Code Scheduling

Code Scheduling

List Scheduling

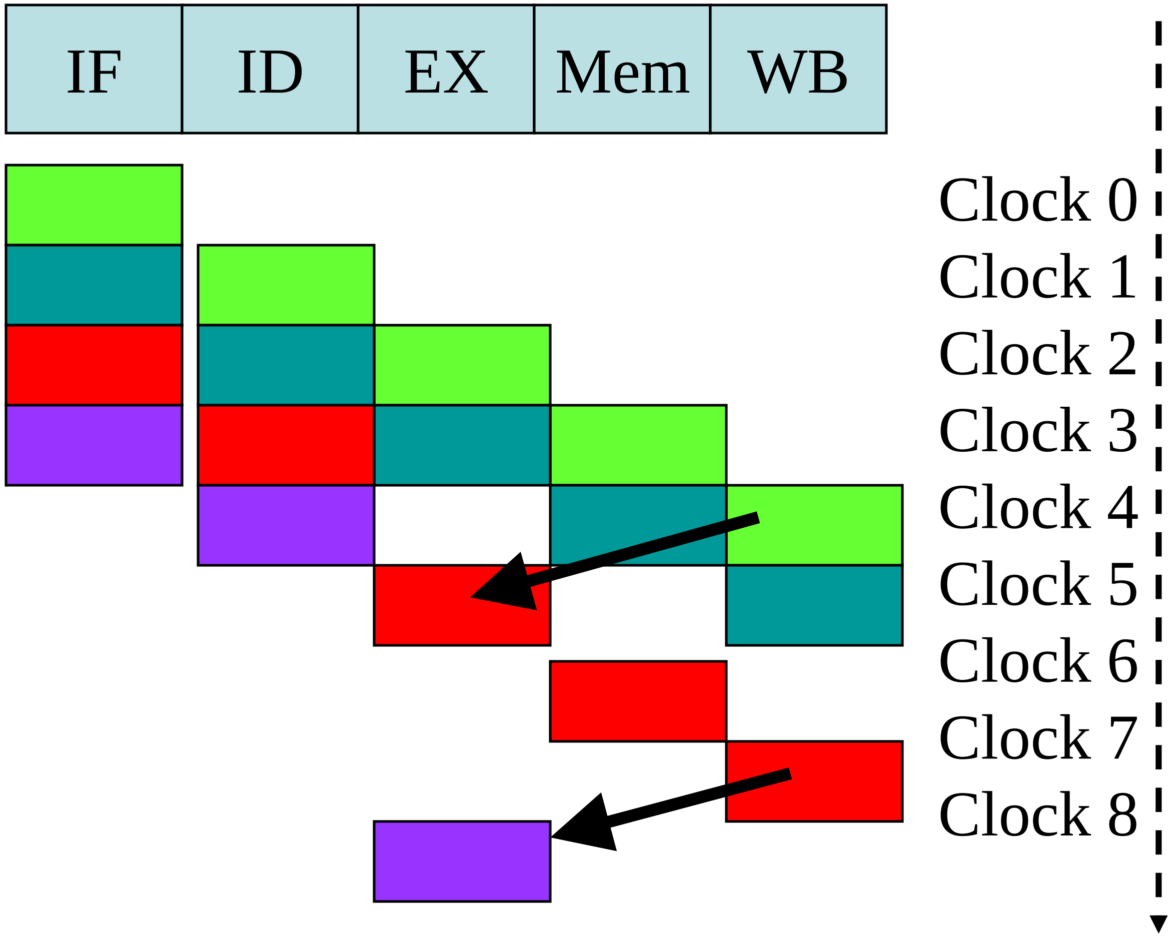
Prepass and Postpass Scheduling

Loop Unrolling

Software Pipelining

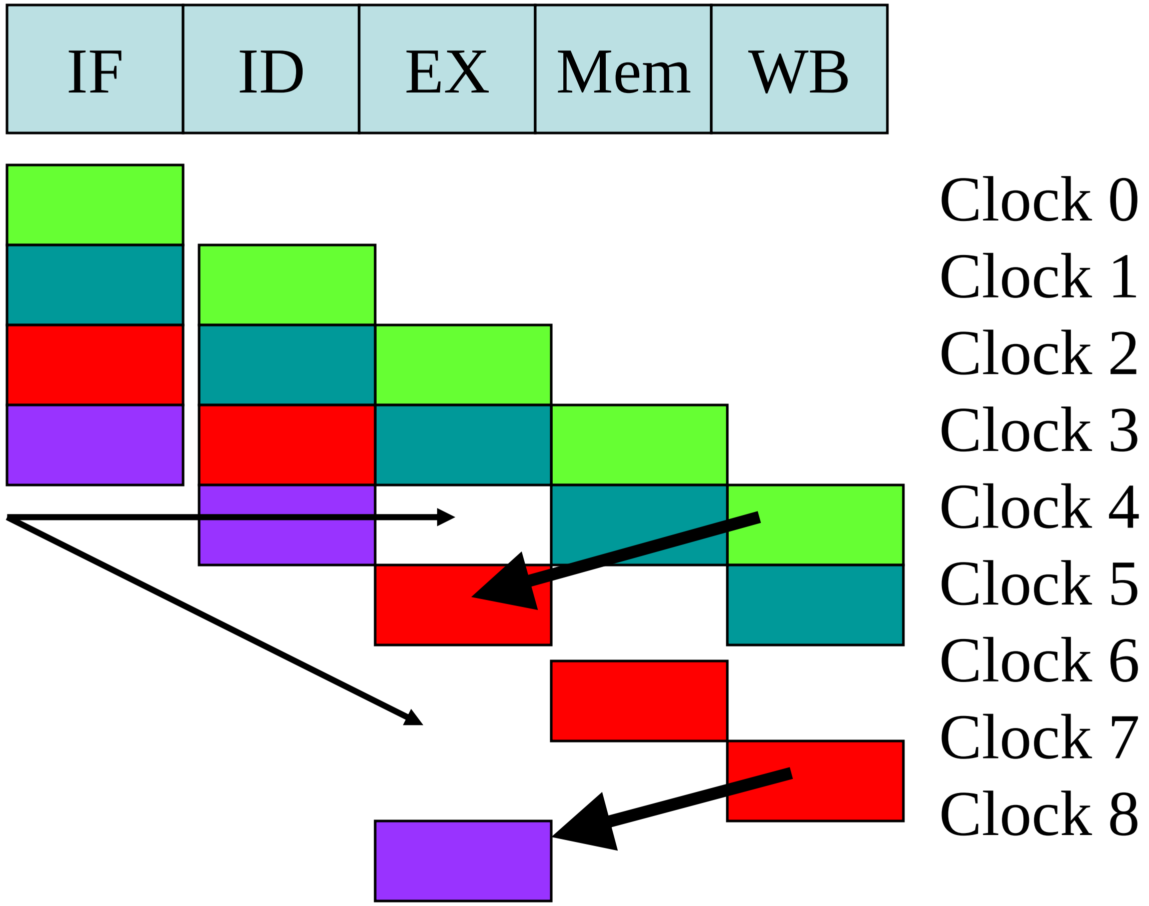
Why Rearrange Code Sequence?

```
lw $8, a
lw $9, b
mul $10,$8,$8
add $11,$9,$10
lw $12,c
add $13,$12,$0
beq $11,$0,L1
```



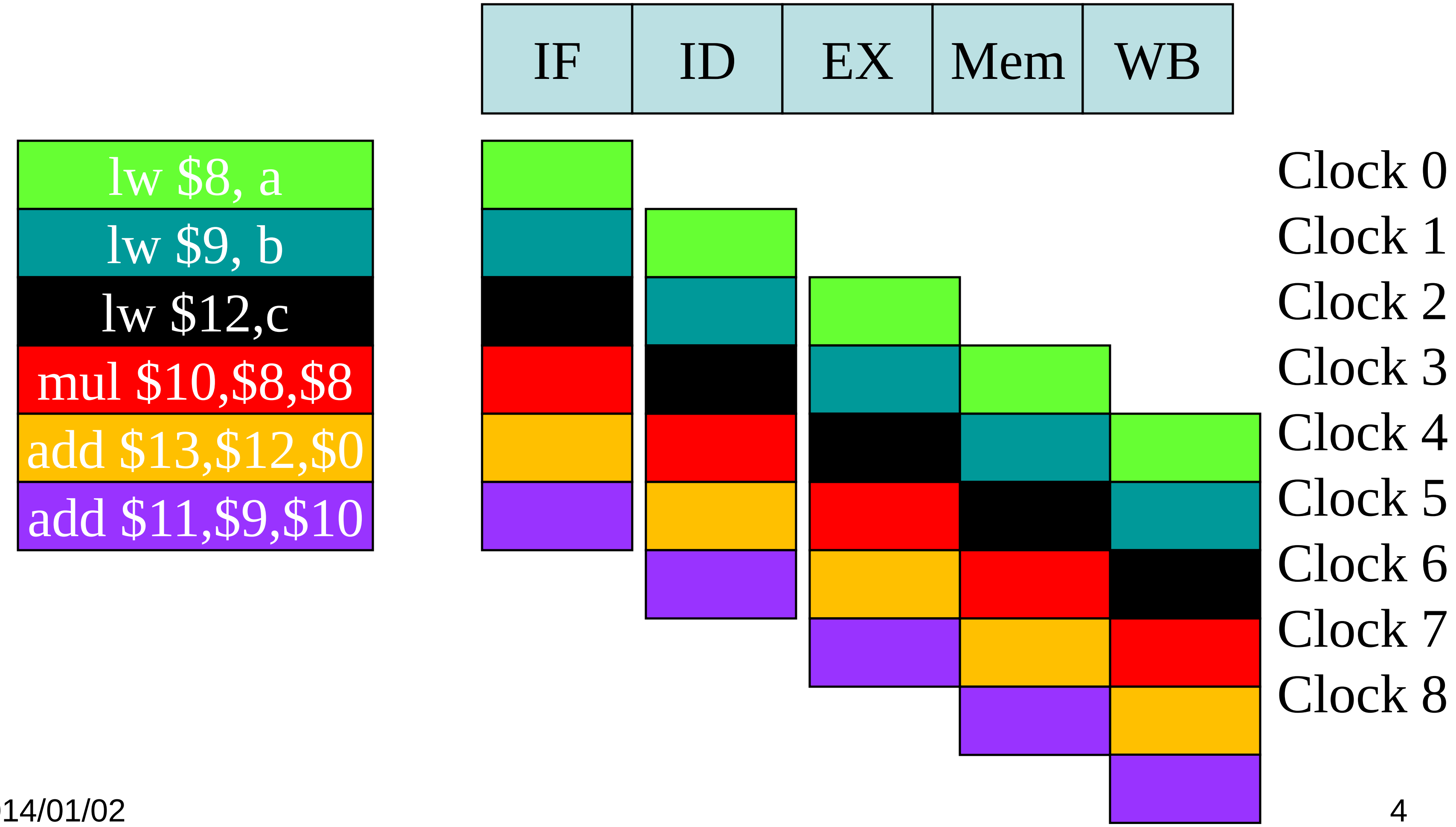
Why Rearrange Code Sequence?

Data and *Control dependences* may generate *bubbles* in the pipeline, reducing execution efficiency



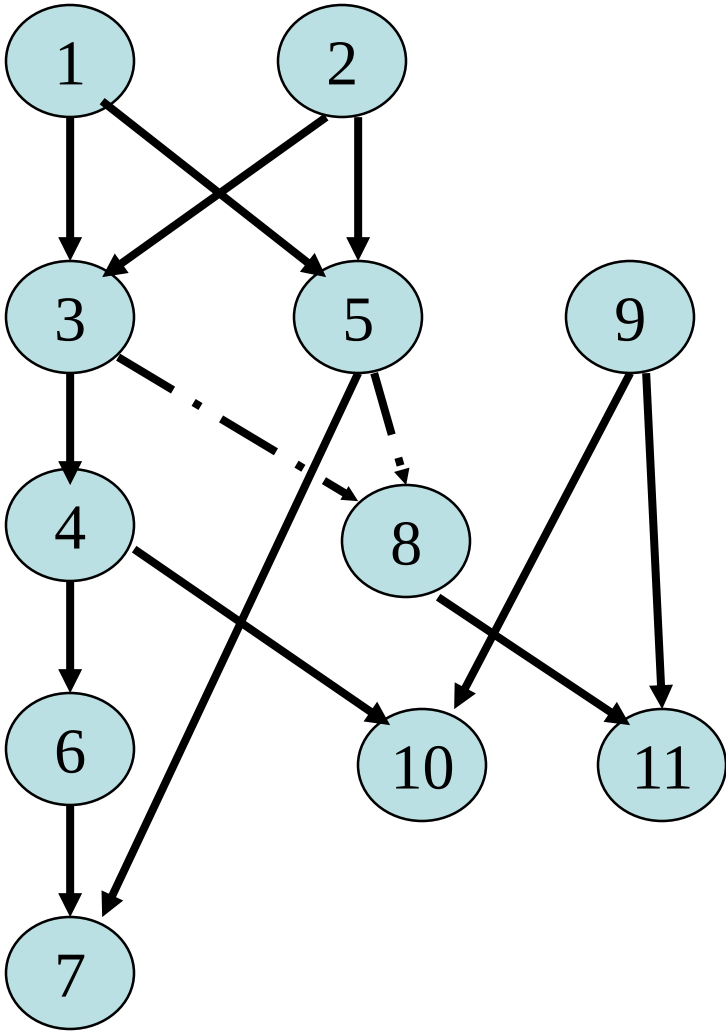
Why Rearrange Code Sequence?

- More efficient execution after rescheduling



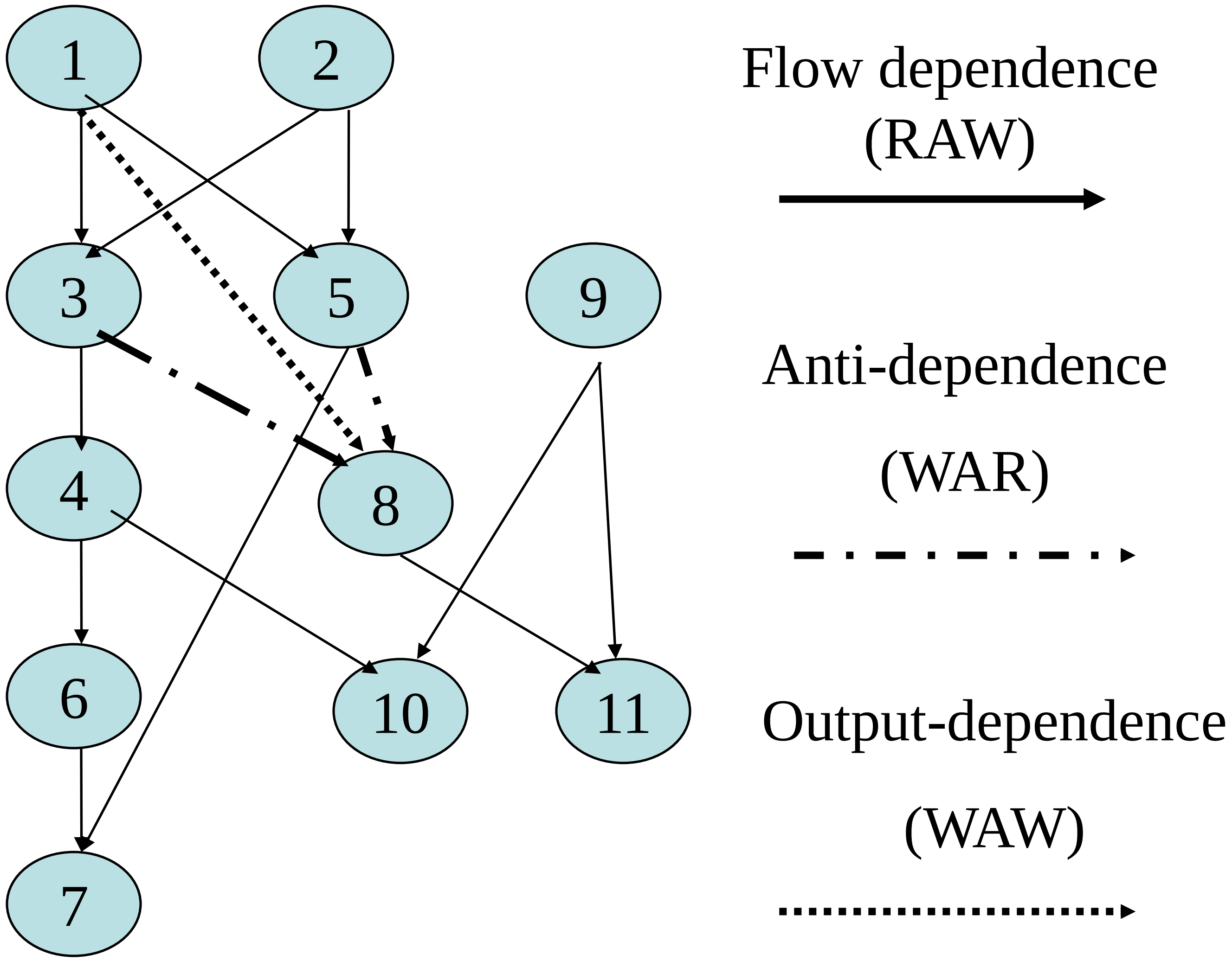
Dependence DAG

```
1    lw $pr1, a
2    lw $pr2, b
3    add $pr3, $pr1,$pr2
4    mul $pr4, $pr3, 3
5    mul $pr5, $pr1, $pr2
6    add $pr6, $pr4, 2
7    sw  $pr6, 0($pr5)
8    lw $pr1, c
9    lw $pr7, d
10  sw $pr7, 0($pr4)
11  add $pr8,$pr7,$pr1
```



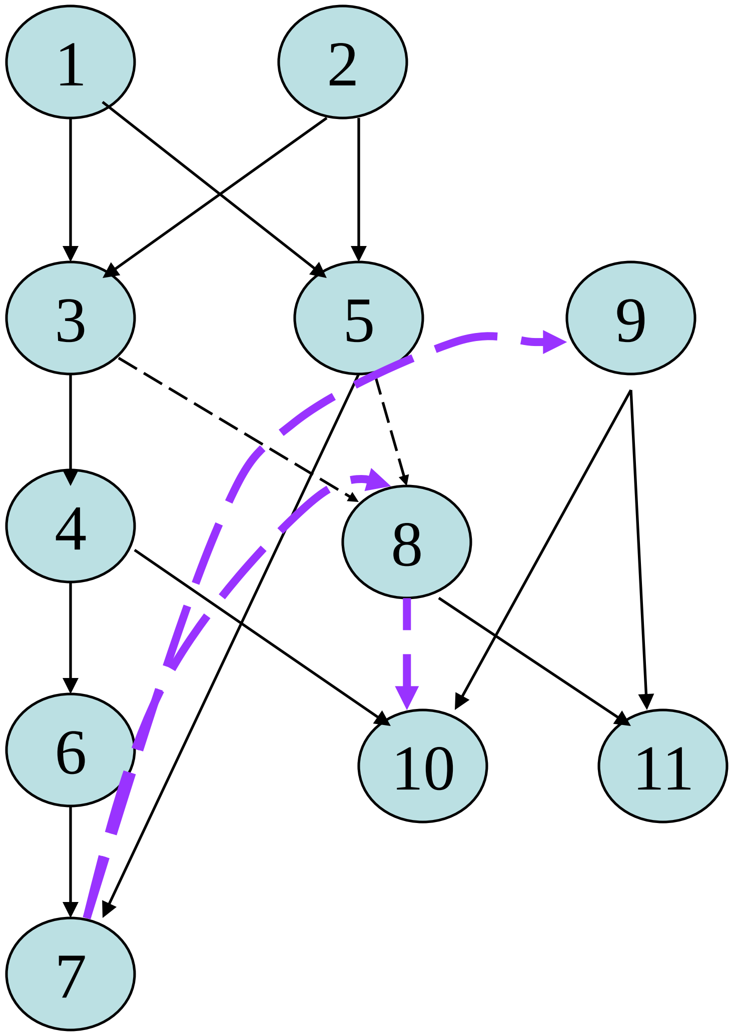
Each node represents an instruction
Each arc represents a dependence

Dependence Arcs

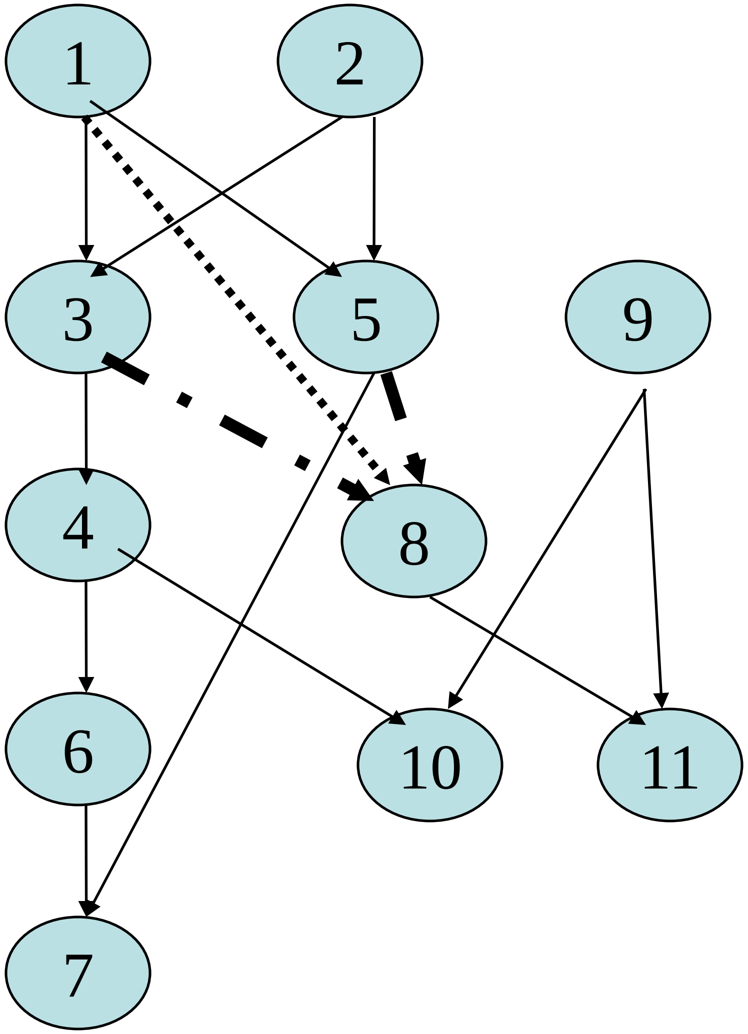


Possible Memory Dependency
(due to potential aliasing)

```
1   lw $pr1, a
2   lw $pr2, b
3   add $pr3, $pr1,$pr2
4   mul $pr4, $pr3, 3
5   mul $pr5, $pr1, $pr2
6   add $pr6, $pr4, 2
7   sw  $pr6, 0($pr5)
8   lw $pr1, c
9   lw $pr7, d
10  sw $pr7, 0($pr4)
11  add $pr8,$pr7,$pr1
```



Legal Schedules

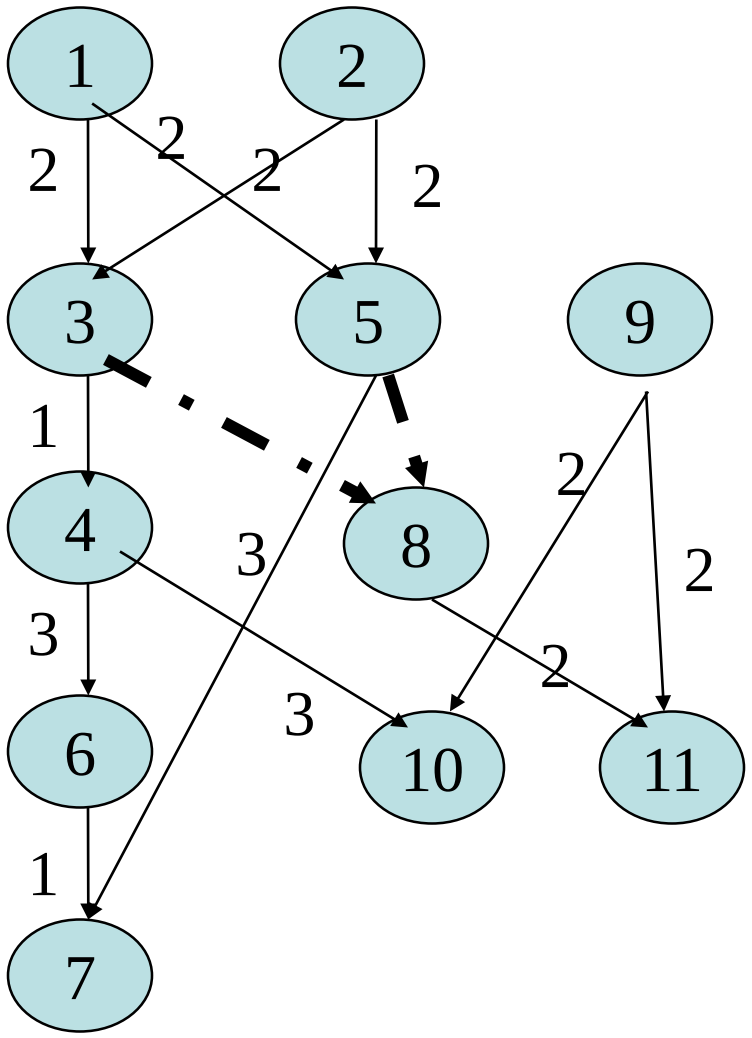


Any *topological* sort of the DAG will represent a correct schedule.

Topological sort is the enumeration of the nodes in which *each node appears before its children*.

Some legal schedules:
1,2,3,5,9,4,8,6,10,11,7
9,1,2,3,4,10,6,5,8,7,11

Weight of Arcs



Weight indicates the *operation latency*

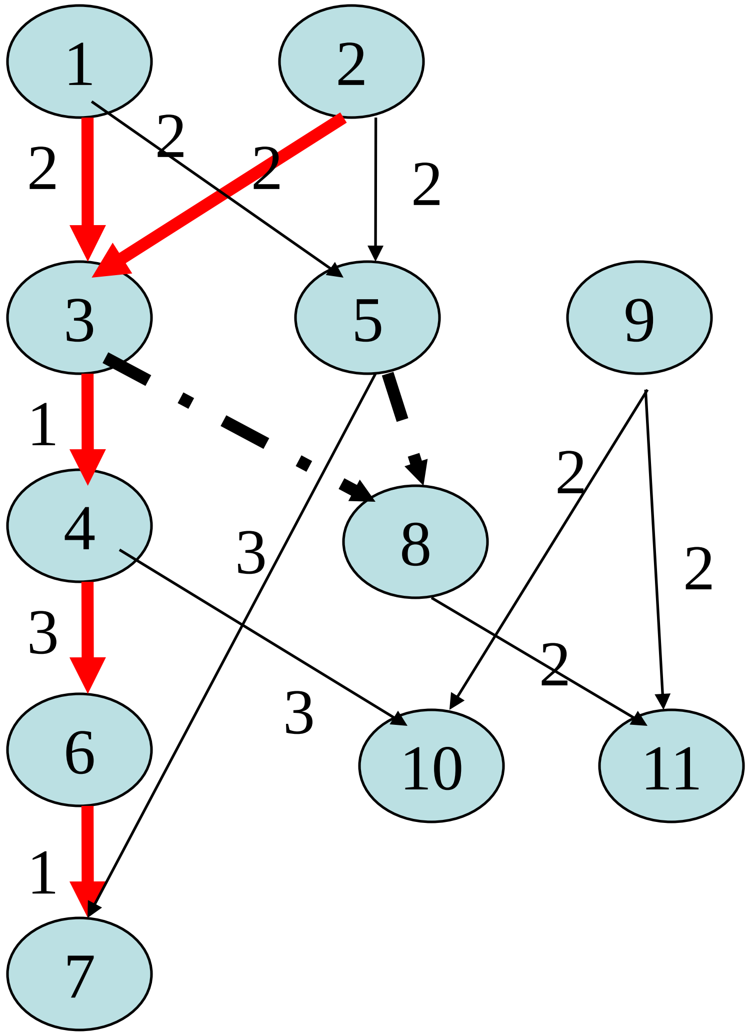
Here we assume a

Load takes 2 cycles

Multiply takes 3 cycles

Add takes 1 cycle

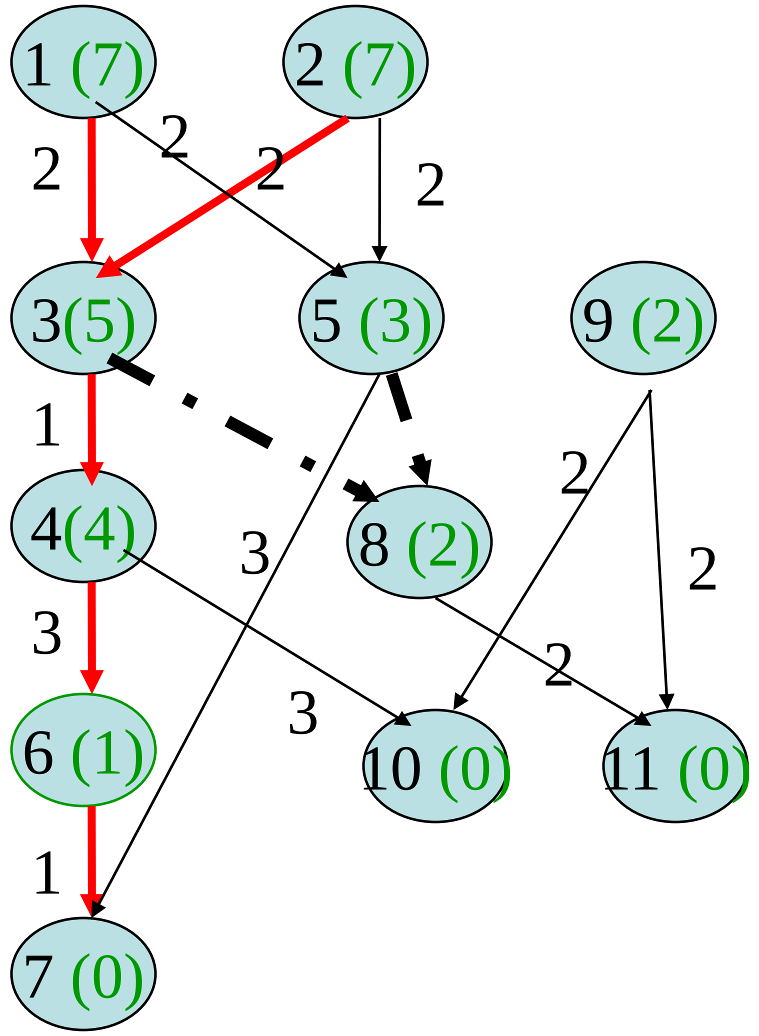
Critical Path



General guideline:

Nodes on the critical paths
should be scheduled
as early as possible

Mark each node with maximum delay



Traversing the DAG from the leaves towards the roots labeling each node with the maximum possible delay from that node to the end of the block. (reverse topological sort)

Maximum delay of each node is computed as:

MD = max (each child's MD + arc_latency)

List Scheduling

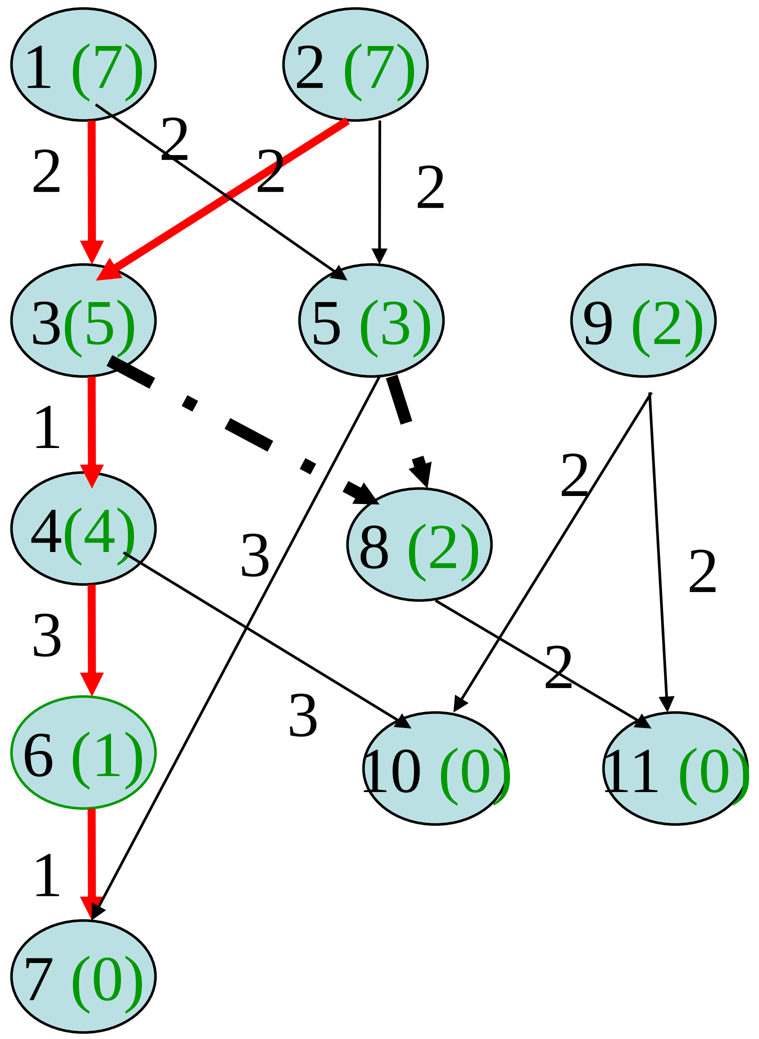
Basic Block scheduling

1. Build Dependence DAG
2. Mark each node with maximum delay to the end of block
3. Traverse the DAG from the root toward the leaves, selecting nodes from a candidate list to schedule. Keep tracking of the current cycle to minimize possible stalls.

Commonly used heuristics

- The following heuristics are often used
  - Favor nodes that can start without stalling
  - If there is a tie, favor nodes with the maximum delay to the end of block
  - If there is a tie,
    - Favor the original order
    - Favor nodes that have a large number of children
    - Favor nodes that are the final use of a reg

Example



Candidate list ={1, 2, 9}
Select 1,
Select 2,
Clist = {3,5,9}
Select 9 – why?
Clist = {3,5}
Select 3 – why?
Clist = {4,5}
Select 4
Clist = {6,10,5}
Select 5
Clist = {6,10,8}
Select 8
Clist = {6,10,11}
Select 6

Execution Cycle Comparison

Original code

Issue cycle		
0	1	lw $pr1, a
1	2	lw $pr2, b
3	3	add $pr3, $pr1,$pr2
4	4	mul $pr4, $pr3, 3
5	5	mul $pr5, $pr1, $pr2
7	6	add $pr6, $pr4, 2
8	7	sw $pr6, 0($pr5)
9	8	lw $pr1, c
10	9	lw $pr7, d
12	10	sw $pr7, 0($pr4)
13	11	add $pr8,$pr7,$pr1

Scheduled code

Issue cycle		
0	1	lw $pr1, a
1	2	lw $pr2, b
2	9	lw $pr7, d
3	3	add $pr3, $pr1,$pr2
4	4	mul $pr4, $pr3, 3
5	5	mul $pr5, $pr1, $pr2
6	8	lw $pr1, c
7	6	add $pr6, $pr4, 2
8	10	sw $pr7, 0($pr4)
9	7	sw $pr6, 0($pr5)
10	11	add $pr8,$pr7,$pr1

PrePass Scheduling

- PrePass scheduling means code scheduling before the Register Assignment phase. Scheduling is based on pseudo registers.
- PrePass scheduling tends to increase pseudo register lifetime, thus may require more registers than what is available. This may cause register spilling at assignment time.

PrePass Scheduling Example

```
1   lw $pr1, a
2   lw $pr2, b
3   add $pr3, $pr1,$pr2
4   mul $pr4, $pr3, 3
5   mul $pr5, $pr1, $pr2
6   add $pr6, $pr4, 2
7   sw  $pr6, 0($pr5)
8   lw $pr7, c
9   lw $pr8, d
10  sw $pr8, 0($pr4)
11  add $pr9,$pr8,$pr7
```

```
1   lw $pr1, a
2   lw $pr2, b
3   lw $pr7, c
4   lw $pr8, d
5   add $pr3, $pr1,$pr2
6   mul $pr4, $pr3, 3
7   mul $pr5, $pr1, $pr2
8   add $pr6, $pr4, 2
9   sw $pr8, 0($pr4)
10  add $pr9,$pr8,$pr7
11  sw  $pr6, 0($pr5)
```

Need 6 or more registers

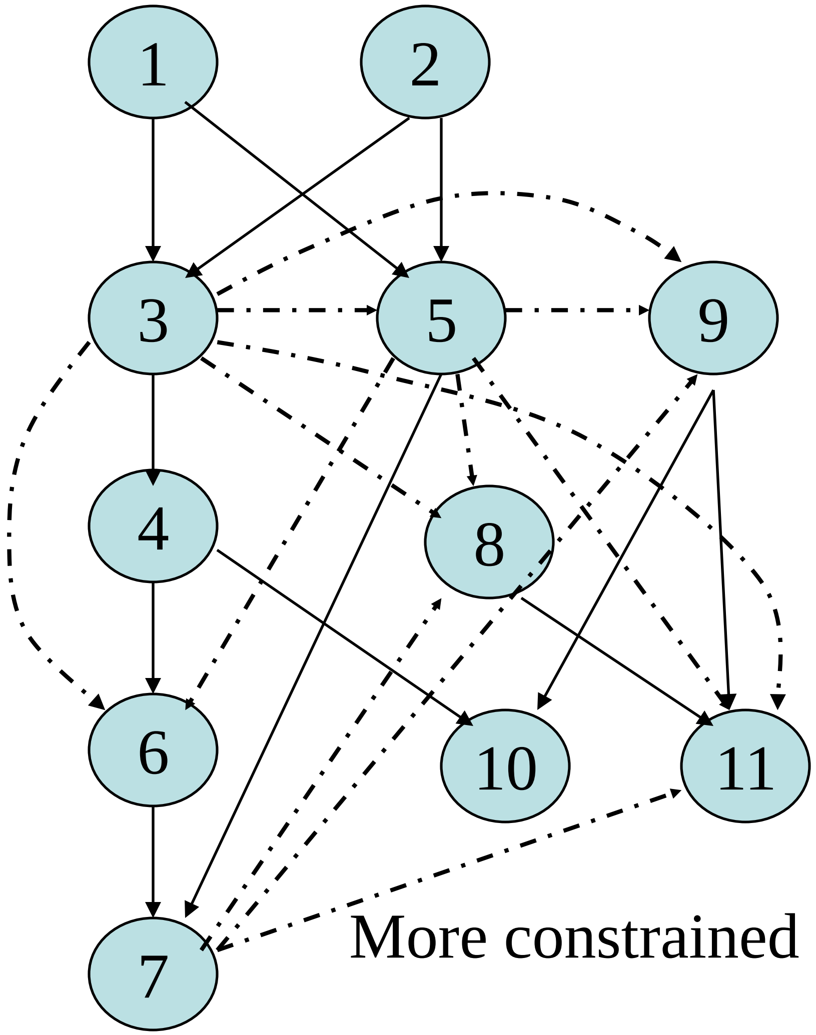
PostPass Scheduling

- PostPass scheduling means code scheduling after the Register Assignment phase. Scheduling is based on physical registers.
- PostPass scheduling is often less effective since the register assignment phase may assign the same physical register to unrelated instructions, introducing new dependency constraints.

PostPass Scheduling Example

```
1   lw $r1, a
2   lw $r2, b
3   add $r3, $r1,$r2
4   mul $r3, $r3, 3
5   mul $r1, $r1, $r2
6   add $r2, $r3, 2
7   sw  $r2, 0($r1)
8   lw $r1, c
9   lw $r2, d
10  sw $r2, 0($r3)
11  add $r1,$r2,$r1
```

Three registers are used



More constrained DAG

Compromised Scheduling

- Two pass scheduling
  - PrePass scheduling followed by another round of scheduling (i.e. postpass scheduling) to hide latency of spilled instructions.
- Integrated scheduling
  - Schedule to hide latency when registers are available
  - Switch to schedule for register pressure reduction when available registers are running low.
- DAG reshaping
  - Balance the shape of a DAG, if too fat, trim some branches to trade width with height.

Loop Unrolling

- Basic block scheduling is often very limited

```
Loop:
      a[j] = b[j] * c;
end;
      lw     $8, b($9)    // $9 holds the loop index
      mul    $8,$8,$10    // assuming c is in $10
      sw     $8, a($9)
      add    $9, 4              Not much we can schedule!!
      blt    $9, N, loop
```

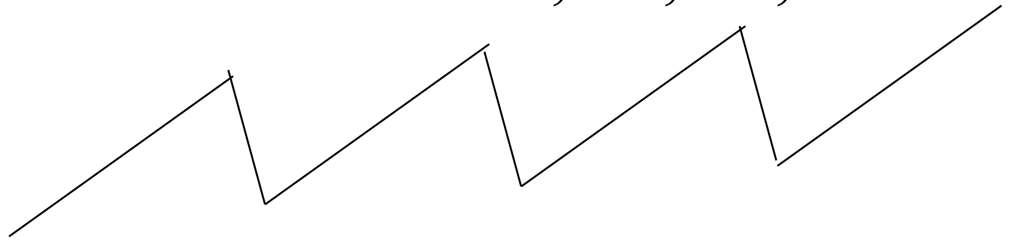
Loop Unrolling (cont.)

- Unrolling the loop creates many more independent code sequences for scheduling.

```
lw     $t1, b($s1)
mul    $t1,$t1,$s2
sw     $t1, a($s1)
lw     $t2, (b+4)($s1)
mul    $t2,$t2,$s2
sw     $t2, (a+4)($s1)
add    $s1, 8
```

```
lw     $t1, b($s1)
lw     $t2, (b+4)($s1)
mul    $t1,$t1,$s2
mul    $t2,$t2,$s2
sw     $t1, a($s1)
sw     $t2, (a+4)($s1)
add    $s1, 8
```

- Shortcomings of loop unrolling
  - Code expansion
  - Increased scheduling time (note that DAG building time is ~O(N*N))
  - Saw tooth effect: if a loop is unrolled 8 times, the speed up will be much lower when the iteration count is 7, 15, 23, … etc.



  - Start-up and shut down cost is not effectively hidden

Software Pipelining

- Software pipelining works by allowing parts of several iterations of a loop to be in process at the same time.

Example:

loop:

```
ld.s    $f10, a($t0)
add.s  $f8, $f10,$f9
st.s    $f8, b($t0)
add    $t0, $t0, 4
blt     $t0, 400, loop
```

$\leftarrow$ To hide the long fp latency, the loop needs to unroll many times.

Assume ld.s and add.s each takes 2 cycles

Execution cycles of example code

cycle	instruction
0	ld.s $f10, a($t0)
1	
2	add.s $f8, $f10,$f9
3	
4	st.s $f8, b($t0)
5	add $t0, $t0, 4
6	blt $t0, 400, loop
7	ld.s $f10, a($t0)
8	

Two bubles per iteration

Software Pipelining (cont.)

Example:

```
        ld.s     $f10,    a($t0)
        add.s  $f8, $f10,$f9
        ld.s     $f10, (a+4)($t0)
loop:
        st.s     $f8, b($t0)
        add.s   $f8, $f10,$f9
        ld.s     $f10, (a+8)($t0)
        add      $t0, $t0, 4
        blt      $t0, 392, loop
```

cycle	instruction
0	add.s $f8, $f10,$f9
1	ld.s $f10, (a+4)($t0)
2	st.s $f8, b($t0)
3	add.s $f8, $f10,$f9
4	ld.s $f10, (a+8)($t0)
5	add $t0,$t0,4
6	blt $t0,392,loop
7	st.s $f8, b($t0)
8	add.s $f8, $f10,$f9

Scheduling for Memory Ops

- Should the compiler schedule for cache hit or cache miss?
  - Schedule for hit: Loads hit in cache more often than miss. Schedule for miss will use many more registers than necessary.
  - Schedule for miss: Cache miss penalty is high. Schedule for hit may miss the opportunity for overlapping misses.
- Miss Profile driven scheduling – problems?
- Balanced scheduling – schedule for miss unless the register pressure is high.

Scheduling for OOO machines

Is scheduling for OOO machines simpler than for in-order issue machines?

- *Intuitively, it seems simpler, but it is more difficult to get the best performance*
- *Whether the compiler should schedule for latency hiding or retirement is dependent on whether the reorder buffer (i.e. the rename queue) is full or nearly empty (when branch mispredictions happen).*
- *The compiler must make sure there are sufficient independent operations in the re-order buffer, so it still needs to interleave instructions from independent trees (or sequences)*

Summary

- Code Scheduling improves execution throughput on pipelined/superscalar processors.
- Scheduling process
  - DAG
  - Critical path
  - List scheduling
- Loop unrolling to expose more ILP
- Software pipelining to achieve higher overlap