

12. Instruction Scheduling

2025 Fall

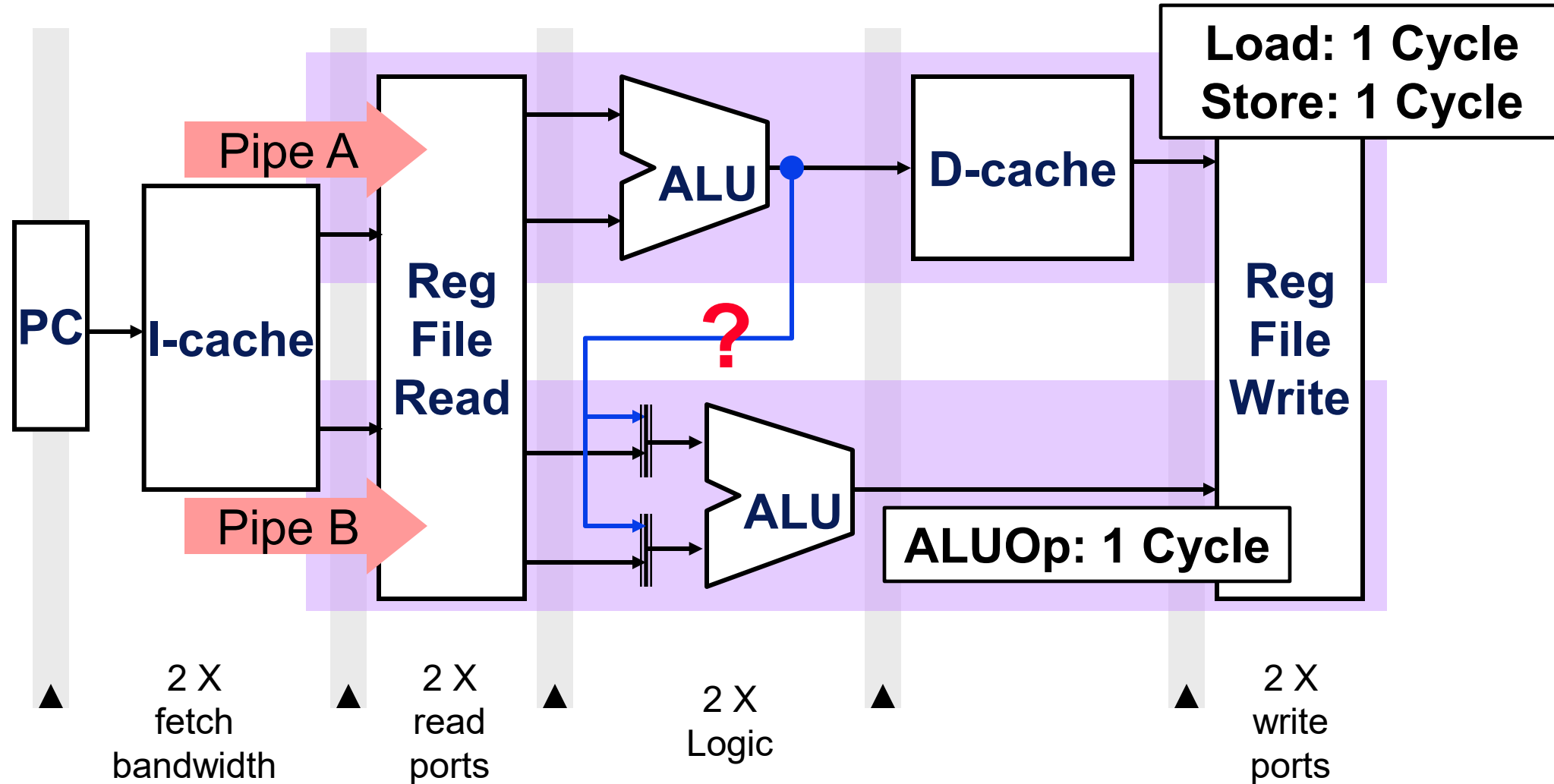
Hunjun Lee

Hanyang University

Instruction Scheduling

- **There are some opportunities to improve the performance using scheduling and reordering**
- **The instruction scheduling should not affect the functionality**
- **We cannot reschedule all the instructions due to the dependencies**

CPU Microarchitecture (Superscalar)

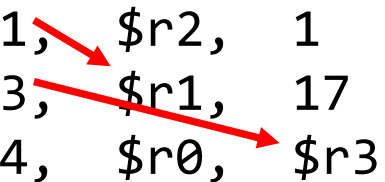


ILP: Instruction-level parallelism

- **ILP is the parallel or simultaneous execution of a sequence of instructions**
 - Inter-dependent instructions cannot be executed in parallel
- **Program ILP = Avg. # of instructions / Cycle (step)**
 - How many instructions are simultaneously executed in parallel

code1:

```
addi $r1, $r2, 1
divi $r3, $r1, 17
sub  $r4, $r0, $r3
```



Max ILP = 1 (execute serially)

code2:

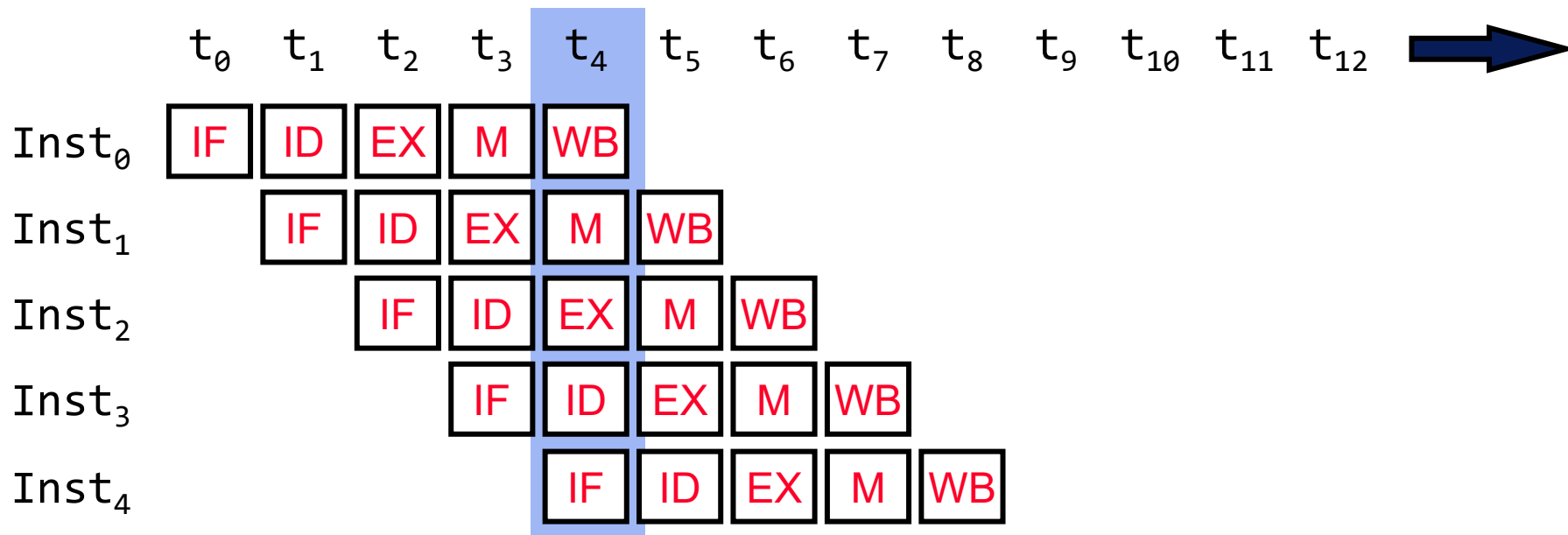
```
addi $r1, $r2, 1
divi $r3, $r9, 17
sub  $r4, $r0, $r10
```

Max ILP = 3 (execute parallel)

Pipeline + SuperScalar

◆ Pipelining: executing multiple instructions in parallel

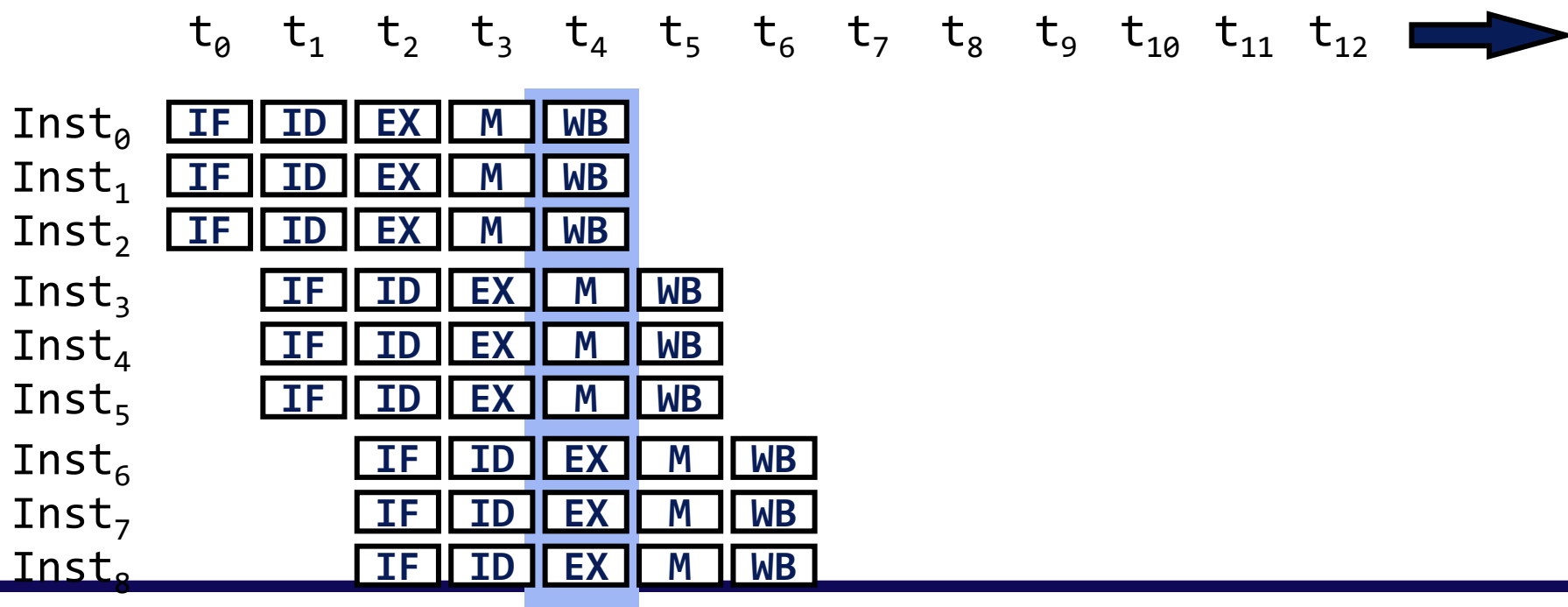
- Operation latency = 1
- Peak IPC = 1
- HW ILP = # of instructions / # of cycles required = 1



Pipeline + SuperScalar

- Superscalar (+ pipelined) execution

- Operation latency = 1 baseline cycle
- Peak IPC = N per baseline cycle
- HW ILP = # of instructions / # of cycles required = N



Hazards in the dual-issue CPU

- More instructions are executed in parallel
- EX data hazard
 - Can't use ALU result in load/store in same packet

Slot 0 { add **\$t0**, \$s0, \$s1

Slot 1 { load \$s2, 0(**\$t0**)

- Load-use hazard
 - Still one cycle use latency

Slot 0 { load **\$t0**, 0(\$s0)

Slot 1 { add \$t2, **\$t0**, \$s1

↘ 1 cycle stall

Hazards in the dual-issue CPU

- It also suffers from false dependencies
- **Write after write hazard**
 - The two packed instructions cannot write to the same register

load **\$t0**, 0(\$s0)

add **\$t0**, \$t1, \$s1

Scheduling Constraints

- **Resource Constraints**

- Processors have finite number of resources → Limits on how these resources can be used together
 - Fixed issue width (4 ~ 8 instructions)
 - Limited functional units per given instruction type
 - Limited pipelining with a given functional unit (division?)

- **Program Constraints (Dependence, Precedence ...)**

- There are ordering relationships in the program
 - Dependence #1: Data Dependence
 - Dependence #2: Control Dependence
- There are aggressive scheduling techniques to overcome the dependency

Scheduling Constraints

- **Resource Constraints**

- Processors have finite number of resources → Limits on how these resources can be used together
 - Fixed issue width (4 ~ 8 instructions)
 - Limited functional units per given instruction type
 - Limited pipelining with a given functional unit (division?)

- **Program Constraints (Dependence, Precedence ...)**

- There are ordering relationships in the program
 - Dependence #1: Data Dependence
 - Dependence #2: Control Dependence
- There are aggressive scheduling techniques to overcome the dependency

Finite Issue Width

- In a superscalar machine \rightarrow we cannot issue more than N different instructions within a cycle

	t_0	t_1	t_2	t_3	t_4	t_5
Inst_0	IF	ID	EX	M	WB	
Inst_1	IF	ID	EX	M	WB	
Inst_2	IF	ID	EX	M	WB	
Inst_3		IF	ID	EX	M	WB
Inst_4		IF	ID	EX	M	WB
Inst_5		IF	ID	EX	M	WB

Superscalar Parallelism

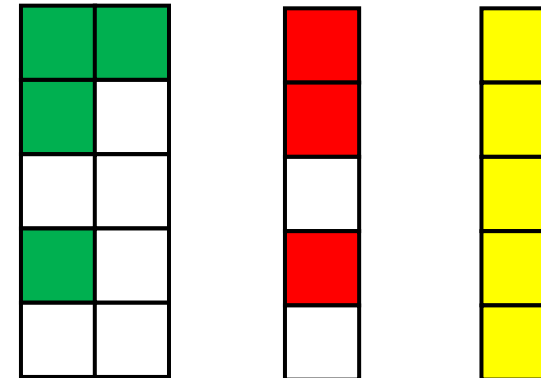
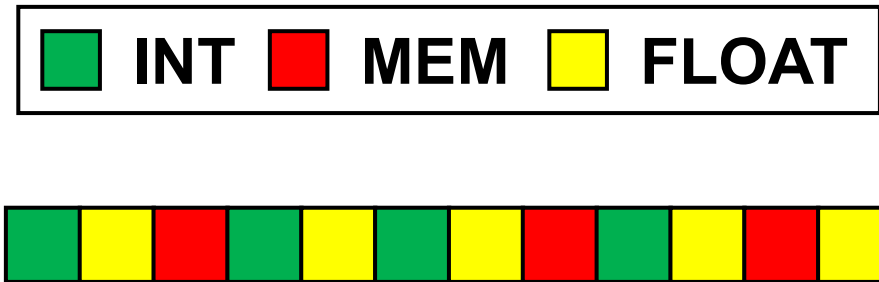
Operation Latency: 1

Issuing Rate: N

Superscalar Degree: N

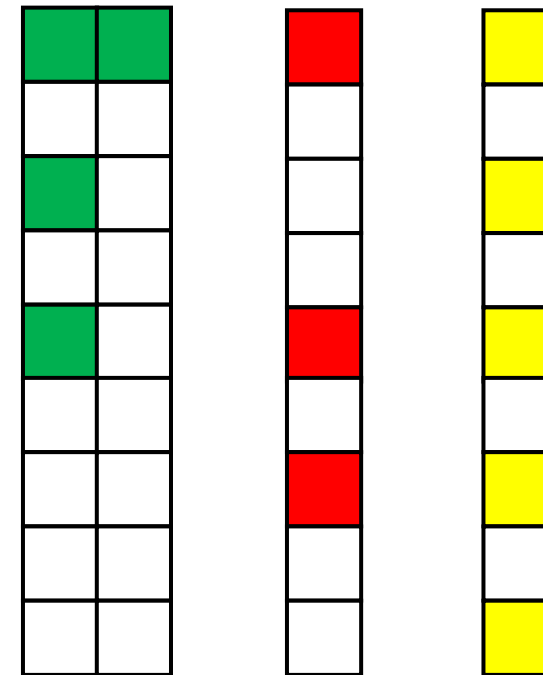
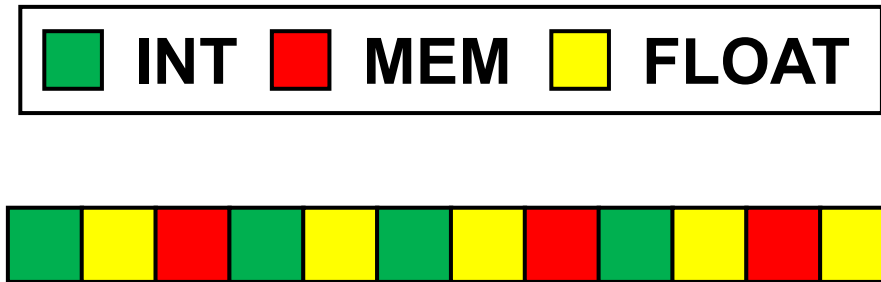
Limited FUs per Inst. Type

- We cannot issue an instruction for a given functional unit if it is fully utilized
 - Ex) 4-way superscalar with 2 integer units, 1 memory units, and 1 floating-point units



Limited FUs per Inst. Type

- Another requirement: A floating-point operation takes two cycles



Scheduling Constraints

- **Resource Constraints**

- Processors have finite number of resources → Limits on how these resources can be used together
 - Fixed issue width (4 ~ 8 instructions)
 - Limited functional units per given instruction type
 - Limited pipelining with a given functional unit (division?)

- **Program Constraints (Dependence, Precedence ...)**

- There are ordering relationships in the program
 - Dependence #1: Data Dependence
 - Dependence #2: Control Dependence
- There are aggressive scheduling techniques to overcome the dependency

Dependencies Limit Parallelization

- We cannot execute consecutive instructions in parallel upon control and data dependencies

Data Dependency

Parallel
Group

```
z = z + 1
x = y + 1
w = x * 10
```

Control Dependency

Parallel
Group

```
z = z + 1
x = y + 1
if (cc)
    w = w + 1
```

Control Dependence

- We cannot parallelize instructions when there is a control dependence
 - We cannot move instruction inside the branch $b = a * a$

Parallel
Group

```
z = z + 1
x = y + 1
if (a > t) then {
    b = a * a
}
c = a * d
```

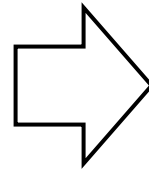

Overcoming Control Dependence - 1

- **Speculative code motion**

- Move control-dependent instruction ahead of a branch so that it can be executed speculatively in a parallel group

**Parallel
Group**

```
...  
d = d + 1  
  
if ( a > t ) then {  
    b = a * a  
}  
  
c = a + d
```



**Parallel
Group**

**b is dead
after branch**

```
...  
d = d + 1  
b = a * a  
  
if ( a > t ) then {  
    b = a * a  
}  
  
c = a + d
```

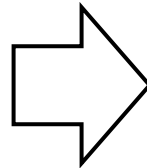
Overcoming Control Dependence - 2

- **Speculative code motion w/ correction**
 - This is not applicable to store operations (no speculative store)

Parallel Group

```
...  
d = d + 1  
  
if ( a > t ) then {  
    b = a * a  
}  
  
c = a + b
```

b is live



Parallel Group

```
...  
d = d + 1  
b' = a * a  
  
if ( a > t ) then {  
    b = a * a  
    b = b'  
}  
  
c = a + b
```

Speculative Code Motion Summary

- **Speculative code motion**

- Move control-dependent instruction ahead of a branch so that it can be executed speculatively in a parallel group

- **Effectiveness of speculation**

- Branch taken: GOOD
- Branch not-taken: Nothing to lose
- This should be done to exploit underutilized resources

- **Correctness Problem**

- Liveness, Exception (e.g., division), Permanency (e.g., store)

Data Dependence

- **Must maintain the order of accesses to the same locations**
 - True dependence: write \rightarrow read
 - Anti dependence: read \rightarrow write
 - Output dependence: write \rightarrow write
- **We cannot move instructions if there is a data dependence**

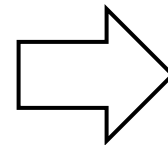
$\begin{aligned} r1 &= r4 + r5 \\ r2 &= r1 + 1 \end{aligned}$

False Data Dependence

- **We can rename registers using copies**
 - Remove data dependence (false dependence) and move the instructions

**Parallel
Group**

```
r1 = r4 + r5
...
r2 = r1 + 1
r1 = r3 - 2
```



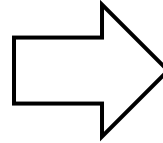
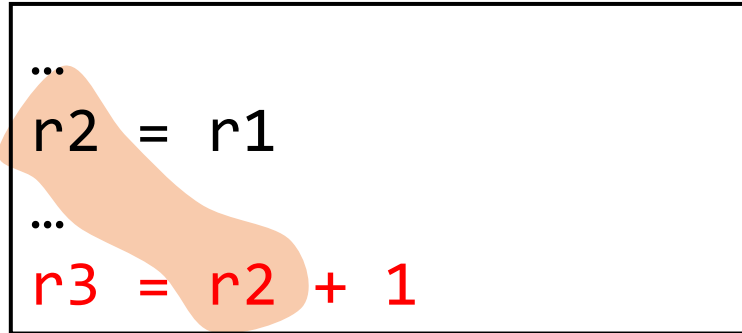
**Parallel
Group**

```
r1 = r4 + r5
...
r1' = r3 - 2
r2 = r1 + 1
r1 = r1'
```

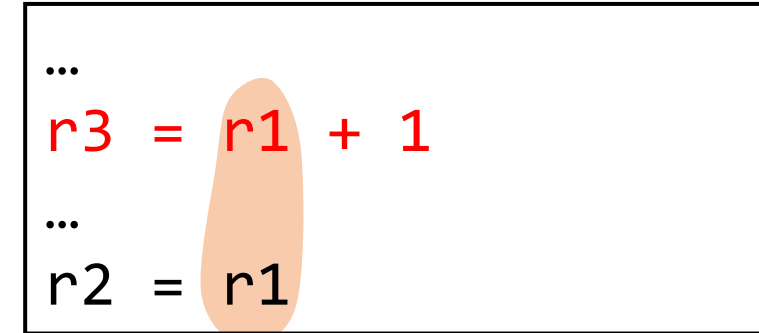
True Data Dependence

- **We can rename registers using copies**
 - Perform forward substitution to mitigate true dependencies

```
...  
r2 = r1  
...  
r3 = r2 + 1
```



```
...  
r3 = r1 + 1  
...  
r2 = r1
```



Basic Block Scheduling

- **Basic block scheduling**

- List scheduling
- Interaction between register allocation and scheduling

- **Global scheduling**

- Cross-block code scheduling

- **Software pipelining**

Virtual CPU Model

- **All registers are read at the beginning of a cycle → and are written at the end of a cycle**
- **Example: The following two instructions can be executed in parallel**
 - `load r2 0(r1)`
 - `add r1 r3, r4`
 - Load will use an old value (before it is written by add)

List Scheduling

- **The most common technique: scheduling instructions within a basic block**
 - We do not care about control flow ... ~~(covered later)~~
- **We care about ... data dependences and hardware resources**
- **This is an NP-hard problem ☹**

List Scheduling - 1

- **Input:**

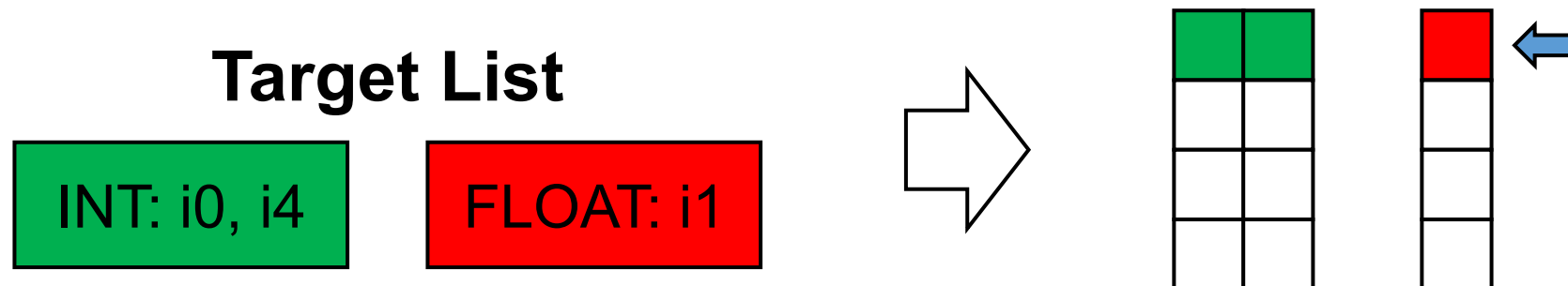
- Data Precedence Graph (DPG): The graph structure of the instructions according to the dependences between instructions
- Machine Parameters: The available resources and execution latency ...

- **Output:**

- The scheduled instruction code (grouped together to maximize the performance)

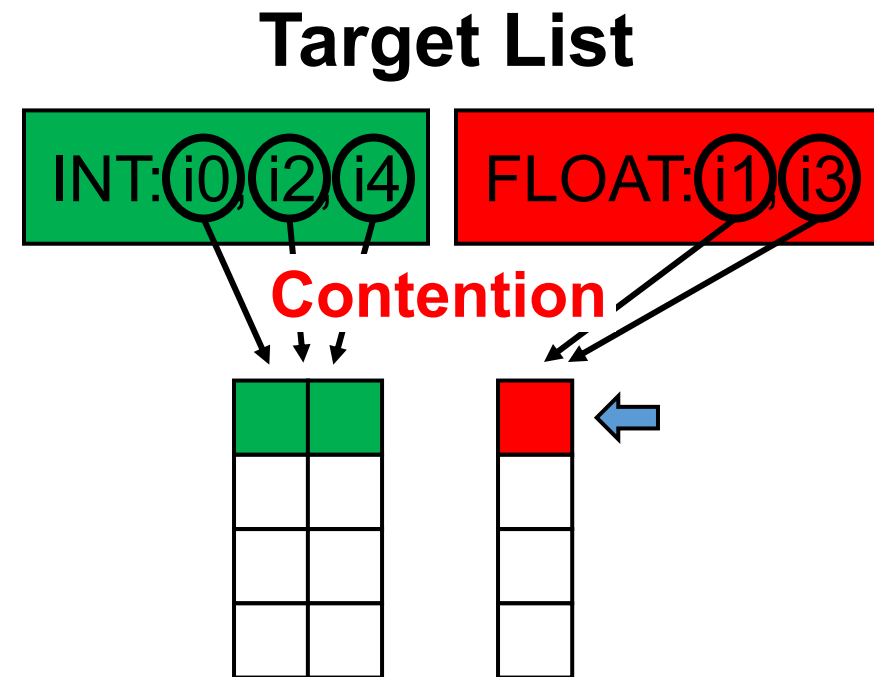
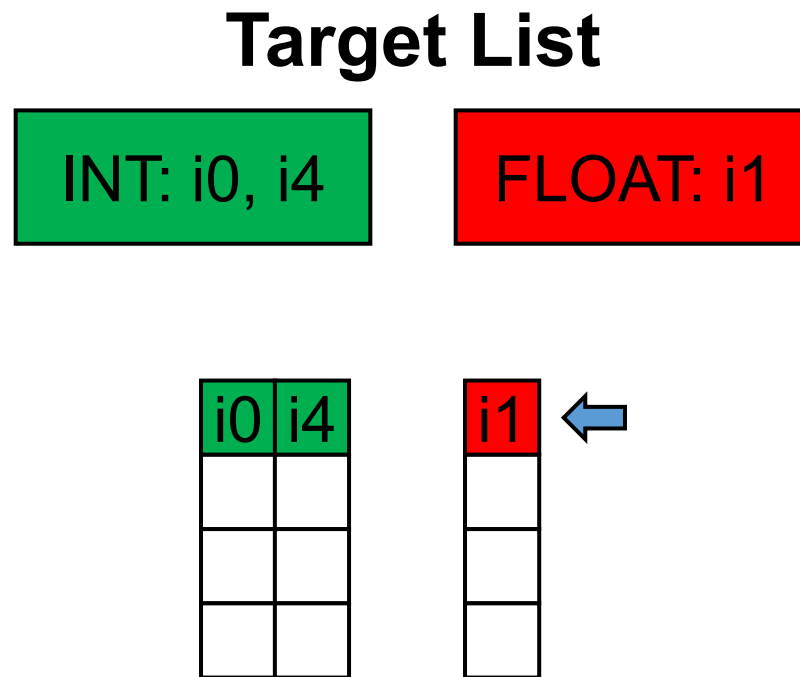
List Scheduling - 2

- **There is a list to keep the list of ready instructions**
 - Req #1. All the operands are ready to execute (and no false dependences)
 - Req #2. The target resources are available
- **Iteratively conduct scheduling in a cycle-by-cycle manner**
 - Choose target instructions from the list allocate
 - Update the list and iterate over the same procedure again



Key Challenge

- There is a problem when there are multiple ready instructions, but we do not have enough resources

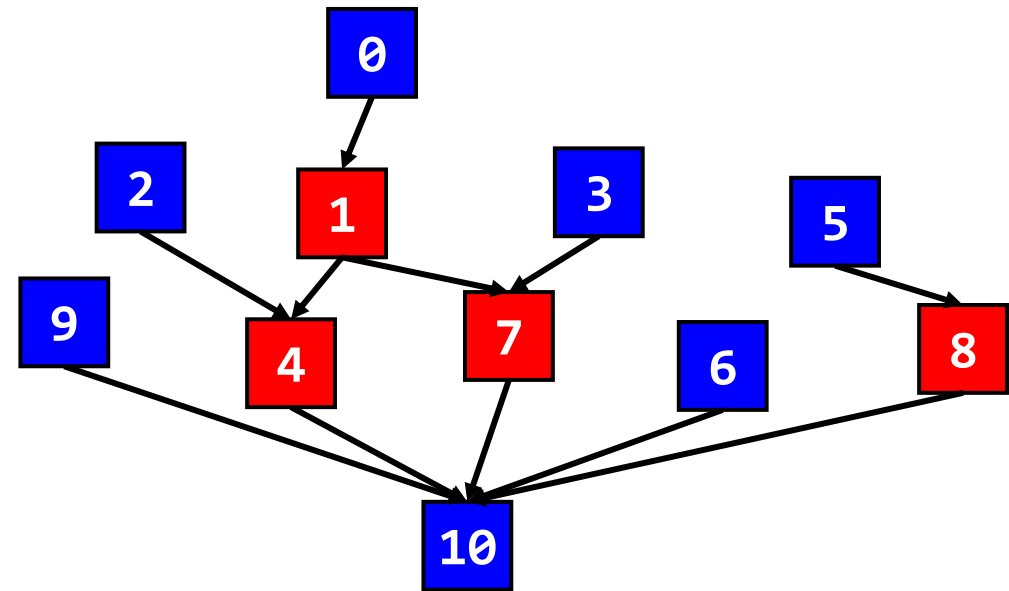
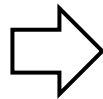


Data Precedence Graph (DPG) - 1

- **Data dependence graph:**
 - nodes: instructions
 - edges: data dependence constraints

0:	a = 1
1:	f = a + x
2:	b = 7
3:	c = 9
4:	g = f + b
5:	d = 13
6:	e = 19
7:	h = f + c
8:	j = d + y
9:	z = -1
10:	j L1

Add: 2 Cycles
Others: 1 Cycle



Data Precedence Graph (DPG) - 2

- **There are two types of edges in the DPG**

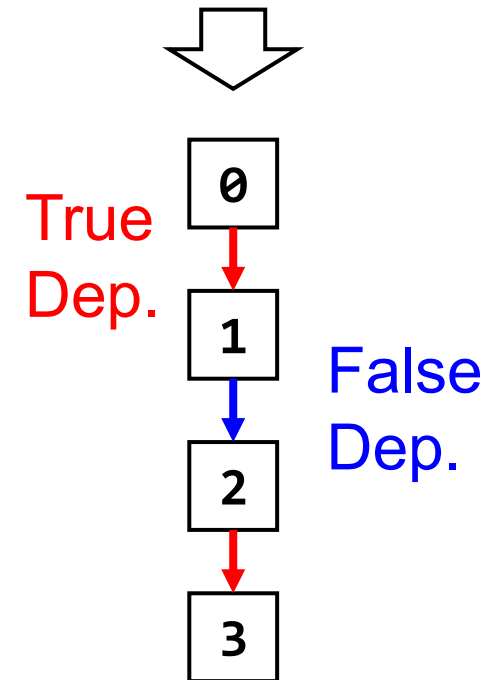
- True dependency
- False dependency

0:	x	=	1
1:	y	=	x
2:	x	=	2
3:	z	=	x

- **Q1. Should we treat the RAW and WAR dependency separately?**

- **Q2. What about WAW dependency?**

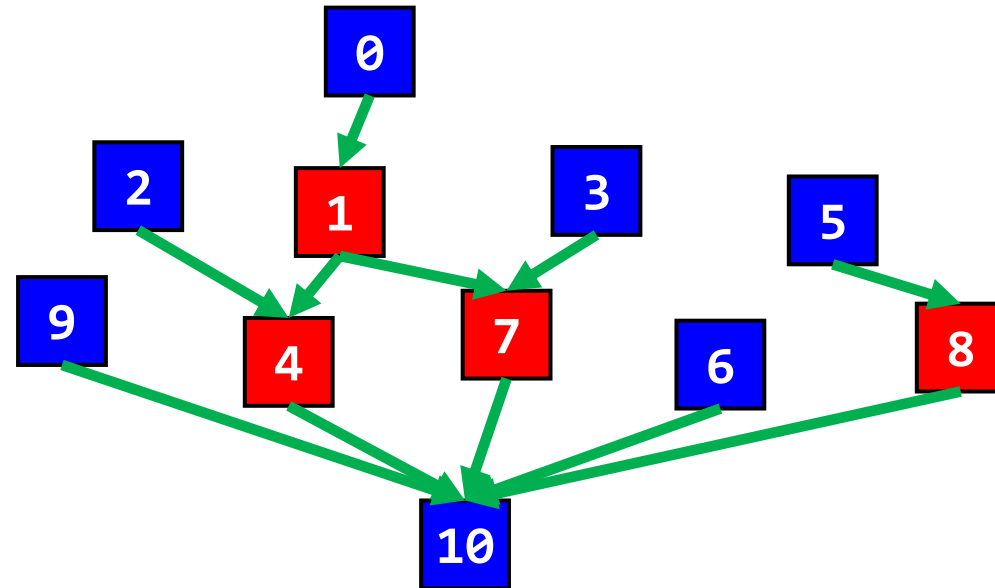
- This should be removed after dead code elimination (within a basic block!)



Determining Priorities in Contention

- Let's assume that everything is true dependences
- Priority: latency-weighted depth

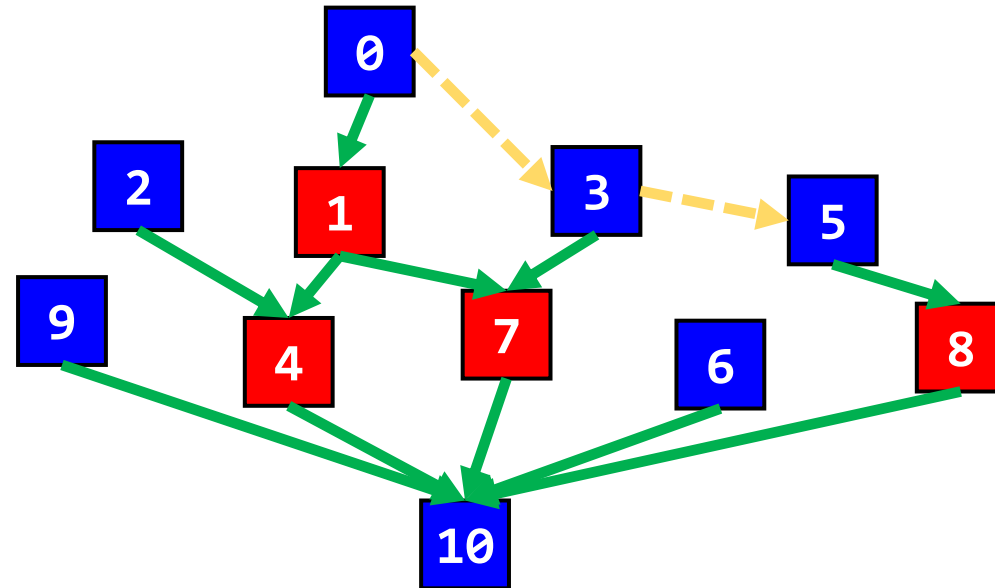
$$priority(x) = latency(x) + \max_{(x,y) \in E} (priority(y))$$



Determining Priorities in Contention

- Now consider the exact effect of the anti-dependences
 - We can schedule two anti-dependent instruction at once (instead of waiting for the predecessor)

$$priority(x) = \max(\text{latency}(x) + \max_{(x,y) \in E} (priority(y)), \max_{(x,y) \in E'} (priority(y)))$$



List Scheduling Algorithm

```
cycle = 0
ready-list = root nodes in DPG           // Indicates the ready list
inflight-list = {}                       // Indicates the executing instructions (at the pipeline)

while (ready-list or inflight-list not empty) {
    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
        }
}
}
```

What is there is a tie?

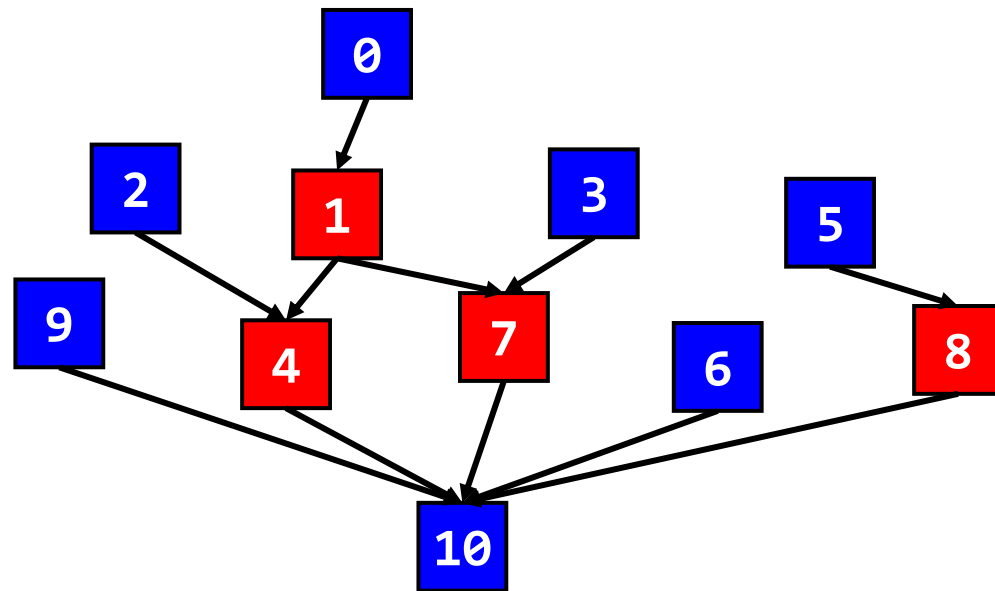
decreasing priority order

Discussions on Breaking Ties

$$priority(x) = \max(latency(x) + \max_{(x,y) \in E} (priority(y)), \max_{(x,y) \in E'} (priority(y)))$$

Break ties by lower instruction number

Ready List



Add takes two cycles;
Others take one cycles



0	2
1	3
5	6
4	7
8	9
10	

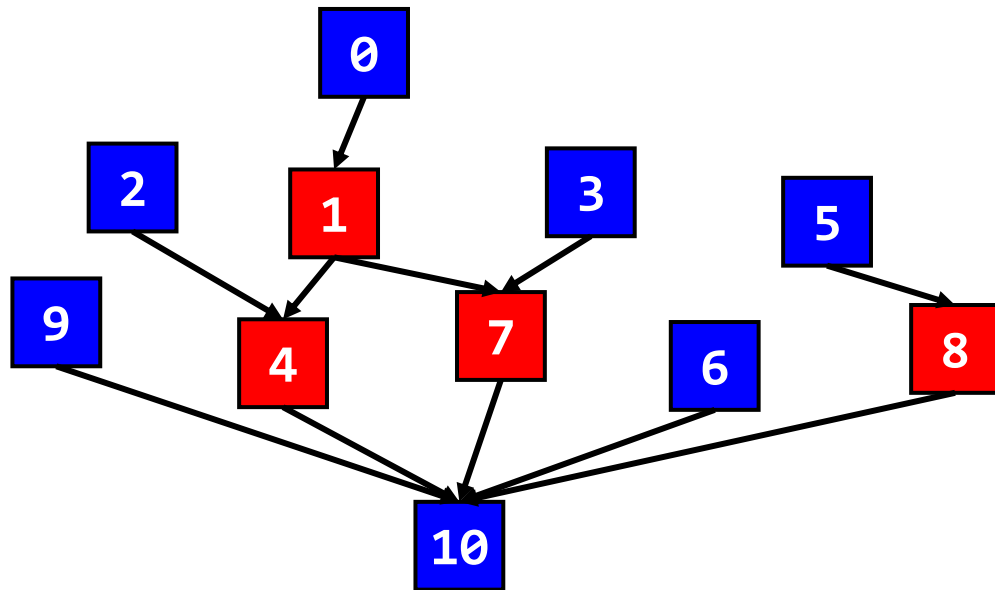
← 0, 2, 3, 5, 6, 9
 ← 1, 3, 5, 6, 9
 ← 5, 6, 9
 ← 4, 7, 8, 9
 ← 8, 9
 ←
 ← 10

Discussions on Breaking Ties

$$priority(x) = \max(latency(x) + \max_{(x,y) \in E} (priority(y)), \max_{(x,y) \in E'} (priority(y)))$$

What about this?

Ready List



Add takes two cycles;
Others take one cycles

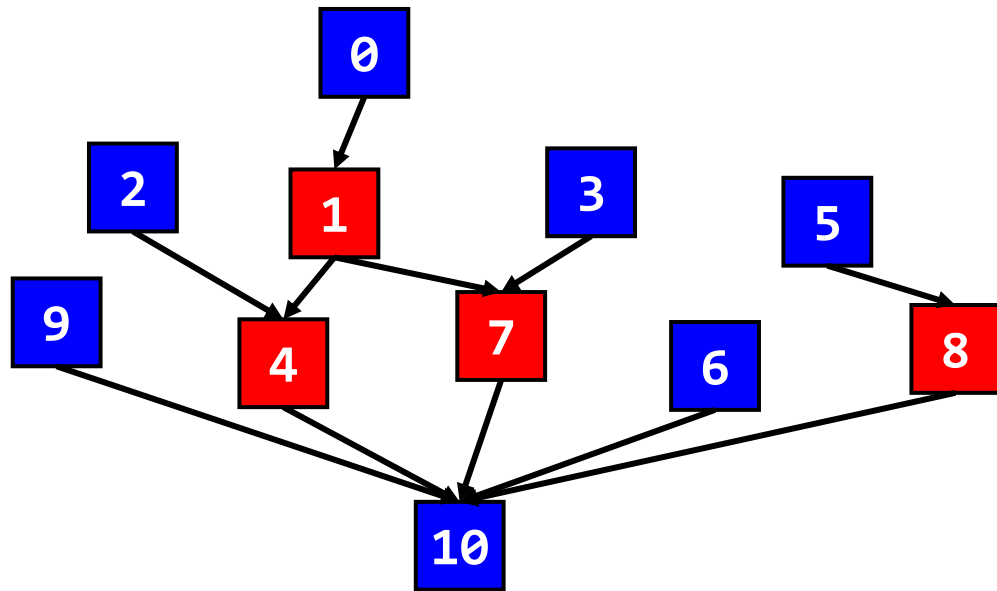


0	2
1	5
3	8
4	7
6	9
10	

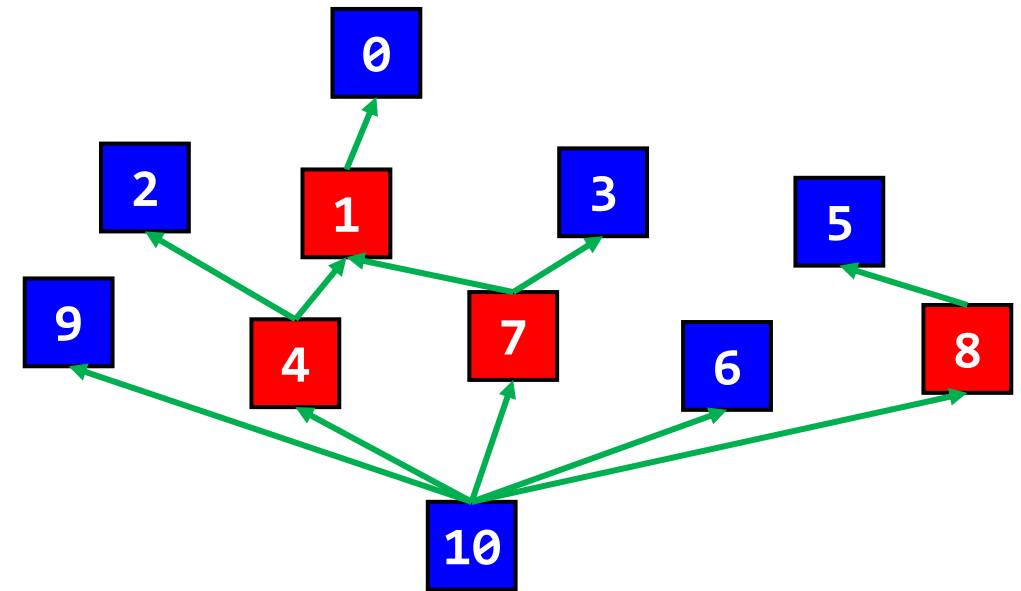
← 0, 2, 3, 5, 6, 9
 ← 1, 3, 5, 6, 9
 ← 3, 6, 8, 9
 ← 4, 6, 7, 9
 ← 6, 9
 ← 10

Alternative Approach

- **Scheduling from backwards ...**
 - Schedule the finish times instead of the start times



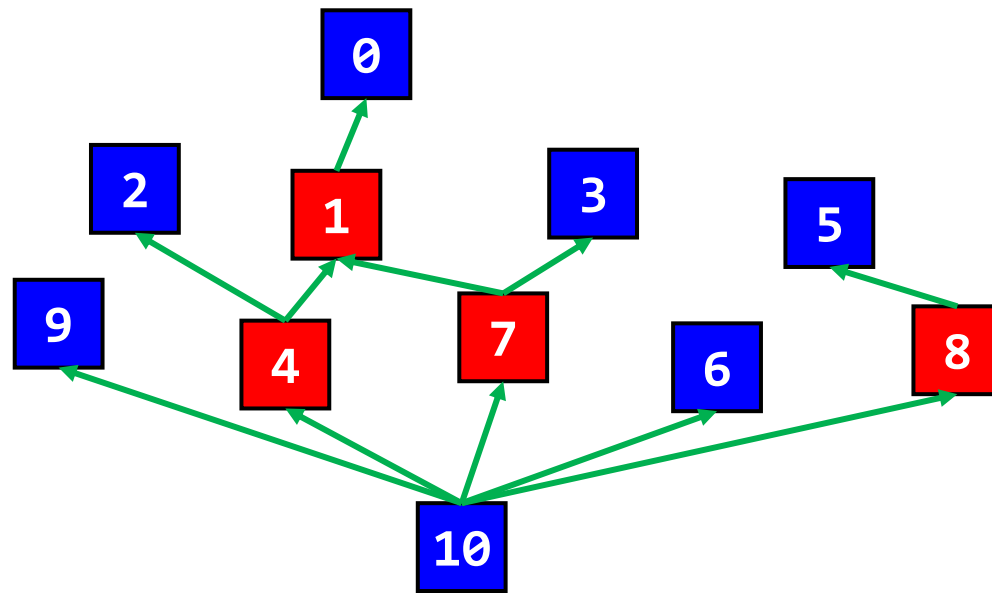
Forward Scheduling



Backward Scheduling

Backward List Scheduling

- Reverse the direction of edges & schedule the finish time
- This can be randomly good or bad ...



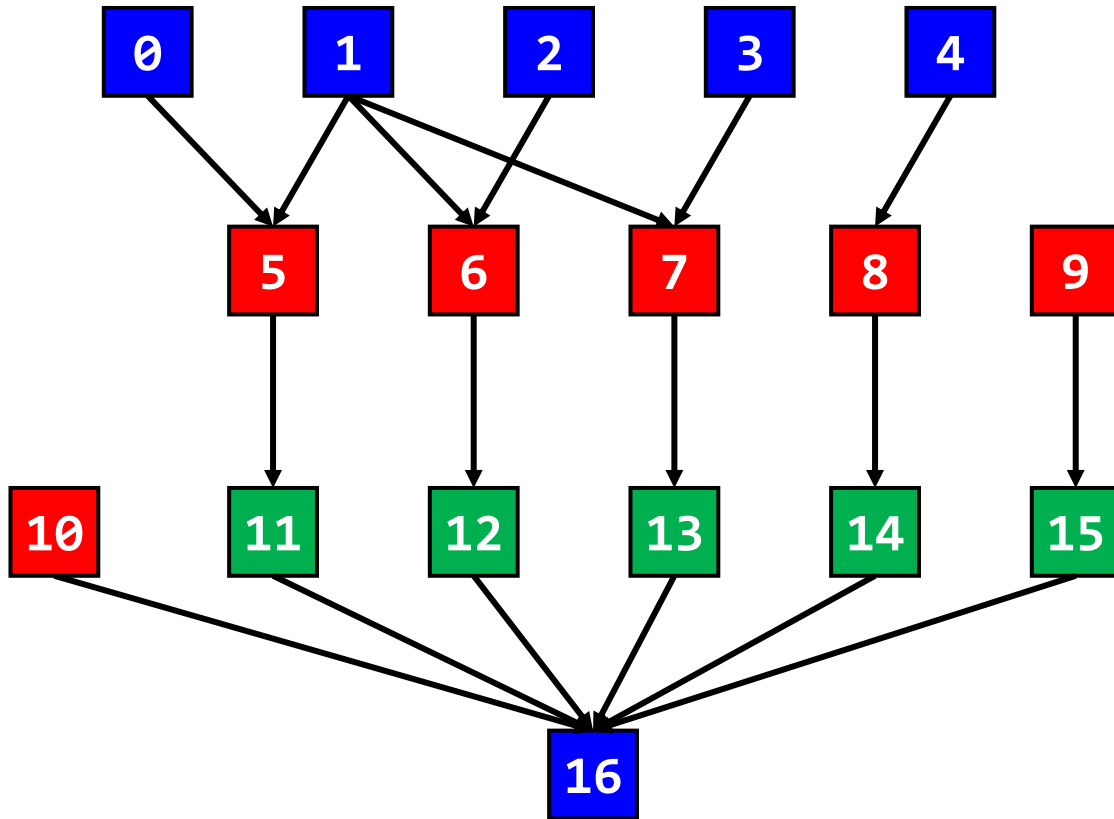
Add takes two cycles;
Others take one cycles



0	5	←	0, 5
1	3	←	1, 3
2	8	←	2, 3, 8
4	7	←	4, 7, 8
6	9	←	6, 9
10		←	10

List Scheduling (Forward Scheduling)

Assume pipelined HW

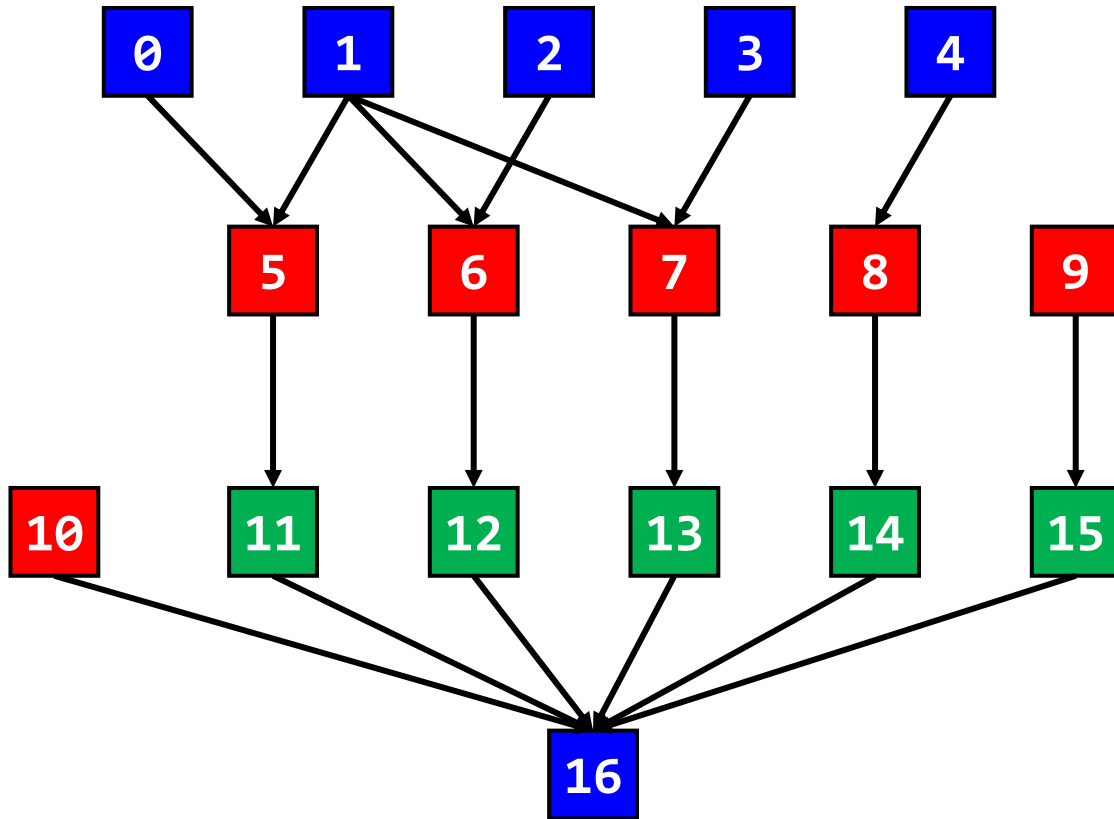


1 cyc	2 cyc	3 cyc
0	9	
1	10	
2	5	15
3	6	
4	7	11
	8	12
		13
		14
		16

0, 1, 2, 3, 4, 9, 10
 1, 2, 3, 4, 10
 2, 3, 4, 5, 15
 3, 4, 6
 4, 7, 11
 8, 12
 13
 14
 -
 -
 16

List Scheduling (Backward Scheduling)

Assume pipelined HW



1 cyc	2 cyc	3 cyc
4		
3	9	
2	8	
1	7	15
0	6	14
	5	13
		12
		11
	10	
		16

4
 3, 4, 9
 2, 3, 8
 1, 2, 7, 15
 0, 6, 14, 15
 5, 13, 14, 15
 12, 13, 14, 15
 11, 12, 13, 14, 15
 10
 -
 16

Advanced Approaches

- **RBF scheduling:**
 - Schedule beach block M times forward and backward
 - Break ties randomly for each trial

Basic Block Scheduling

- **Basic block scheduling**

- List scheduling
- Interaction between register allocation and scheduling

- **Global scheduling**

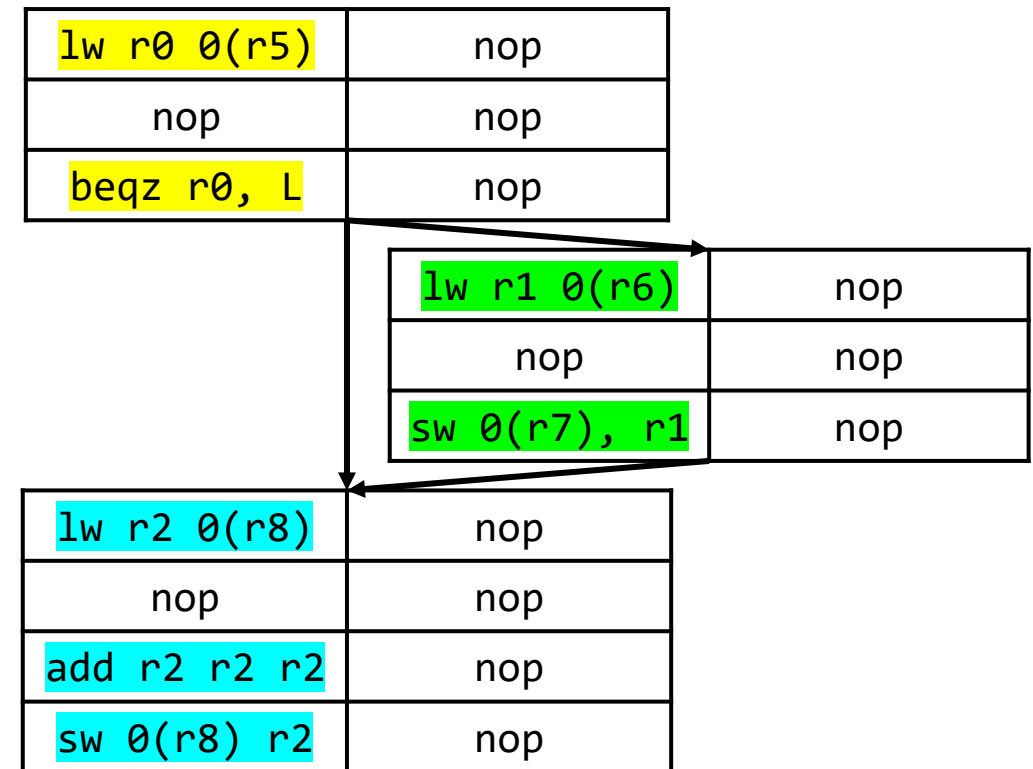
- Cross-block code scheduling

- ~~**Software pipelining**~~

Global Scheduling Example

- **Machine Model:**

- Fully-pipelined execution path
 - LD takes two cycles + Others take one cycle
- Two parallel general-execution path



Basic Features

- **Control equivalence:** If $i1$ and $i2$ are control equivalent \rightarrow if $i1$ is executed if and only if $i2$ is executed
- **Speculation:** An instruction is speculatively executed if it is executed before all the control-dependent instructions have been executed
 - \rightarrow As long as there are (1) no side-effects, (2) no exception, (3) does not violate data dependence

Code Motion

- **Goal: Probabilistically reduce the execution time based on the frequency of the execution**
- **There are two different options:**
 - You may either move the instructions downwards to successor basic blocks
 - You may also move the instructions upwards to predecessor basic blocks

Problem: to where and how to move the instructions?

Global Scheduling Basics

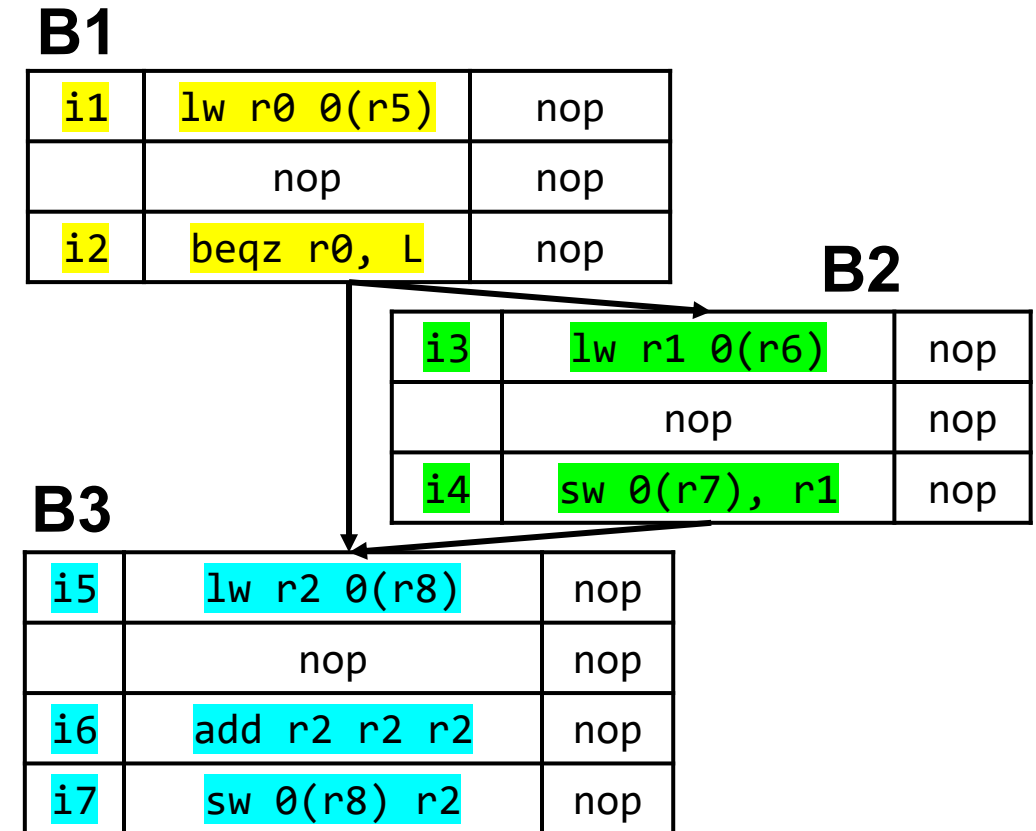
- **Schedule innermost loops first:**
 - The instructions should escape from the most executed basic blocks
- **Apply upward code-motion to the following two options:**
 - **Non-speculative:** A control-equivalent block
 - **Speculative:** A control-equivalent block of a dominating predecessor

Scheduling Algorithm

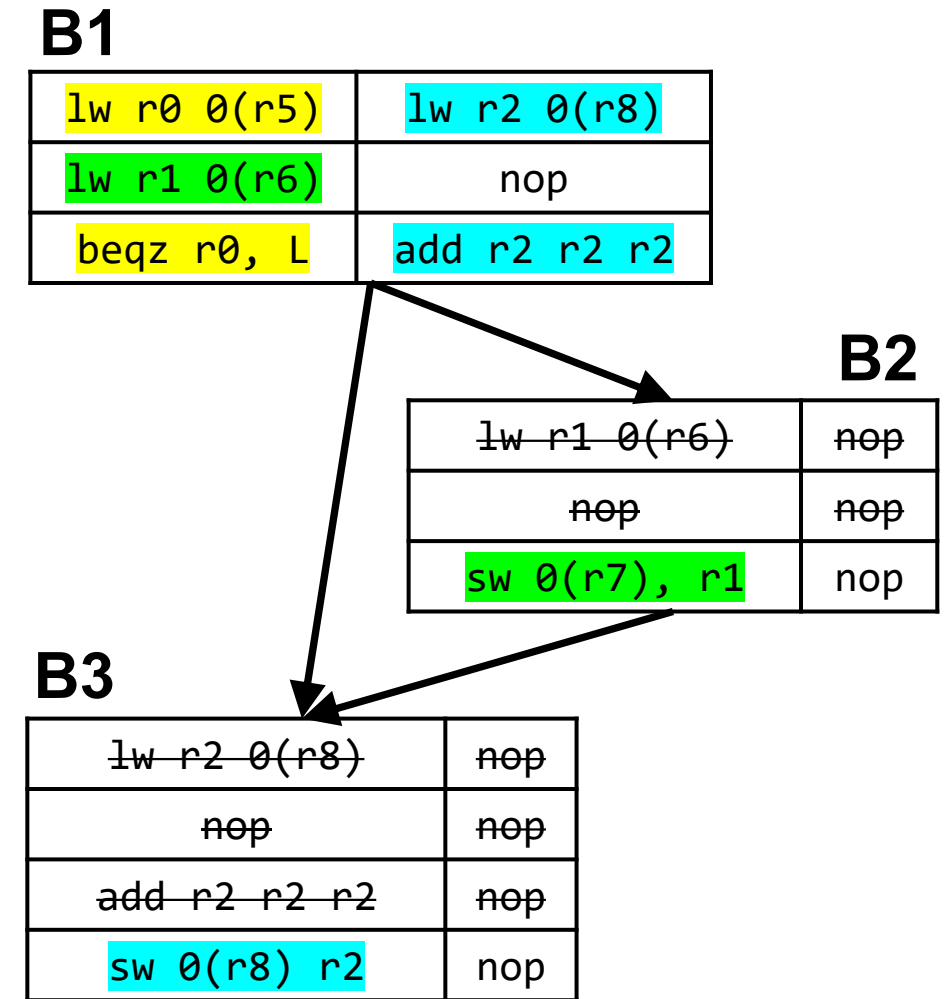
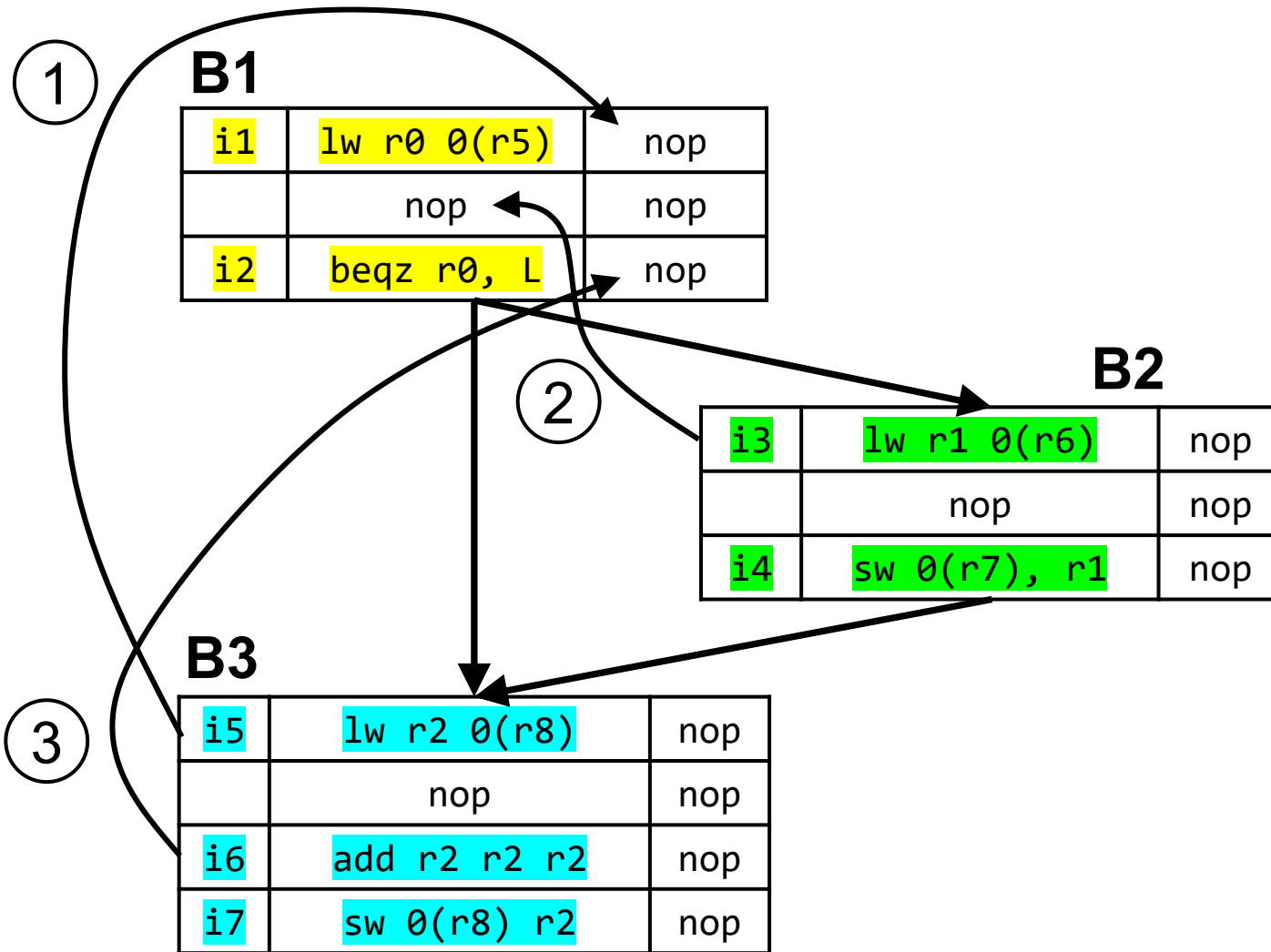
```
Compute data dependences;
For each B in BB list in R (in topological order w/o back edge) {
    CandInsts = ready instructions in NonSpeculative(B) U Speculative(B)
    // All the incoming dependences have already been scheduled
    For t slots until all the instructions in B has been scheduled {
        For n in CandInsts in priority order {
            // Prioritize non-speculative over speculative
            if (ok to move n to B && n has no resource conflicts @t) {
                // OK: do not speculatively move exception & store ...
                // Same as the List Scheduling
                Schedule the inst to (B, t)
                Update resource commitments & dependences
            }
        }
    }
}
Update CandInsts // some insts may become ready!
}
```

Scheduling Example

- **Machine Model:**
 - Fully-pipelined execution path
 - LD takes two cycles + Others take one cycle
 - Two parallel general-execution path
- **Priority order: B1, B2, B3**
- **Control equivalence: {B1, B3}, {B2}**
- **Non-speculative(B1) = {B1, B3}**
- **Speculative(B1) = {B2}**
- **Candidates = {i1, i3, i5}**

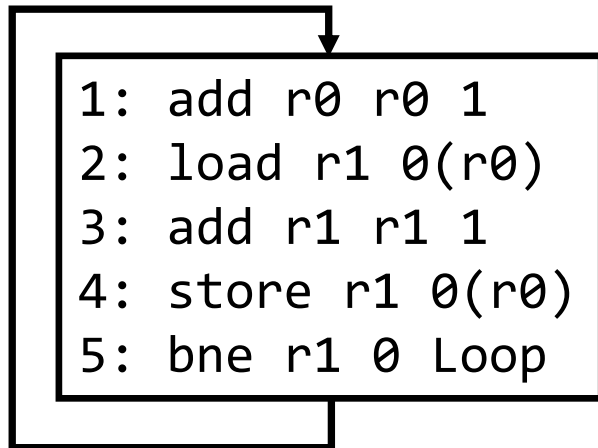


Scheduling Example



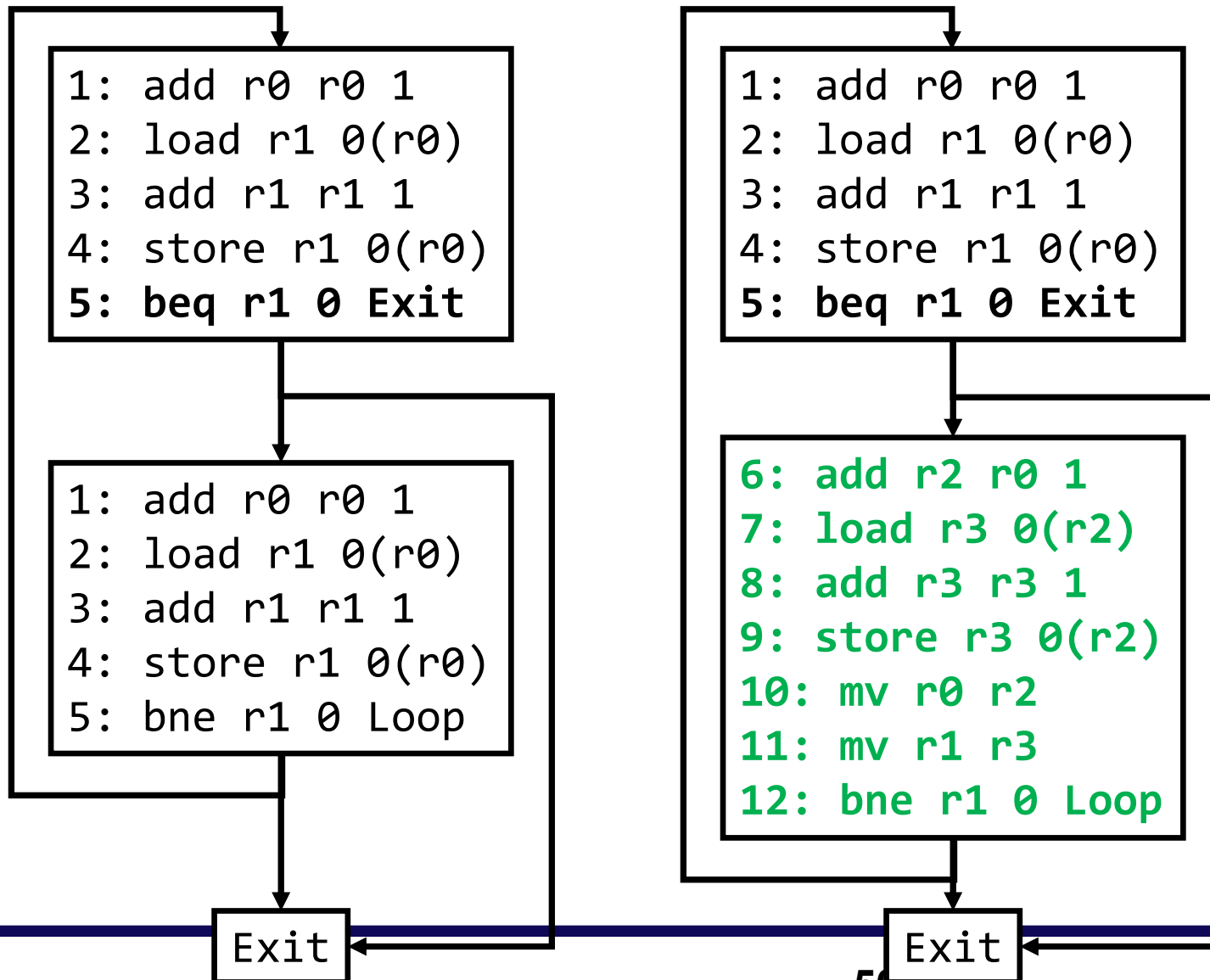
Unrolling and Instruction Scheduling - 1

- Assumption:
 - Two general purpose pipelines
 - (mem + alu take two cycles) & (branch & copy take one cycle)
- Unrolling enables a new scheduling opportunities



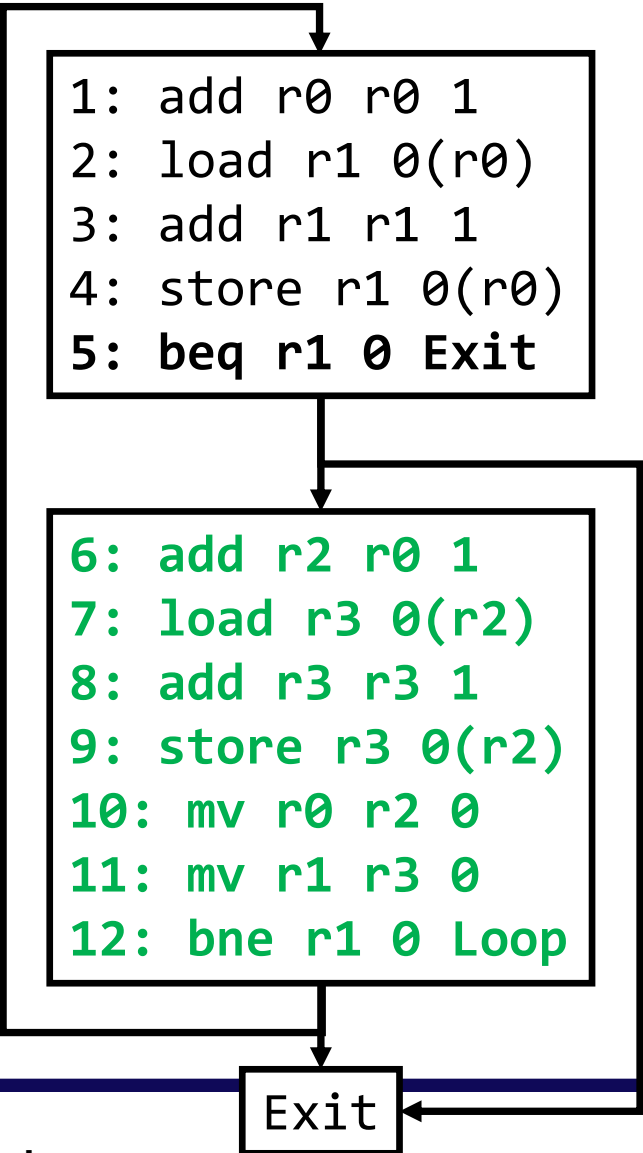
Path0	Path1
	1
2	
	3
4	5

Unrolling and Instruction Scheduling - 2



**Register renaming
to remove false
dependencies**

Unrolling and Instruction Scheduling - 2



Path0	Path1
1: add r0 r0 1	-
2: load r1 0(r0)	6: add r2 r0 1
10: mv r0 r2	
3: add r1 r1 1	7: load r3 0(r2)
4: store r1 0(r0)	5: beq r1 0 Exit
8: add r3 r3 1	
11: mv r1 r3	9: store r3 0(r2)
12: bne r1 0 Loop	

