UNT
UNIVERSITY OF NORTH TEXAS

CSCE 5610
Computer System Architecture

Instruction Level Parallelism

Pipeline hazards

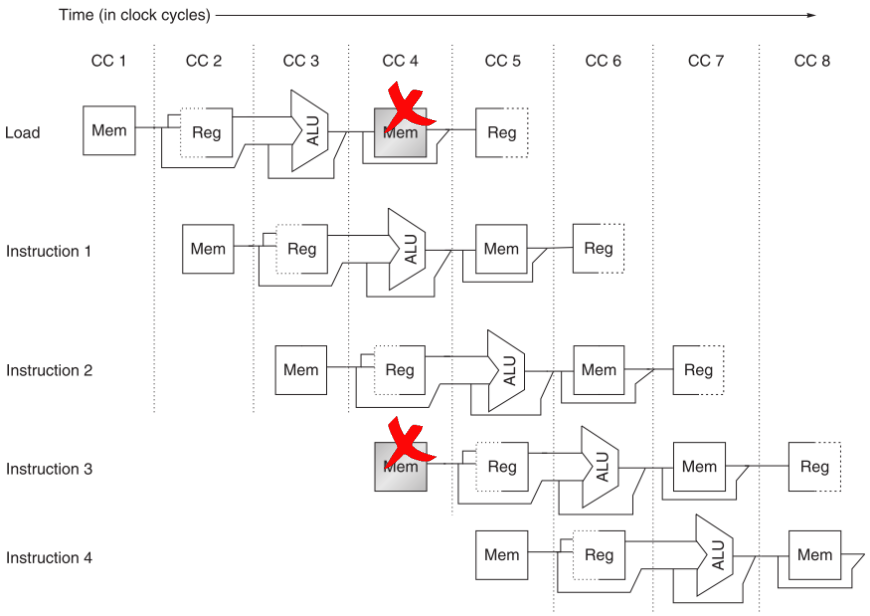
- Hazards: Situations in pipelined Datapath, in which the next instruction cannot execute in the following clock cycle.

- Three types of hazards:
  - Structural Hazards
  - Data Hazards
  - Control Hazards

Structural Hazards

- Structural Hazards: The hardware cannot support the combination of instructions that are required to be executed in the same clock cycle
- MIPS instruction set is designed to be pipelined making it easy to avoid structural hazards.

Suppose we had a single memory instead of two data and instruction memories

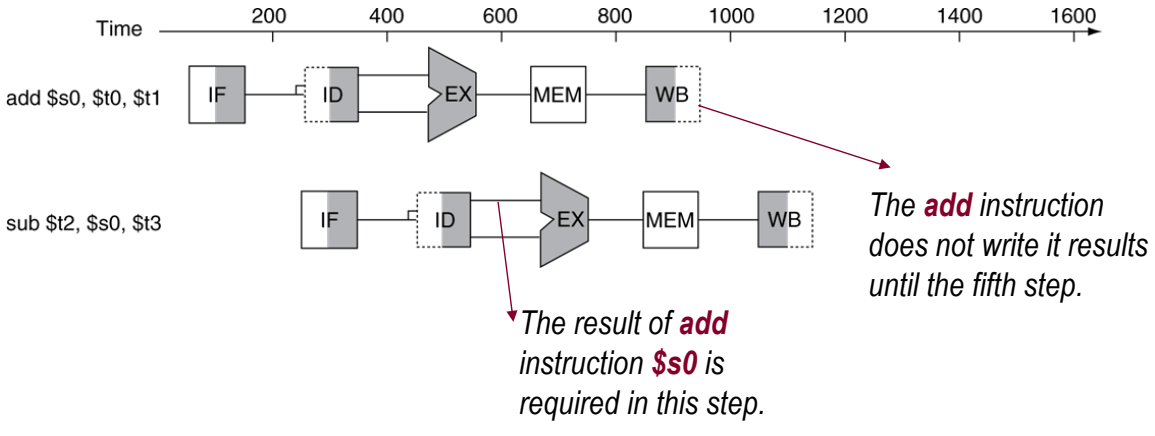


Data Hazards

- Data Hazards occur when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.

- Example 1:

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```

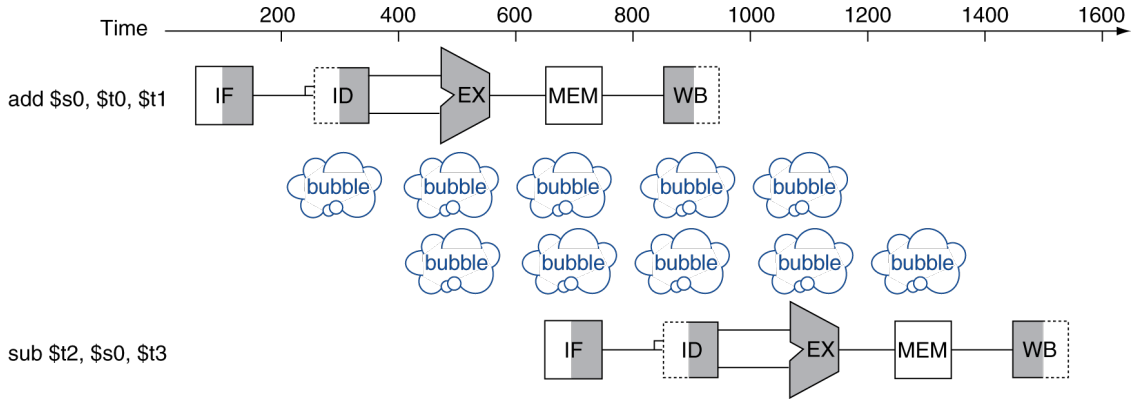


Data Hazards

- Example 1:

```
add $s0, $t1, $t2
sub $t2, $s0, $t3
```

- Stalling: a *simple* but *slow* solution



Pipeline Performance with stalls

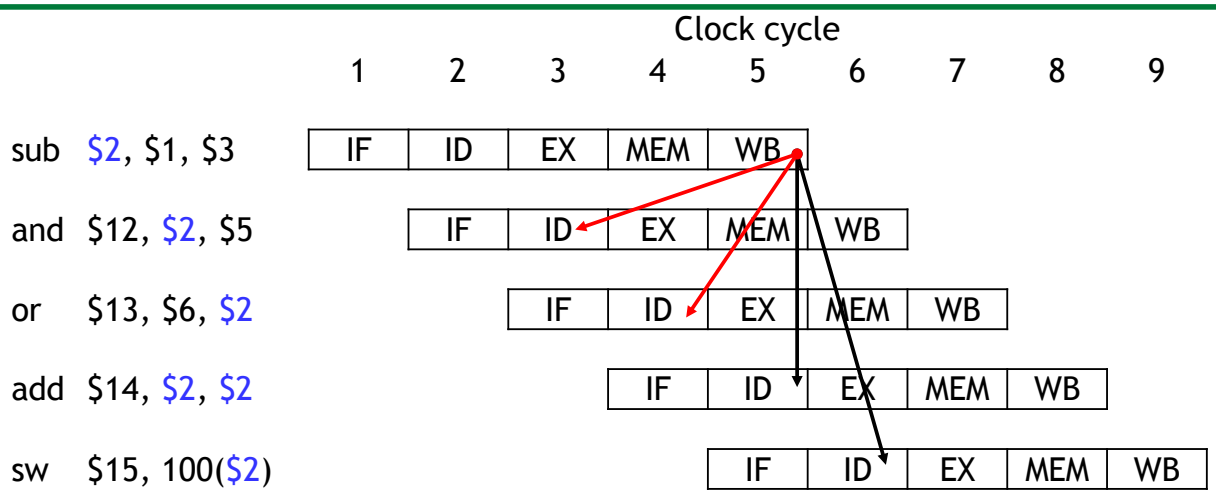
- Assume the stages are balanced, speedup of pipelining?

Pipeline Performance with stalls

- If there are no stalls, the speedup is equal to the number of pipeline stages. However, if we add two stalls after each instruction in a 5-stage pipelined MIPS processor:

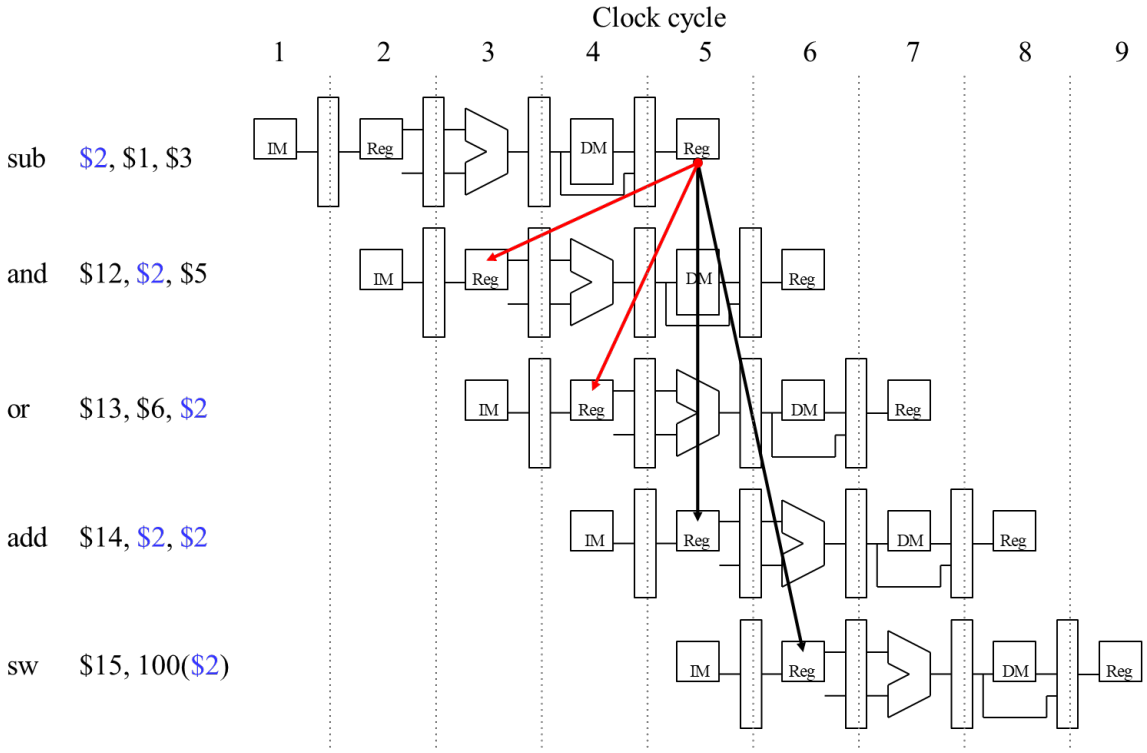
$$\text{Maximum Speedup from pipelining} = \frac{1}{3} \times 5 = 1.67$$

Data hazards – Dependency Arrow



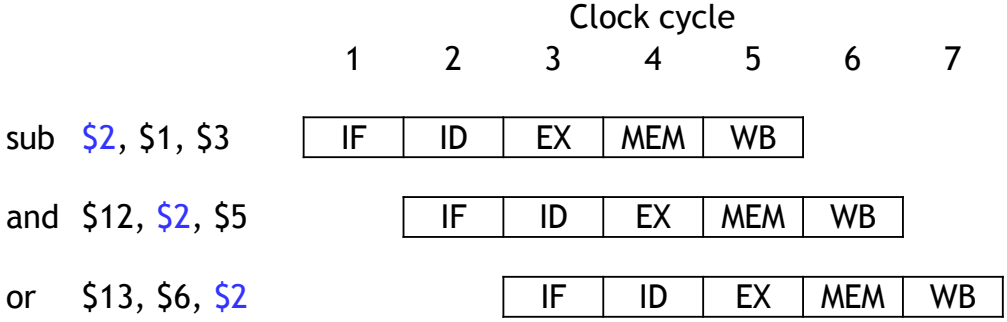
- Arrows indicate the flow of data between instructions.
  - The tails of the arrows show when register $2 is written.
  - The heads of the arrows show when $2 is read.
- Any arrow that points backwards in time represents a data hazard in our basic pipelined datapath. Here, hazards exist between instructions 1 & 2 and 1 & 3.

A Fancier Pipeline Diagram



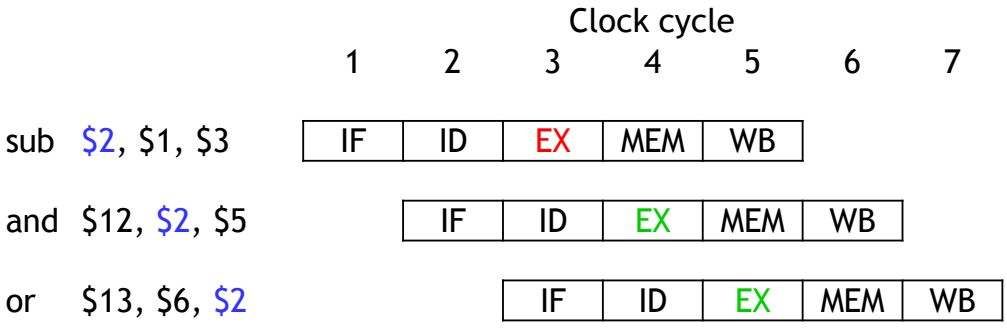
A more Detailed Look at the Pipeline

- We have to eliminate the hazards, so the AND and OR instructions in our example will use the correct value for register $2.
- When is the data is actually produced and consumed?
- What can we do?



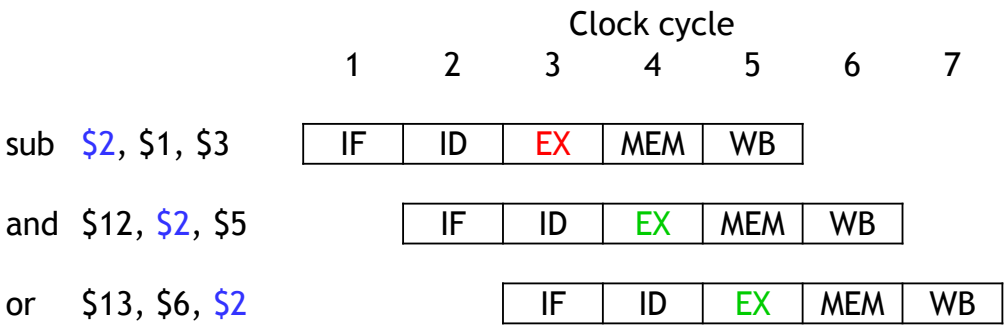
A more Detailed Look at the Pipeline

- We have to eliminate the hazards, so the AND and OR instructions in our example will use the correct value for register $2.
- Let's look at when the data is actually produced and consumed.
  - The SUB instruction produces its result in its EX stage, during cycle 3 in the diagram below.
  - The AND and OR need the new value of $2 in their EX stages, during clock cycles 4-5 here.



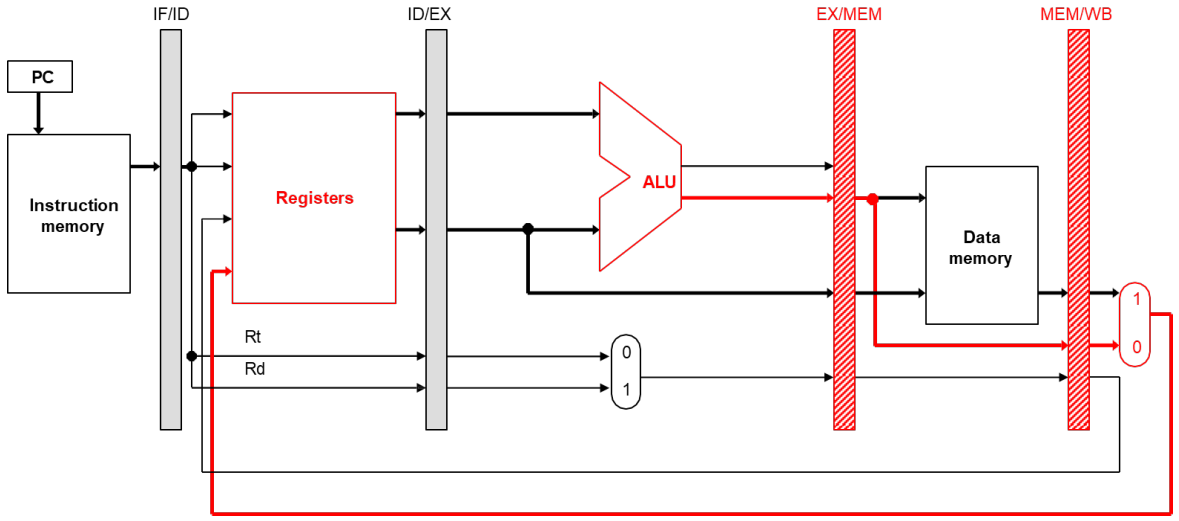
A more Detailed Look at the Pipeline

- The actual result $1 -$3 is computed in clock cycle 3, *before* it's needed in cycles 4 and 5.
- If we could somehow bypass the writeback and register read stages when needed, then we can eliminate these data hazards.
  - Now, we'll focus on hazards involving arithmetic instructions.
  - Next time, we'll examine the lw instruction.
- Essentially, we need to pass the ALU output from SUB directly to the AND and OR instructions, without going through the register file.



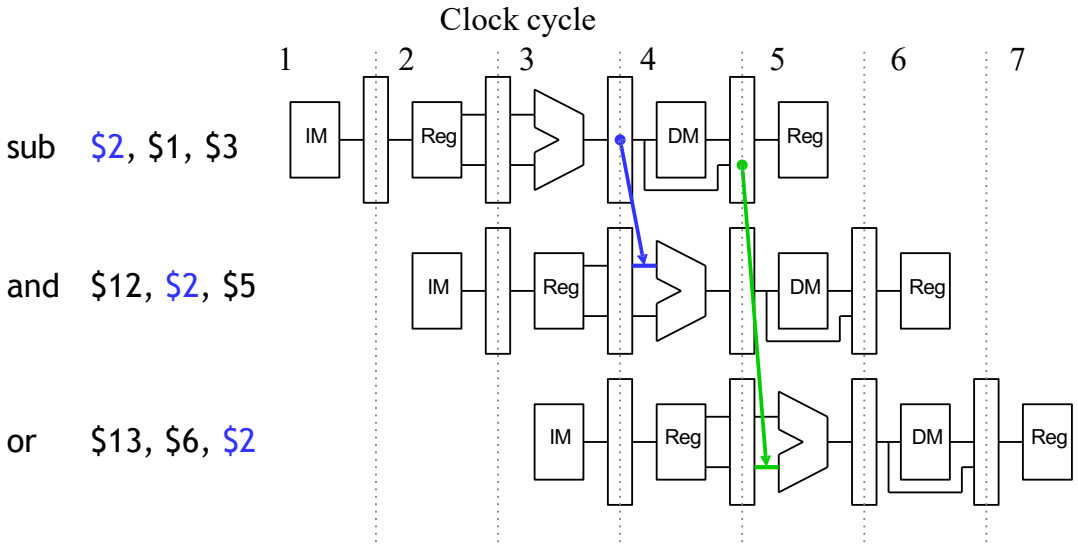
Where to find the ALU result

- The ALU result generated in the EX stage is normally passed through the pipeline registers to the MEM and WB stages, before it is finally written to the register file.
- This is an abridged diagram of our pipelined datapath.



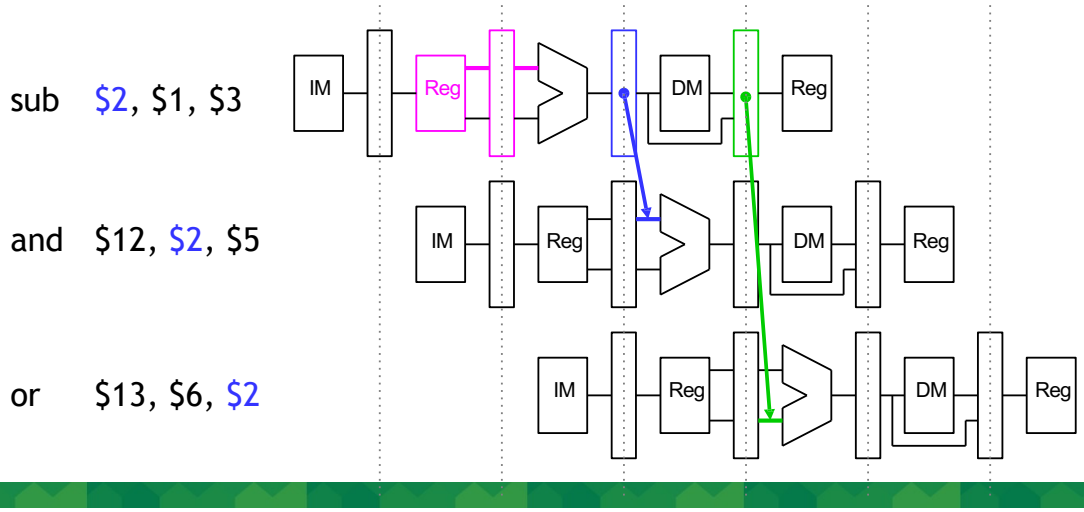
Forwarding/Bypassing

- Since the pipeline registers already contain the ALU result, we could just forward that value to subsequent instructions, to prevent data hazards.
  - In clock cycle 4, the AND instruction can get the value $1 -$3 from the EX/MEM pipeline register used by sub.
  - Then in cycle 5, the OR can get that same result from the MEM/WB pipeline register being used by SUB.

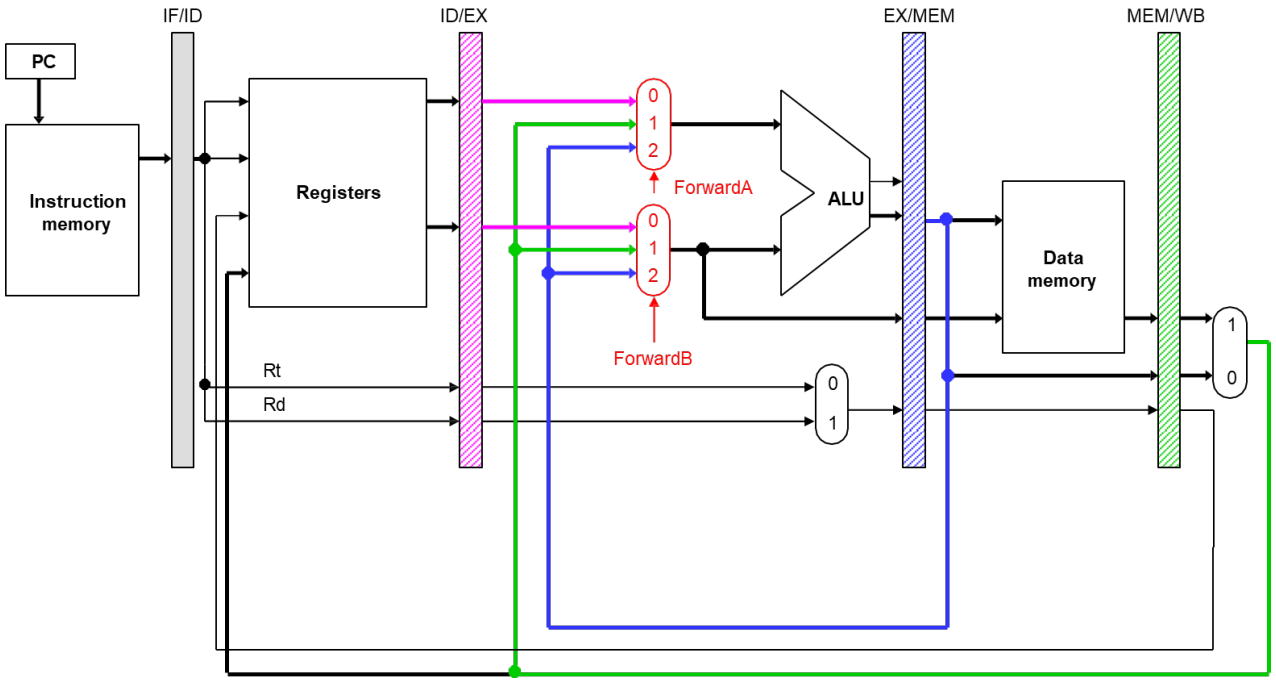


Outline of Forwarding Hardware

- A forwarding unit selects the correct ALU inputs for the EX stage.
  - If there is no hazard, the ALU's operands will come from the register file, just like before.
  - If there is a hazard, the operands will come from either the EX/MEM or MEM/WB pipeline registers instead.
- The ALU sources will be selected by two new multiplexers, with control signals named ForwardA and ForwardB.



Simplified Datapath with Forwarding Muxes

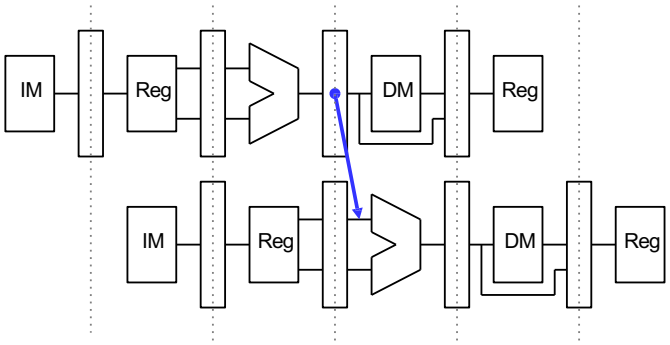


Detecting EX/MEM data hazards

- So how can the hardware determine if a hazard exists?

sub $2, $1, $3

and $12, $2, $5

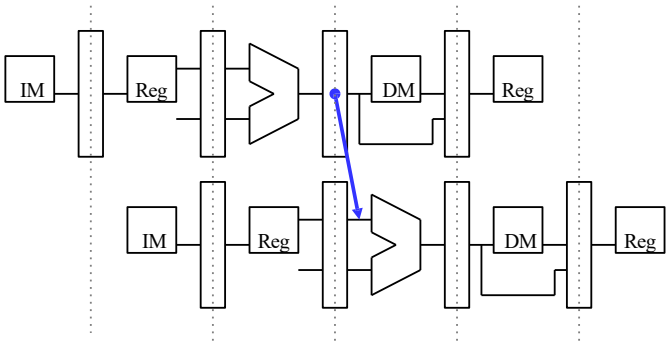


Detecting EX/MEM data hazards

- So how can the hardware determine if a hazard exists?
- An EX/MEM hazard occurs between the instruction currently in its EX stage and the previous instruction if:
  1. The previous instruction will write to the register file, *and*
  2. The destination is one of the ALU source registers in the EX stage.
- There is an EX/MEM hazard between the two instructions below.

sub $2, $1, $3

and $12, $2, $5



- Data in a pipeline register can be referenced using a class-like syntax. For example, ID/EX.RegisterRt refers to the rt field stored in the ID/EX pipeline.

EX/MEM data hazard equations

- The first ALU source comes from the pipeline register when necessary.

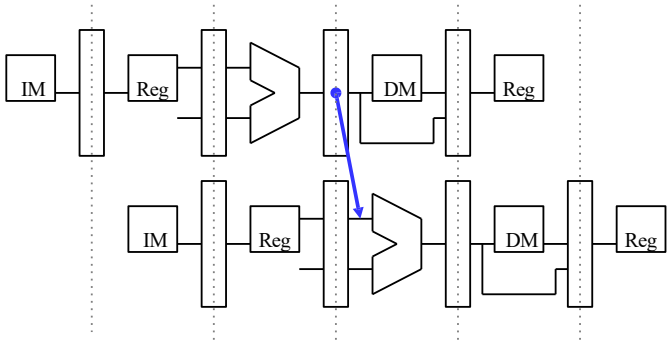
  if (EX/MEM.RegWrite = 1
  and EX/MEM.RegisterRd = ID/EX.RegisterRs)
  then ForwardA = 2

- The second ALU source is similar.

  if (EX/MEM.RegWrite = 1
  and EX/MEM.RegisterRd = ID/EX.RegisterRt)
  then ForwardB = 2

sub $2, $1, $3

and $12, $2, $5



Detecting MEM/WB data hazards

- A MEM/WB hazard may occur between an instruction in the EX stage and the instruction from *two* cycles ago.
- One new problem is if a register is updated twice in a row.

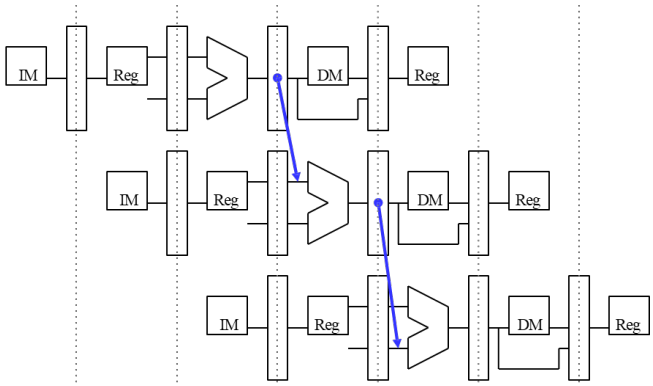
```
add  $1, $2, $3
add  $1, $1, $4
sub  $5, $5, $1
```

- Register $1 is written by *both* of the previous instructions, but only the most recent result (from the second ADD) should be forwarded.

add $1, $2, $3

add $1, $1, $4

sub $5, $5, $1



MEM/WB hazard equations

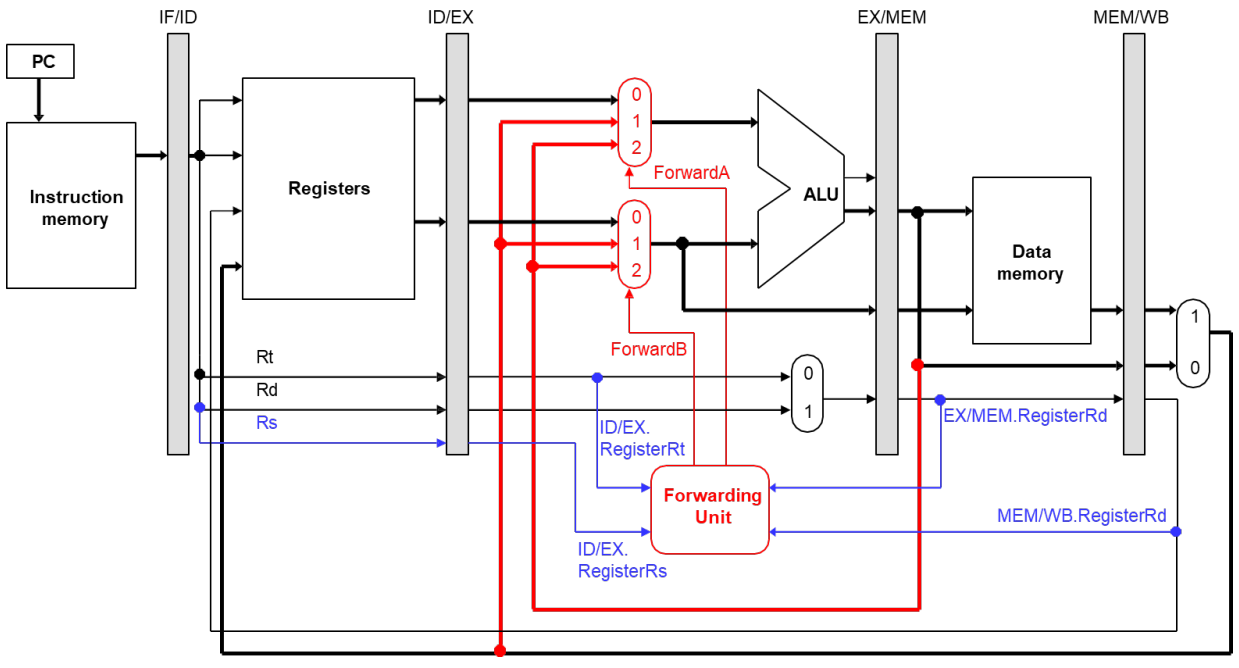
- Here is an equation for detecting and handling MEM/WB hazards for the first ALU source.

  if (MEM/WB.RegWrite = 1
  and MEM/WB.RegisterRd = ID/EX.RegisterRs
  and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs or EX/MEM.RegWrite = 0)
  then ForwardA = 1

- The second ALU operand is handled similarly.

  if (MEM/WB.RegWrite = 1
  and MEM/WB.RegisterRd = ID/EX.RegisterRt
  and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRt or EX/MEM.RegWrite = 0)
  then ForwardB = 1

Simplified Datapath with Forwarding



The Forwarding Unit

- The forwarding unit has several control signals as inputs.

ID/EX.RegisterRs	EX/MEM.RegisterRd	MEM/WB.RegisterRd
ID/EX.RegisterRt	EX/MEM.RegWrite	MEM/WB.RegWrite

  (The two RegWrite signals are not shown in the diagram, but they come from the control unit.)
- The fowarding unit outputs are selectors for the ForwardA and ForwardB multiplexers attached to the ALU. These outputs are generated from the inputs using the equations on the previous pages.
- Some new buses route data from pipeline registers to the new muxes.

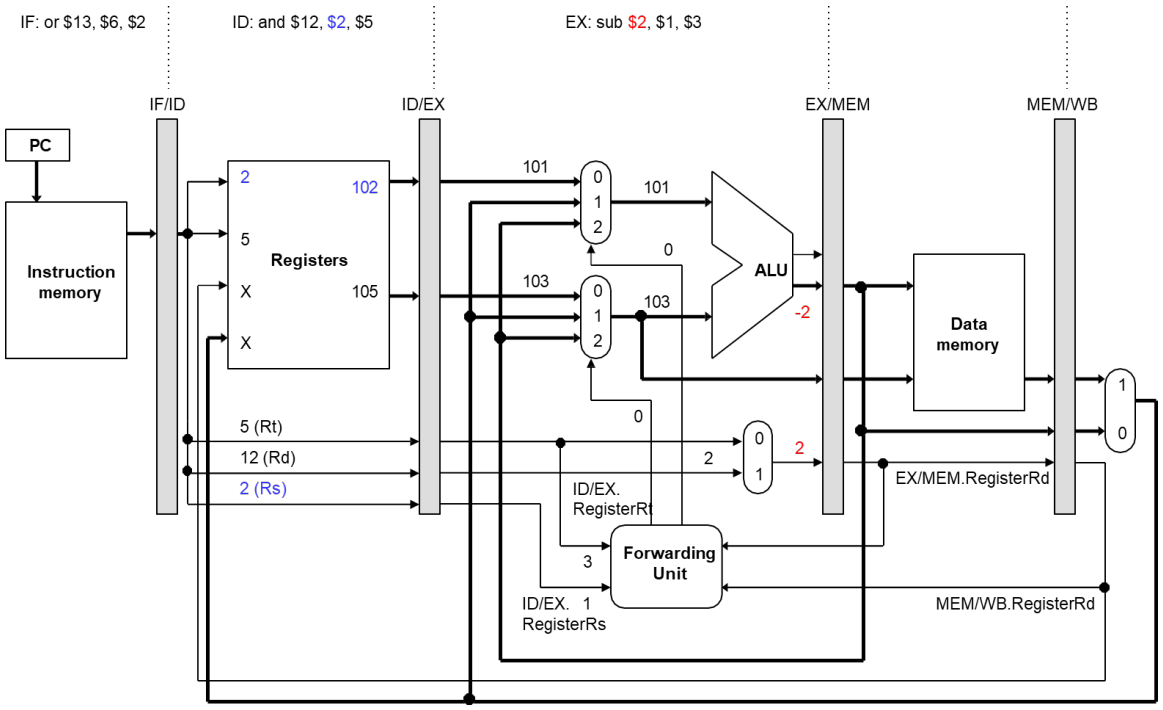
Example

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

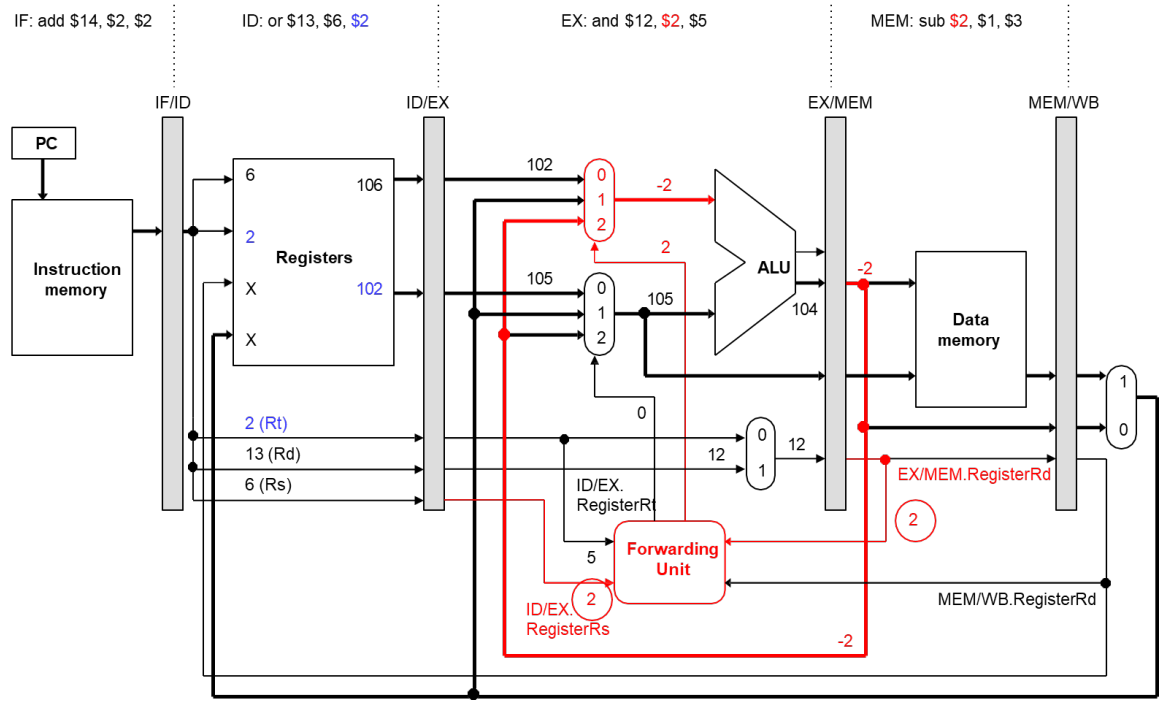
- Assume again each register initially contains its number plus 100.
  - After the first instruction, $2 should contain -2 (101 -103).
  - The other instructions should all use -2 as one of their operands.

- We'll try to keep the example short.
  - Assume no forwarding is needed except for register $2.
  - We'll skip the first two cycles, since they're the same as before.

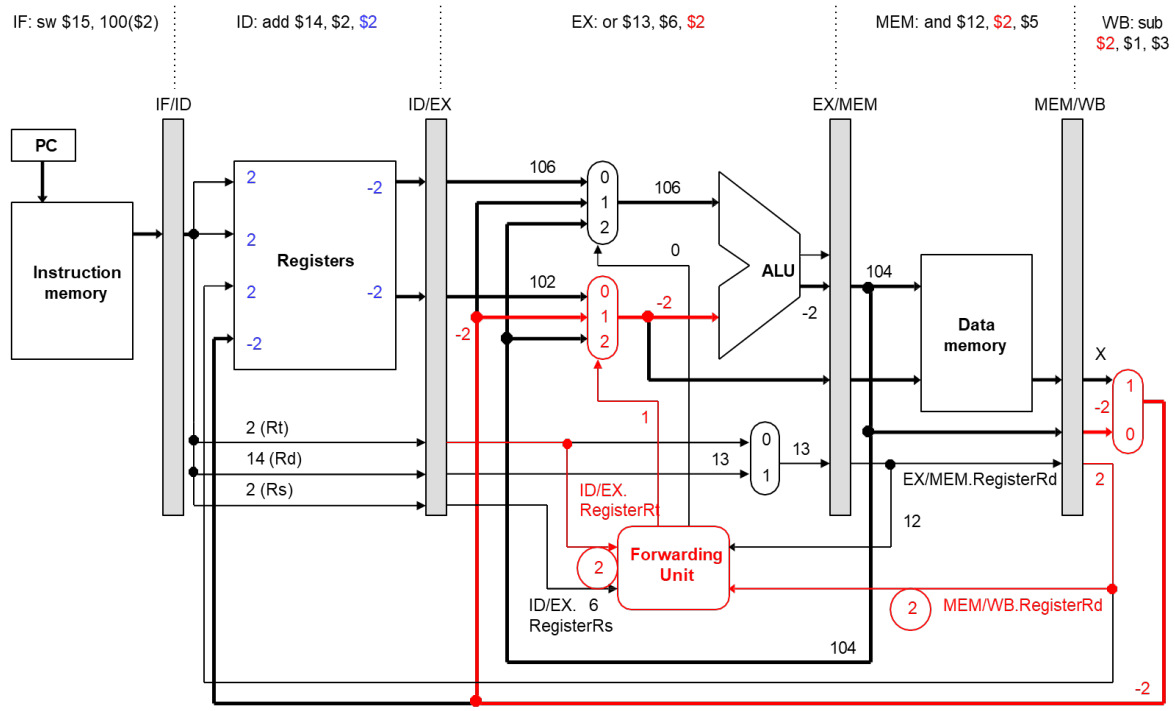
Clock cycle 3



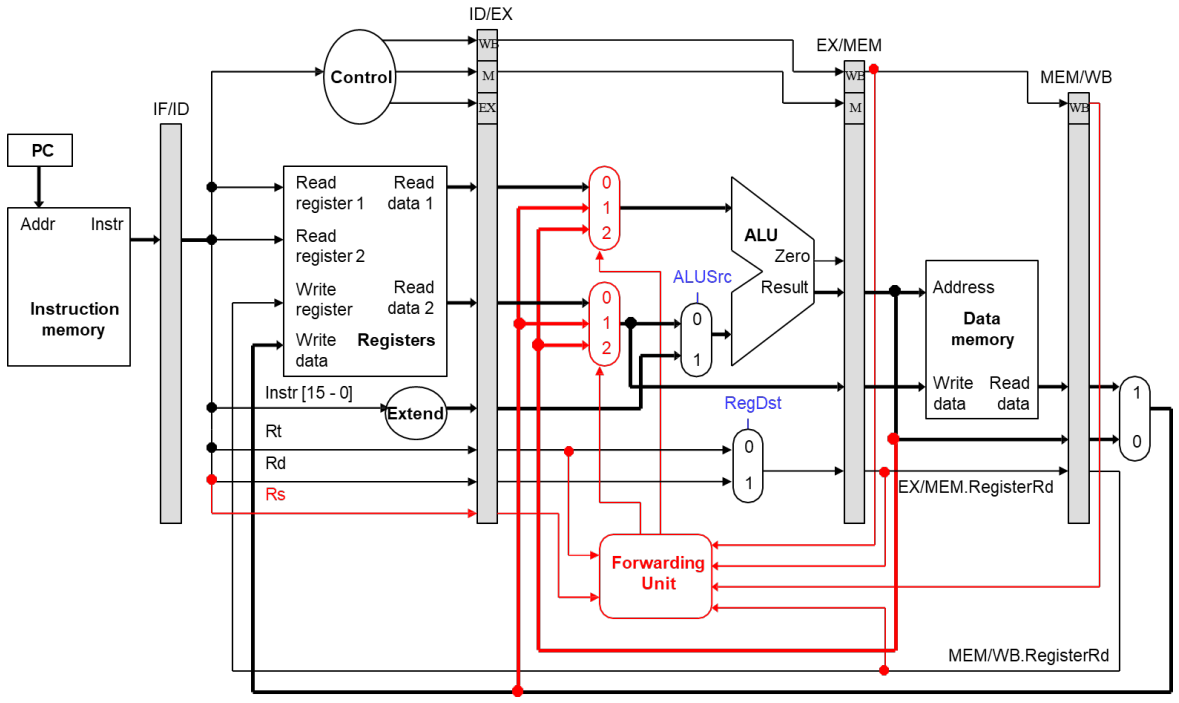
Clock cycle 4: Forwarding $2 from EX/MEM



Clock cycle 5: Forwarding $2 from MEM/WB



Complete Pipelined Datapath...so far



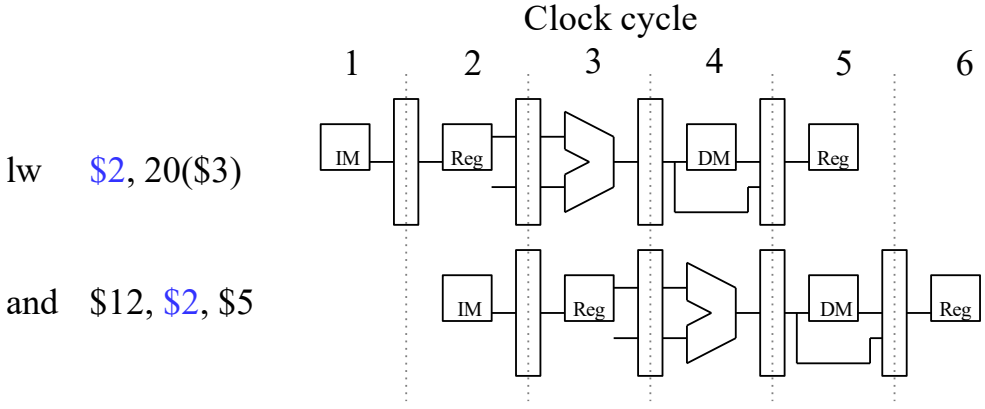
Stalls and Flushes

- So far, we have discussed data hazards that can occur in pipelined CPUs if some instructions depend upon others that are still executing.
  - Many hazards can be resolved by forwarding data from the pipeline registers, instead of waiting for the writeback stage.
  - The pipeline continues running at full speed, with one instruction beginning on every clock cycle.
- Now, we'll see some real limitations of pipelining.
  - Forwarding may not work for data hazards from load instructions.
  - Branches affect the instruction fetch for the next clock cycle.
- In both of these cases we may need to slow down, or stall, the pipeline.

What about loads?

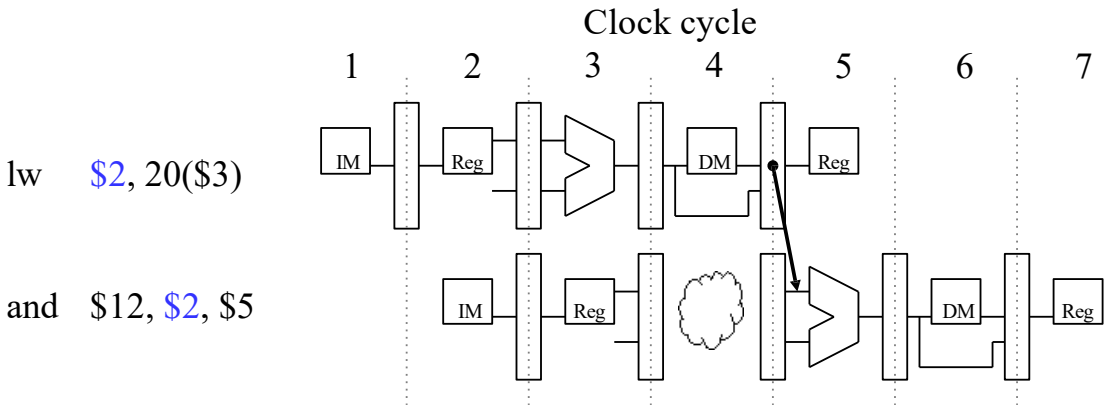
- Imagine if the first instruction in the example was LW instead of SUB.
  - How does this change the data hazard?

- Example 2:



Stalling

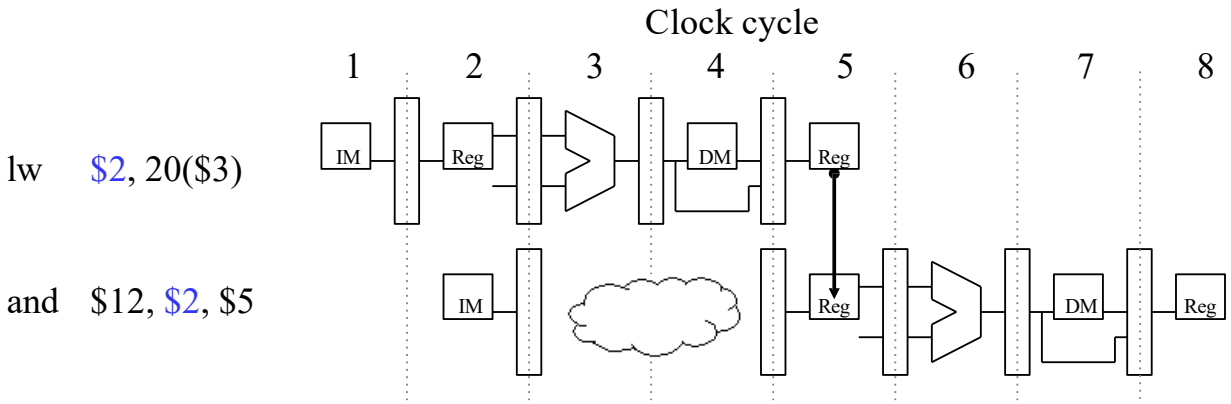
- The easiest solution is to stall the pipeline.
- We could delay the AND instruction by introducing a one-cycle delay into the pipeline, sometimes called a bubble.



- Notice that we're still using forwarding in cycle 5, to get data from the MEM/WB pipeline register to the ALU.

Stalling and Forwarding

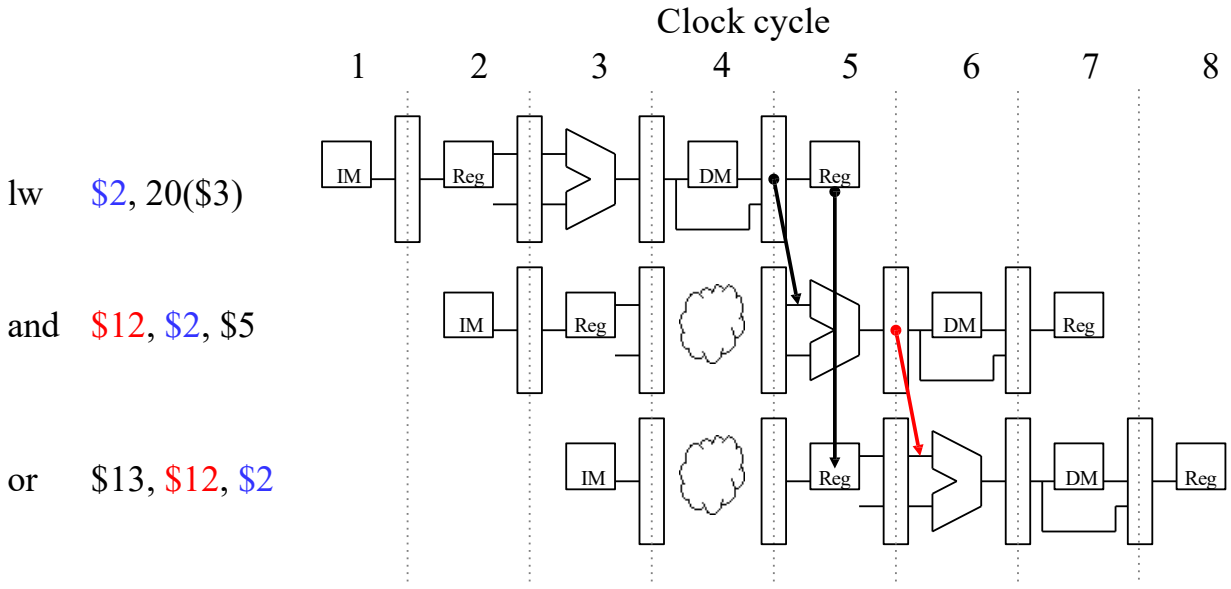
- Without forwarding, we'd have to stall for *two* cycles to wait for the LW instruction's writeback stage.



- In general, you can always stall to avoid hazards—but dependencies are very common in real code, and stalling often can reduce performance by a significant amount.

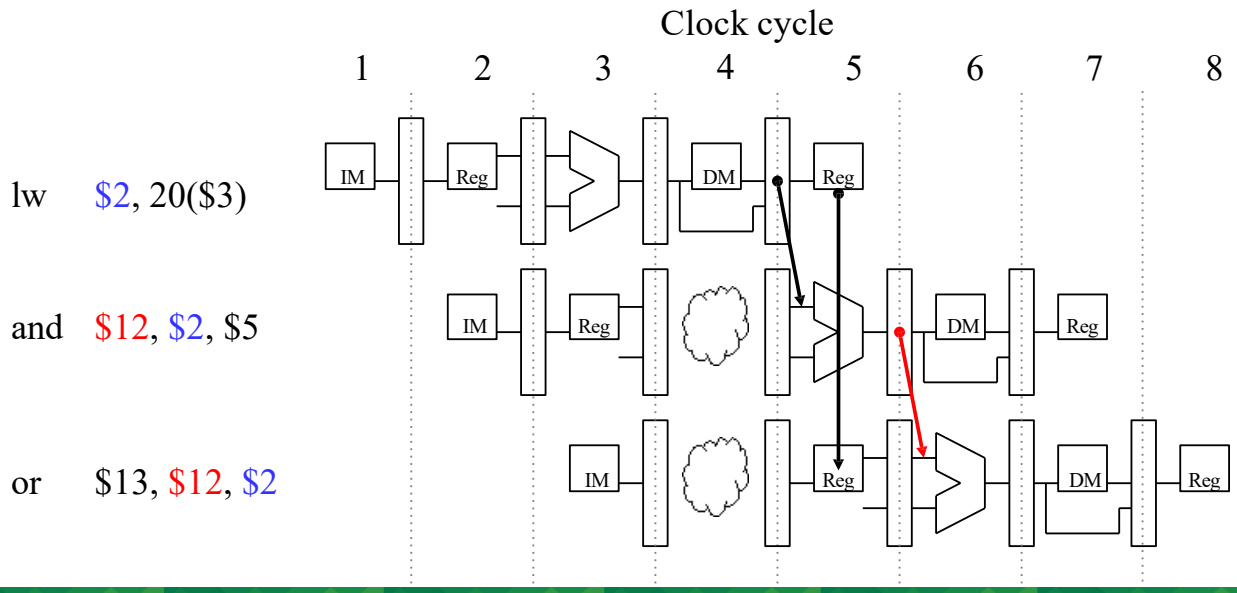
Stalling Delays the Entire Pipeline

- If we delay the second instruction, we'll have to delay the third one too.
  - Why?

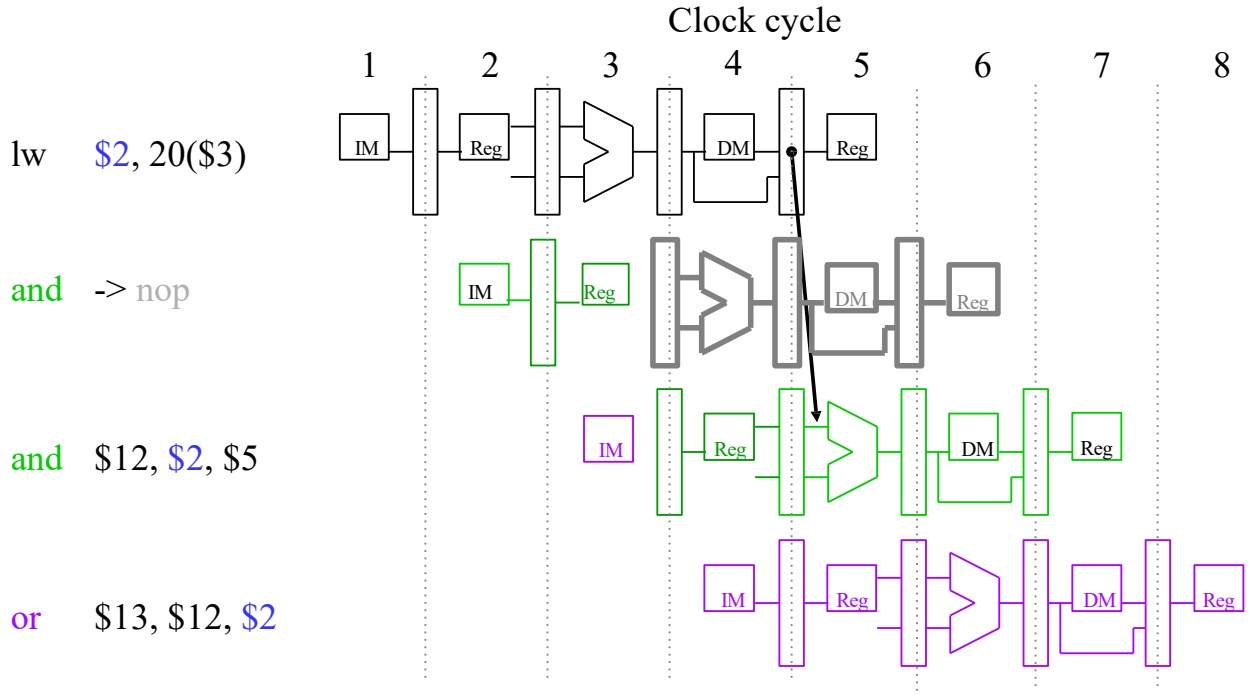


Stalling Delays the Entire Pipeline

- If we delay the second instruction, we'll have to delay the third one too.
  - This is necessary to make forwarding work between AND and OR.
  - It also prevents problems such as two instructions trying to write to the same register in the same cycle.



Stall = Nop conversion



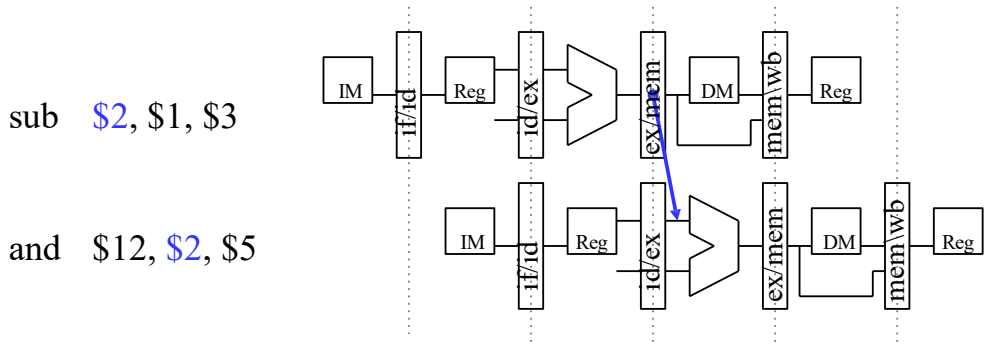
- The effect of a load stall is to insert an empty or nop instruction into the pipeline

Detecting stalls

- Detecting stall is much like detecting data hazards.

- Recall the format of hazard detection equations:

if (EX/MEM.RegWrite = 1
and EX/MEM.RegisterRd = ID/EX.RegisterRs)
then Bypass Rs from EX/MEM stage latch

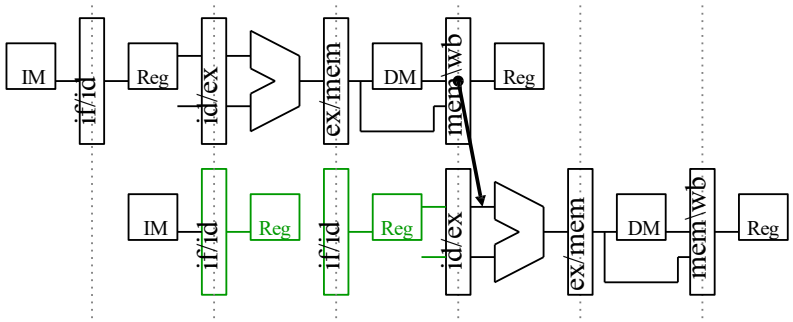


Detecting Stalls, cont.

- When should **stalls** be detected?

lw $2, 20($3)

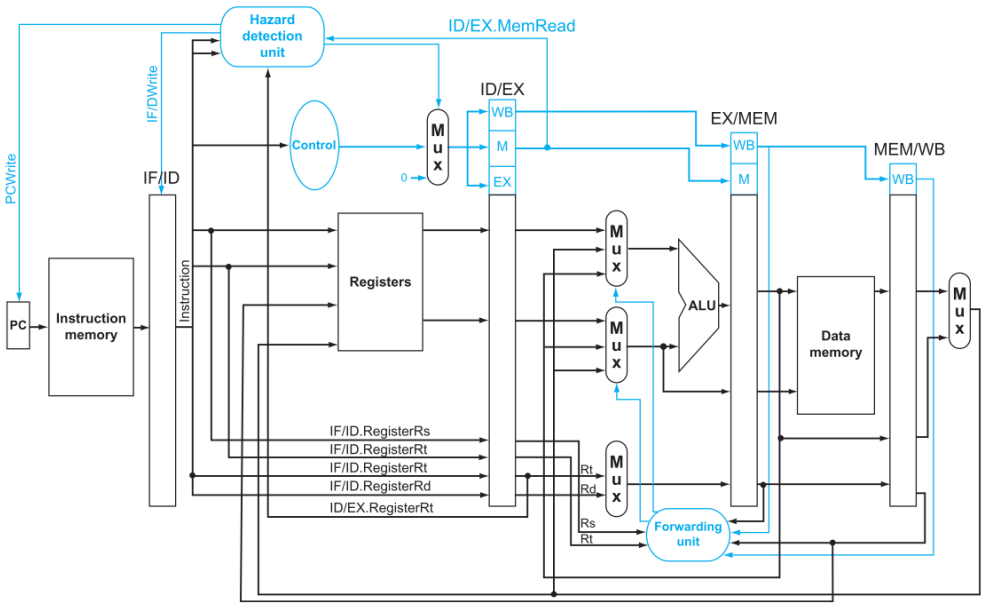
and $12, $2, $5



- What is the stall condition?

if (ID/EX.MemRead = 1
and (ID/EX.RegisterRt = IF/ID.RegisterRs or
ID/EX.RegisterRt = IF/ID.RegisterRt)
)
then stall

Hazard detection unit



- To stall the pipeline, the instructions in ID stage should be stalled ➔ We can do it by continuing to read the same PC
- To insert bubbles or **nops,** we set all the nine control signals to “0” ➔ no register or memories are written, which will create a “do nothing” instruction.

Data Hazards: Reordering

Given: Consider the following code segment in C:

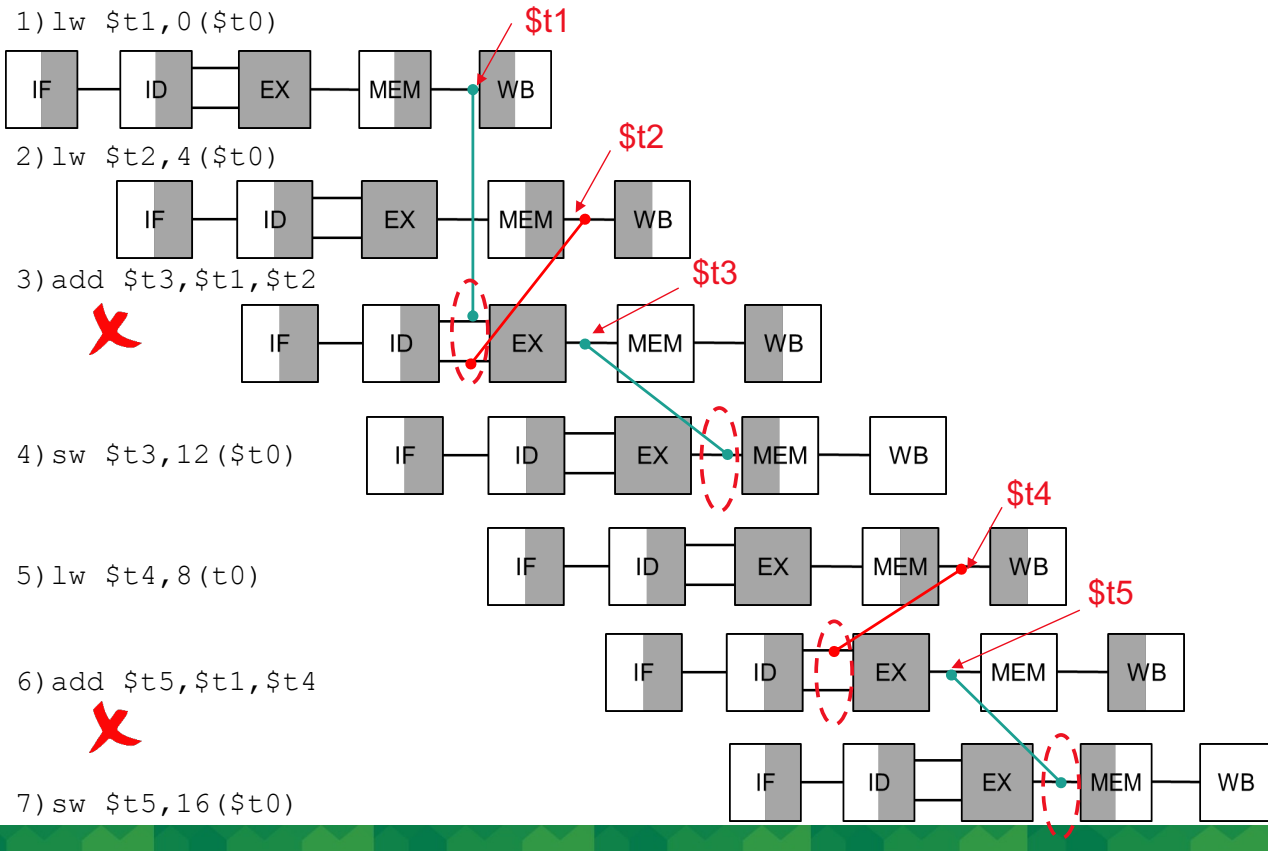
```
a = b + e;
C = b + f;
```

Here is the generated MIPS code for this segment, assuming all variables are in memory and are addressable as offsets from $t0:

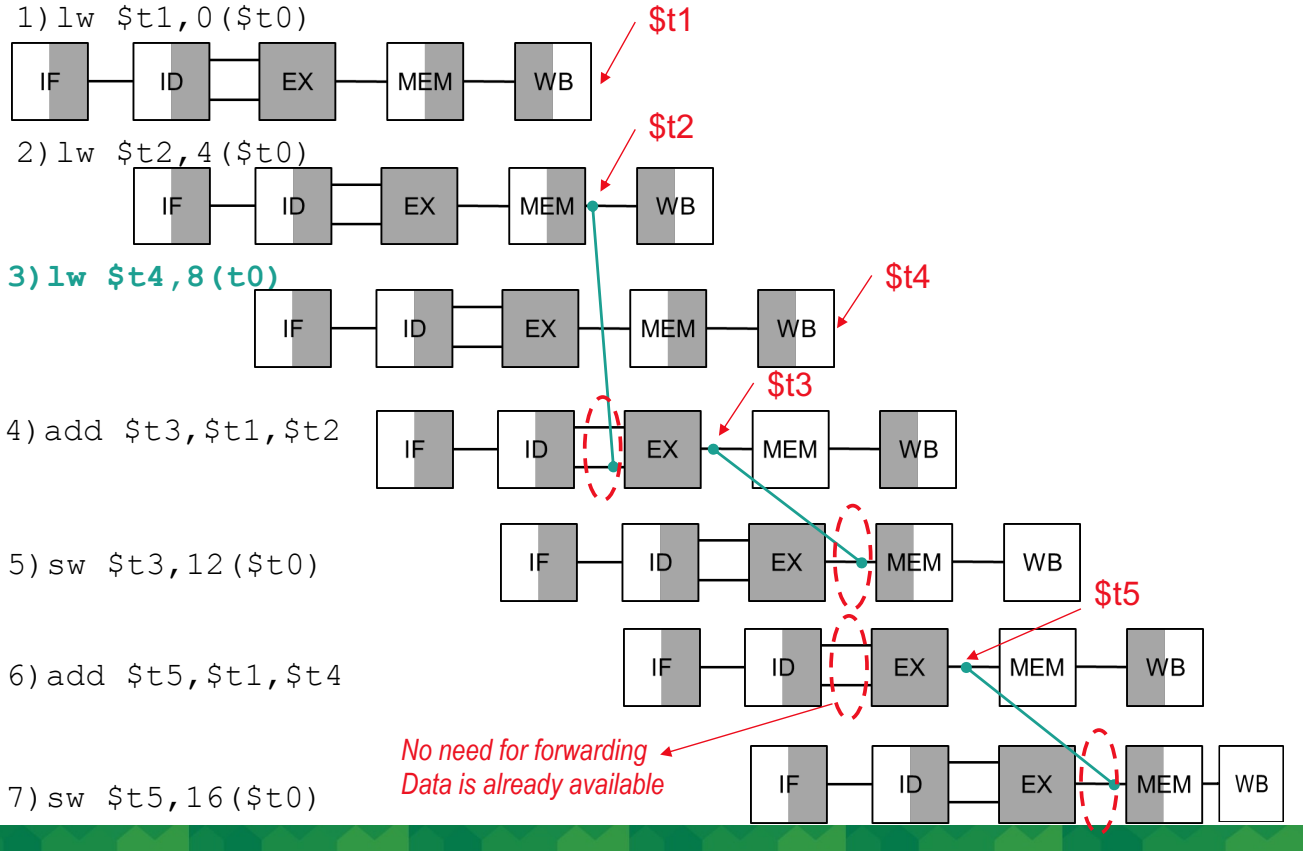
```
lw      $t1, 0($t0)
lw      $t2, 4($t0)
add     $t3, $t1,$t2
sw      $t3, 12($t0)
lw      $t4, 8(t0)
add     $t5, $t1, $t4
sw      $t5, 16($t0)
```

Sought: Find the hazards in the following code segment and reorder the instructions to avoid any pipeline stalls. (The pipelined Datapath is equipped with forwarding/bypassing mechanism)

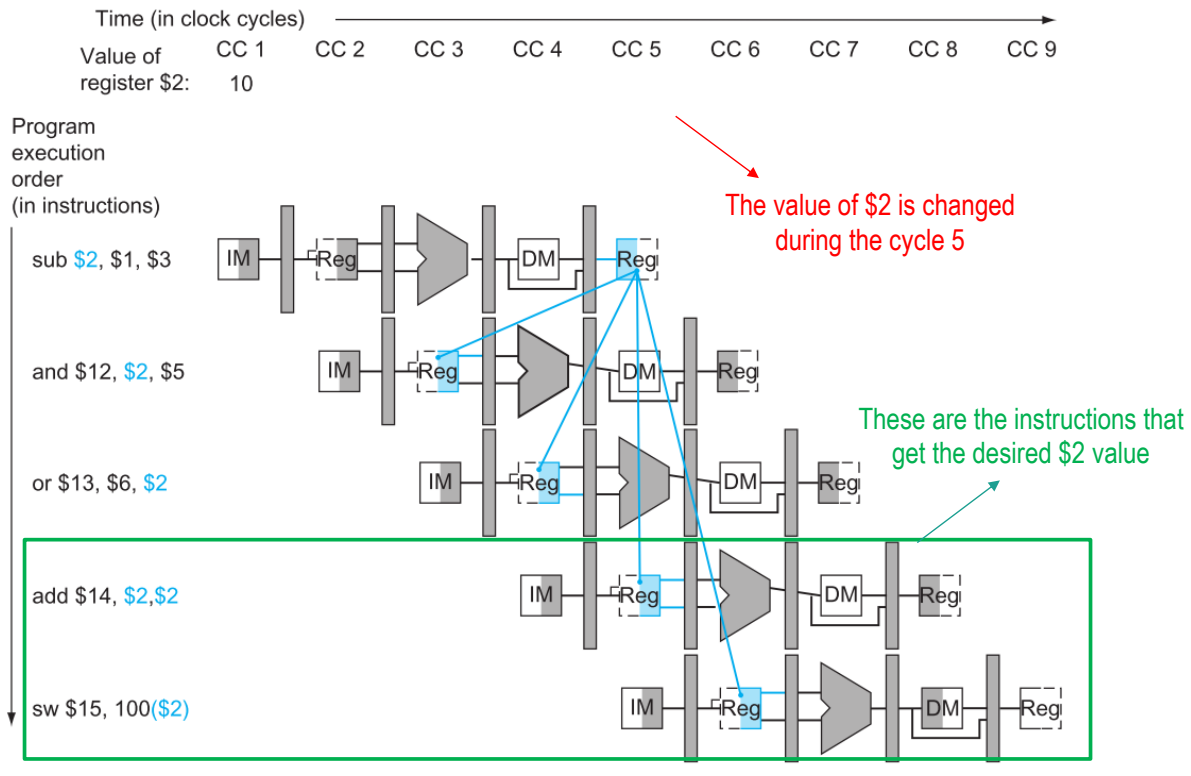
Data Hazards: reordering



Data Hazards: Reordering



Data Hazards Detection



Other types of data hazard

- Write After Read (WAR) Hazard a.k.a "*antidependence*"

```
add $t1, $t0, $s1
and $t0, $s3, $s4
```

  - WAR Hazard is the result of the developer's choice to choose the same register for two independent instructions, which can happen due to the limitation in the number available registers in MIPS processor, i.e. 32 registers.
  - *Antidependence* does not cause a hazard in simple MIPS pipeline, however, it could lead to a data hazard in out-of-order processors if the dependent instruction (`and`) is moved too early (before `add` in this example)

- Write After Write (WAW) Hazard a.k.a "*output dependence*"

```
add $t0, $t1, $s1
and $t0, $s3, $s4
```

  - If the `and` instruction is moved before `add` instructions, the following instructions would work with the wrong data

Loop Unrolling and scheduling

- Let's see how the compiler can increase the amount of instruction level parallelism by unrolling loops.
- Image the below code segment, in which **x** and **s** are double-precision floating point numbers:

```
for (i=999; i>=0; i=i-1)
        x[i]=x[i]+s;
```

- Assuming `$s1` include the address of the element in the array with the highest address, `$s2` is the address of the last element in the array, and `$f2` contains the value `s`, the MIPS code for the above loop would be:

```
Loop: l.d   $f0,0($s1)      #$f0=array element
      add.d $f4,$f0,$f2     #add s is $f2
      s.d   $f4,0($s1)      #store result
      addi  $s1,$s1,-8      #decrement pointer
      bne   $s1,$s2,Loop    #branch $s1!= $s1
```

Loop Unrolling and scheduling

- Given the below latencies for the dependent Floating-Point (FP) and Integer operations in a MIPS processor:

Instruction Producing Results	Instruction Using Results	Latency in clock cycles
FP ALU operation	FP ALU operation	3
FP ALU operation	Store double	2
Load double	FP ALU operation	1
Load double	Store double	0

- Without any reordering/scheduling the loop will execute as follows:

```
Loop: l.d   $f0,0($s1)
      add.d $f4,$f0,$f2
      s.d   $f4,0($s1)
      addi  $s1,$s1,-8
      bne   $s1,$s2,Loop
```

```
Loop: l.d   $f0,0($s1)
      <Stall>
      add.d $f4,$f0,$f2
      <Stall>
      <Stall>
      s.d   $f4,0($s1)
      addi  $s1,$s1,-8
      bne   $s1,$s2,Loop
```

IPC= 5 inst/ 8 clk = 0.625

Loop Unrolling

- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead
- Use different registers per replication
  - Called “register renaming”
  - Avoid loop-carried “anti-dependencies”
  - Increasing Instruction-Level Parallelism:
  - Avoiding Data Hazards

Loop Unrolling and scheduling

- Image the below code segment, in which **x** and **s** are double-precision floating point numbers:

```
for (i=2; i>=0; i=i-1)
    x[i]=x[i]+s;
```

- Assuming `$s1` include the address of the element in the array with the highest address, `$s2` is the address of the last element in the array, and `$f2` contains the value `s`, the MIPS code for the above loop would be:

```
Loop: l.d    $f0,0($s1)
      add.d  $f4,$f0,$f2
      s.d    $f4,0($s1)
      addi   $s1,$s1,-8
      bne    $s1,$s2,Loop
```

Unrolling

```
l.d    $f0,0($s1)
add.d  $f4,$f0,$f2
s.d    $f4,0($s1)
addi   $s1,$s1,-8

l.d    $f0,0($s1)
add.d  $f4,$f0,$f2
s.d    $f4,0($s1)
addi   $s1,$s1,-8

l.d    $f0,0($s1)
add.d  $f4,$f0,$f2
s.d    $f4,0($s1)
addi   $s1,$s1,-8
```

```
l.d    $f0,0($s1)
add.d  $f4,$f0,$f2
s.d    $f4,0($s1)

l.d    $f0,-8($s1)
add.d  $f4,$f0,$f2
s.d    $f4,-8($s1)

l.d    $f0,-16($s1)
add.d  $f4,$f0,$f2
s.d    $f4,-16($s1)
addi   $s1,$s1,-24
```

Loop Unrolling and scheduling

- What is the maximum IPC that can be achieved when unrolling the loop with a **factor of 3?**

```
Loop: l.d    $f0,0($s1)
      <Stall>
      add.d $f4,$f0,$f2
      <Stall>
      <Stall>
      s.d    $f4,0($s1)

      l.d    $f0,-8($s1)
      <Stall>
      add.d $f4,$f0,$f2
      <Stall>
      <Stall>
      s.d    $f4,-8($s1)

      l.d    $f0,-16($s1)
      <Stall>
      add.d $f4,$f0,$f2
      <Stall>
      <Stall>
      s.d    $f4,-16($s1)

      addi   $s1,$s1,-24
      bne    $s1,$s2,Loop
```

Register renaming to remove antidependance

```
Loop: l.d    $f0,0($s1)
      <Stall>
      add.d $f4,$f0,$f2
      <Stall>
      <Stall>
      s.d    $f4,0($s1)

      l.d    $f6,-8($s1)
      <Stall>
      add.d $f8,$f6,$f2
      <Stall>
      <Stall>
      s.d    $f8,-8($s1)

      l.d    $f10,-16($s1)
      <Stall>
      add.d $f12,$f10,$f2
      <Stall>
      <Stall>
      s.d    $f12,-16($s1)

      addi   $s1,$s1,-24
      bne    $s1,$s2,Loop
```

Loop Unrolling and scheduling

- Now let's do reordering to increase IPC:

```
Loop: l.d   $f0,0($s1)
      <Stall>
      add.d $f4,$f0,$f2
      <Stall>
      <Stall>
      s.d   $f4,0($s1)

      l.d   $f6,-8($s1)
      <Stall>
      add.d $f8,$f6,$f2
      <Stall>
      <Stall>
      s.d   $f8,-8($s1)

      l.d   $f10,-16($s1)
      <Stall>
      add.d $f12,$f10,$f2
      <Stall>
      <Stall>
      s.d   $f12,-16($s1)

      addi  $s1,$s1,-24
      bne   $s1,$s2,Loop
```

Reordering/Scheduling

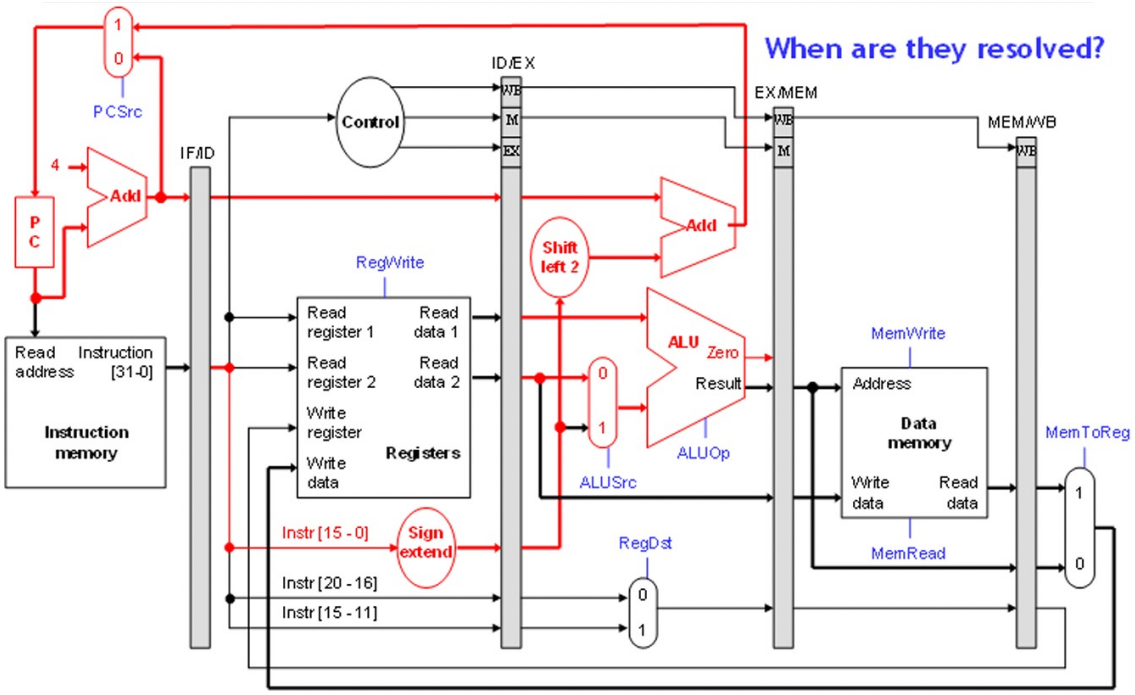
```
Loop: l.d   $f0,0($s1)
      l.d   $f6,-8($s1)
      l.d   $f10,-16($s1)
      add.d $f4,$f0,$f2
      add.d $f8,$f6,$f2
      add.d $f12,$f10,$f2
      s.d   $f4,0($s1)
      s.d   $f8,-8($s1)
      s.d   $f12,-16($s1)
      addi  $s1,$s1,-24
      bne   $s1,$s2,Loop
```

IPC= 11 inst/ 11 clk = 1

Control hazards

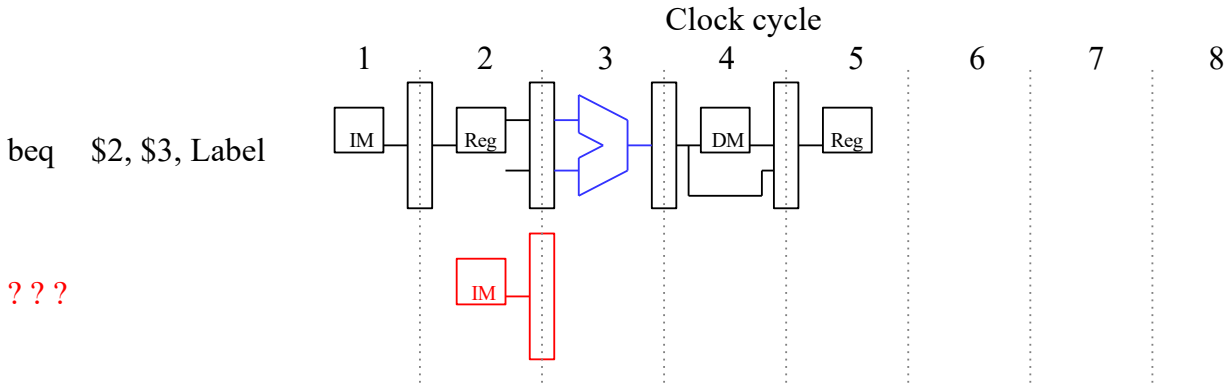
- Control Hazards arise from the need to make a decision based on the results of one instruction while other are executing.
- Branch Instruction: The pipeline cannot know what the next instruction should be!

Branches in the original Pipelined Datapath



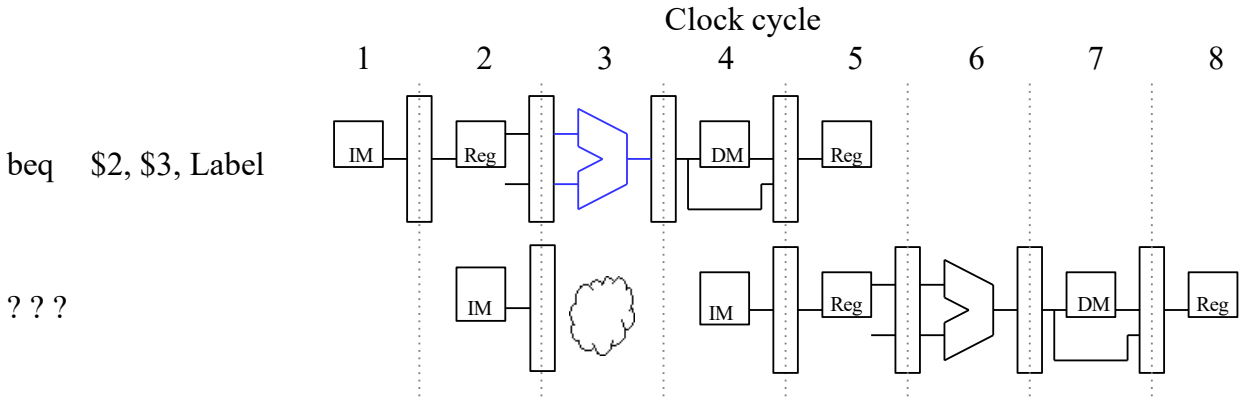
Branches

- Most of the work for a branch computation is done in the EX stage.
  - The branch target address is computed.
  - The source registers are compared by the ALU, and the Zero flag is set or cleared accordingly.
- Thus, the branch decision cannot be made until the end of the EX stage.
  - But we need to know which instruction to fetch next, in order to keep the pipeline running!
  - This leads to what's called a control hazard.



Stalling is one Solution

- Again, stalling is always one possible solution.



- Here we just stall until cycle 4, after we do make the branch decision.

Branch prediction

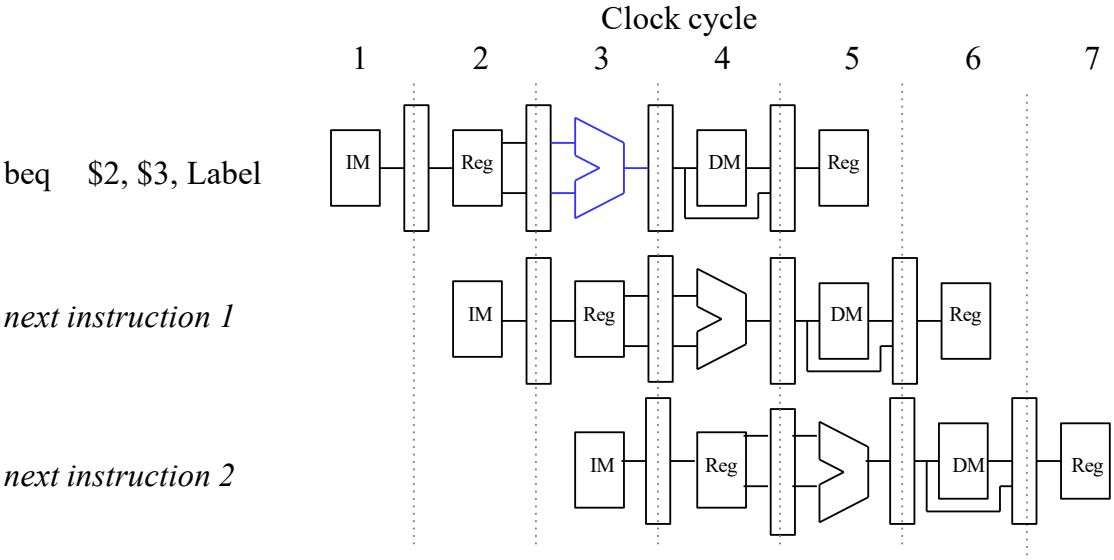
- Branch Prediction: A method of resolving a branch hazard that assumes a given outcome for the branch and proceeds according to that assumption rather then waiting to ensure the actual outcome.

- Possible Static Approaches:
  1. always predict that branches will be untaken.
  2. always predict that branches at the bottom of the loops will be taken, and branches at the top of the loop will be untaken.

- Dynamic Branch Prediction: Keep history for each branch as taken or untaken, and then use the recent past behavior to predict the future.

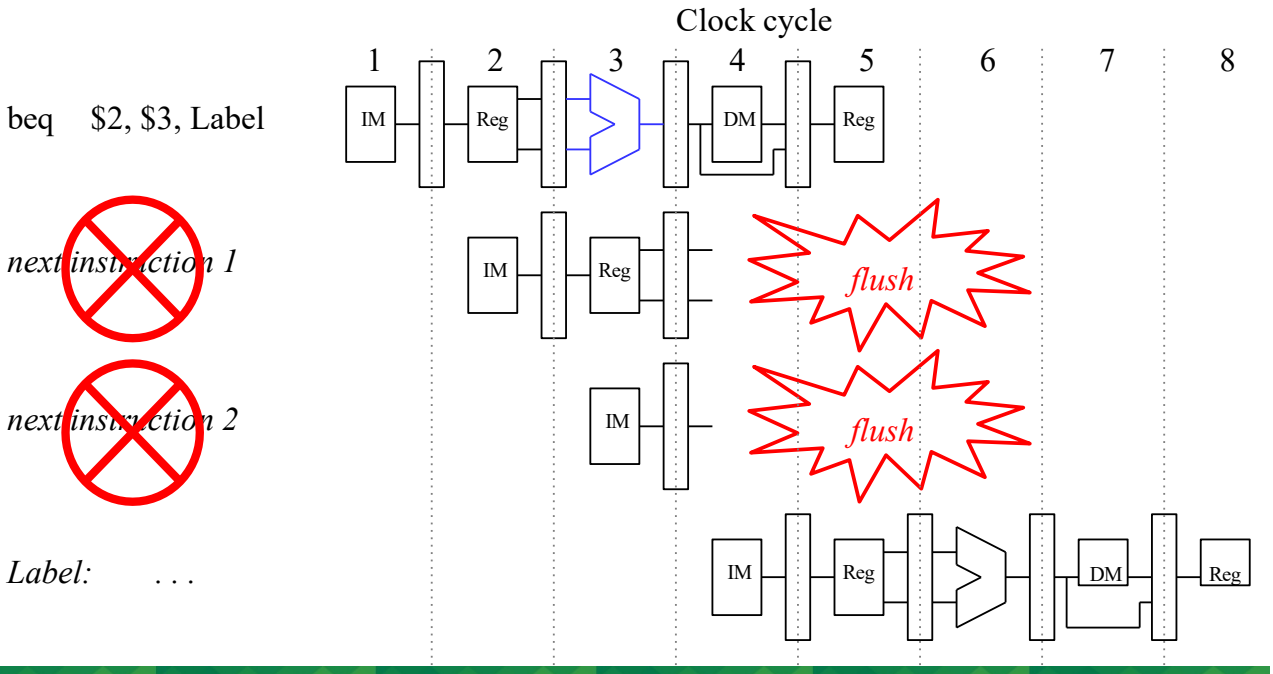
Branch prediction

- Another approach is to *guess* whether or not the branch is taken.
  - In terms of hardware, it's easier to assume the branch is *not* taken.
  - This way we just increment the PC and continue execution, as for normal instructions.
- If we're correct, then there is no problem and the pipeline keeps going at full speed.



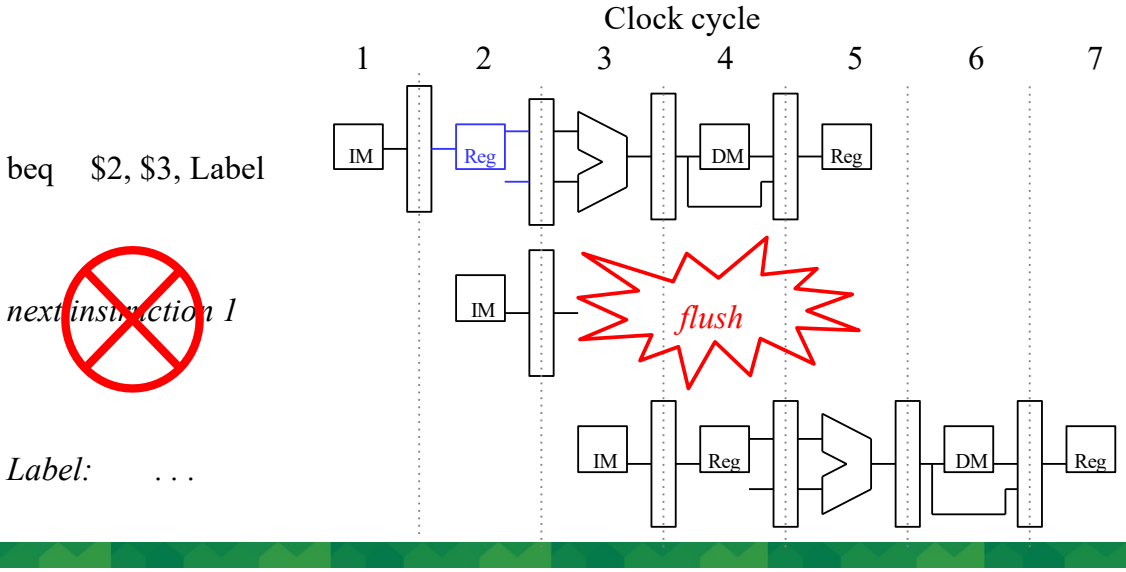
Branch misprediction

- If our guess is wrong, then we would have already started executing two instructions incorrectly. We'll have to discard, or flush, those instructions and begin executing the right ones from the branch target address, Label.



Implementing branches

- We can actually decide the branch a little earlier, in ID instead of EX.
  - Our sample instruction set has only a BEQ.
  - We can add a small comparison circuit to the ID stage, after the source registers are read.
- Then we would only need to flush one instruction on a misprediction.



Implementing flushes

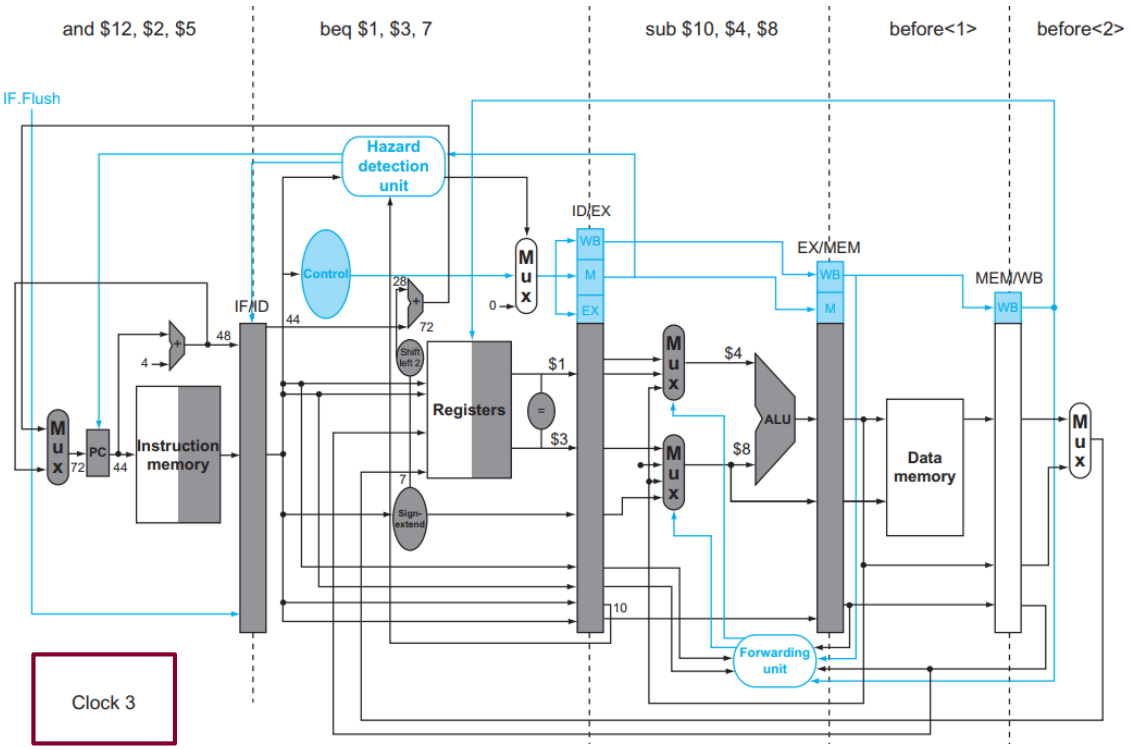
- We must flush one instruction (in its IF stage) if the previous instruction is BEQ and its two source registers are equal.
- We can flush an instruction from the IF stage by replacing it in the IF/ID pipeline register with a harmless nop instruction.
  - MIPS uses sll $0, $0, 0 as the nop instruction.
  - This happens to have a binary encoding of all 0s: 0000 .... 0000.
- Flushing introduces a bubble into the pipeline, which represents the one- cycle delay in taking the branch.

Branch prediction

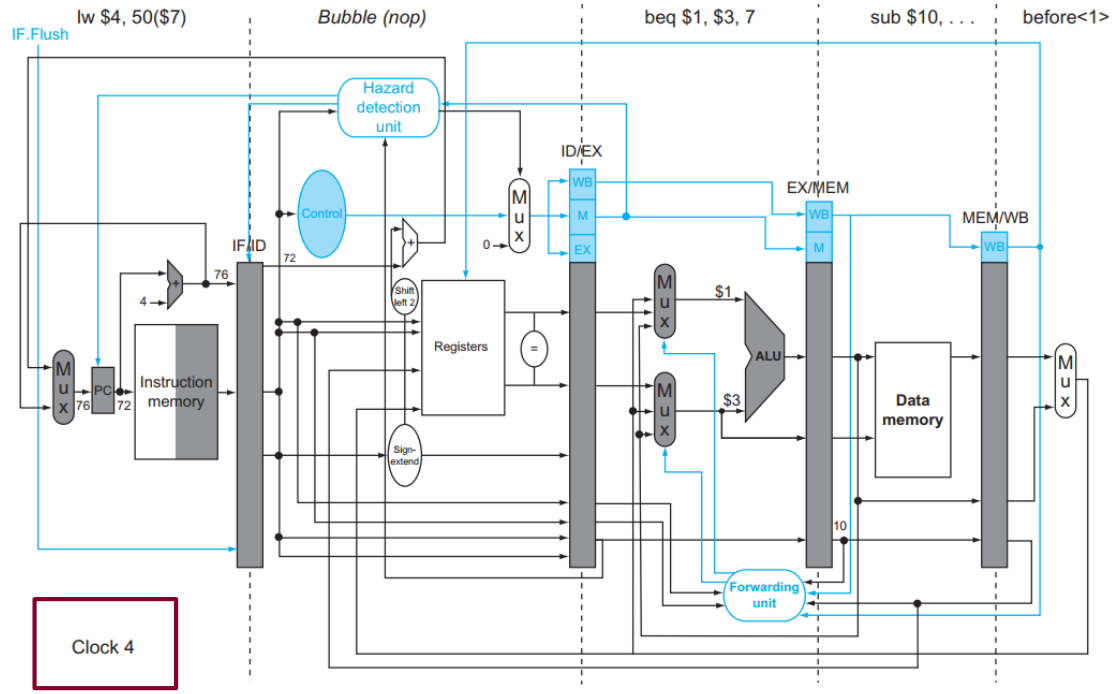
Example: Show what happens when the branch is taken in the below instruction sequence, assuming that branch is predicted as “not taken”.

```
36 sub $10, $4, $8
40 beq $1, $3, 7 # PC-relative branch to 40+4+7*4=72
44 and $12, $2, $5
48 or  $13, $2, $6
52 add $14, $4, $2
56 slt $15, $6, $7
. . .
72 lw $4, 50($7)
```

Branch prediction

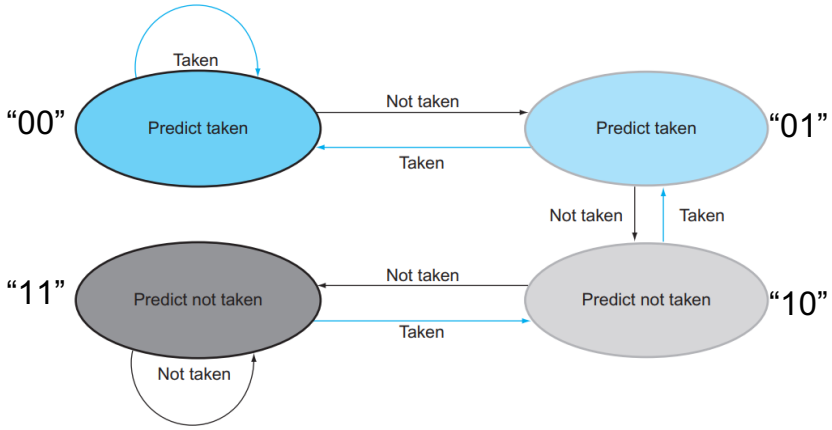


Branch prediction



Dynamic Branch prediction

2-bit Dynamic Branch Prediction Scheme:

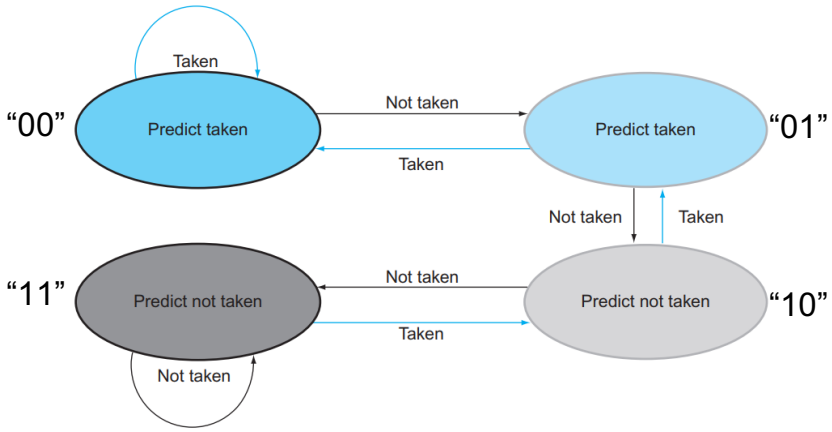


Given: Consider a loop that branches seven times in a row and then is not taken once. What is the prediction accuracy using the above 2-bit prediction scheme with the initial state of "00"?

Branch	taken	taken	taken	taken	taken	taken	taken	untaken
State	00	00	00	00	00	00	00	00 → 01
Prediction	True	True	True	True	True	True	True	False
Accuracy	7/8 = 87.5%							

Dynamic Branch prediction

2-bit Dynamic Branch Prediction Scheme:



Given: Consider a loop that branches seven times in a row and then is not taken once. What is the prediction accuracy using the above 2-bit prediction scheme with the initial state of "10"?

Branch	taken	taken	taken	taken	taken	taken	taken	untaken
State	10	01	00	00	00	00	00	00 → 01
Prediction	False	True	True	True	True	True	True	False
Accuracy	6/8 = 75%							