



《编译原理》

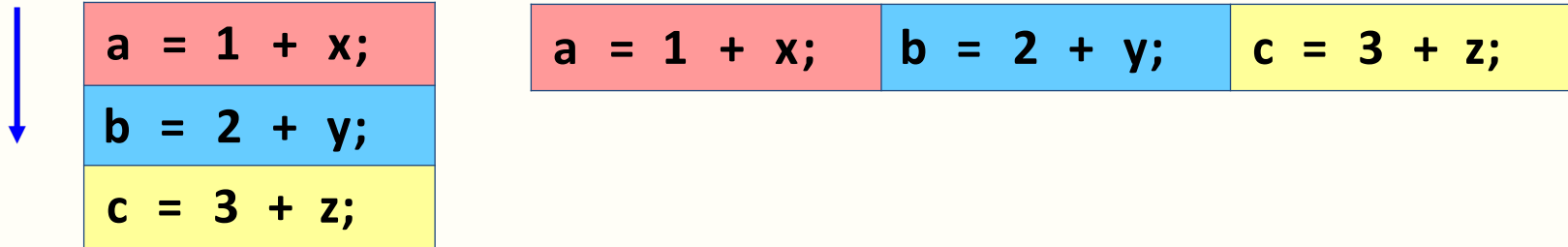
第7讲：指令调度

何冬杰
重庆大学

The Goal of Instruction Scheduling

- ❑ Assume that the **remaining instructions are all essential**
 - otherwise, earlier passes would have eliminated them
- ❑ How can we perform this **fixed amount of work in less time**?
 - Answer: *execute the instructions in parallel*

Time

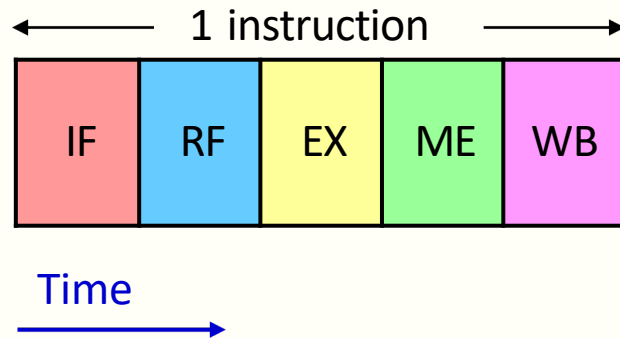


- ❑ Hardware Support for Parallel Execution
 - Pipelining
 - Superscalar Processing
 - Multicore: **Automatic Parallelization**
- } **Instruction Scheduling**

Pipelining

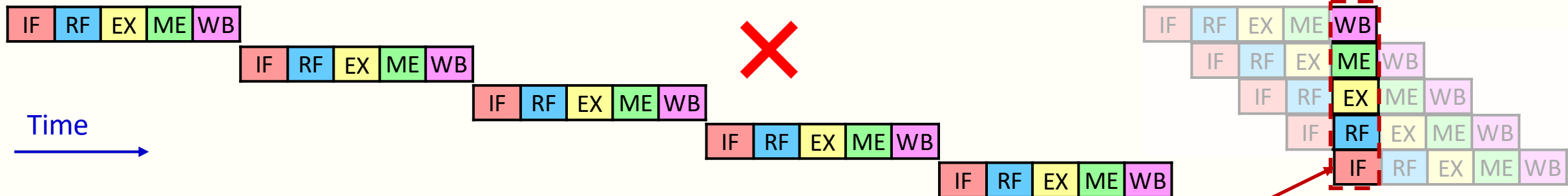
❑ Basic idea: break instruction into *stages* that can be overlapped

- Example: simple 5-stage pipeline from early RISC machines



IF = Instruction Fetch
RF = Decode & Register Fetch
EX = Execute on ALU
ME = Memory Access
WB = Write Back to Register File

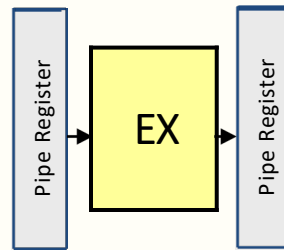
- Pipelining Illustration



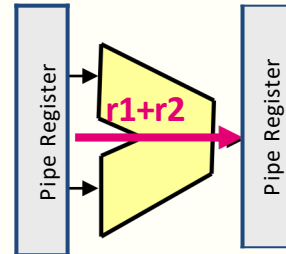
- In a given cycle, each instruction is in a different stage

Superscalar Processing

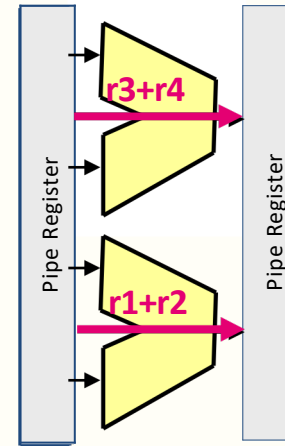
- ❑ Basic Idea: **multiple (independent) instructions** can proceed **simultaneously** through the **same pipeline stages**
- ❑ Requires **additional hardware**
 - example: “Execute” stage



Abstract
Representation



Hardware for
Scalar Pipeline:
1 ALU



Hardware for
2-way Superscalar:
2 ALUs

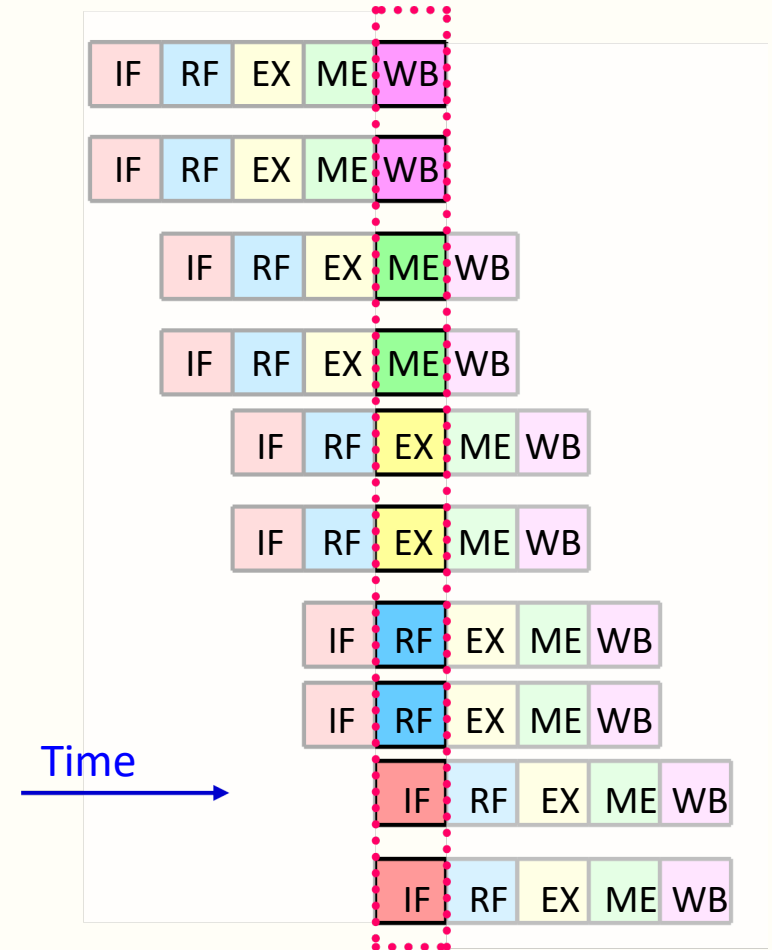
Superscalar Pipeline Illustration

❑ Original (scalar) pipeline:

- Only **one instruction** in a **given pipe stage** at a **given time**

❑ Superscalar pipeline:

- **Multiple instructions** in the **same pipe stage** at the **same time**
- Unlike SIMD/vector instructions, instructions of **different types** can be in the same pipe stage at same time



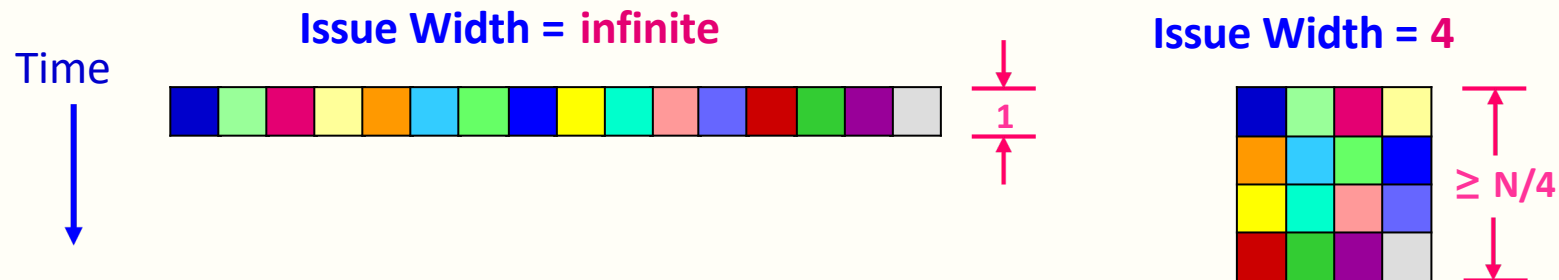
Constraints on Instruction Scheduling

❑ Hardware Resources

- Processors have **finite resources**, and there are often **constraints on how these resources can be used**.

❑ Hardware constraint 1: Finite Issue Width

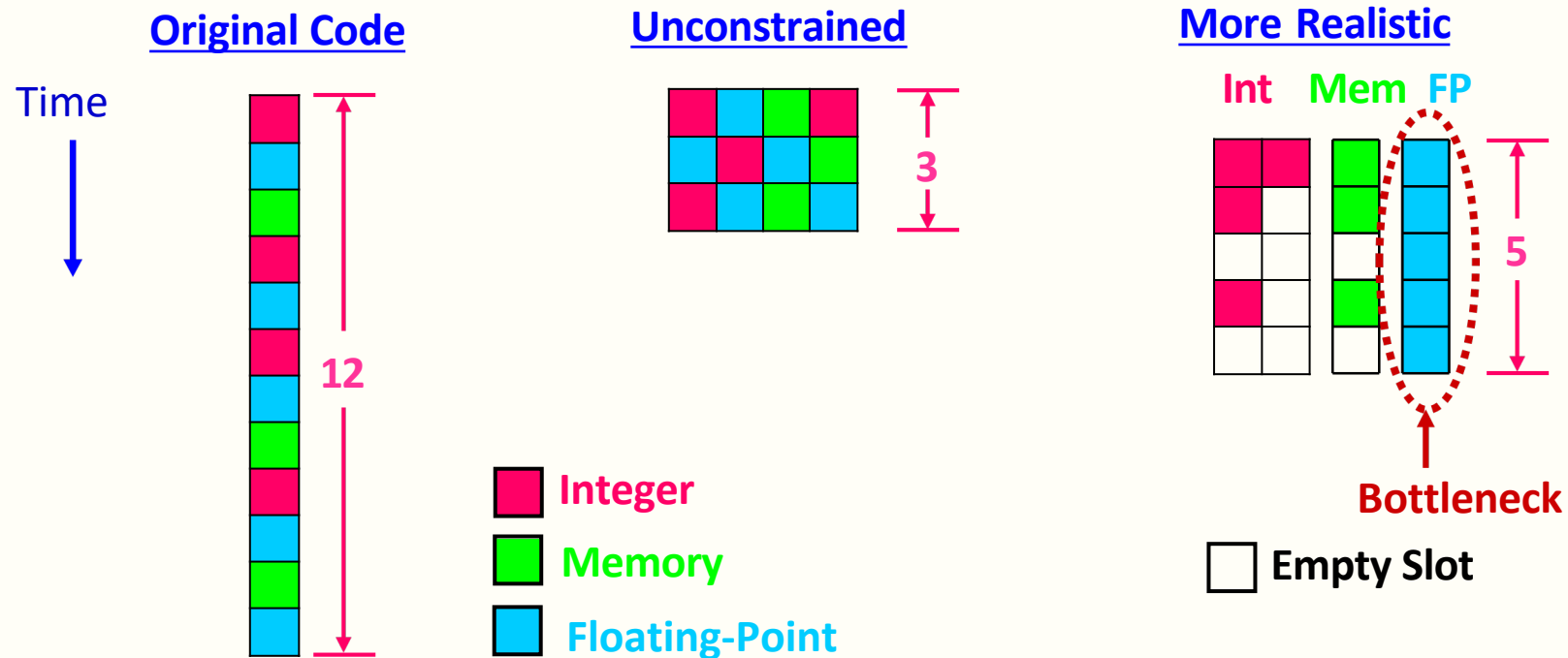
- **Prior to superscalar processing:**
 - processors only “issued” one instruction per cycle
- **Even with superscalar processing:**
 - **limit on total # of instructions issued per cycle**



Constraints on Instruction Scheduling

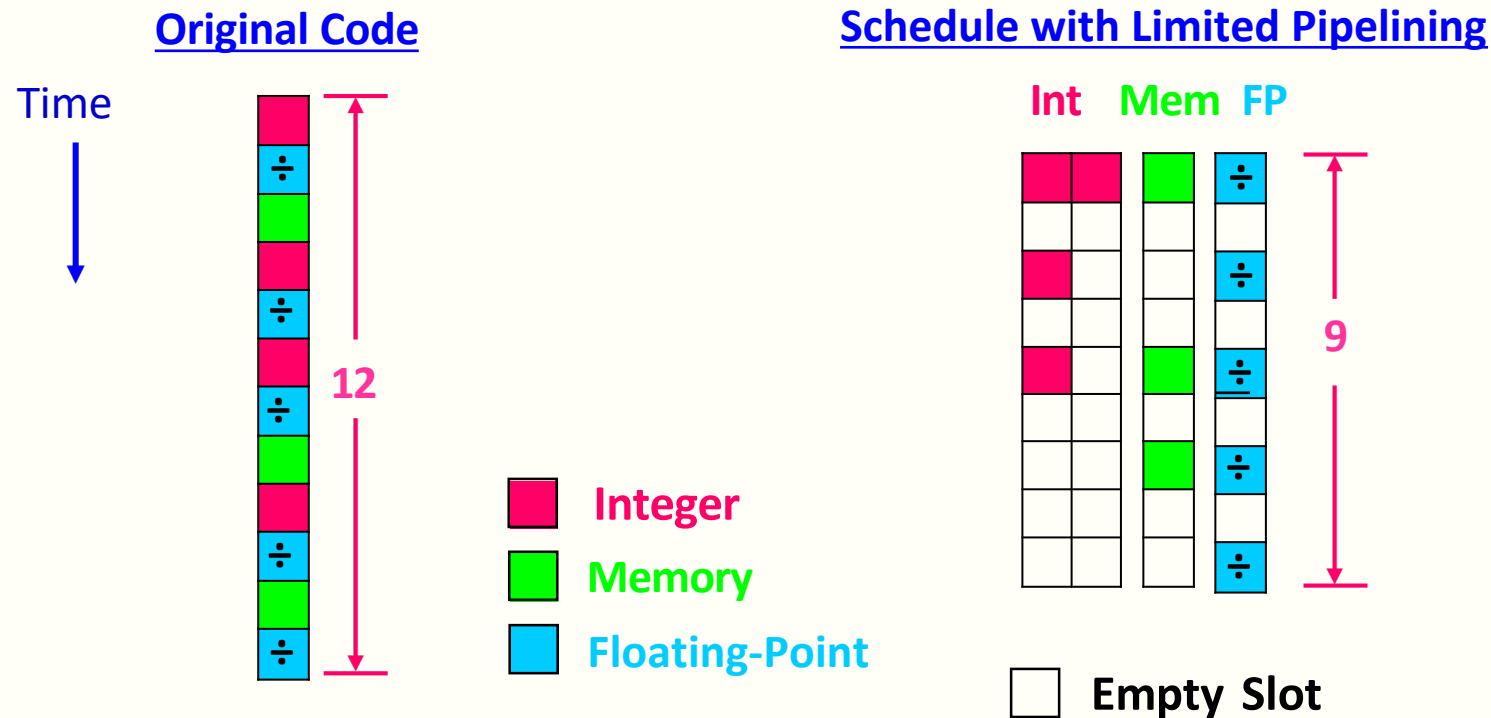
❑ Hardware Constraint 2: Limited FUs per Instruction Type

- e.g., a 4-way superscalar might only be able to issue up to 2 integer, 1 memory, and 1 floating-point instructions per cycle



Constraints on Instruction Scheduling

- ❑ Hardware constraint 3: Limited Pipelining within a Functional Unit
 - e.g., only 1 new floating-point division **once every 2 cycles**



Constraints on Instruction Scheduling

□ Data Dependences

- If we read or write a data location “too early”, the program may behave incorrectly.
- Assume that initially, $x = 0$

??? $x = 1;$
 $y = x;$

Read-after-Write
 (“True” dependence)

Fundamental
(no simple fix)

??? $x = 1;$
 $x_1 = 2;$

Write-after-Write
 (“Output” dependence)

??? $y = x;$
 $x_1 = 1;$

Write-after-Read
 (“Anti” dependence)

Can potentially fix through *renaming*.

Why Data Dependences are Challenging

❑ Which of these instructions can be reordered?

```
x = a[i];  
*p = 1;  
y = *q;  
*r = z;
```

❑ Ambiguous data dependences are very common in practice

- difficult to resolve, despite fancy *pointer analysis*

❑ Given Ambiguous Data Dependences, What To Do?

- Conservative approach: don't reorder instructions
 - ensures correct execution
 - but may suffer poor performance
- Aggressive approach?
 - is there a way to safely reorder instructions?

Hardware Limitations: Multi-cycle Execution Latencies

❑ Simple instructions often “execute” in one cycle

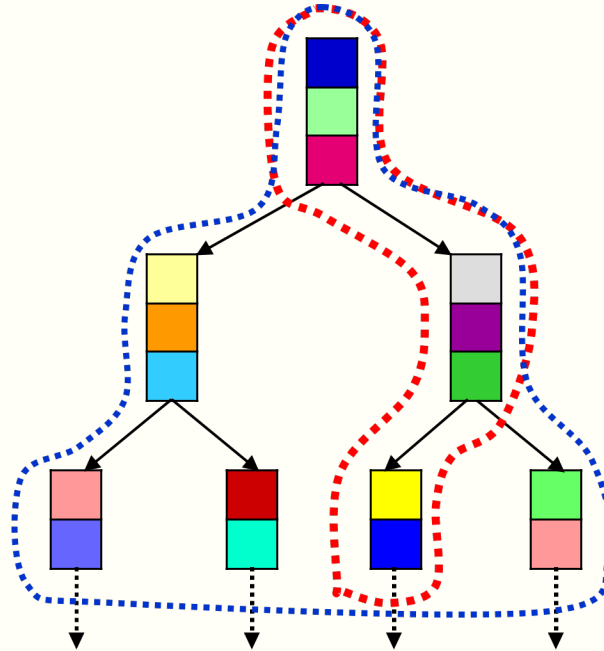
- as observed by other instructions in the pipeline
- e.g., integer addition

❑ More complex instructions may require multiple cycles

- e.g., integer division, square-root
- cache misses!

❑ These latencies, when combined with data dependencies, can result in non-trivial critical path lengths through code

Constraints on Instruction Scheduling



□ Control Dependences

- What do we do when we reach a conditional branch?
 - choose a “frequently-executed” path?
 - choose multiple paths?

Scheduling Constraints: Summary

❑ Hardware Resources

- finite set of FUs with instruction type, bandwidth, and latency constraints
- cache hierarchy also has many constraints

❑ Data Dependences

- can't consume a result before it is produced
- ambiguous dependences create many challenges

❑ Control Dependences

- impractical to schedule for all possible paths
- choosing an “expected” path may be difficult
 - recovery costs can be non-trivial if you are wrong

Instruction Scheduling within a Basic Block

❑ The most common technique for scheduling instructions **within a basic block**

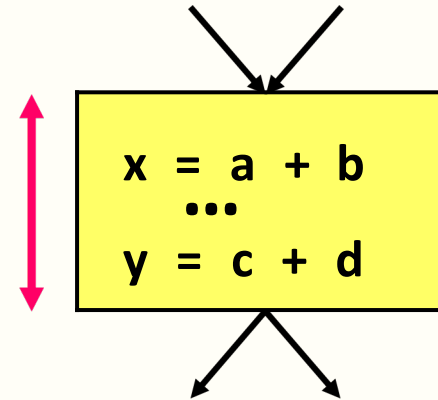
❑ Basic block scheduling doesn't need to worry about:

- control flow

❑ Does need to worry about:

- data dependences

- hardware resources



❑ Even without control flow, the problem is still **NP-hard**

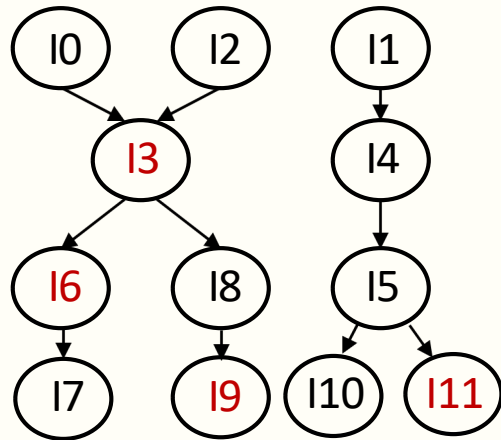
❑ We introduce the **List Scheduling Algorithm**

List Scheduling Algorithm

□ An Example First

Inputs:

Data Precedence Graph (DPG)



Machine Parameters

of FUs:

2 INT, 1 FP

Latencies:

add = 1 cycle, ...

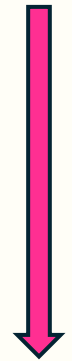
Pipelining:

1 add/cycle, ...

Output:

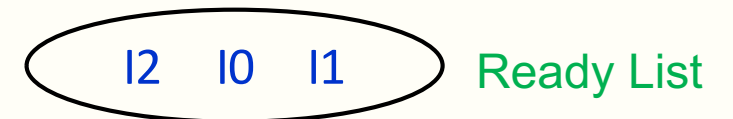
Scheduled Code

ALU 0	ALU 1	FP	Cycle
I0	I2	---	0
---	I1	I3	1
I4	I8	I6	2
I5	---	I9	3
I7	I10	I11	4



□ The Basic Idea

- Maintain a **list** of instructions that are **ready to execute**
 - data dependence constraints would be preserved and machine resources are available
- Moving **cycle-by-cycle** through the schedule template:
 - choose instructions from the list & schedule them; then update the list for the next cycle



What Makes Life Interesting: Choice

❑ Easy case:

- all ready instructions can be scheduled this cycle



❑ Interesting case:

- we need to pick a **subset** of the ready instructions



❑ List scheduling makes choices based upon **priorities**

- assigning priorities correctly is a **key challenge**

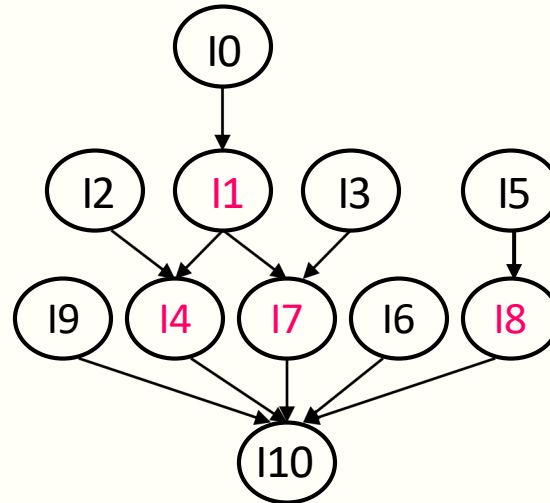
A List Scheduling Example

□ Suppose: Assign priorities based on instruction number

● 2 identical fully-pipelined Function Units

● adds take 2 cycles; all other instructions take 1 cycle

I0: a = 1
I1: f = a + x
I2: b = 7
I3: c = 9
I4: g = f + b
I5: d = 13
I6: e = 19
I7: h = f + c
I8: j = d + y
I9: z = -1
I10: JMP L1



DPG

		Cycle
I0	I2	0
I1	I3	1
I5	I6	2
I4	I7	3
I8	I9	4
--	--	5
I10	--	6

Representing Data Dependences

□ The Data Precedence Graph (DPG)

- Two different kinds of edges: **True edge** and **Anti-edge**

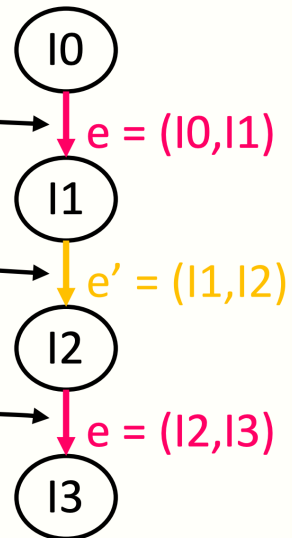
Code

```
I0: x = 1;  
I1: y = x;  
I2: x = 2;  
I3: z = x;
```

true “edges”: **E**
(read-after-write)

“anti-edges”: **E'**
(write-after-read)

DPG



□ Why distinguish them? Do they affect scheduling differently?

- **RAW**: read waits for the value to be computed
- **WAR**: write only needs to ensure it's not started ahead of the read

□ What about output dependences?

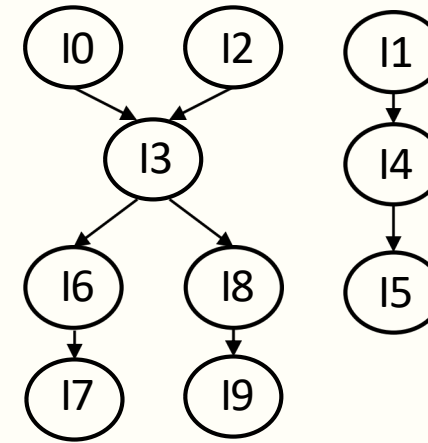
- **WAW**: earlier write is removed by Dead Code Elimination
- Recall we are scheduling a single basic block, so WAW is unconditional

Intuition Behind Priorities

□ Intuitively, what should the priority correspond to?

□ What factors are used to compute it?

- Data dependences?
 - True-dependence (RaW), Anti-dependence (WaR)
- Machine parameters?



of FUs:

2 INT, 1 FP

Latencies:

add = 1 cycle, ...

Pipelining:

1 add/cycle, ...

□ Computing Priorities

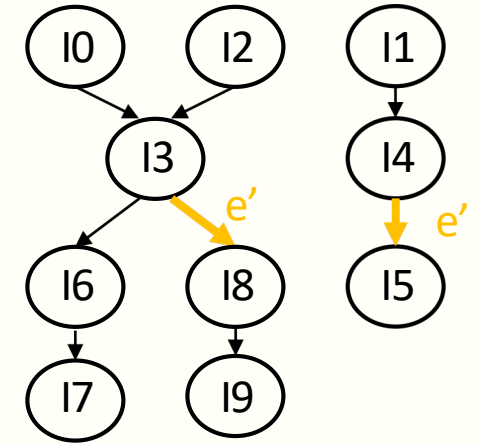
- Let's start with just true dependences (i.e. “edges” in DPG)
- **Priority** = **latency-weighted depth** in the DPG

$$priority(x) = \max(\forall l \in leaves(DPG) \forall p \in paths(x, ..., l) \sum_{p_i = x}^l latency(p_i))$$

Computing Priorities (Cont.)

□ Now let's also take **anti-dependencies** into account

● i.e. **anti-edges** in the set E' , **true edges** in E

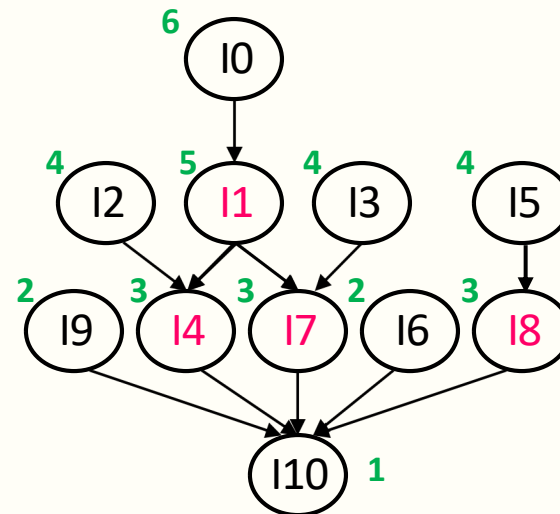


$$priority(x) = \begin{cases} latency(x) & \text{if } x \text{ is a leaf} \\ \max(latency(x) + \max_{(x,y) \in E} (priority(y)), \max_{(x,y) \in E'} (priority(y))) & \text{otherwise.} \end{cases}$$

□ **An Example**

I0: a = 1
I1: f = a + x
 I2: b = 7
 I3: c = 9
I4: g = f + b
 I5: d = 13
 I6: e = 19
I7: h = f + c
I8: j = d + y
 I9: z = -1
I10: JMP L1

- 2 identical **fully-pipelined** FUs
- **adds take 2 cycles**; others take 1 cycle



I0	I2
I1	I3
I5	I6
I4	I7
I8	I9
--	--
I10	--

Cycle

0
1
2
3
4
5
6

I0	I2
I1	I5
I3	I8
I4	I7
I6	I9
I10	--

Cycle

0
1
2
3
4
5
6

Break ties by lower instruction number

- What if we break ties differently?
- **Breaking ties arbitrarily** may not be best

List Scheduling Algorithm

```
cycle = 0;
ready-list = root nodes in DPG;
inflight-list = {};

while (|ready-list|+|inflight-list| > 0) {
    for op = (all nodes in ready-list in decreasing priority order) {
        if (an FU exists for op to start at cycle) {
            remove op from ready-list and add to inflight-list;
            add op to schedule at time cycle;
            if (op has an outgoing anti-edge)
                add all targets of op's anti-edges that are ready to ready-list;
        }
    }
    cycle = cycle + 1;
    for op = (all nodes in inflight-list)
        if (op finishes at time cycle) {
            remove op from inflight-list;
            check nodes waiting for op &
                add to ready-list if all operands available;
        }
    }
}
```

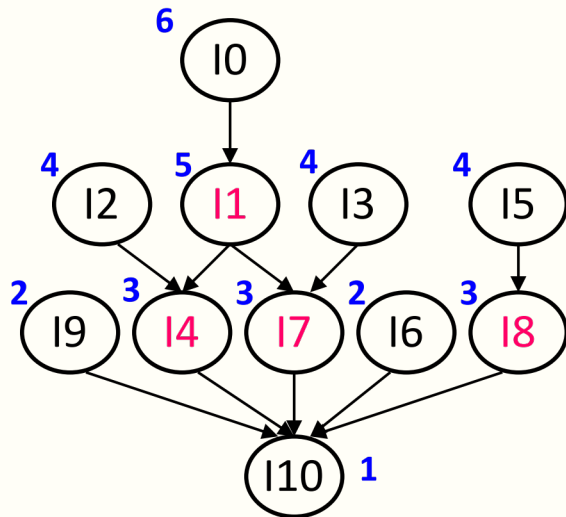
ties?

Different policies break ties differently

Backward List Scheduling

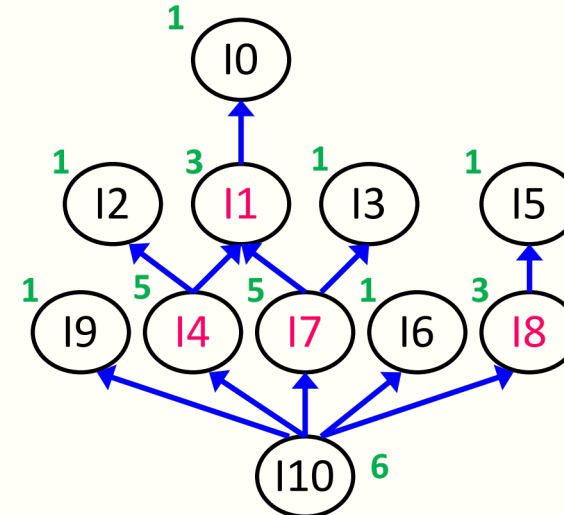
□ Modify the algorithm as follows:

- reverse the direction of all edges in the DPG
- schedule the **finish times** of each operation
 - start times must still be used to ensure Functional Unit availability



Forward Scheduling Priorities

(build up priorities upwards, schedule downwards)



Backward Scheduling Priorities

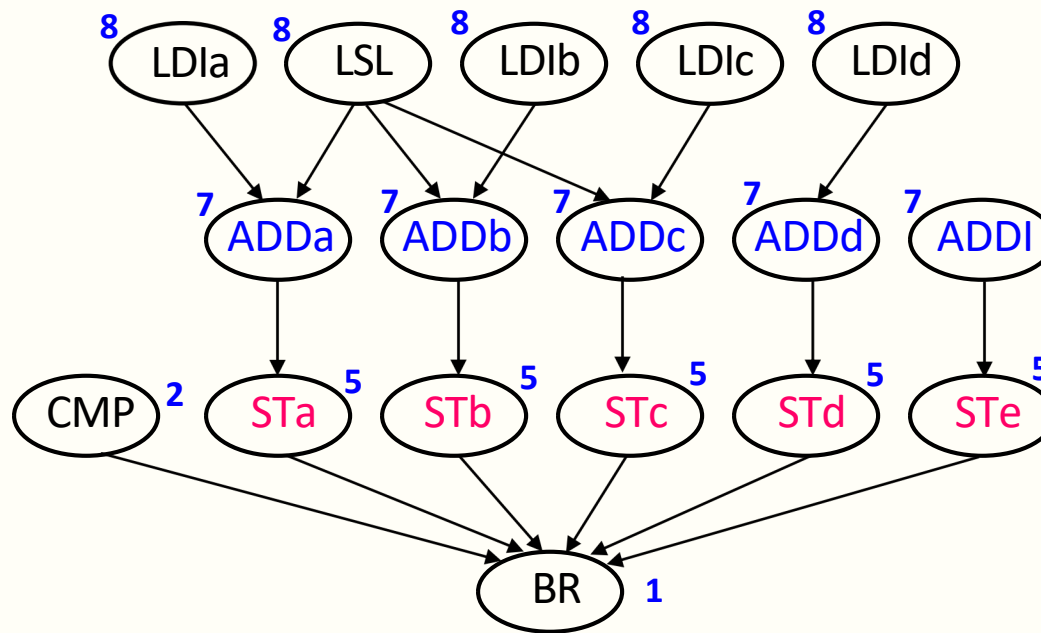
(build up priorities downwards, schedule upwards)

□ Impact of scheduling backwards:

- **clusters** operations near the **end** (vs. the beginning)
- may be **either better or worse** than forward scheduling

A Backward List Scheduling Example

□ Let's Schedule it **Forward** First



Hardware parameters:

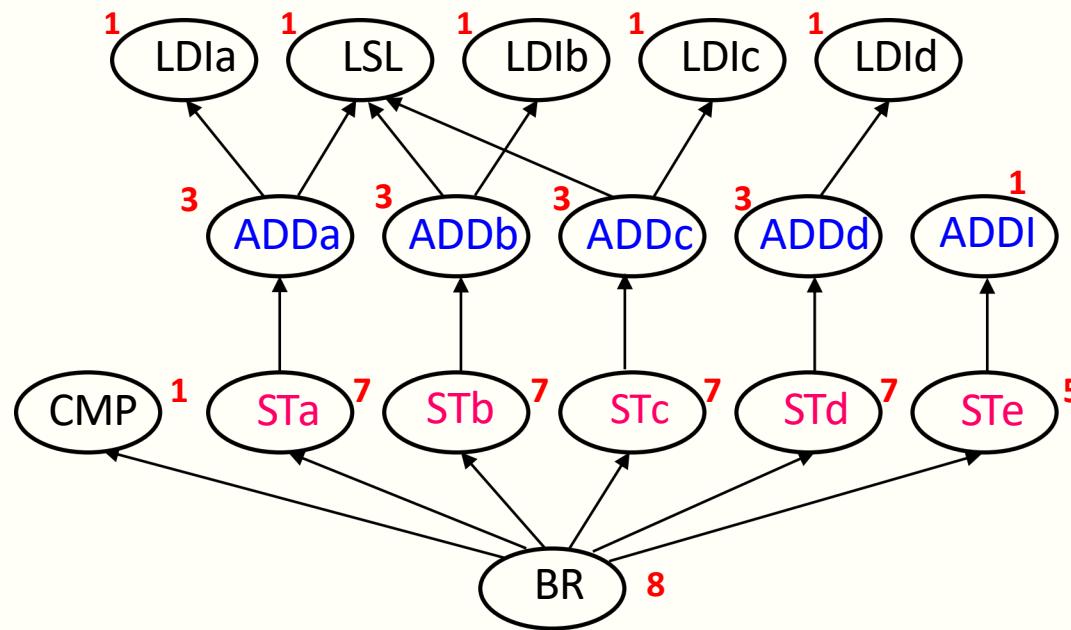
- 2 INT units: **ADDs** take **2 cycles**; others take 1 cycle
- 1 MEM unit: stores (**ST**) take **4 cycles**

INT	INT	MEM	<u>Cycle</u>
LDla	LSL	----	0
LDlb	LDlc	----	1
LDld	ADDa	----	2
ADDb	ADDc	----	3
ADDd	ADDI	STa	4
CMP	----	STb	5
----	----	STc	6
----	----	STd	7
----	----	STe	8
----	----	----	9
----	----	----	10
----	----	----	11
BR	----	----	12

Break ties left-to-right in the left DPG

A Backward List Scheduling Example

□ Now Let's Try Scheduling **Backward**



Hardware parameters:

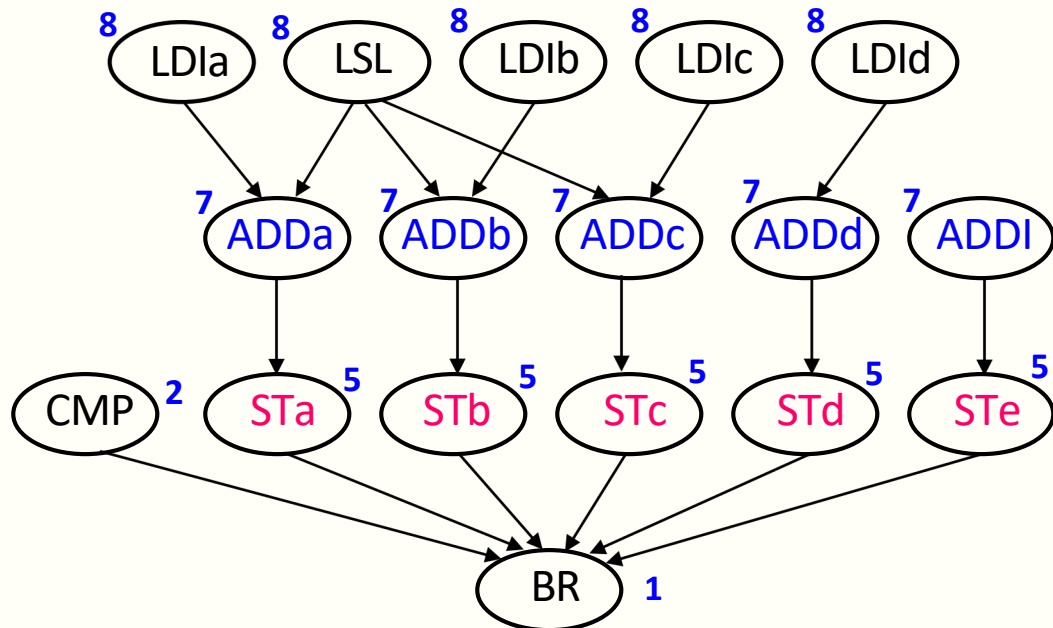
- 2 INT units: **ADDs** take **2 cycles**; others take 1 cycle
- 1 MEM unit: stores (**ST**) take **4 cycles**

INT	INT	MEM	<u>Cycle</u>
LDld	----	----	0
ADDI	LDlc	----	1
ADDd	LSL	----	2
ADDc	LDlb	STe	3
ADDb	LDla	STd	4
ADDa	----	STc	5
----	----	STb	6
----	----	STa	7
----	----	----	8
----	----	----	9
CMP	----	----	10
BR	----	----	11

Break ties left-to-right in the left DPG

Forward vs. Backward List Scheduling

- ❑ Backward scheduling **clusters work near the end**
- ❑ Backward is **better** in this case, but this is **not always** true



Forward

INT	INT	MEM	<u>Cycle</u>
LDla	LSL	---	0
LDlb	LDlc	---	1
LDld	ADDa	---	2
ADDb	ADDc	---	3
ADDd	ADDI	STa	4
CMP	---	STb	5
---	---	STc	6
---	---	STd	7
---	---	STe	8
---	---	---	9
---	---	---	10
---	---	---	11
BR	---	---	12

Backward

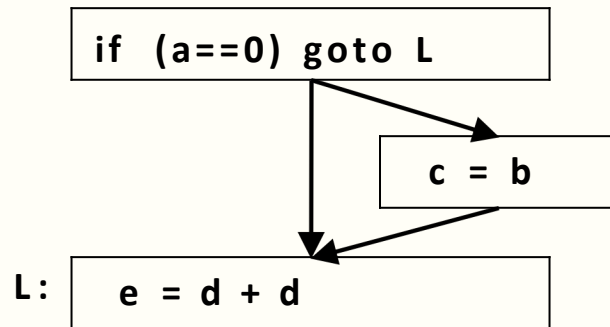
INT	INT	MEM	<u>Cycle</u>
LDld	---	---	0
ADDI	LDlc	---	1
ADDd	LSL	---	2
ADDc	LDlb	STe	3
ADDb	LDla	STd	4
ADDa	---	STc	5
---	---	STb	6
---	---	STa	7
---	---	---	8
---	---	---	9
CMP	---	---	10
BR	---	---	11

List Scheduling Wrap-Up

- ❑ The **priority** function can be **arbitrarily sophisticated**
 - e.g., filling branch delay slots in early RISC processors
- ❑ List scheduling is widely used for instruction scheduling on in-order processors, and it works fairly well
 - 感兴趣，可以读一读LLVM的指令调度源码：
 - [llvm/lib/CodeGen/MachineScheduler.cpp](#)
- ❑ However, it has **two limitations**:
 - It schedules only within a basic block
 - Modern out-of-order processors perform their own dynamic scheduling
 - List scheduling can be used to feed the dynamic scheduler in a good order

An Example of Global Scheduling

Assume each clock can execute
2 operations of any kind
Assume LD takes 2 cycles, fully pipelined



LD R6, 0(R1) ; NOP
NOP ; NOP
BEQZ R6, L ; NOP

B₁ Result of List Scheduling

LD R7, 0(R2) ; NOP
NOP ; NOP
ST R7, 0(R3) ; NOP

B₂

L: LD R8, 0(R4) ; NOP
NOP ; NOP
ADD R8, R8, R8 ; NOP
ST R8, 0(R5)

B₃

LD R6, 0(R1) ; LD R8, 0(R4)
LD R7, 0(R2) ; NOP
ADD R8, R8, R8 ; BEQZ R6, L

B₁

Result of Global Scheduling

L: ST R8, 0(R5) ; NOP

B₃

ST R8, 0(R5) ; ST R7, 0(R3)

B₃'

Useful Definitions

□ Blocks B and B' are **control equivalent** if

- B is executed if and only if B' is executed
- E.g., which sets of blocks are control equivalent?
 - maximal sets: {B1,B4,B6}, {B2}, {B3}, {B5}
- Two ops (instructions) are control equivalent iff their basic blocks are control equivalent (could be from the same basic block)

□ **NonSpeculative(B)**

- all blocks that are control equivalent to B and dominated by B

□ **Speculative(B)** = all blocks B' not control equivalent to B such that

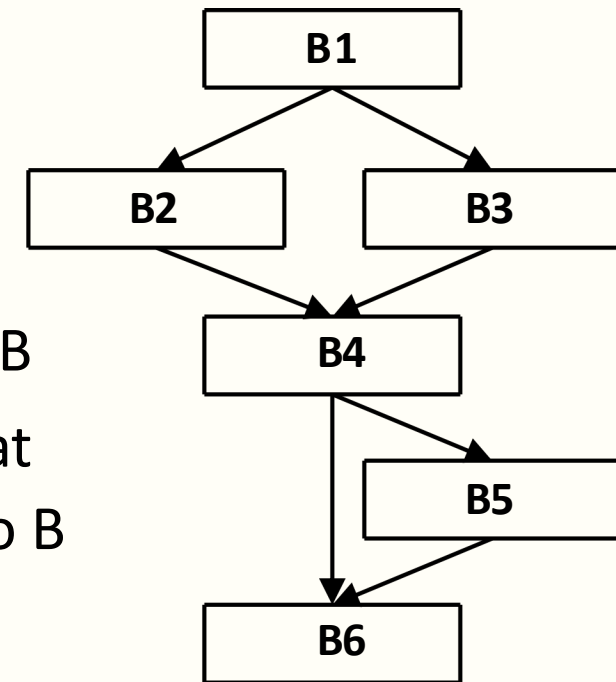
- B' succeeds at least one block B'' that is control equivalent to B
- **B' is dominated by B''**

NonSpeculative(B1)? {B1,B4,B6}

NonSpeculative(B2)? {B2}

Speculative(B1)? {B2,B3,B5}

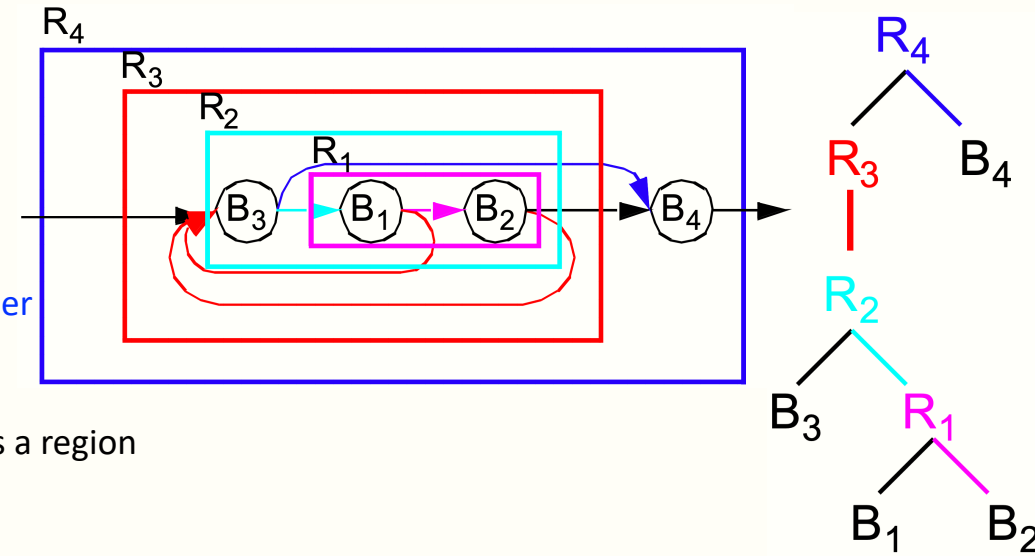
Speculative(B2)? {}



A Basic Global Scheduling Algorithm

□ Program Representation

- A **region** in a control flow graph (CFG) is:
 - a set of **basic blocks** and all the **edges** connecting these blocks
 - expect possibly back edges into the header
 - such that control from outside the region **must enter through the header**
- A **procedure** is represented as a **hierarchy of loop regions**
 - The entire control flow graph is a region
 - Each natural loop (single entry with back edge to it) in the flow graph is a region
 - Natural loops are hierarchically nested



□ Schedule regions **from inner to outer** (innermost loops first, then outer loops)

- Treat inner loop as a black box unit: **can schedule around it but not into it**
- **Ignore all the loop-back edges**, then get an acyclic graph

□ Only **upward code motion**, to either:

- a “control-equivalent” block (non-speculative), or
- a control-equivalent block of a dominating predecessor
 - speculative, 1 branch

OK to move instructions up to a control-equivalent block? **yes**

Ok to move instructions up to a control-equivalent block of a dominating predecessor? **yes**

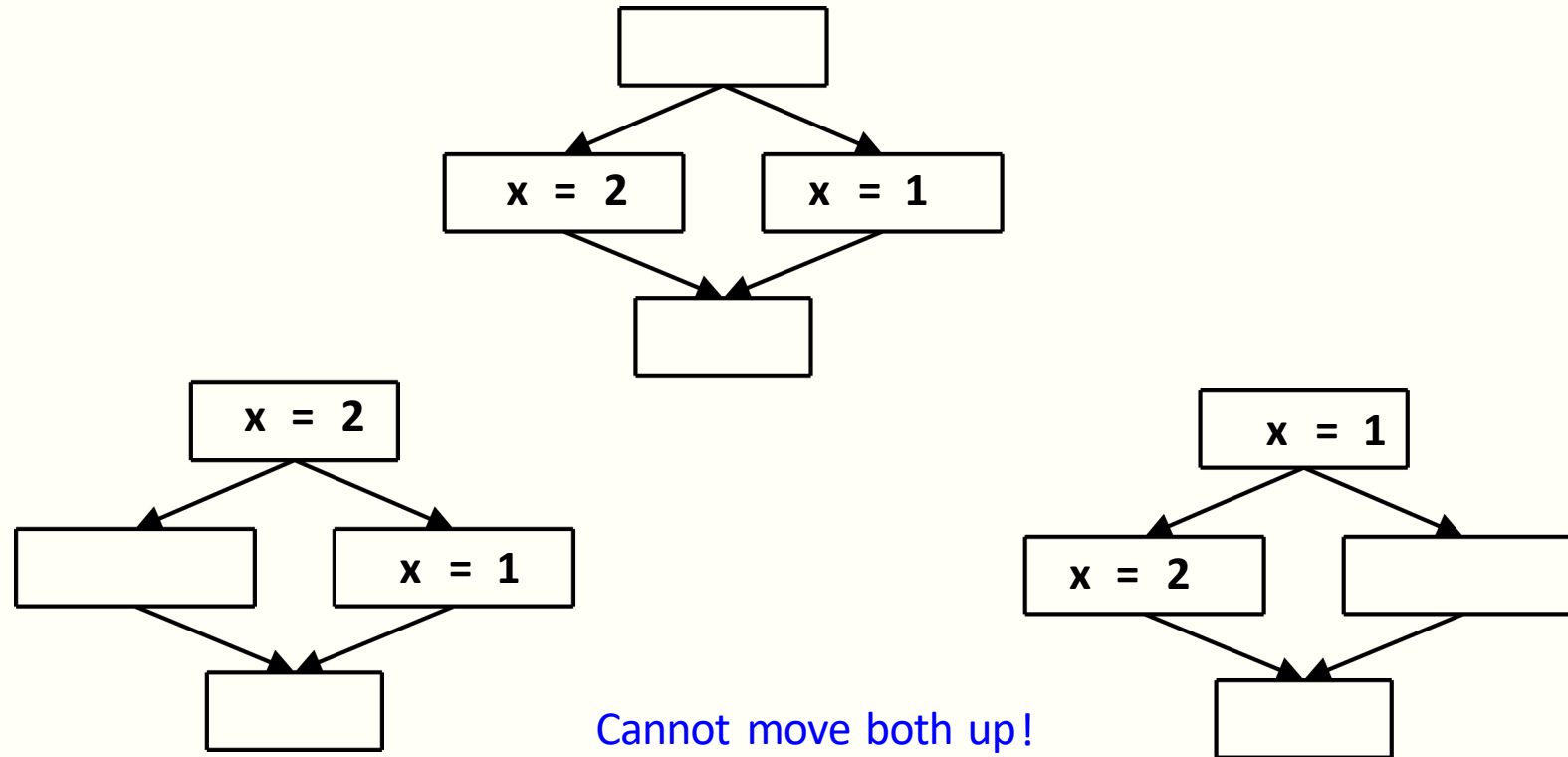
□ No creation of copies

Basic Algorithm

```
Compute data dependences;
For each region R in the hierarchy of loop regions from inner to outer {
  For each basic block B of R in prioritized topological order {
    CandInsts = ready instructions in NonSpeculative(B)  $\cup$  Speculative(B);
    For (t = 1, 2, ... until all instructions from B are scheduled) { // schedule time slots in order
      For (n in CandInst in priority order) { // may or may not be from B
        if (ok to move n to B && n has no resource conflicts at time t) {
          S(n) = < B, t >; // instruction n is mapped to basic block B and time slot t
          Update resource commitments;
          Update data dependences; // what could have changed? see next slide
        }
      }
    }
    Update CandInsts; // scheduled insts will often make new insts ready
  }
}
```

- **Priority functions:** Non-speculative before speculative, and otherwise use the same priority as in list scheduling
- **Ok to move:** Don't speculatively move a store instruction; don't move a procedure call, etc

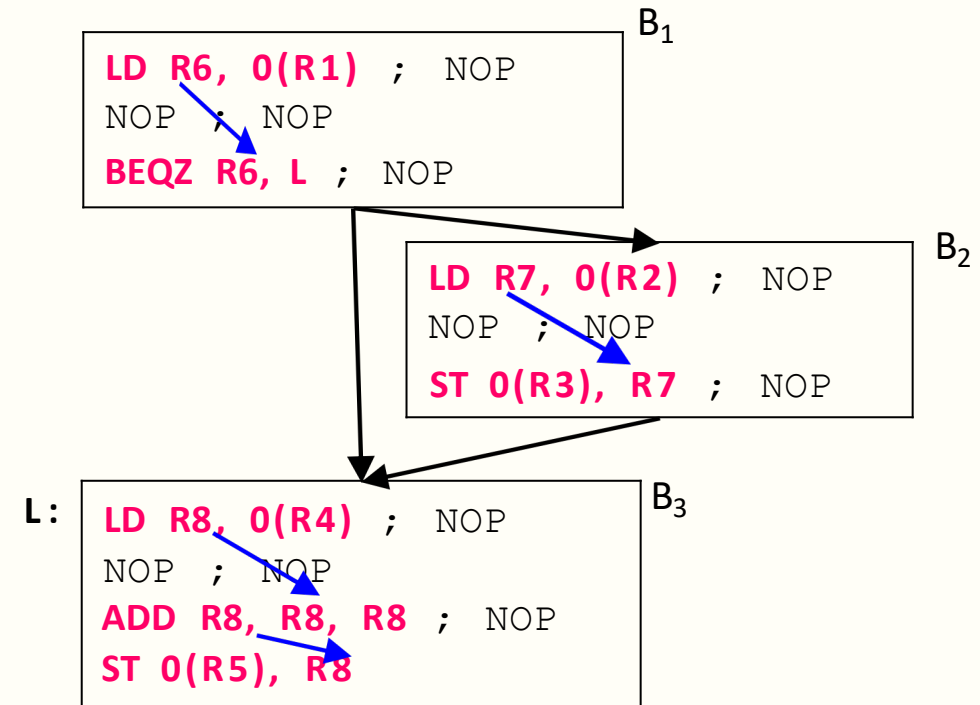
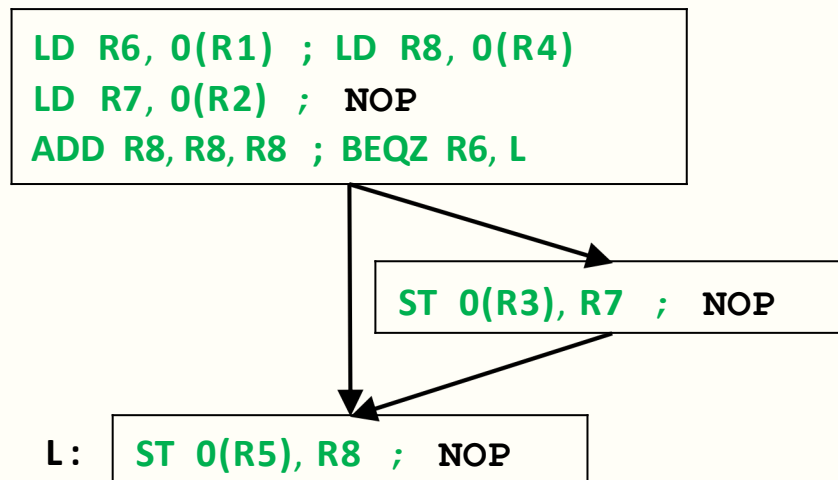
Updating Data Dependence after Code Motion



If a variable is live at a program point, then we cannot move a speculative definition to the variable above that program point

Basic Global Scheduling Algorithm: Example

- Each clock: 2 operations of any kind
- LD takes 2 cycles, fully pipelined
- Priority order of blocks: B_1, B_2, B_3
- Data dependences? **Blue arcs at right**
- Control equivalent Blocks? $\{B_1, B_3\}, \{B_2\}$
- NonSpeculative(B_1)? $\{B_1, B_3\}$
- Speculative(B_1)? $\{B_2\}$
- CandInsts? $\{\text{LD R6; LD R8; LD R7}\}$

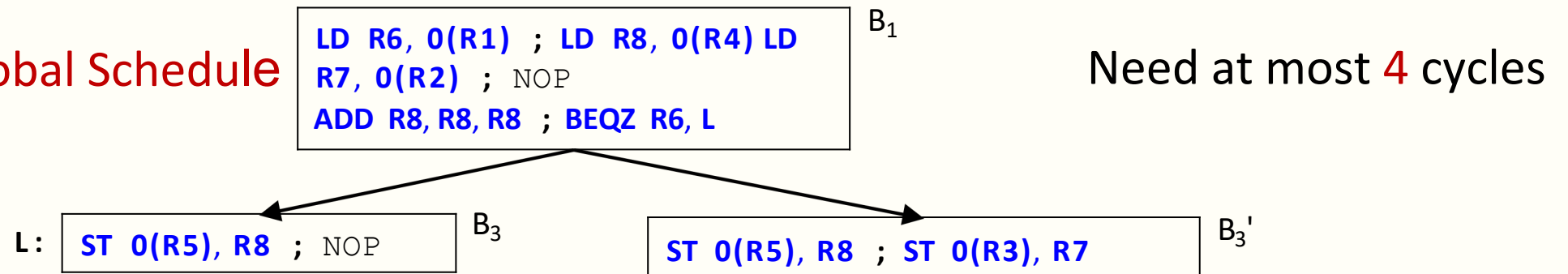


- $t=2$? $\{\text{ADD R8; BEQZ R6; LD R7}\}$
- $t=3$? $\{\text{ADD R8; BEQZ R6; ST R7}\}$
- NonSpeculative(B_2)? $\{B_2\}$
- Speculative(B_2)? $\{\}$
- CandInsts for B_2 ? $\{\text{ST R7}\}$
- CandInsts for B_3 ? $\{\text{ST R8}\}$

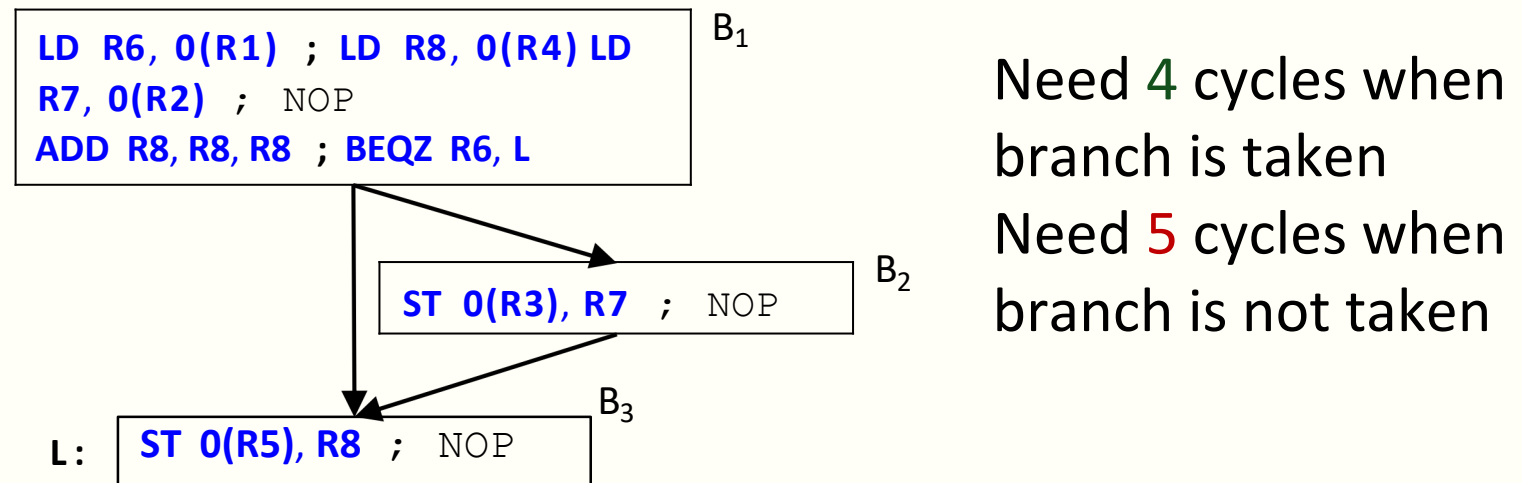
Comparison to Earlier Global Schedule

❑ Basic Algorithm's schedule requires one more cycle when branch not taken

Earlier Global Schedule



Basic Algo's Schedule



Extension

❑ In region-based scheduling, iteration boundaries limit code motion:

- Operations from one iteration cannot overlap with those from another

❑ Prepass before scheduling: **loop unrolling**

- Especially important to move operations up loop back edges

❑ An Example: 假设N为偶数

```
for (i = 0; i < N; i++) {  
    a[i] = b[i] + 1;  
}
```

Original Loop



```
for (i = 0; i < N; i += 2) {  
    a[i] = b[i] + 1;  
    a[i+1] = b[i+1] + 1;  
}
```

Unrolled Loop

调度前

```
Loop:  
I1: x0 = load b[i]  
I2: y0 = x0 + 1  
I3: store a[i], y0  
I4: x1 = load b[i+1]  
I5: y1 = x1 + 1  
I6: store a[i+1], y1  
  
I7: i = i + 2  
I8: if (i < N) goto Loop
```

调度后

```
Loop:  
I1: x0 = load b[i]  
I4: x1 = load b[i+1]  
I2: y0 = x0 + 1  
I5: y1 = x1 + 1  
I3: store a[i], y0  
I6: store a[i+1], y1  
  
I7: i = i + 2  
I8: if (i < N) goto Loop
```

References

- ❑ “An Experimental Evaluation of List Scheduling”, Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Rice University, Dept. of Computer Science Tech. Rep. 98-326, 1998.
- ❑ Gibbons, Philip B., and Steven S. Muchnick. "Efficient instruction scheduling for a pipelined architecture." *Proceedings of the 1986 SIGPLAN symposium on Compiler construction*. 1986.

Hooray! No Homework! 🎉