12. Instruction Scheduling

2025 Fall

Hunjun Lee

Hanyang University

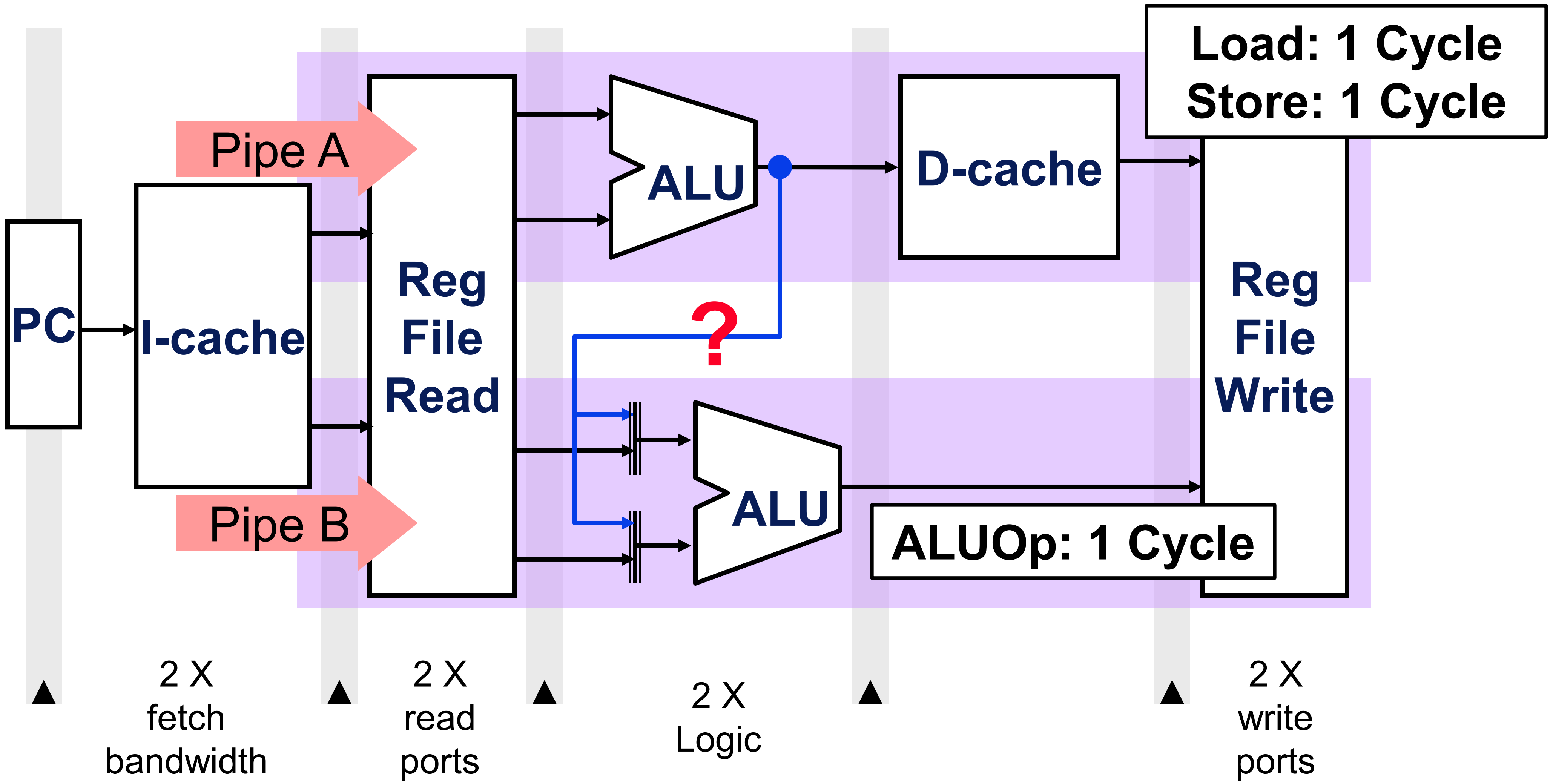
Instruction Scheduling

- There are some opportunities to improve the performance using scheduling and reordering

- The instruction scheduling should not affect the functionality

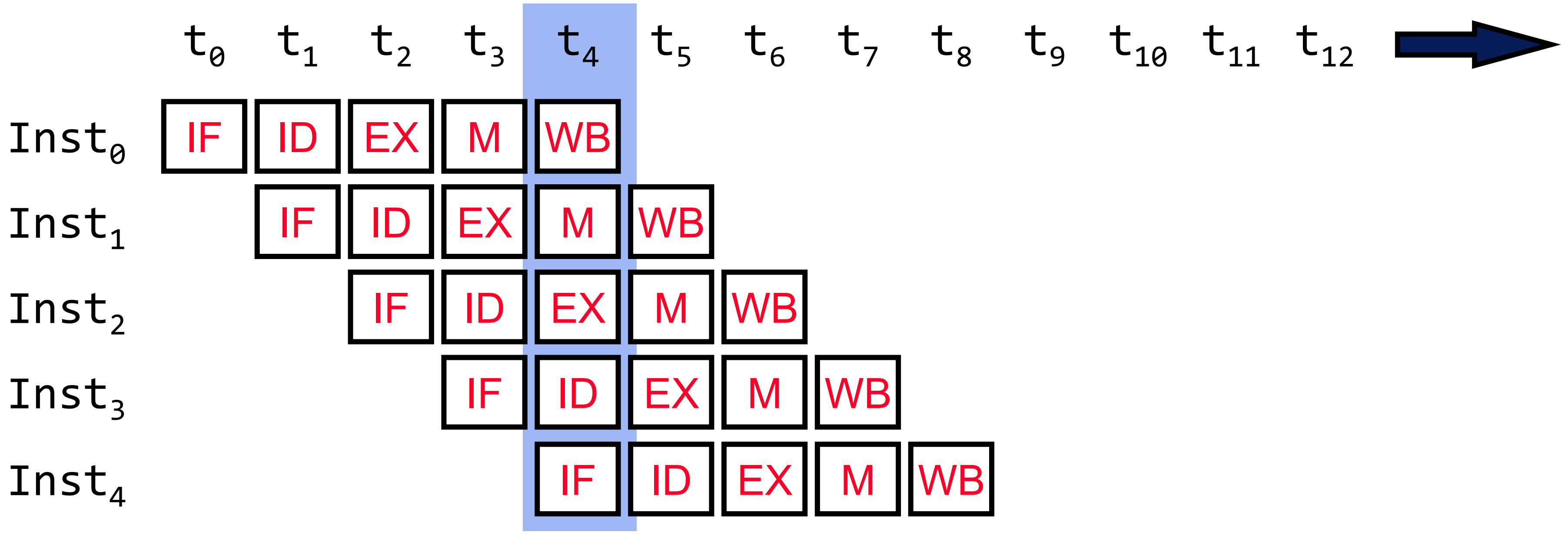
- We cannot reschedule all the instructions due to the dependencies

CPU Microarchitecture (Superscalar)



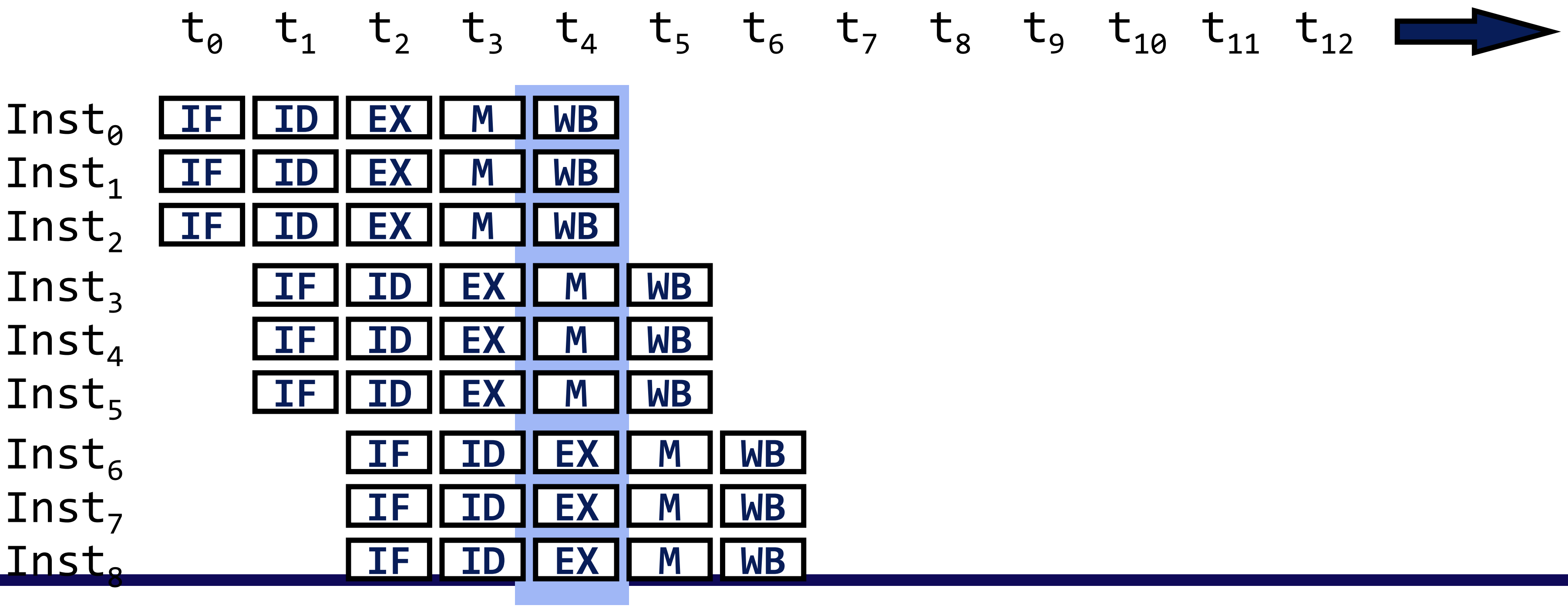
Pipeline + SuperScalar

- **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
```

$\curvearrowright$ 1 cycle stall

Hazards in the dual-issue CPU

- **It also suffers from false dependencies**
- **Write after read hazard**
  - You cannot place two instruction with WAR hazard

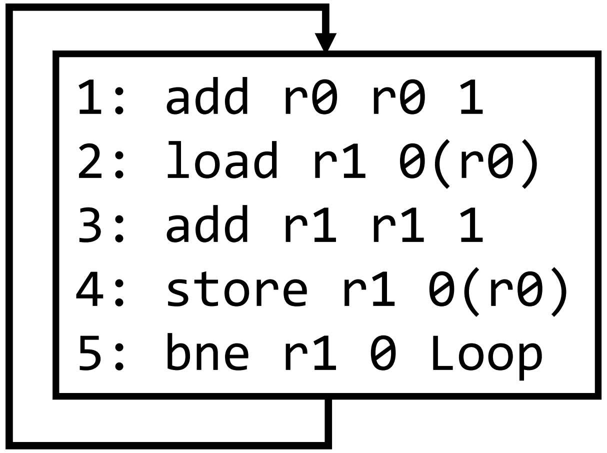
```
add  $t0, $s0, $s1
load $s0, 0($t1)
```

- **Write after write hazard**
  - The two packed instructions cannot write to the same register

```
load $t0, 0($s0)
add  $t0, $t1, $s1
```

Unrolling and Instruction Scheduling - 1

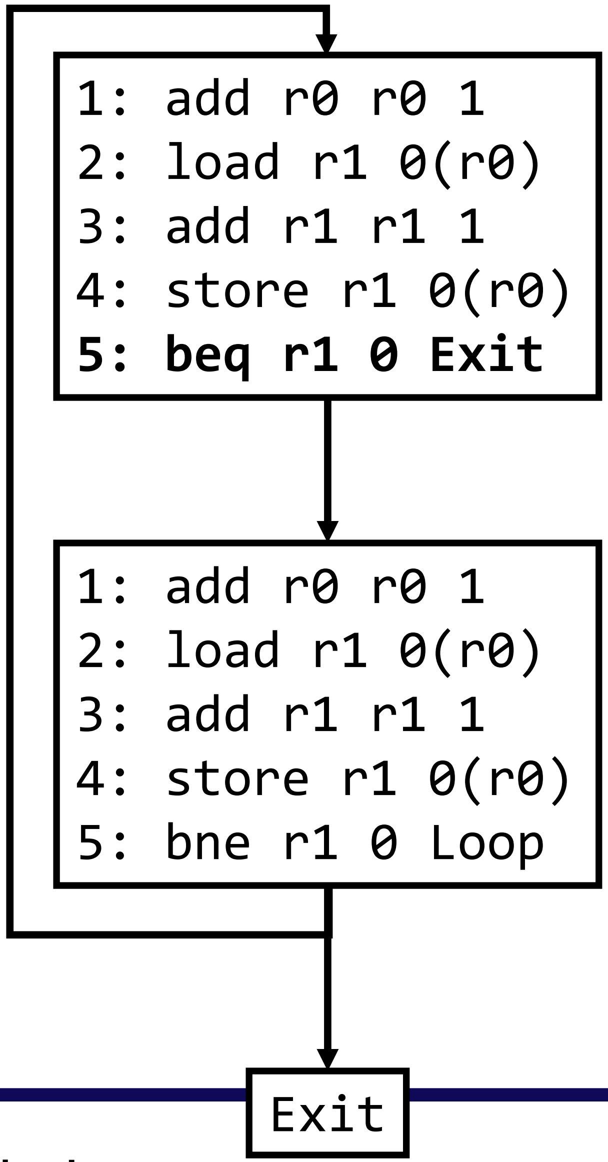
- Assumption:
  - two memory pipeline + two ALU (+branch) pipeline
  - (copy takes one cycle & others take two cycles)
- Unrolling enables a new scheduling opportunities



```
1: add r0 r0 1
2: load r1 0(r0)
3: add r1 r1 1
4: store r1 0(r0)
5: bne r1 0 Loop
```

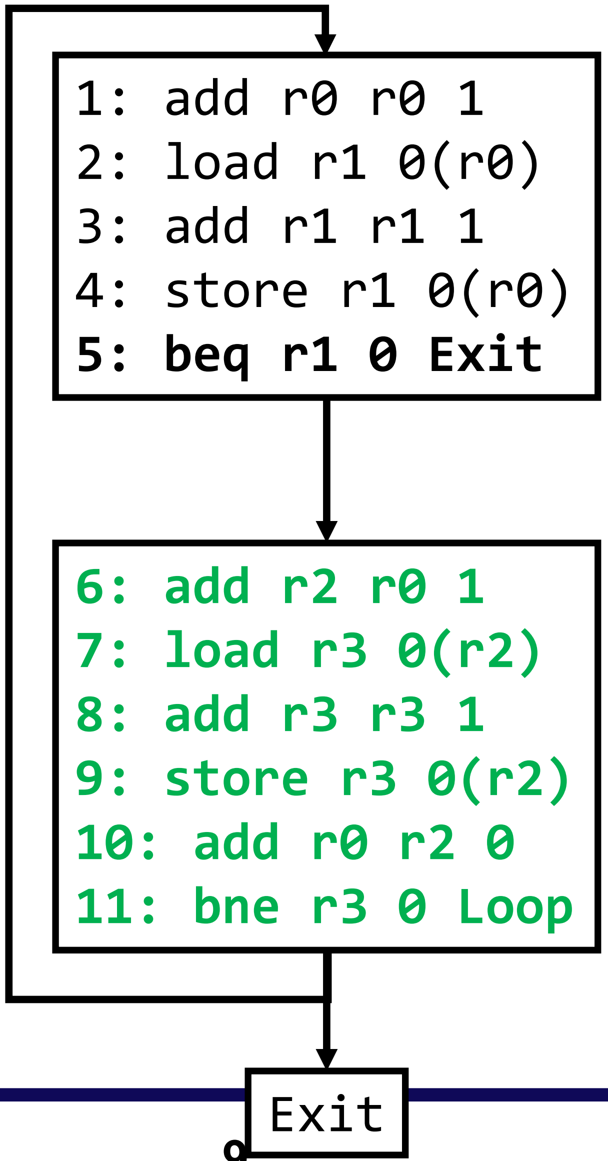
MEM0	MEM1	ALU0	ALU1
		1	
2			
		3	
4		5	

Unrolling and Instruction Scheduling - 2



```
1: add r0 r0 1
2: load r1 0(r0)
3: add r1 r1 1
4: store r1 0(r0)
5: beq r1 0 Exit
```

```
1: add r0 r0 1
2: load r1 0(r0)
3: add r1 r1 1
4: store r1 0(r0)
5: bne r1 0 Loop
```

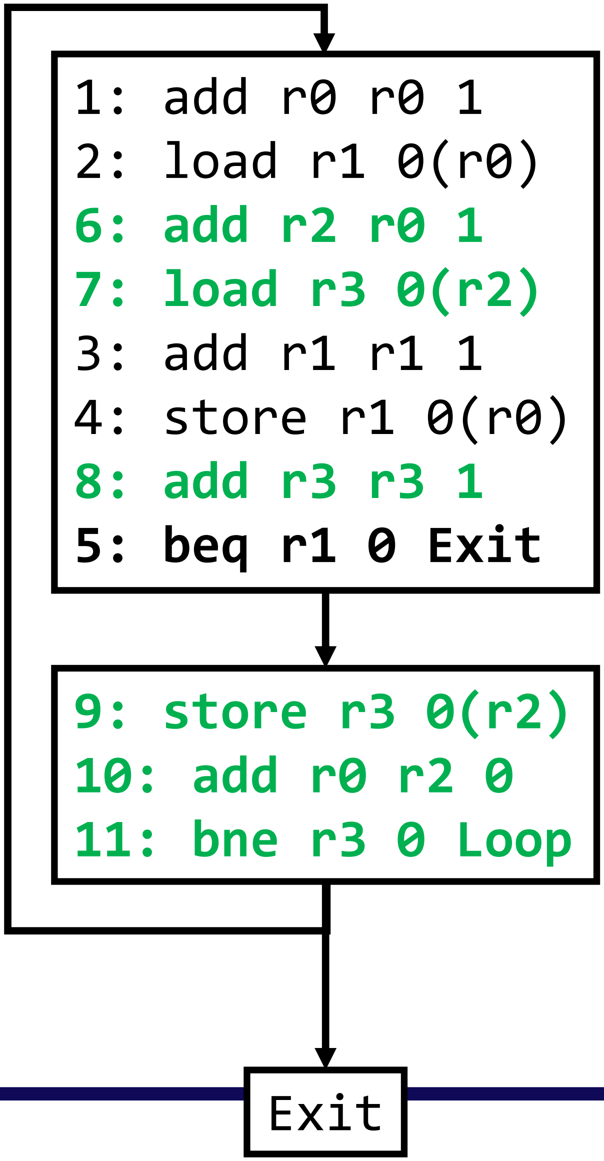


```
1: add r0 r0 1
2: load r1 0(r0)
3: add r1 r1 1
4: store r1 0(r0)
5: beq r1 0 Exit
```

```
6: add r2 r0 1
7: load r3 0(r2)
8: add r3 r3 1
9: store r3 0(r2)
10: add r0 r2 0
11: bne r3 0 Loop
```

MEM0	MEM1	ALU0	ALU1
		1	
2		6	
7		3	
4		5	8
9		10	11

Unrolling and Instruction Scheduling - 2



```
1: add r0 r0 1
2: load r1 0(r0)
6: add r2 r0 1
7: load r3 0(r2)
3: add r1 r1 1
4: store r1 0(r0)
8: add r3 r3 1
5: beq r1 0 Exit
```

```
9: store r3 0(r2)
10: add r0 r2 0
11: bne r3 0 Loop
```

Assumption!: registers are dead after the loop

MEM0	MEM1	ALU0	ALU1
		1	
2		6	
7		3	
4		5	8
9		10	11

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 → 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

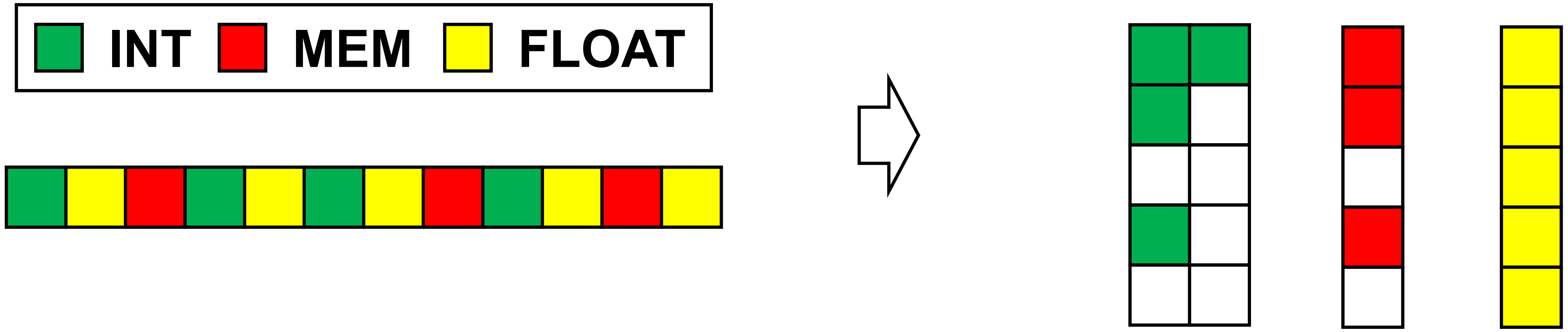
Operation Latency: 1

Issuing Rate: **N**

Superscalar Degree: **N**

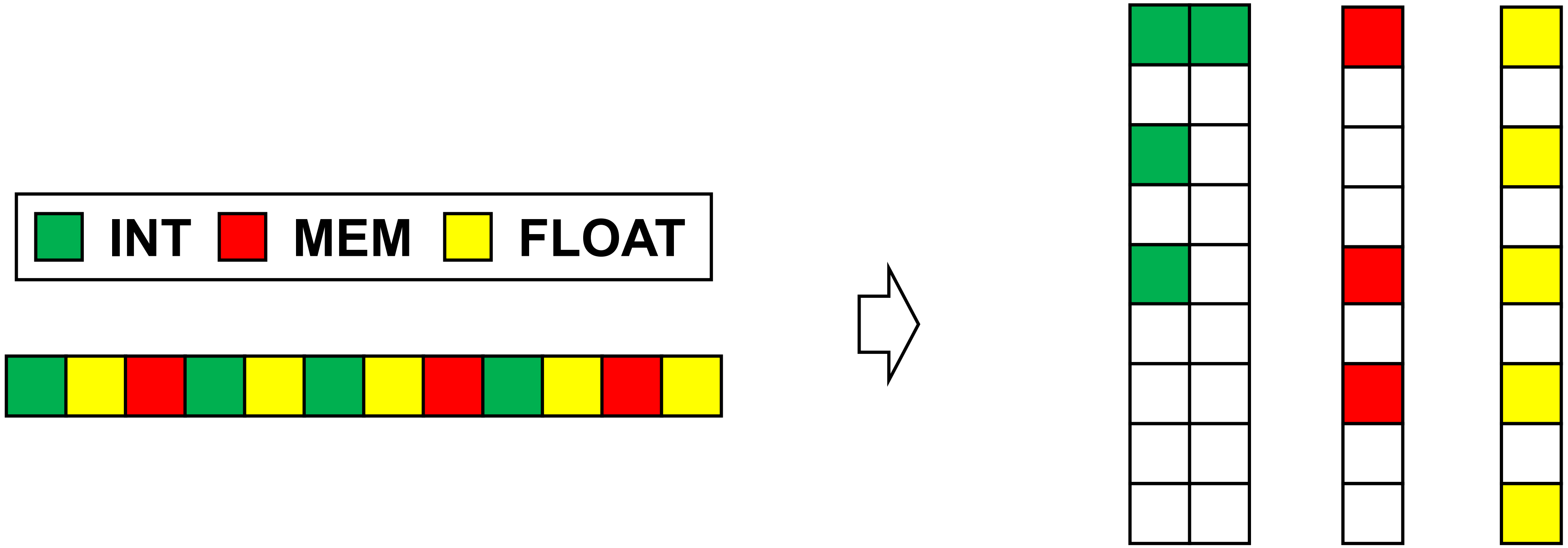
Limited FUs per Inst. Type

- **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
```

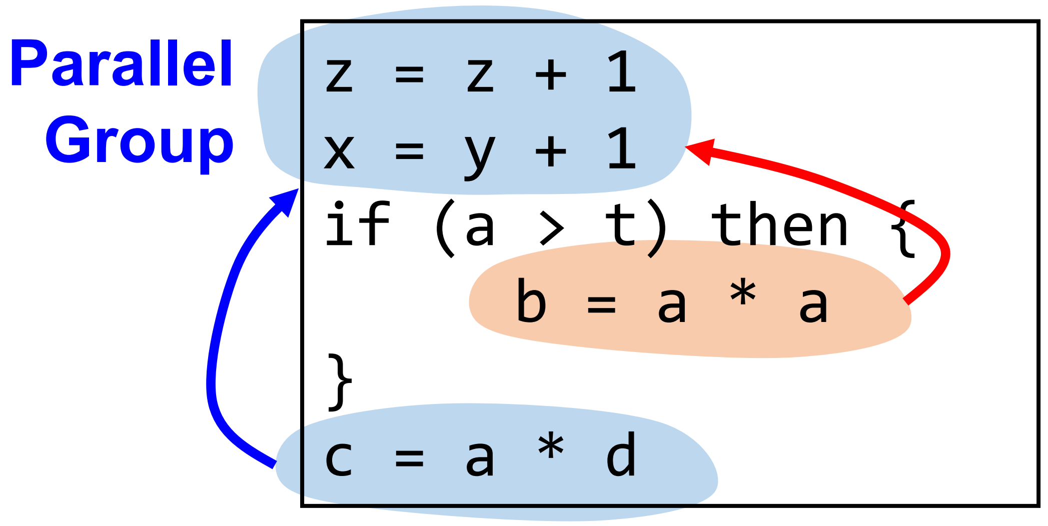
Control Dependency

Parallel Group

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

Control Dependence

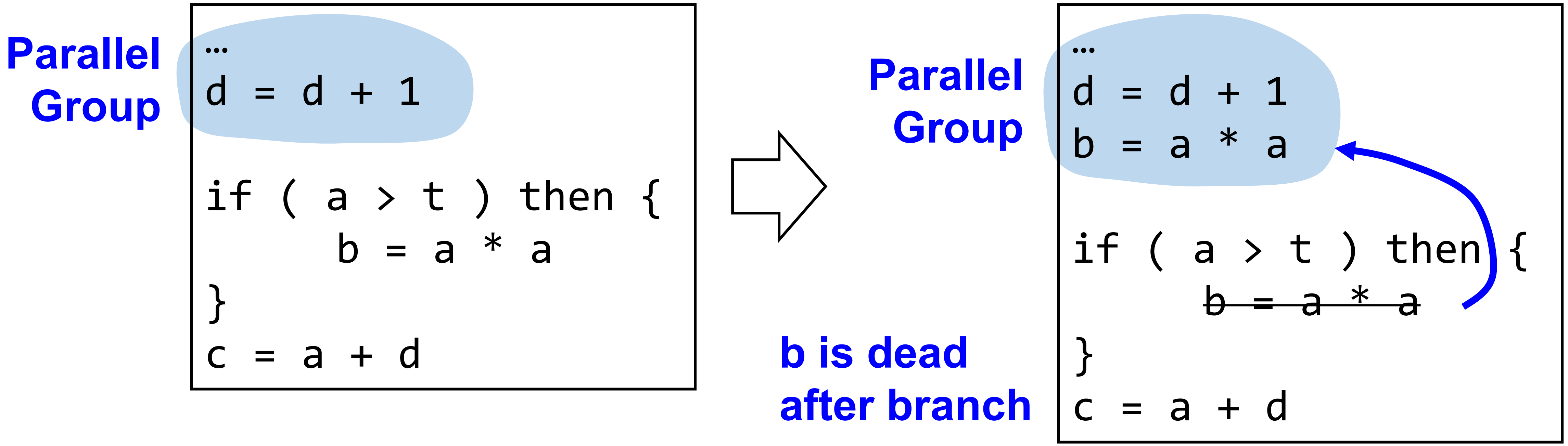
- **We cannot parallelize instructions when there is a control dependence**
  - We cannot move instruction inside the branch `b = a * a`



```
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

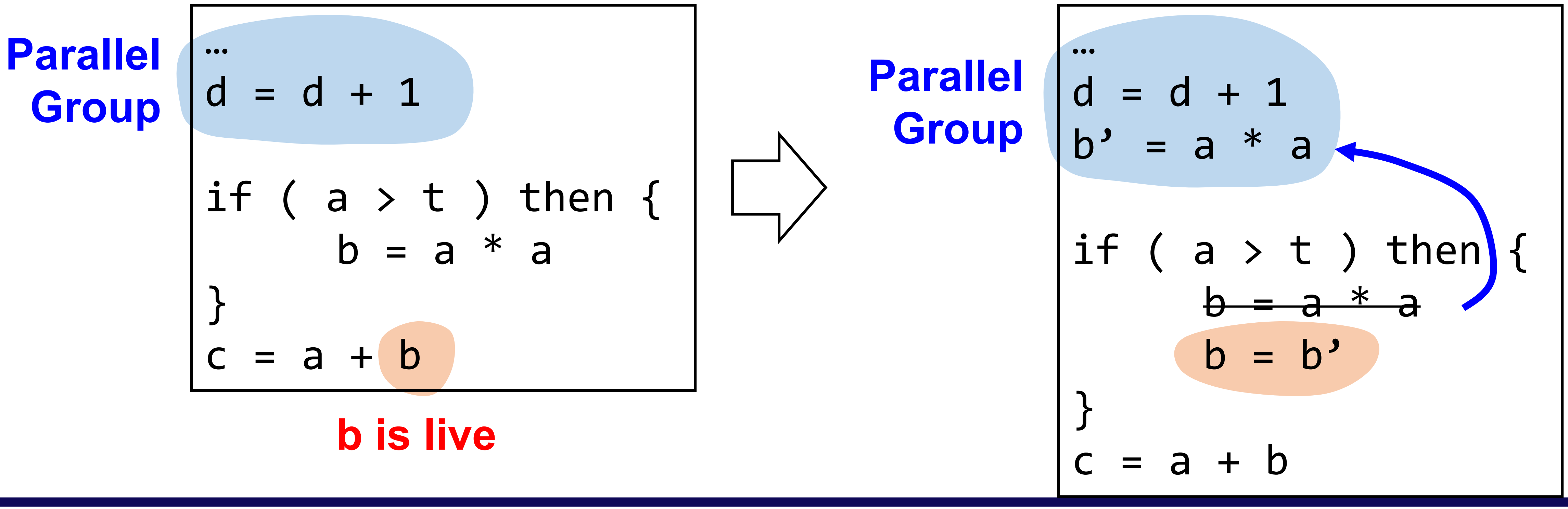


```
…
d = d + 1

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

```
…
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)



```
…
d = d + 1

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

```
…
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 → read
  - Anti dependence: read → write
  - Output dependence: write → write

- **We cannot move instructions if there is a data dependence**

```
r1 = r4 + r5
r2 = r1 + 1
```

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
```

$\Rightarrow$

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
```

$\Rightarrow$

```
…
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

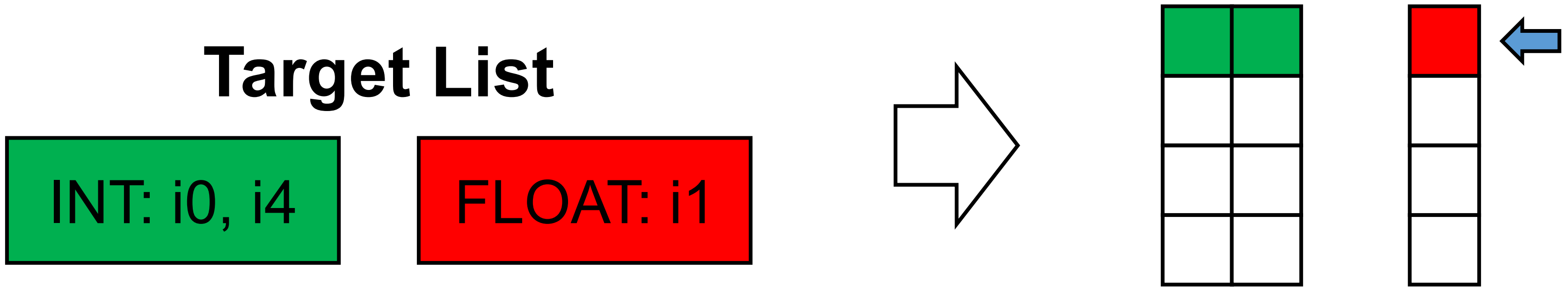
- **Input:**
  - Data Precedence Graph (DPG): The graph structure of the instructions according to the dependences between instructions
  - Machine Parameters: The available resources and execution latency …

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

List Scheduling - 2

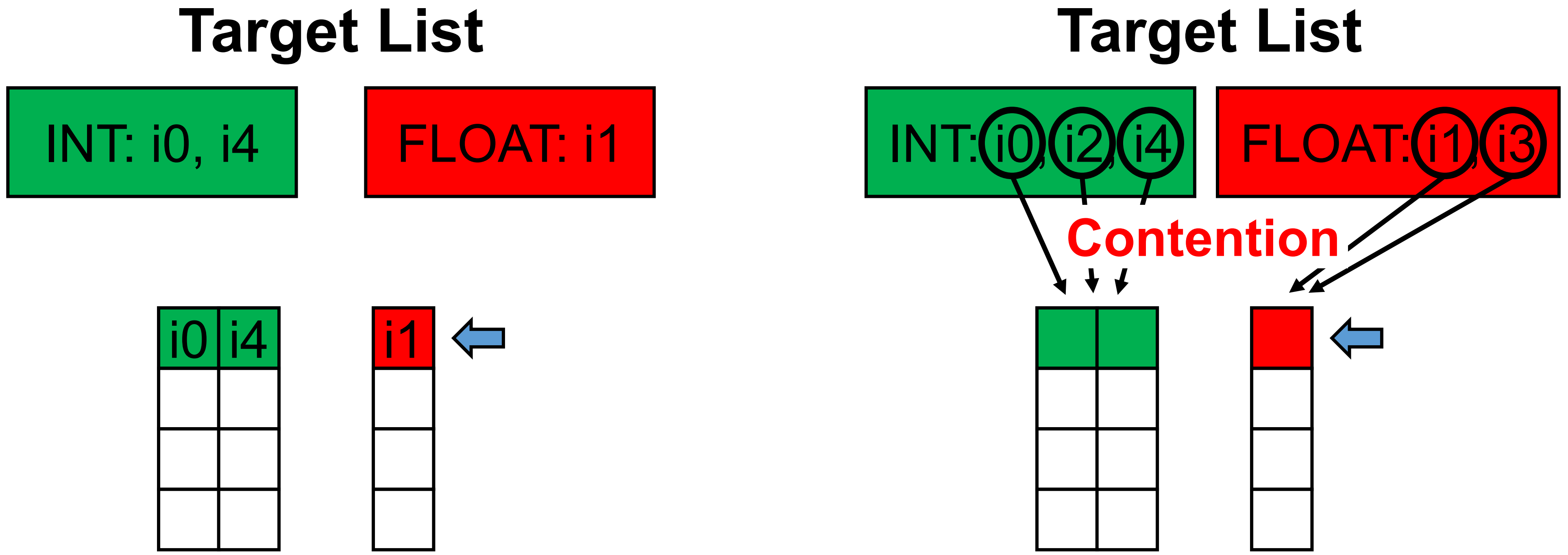
- **There is a list to keep the list of ready instructions**
  - Req #1. All the operands are ready to execute (and no false dependences)
  - Req #2. The target resources are available

- **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

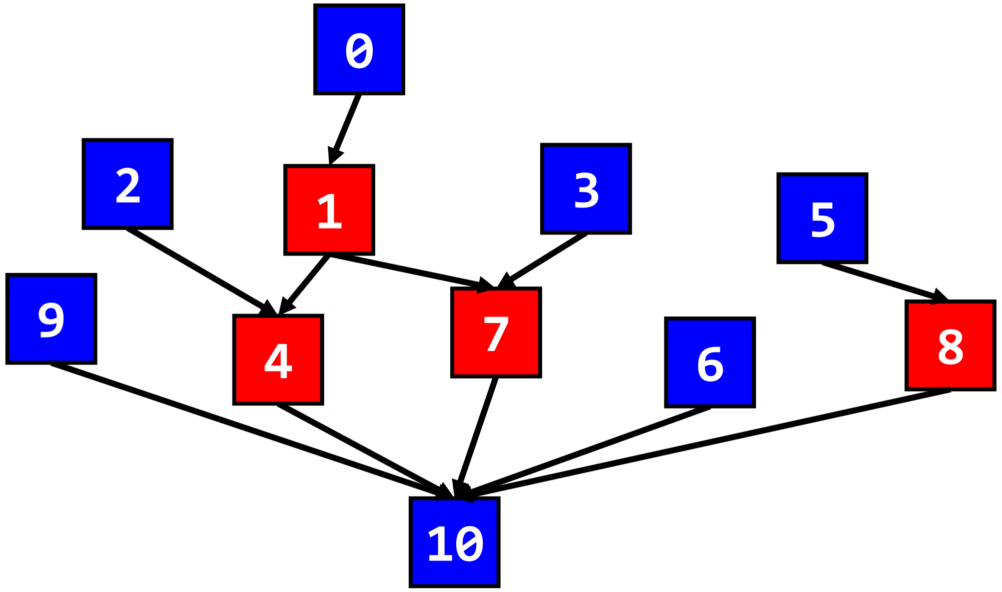


Data Precedence Graph (DPG) - 1

- **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



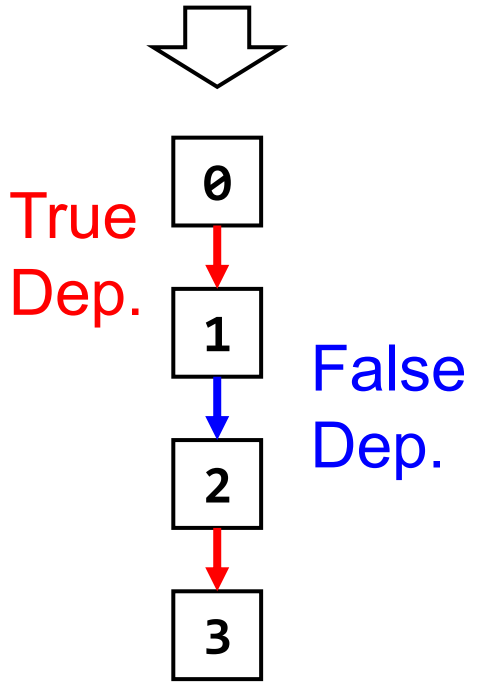
Data Precedence Graph (DPG) - 2

- **There are two types of edges in the DPG**
  - True dependency
  - False dependency

- **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!)

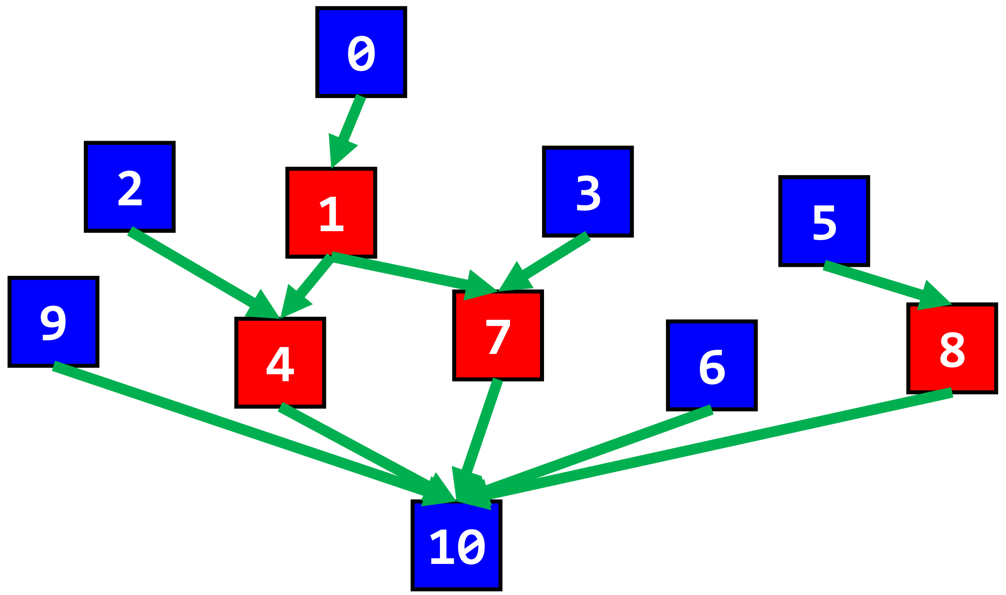
```
0: x = 1
1: y = x
2: x = 2
3: z = x
```



Determining Priorities in Contention

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

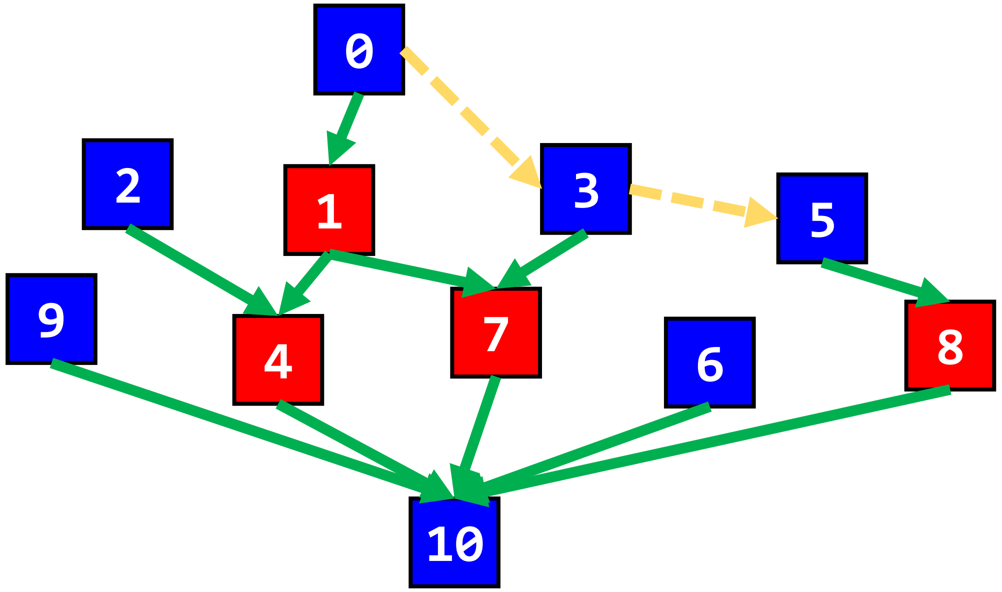
$$priority(x) = latency(x) + max_{(x,y)\in E}\left(priority(y)\right)$$



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)

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



List Scheduling Algorithm

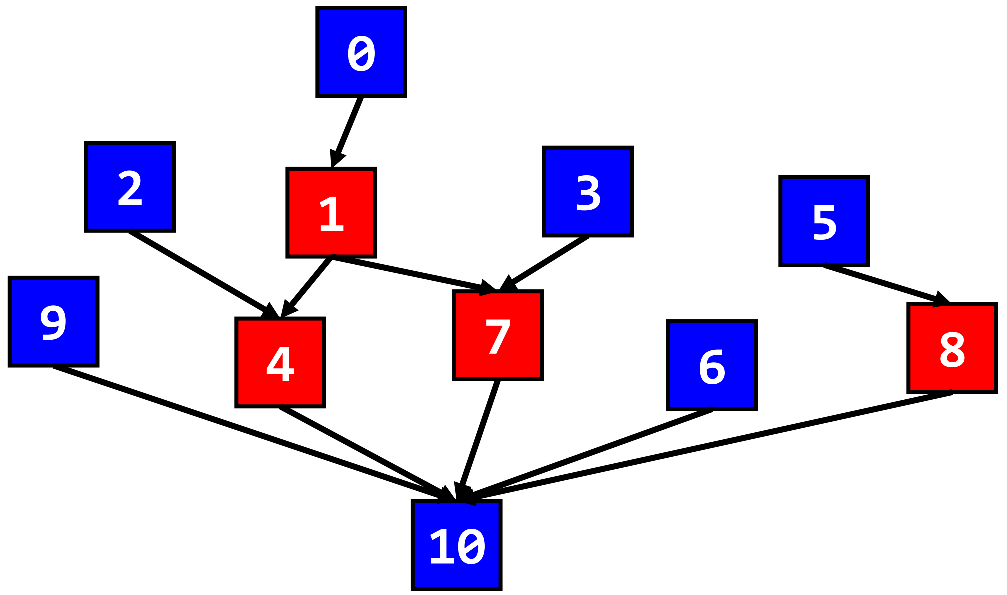
```
cycle = 0
ready-list = root nodes in DPG              // Indicates the ready list
inflight-list = {}                          // Indicates the executing instructions (at the pipeline)

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?

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



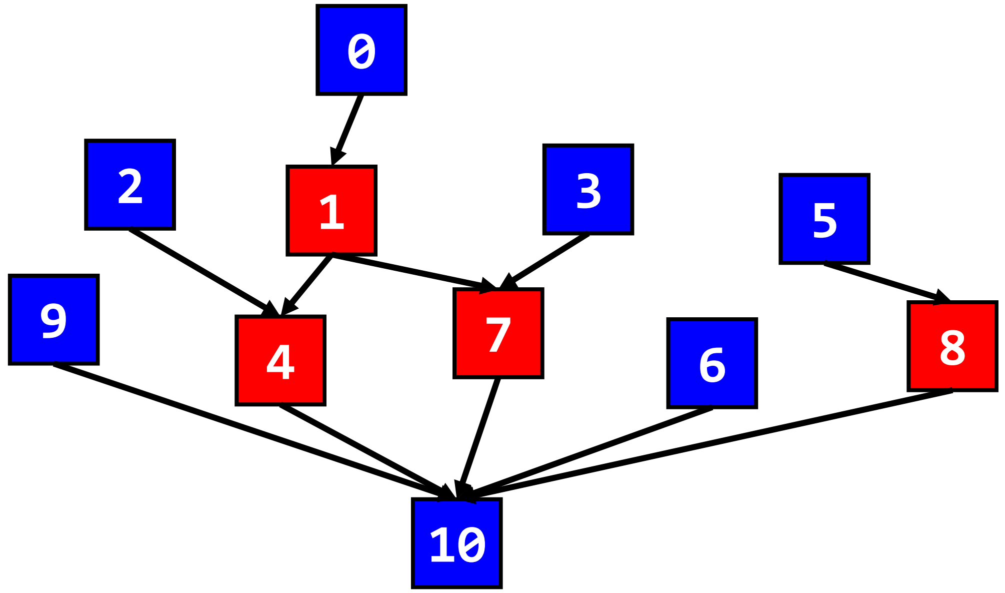
Add takes two cycles;
Others take one cycles

		Ready List
0	2	0, 2, 3, 5, 6, 9
1	3	1, 3, 5, 6, 9
5	6	5, 6, 9
4	7	4, 7, 8, 9
8	9	8, 9
10		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?

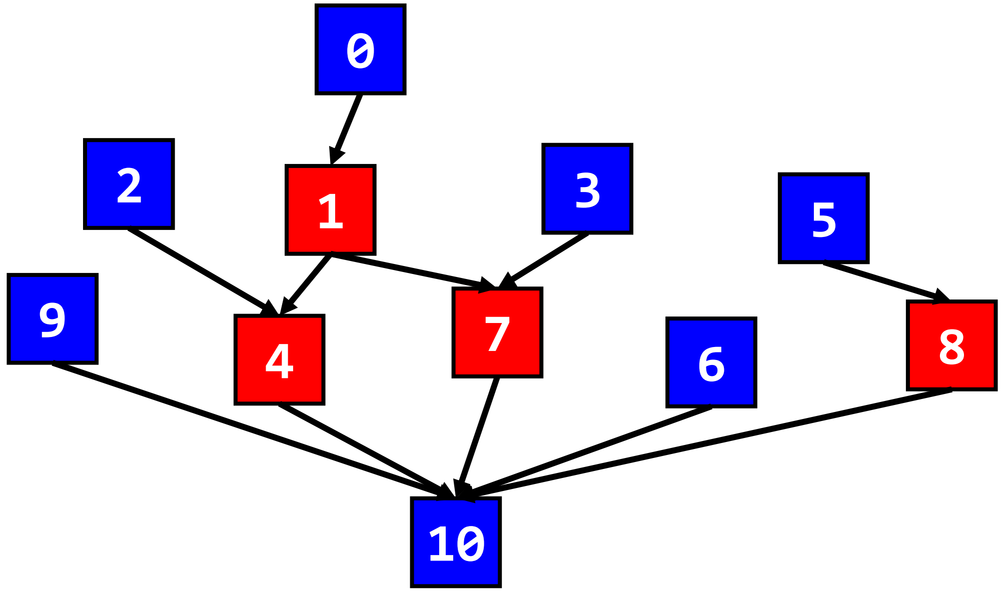


Add takes two cycles;
Others take one cycles

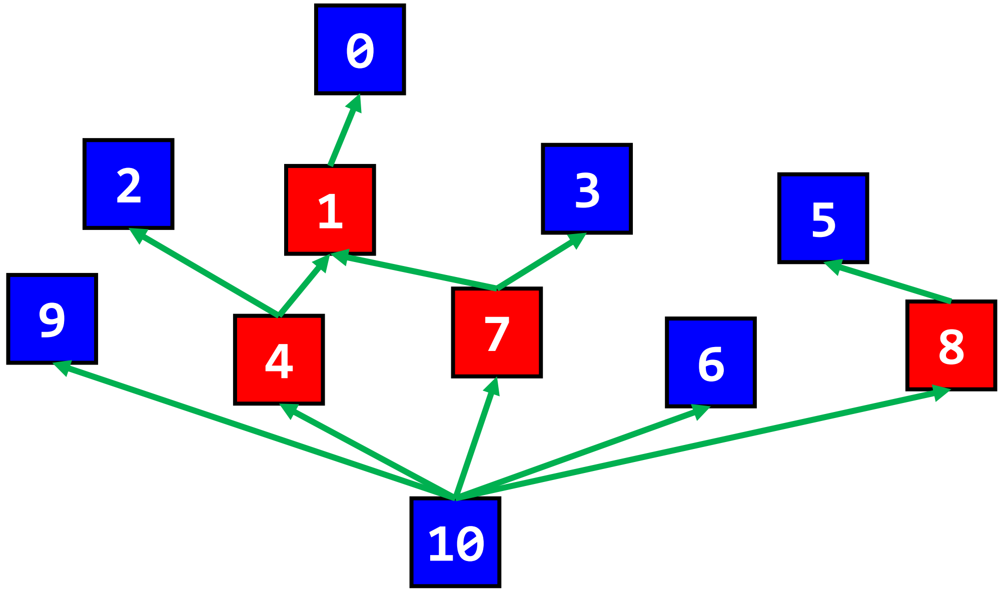
		Ready List
0	2	0, 2, 3, 5, 6, 9
1	5	1, 3, 5, 6, 9
3	8	3, 6, 8, 9
4	7	4, 6, 7, 9
6	9	6, 9
10		10

Alternative Approach

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



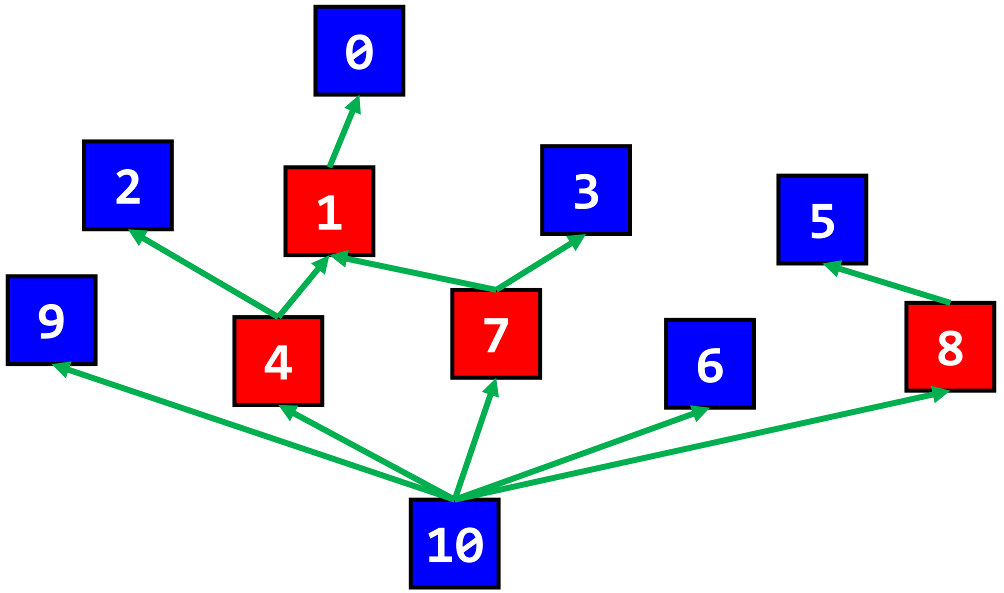
Forward Scheduling



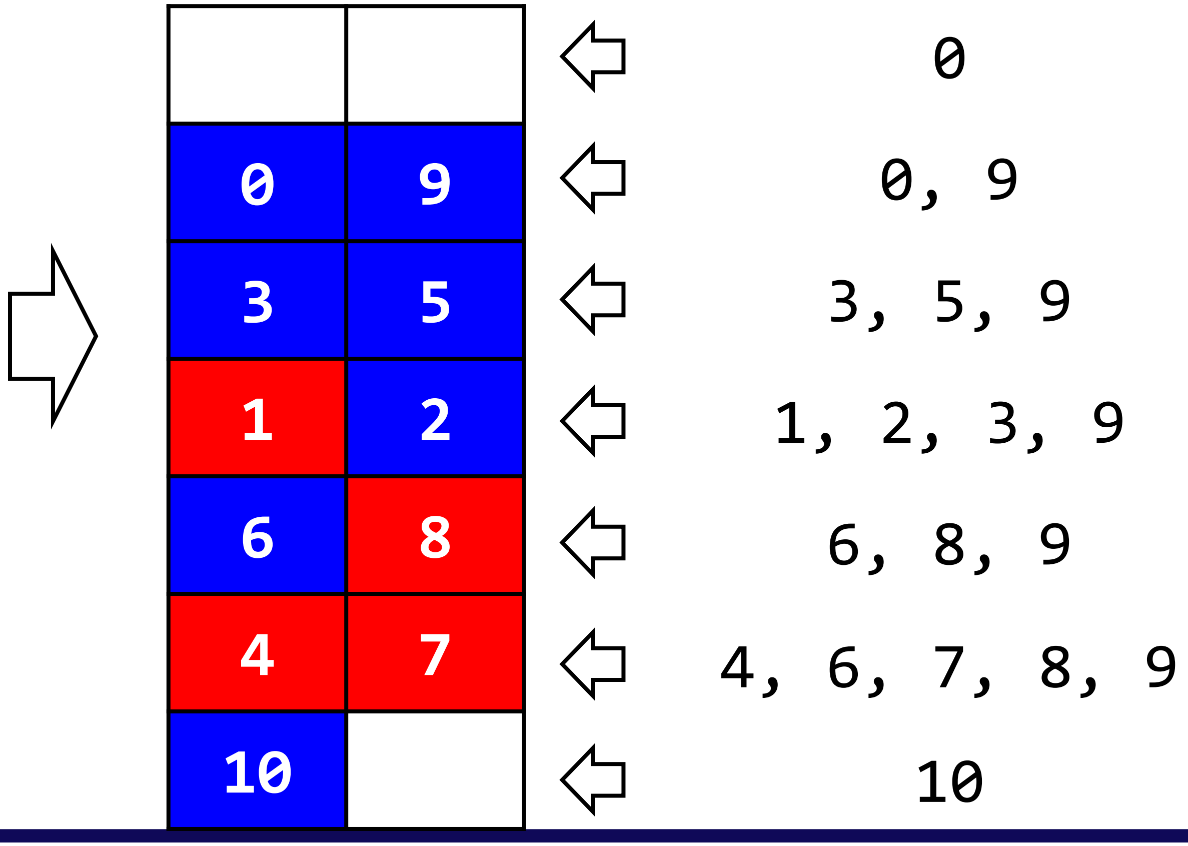
Backward Scheduling

Backward List Scheduling

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

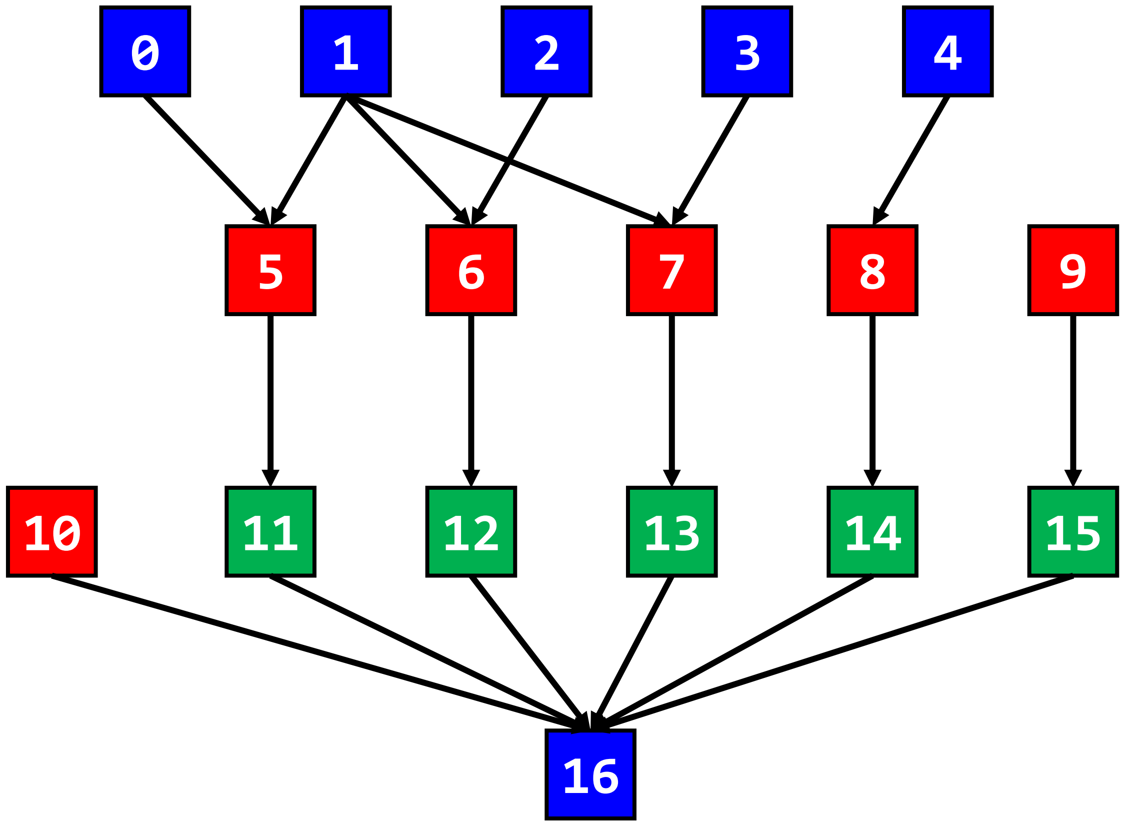


Add takes two cycles;
Others take one cycles



List Scheduling Examples

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

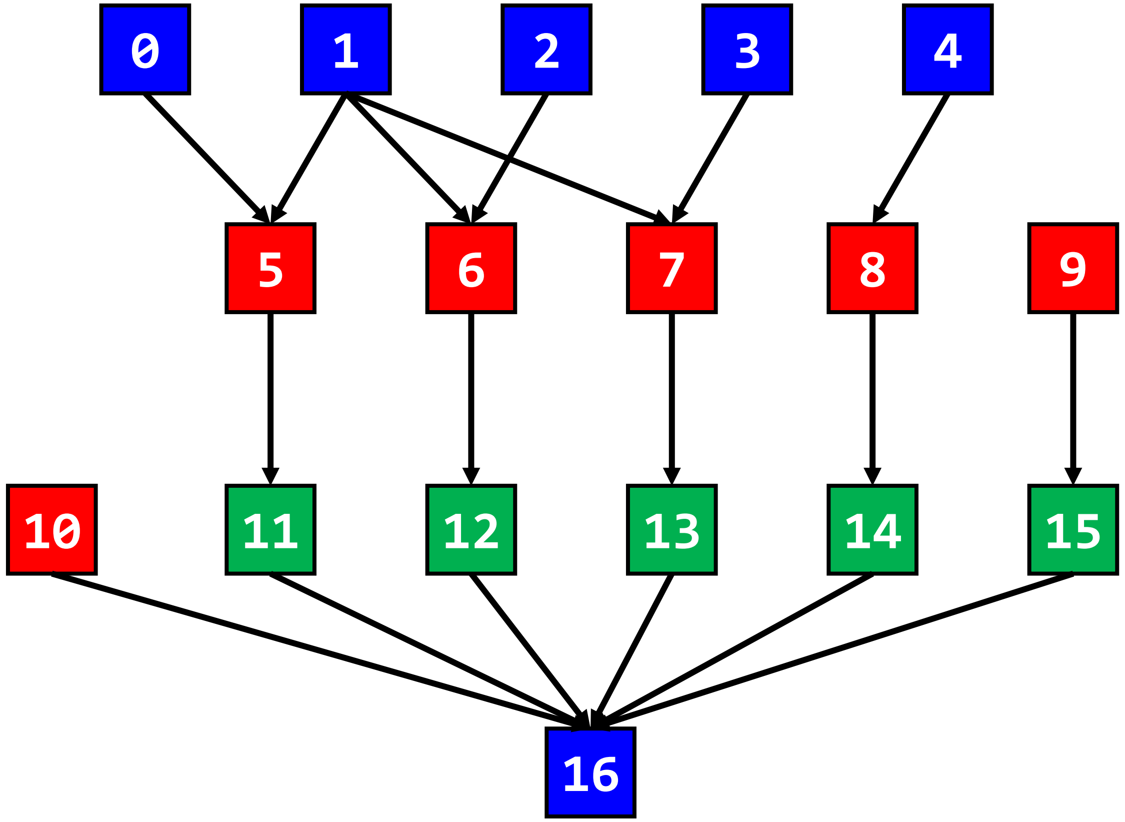
-

-

16

List Scheduling Examples

Assume pipelined HW



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

Advanced Approaches

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