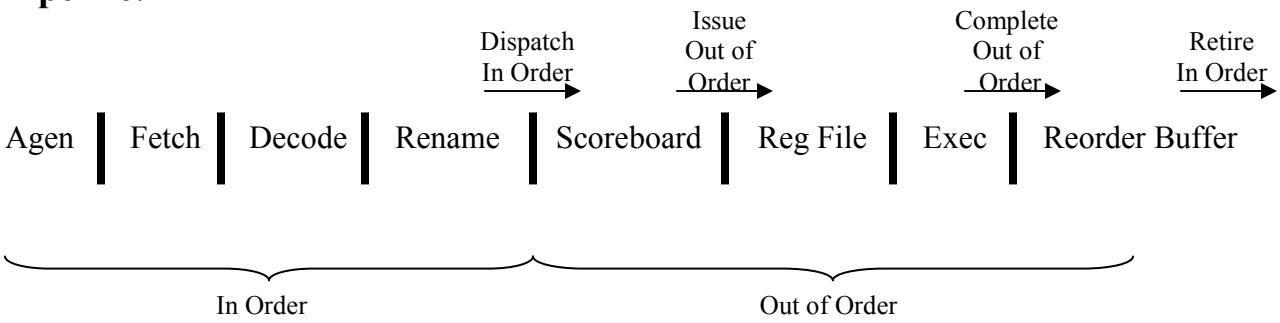
ECE 511
Computer Architecture
9/22/04

Out of Order Execution

Pipeline:



First Ridiculous Assumption:

The instruction stream contains no branches, loads, or stores.

Given the following C-Code:

```
x = (a * b) - (c + d)
y = (r + s) / (t + v)
```

The compiler generates:

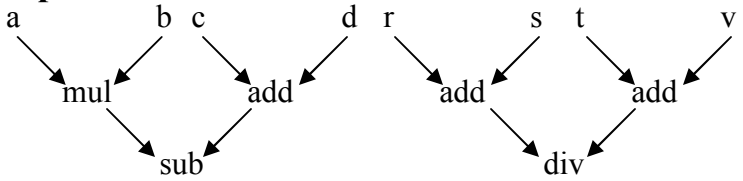
instr:	code:		
1	mul	$R_m \leftarrow$	R_a, R_b
2	add	$R_n \leftarrow$	R_c, R_d
3	sub	$R_x \leftarrow$	R_m, R_n
4	add	$R_m \leftarrow$	R_r, R_s
5	add	$R_n \leftarrow$	R_t, R_v
6	div	$R_y \leftarrow$	R_m, R_n

Where R_i are compiler generated temporary variables that represent the letter i for all i

Assume the Scoreboard and the Reorder Buffer are implemented with queues.
Furthermore, let instruction 1 be at the head of the scoreboard queue and instruction 7 at the tail.
Instructions 1 and 2 are immediately legal to execute, while 4 and 5 are bad choices due to dependences.

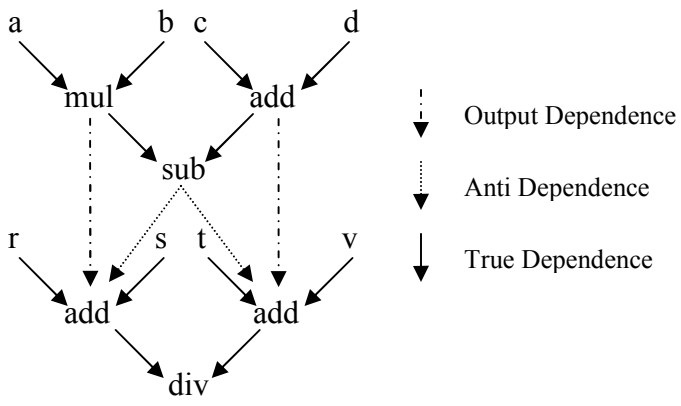
Dependence Graphs:

Ideal:



But reuse of temporary variables creates restrictions on concurrency!

Actual:



Artificial, or Storage, Dependence:

A dependence created by the reuse of a machine resource.
An output or anti dependence.

Can't ignore dependences so avoid or remove them:

First Attempt (way too conservative)

The only instruction legal to execute is the one at the head of the scoreboard, and then only if all previous instructions have completed.

This guarantees the register file always has a consistent set of values; that is, consistent with the values that would be present in an in-order execution register file at the same point.

Assumptions:

1) mul takes 10 cycles, add takes 5, sub takes 5, div takes 20
2) Execute is fully pipelined (i.e. an instruction can issue every cycle)
3) When an instruction completes, the register files is written and the reorder buffer is marked.

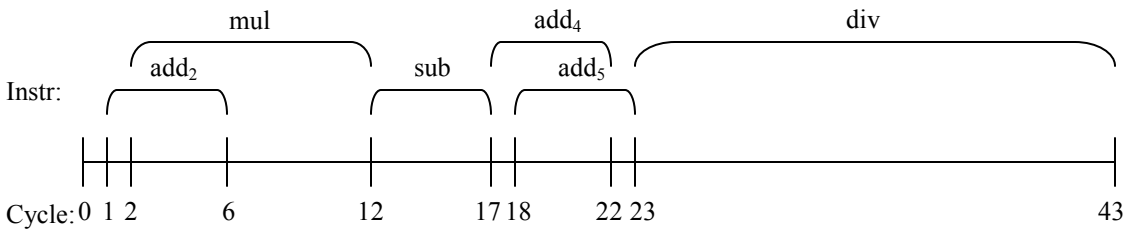
Completion:

An instruction is complete when it has written its destination register.

Result: Executing these instructions one after the other takes:

10 + 5 + 5 + 5 + 5 + 20 = 50 cycles

Observation: That's too strict! Instructions 1 and 2, for instance, could both issue immediately.
A more efficient implementation might have achieved the following:



Second Attempt: (still too conservative)

An instruction may issue if and only if all of the following are true:

1) All previous instructions that write its source registers have completed
2) All previous instructions that write its destination register have completed
3) All previous instructions have read their source registers

(1) eliminates true dependences
(2) eliminates output dependences
(3) eliminates anti dependences

Result:

Assumptions: Single issue per cycle, priority given to earliest fetched instruction
Note: all actions (issue, completion, rule compliance) take place at the beginning of the indicated cycle

	Complies with Rule				
Instr	**1**	**2**	**3**	**Issues**	**Completes**
1	1	1	1	1	11
2	1	1	2	2	7
3	11	1	3	11	16
4	1	11	12	12	17
5	1	7	13	13	18
6	18	1	14	18	38

Third Attempt:

An instruction may issue if and only if all of the following are true:

1) All previous instructions that write its source registers have completed
2) All previous instructions that write its destination register have completed
3) All previous instructions that read its destination register have issued

(1) eliminates true dependences
(2) eliminates output dependences
(3) eliminates anti dependences

Result:

Assumptions: Single issue per cycle, priority given to earliest fetched instruction
Note: all actions (issue, completion, rule compliance) take place at the beginning of the indicated cycle

	Complies with Rule				
Instr	**1**	**2**	**3**	**Issues**	**Completes**
1	1	1	1	1	11
2	1	1	1	2	7
3	11	1	1	11	16
4	1	11	12	12	17
5	1	7	12	13	18
6	18	1	1	18	38