

Dependencies

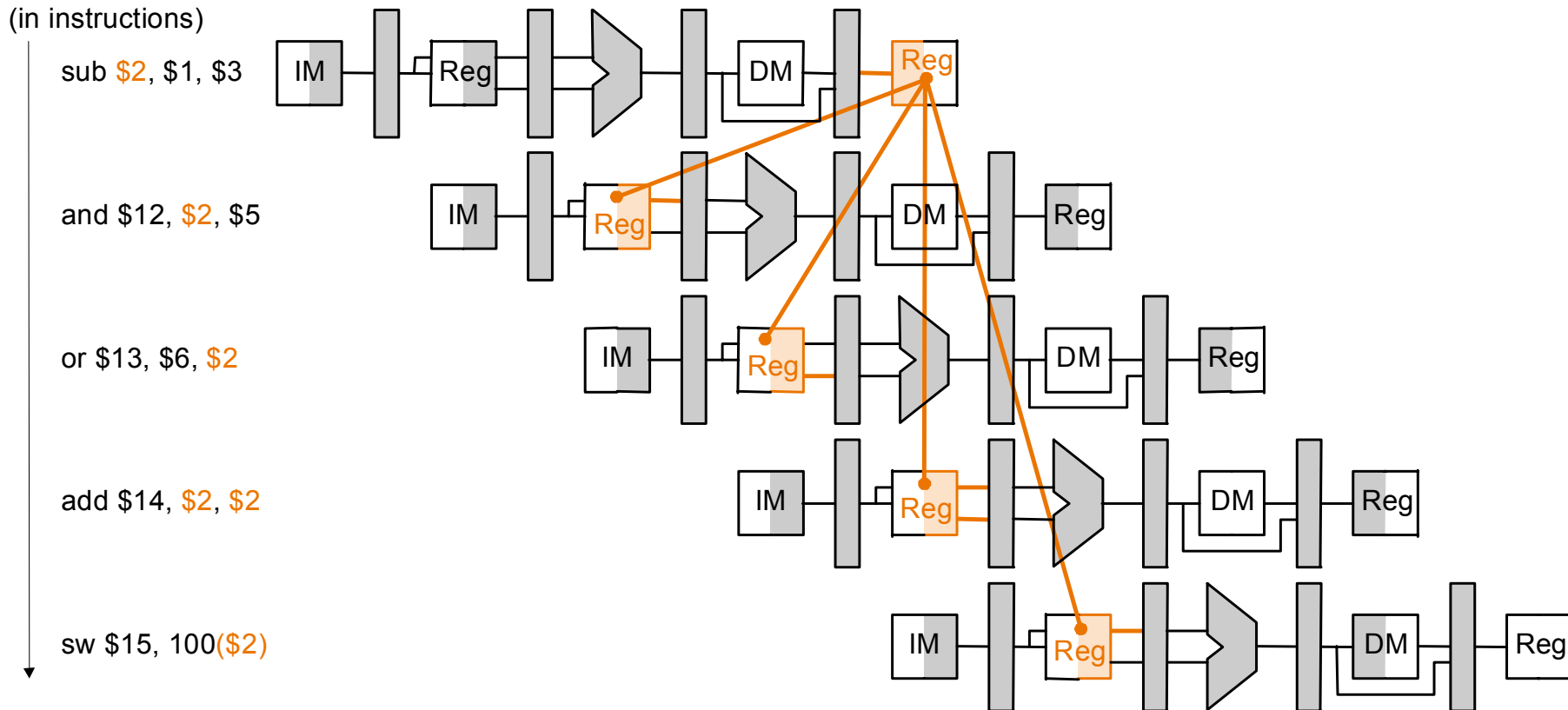
- ❑ **Problem with starting next instruction before first is finished**
 - dependencies that “go backward in time” are data hazards

Examples of Dependencies

Time (in clock cycles) →

Value of register \$2:	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
	10	10	10	10	10/-20	-20	-20	-20	-20

Program execution order (in instructions)



Software Solution

- ❑ Have compiler guarantee no hazards
- ❑ Where do we insert the “nops” ?

```
sub    $2, $1, $3
and    $12, $2, $5
or     $13, $6, $2
add    $14, $2, $2
sw     $15, 100($2)
```

- ❑ Problem: this really slows us down!

Forwarding

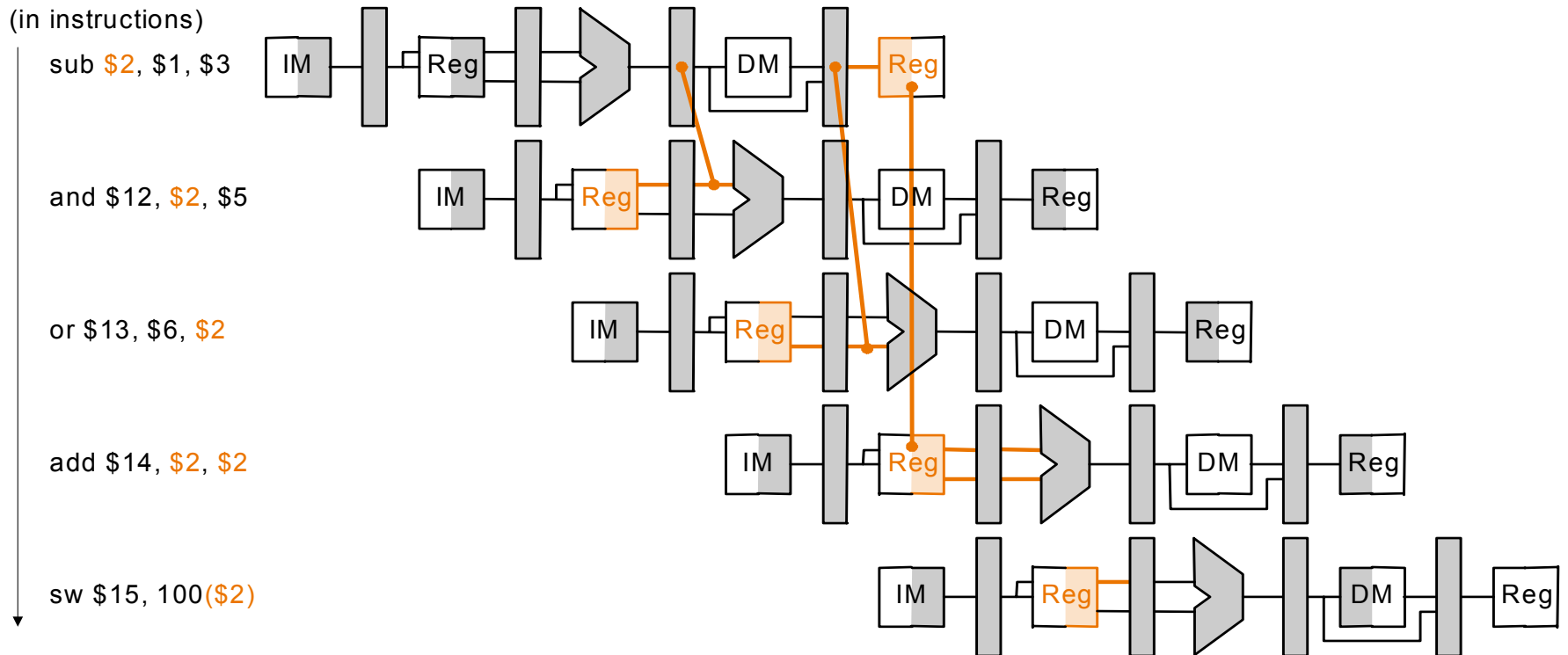
- **Use temporary results, don't wait for them to be written**
 - register file forwarding to handle read/write to same register
 - ALU forwarding

Examples of Forwarding

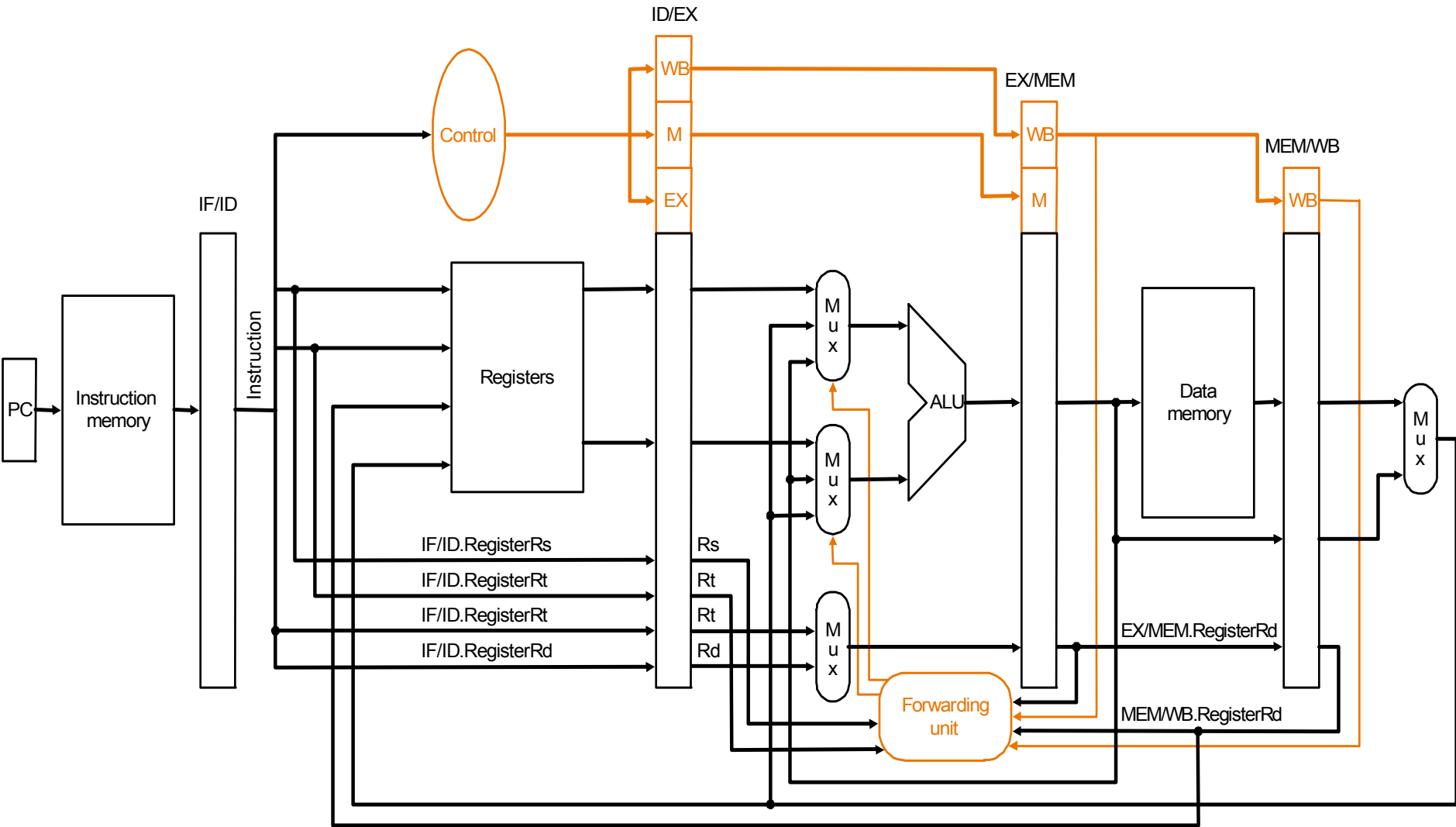
Time (in clock cycles) →

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register \$2 :	10	10	10	10	10/-20	-20	-20	-20	-20
Value of EX/MEM :	X	X	X	-20	X	X	X	X	X
Value of MEM/WB :	X	X	X	X	-20	X	X	X	X

Program execution order (in instructions)



Forwarding



Can't always forward

❑ Load word can still cause a hazard:

- an instruction tries to read a register following a load instruction that writes to the same register.

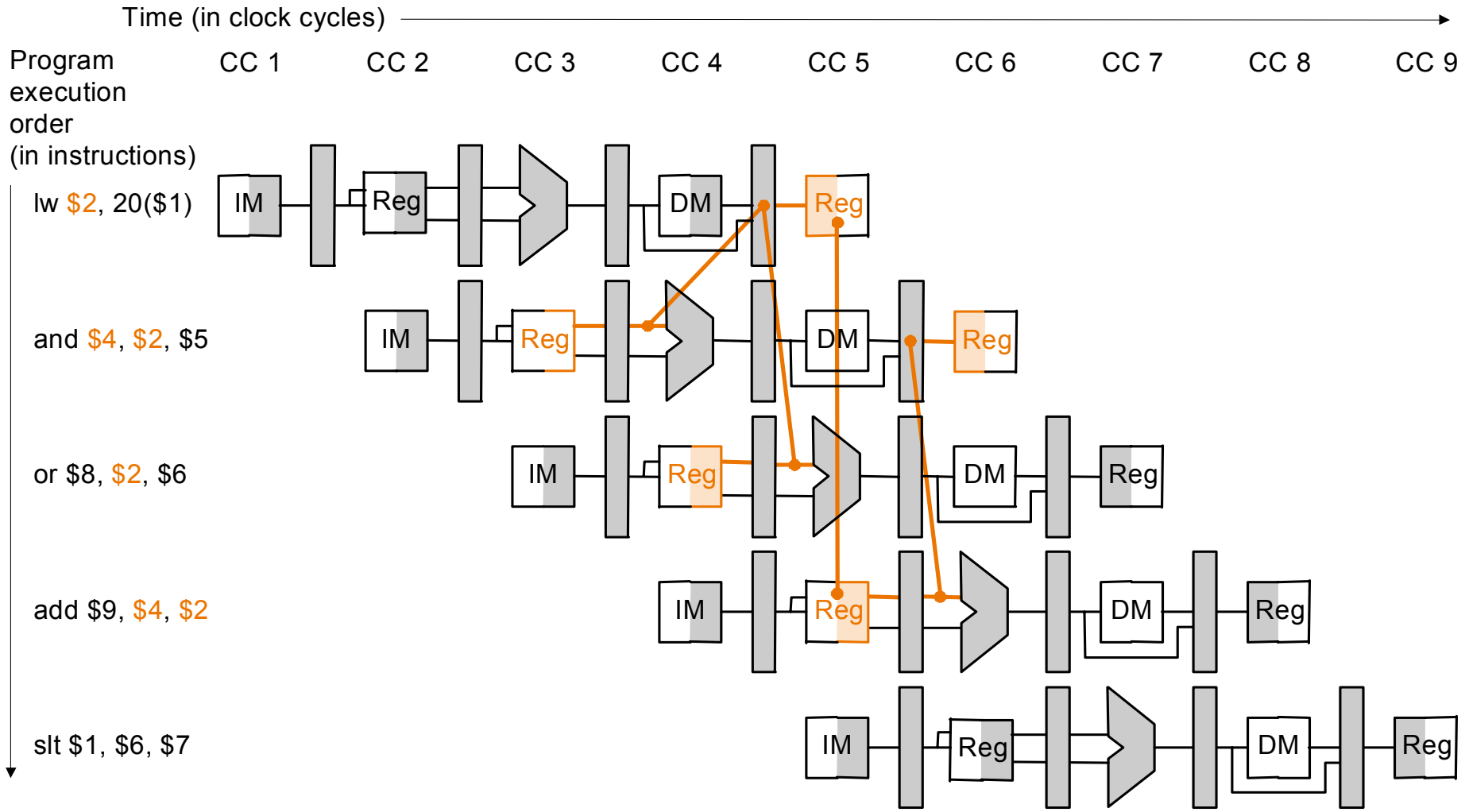
❑ Thus, we need a hazard detection unit to “stall” the load instruction

- We can stall the pipeline by keeping an instruction in the same stage

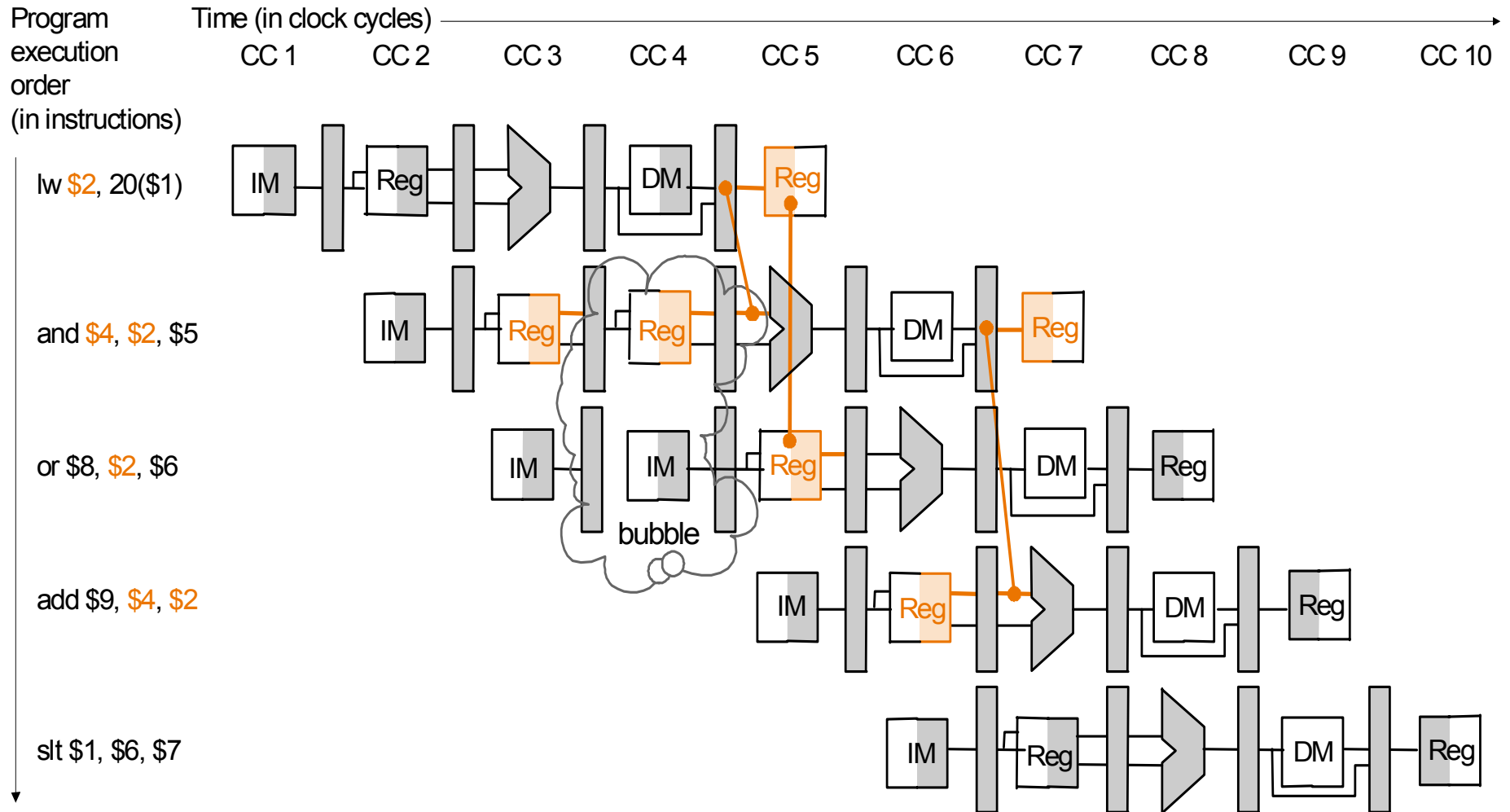
❑ Or, we use the software solution:

- insert nop between lw and the use of its result

Example of Stalls

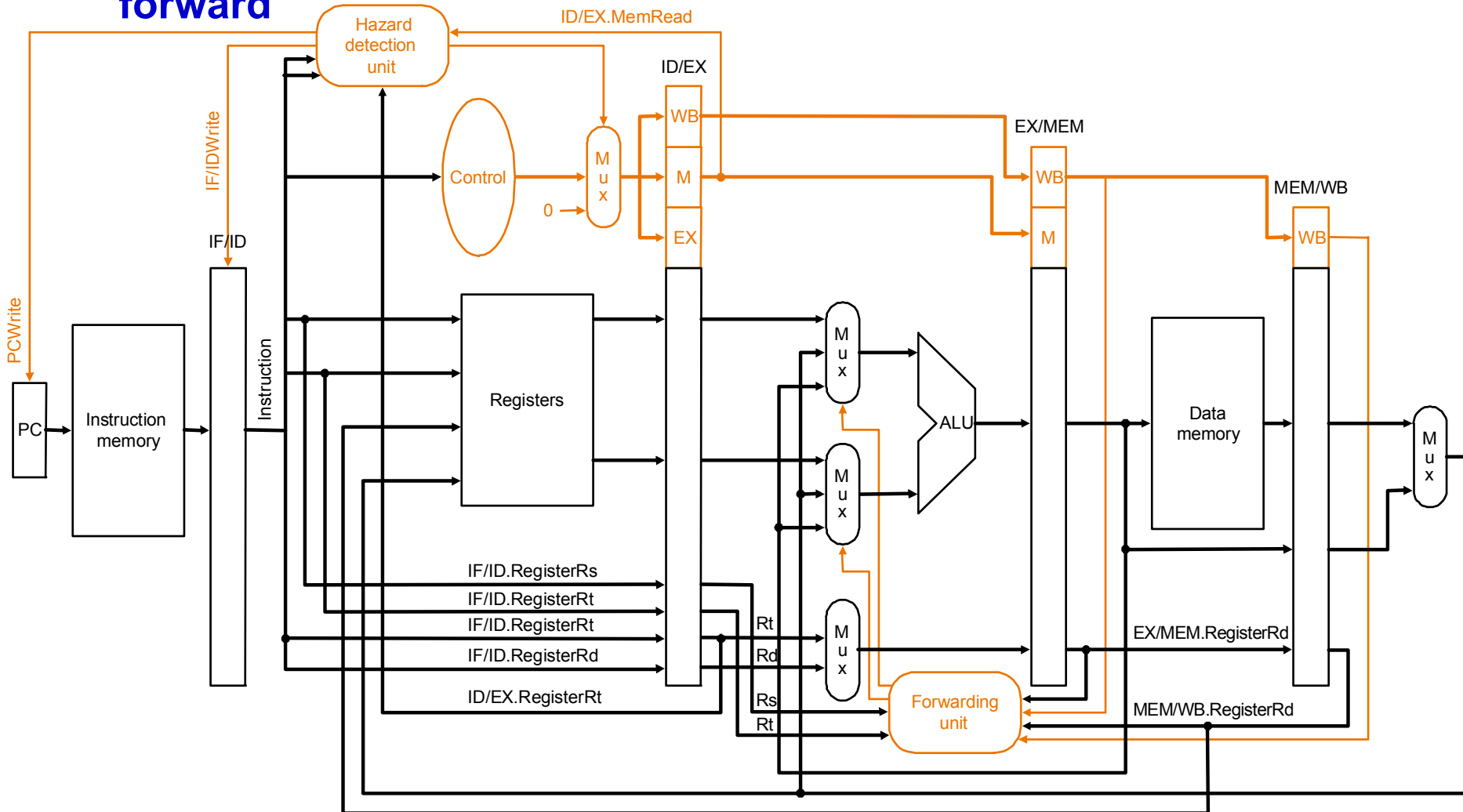


Stalling



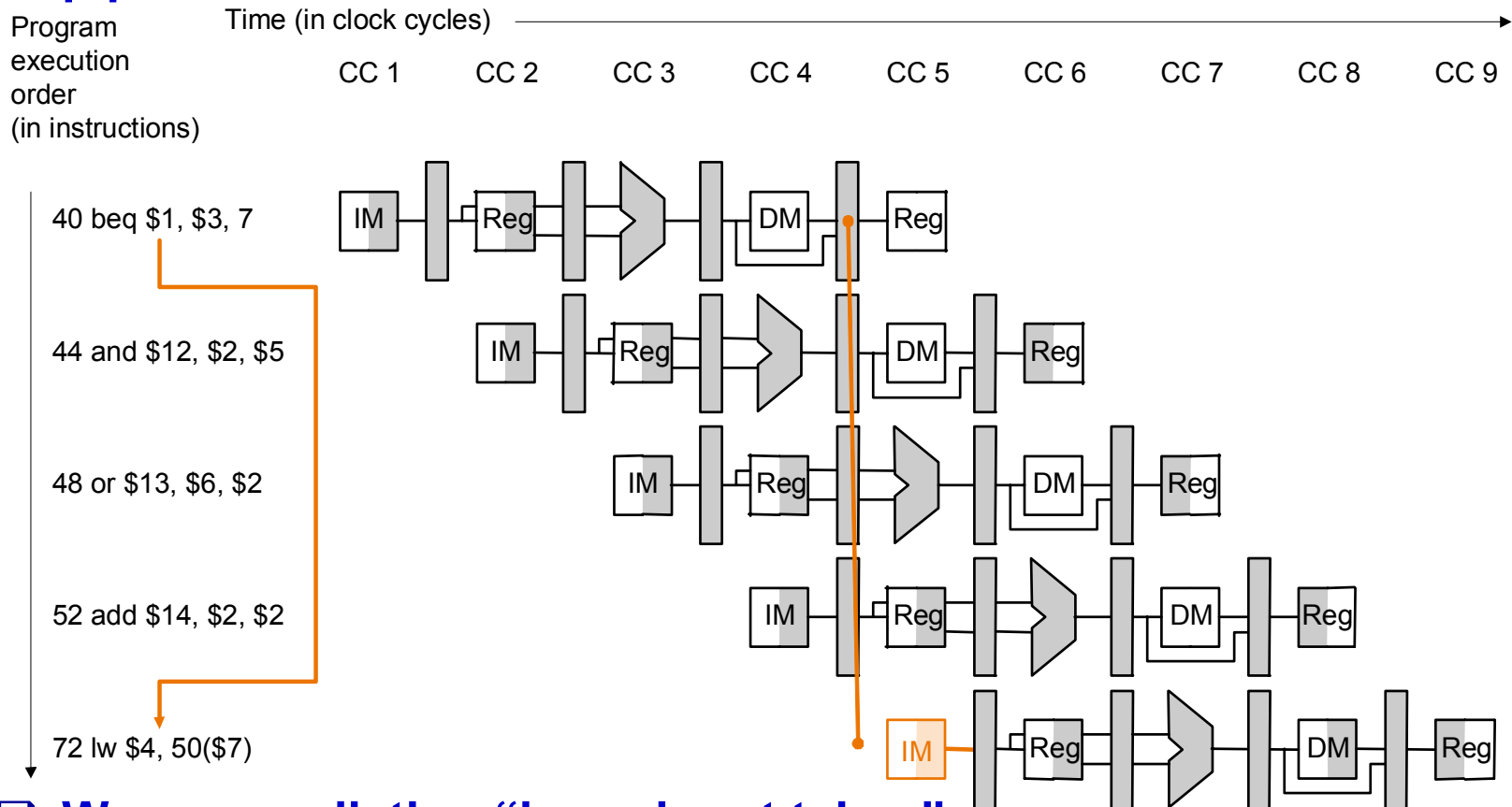
Hazard Detection Unit

- ❑ Stall by letting an instruction that won't write anything go forward



Branch Hazards

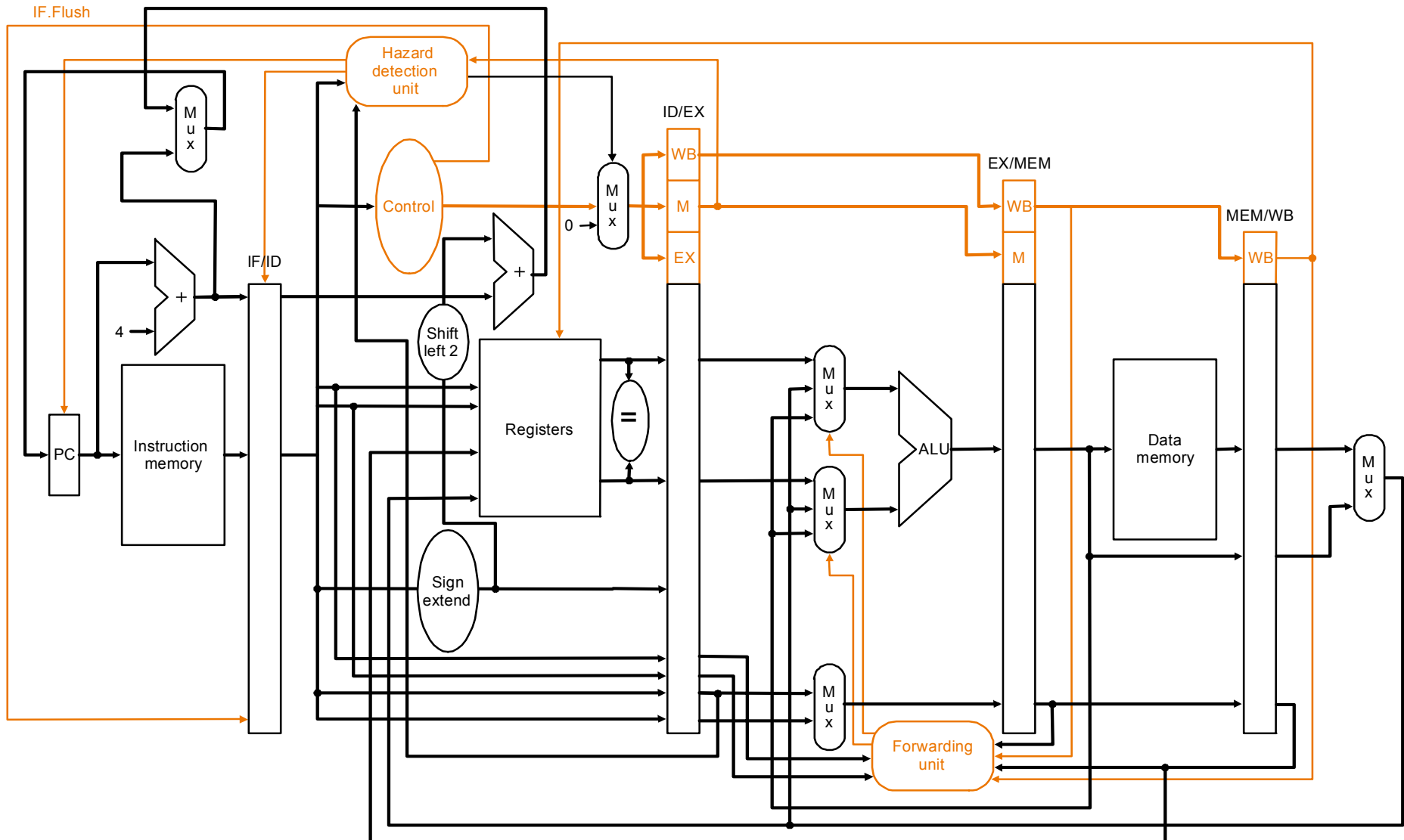
- ❑ When we decide to branch, other instructions are in the pipeline!



- ❑ We are predicting “branch not taken”

– need to add hardware for flushing instructions if we are wrong

Flushing Instructions



Improving Performance

- Try and avoid stalls! E.g., reorder these instructions:

```
lw $t0, 0($t1)
lw $t2, 4($t1)
add $t3, $t2, $t1
add $t4, $t0, $t1
```

```
lw $t0, 0($t1)
lw $t2, 4($t1)
add $t4, $t0, $t1
add $t3, $t2, $t1
```

- Add a “branch delay slot”
 - the next instruction after a branch is always executed
 - rely on compiler to “fill” the slot with something useful
- Superscalar: start more than one instruction in the same cycle

Reducing the Impact of Branch Hazards

❑ Branch delay slot filling

- Insert an instruction after the BR that will always be executed

❑ Static branch prediction

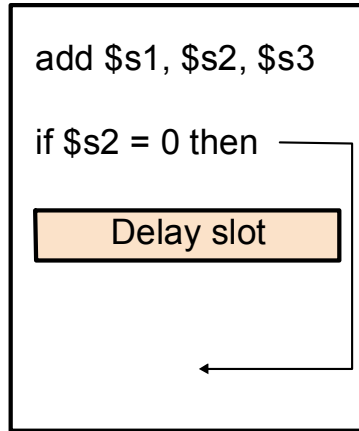
- Compiler directs the hardware to predict a BR taken or not

❑ Dynamic branch prediction

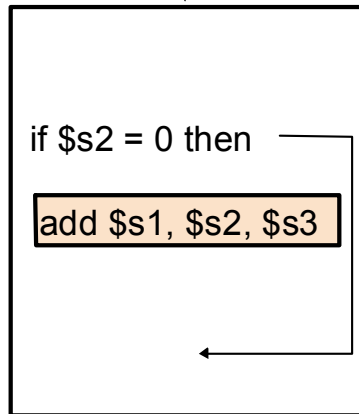
- Determine (predict) at run-time whether a BR will be taken or not

Delay slot filling

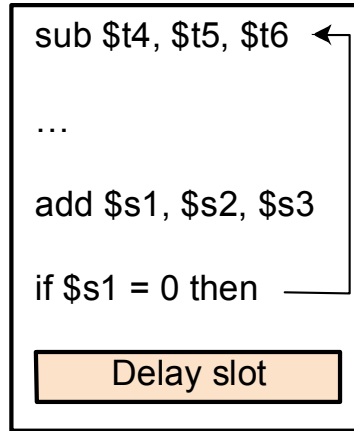
a. From before



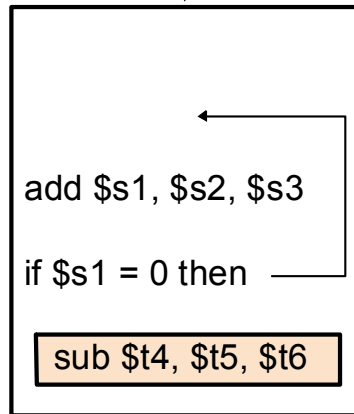
Becomes



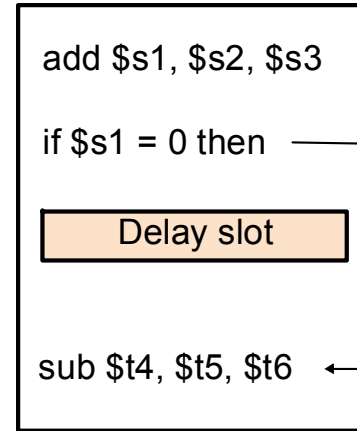
b. From target



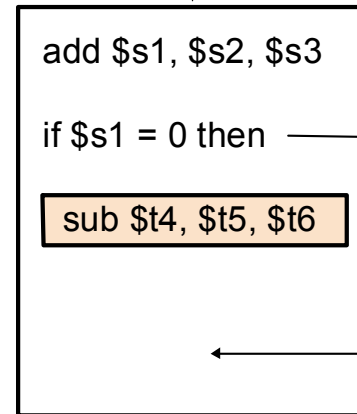
Becomes



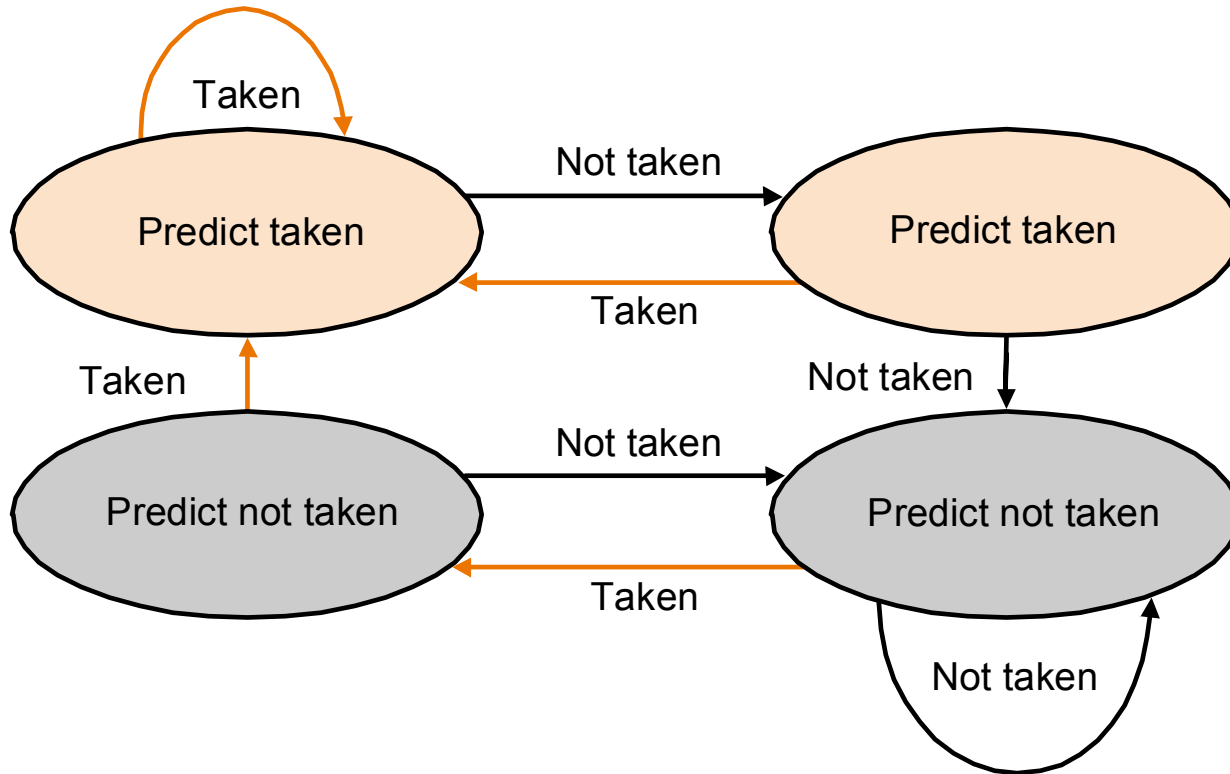
c. From fall through



Becomes



Dynamic branch prediction



Dynamic Scheduling

- ❑ **The hardware performs the “scheduling”**
 - hardware tries to find instructions to execute
 - out of order execution is possible
 - speculative execution and dynamic branch prediction
- ❑ **All modern processors are very complicated**
 - DEC Alpha 21264: 9 stage pipeline, 6 instruction issue
 - PowerPC and Pentium: branch history table
 - Compiler technology important
- ❑ **This class has given you the background you need to learn more**