Instruction Scheduling

Advanced Compiler Techniques
2005
Erik Stenman
Virtutech

Simple Machine Model

- Instructions are executed in sequence.
  - Fetch, decode, execute, store results.
  - One instruction at a time.
- For branch instructions, start fetching from a different location if needed.
  - Check branch condition.
  - Next instruction may come from a new location given by the branch instruction.

Simple Execution Model

5 Stage pipe-line:

Cycle:	1	2	3	4	5
	fetch	**decode**	**execute**	**memory**	**write back**

Fetch: get the next instruction.

Decode: figure out what that instruction is.

Execute: perform ALU operation.

- address calculation in a memory op

Memory: do the memory access in a mem. op.

Write Back: write the results back.

Execution Models

time (cycles) →

Model 1

Inst 1	IF	DE	EXE	MEM	WB					
Inst 2						IF	DE	EXE	MEM	WB

One instruction finish every 5 cycles.

Model 2

Cycle:	1	2	3	4	5	6	7	8	9	10
Inst 1	IF	DE	EXE	MEM	WB					
Inst 2		IF	DE	EXE	MEM	WB				
Inst 3			IF	DE	EXE	MEM	WB			
Inst 4				IF	DE	EXE	MEM	WB		
Inst 5					IF	DE	EXE	MEM	WB	

One instruction finish every cycle.

Handling Branch Instructions

Problem: We do not know the location of the next instruction until later.

- after DE in jump instructions
- after EXE in conditional branch instructions

Branch	IF	DE	EXE	MEM	WB			
???		IF	DE	EXE	MEM	WB		
???			IF	DE	EXE	MEM	WB	
Inst				IF	DE	EXE	MEM	WB

What to do with the middle 2 instructions?

Handling Branch Instructions

What to do with the middle 2 instructions?

1. Stall the pipeline in case of a branch until we know the address of the next instruction:
   - wasted cycles

Cycle:	1	2	3	4	5	6	7	8
Branch	IF	DE	EXE	MEM	WB			
		IF Empty	DE Empty	EXE Empty	MEM Empty	WB Empty		
			IF Empty	DE Empty	EXE Empty	MEM Empty	WB Empty	
Next inst				IF	DE	EXE	MEM	WB

Handling Branch Instructions

What to do with the middle 2 instructions?

2. Delay the action of the branch

- Make branch affect only after two instructions
- Following two instructions after the branch get executed regardless of the branch

Branch	IF	DE	EXE	MEM	WB			
Next seq inst		IF	DE	EXE	MEM	WB		
Next seq inst			IF	DE	EXE	MEM	WB	
Branch target inst				IF	DE	EXE	MEM	WB

Branch Delay Slot(s)

MIPS has a branch delay slot

- The instruction after a conditional branch gets executed even if the code branches to target
- Fetching from the branch target takes place only after that

```
ble r3, foo
```

[ ] Branch delay slot

What instruction to put in the branch delay slot?

Filling the Branch Delay Slot

Simple Solution: Put a no-op.

Wasted instruction, just like a stall.

```
ble r3, lbl
nop
```

Branch delay slot

Filling the Branch Delay Slot

Move an instruction from above the branch.

~~`prev_instr`~~

```
ble r3, lbl
prev_instr
```

Branch delay slot

- Moved instruction executes iff branch executes.
  - Get the instruction from the same basic block as the branch.
  - Don’t move a branch instruction!
- Instruction need to be moved over the branch.
  - Branch does not depend on the result of the inst.

Filling the Branch Delay Slot

Move an instruction dominated by the branch instruction.

```
ble r3, lbl
dom_instr
```

Branch delay slot

```
lbl:
```

~~dom_instr~~

Filling the Branch Delay Slot

Move an instruction from the branch target.

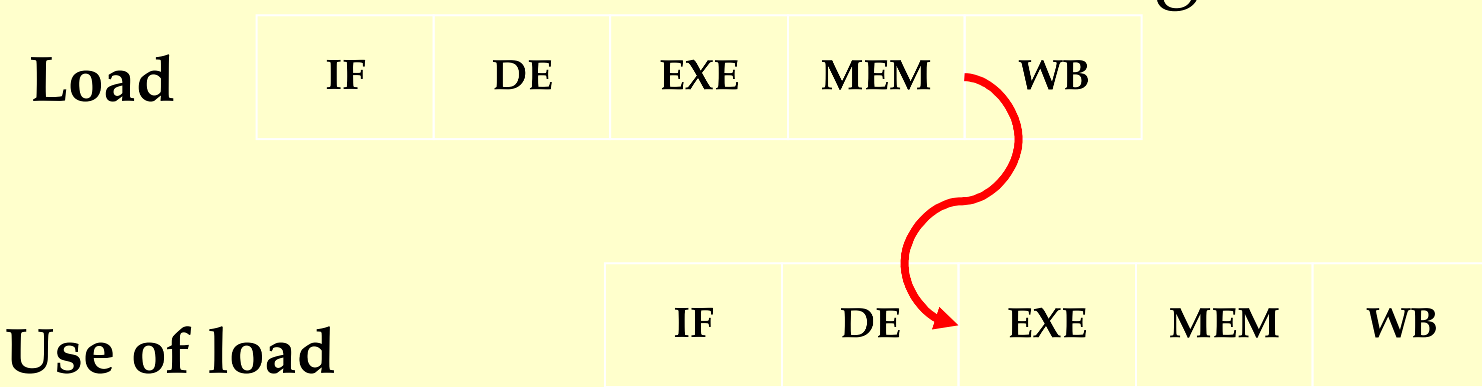
- Instruction dominated by target.
- No other ways to reach target (if so, take care of them).
- If conditional branch, instruction should not have a lasting effect if the branch is not taken.

```
ble r3, lbl
instr            Branch delay slot


lbl:
instr
```

Load Delay Slots

Problem: Results of the loads are not available until end of MEM stage



If the value of the load is used…what to do??

Load Delay Slots

If the value of the load is used…what to do?

Always stall one cycle.

- Stall one cycle if next instruction uses the value.
  - Need hardware to do this.
- Have a delay slot for load.
  - The new value is only available after two instructions.
  - If next inst. uses the register, it will get the old value.

Load	**IF**	**DE**	**EXE**	**MEM**	**WB**		
???		**IF**	**DE**	**EXE**	**MEM**	**WB**	
Use of load			**IF**	**DE**	**EXE**	**MEM**	**WB**

Example

```
r2 = *(r1 + 4)
r3 = *(r1 + 8)
r4 = r2 + r3
r5 = r2 - 1
goto L1
```

Example

```
r2 = *(r1 + 4)
r3 = *(r1 + 8)
noop
r4 = r2 + r3
r5 = r2 - 1
goto L1
noop
```

Assume 1 cycle delay on branches and 1 cycle latency for loads

Example

```
r2 = *(r1 + 4)
r3 = *(r1 + 8)
noop
r4 = r2 + r3
r5 = r2 - 1
goto L1
noop
```

Example

```
r2 = *(r1 + 4)
r3 = *(r1 + 8)
r5 = r2 - 1
r4 = r2 + r3

goto L1
noop
```

Example

```
r2 = *(r1 + 4)
r3 = *(r1 + 8)
r5 = r2 - 1


goto L1
r4 = r2 + r3
```

Example

```
r2 = *(r1 + 4)
r3 = *(r1 + 8)
r5 = r2 - 1
goto L1
r4 = r2 + r3
```

Final code after delay slot filling

From a Simple Machine Model to a Real Machine Model

- Many pipeline stages.
  - MIPS R4000 has 8 stages.
- Different instructions take different amount of time to execute.
  - mult 10 cycles
  - div 69 cycles
  - ddiv 133 cycles
- Hardware to stall the pipeline if an instruction uses a result that is not ready.

Real Machine Model cont.

- Most modern processors have multiple execution units (superscalar).
  - If the instruction sequence is correct, multiple operations will take place in the same cycles.
  - Even more important to have the right instruction sequence.

Instruction Scheduling

Goal: Reorder instructions so that pipeline stalls are minimized.

Constraints on Instruction Scheduling:

- Data dependencies.
- Control dependencies .
- Resource constraints.

Data Dependencies

- If two instructions access the same variable, they can be dependent.
- Kinds of dependencies:
  - True: write → read. (Read After Write, RAW)
  - Anti: read → write. (Write After Read, WAR)
  - Anti (Output): write → write. (Write After Write, WAW)
- What to do if two instructions are dependent?
  - The order of execution cannot be reversed.
  - Reduce the possibilities for scheduling.

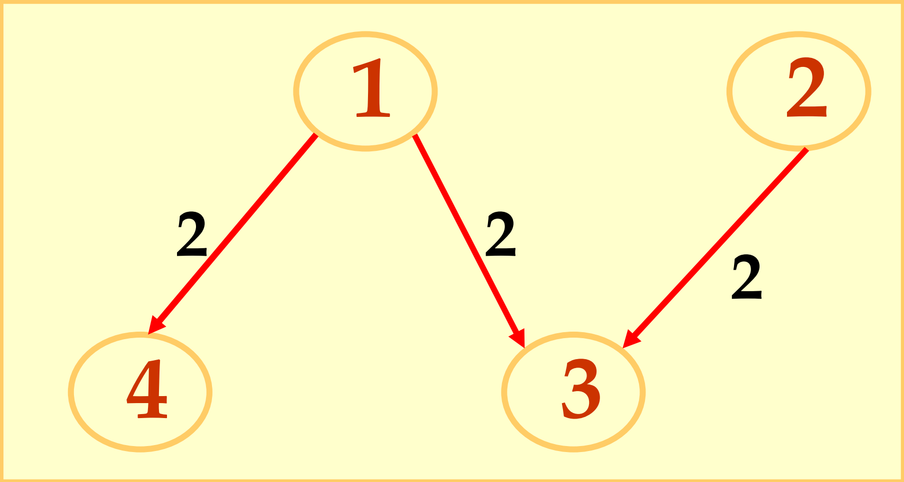
Computing Data Dependencies

- For basic blocks, compute dependencies by walking through the instructions.
- Identifying register dependencies is simple.
  - is it the same register?
- For memory accesses.
  - simple: base + offset1 ?= base + offset2
  - data dependence analysis: a[2i] ?= a[2i+1]
  - interprocedural analysis: global ?= parameter
  - pointer alias analysis: p1 ?= p

Representing Dependencies

- Using a dependence DAG, one per basic block.
- Nodes are instructions, edges represent dependencies.

```
1: r2 = *(r1 + 4)
2: r3 = *(r1 + 8)
3: r4 = r2 + r3
4: r5 = r2 - 1
```

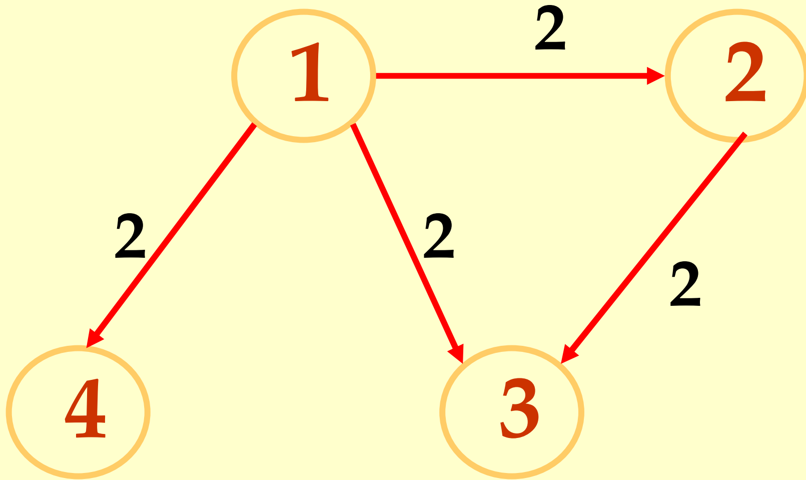


Edge is labeled with latency:

$v(i \rightarrow j)$ = delay required between initiation times of i and j minus the execution time required by i.

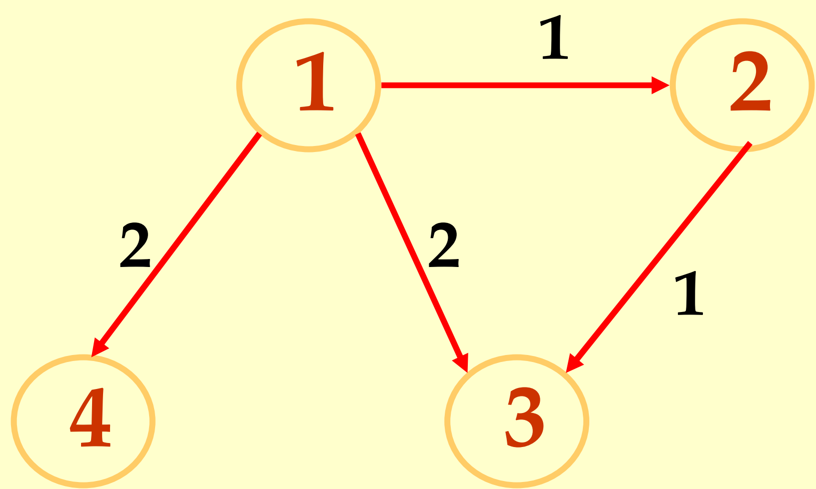
Example

```
1: r2 = *(r1 + 4)
2: r3 = *(r2 + 4)
3: r4 = r2 + r3
4: r5 = r2 - 1
```



Another Example

```
1: r2 = *(r1 + 4)
2: *(r1 + 4) = r3
3: r3 = r2 + r3
4: r5 = r2 - 1
```



Control Dependencies and Resource Constraints

- For now, let's only worry about basic blocks.
- For now, let's look at simple pipelines.

Example

- Assume:
  - Memory cached, available in 1 cycle.
  - Mul 3 cycles
  - Div 4 cycles
  - Other 1 cycle

Example

	Instruction	Operands	Results available in
1:	LA	r1,array	1 cycle
2:	LD	r2,4(r1)	1 cycle
3:	AND	r3,r3,0x00FF	1 cycle
4:	MULC	r6,r6,100	3 cycles
5:	ST	r7,4(r6)	
6:	DIVC	r5,r5,100	4 cycles
7:	ADD	r4,r2,r5	1 cycle
8:	MUL	r5,r2,r4	3 cycles
9:	ST	r4,0(r1)	

14 cycles!

1	**2**	**3**	**4**	**st**	**st**	**5**	**6**	**st**	**st**	**st**	**7**	**8**	**9**

List Scheduling Algorithm

- Idea:
  - Do a topological sort of the dependence DAG.
  - Consider when an instruction can be scheduled without causing a stall.
  - Schedule the instruction if it causes no stall and all its predecessors are already scheduled.
- Optimal list scheduling is NP-complete.
  - Use heuristics when necessary.

List Scheduling Algorithm

- Create a dependence DAG of a basic block.
- Topological Sort.

  READY = nodes with no predecessors.

  Loop until READY is empty.

    Schedule each node in READY when no stalling

    READY += nodes whose predecessors have all been scheduled.

Heuristics for selection

Heuristics for selecting from the READY list (the priority of the node) :

1. pick the node with the longest path to a leaf in the dependence graph.
2. pick a node with the most immediate successors.
3. pick a node that can go to a less busy pipeline (in a superscalar implementation).

Heuristics for selection

Pick the node with the longest path to a leaf in the dependence graph

Algorithm (for node x)

- If x has no successors $d_x = 0$
- $d_x = \mathrm{MAX}_{\forall y \in succ(x)}(d_y + v(x \rightarrow y))$.

Use reverse breadth-first visiting order

Heuristics for selection

Pick a node with the most immediate successors.

Algorithm (for node x):

- f_x = number of successors of x

Heuristics for selection from the READY list

The priority of the node:

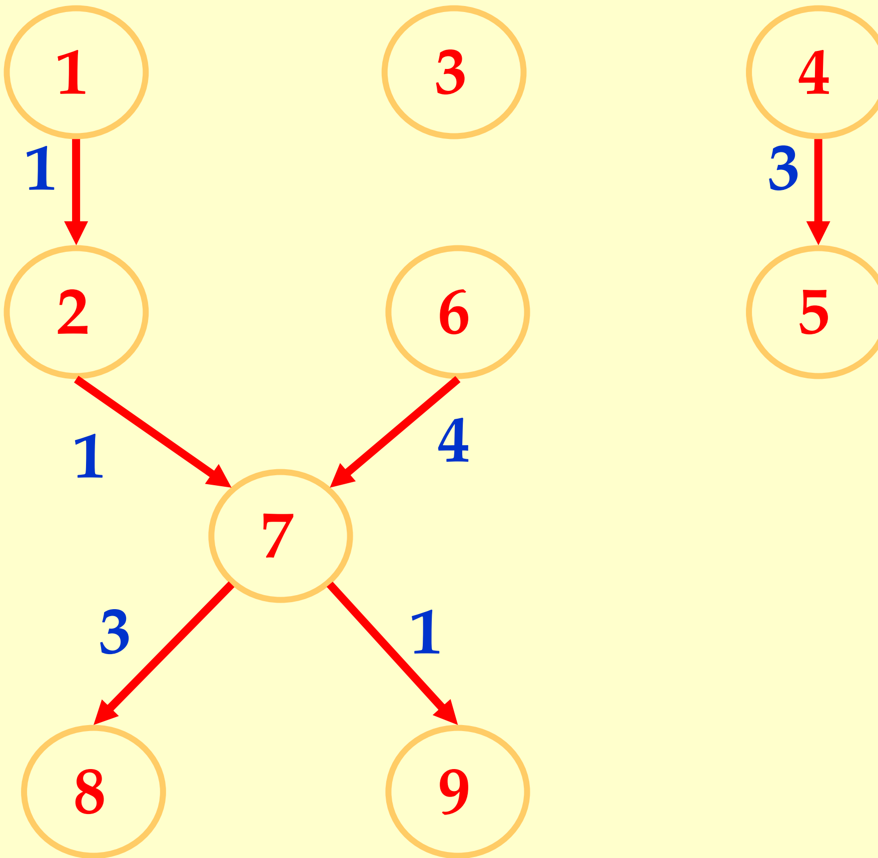
1. pick the node with the longest path to a leaf in the dependence graph: **Largest d_x.**
2. pick a node with the most immediate successors: **Largest f_x.**

Example

			Results available in
1:	LA	r1,array	1 cycle
2:	LD	r2,4(r1)	1 cycle
3:	AND	r3,r3,0x00FF	1 cycle
4:	MULC	r6,r6,100	3 cycles
5:	ST	r7,4(r6)	
6:	DIVC	r5,r5,100	4 cycles
7:	ADD	r4,r2,r5	1 cycle
8:	MUL	r5,r2,r4	3 cycles
9:	ST	r4,0(r1)	

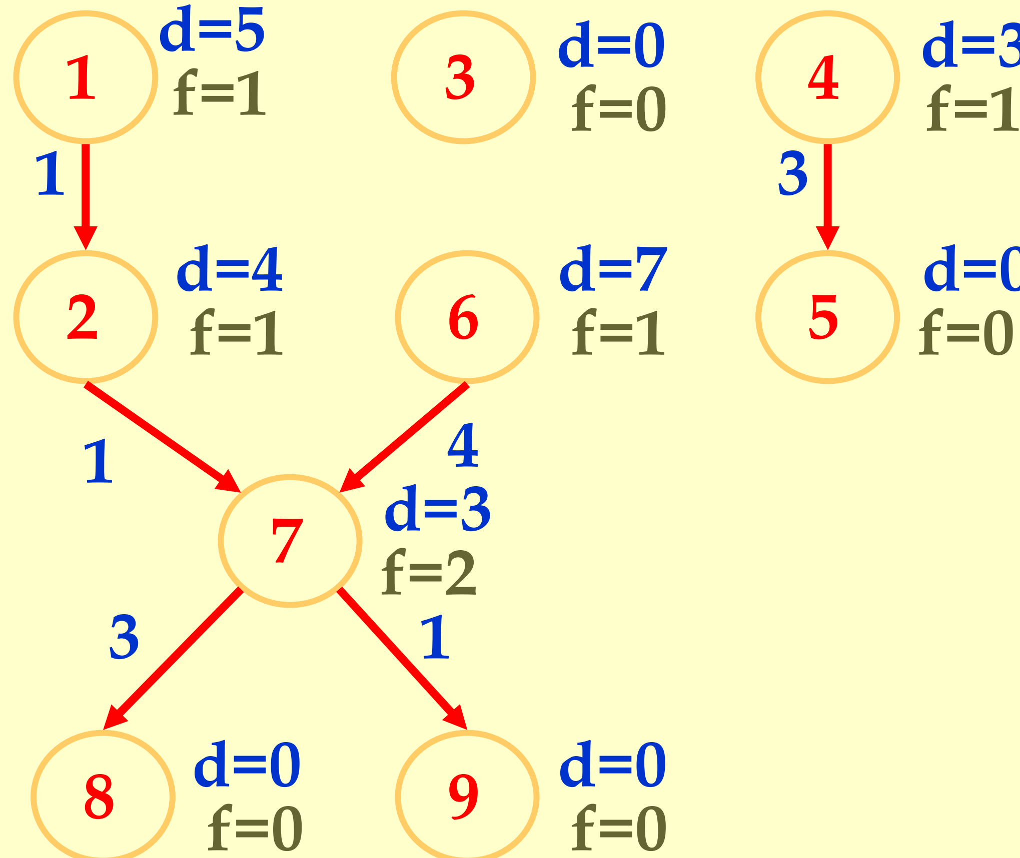
Example

```
1: LA    r1,array
2: LD    r2,4(r1)
3: AND   r3,r3,0x00FF
4: MULC  r6,r6,100
5: ST    r7,4(r6)
6: DIVC  r5,r5,100
7: ADD   r4,r2,r5
8: MUL   r5,r2,r4
9: ST    r4,0(r1)
```



Example

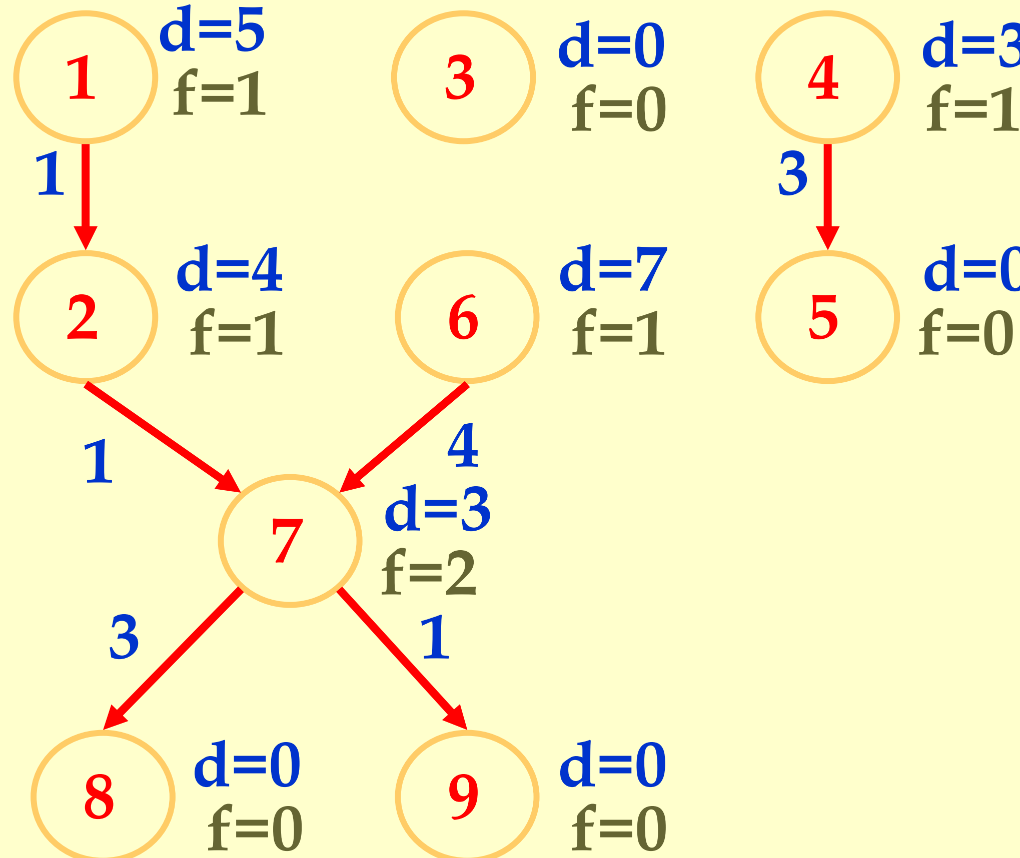
READY = { }



Example

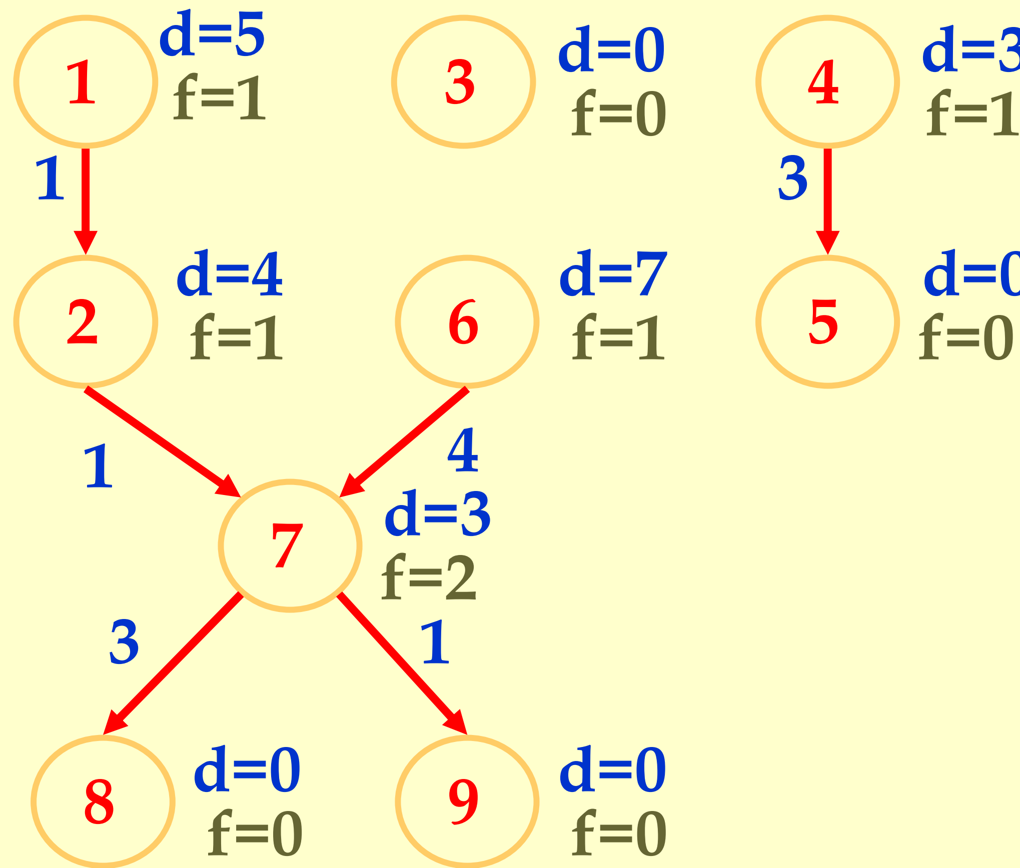
1, 3, 4, 6

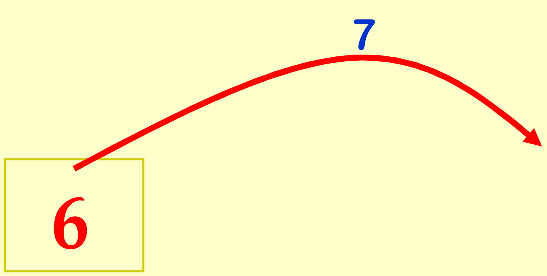
READY = { 6, 1, 4, 3}



Example

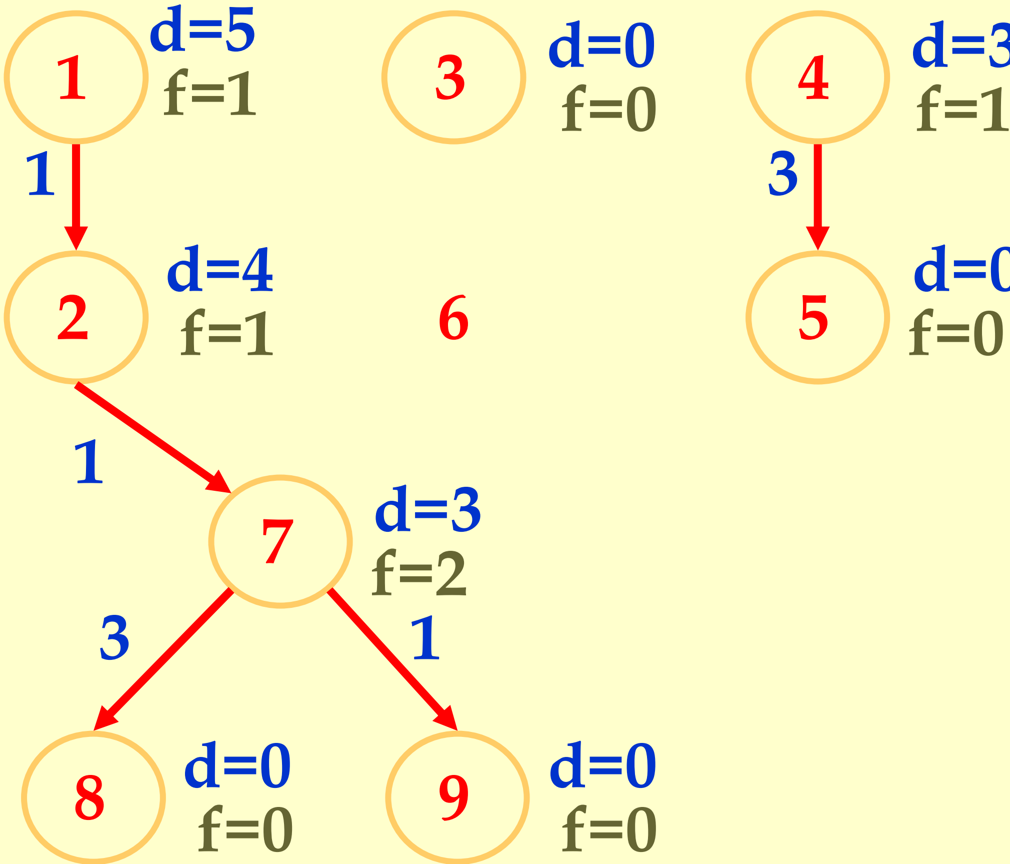
READY = { 6, 1, 4, 3 }

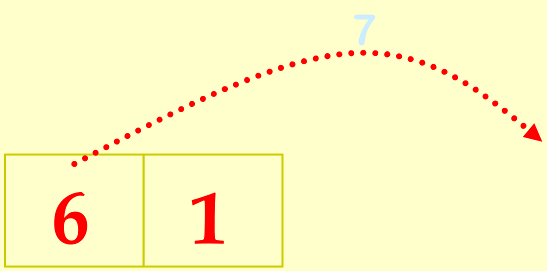




Example

READY = { 1, 4, 3 }

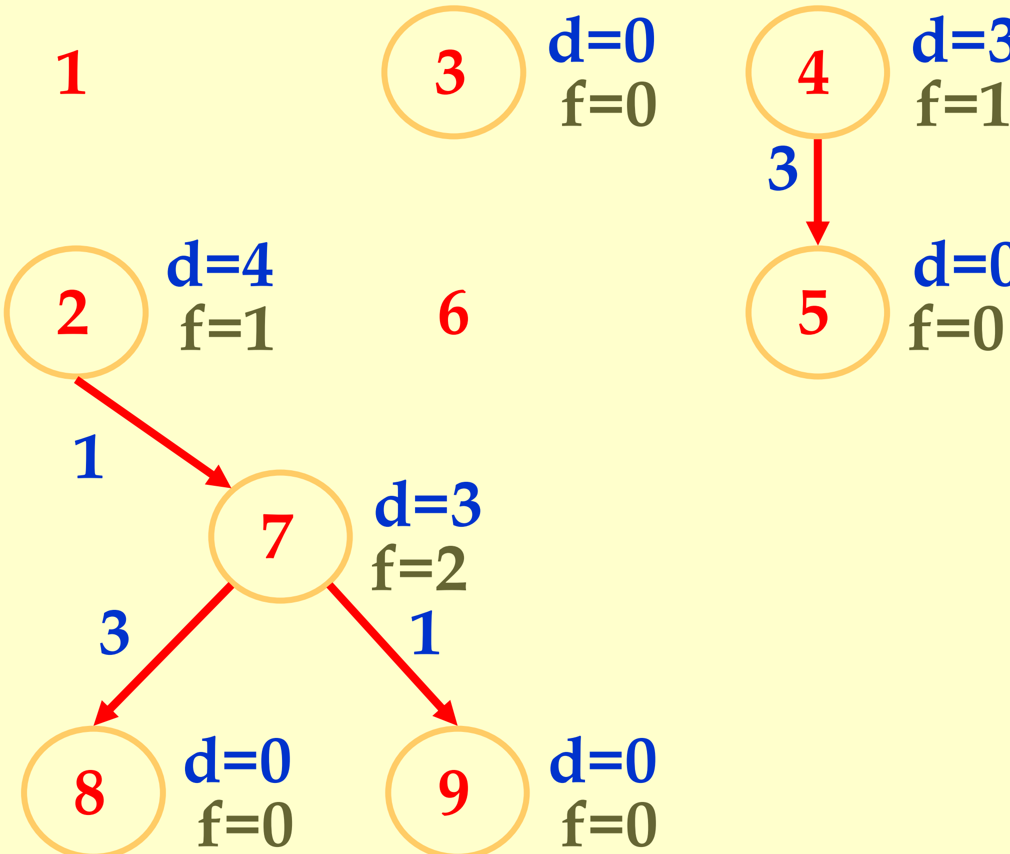


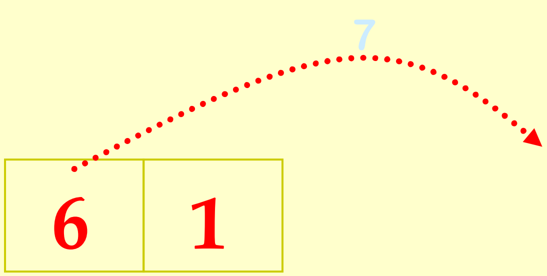


Example

2

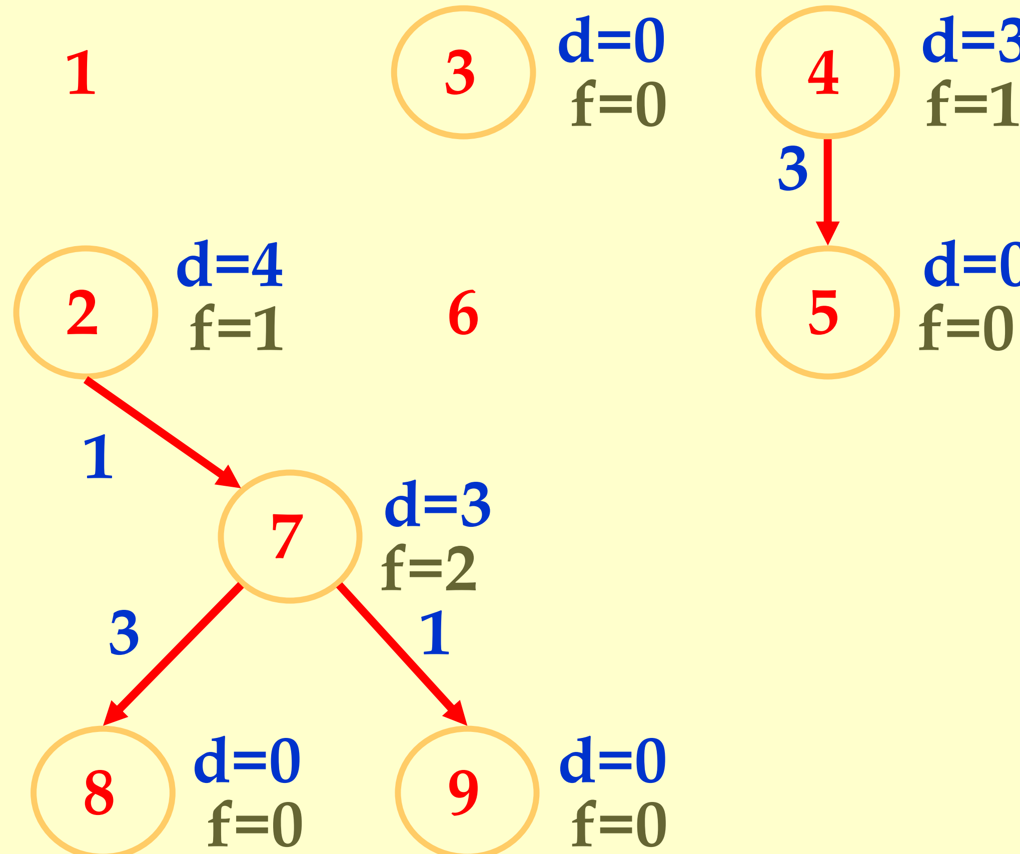
READY = { 4, 3 }

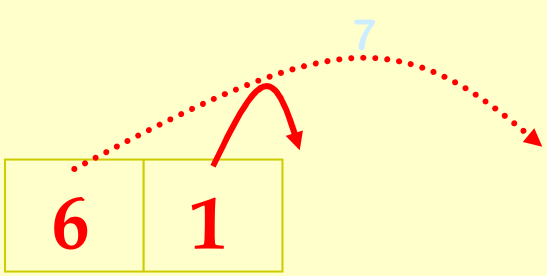




Example

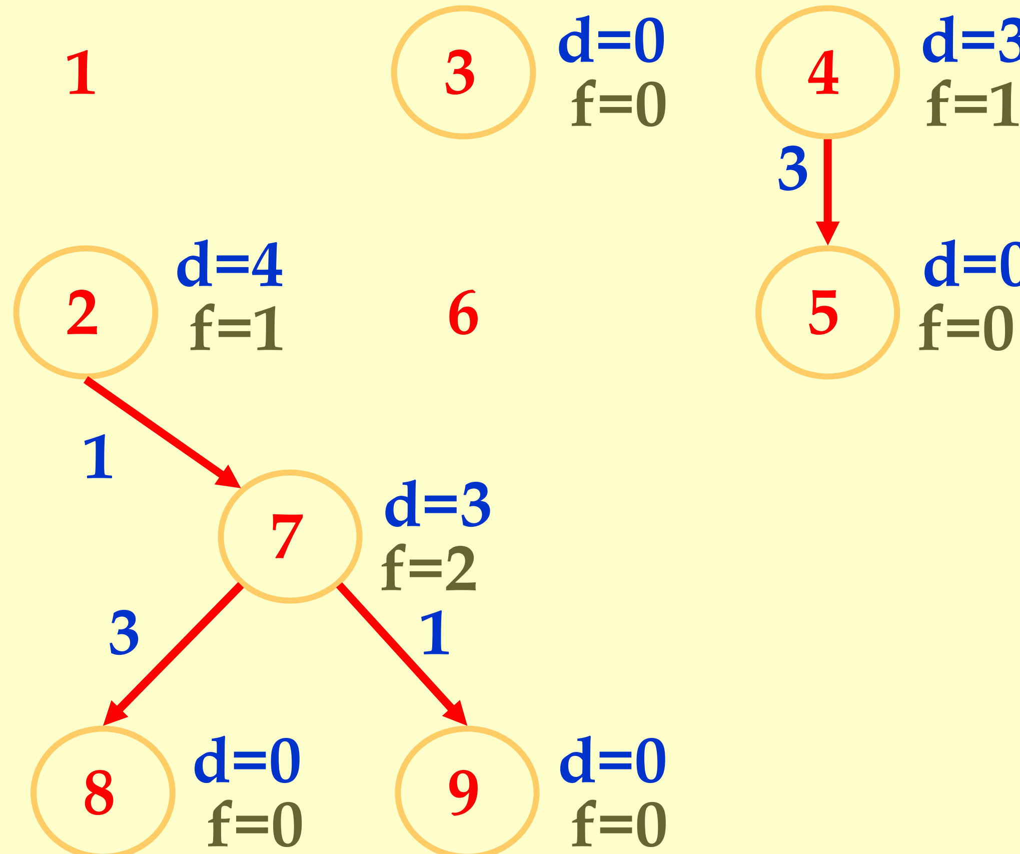
READY = { 2, 4, 3 }

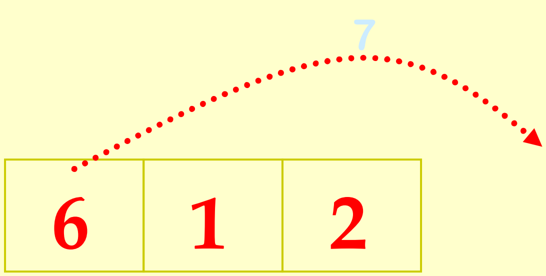




Example

READY = { 2, 4, 3 }

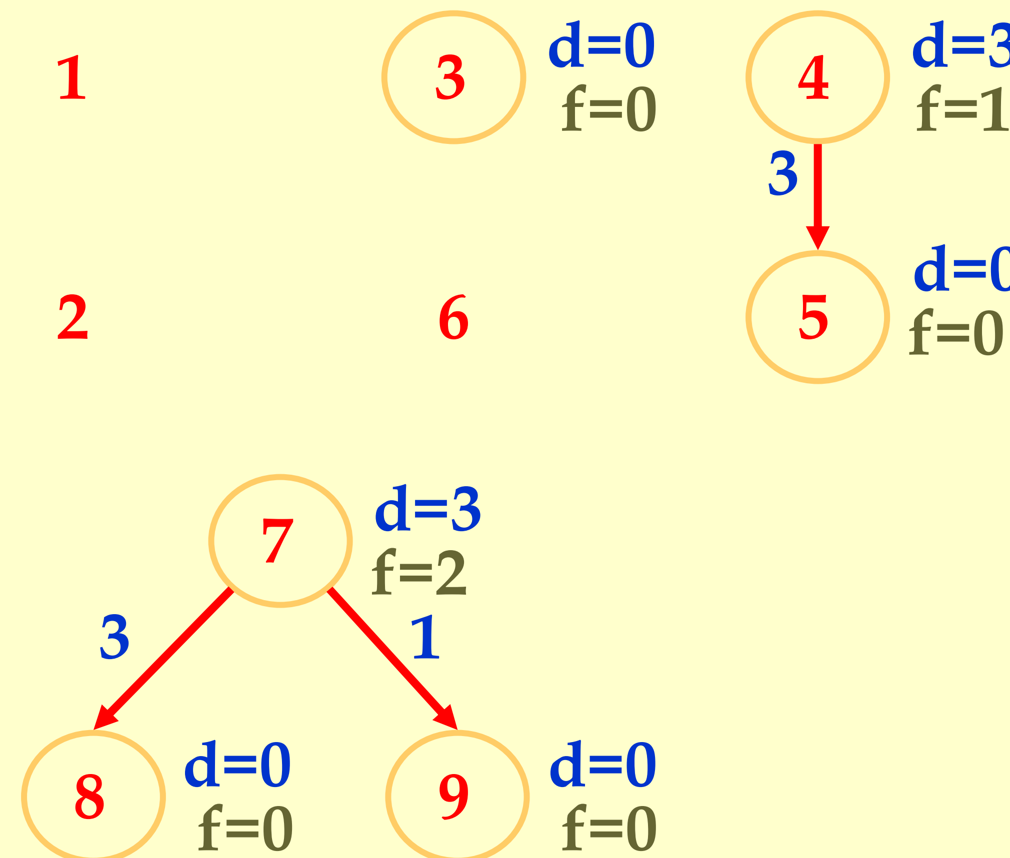


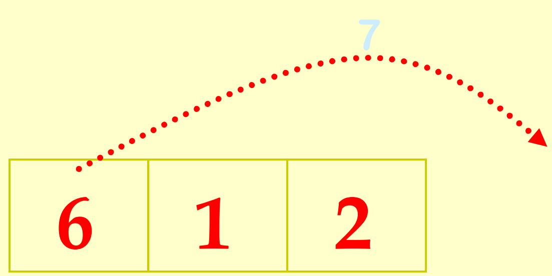


Example

7

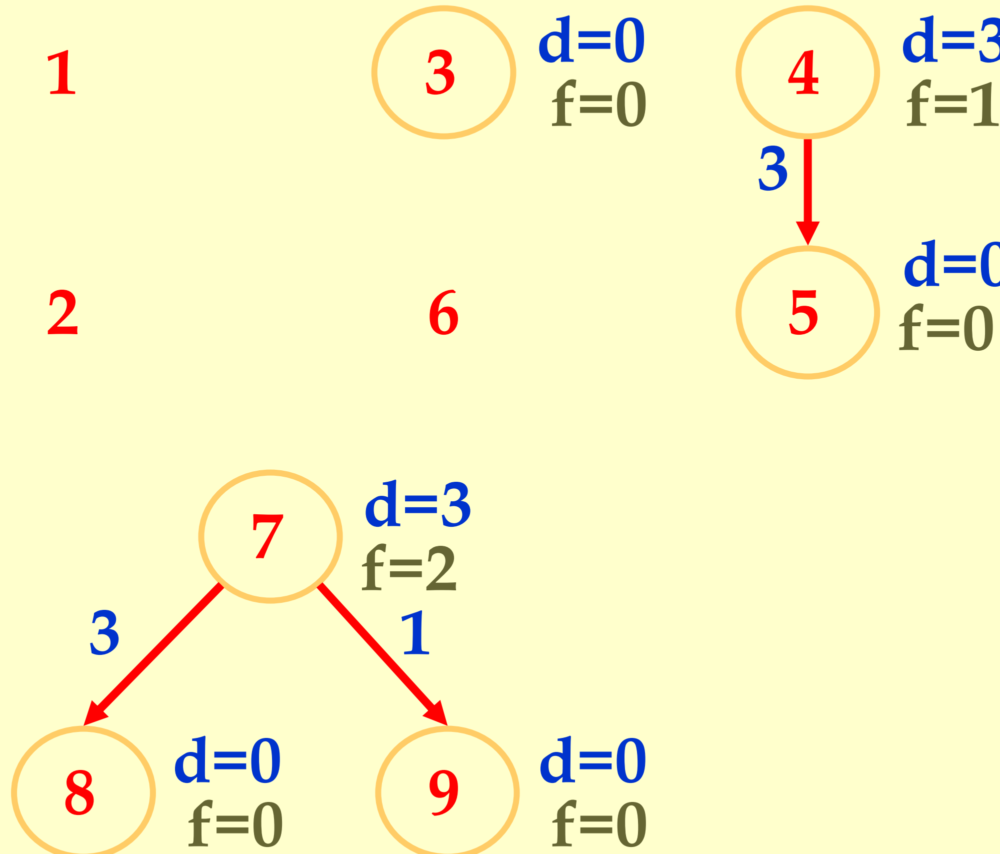
READY = { 4, 3 }

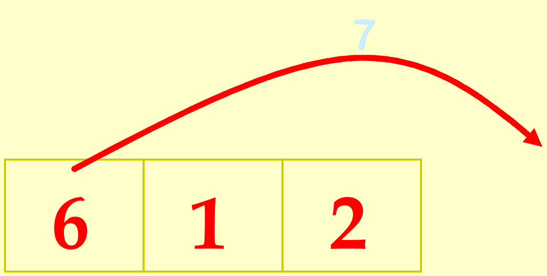




Example

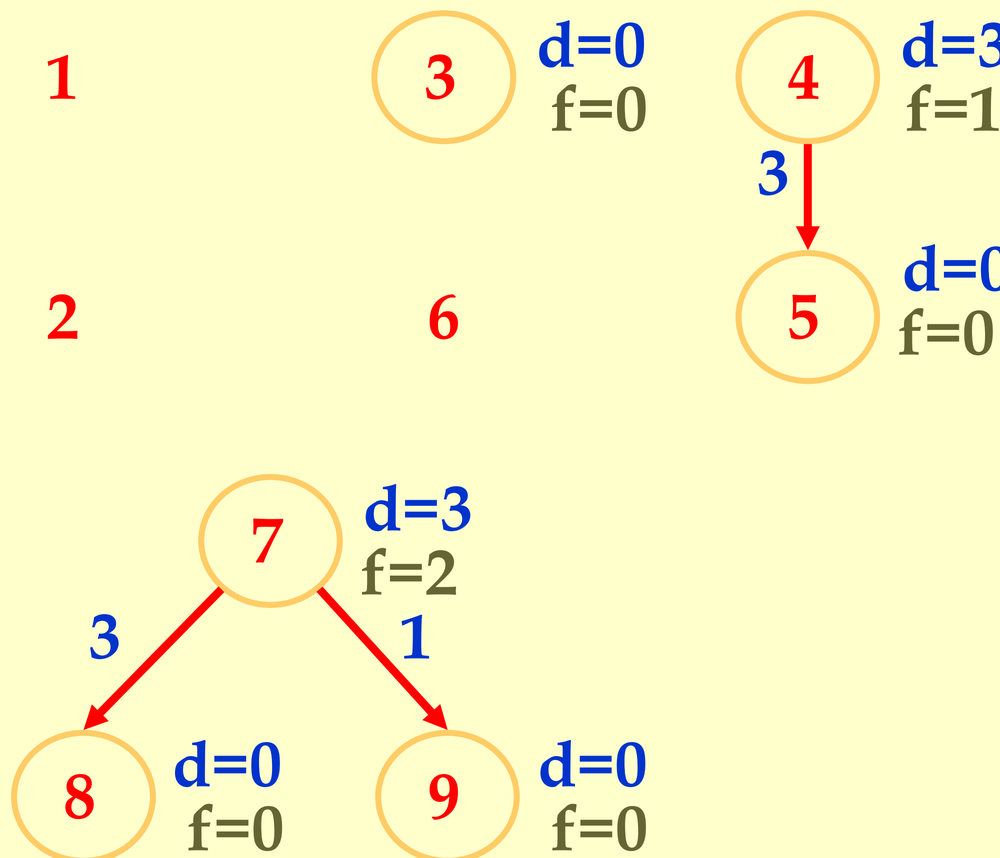
READY = { 7, 4, 3 }

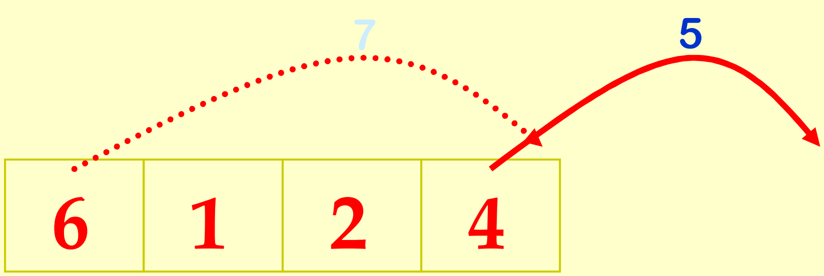




Example

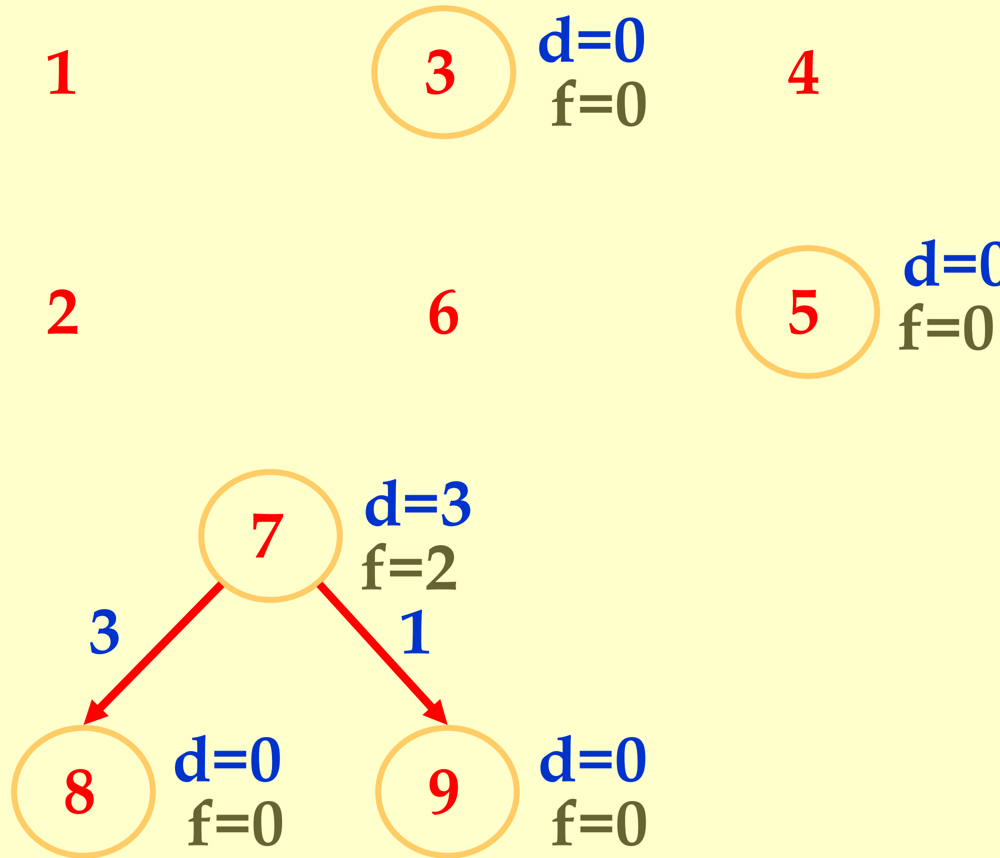
READY = { 7, 4, 3 }

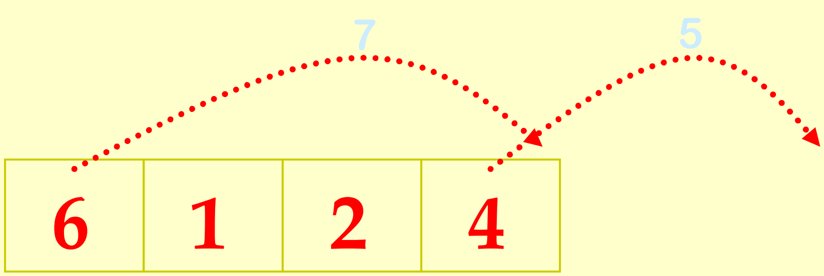




Example

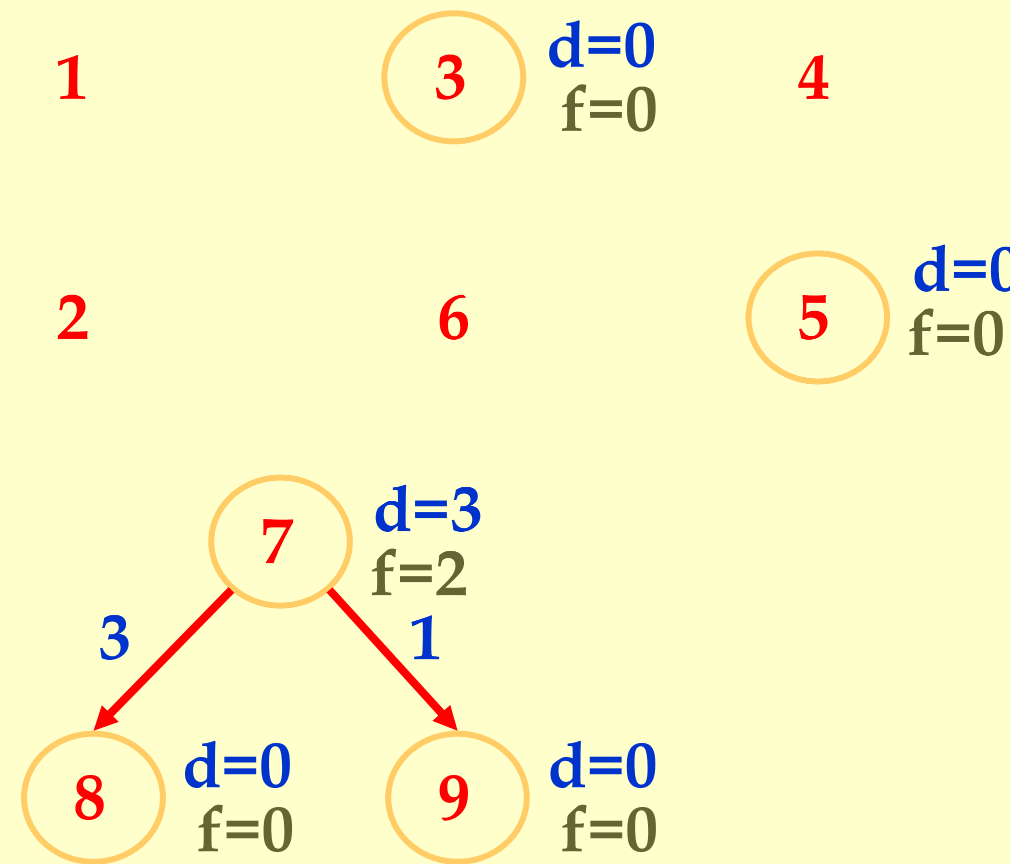
5
READY = { 7, 3 }

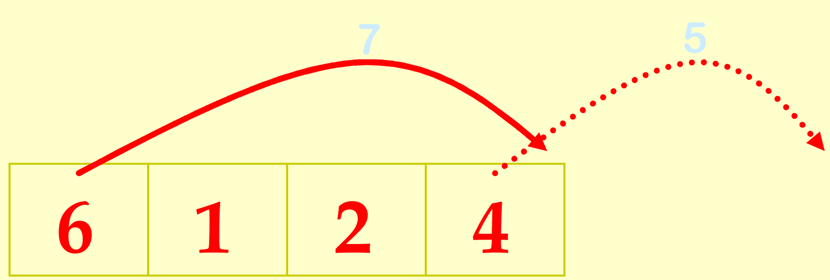




Example

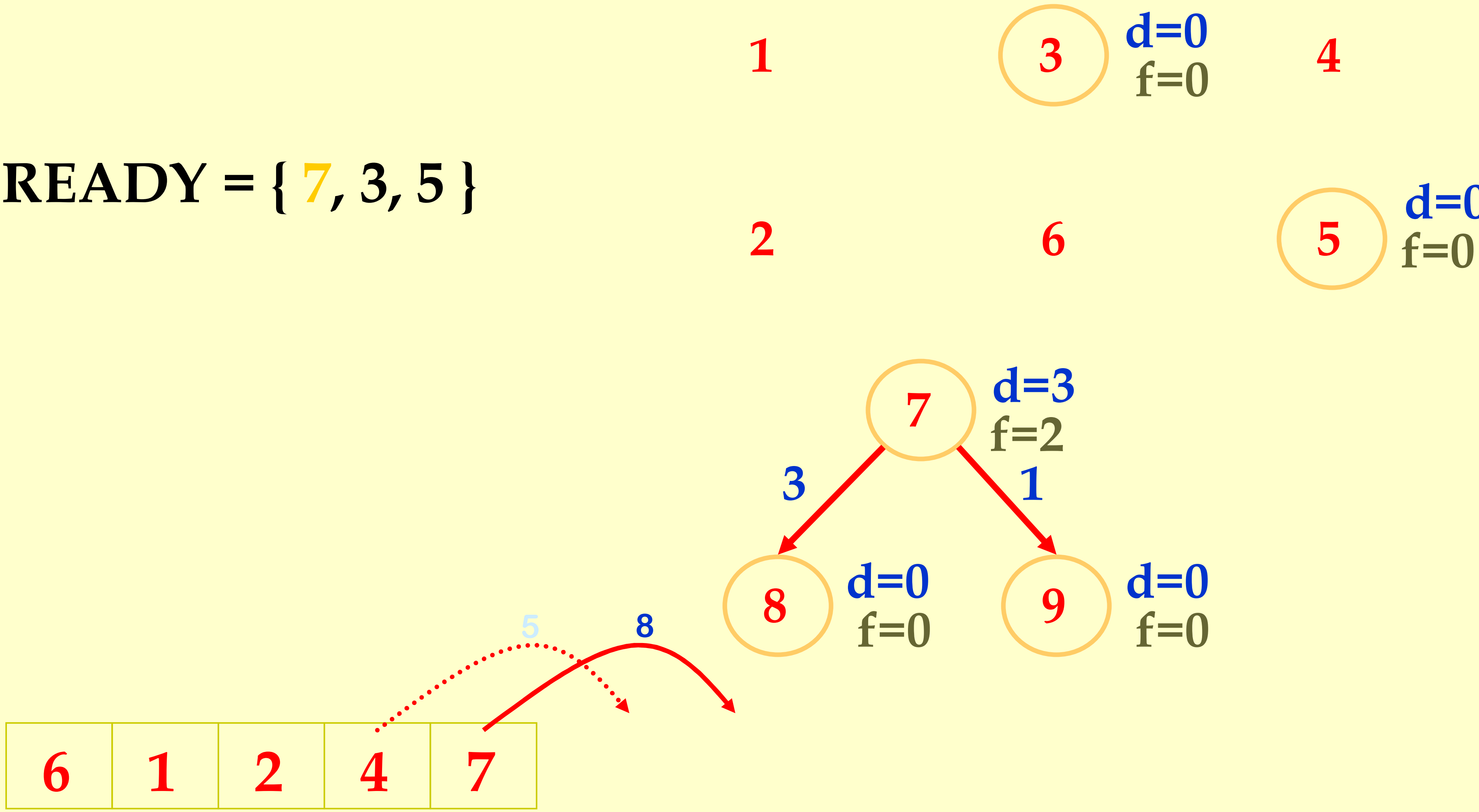
READY = { 7, 3, 5 }





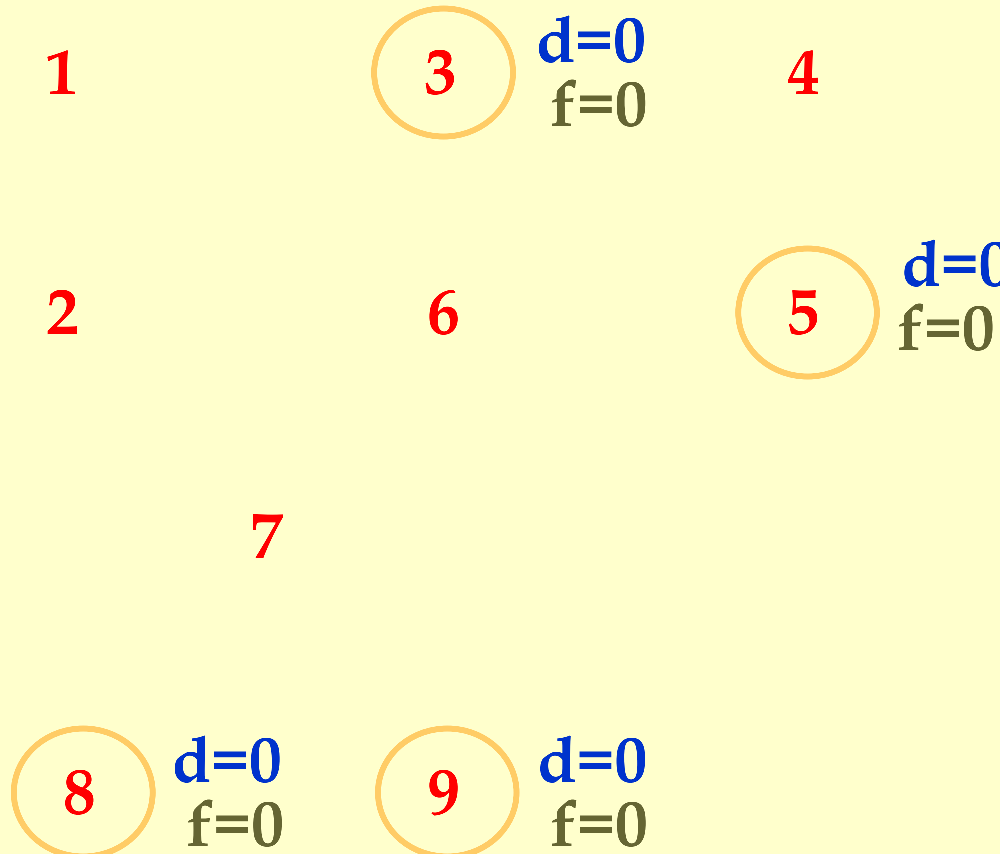
Example

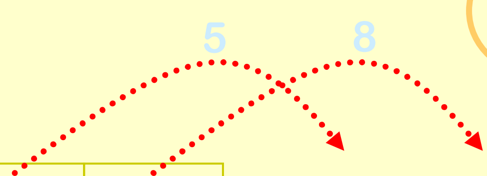
READY = { 7, 3, 5 }



Example

READY = { 3, 5 } 8, 9

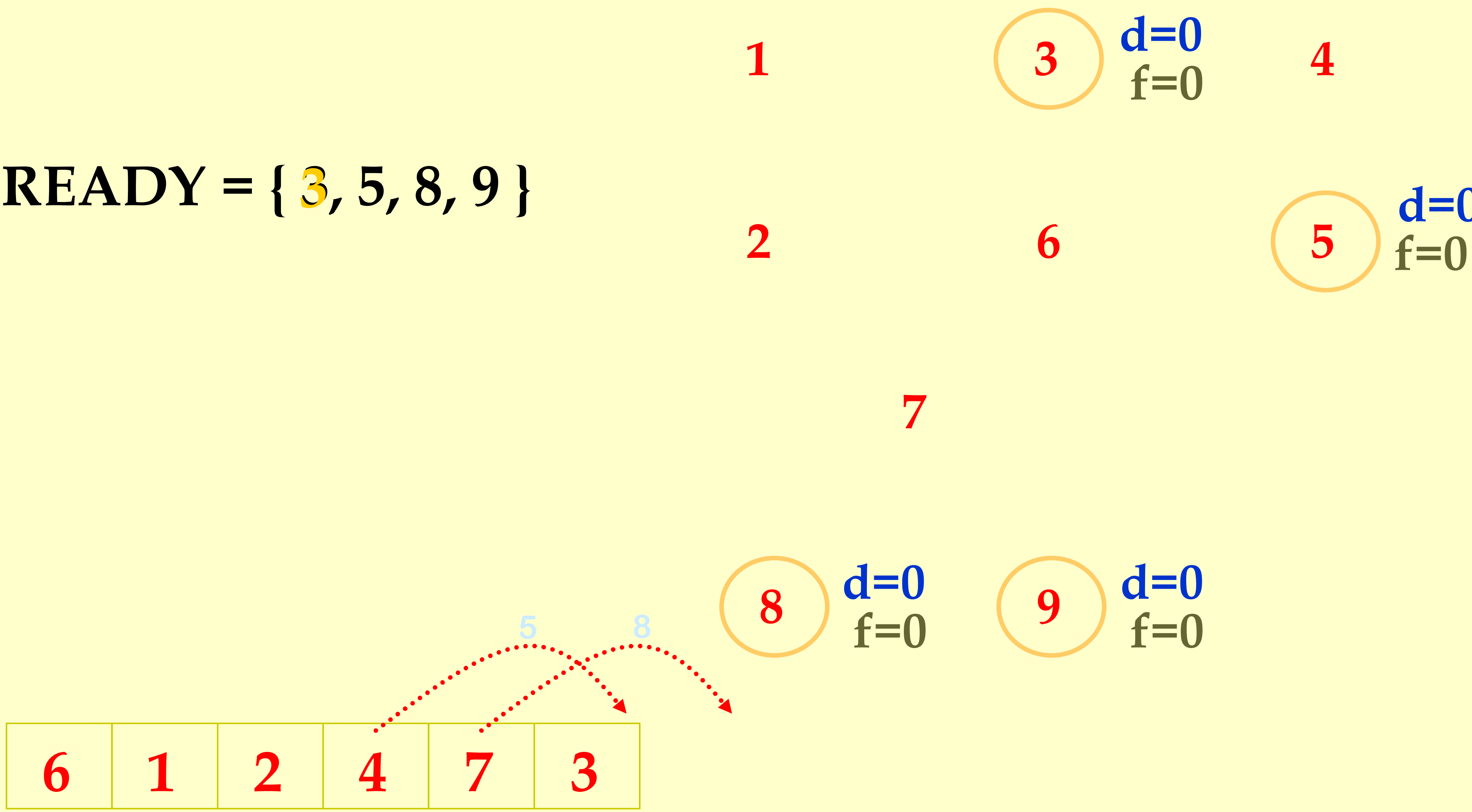




6	1	2	4	7

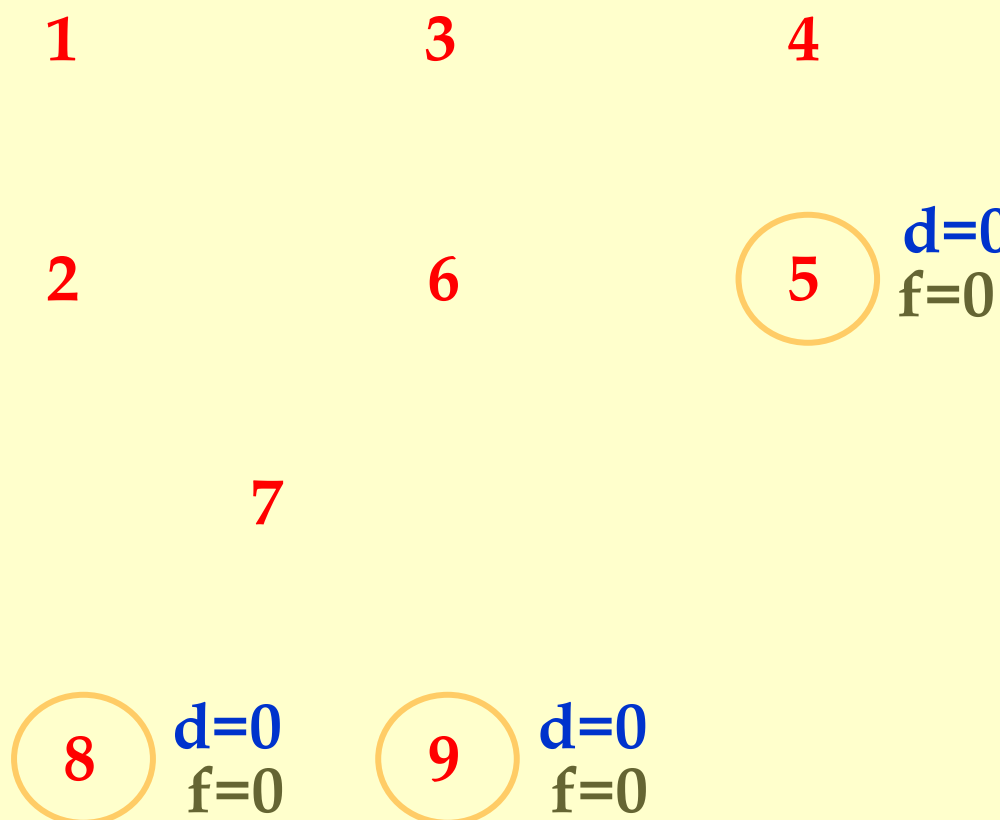
Example

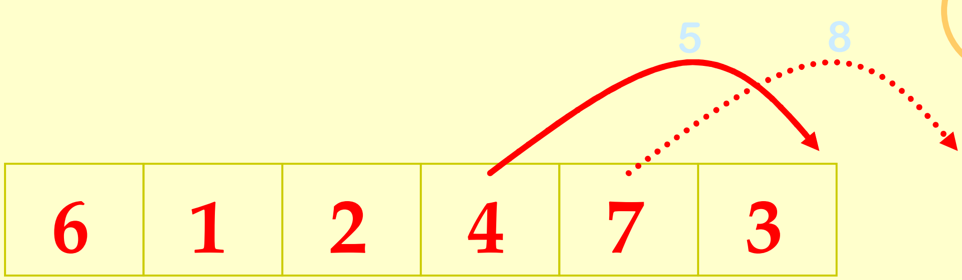
READY = { 3, 5, 8, 9 }



Example

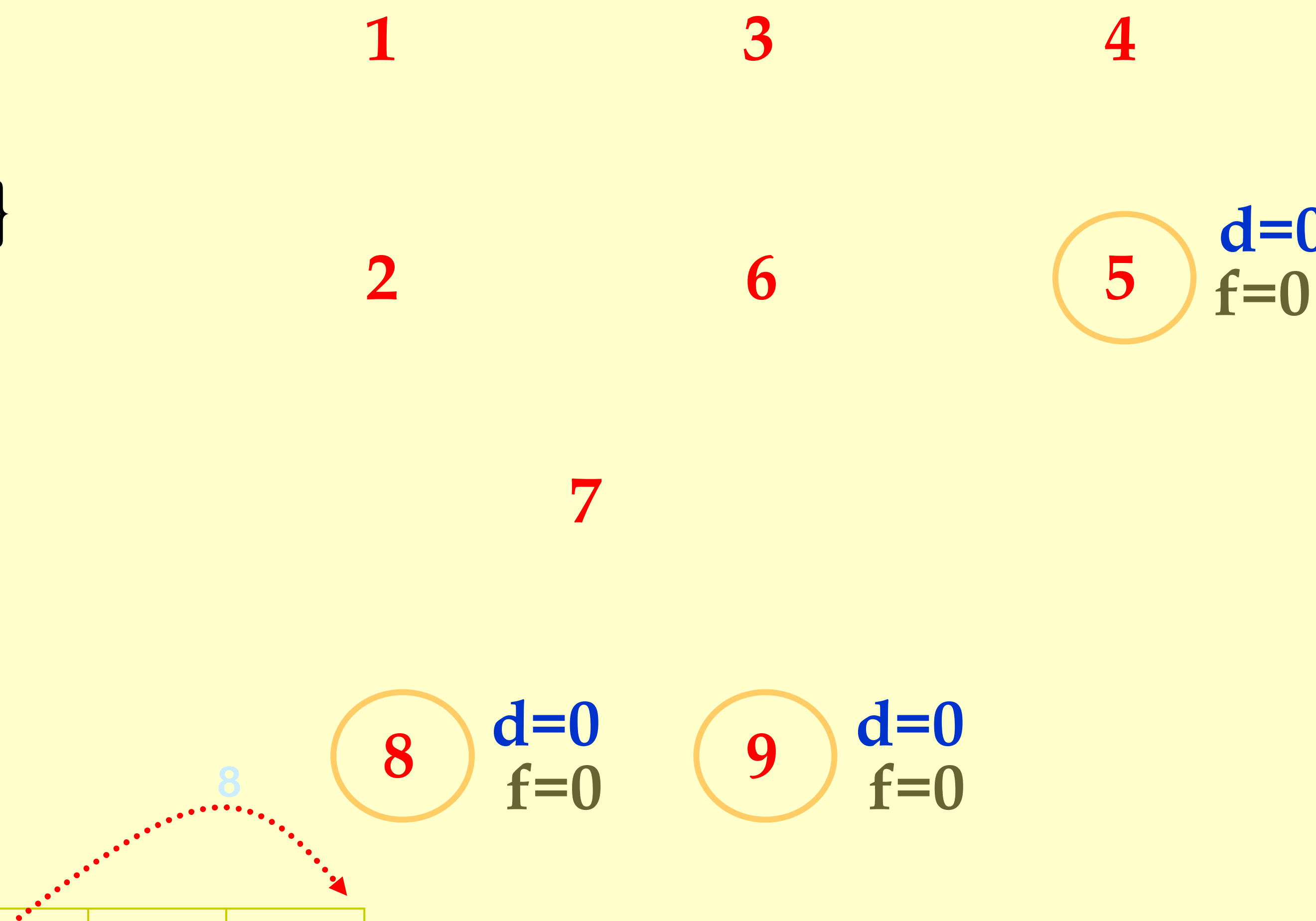
READY = { 5, 8, 9 }





Example

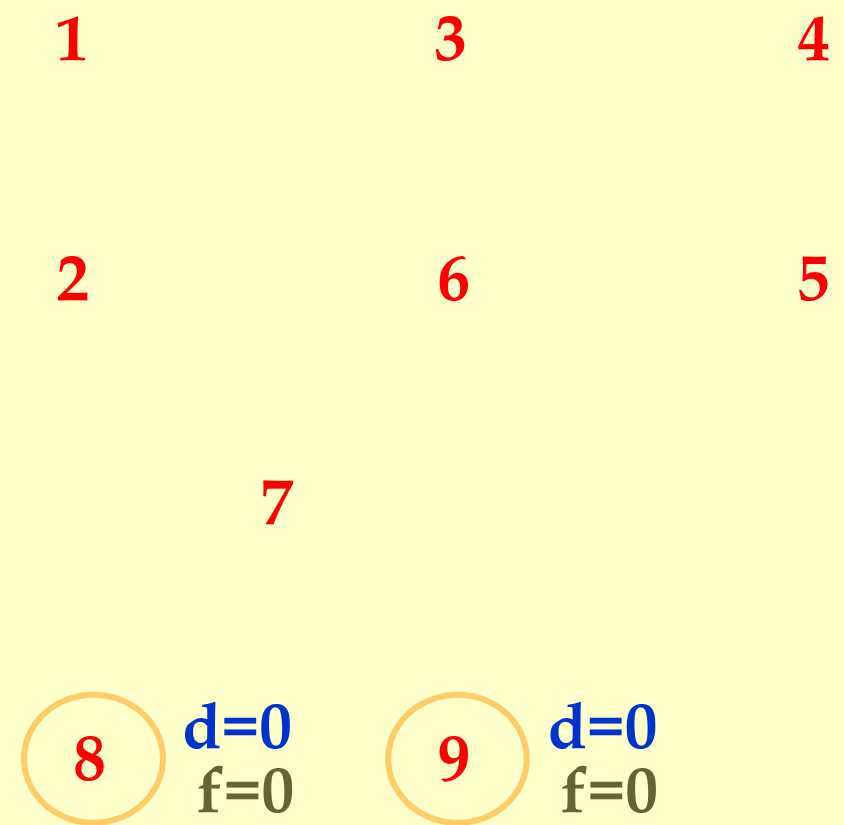
READY = { 5, 8, 9 }



6	1	2	4	7	3	5

Example

READY = { 8, 9 }

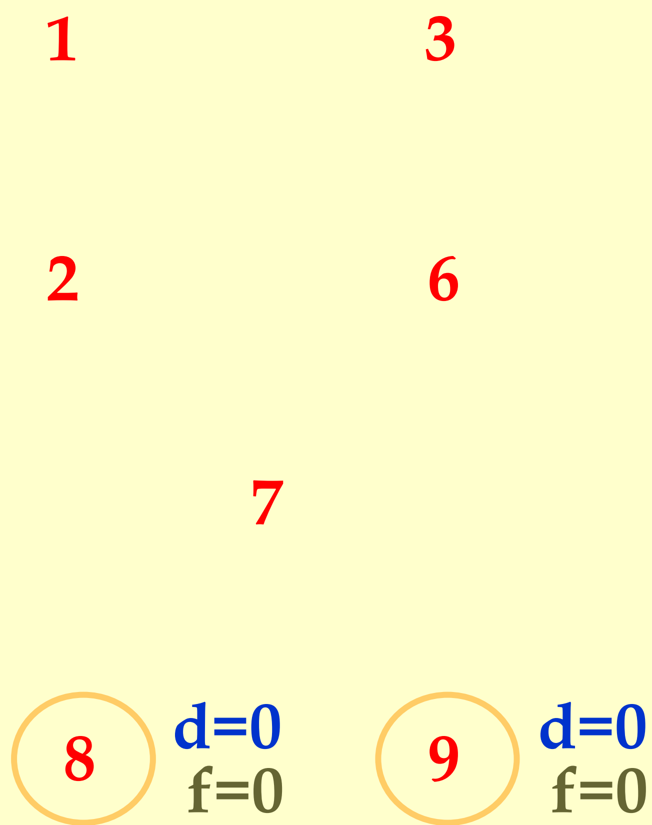


8

6	1	2	4	7	3	5

Example

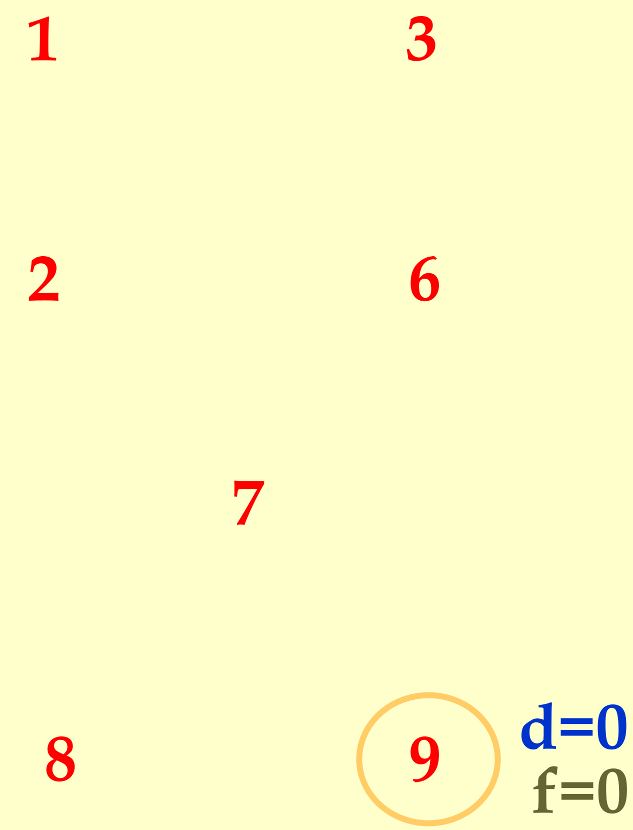
READY = { 8, 9 }



6	1	2	4	7	3	5	8

Example

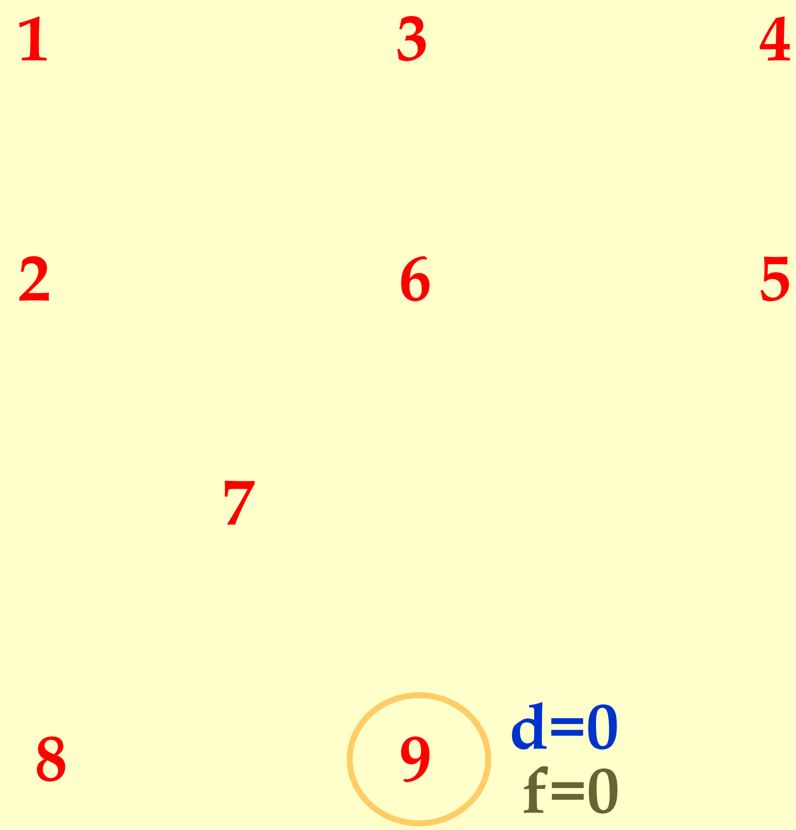
READY = { 9 }



6	1	2	4	7	3	5	8

Example

READY = { 9 }



6	1	2	4	7	3	5	8	9

Example

READY = { }

1 3 4

2 6 5

7

8 9

6	1	2	4	7	3	5	8	9

Example

		Results available in
1: LA	r1,array	1 cycle
2: LD	r2,4(r1)	1 cycle
3: AND	r3,r3,0x00FF	1 cycle
4: MULC	r6,r6,100	3 cycles
5: ST	r7,4(r6)	
6: DIVC	r5,r5,100	4 cycles
7: ADD	r4,r2,r5	1 cycle
8: MUL	r5,r2,r4	3 cycles
9: ST	r4,0(r1)	

1	2	3	4	st	st	5	6	st	st	st	7	8	9

6	1	2	4	7	3	5	8	9

14 cycles vs. 9 cycles

Resource Constraints

- Modern machines have many resource constraints.
- Superscalar architectures:
  - can run few parallel operations.
  - but have constraints.

Resource Constraints of a Superscalar Processor

Example:

- 1 integer operation, e.g.,
  ALUop dest, src1, src2 # in 1 clock cycle

In parallel with

- 1 memory operation, e.g.,
  LD dst, addr # in 2 clock cycles
  ST src, addr # in 1 clock cycle

List Scheduling Algorithm with Resource Constraints

- Represent the superscalar architecture as multiple pipelines.
  - Each pipeline represents some resource.

List Scheduling Algorithm with Resource Constraints

- Represent the superscalar architecture as multiple pipelines
  - Each pipeline represents some resource
- Example:
  - One single cycle ALU unit.
  - One two-cycle pipelined memory unit.

ALUop							
MEM 1							
MEM 2							

List Scheduling Algorithm with Resource Constraints

- Create a dependence DAG of a basic block.
- Topological Sort

  READY = nodes with no predecessors

  Loop until READY is empty

    Let n ∈ READY be the node with the highest priority

    Schedule n in the earliest slot

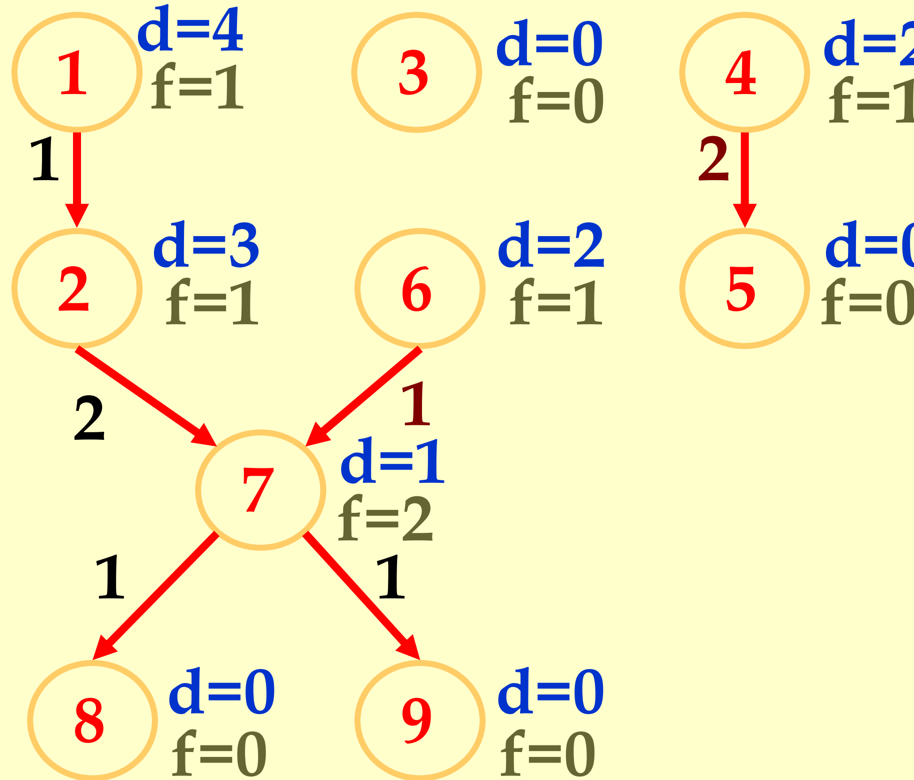
      that satisfies precedence + resource constraints

    Update READY

Example

(Slightly different from previous example.)

```
1: LA     r1,array
2: LD     r2,4(r1)
3: AND    r3,r3,0x00FF
4: LD     r6,8(sp)
5: ST     r7,4(r6)
6: ADD    r5,r5,100
7: ADD    r4,r2,r5
8: MUL    r5,r2,r4
9: ST     r4,0(r1)
```

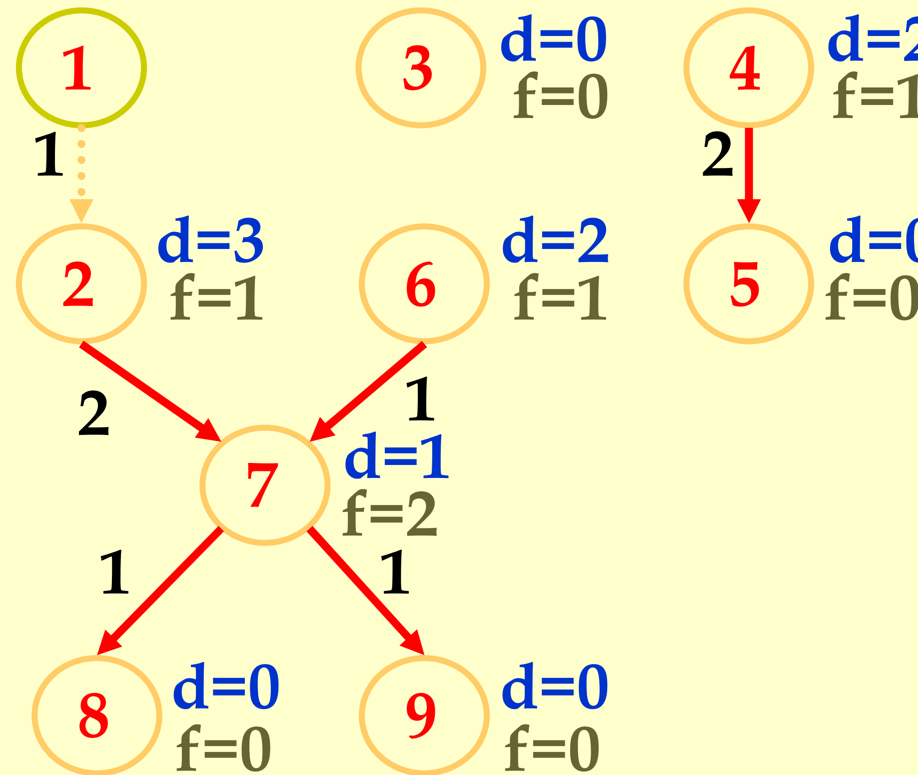


READY = { 1, 6, 4, 3 }

ALUop	1						
MEM 1							
MEM 2							

Example

```
1: LA     r1,array
2: LD     r2,4(r1)
3: AND    r3,r3,0x00FF
4: LD     r6,8(sp)
5: ST     r7,4(r6)
6: ADD    r5,r5,100
7: ADD    r4,r2,r5
8: MUL    r5,r2,r4
9: ST     r4,0(r1)
```

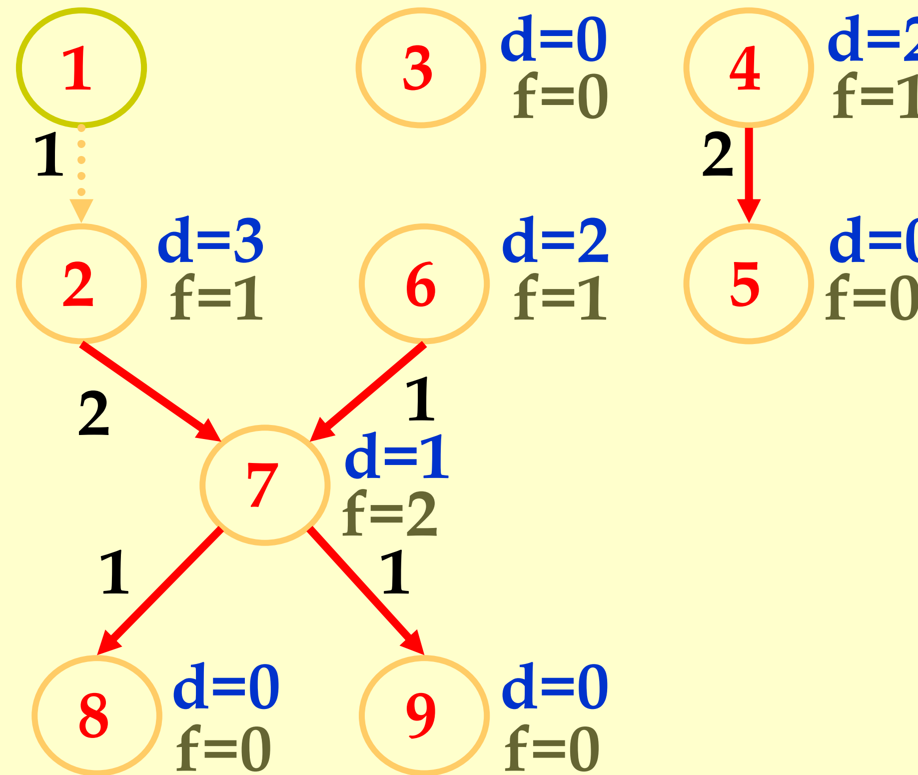


READY = { 6, 4, 3 } ⟵ 2

ALUop	1						
MEM 1							
MEM 2							

Example

```
1: LA    r1,array
2: LD    r2,4(r1)
3: AND   r3,r3,0x00FF
4: LD    r6,8(sp)
5: ST    r7,4(r6)
6: ADD   r5,r5,100
7: ADD   r4,r2,r5
8: MUL   r5,r2,r4
9: ST    r4,0(r1)
```

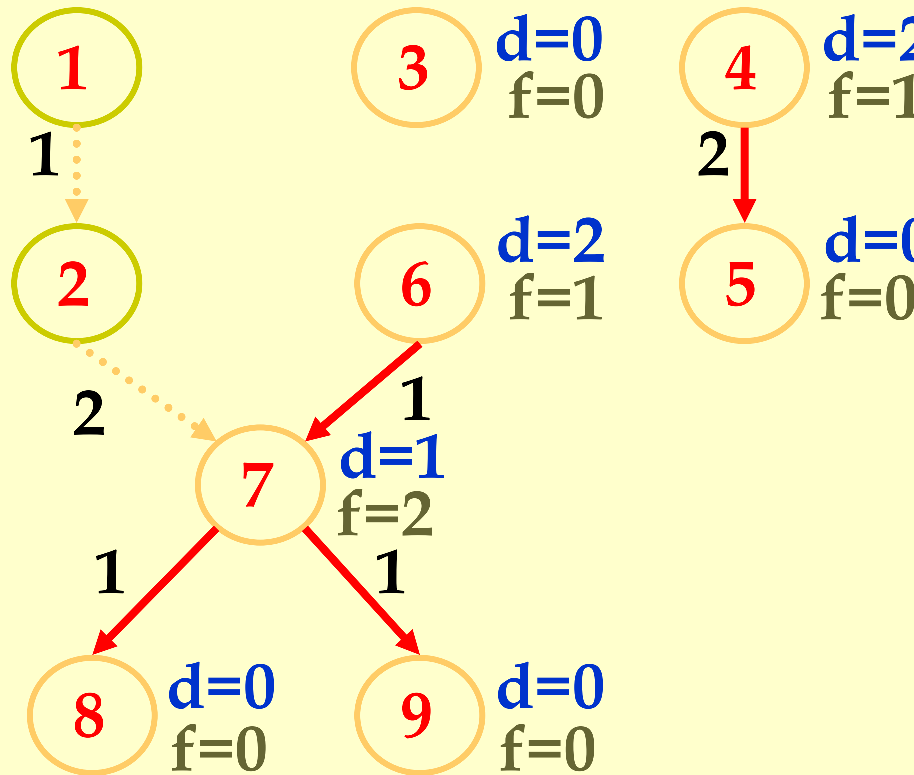


READY = { 2, 6, 4, 3 }

ALUop	1						
MEM 1		2					
MEM 2			2				

Example

```
1: LA    r1,array
2: LD    r2,4(r1)
3: AND   r3,r3,0x00FF
4: LD    r6,8(sp)
5: ST    r7,4(r6)
6: ADD   r5,r5,100
7: ADD   r4,r2,r5
8: MUL   r5,r2,r4
9: ST    r4,0(r1)
```

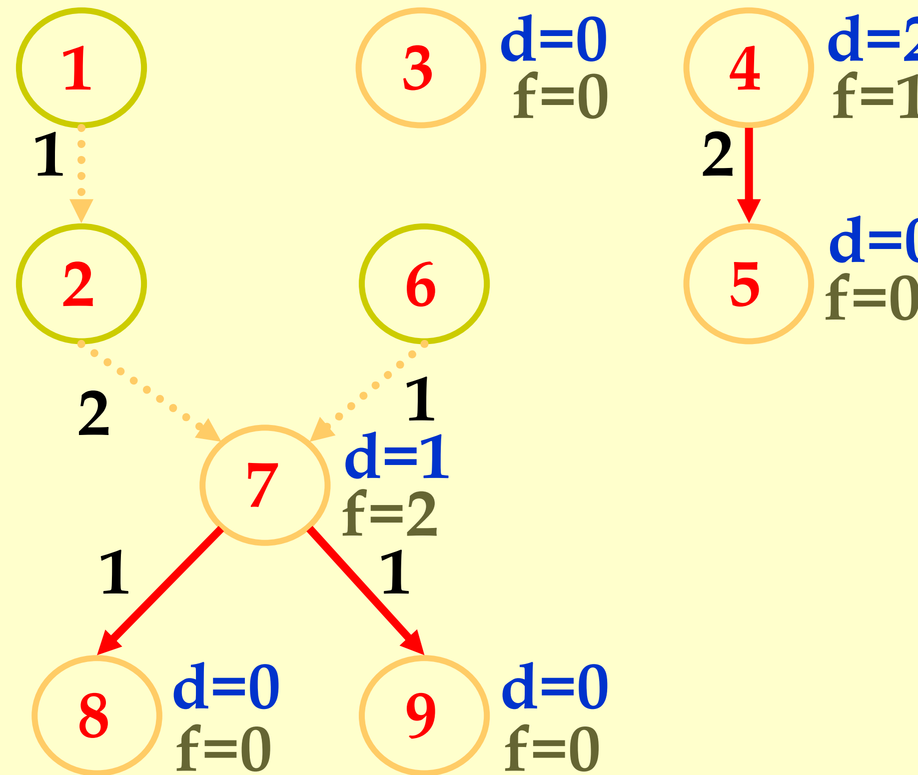


READY = { 6, 4, 3 }

ALUop	1	6					
MEM 1		2					
MEM 2			2				

Example

```
1: LA     r1,array
2: LD     r2,4(r1)
3: AND    r3,r3,0x00FF
4: LD     r6,8(sp)
5: ST     r7,4(r6)
6: ADD    r5,r5,100
7: ADD    r4,r2,r5
8: MUL    r5,r2,r4
9: ST     r4,0(r1)
```

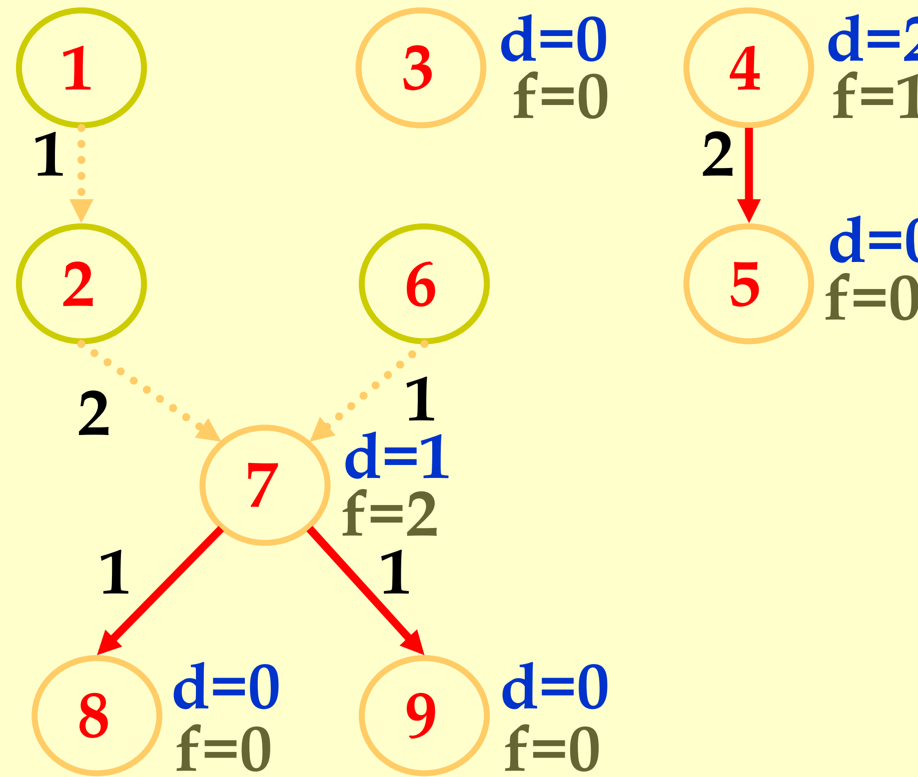


READY = { 4, 3 } ← 7

ALUop	1	6					
MEM 1		2					
MEM 2			2				

Example

```
1: LA     r1,array
2: LD     r2,4(r1)
3: AND    r3,r3,0x00FF
4: LD     r6,8(sp)
5: ST     r7,4(r6)
6: ADD    r5,r5,100
7: ADD    r4,r2,r5
8: MUL    r5,r2,r4
9: ST     r4,0(r1)
```

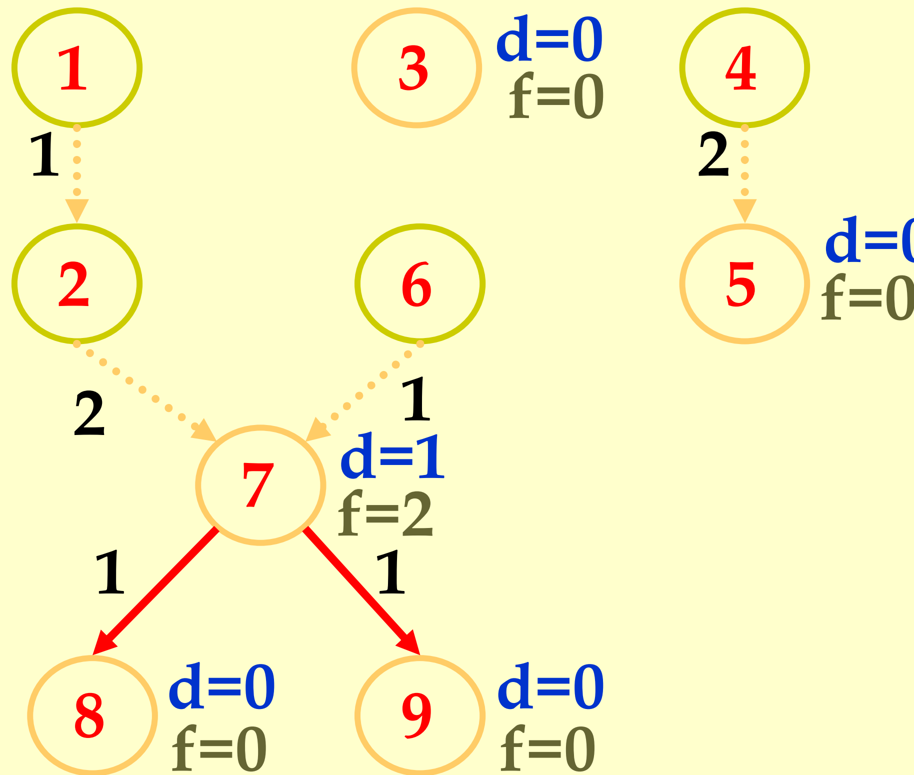


READY = { 4, 7, 3 }

ALUop	1	6					
MEM 1	4	2					
MEM 2		4	2				

Example

```
1: LA     r1,array
2: LD     r2,4(r1)
3: AND    r3,r3,0x00FF
4: LD     r6,8(sp)
5: ST     r7,4(r6)
6: ADD    r5,r5,100
7: ADD    r4,r2,r5
8: MUL    r5,r2,r4
9: ST     r4,0(r1)
```

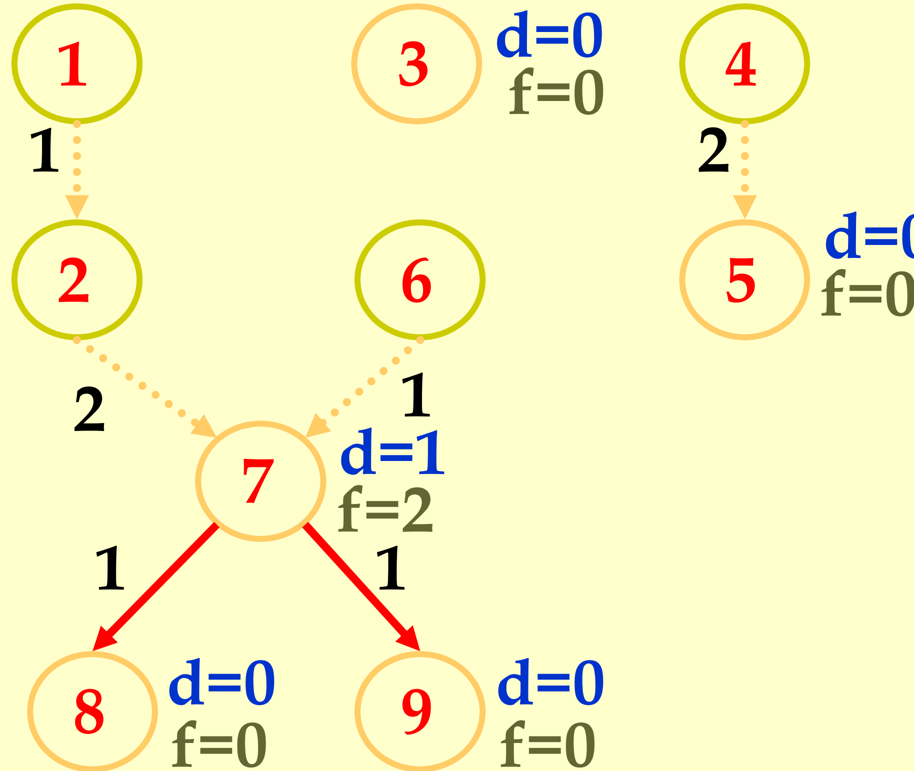


READY = { 7, 3 } ← 5

ALUop	1	6					
MEM 1	4	2					
MEM 2		4	2				

Example

```
1: LA    r1,array
2: LD    r2,4(r1)
3: AND   r3,r3,0x00FF
4: LD    r6,8(sp)
5: ST    r7,4(r6)
6: ADD   r5,r5,100
7: ADD   r4,r2,r5
8: MUL   r5,r2,r4
9: ST    r4,0(r1)
```

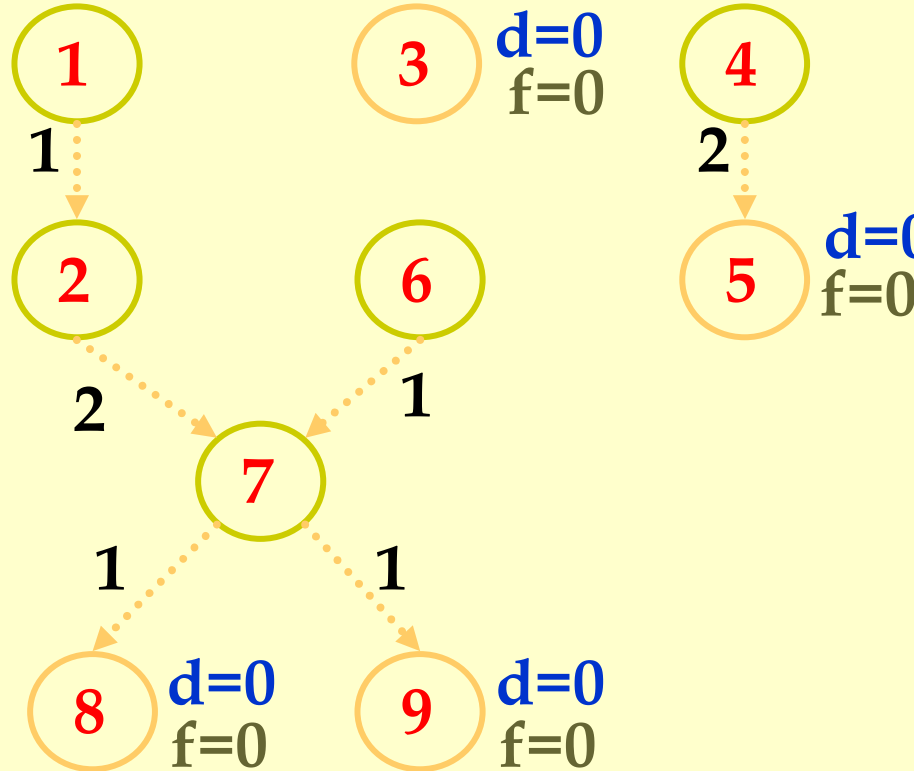


READY = { 7, 3, 5 }

ALUop	1	6		7			
MEM 1	4	2					
MEM 2		4	2				

Example

```
1:  LA     r1,array
2:  LD     r2,4(r1)
3:  AND    r3,r3,0x00FF
4:  LD     r6,8(sp)
5:  ST     r7,4(r6)
6:  ADD    r5,r5,100
7:  ADD    r4,r2,r5
8:  MUL    r5,r2,r4
9:  ST     r4,0(r1)
```

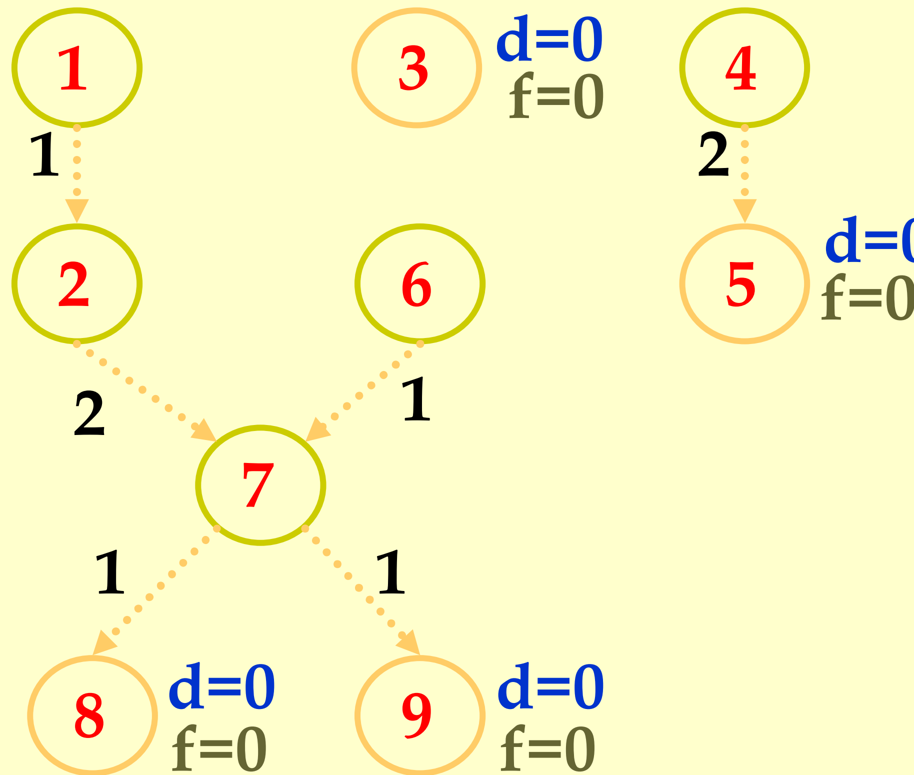


READY = { 3, 5 } ← 8, 9

ALUop	1	6		7			
MEM 1	4	2					
MEM 2		4	2				

Example

```
1: LA     r1,array
2: LD     r2,4(r1)
3: AND    r3,r3,0x00FF
4: LD     r6,8(sp)
5: ST     r7,4(r6)
6: ADD    r5,r5,100
7: ADD    r4,r2,r5
8: MUL    r5,r2,r4
9: ST     r4,0(r1)
```

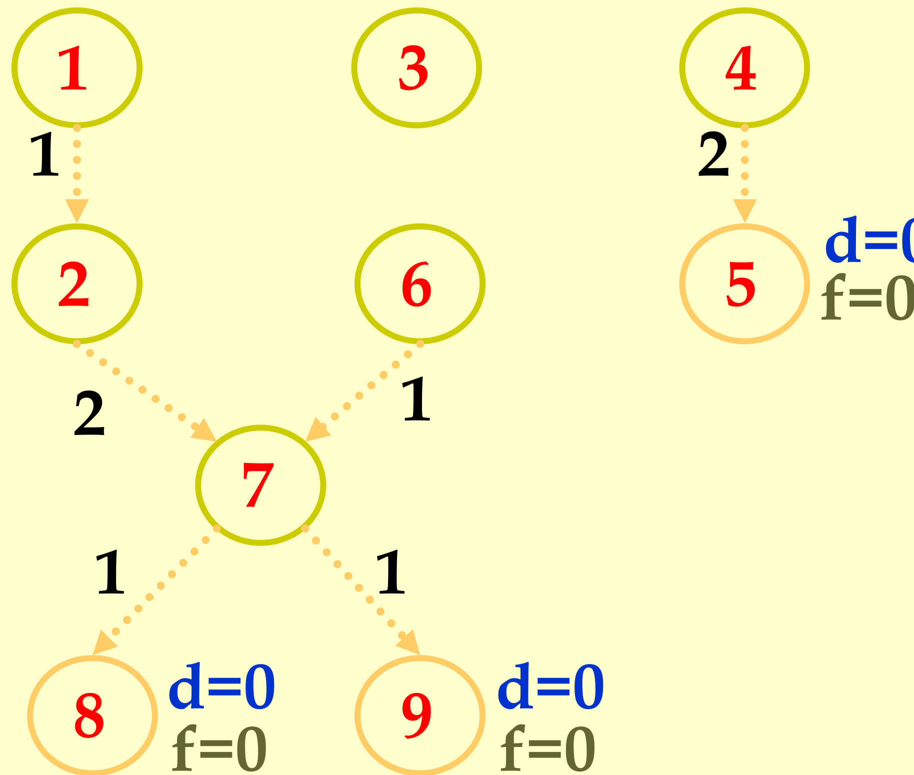


READY = { 3, 5, 8, 9 }

ALUop	1	6	3	7			
MEM 1	4	2					
MEM 2		4	2				

Example

```
1: LA    r1,array
2: LD    r2,4(r1)
3: AND   r3,r3,0x00FF
4: LD    r6,8(sp)
5: ST    r7,4(r6)
6: ADD   r5,r5,100
7: ADD   r4,r2,r5
8: MUL   r5,r2,r4
9: ST    r4,0(r1)
```

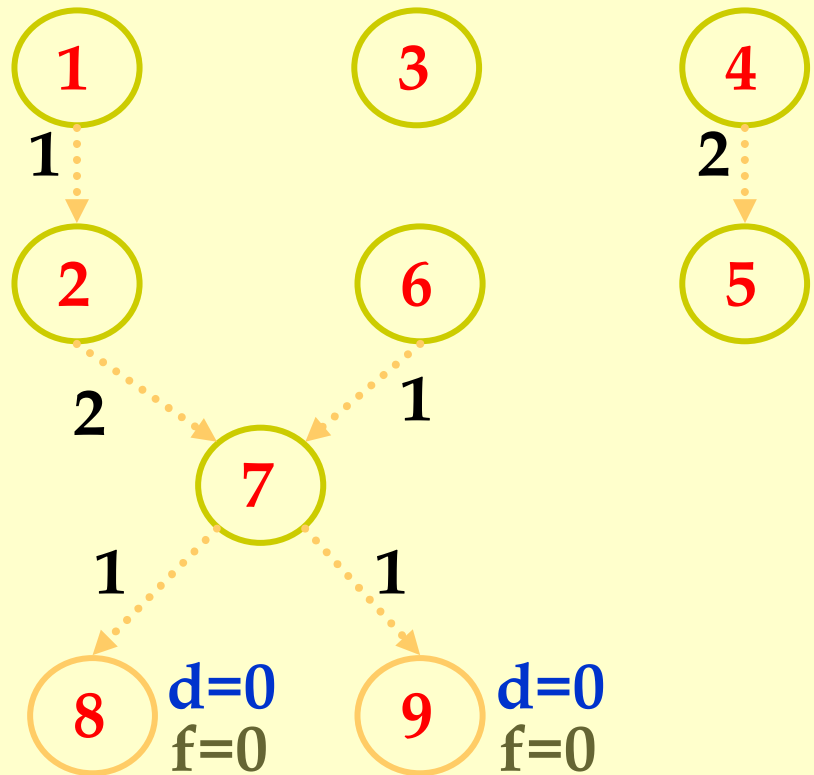


READY = { 5, 8, 9 }

ALUop	1	6	3	7			
MEM 1	4	2	5				
MEM 2		4	2				

Example

```
1: LA     r1,array
2: LD     r2,4(r1)
3: AND    r3,r3,0x00FF
4: LD     r6,8(sp)
5: ST     r7,4(r6)
6: ADD    r5,r5,100
7: ADD    r4,r2,r5
8: MUL    r5,r2,r4
9: ST     r4,0(r1)
```

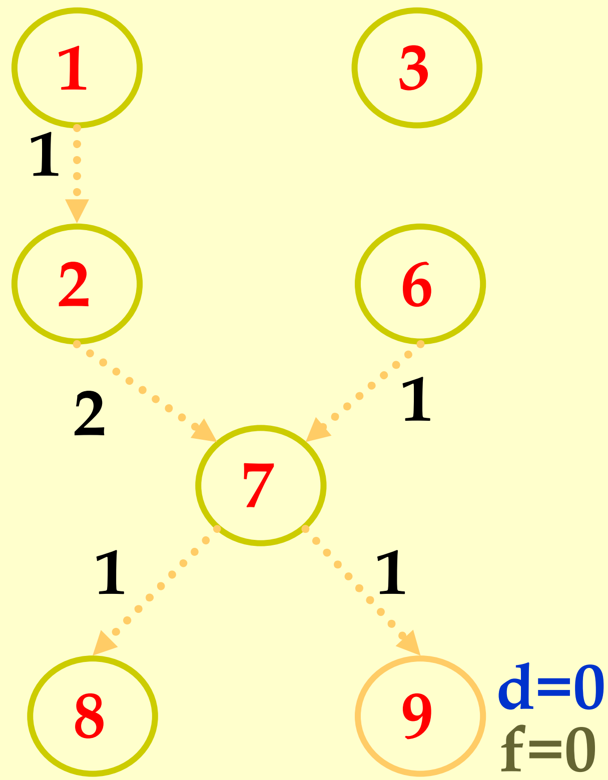


READY = { 8, 9 }

ALUop	1	6	3	7	8		
MEM 1	4	2	5				
MEM 2		4	2				

Example

```
1:  LA        r1,array
2:  LD        r2,4(r1)
3:  AND       r3,r3,0x00FF
4:  LD        r6,8(sp)
5:  ST        r7,4(r6)
6:  ADD       r5,r5,100
7:  ADD       r4,r2,r5
8:  MUL       r5,r2,r4
9:  ST        r4,0(r1)
```

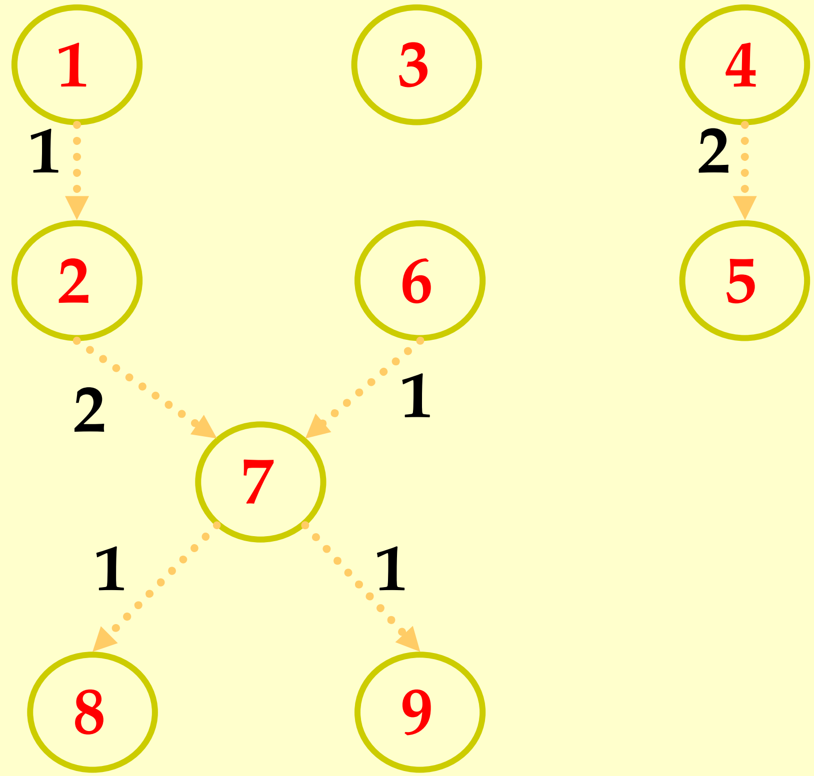


READY = { 9 }

ALUop	1	6	3	7	8		
MEM 1	4	2	5		9		
MEM 2		4	2				

Example

```
1: LA     r1,array
2: LD     r2,4(r1)
3: AND    r3,r3,0x00FF
4: LD     r6,8(sp)
5: ST     r7,4(r6)
6: ADD    r5,r5,100
7: ADD    r4,r2,r5
8: MUL    r5,r2,r4
9: ST     r4,0(r1)
```



READY = { }

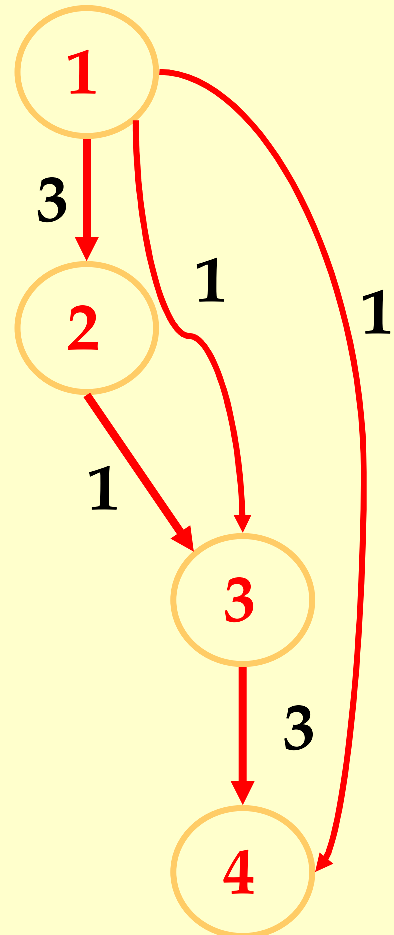
ALUop	1	6	3	7	8		
MEM 1	4	2	5		9		
MEM 2		4	2				

Register Allocation and Instruction Scheduling

- If register allocation is performed before instruction scheduling:
  - the choices for scheduling are restricted.

Example

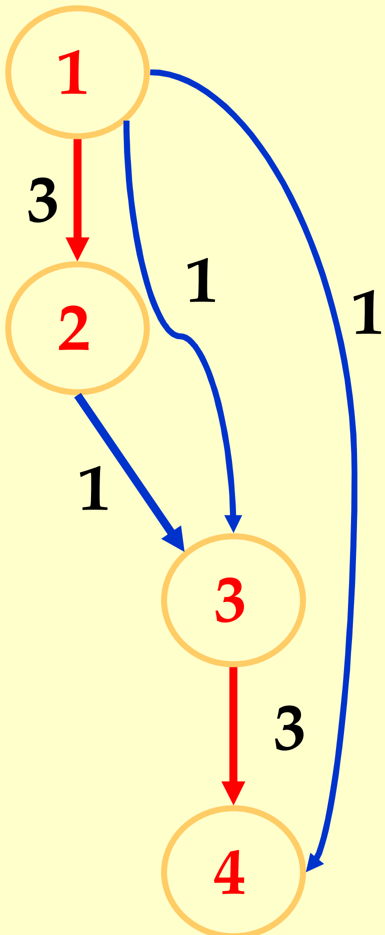
```
1:  LD      r2,0(r1)
2:  ADD     r3,r3,r2
3:  LD      r2,4(r5)
4:  ADD     r6,r6,r2
```



ALUop			2			4
MEM 1	1			3		
MEM 2		1			3	

Example

```
1: LD     r2,0(r1)
2: ADD    r3,r3,r2
3: LD     r2,4(r5)
4: ADD    r6,r6,r2
```

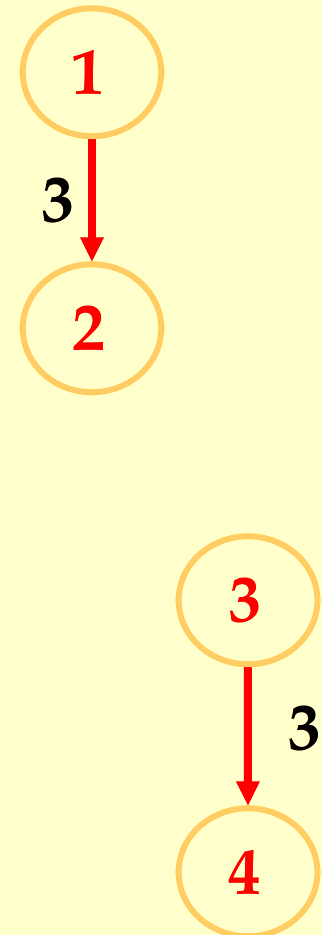


False dependencies
(Anti-dependencies)

How about using a different register?

Example

```
1: LD     r2,0(r1)
2: ADD    r3,r3,r2
3: LD     r4,4(r5)
4: ADD    r6,r6,r4
```



ALUop			2	4
MEM 1	1	3		
MEM 2		1	3	

Register Allocation and Instruction Scheduling

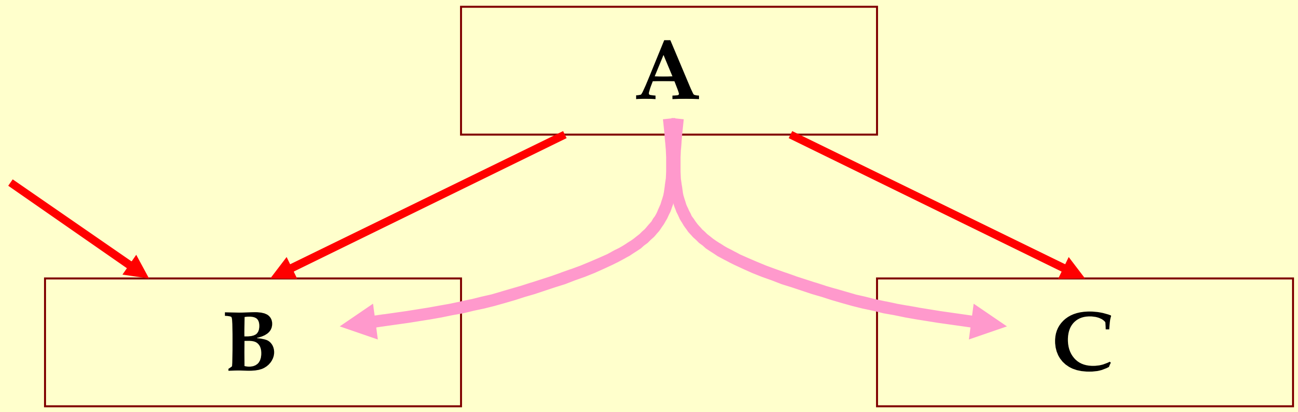
- If register allocation is performed before instruction scheduling:
  - the choices for scheduling are restricted.
- If instruction scheduling is performed before register allocation:
  - register allocation may spill registers.
  - will change the carefully done schedule!

Scheduling across basic blocks

- Number of instructions in a basic block is small.
  - Cannot keep a multiple units with long pipelines busy by just scheduling within a basic block.
- Need to handle control dependencies.
  - Scheduling constraints across basic blocks.
  - Scheduling policy.

Moving across basic blocks

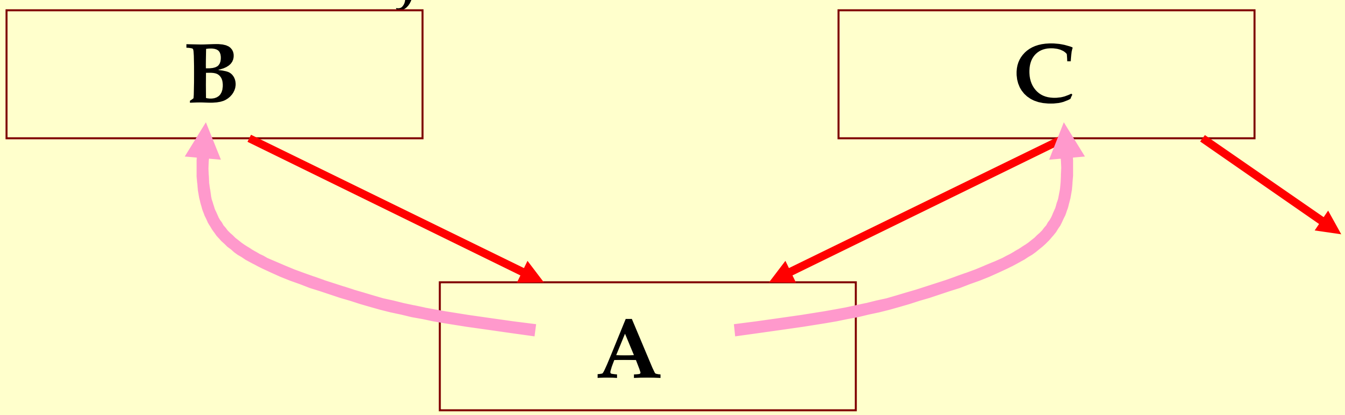
Downward to adjacent basic block



A path to **B** that does not execute **A**?

Moving across basic blocks

Upward to adjacent basic block



A path from **C** that does not reach **A**?

Control Dependencies

Constraints in moving instructions across basic blocks

```
if ( . . . )
    a = b op c
```

```
if ( . . . )
    d = *(a1)
```

Not allowed if e.g.

```
if (c != 0 )
    a = b / c
```

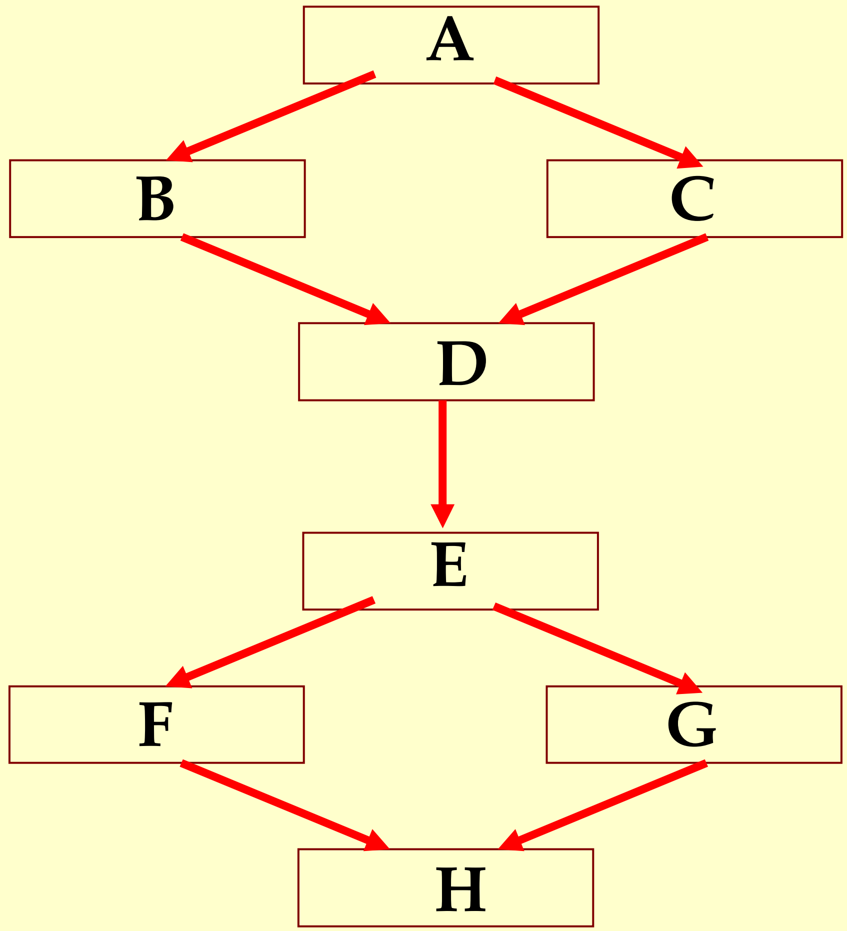
Not allowed if e.g.

```
if(valid_address(a1))
    d = *(a1)
```

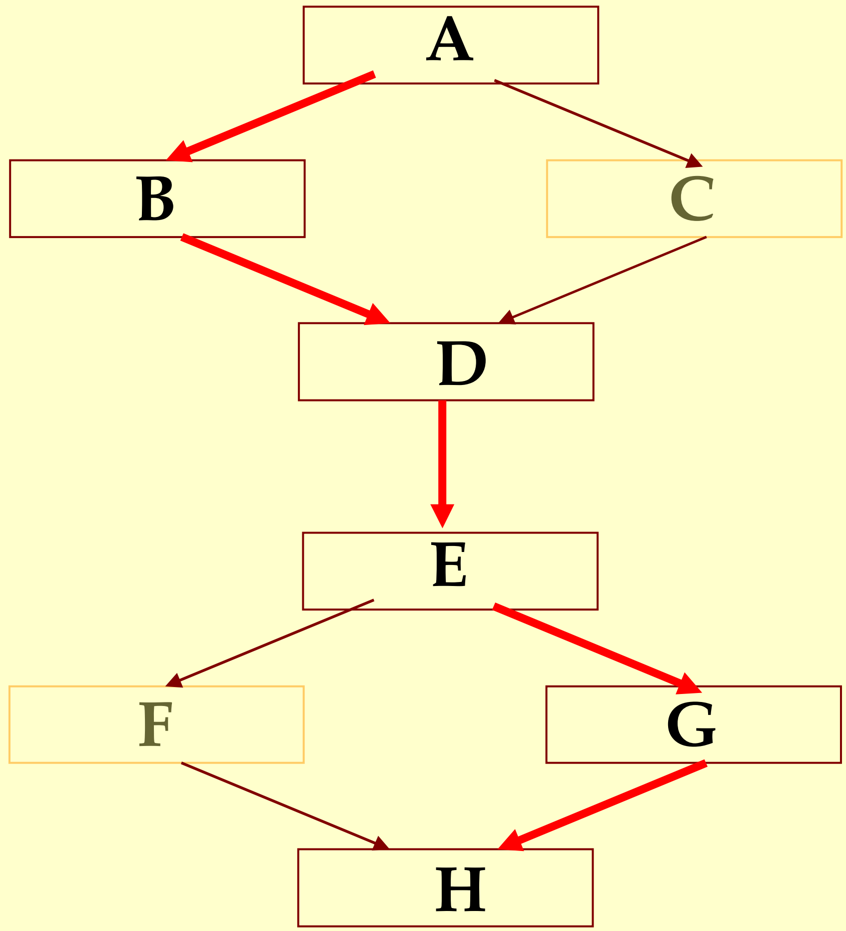
Trace Scheduling

- Find the most common trace of basic blocks.
  - Use profile information.
- Combine the basic blocks in the trace and schedule them as one block.
- Create compensating (clean-up) code if the execution goes off-trace.

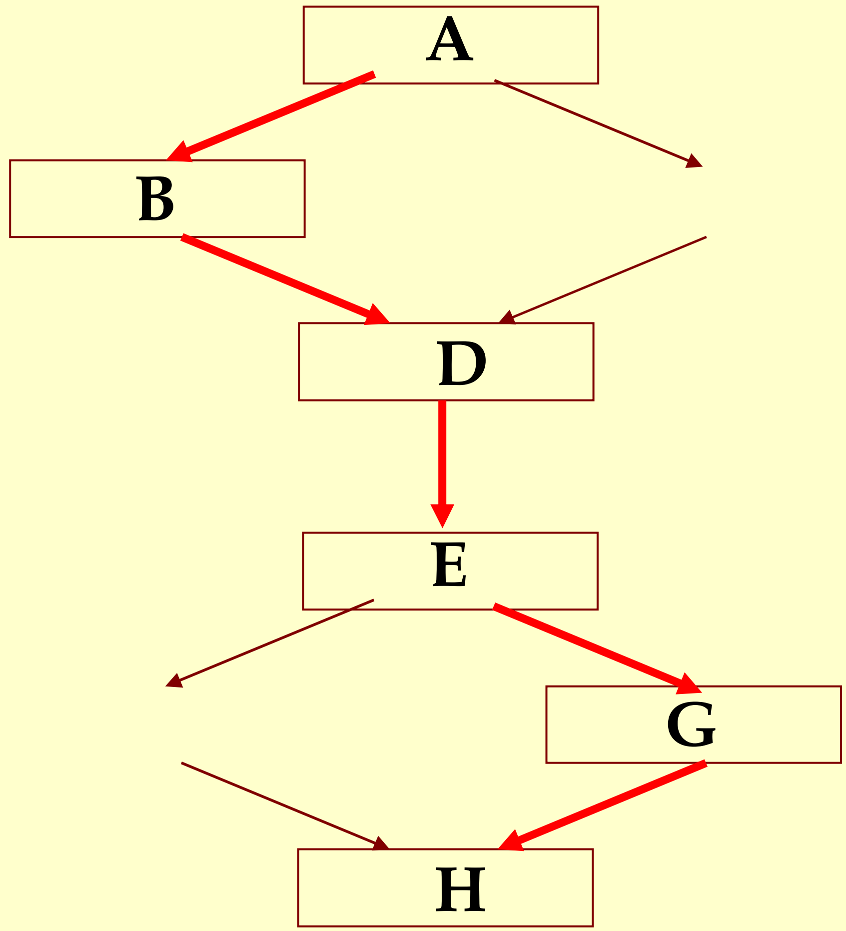
Trace Scheduling



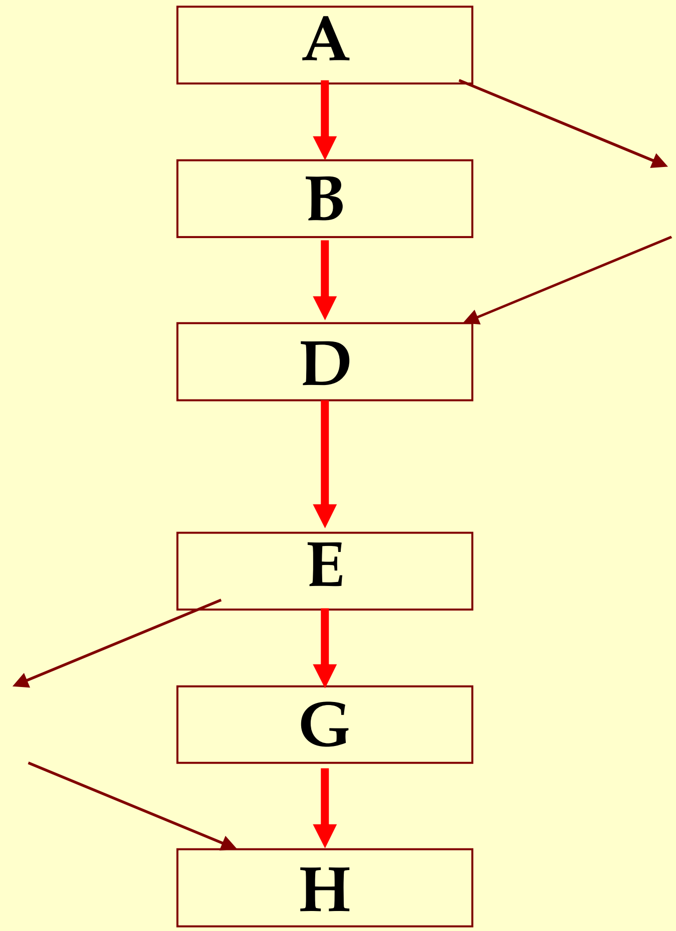
Trace Scheduling



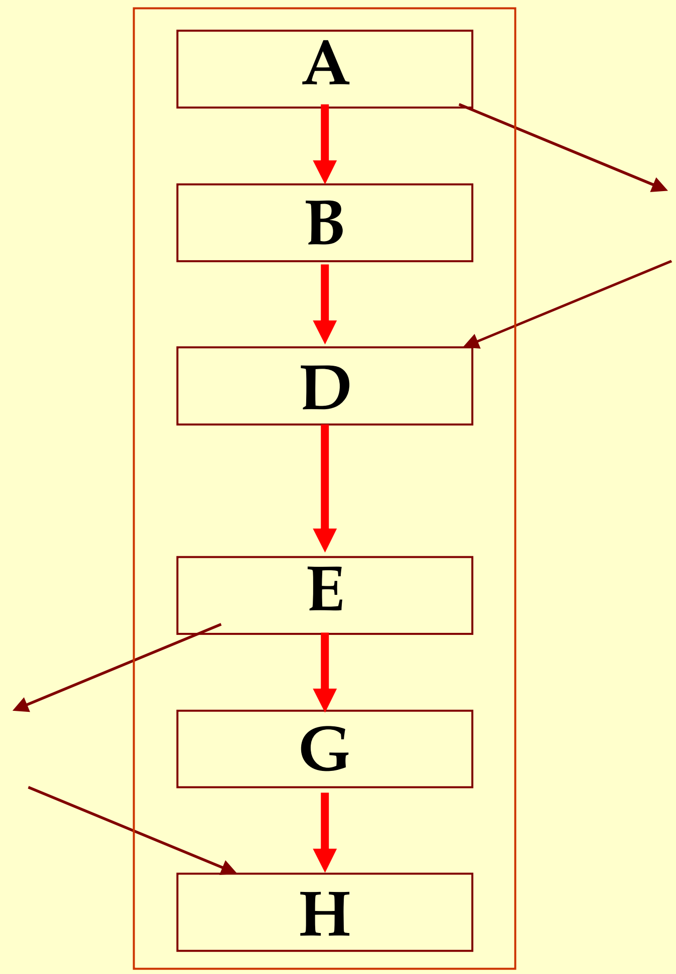
Trace Scheduling



Trace Scheduling

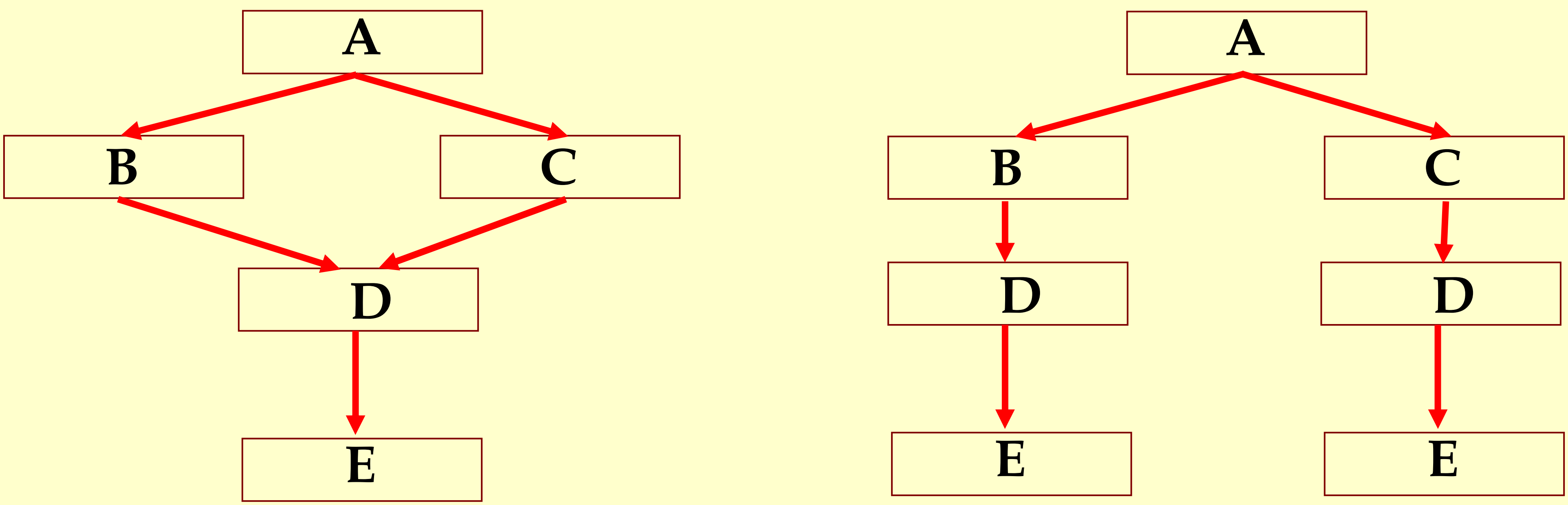


Trace Scheduling



Large Basic Blocks via Code Duplication

- Creating large extended basic blocks by duplication.
- Schedule the larger blocks.



Summary

Scheduling for Loops

- Loop bodies are typically small.
- But a lot of time is spend in loops due to their iterative nature.
- Need better ways to schedule loops.

Loop Example

Machine:

- One load/store unit
  - load 2 cycles
  - store 2 cycles
- Two arithmetic units
  - add 2 cycles
  - branch 2 cycles (no delay slot)
  - multiply 3 cycles
- Both units are pipelined (initiate one op each cycle)

Loop Example

Source Code

```
for i = 1 to N
   A[i] = A[i] * b
```

Assembly Code

```
loop:
   ld  r6, (r2)
   mul r6, r6, r3
   st  r6, (r2)
   add r2, r2, 4
   ble r2, r5, loop
```

Loop Example

<u>Assembly Code</u>

```
loop:
    ld  r6, (r2)
    mul r6, r6, r3
    st  r6, (r2)
    add r2, r2, 4
    ble r2, r5, loop
```

Schedule (9 cycles per iteration)

Mem	**ld**					**st**					
		ld					**st**				
ALU1			**mul**					**ble**			
				mul					**ble**		
					mul						
ALU2						**add**					
							add				

Loop Unrolling

Oldest compiler trick of the trade:

Unroll the loop body a few times

Pros:

- Creates a much larger basic block for the body.
- Eliminates few loop bounds checks.

Cons:

- Much larger program.
- Setup code (# of iterations < unroll factor).
- Beginning and end of the schedule can still have unused slots.

Loop Example

```
loop:
   ld  r6, (r2)
   mul r6, r6, r3
   st  r6, (r2)
   add r2, r2, 4
   ble r2, r5, loop
```

```
loop:
   ld  r6,(r2)
   mul r6, r6, r3
   st  r6,(r2)
   add r2, r2, 4
   ld  r6,(r2)
   mul r6, r6, r3
   st  r6,(r2)
   add r2, r2, 4
   ble r2, r5, loop
```

Schedule (8 cycles per iteration)

Mem	ld					st		ld				st				
		ld					st		ld				st			
ALU1			mul							mul					ble	
				mul							mul					ble
					mul							mul				
ALU2						add						add				
							add						add			

Loop Unrolling

- Rename registers.
  - Use different registers in different iterations.

Loop Example

```
loop:
   ld  r6, (r2)
   mul r6, r6, r3
   st  r6, (r2)
   add r2, r2, 4
   ld  r6, (r2)
   mul r6, r6, r3
   st  r6, (r2)
   add r2, r2, 4
   ble r2, r5, loop
```

```
loop:
   ld  r6, (r2)
   mul r6, r6, r3
   st  r6, (r2)
   add r2, r2, 4
   ld  r7, (r2)
   mul r7, r7, r3
   st  r7, (r2)
   add r2, r2, 4
   ble r2, r5, loop
```

Loop Unrolling

- Rename registers.
  - Use different registers in different iterations.
- Eliminate unnecessary dependencies.
  - again, use more registers to eliminate true, anti and output dependencies.
  - eliminate dependent-chains of calculations when possible.

Loop Example

```
loop:
   ld  r6, (r2)
   mul r6, r6, r3
   st  r6, (r2)
   add r2, r2, 4
   ld  r7, (r2)
   mul r7, r7, r3
   st  r7, (r2)
   add r2, r2, 4
   ble r2, r5, loop
```

```
loop:
   ld  r6, (r1)
   mul r6, r6, r3
   st  r6, (r1)
   add r2, r1, 4
   ld  r7, (r2)
   mul r7, r7, r3
   st  r7, (r2)
   add r1, r2, 4
   ble r1, r5, loop
```

Loop Example

```
loop:
   ld  r6, (r1)
   mul r6, r6, r3
   st  r6, (r1)
   add r2, r1, 4
   ld  r7, (r2)
   mul r7, r7, r3
   st  r7, (r2)
   add r1, r2, 4
   ble r1, r5, loop
```

```
loop:
   ld  r6, (r1)
   mul r6, r6, r3
   st  r6, (r1)
   add r2, r1, 4
   ld  r7, (r2)
   mul r7, r7, r3
   st  r7, (r2)
   add r1, r2, 4
   ble r1, r5, loop
```

Loop Example

```
loop:
   ld  r6, (r1)
   mul r6, r6, r3
   st  r6, (r1)
   add r2, r1, 4
   ld  r7, (r2)
   mul r7, r7, r3
   st  r7, (r2)
   add r1, r2, 4
   ble r1, r5, loop
```

```
loop:
   ld  r6, (r1)
   mul r6, r6, r3
   st  r6, (r1)
   add r2, r1, 4
   ld  r7, (r2)
   mul r7, r7, r3
   st  r7, (r2)
   add r1, r1, 8
   ble r1, r5, loop
```

Loop Example

```
loop:
   ld  r6, (r1)
   mul r6, r6, r3
   st  r6, (r1)
   add r2, r1, 4
   ld  r7, (r2)
   mul r7, r7, r3
   st  r7, (r2)
   add r1, r1, 8
   ble r1, r5, loop
```

Schedule (4.5 cycles per iteration)

Mem	ld		ld			st		st			
		ld		ld			st		st		
ALU1			mul		mul			ble			
				mul		mul			ble		
					mul		mul				
ALU2	add					add					
		add					add				

Software Pipelining

- Try to overlap multiple iterations so that the slots will be filled.
- Find the steady-state window so that:
  - all the instructions of the loop body is executed.
  - but from different iterations.

Loop Example

Assembly Code

```
loop:
  ld    r6, (r2)
  mul   r6, r6, r3
  st    r6, (r2)
  add   r2, r2, 4
  ble   r2, r5, loop
```

Schedule

ld		ld1		ld2	st	ld3	st1	ld4	st2	ld5	st3	ld6
	ld		ld1		ld2	st	ld3	st1	ld4	st2	ld5	st3
		mul		mul1		mul2	ble	mul3	ble1	mul4	ble2	mul5
			mul		mul1		mul2	ble	mul3	ble1	mul4	ble2
				mul		mul1		mul2		mul3		mul4
					add		add1		add2		add3	
						add		add1		add2		add3

Loop Example

Assembly Code

```
loop:
  ld    r6, (r2)
  mul   r6, r6, r3
  st    r6, (r2)
  add   r2, r2, 4
  ble   r2, r5, loop
```

Schedule (2 cycles per iteration)

ld3	st1
st	ld3
mul2	ble
	mul2
mul1	
	add1
add	

Loop Example

4 iterations are overlapped.

- values of r3 and r5 don't change
- 4 regs for &A[i] (r2)
- each addr. incremented by 4*4
- 4 regs to keep value A[i] (r6)
- Same registers can be reused after 4 of these blocks generate code for 4 blocks, otherwise need to move .

ld3	st1
st	ld3
mul2	ble
	mul2
mul1	
	add1
add	

```
loop:
  ld  r6, (r2)
  mul r6, r6, r3
  st  r6, (r2)
  add r2, r2, 4
  ble r2, r5, loop
```

Software Pipelining

- Optimal use of resources.
- Need a lot of registers.
  - Values in multiple iterations need to be kept.
- Issues in dependencies.
  - Executing a store instruction in an iteration before branch instruction is executed for a previous iteration (writing when it should not have).
  - Loads and stores are issued out-of-order (need to figure-out dependencies before doing this).
- Code generation issues.
  - Generate pre-amble and post-amble code.
  - Multiple blocks so no register copy is needed.