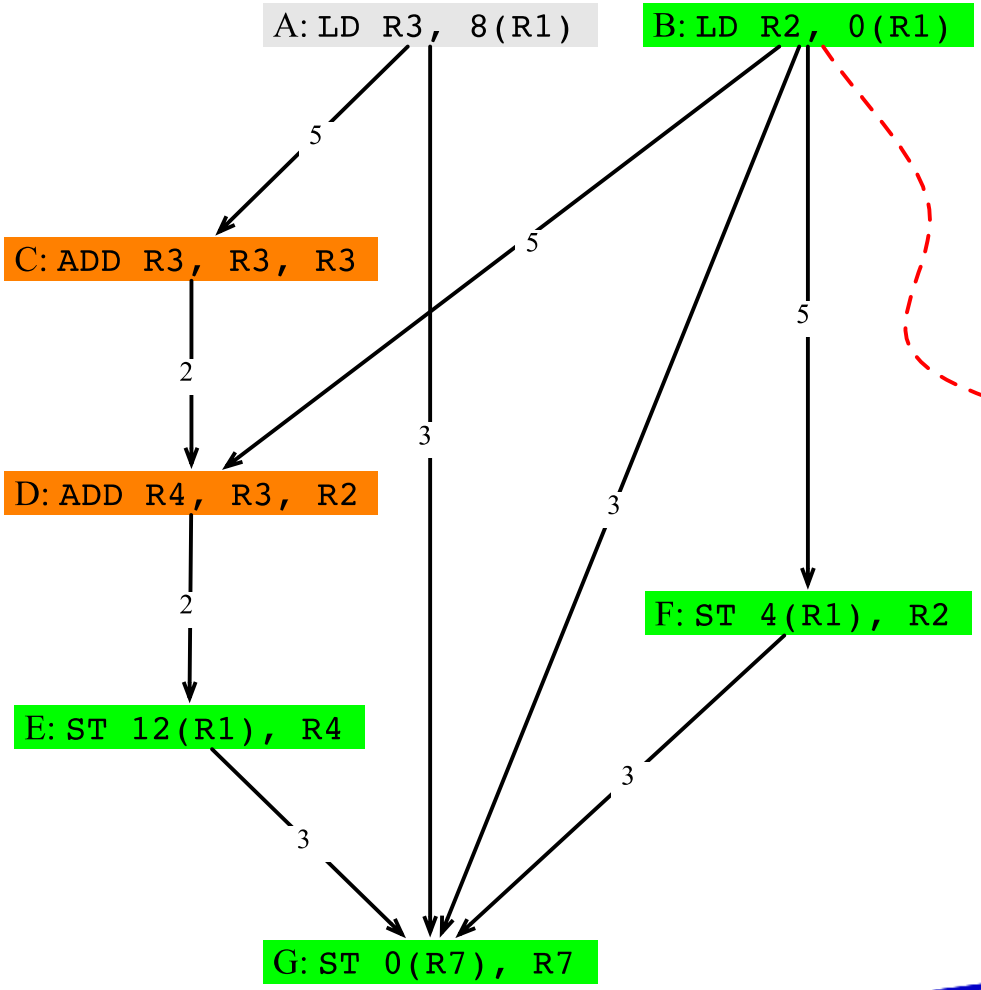
$s = MAX \{S(p) + d_e \mid e = p \rightarrow n \in E\}$

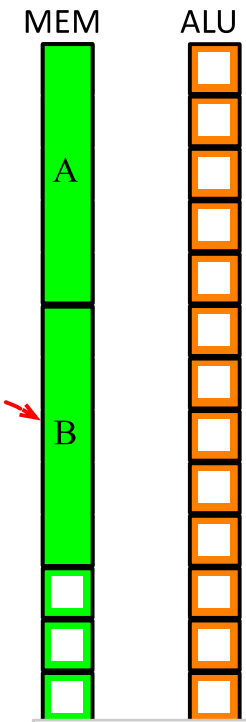
let $i = MIN \{x \mid RT(n, j, x) \in RT\}$ **in**

$s' = s + MAX(0, i - s)$

$RT(n, j, s') = v_n$

Basic Block Scheduling – Example (2)





The next instruction is, naturally, B, as it is the next in the topological ordering of instructions. However, we must schedule it five time-slots after its "s", because its resource, the memory port, is being used by instruction A. That is the job of the second part of our algorithm.

RT = *empty reservation table*

foreach $v_n \in V$ *in* ***some*** *topological order*:

$s = MAX \{S(p) + d_e \mid e = p \rightarrow n \in E\}$

let $i = MIN \{x \mid RT(n, j, x) \in RT\}$ **in**

$s' = s + MAX(0, i - s)$

$RT(n, j, s') = v_n$

Choosing the First Resource to be Free

RT = *empty reservation table*

foreach $v_n \in V$ *in* ***some*** *topological order*:

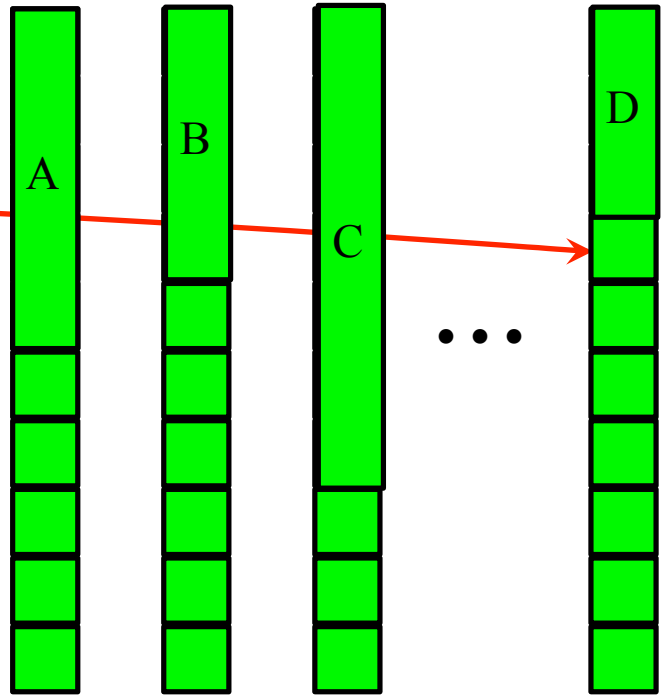
$s = MAX \{S(p) + d_e \mid e = p \rightarrow n \in E\}$

let $i = MIN \{x \mid RT(n, j, x) \in RT\}$ **in**

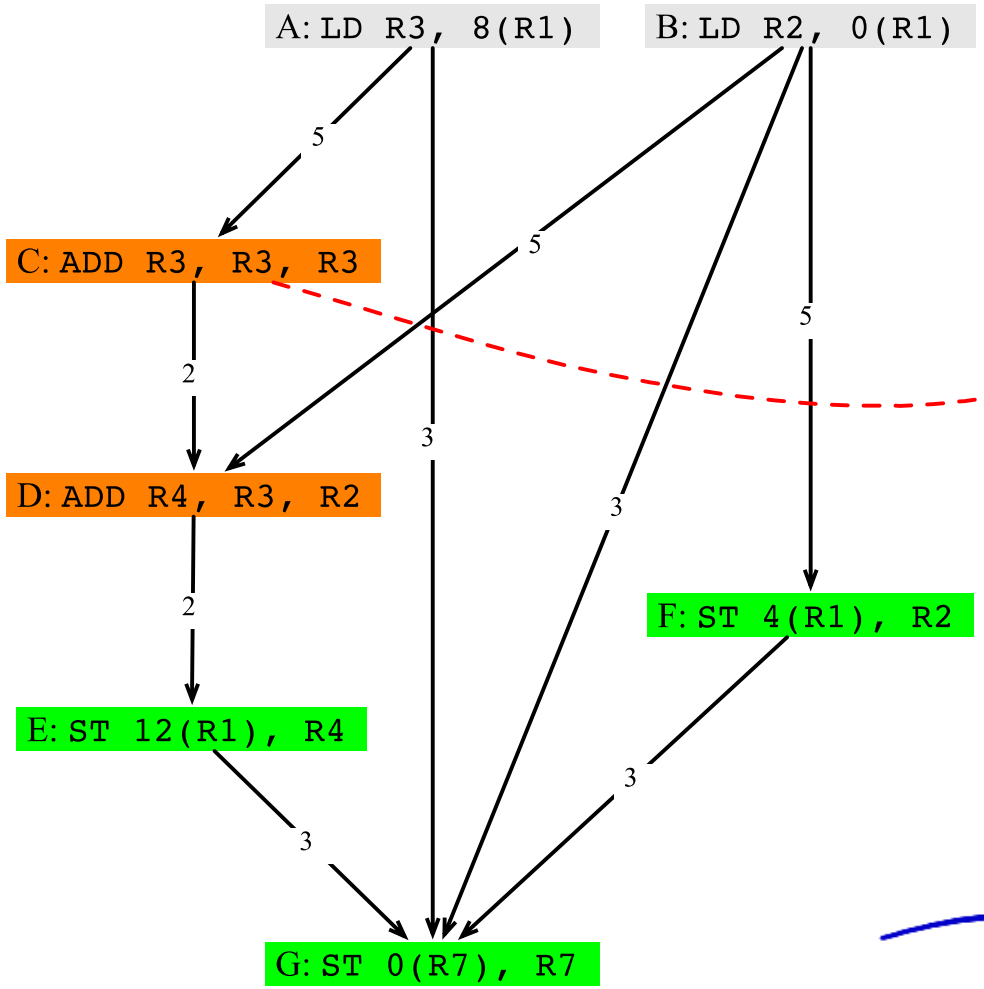
$s' = s + MAX(0, i - s)$

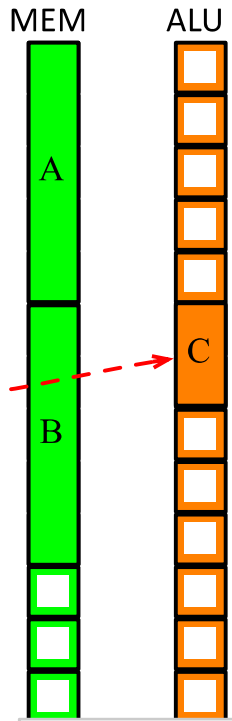
$RT(n, j, s') = v_n$

If we have many units that implement the same resource, then we should choose the **first** one that will be free to allocate the next instruction. In this way, we minimize the height of the pile of occupied resources.



Basic Block Scheduling – Example (3)





The next instruction is either C or F. Let's pick C, because its path to the end of the program is longer than F's. It is the distance s that determines where C is placed, not its allocation table. Indeed, C is the first instruction to be scheduled in the ALU.

RT = *empty reservation table*

foreach $v_n \in V$ *in* ***some*** *topological order*:

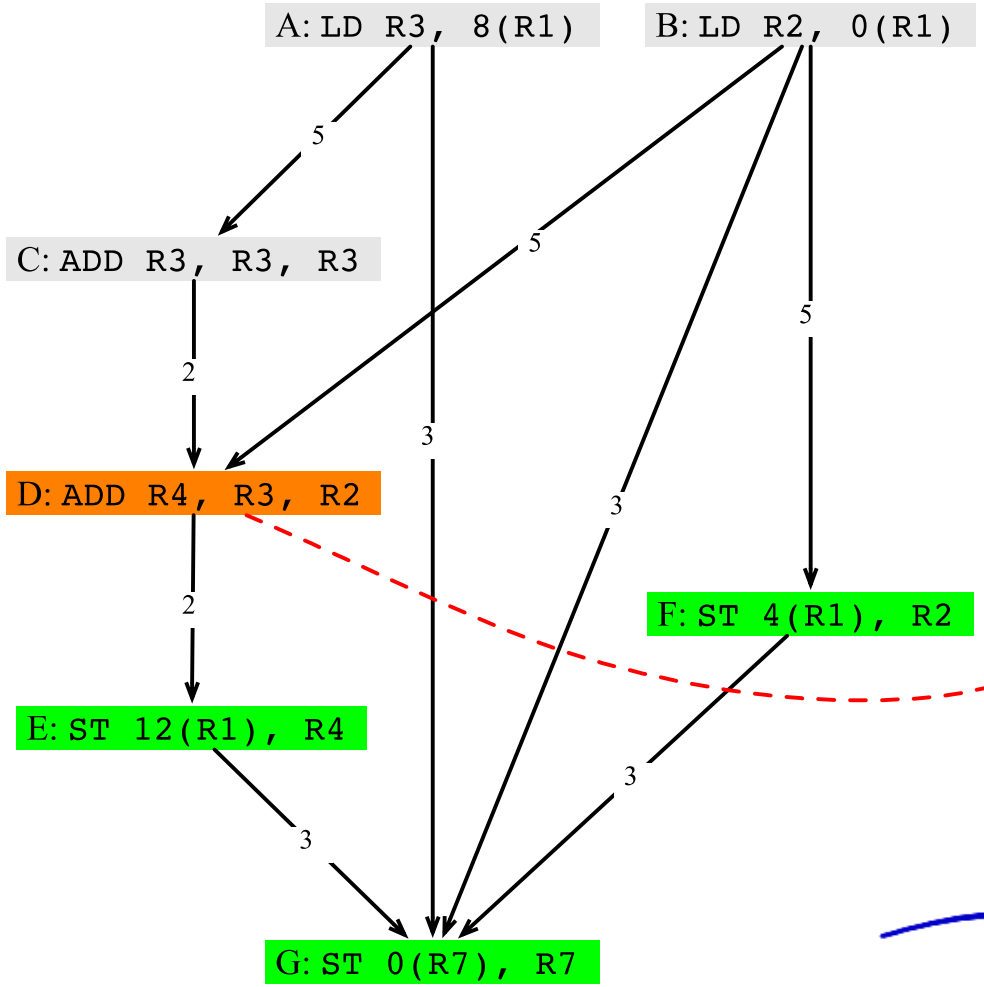
$s = MAX \{S(p) + d_e \mid e = p \rightarrow n \in E\}$

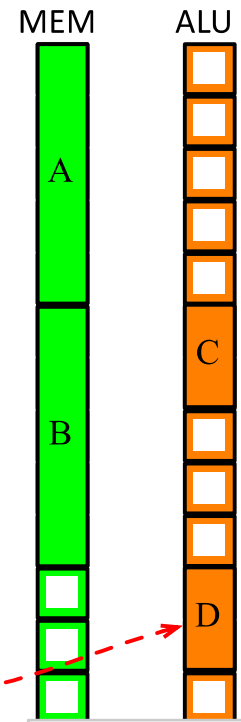
let $i = MIN \{x \mid RT(n, j, x) \in RT\}$ **in**

$s' = s + MAX(0, i - s)$

$RT(n, j, s') = v_n$

Basic Block Scheduling – Example (4)





We have two choices now. Either we grab D or F. We choose D, as it has a longer path till the end of the program. We place it at its s point, which is determined by the dependences of the instruction.

RT = *empty reservation table*

foreach $v_n \in V$ *in* ***some*** *topological order*:

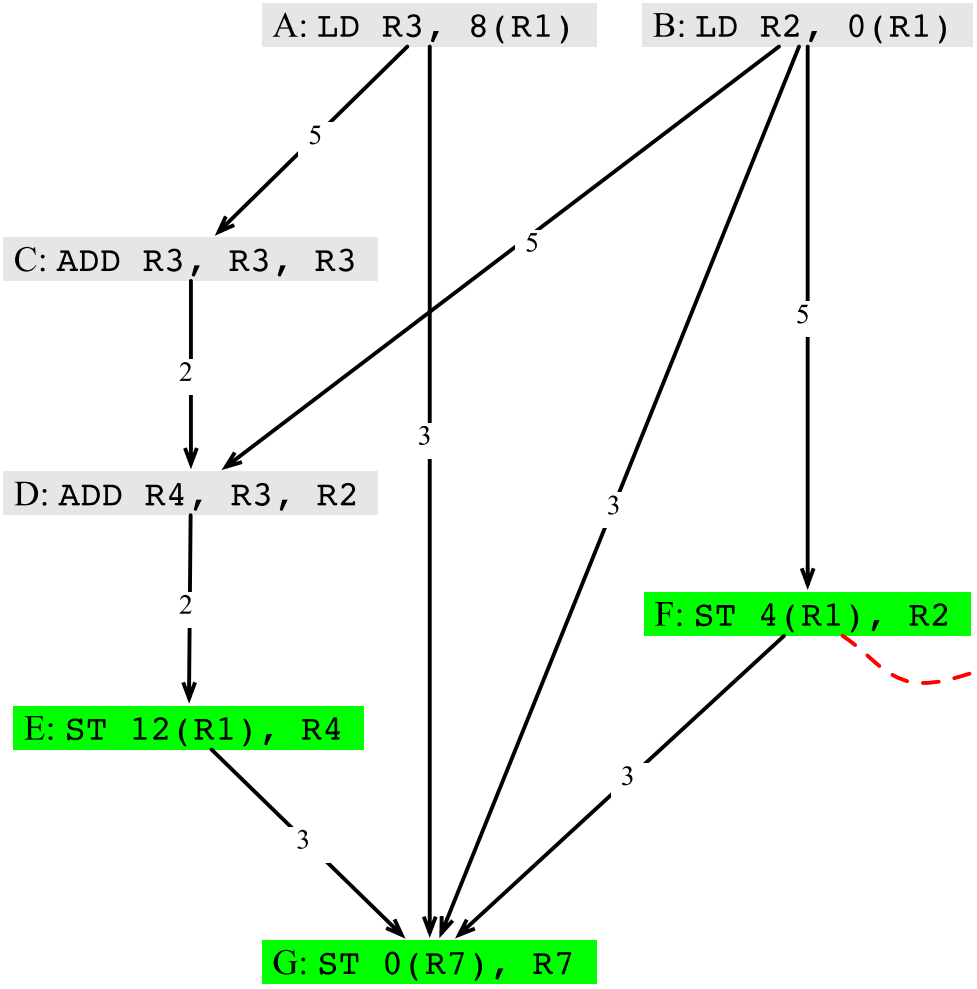
$s = MAX \{S(p) + d_e \mid e = p \rightarrow n \in E\}$

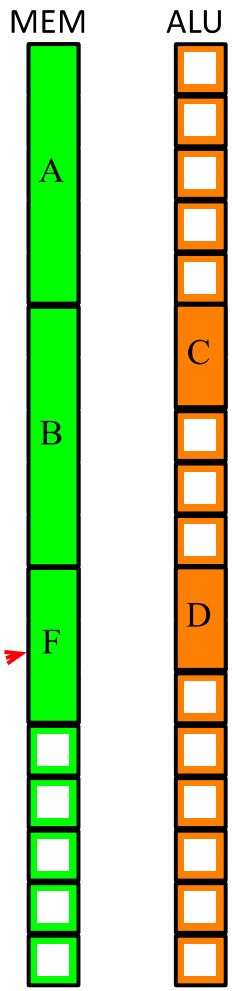
let $i = MIN \{x \mid RT(n, j, x) \in RT\}$ **in**

$s' = s + MAX(0, i - s)$

$RT(n, j, s') = v_n$

Basic Block Scheduling – Example (5)

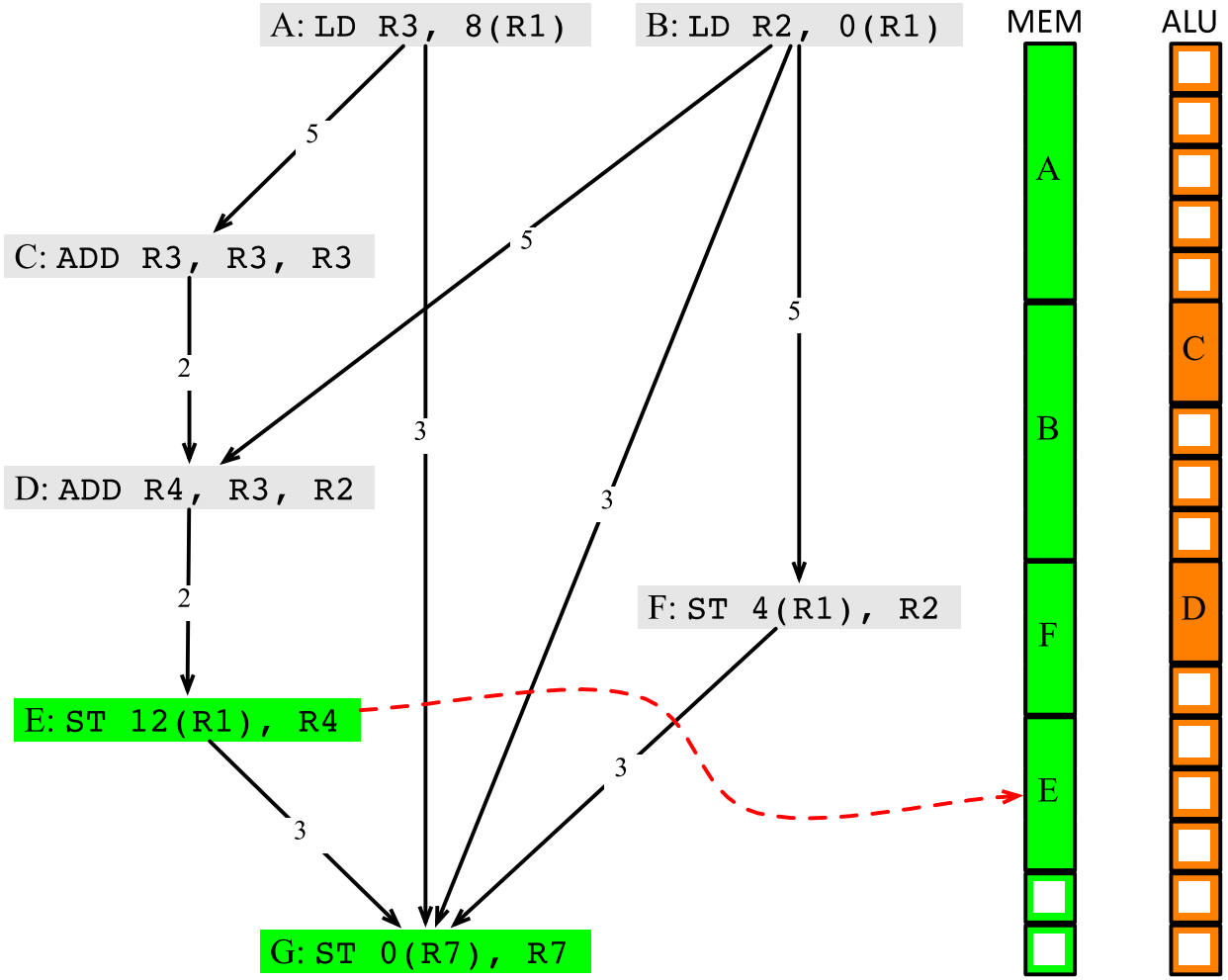




The topological ordering gives us two choices now. Either we grab E or F. If we choose E, then we will have to put up with a larger s, as it must wait for D to finish over. Thus, we get F, which does not depend on any other instruction.

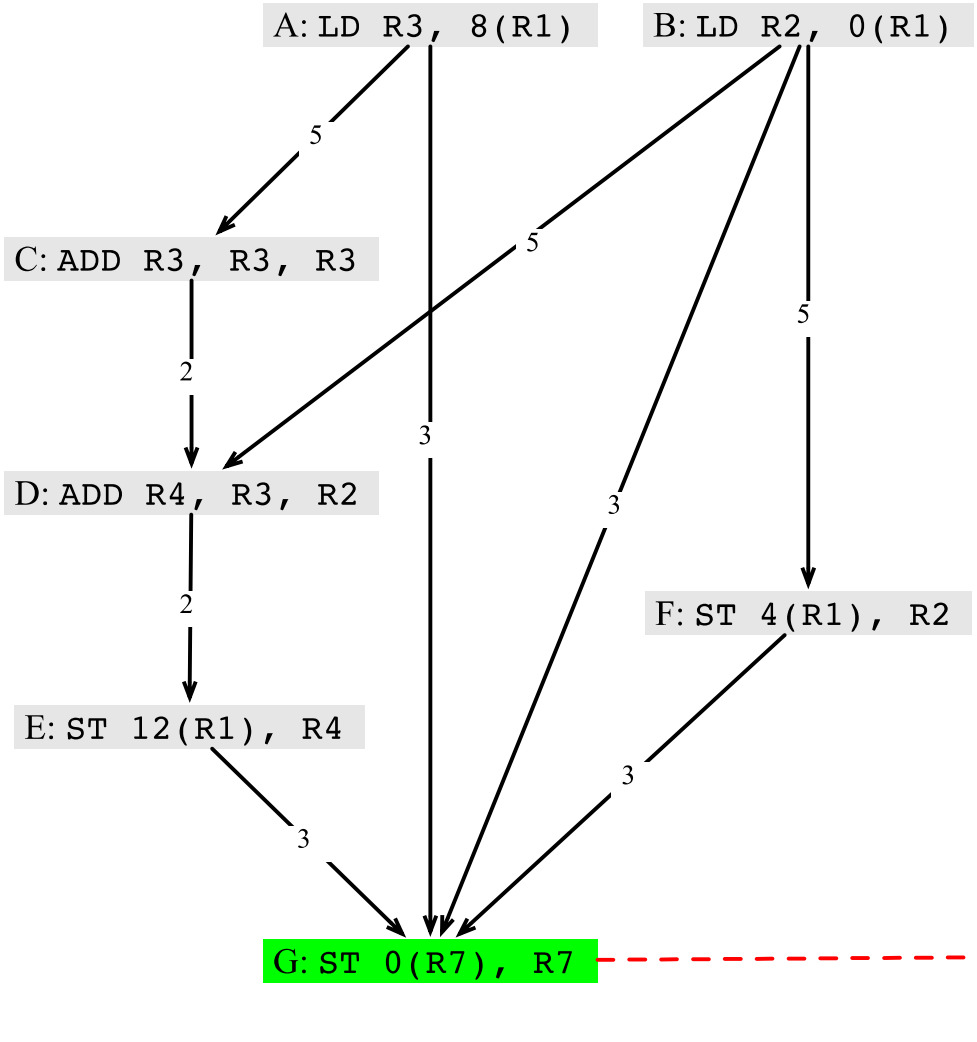
Can you make sure that you understand that picking F is better than picking E at this point?

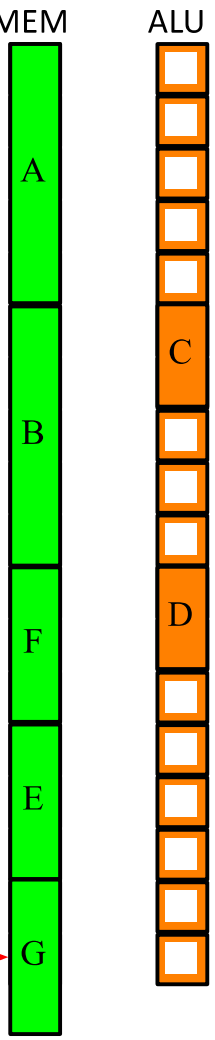
Basic Block Scheduling – Example (6)



The topological order now gives us only one choice: we must pick E, for G depends on E. All the dependences of E have already being solved at the earliest available spot that we have, which is given by the resource allocation table.

Basic Block Scheduling – Example (7)





And now we are done: we have scheduled the last instruction in the basic block.

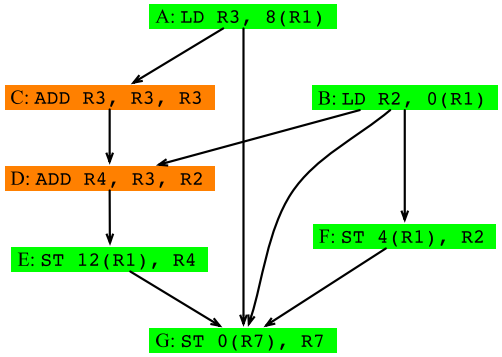
1) Why a topological ordering of the dependence graph is always guaranteed to exist?

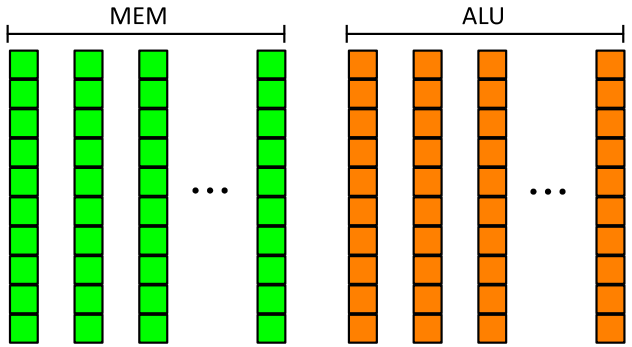
2) What is the complexity of this algorithm?

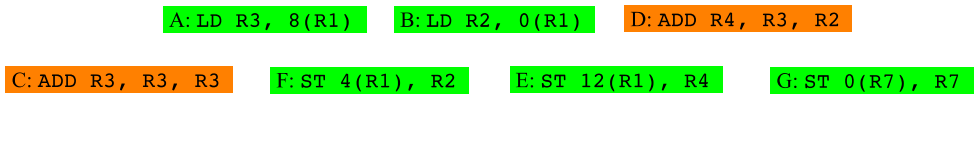
3) Is it optimal? Actually, what does it mean to be optimal in this case?

Complexity Results

1) If we had an infinite surplus of every resource, what would be an optimal solution to basic block scheduling?





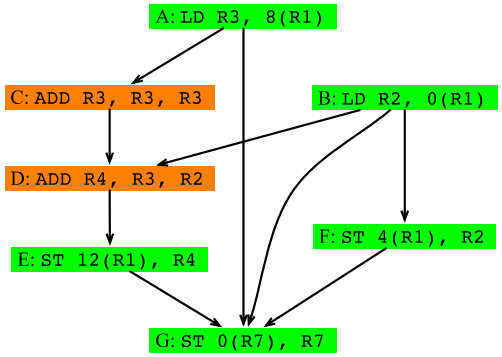


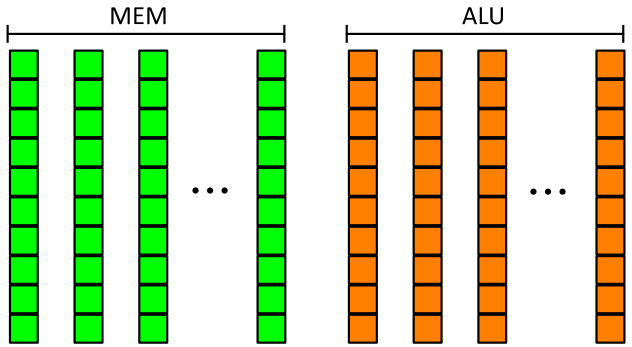
2) If all the instructions were independent, what would be an optimal solution to the problem?

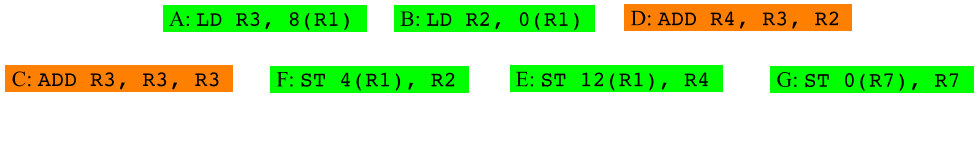
3) And, after all, is there an optimal solution to the general scheduling problem in basic blocks?

Complexity Results

If we had infinite resources, then an optimal solution is given by the longest path between instructions.







If instructions are not dependent, then the best solution is the sum of waiting times across the available resources.

But in general this problem is NP-complete, because we can reduce the scheduling problem to it. Scheduling has been proven to be NP-complete by **Richard Karp** in one of the most celebrated computer science papers ever written[◇].

◇: Reducibility Among Combinatorial Problems, 1972

Compilers Laboratory

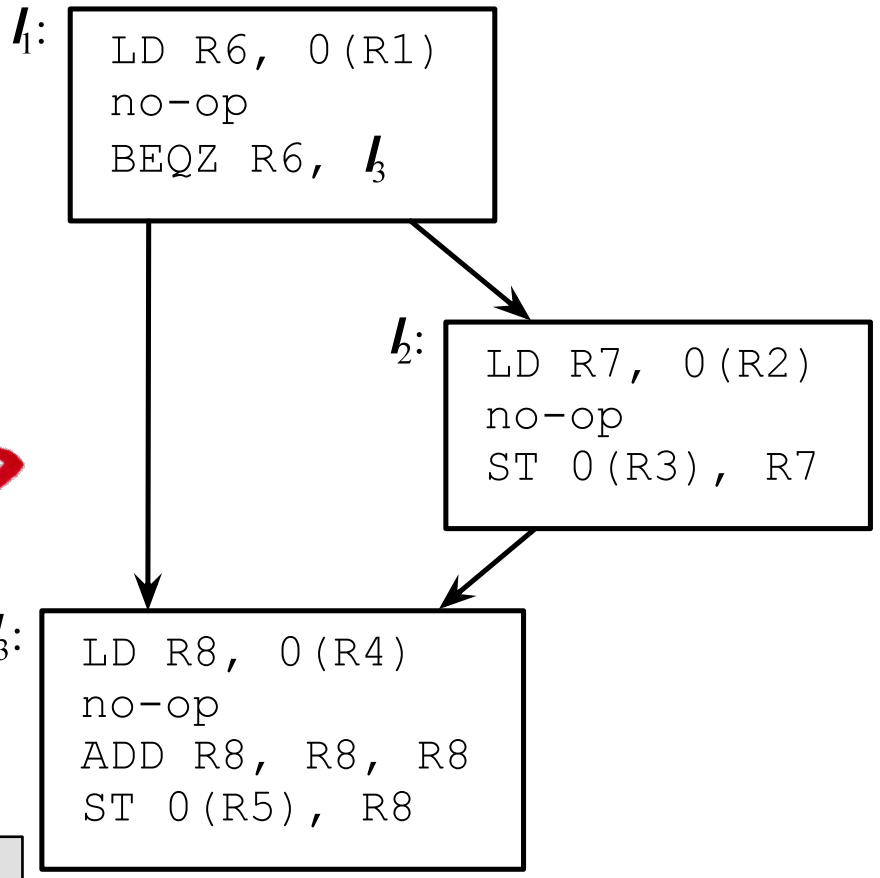
Department of Computer Science
Universidade Federal de Minas Gerais
Federal University of Minas Gerais, Brazil

Global Code Scheduling

DCC 888

Multiple Basic Blocks

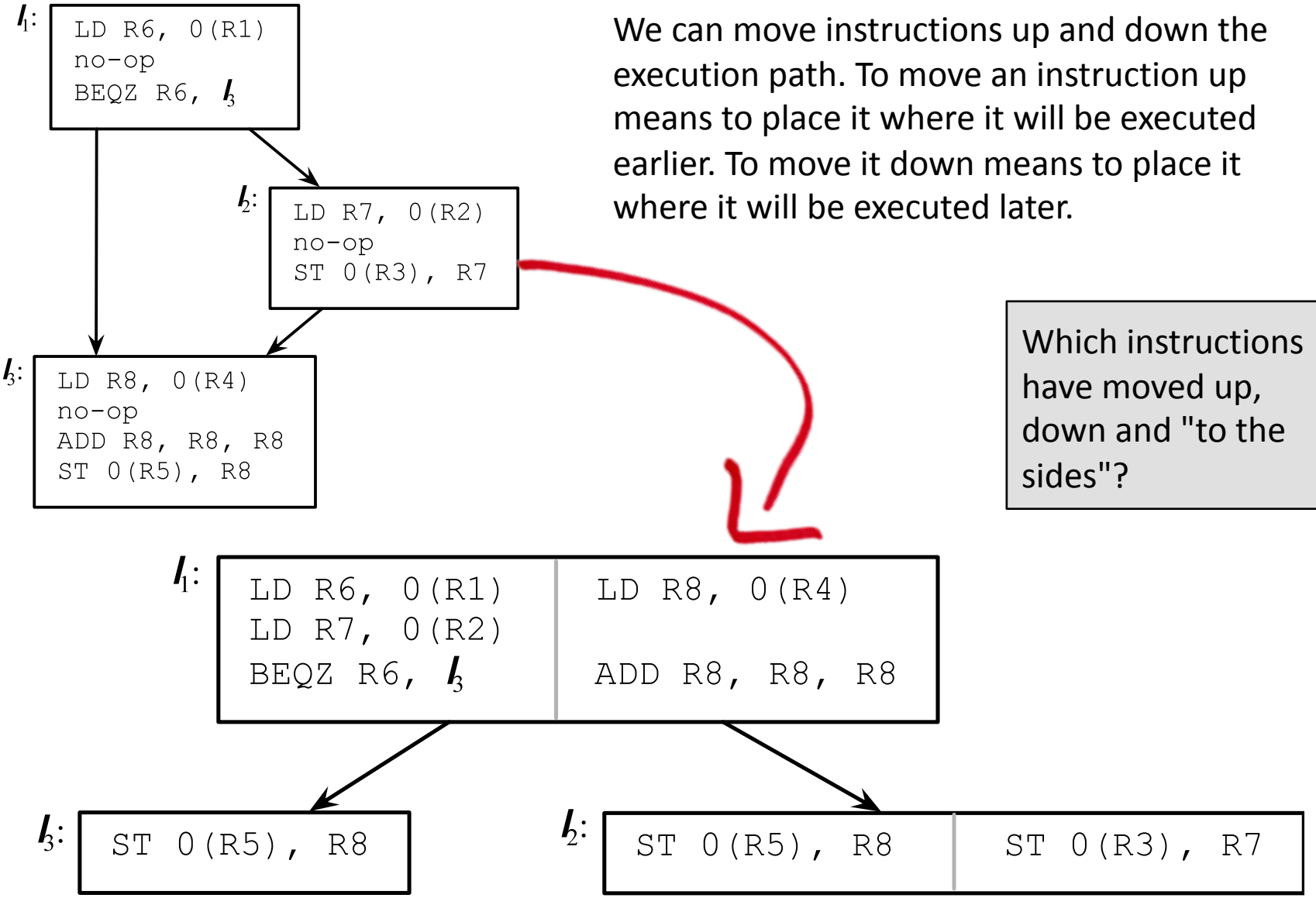
```
if (a != 0) {
    c = b
}
e = d + d
```



Imagine a machine that runs any two instructions in parallel. Which instructions in our target code could we parallelize inside a block?

What if we can move instructions across basic blocks?

Globally Scheduled Machine Code

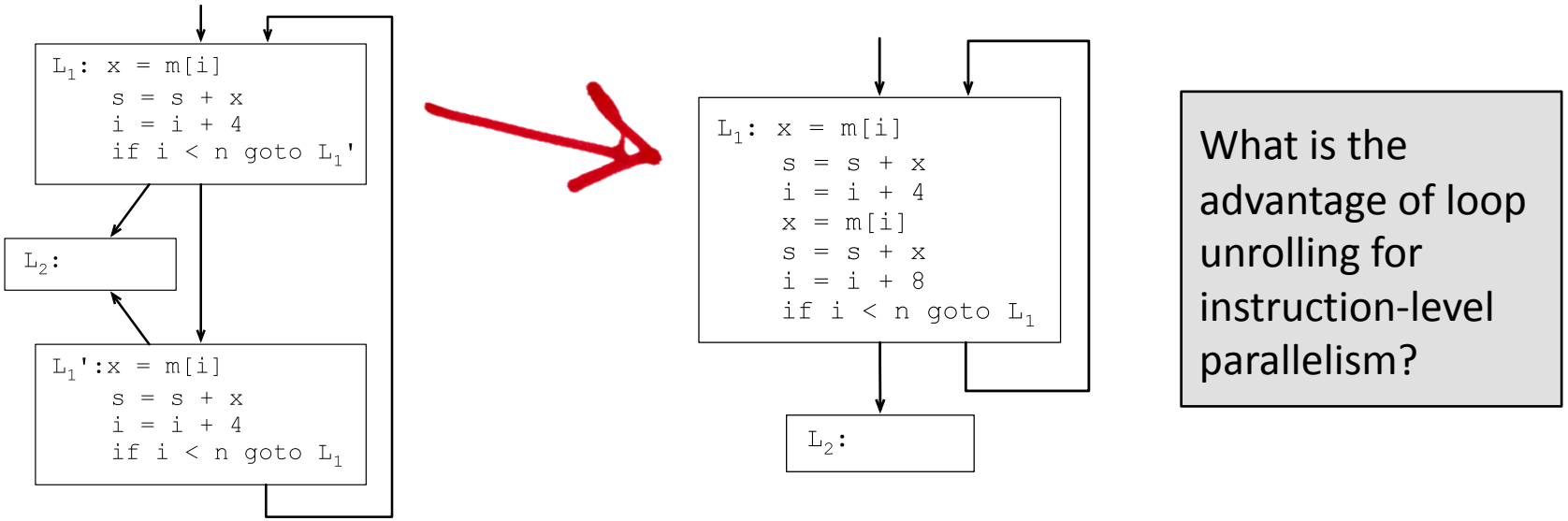


We can move instructions up and down the execution path. To move an instruction up means to place it where it will be executed earlier. To move it down means to place it where it will be executed later.

Which instructions have moved up, down and "to the sides"?

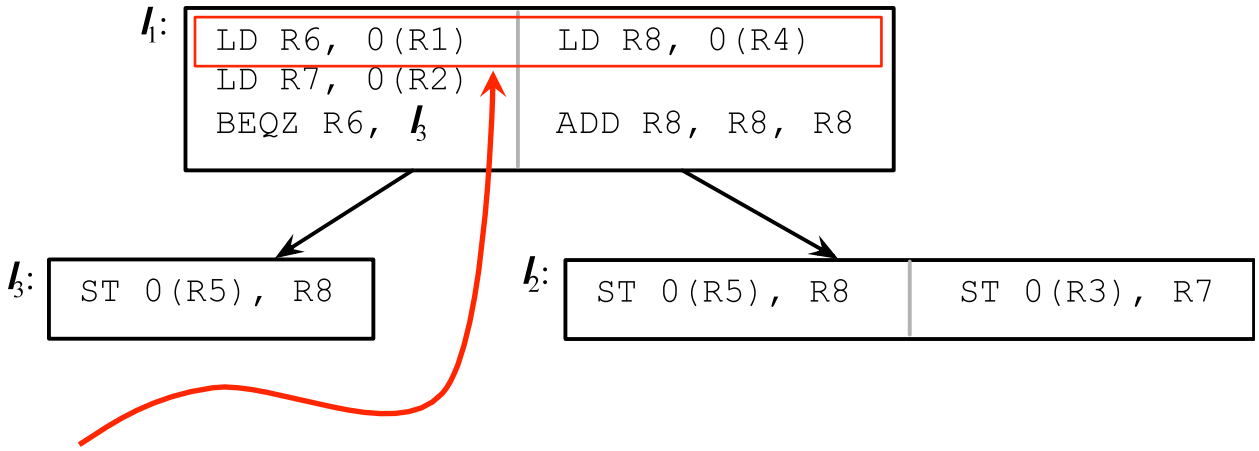
Issues with Global Scheduling

- Substantially more complex than basic block scheduling.
  - Code replication
  - Unsafe (performance-wise) decisions
  - Speculation
- Sometimes compilers can unroll loops, to use basic block scheduling across larger instruction sequences.



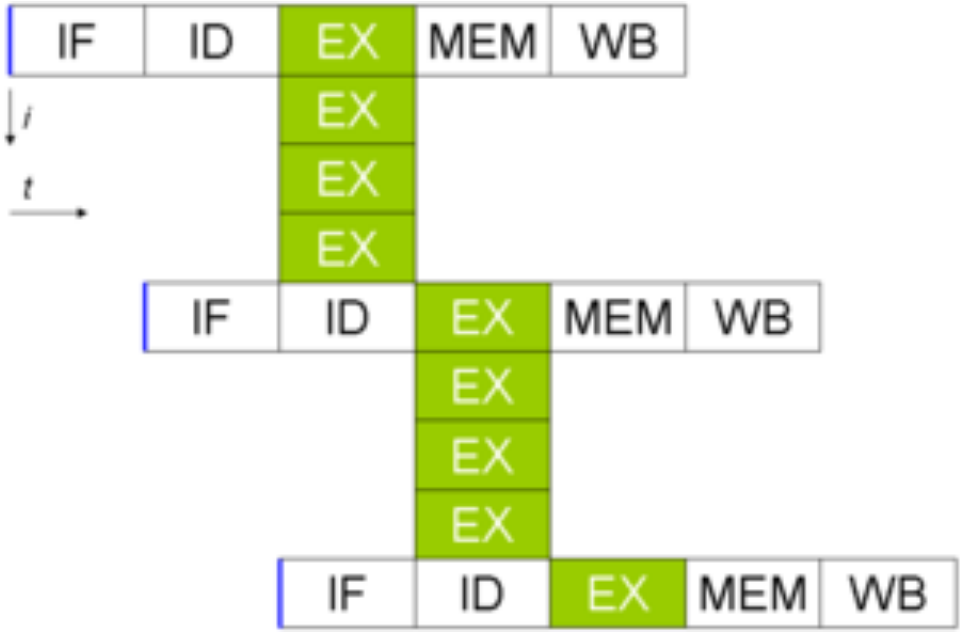
What is the advantage of loop unrolling for instruction-level parallelism?

VLIW Architectures



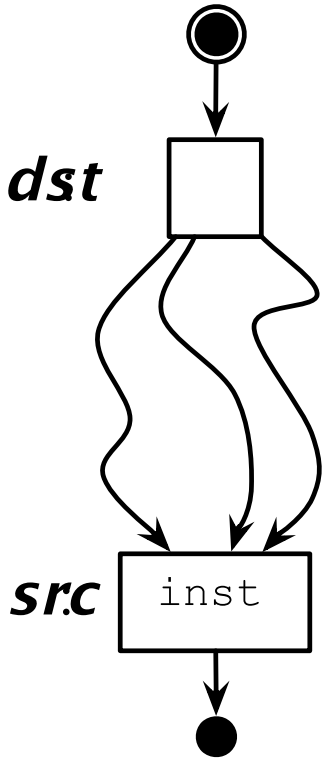
What is a VLIW machine?

Here we are putting instructions next to each other. This kind of execution is a reality in VLIW (Very Long Instruction Word) machines.

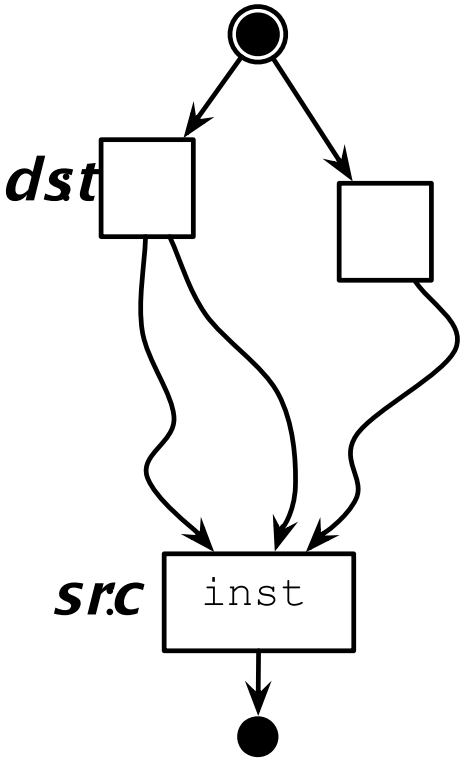


Upward Code Motion

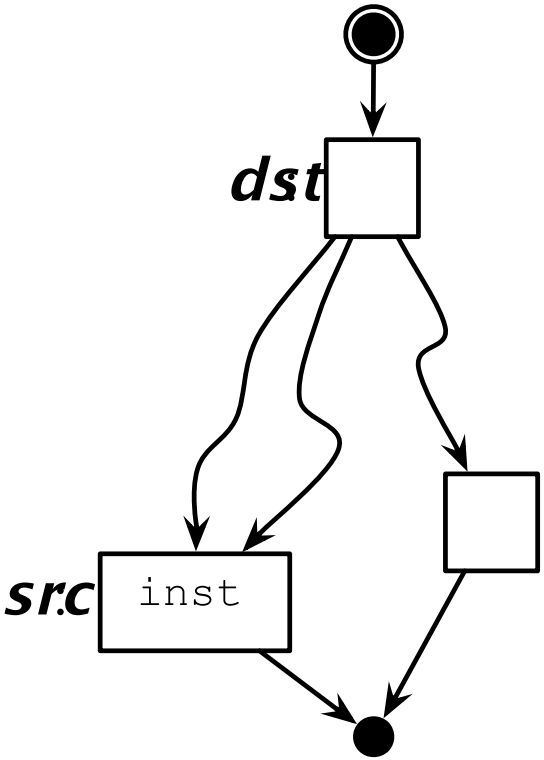
When performing upward code motion, there are a few scenarios that we must consider:



src postdominates dst
dst dominates src

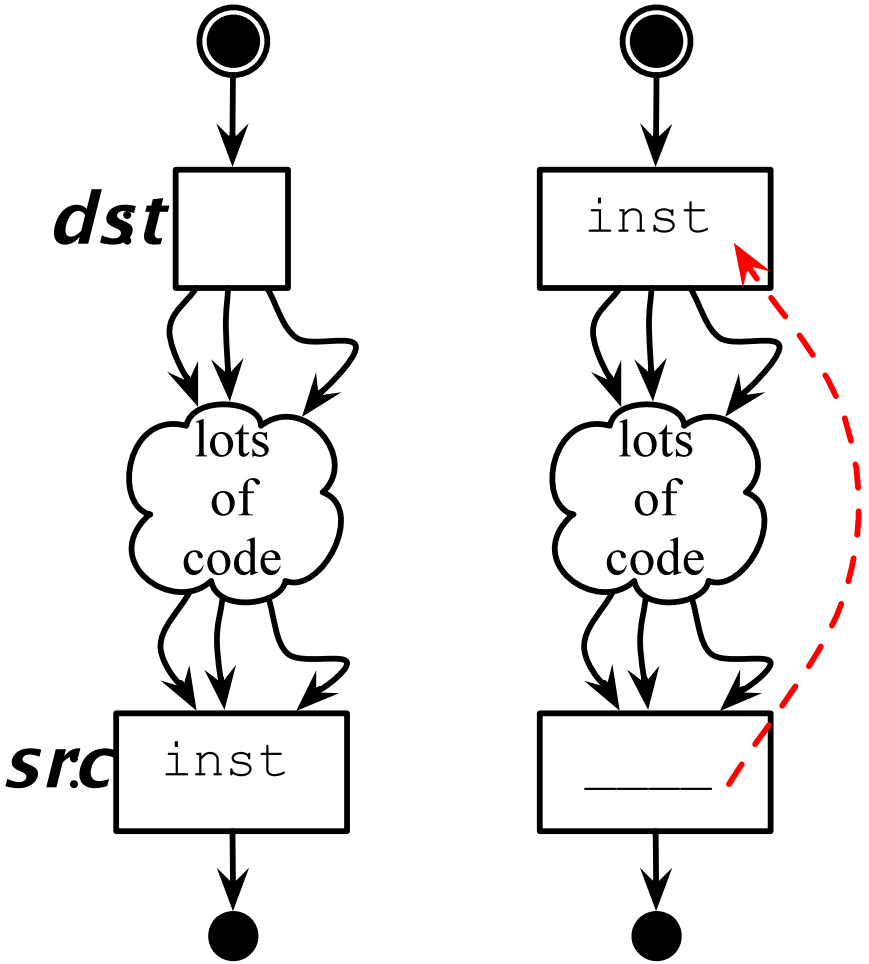


dst does not dominate src



src does not
postdominates dst

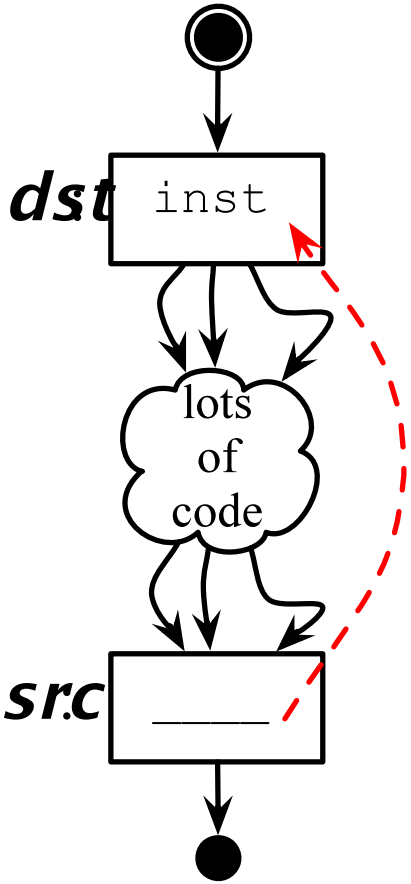
Control Equivalence



If dst dominates src, and src post-dominates dst, then we say that these two blocks are *control-equivalent*. In other words, src will be always executed, whenever dst is executed. Moving instructions up in this case is easier, but we still must observe a few conditions.

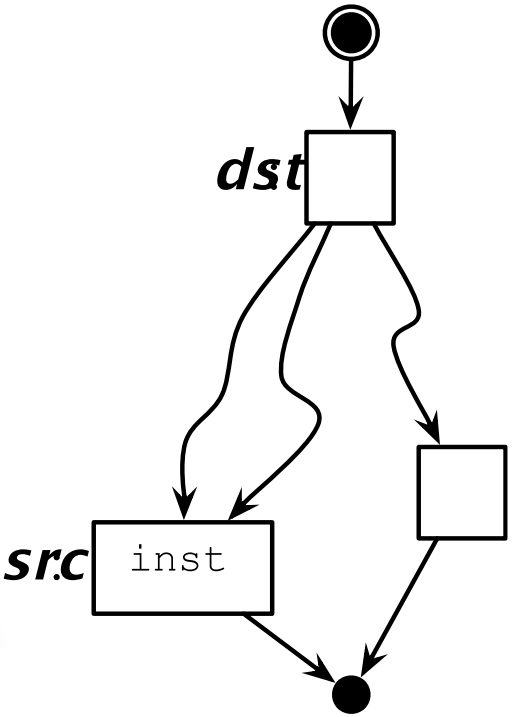
Which conditions guarantee that it is safe to move inst from src to dst?

Control Equivalence

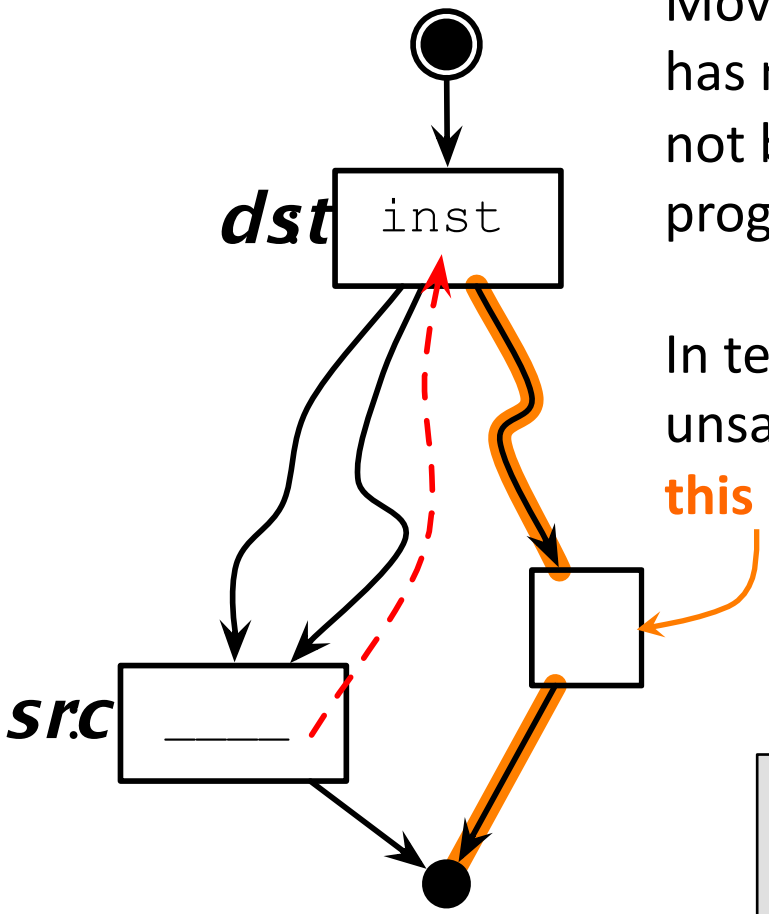


In order to move `inst` up, from src to dst, the only thing that we must ensure is that this change does not break any dependencies. In other words, we must guarantee that there is no code inside "lots of code" that creates dependences with inst.

What about when src does not post-dominate dst? When, and how, can we move inst from src to dst?



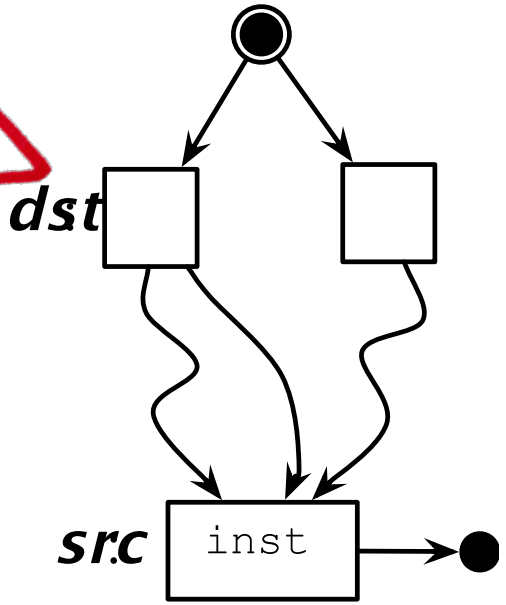
Risky Moves



Moving inst from src to dst is safe only if inst has no side-effects. As before, this change must not break dependence relations within the program.

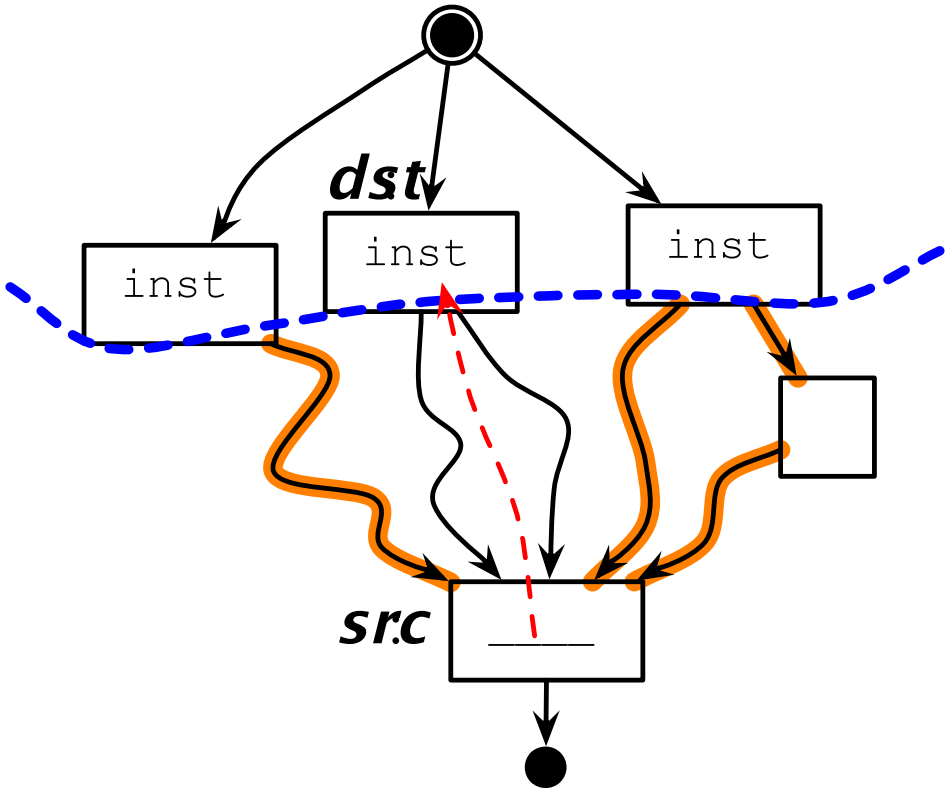
In terms of performance, this change can be unsafe, i.e., it can make the program slower, if **this path** is taken too often.

What if dst does not dominate src? When, and how, can we move inst from src to dst?



Compensation Code

Moving instructions up in case dst does not dominate src is a bit more complicated, because it requires compensation code. We must guarantee that inst is executed independent on the path taken. We do this by inserting copies of inst at every basic block that is part of a **cut set** of the control flow graph. And again, this change my be performance unsafe, if we end up executing more instructions than before. In other words, if the **orange paths** are taken too often, we end up executing inst unnecessarily too many times.

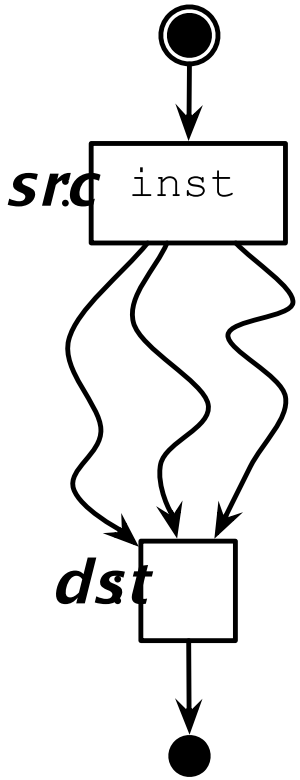


How can we know if **these paths** are taken too often?

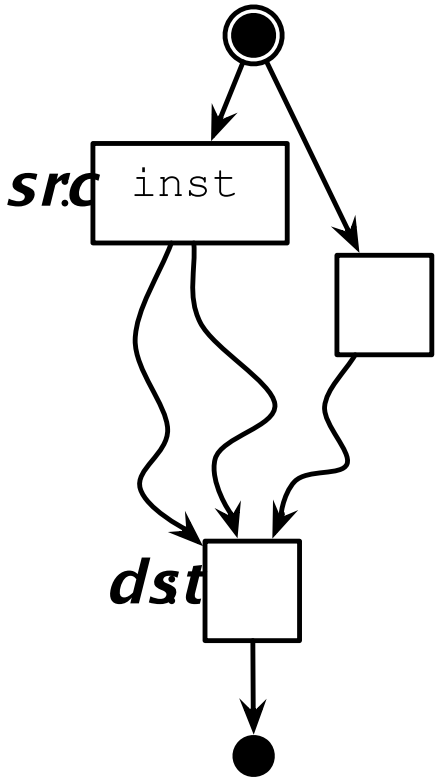
Downward Code Motion

We might also be willing to move instructions downward, i.e., to a program point that will be executed later on.

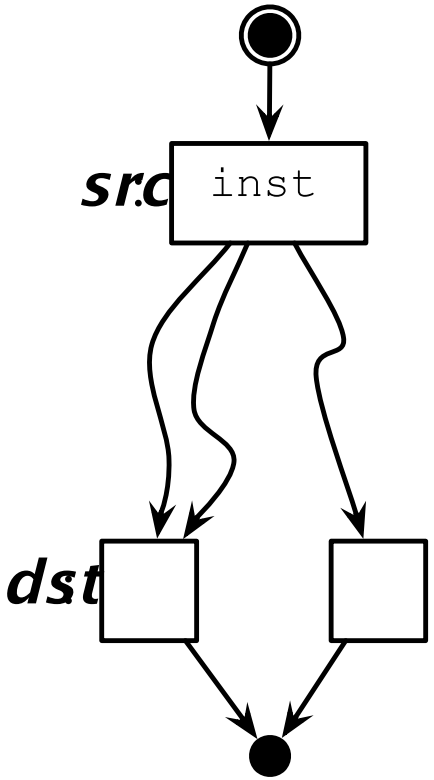
Can you point out the implications of downward code motion in each of the cases below?



src postdominates dst
dst dominates src

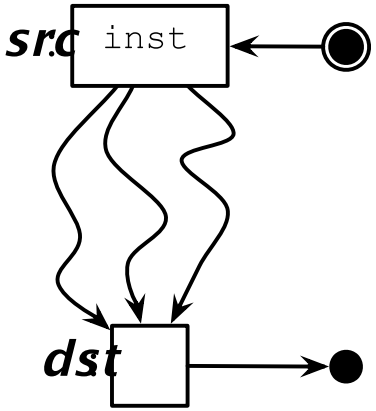


src does not dominate dst

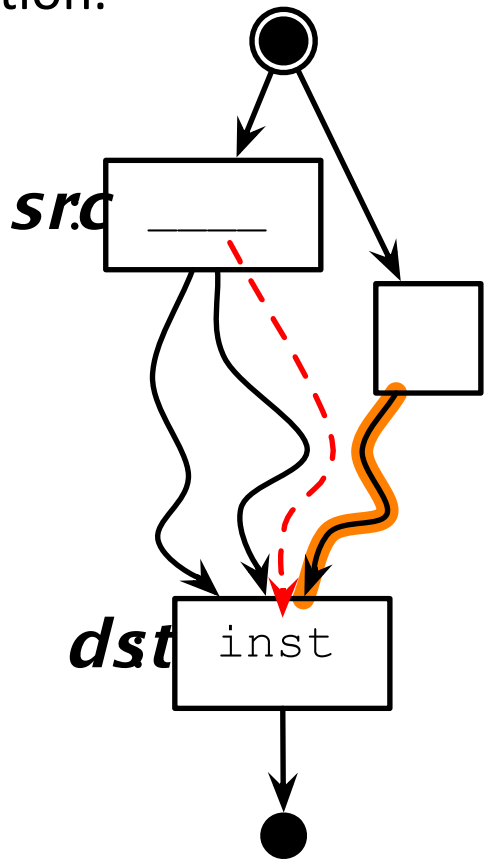


dst does not
postdominates src

Overwriting



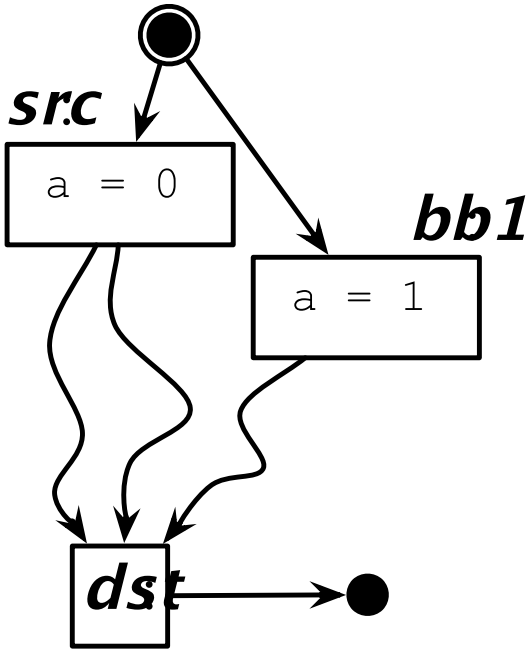
If src and dst are control equivalent, then we only have to ensure that we are not breaking dependences. This is the easy case, just like when we saw upward code motion.

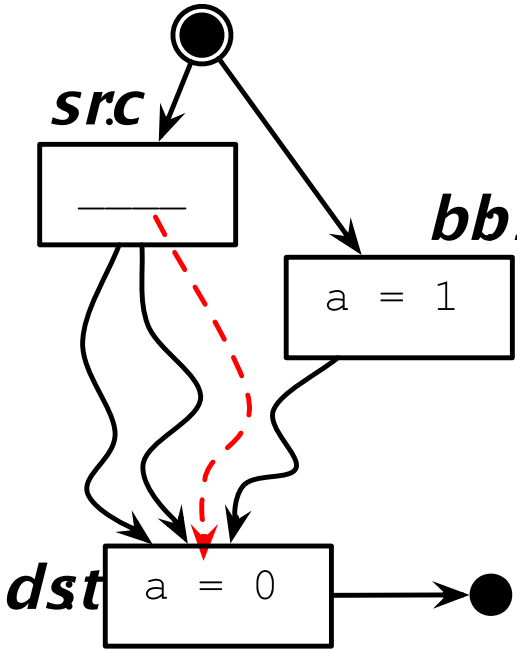


If src does not dominate dst, then we are back on the problem of speculative code execution. We may lose performance if the **orange path** is taken too often. Additionally, we have a second problem, which did not existed in the upward case: overwriting. Inst might write a location that is written also along the **orange path**.

How can we deal with overwriting?

Overwriting



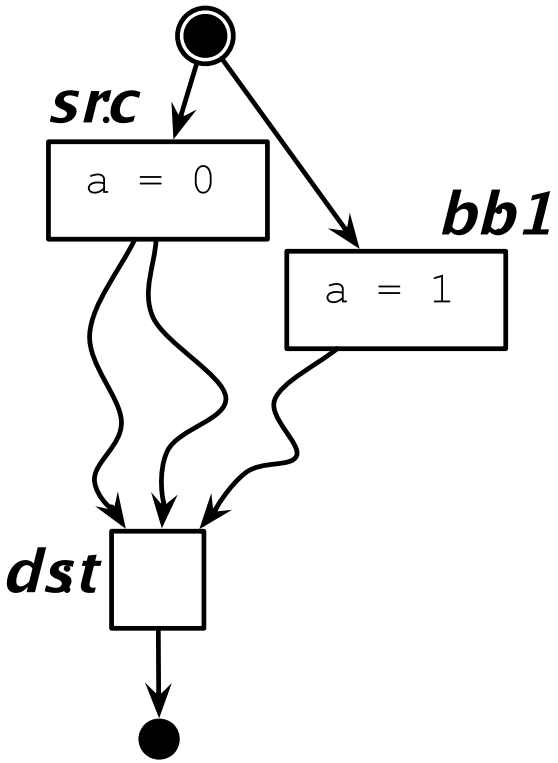


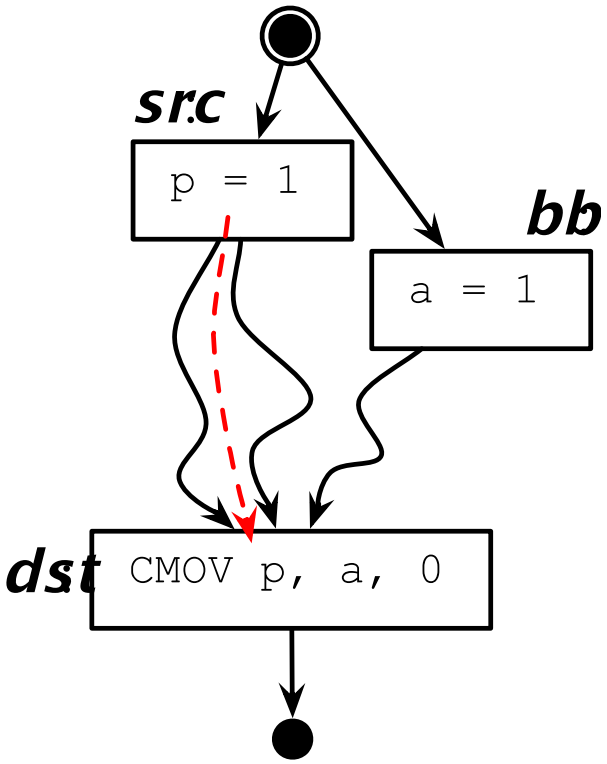
1) Did we have this problem in the case of upward code motion?

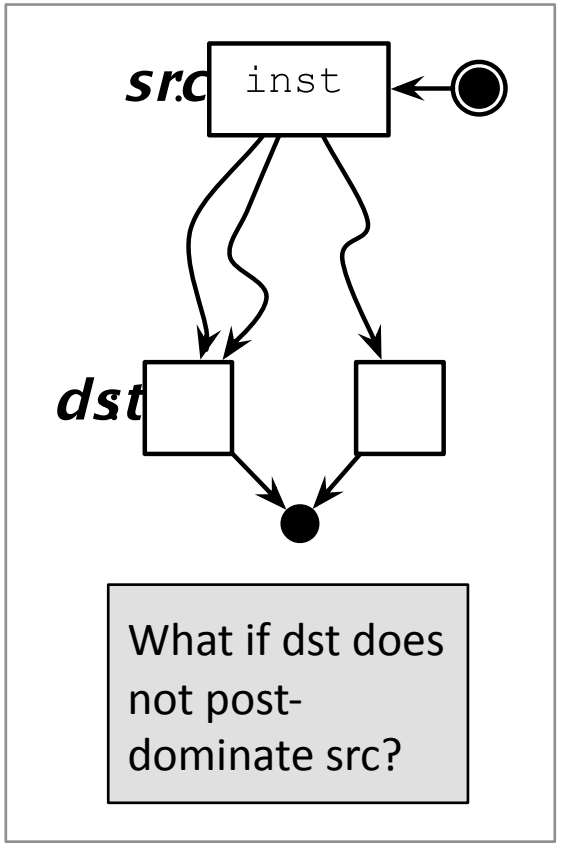
2) How can we deal with this problem, assuming that we still want to move the instruction down?

This example illustrates the problem of overwriting: if we move `a = 0` from src to dst, then we will overwrite the value of `a` if the program flow had gone through bb1. We are, in fact, introducing an output dependence in the code.

Predication



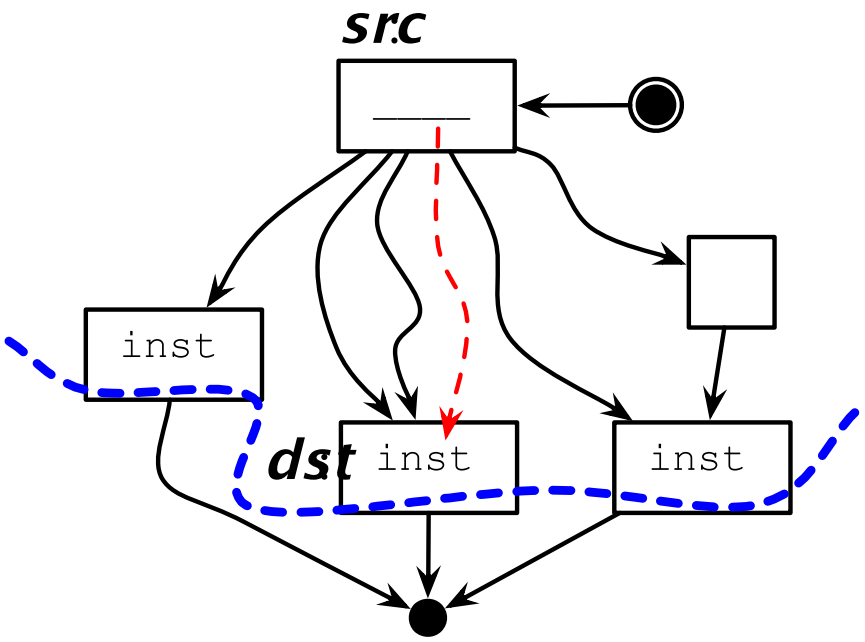




Many computer architectures have something called predicated instructions. These instructions receive an extra parameter, which is treated as a boolean. If it is true, then the instruction executes, otherwise, it does not perform any change of state in the machine. In our example `CMOV p,a,0` will move 0 to `a` if, and only if, `p` is true.

Min Cut (Again)

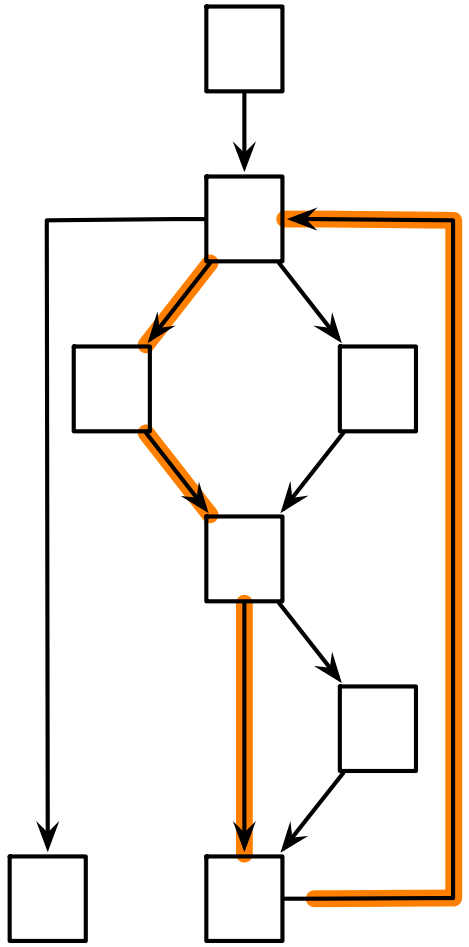
Like in the case of upward code motion, in downward code motion we may have to insert inst in every block of a cut set of the control flow graph. There are many **cut sets**, of course, but we want to find a *more profitable* one. A cut set is more profitable if it is formed by basic blocks that do not execute too often. In this way, we minimize the impact of having to insert multiple copies of an instruction in the program's control flow graph.



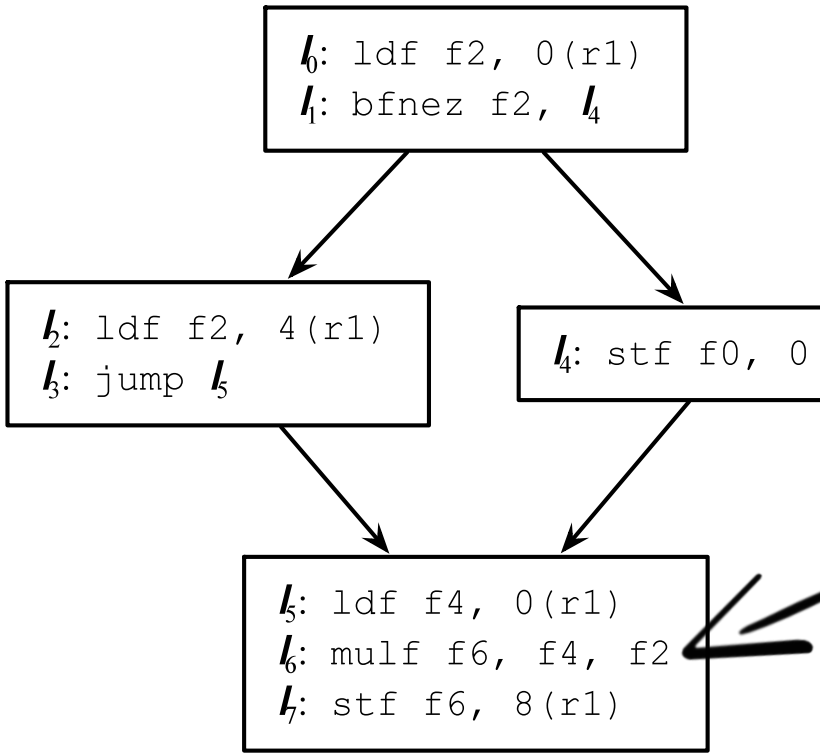
How can we find a profitable cut set?

Super Block Formation

- As we have seen, branches give us a hard time to move instructions around.
- A way to circumvent this problem is to build larger basic blocks, by merging smaller blocks that exist in sequence.
  - These larger blocks are usually called *superblocks*.
  - Not so easy: it requires compensation code and speculation.
- A key challenge, as we have mentioned before, is to determine *profitable paths* inside the program.



Superblocks ♤

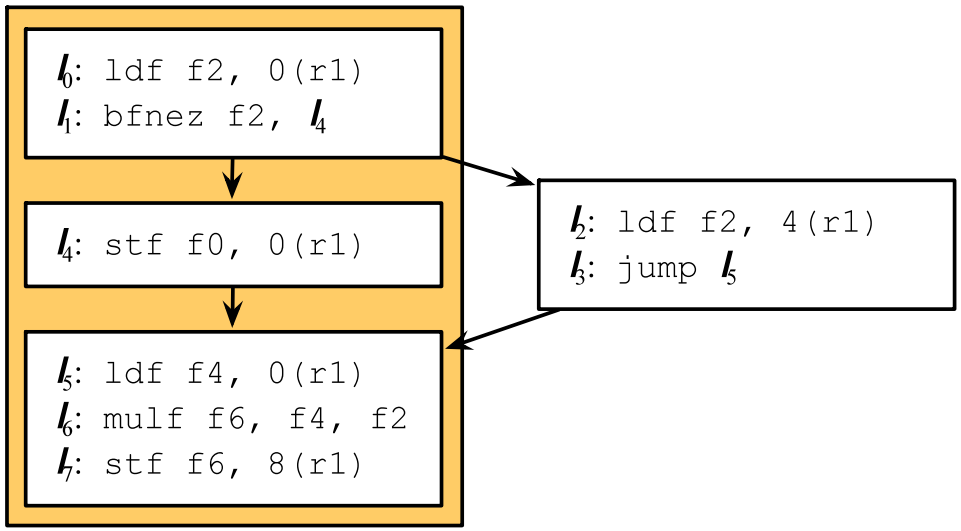


Let's consider the example on the left. Independent on what this program does, if it is scheduled in this way it will suffer an unwanted delay, if we assume that the floating-point multiplication takes three cycles to deliver its result.

What could we do to avoid this delay, assuming that the ℓ_4 branch is taken most of the time?

♤: This example is due to Daniel J. Sorin, from Duke University

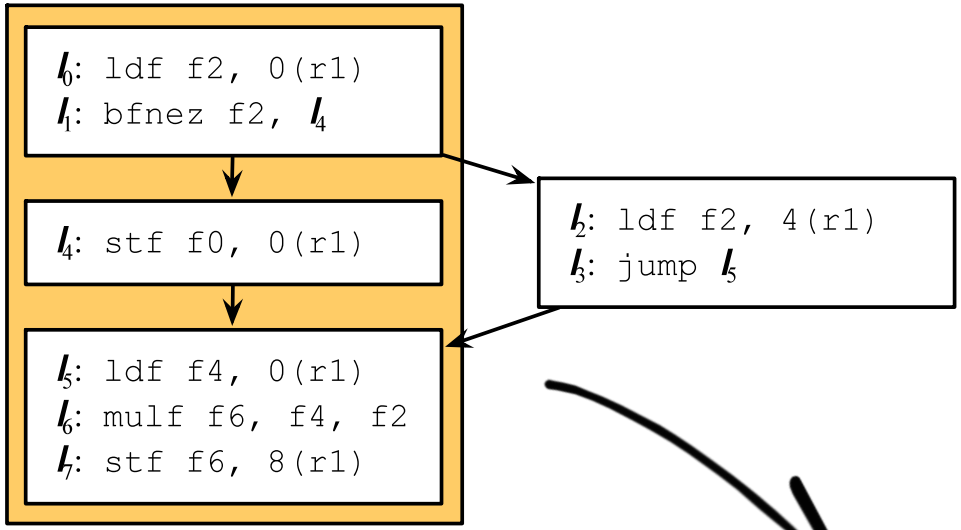
Choosing Superblocks



Ok, there are a few things that we must do to have this program working. How to sequentiallize the instructions?

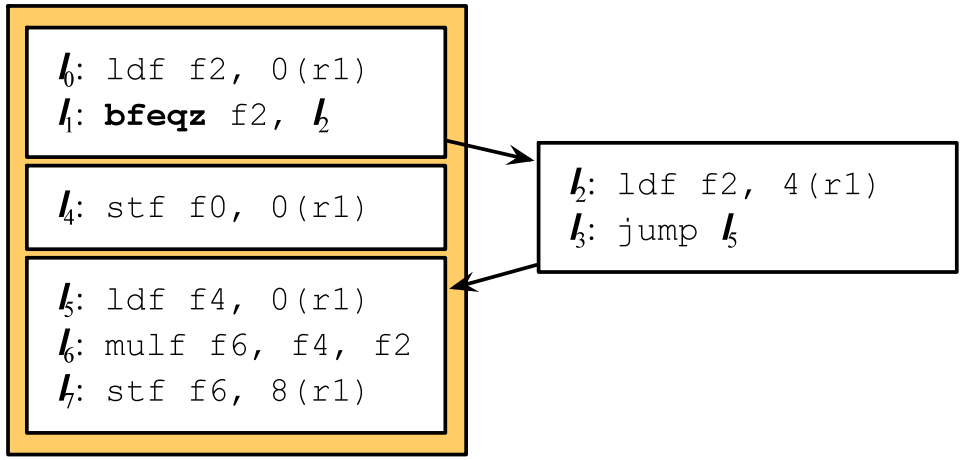
Now, we want to be able to move instructions around, in the marked area. And we want to do this with *freedom*. In other words, we need to change the program a bit, so that even if the program goes through ℓ_2, we are still able to move instructions.

Creating a Superblock



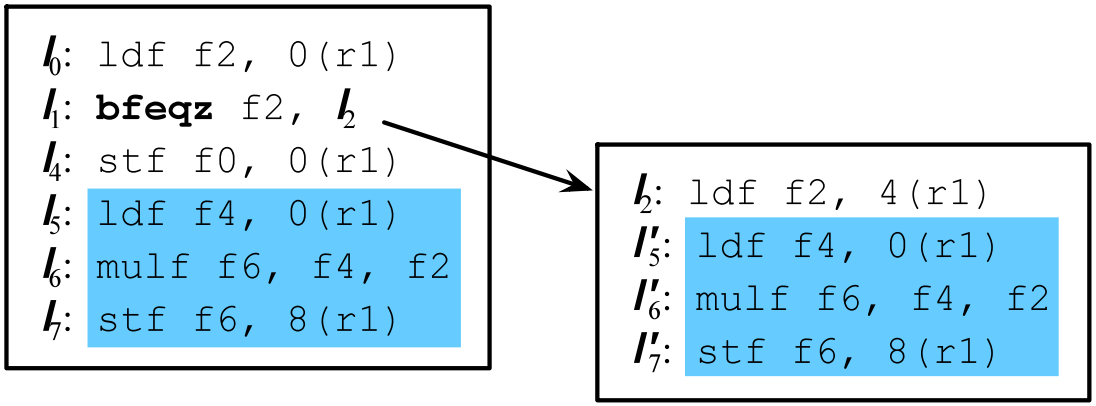
We can start by changing the branch. Before we were jumping to ℓ_4, which is not the next instruction after the new branch. So, we change the branch if not zero to branch if zero, and divert flow to ℓ_2 instead

But we still cannot play with ℓ_5-ℓ_7, because ℓ_3 is still counting on jumping to them. What to do?



Repair Code

We replicate the **code between labels ℓ_5-ℓ_7** (inclusive). In this way, we do not have to worry about overwriting ℓ_2 while doing the multiplication.



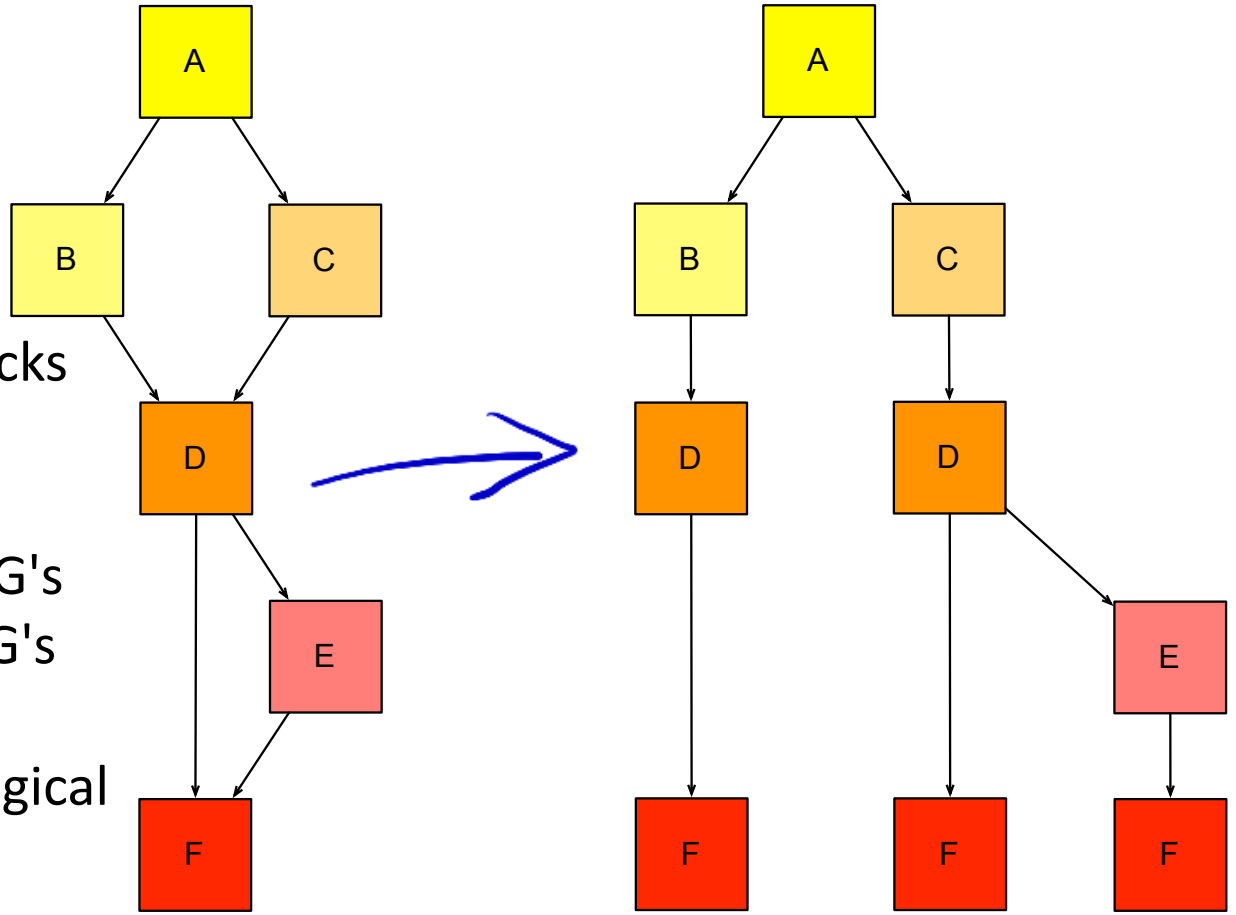
Now we can use our basic block scheduling algorithm to schedule instructions in the superblock.

Yet, one question remains: how do we predict the profitable paths within the CFG?

From DAG's to Trees

CFG's are not DAG's in general. Why are we only talking about DAG's here?

A way to look into this construction of superblocks is to imagine that we are transforming programs whose CFG resemble DAG's into programs whose CFG's resemble trees. In some ways, that is a very ecological program transformation.



How to compute the probability of traversing an edge?

Static Profiling♧

Static Profiling is a way to guess the frequency of use of each basic block in the program. It is based on heuristics that gauge the chance that a branch will be taken.

- **Loop branch heuristic**: edges back to loop heads are taken with 88% of chance.
- **Opcode heuristic**: comparison of an integer for less than zero, less than or equal to zero, or equal to a constant will fail with 84% of chance.
- **Loop exit heuristic**: a comparison in a loop in which no successor is a loop head will not exit the loop with 80% of chance.
- **Loop header heuristic**: a successor that is a loop header and does not post-dominate the block will be reached with 75% of chance.
- **Call heuristic**: A successor that contains a call and does not post-dominate the block will not be taken with 78% of chance.
- **Store heuristic**: a successor that contains a store instruction and does not post-dominate the block will not be taken with 55% of chance.
- **Return heuristic**: a successor that contains a return will not be reached with 72% of chance.

Where do these numbers come from?

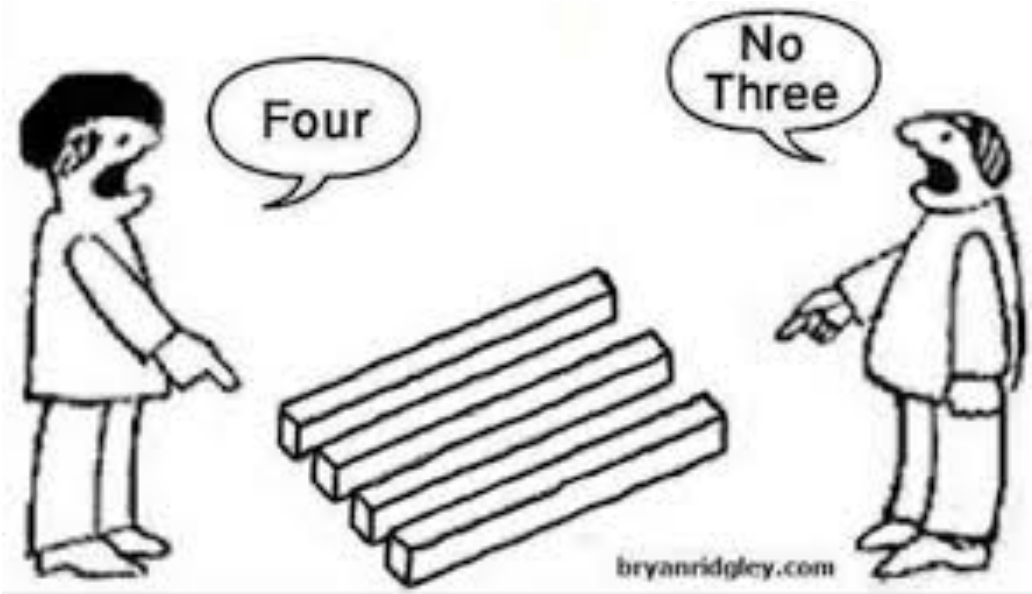
♧: Static Branch Frequency and Program Profile Analysis, MICRO, 1994

Combining Heuristics

- consider the following scenario:
  - A particular heuristic predicts that a given branch takes the true path 70% of the times.
  - Another heuristic predict that the true path is taken 60% of the time.

1) How can you combine these predictions?

2) Do you think the true path is likely to be taken more that 70% of the times, less or just 70%, as before?



Dempster-Shafer

- There are different ways to combine predictions. The original work on static profiling would use the Dempster-Shafer theory to combine them.

$$m_1 \otimes m_2 = \frac{m_1 \times m_2}{m_2 \times m_1 + (1 - m_1) \times (1 - m_2)}$$

In this case, we have that

$m_1 \otimes m_2 =$

$0.7 \times 0.6 / (0.7 \times 0.6 + 0.3 \times 0.4)$

$m_1 \otimes m_2 =$

0.778

Prof. Arthur Dempster

A Bit of History

- Most of the discussion in this section has been taken from Hennessy and Patterson's book.
- The concept of data dependence is old. Early discussions appear in a work due to Kuck *et al*.. They were compiling code for vector machines.
- The algorithm that we use for instruction scheduling is due to Bernstein and Rodeh.

- Hennessy, J. L. and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Third Edition, 2003.
- Kuck, D., Y. Muraoka, and S. Chen, *On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup*. IEEE Transactions on Computers, pp. 1293-1310, 1972.
- Bernstein, D. and M. Rodeh, *Global instruction scheduling for super-scalar machines*, PLDI, pp. 241-255, 1991.