Computer Organization and Architecture

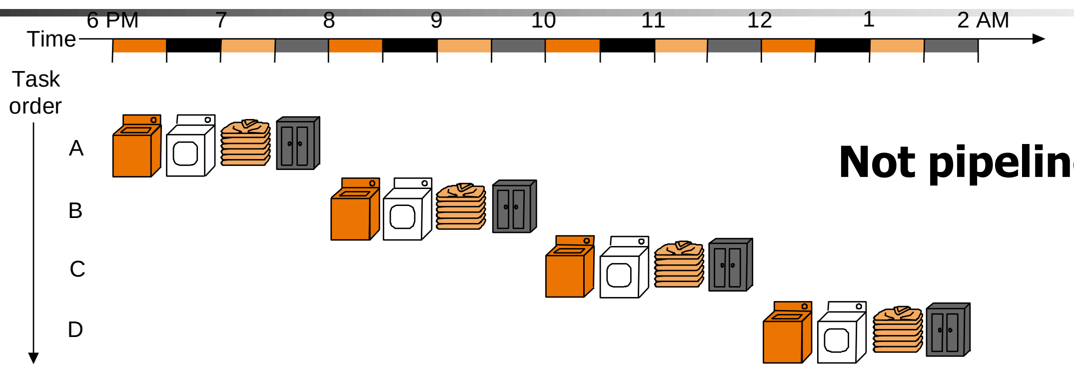
Instructor: Dr. Rushdi Abu Zneit

Slide Sources: Patterson & Hennessy

COD Ch. 6
Enhancing Performance with Pipelining

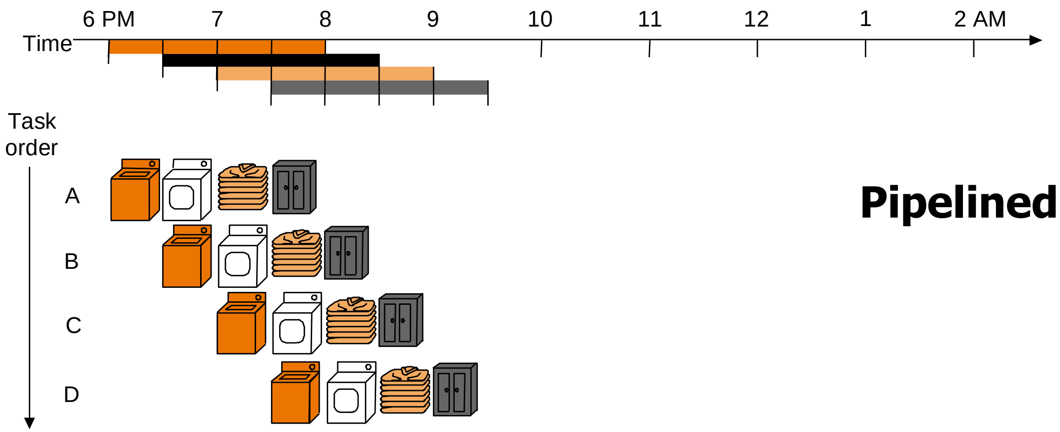
Pipelining

- Start work ASAP!! Do not waste time!



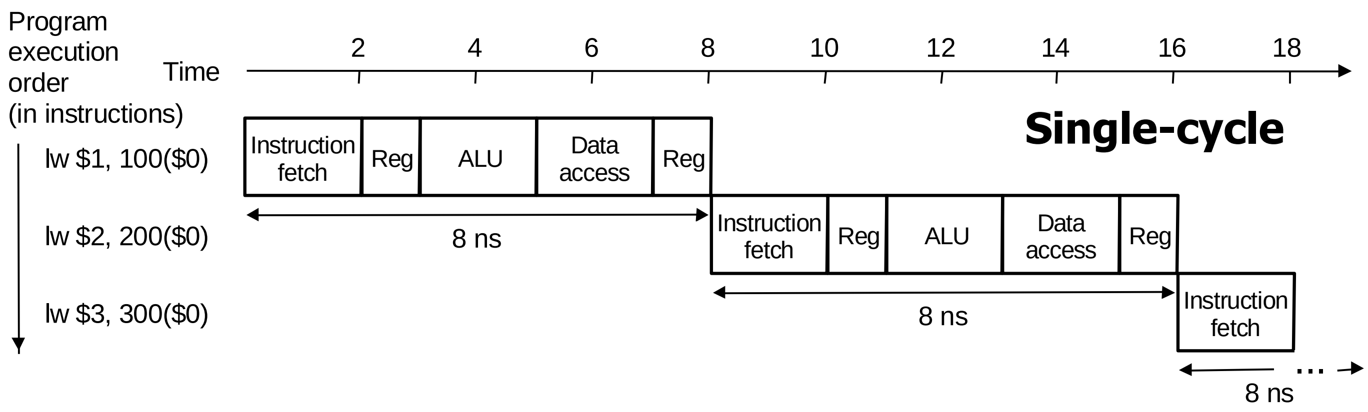
Not pipelined

Assume 30 min. each task – wash, dry, fold, store – and that separate tasks use separate hardware and so can be overlapped

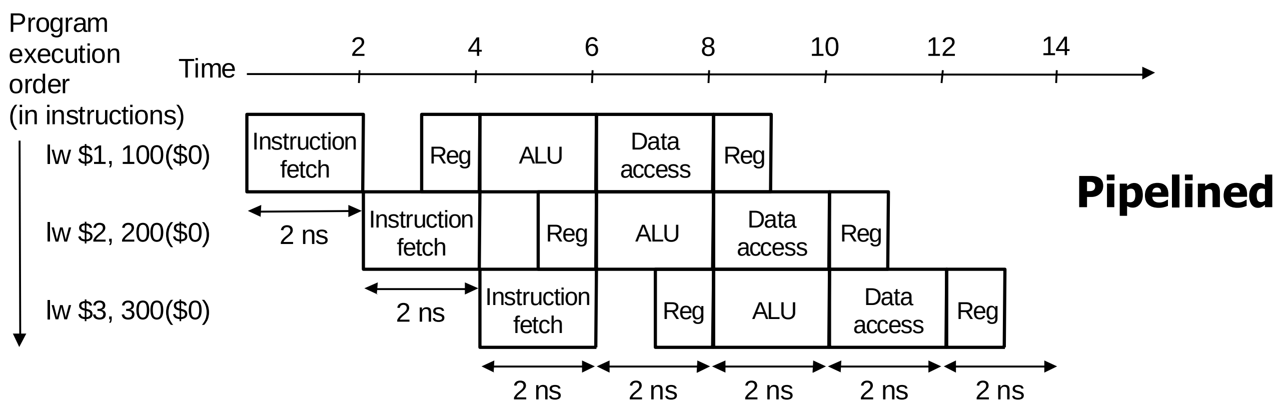


Pipelined

Pipelined vs. Single-Cycle Instruction Execution: the Plan



Assume 2 ns for memory access, ALU operation; 1 ns for register access: therefore, single cycle clock 8 ns; pipelined clock cycle 2 ns.



Pipelining: Keep in Mind

- Pipelining *does not reduce latency* of a single task, it *increases throughput* of entire workload
- Pipeline rate *limited by longest stage*
  - *potential* speedup = number pipe stages
  - *unbalanced lengths* of pipe stages reduces speedup
- Time to *fill* pipeline and time to *drain* it – when there is *slack* in the pipeline – reduces speedup

Example Problem

- *Problem: for the laundry fill in the following table when*
    1. *the stage lengths are 30, 30, 30 30 min., resp.*
    2. *the stage lengths are 20, 20, 60, 20 min., resp.*

Person	Unpipelined finish time	Pipeline 1 finish time	Ratio unpipelined to pipeline 1	Pipeline 2 finish time	Ratio unpiplelined to pipeline 2
1					
2					
3					
4					
n					

- *Come up with a formula for pipeline speed-up!*

Pipelining MIPS

- What makes it easy with MIPS?
  - all *instructions are same length*
    - so fetch and decode stages are similar for all instructions
  - just a *few instruction formats*
    - simplifies instruction decode and makes it possible in one stage
  - *memory operands appear only in load/stores*
    - so memory access can be deferred to exactly one later stage
  - *operands are aligned in memory*
    - one data transfer instruction requires one memory access stage

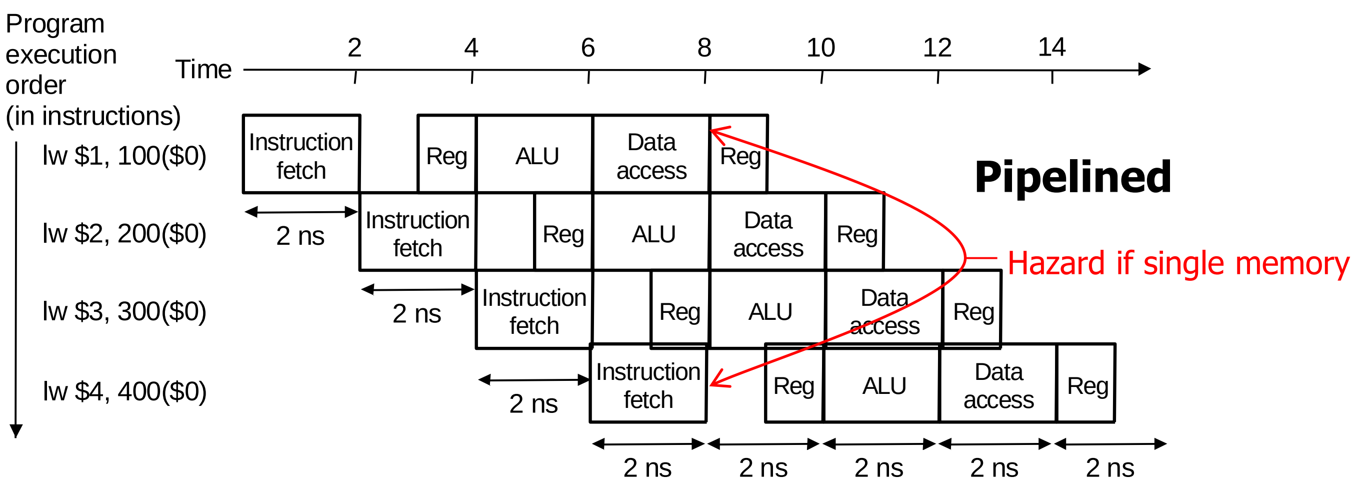
Pipelining MIPS

- What makes it hard?
  - *structural hazards*: different instructions, at different stages, in the pipeline want to use the same hardware resource
  - *control hazards*: succeeding instruction, to put into pipeline, depends on the outcome of a previous branch instruction, already in pipeline
  - *data hazards*: an instruction in the pipeline requires data to be computed by a previous instruction still in the pipeline

- Before actually building the pipelined datapath and control we first briefly examine these potential hazards individually…

Structural Hazards

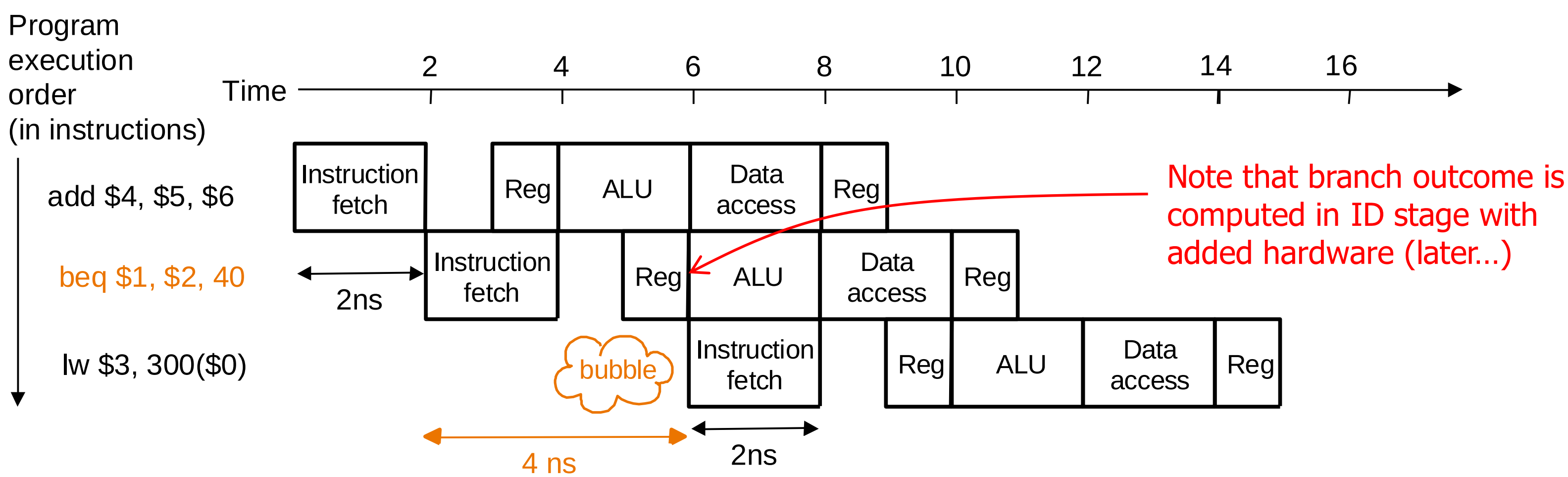
- *Structural hazard*: inadequate hardware to simultaneously support all instructions in the pipeline in the same clock cycle
- E.g., suppose *single – not separate –* instruction and data memory in pipeline below with *one read port*
  - then a structural hazard between first and fourth `lw` instructions



- *MIPS was designed to be pipelined*: structural hazards are easy to avoid!

Control Hazards

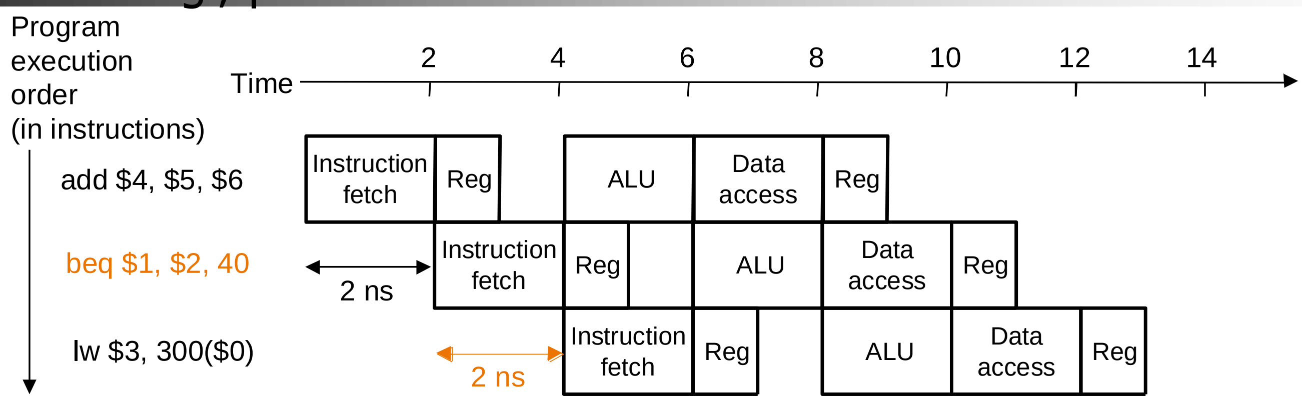
- *Control hazard*: need to make a decision based on the result of a previous instruction still executing in pipeline
- <u>Solution 1</u> *Stall* the pipeline



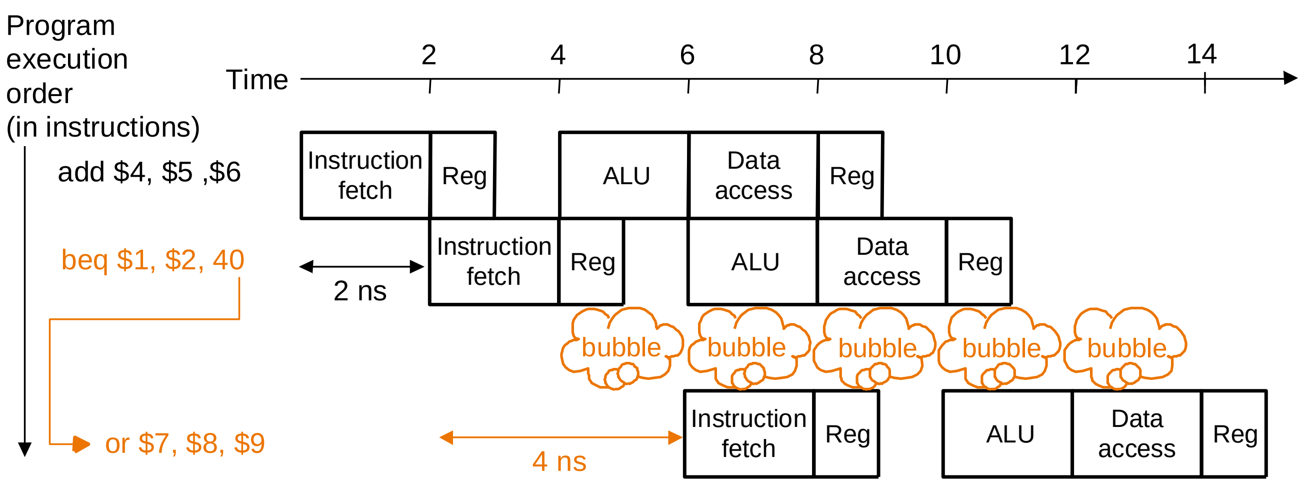
Pipeline stall

Control Hazards

- <u>Solution 2</u> *Predict* branch outcome
  - e.g., predict *branch-not-taken* :



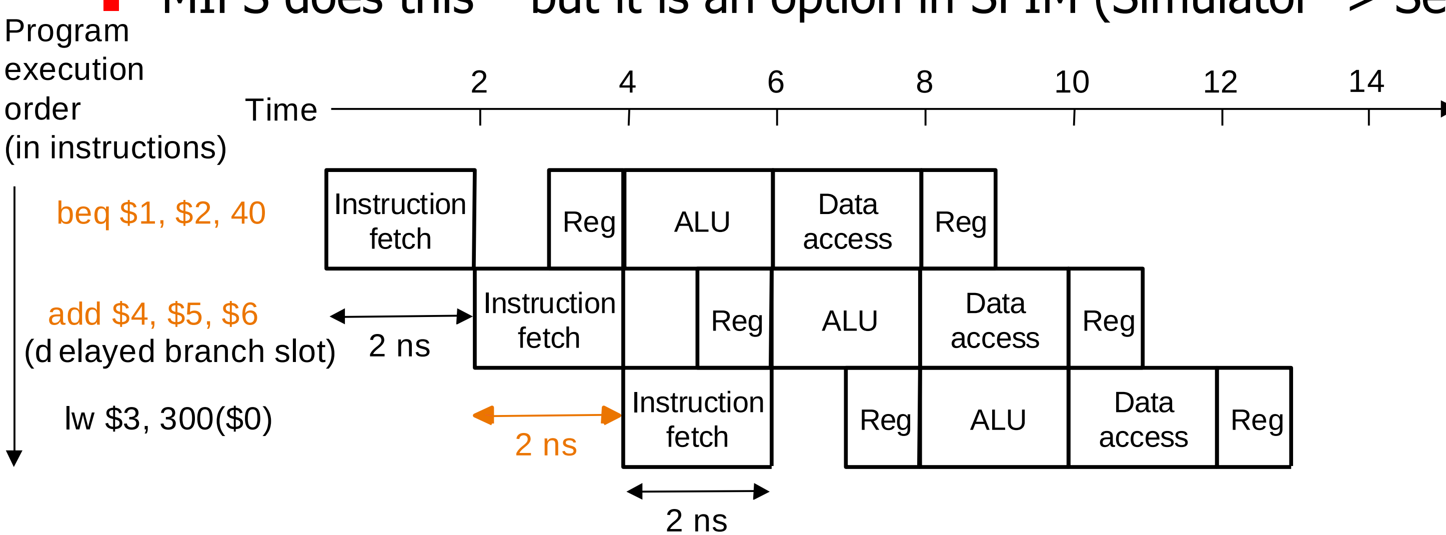
Prediction success



Prediction failure: undo (=flush) `lw`

Control Hazards

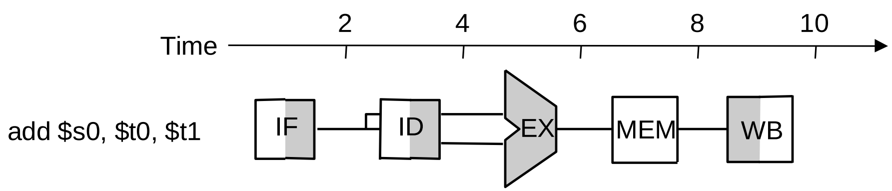
- <u>Solution 3</u> *Delayed branch:* always execute the sequentially next statement with the branch executing after one instruction delay – compiler's job to find a statement that can be put in the slot that is independent of branch outcome
  - MIPS does this – but it is an option in SPIM (Simulator -> Settings)



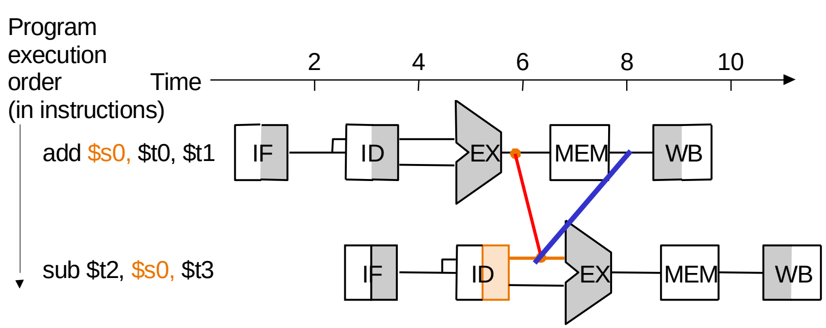
Delayed branch `beq` is followed by `add` that is independent of branch outcome

Data Hazards

- *Data hazard*: instruction needs data from the result of a previous instruction still executing in pipeline
- Solution *Forward* data if possible…



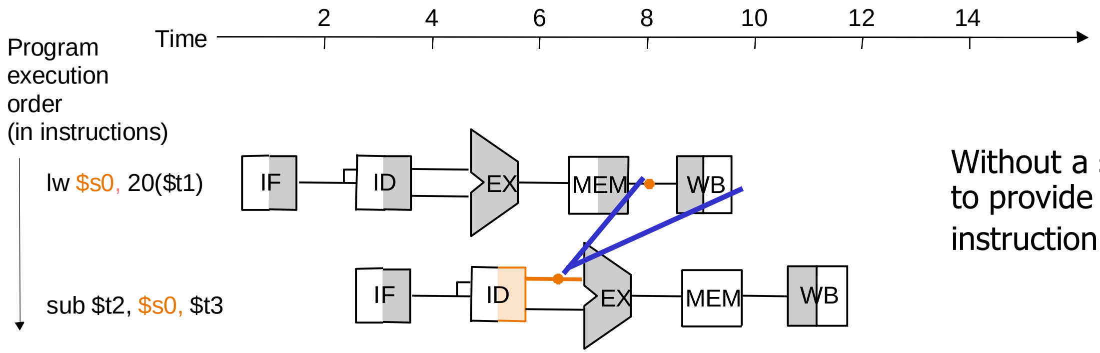
Instruction pipeline diagram: shade indicates use – left=write, right=read



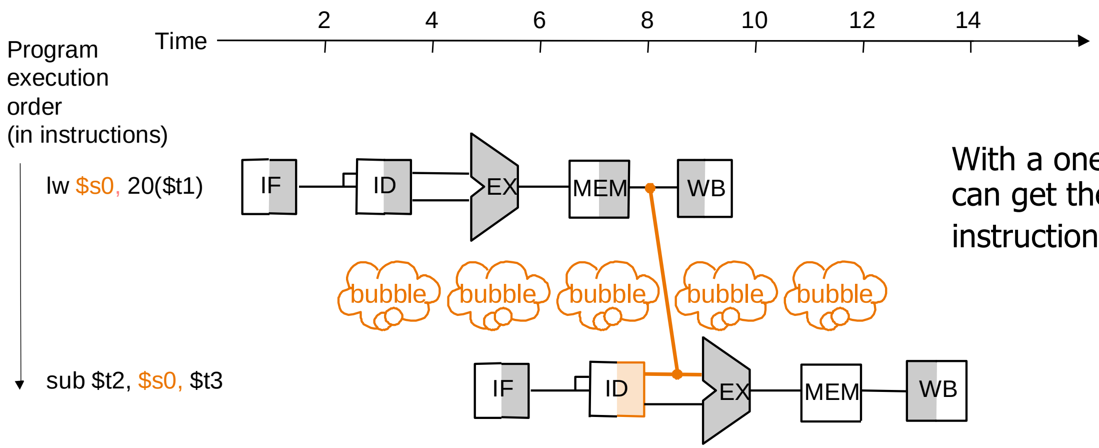
Without forwarding – blue line – data has to go back in time; with forwarding – red line – data is available in time

Data Hazards

- Forwarding may not be enough
  - e.g., if an R-type instruction following a load uses the result of the load – called *load-use data hazard*



Without a stall it is impossible to provide input to the `sub` instruction in time



With a one-stage stall, forwarding can get the data to the `sub` instruction in time

Reordering Code to Avoid Pipeline Stall (Software Solution)

- Example:

```
lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t2, 0($t1)
sw $t0, 4($t1)
```

Data hazard

- Reordered code:

```
lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t0, 4($t1)
sw $t2, 0($t1)
```

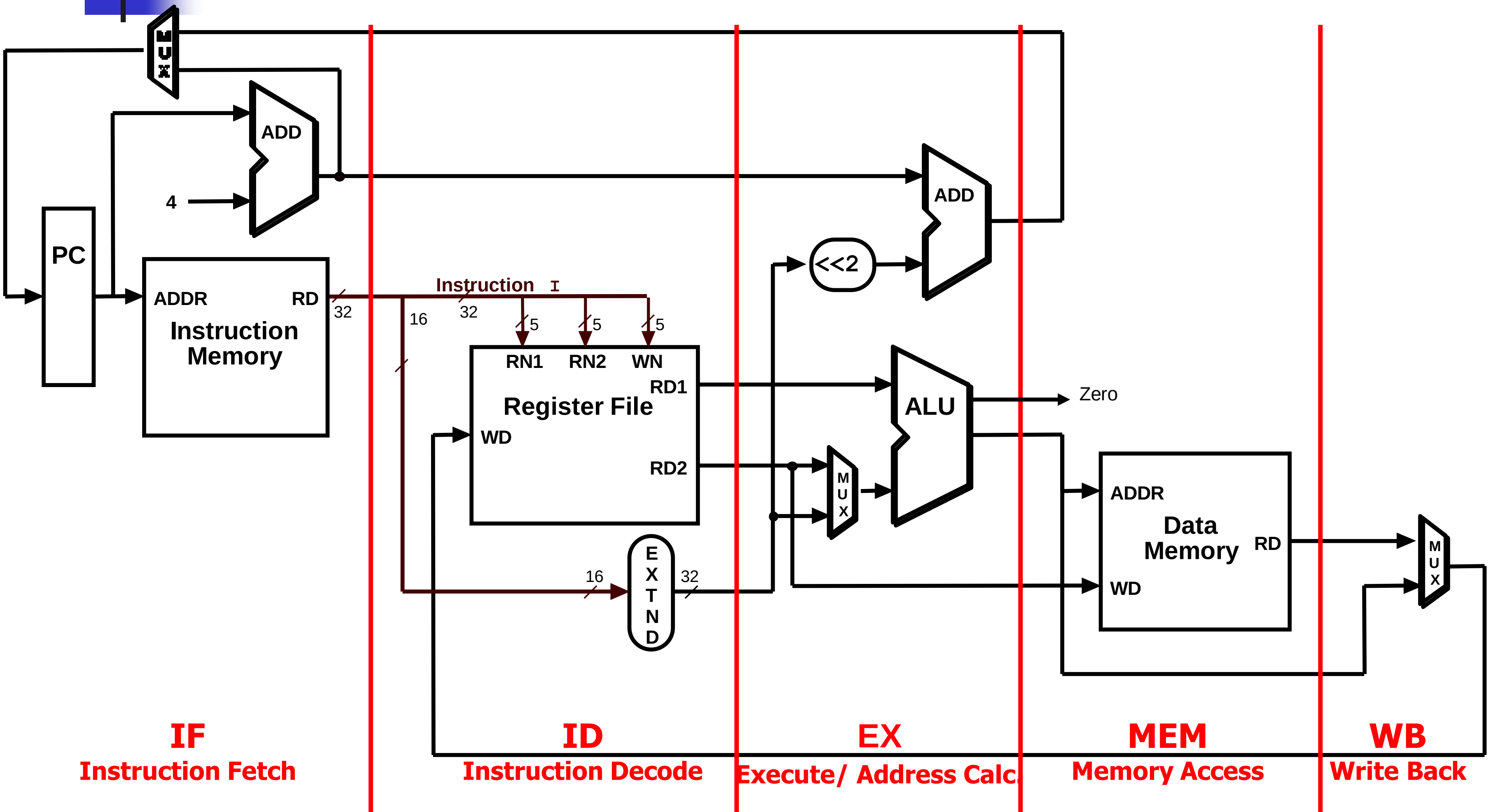
Interchanged

Pipelined Datapath

- We now move to actually building a pipelined datapath
- First recall the 5 steps in instruction execution
  1. Instruction Fetch & PC Increment (IF)
  2. Instruction Decode and Register Read (ID)
  3. Execution or calculate address (EX)
  4. Memory access (MEM)
  5. Write result into register (WB)
- Review: single-cycle processor
  - all 5 steps done in a single clock cycle
  - dedicated hardware required for each step

- *What happens if we break the execution into multiple cycles, but keep the extra hardware?*

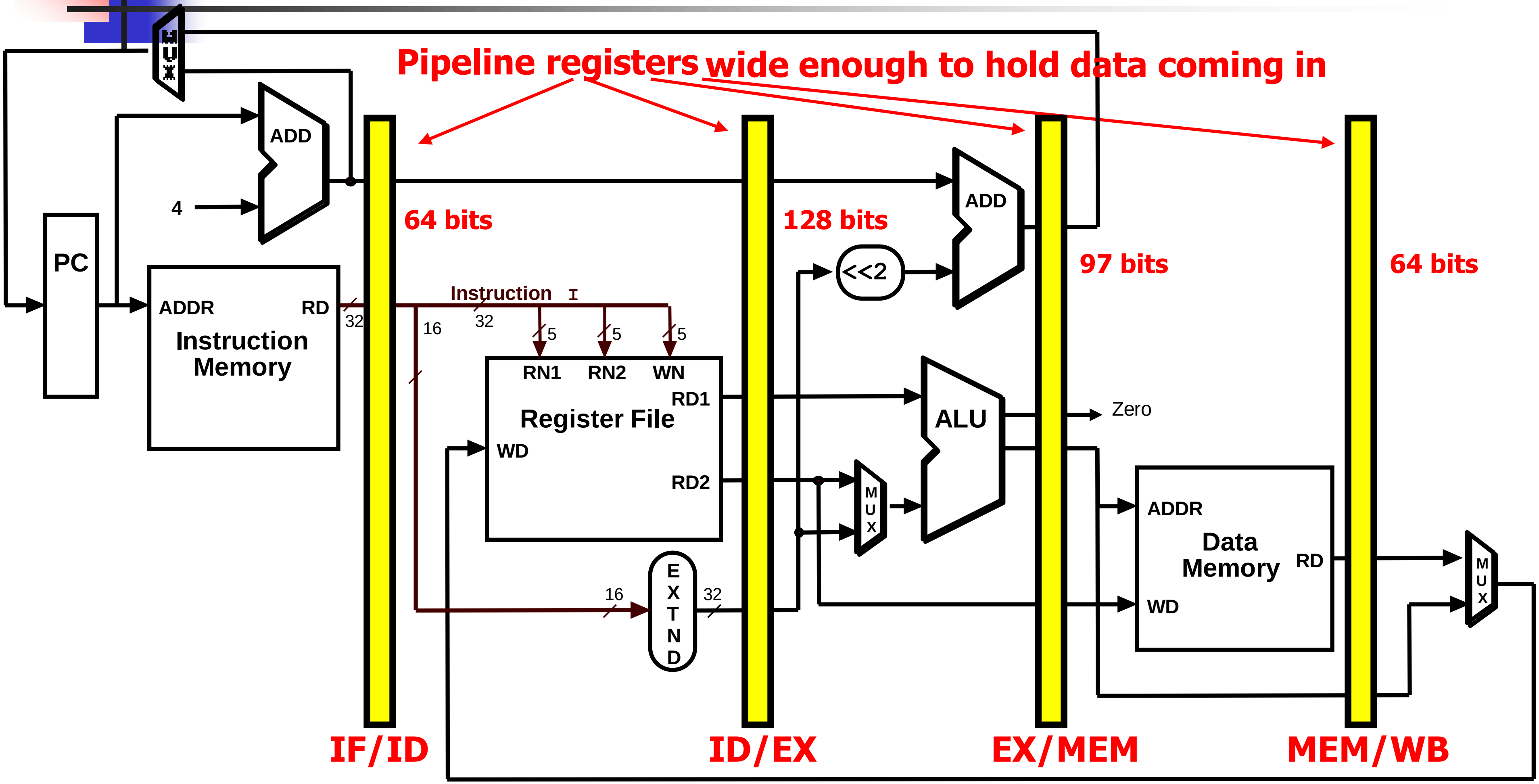
Review - Single-Cycle Datapath "Steps"



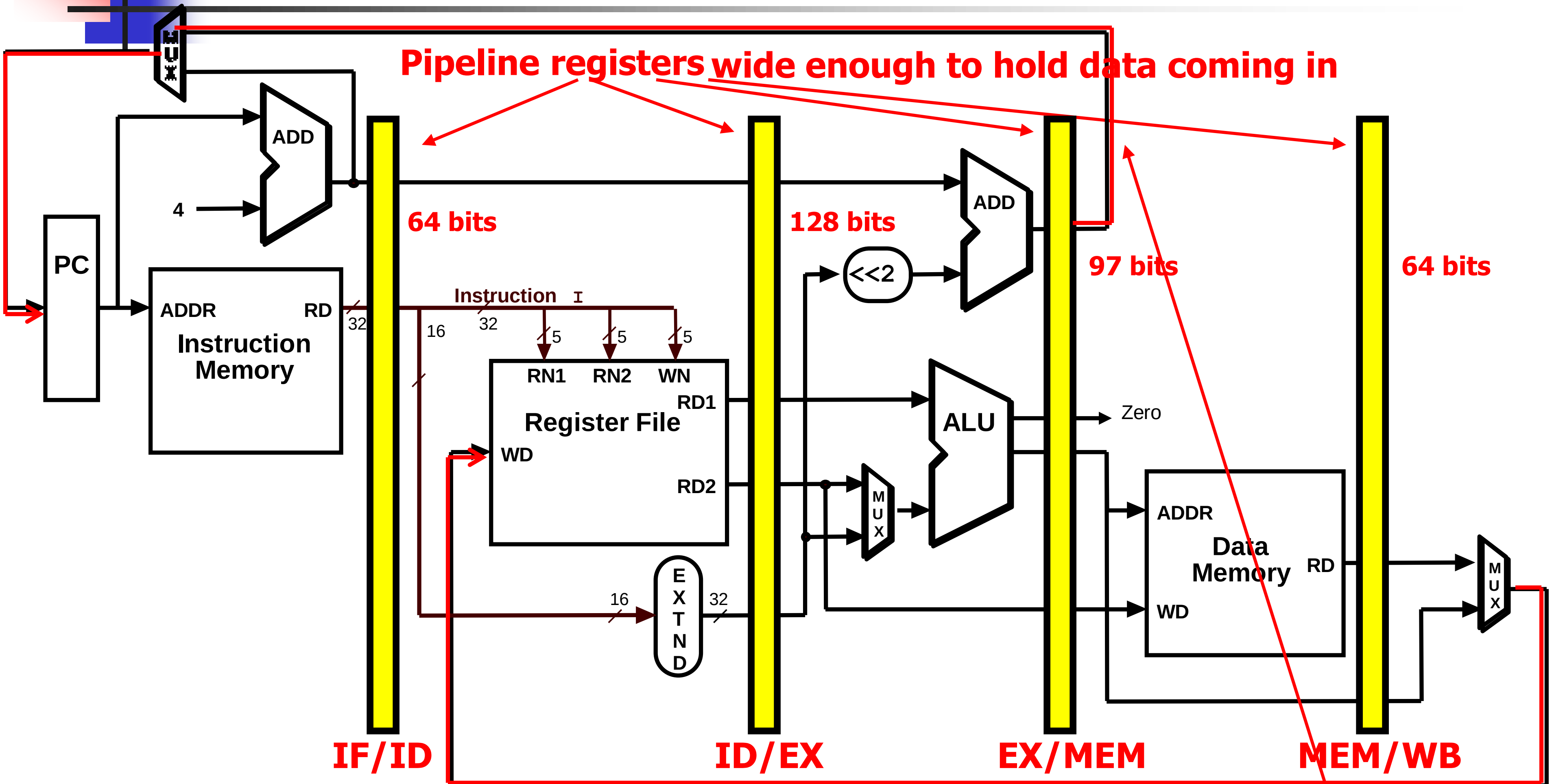
Pipelined Datapath – Key Idea

- *What happens if we break the execution into multiple cycles, but keep the extra hardware?*
  - Answer: *We may be able to start executing a new instruction at each clock cycle* - pipelining
- …but we shall need *extra* registers to hold data between cycles – *pipeline registers*

Pipelined Datapath

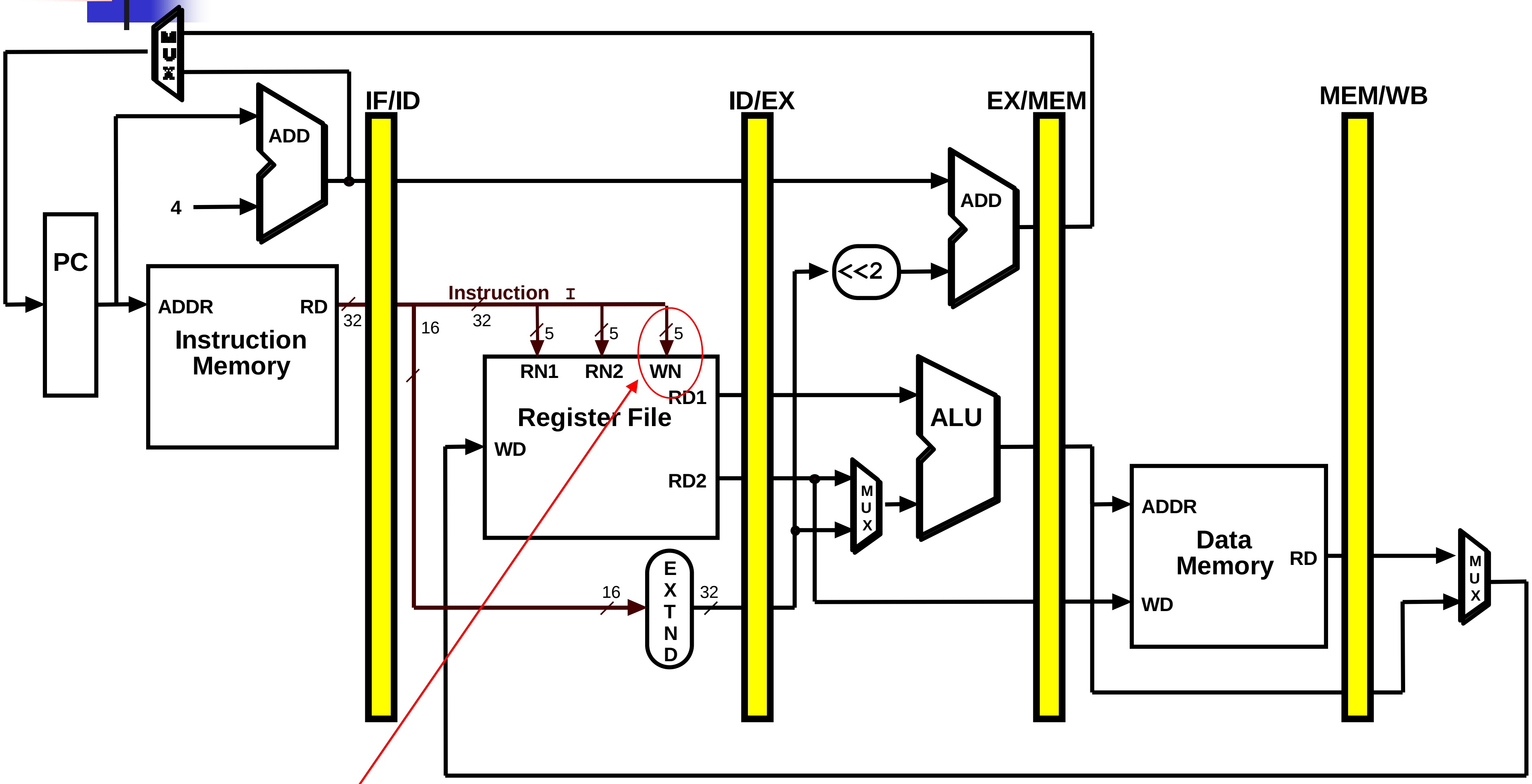


Pipelined Datapath



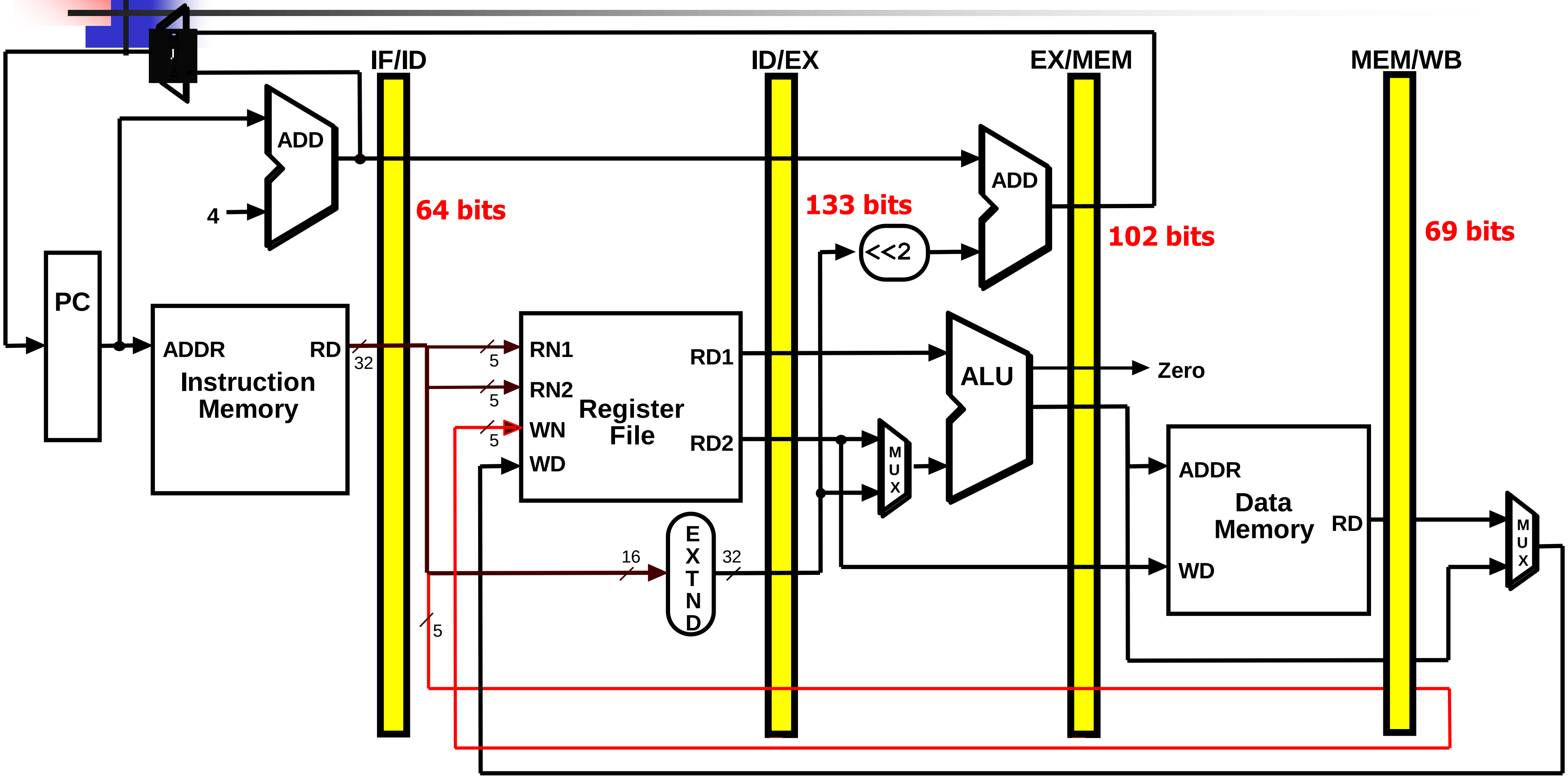
Only data flowing right to left may cause hazard…, why?

Bug in the Datapath



Write register number comes from another *later* instruction!

Corrected Datapath



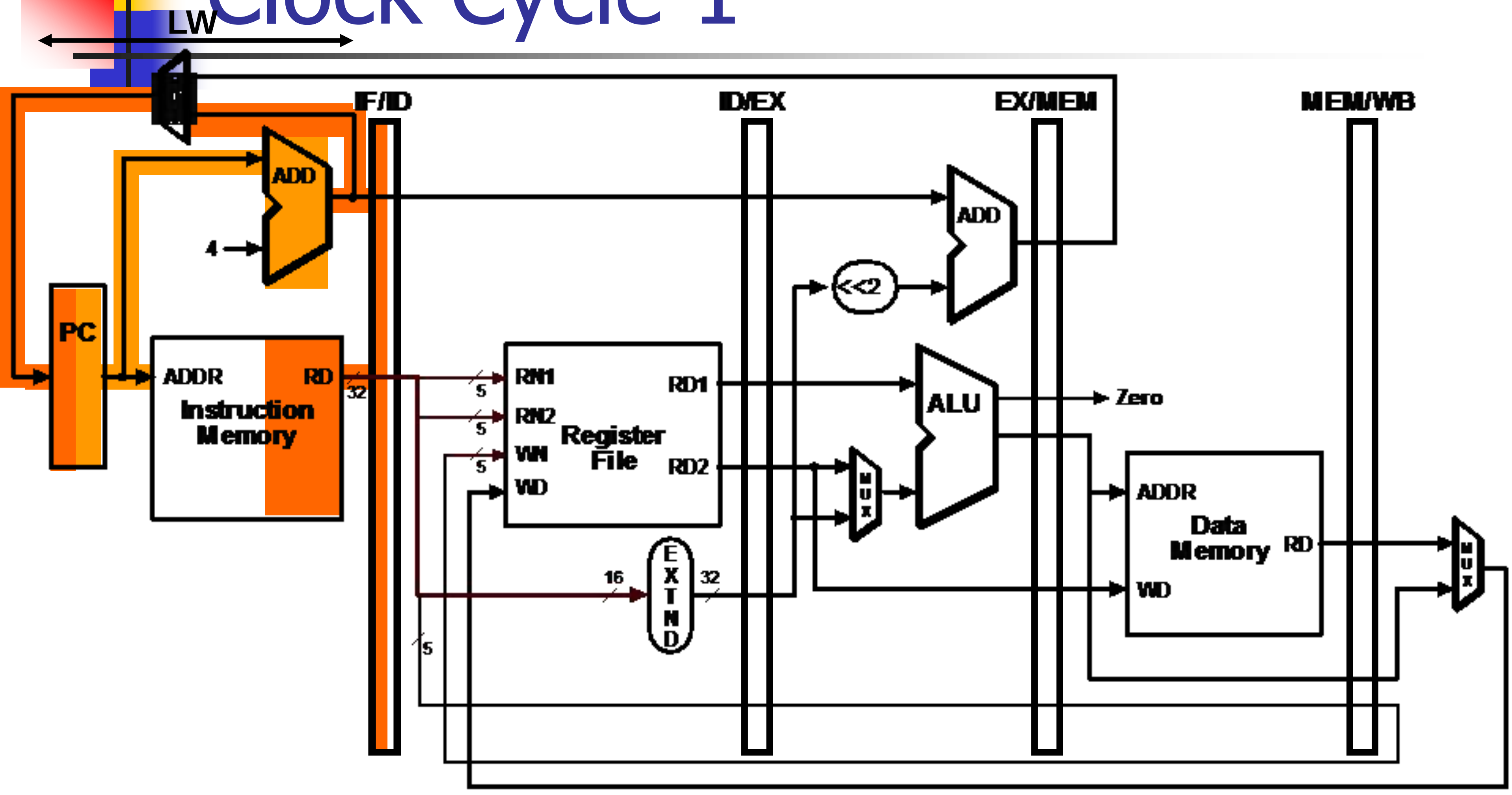
Destination register number is also passed through ID/EX, EX/MEM and MEM/WB registers, which are now wider by 5 bits

Pipelined Example

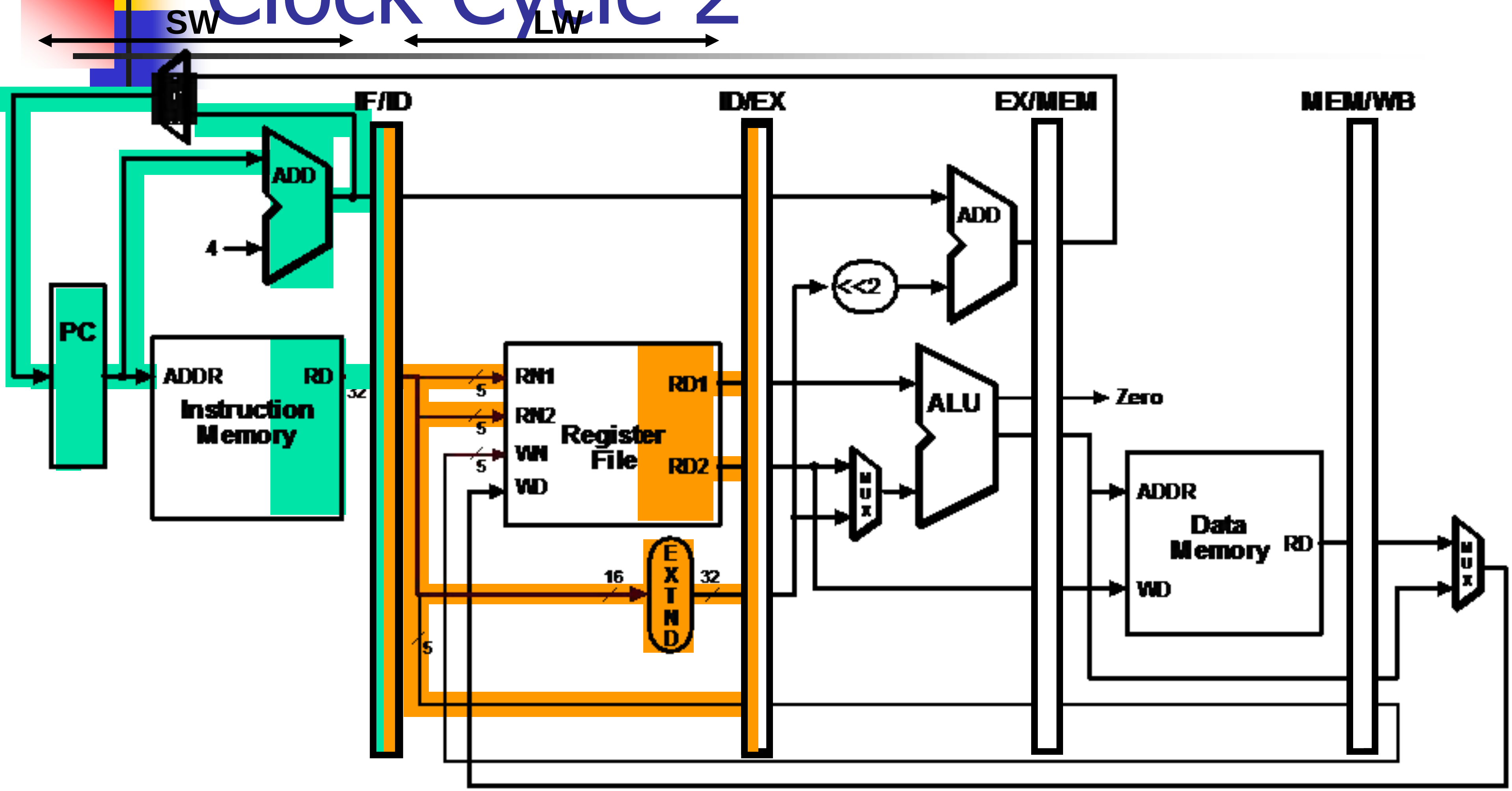
- Consider the following instruction sequence:

```
lw  $t0,  10($t1)
sw  $t3, 20($t4)
add $t5,  $t6,  $t7
sub $t8,  $t9,  $t10
```

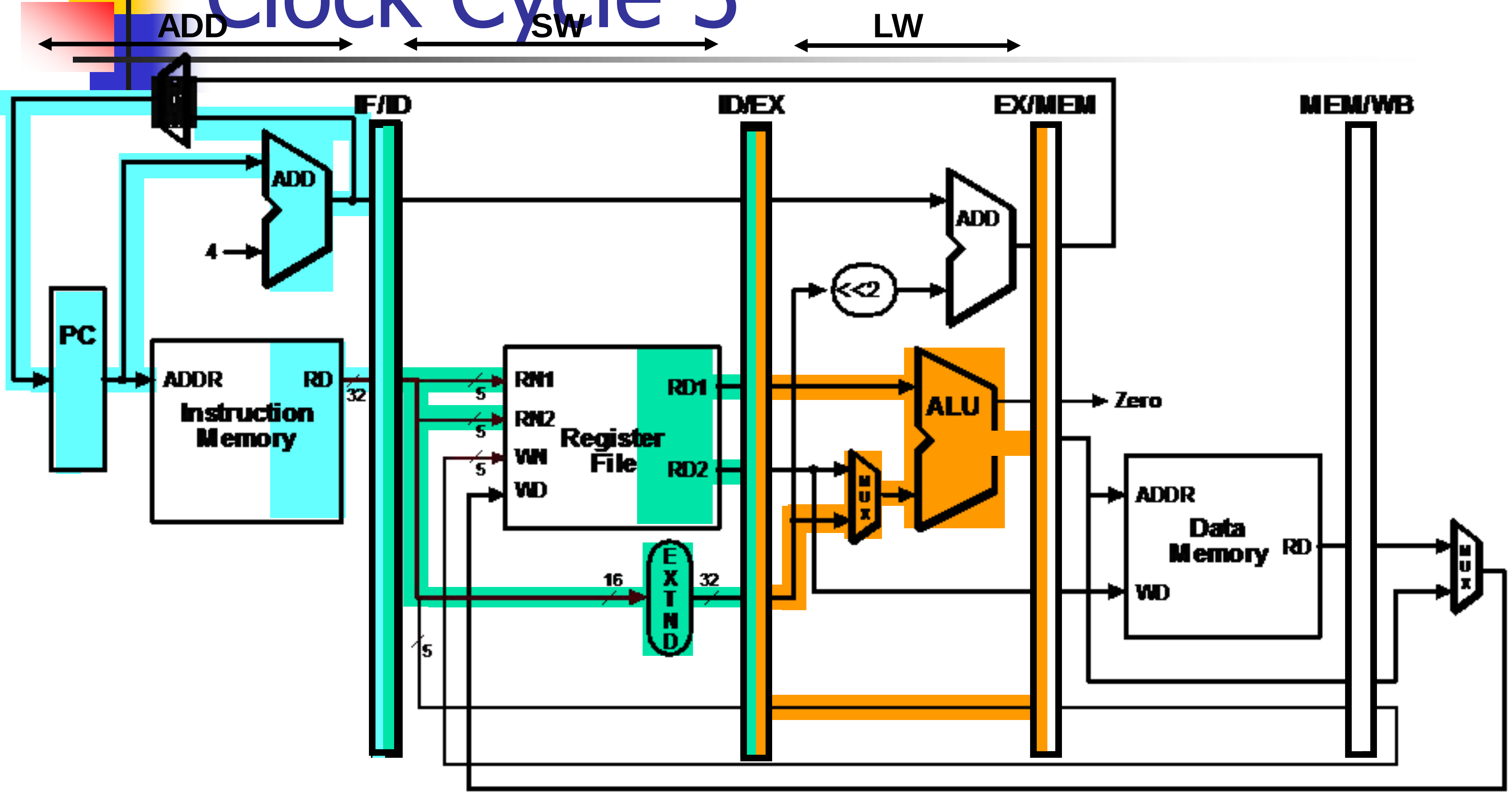
Single-Clock-Cycle Diagram: Clock Cycle 1



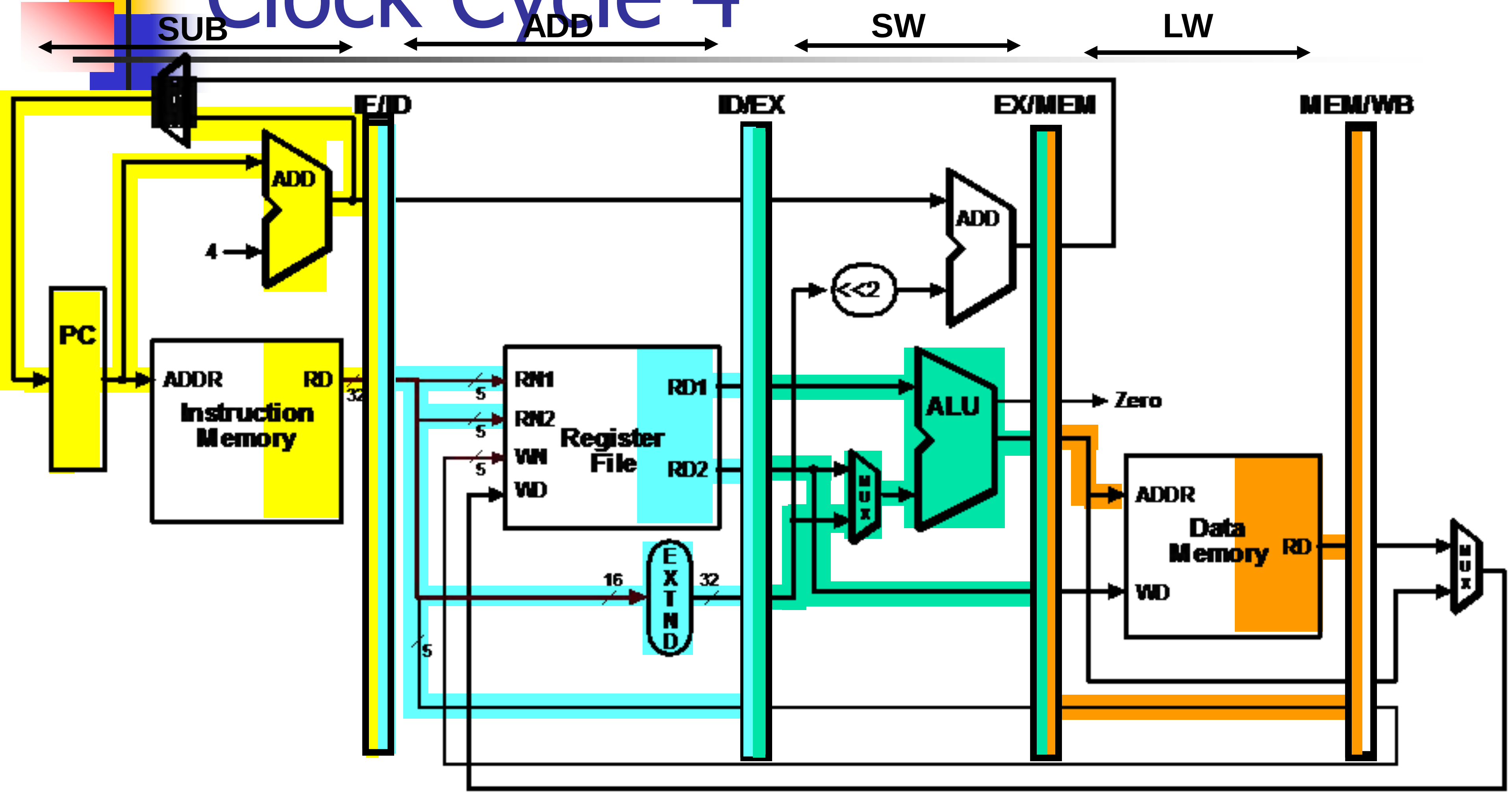
Single-Clock-Cycle Diagram: Clock Cycle 2



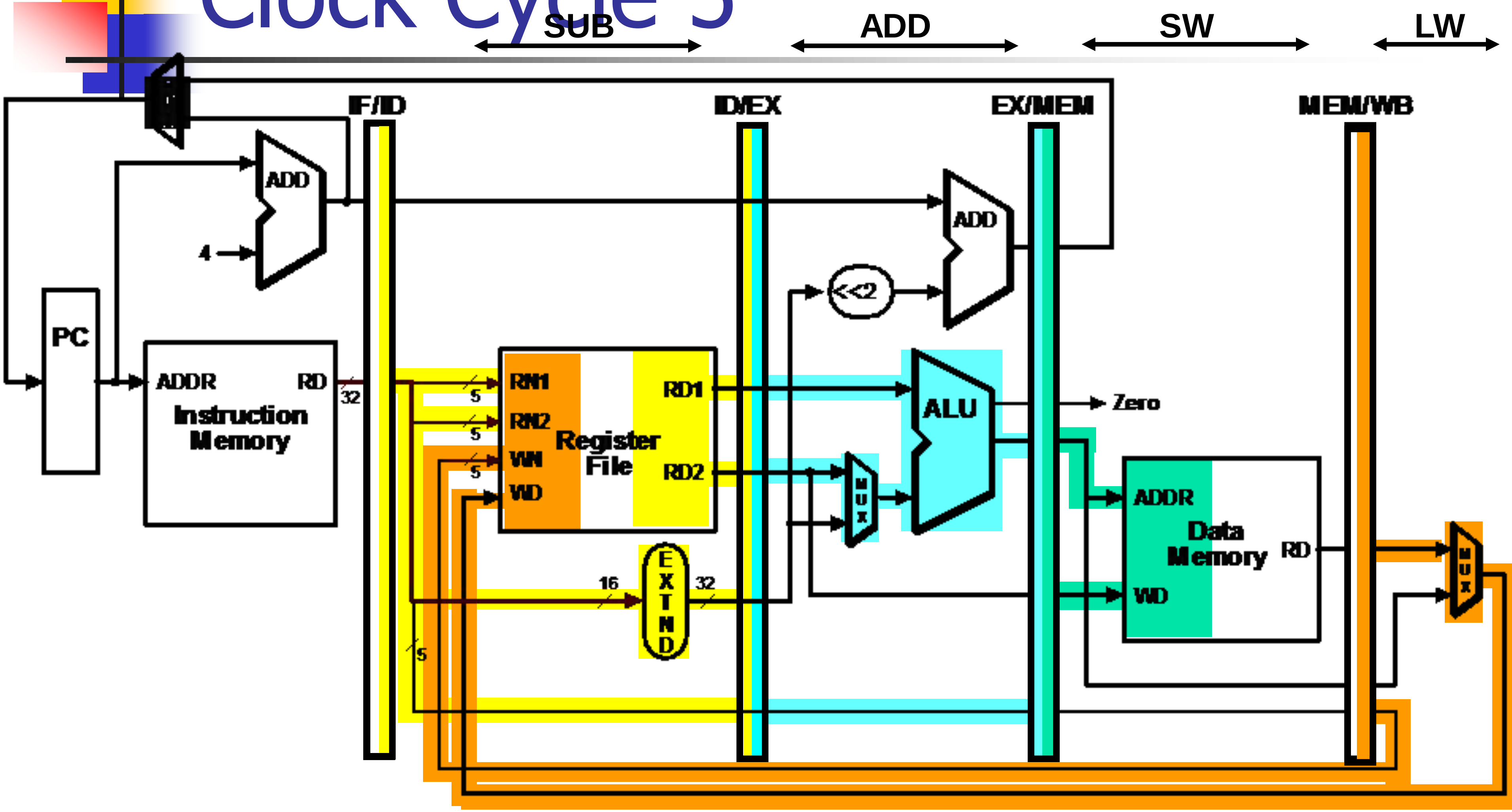
Single-Clock-Cycle Diagram: Clock Cycle 3



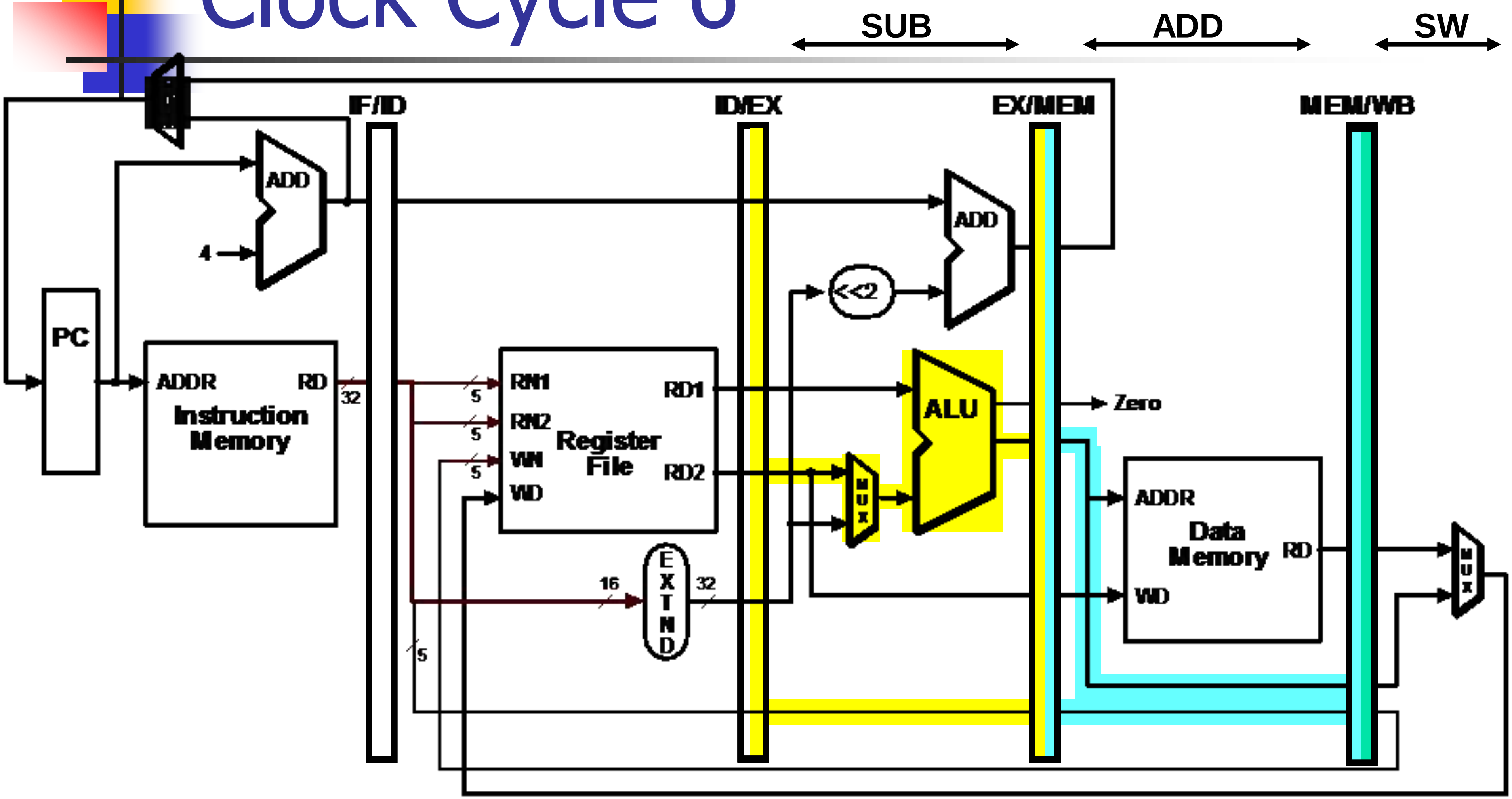
Single-Clock-Cycle Diagram: Clock Cycle 4



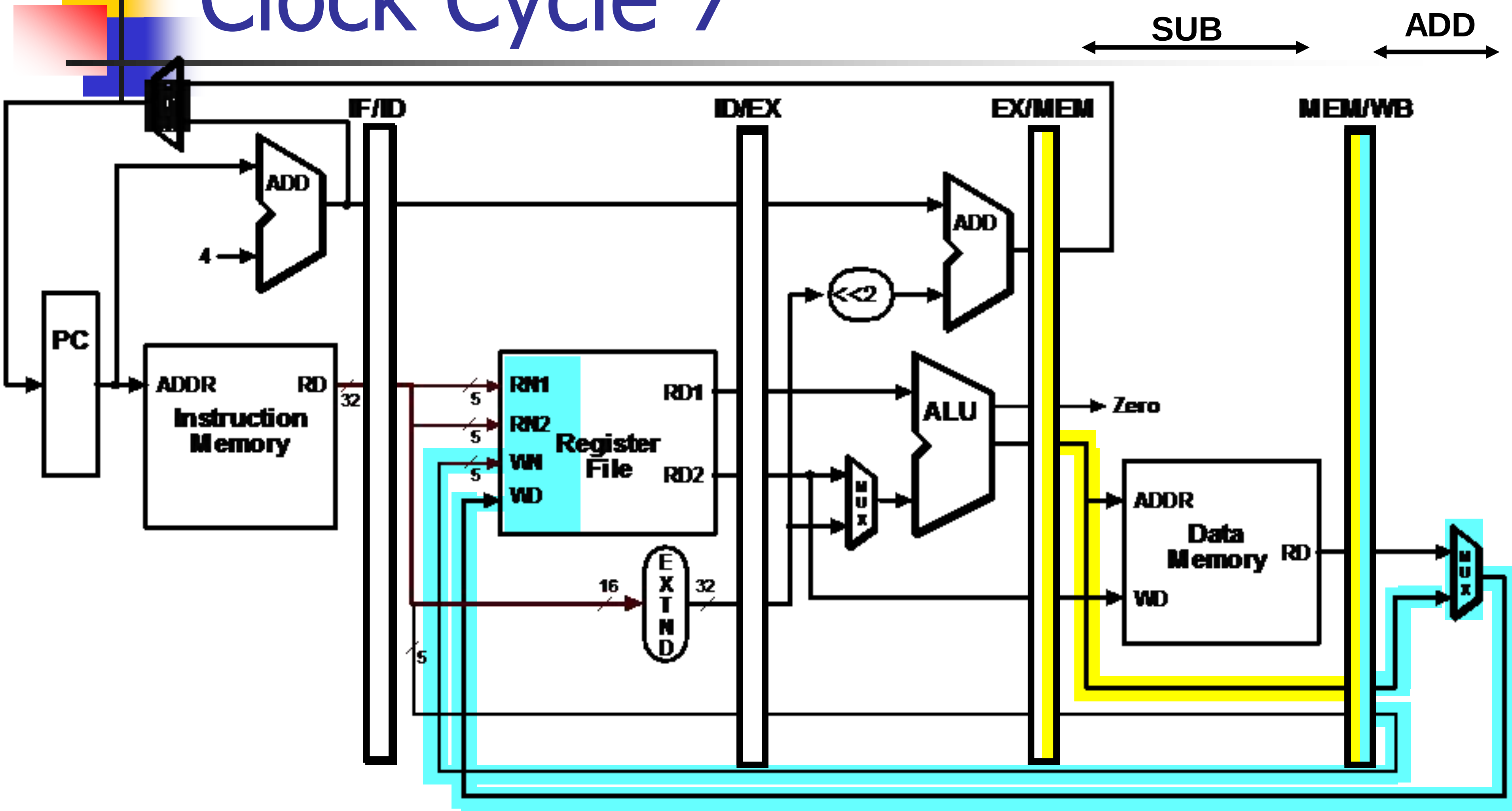
Single-Clock-Cycle Diagram: Clock Cycle 5



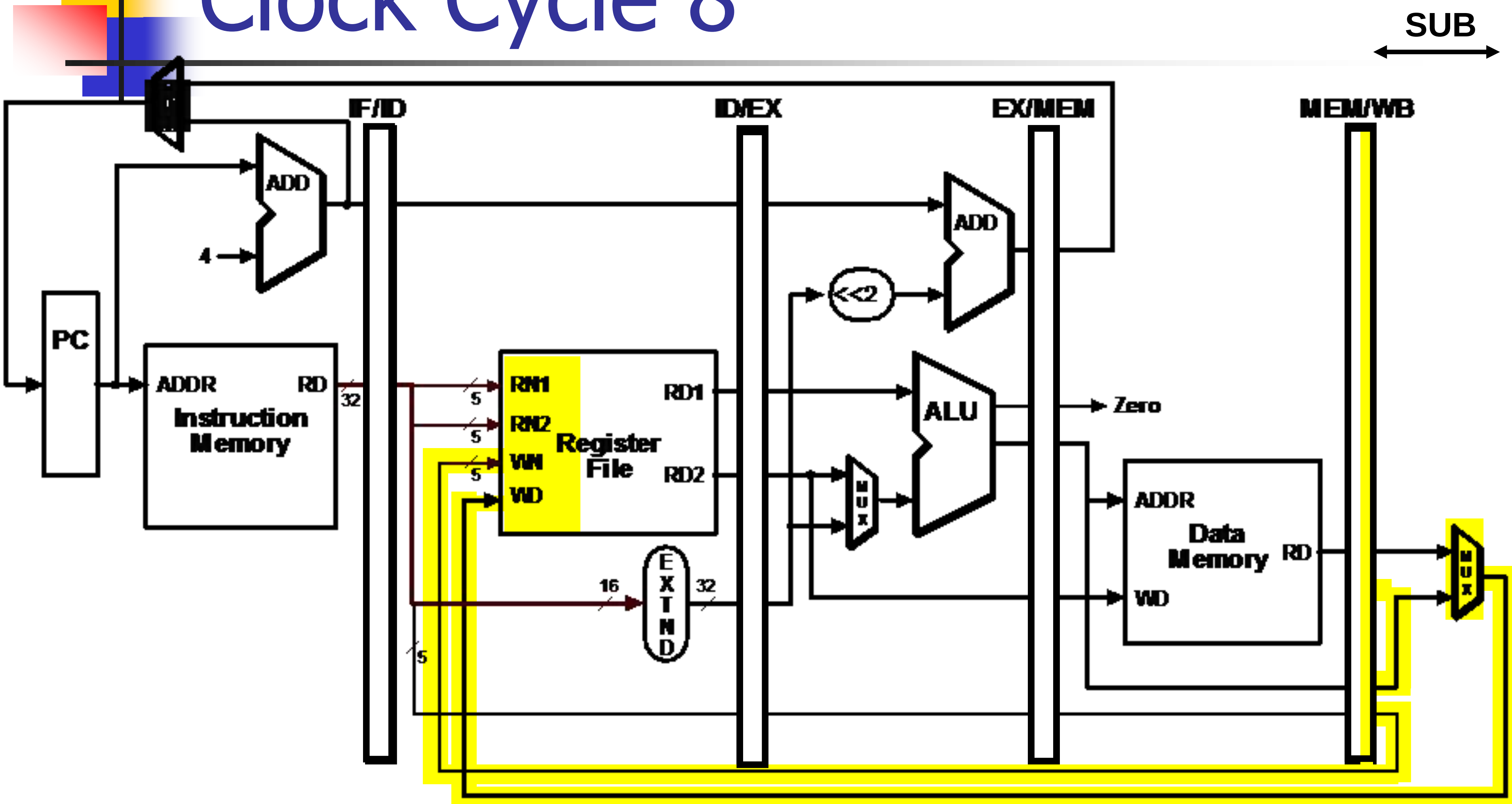
Single-Clock-Cycle Diagram: Clock Cycle 6



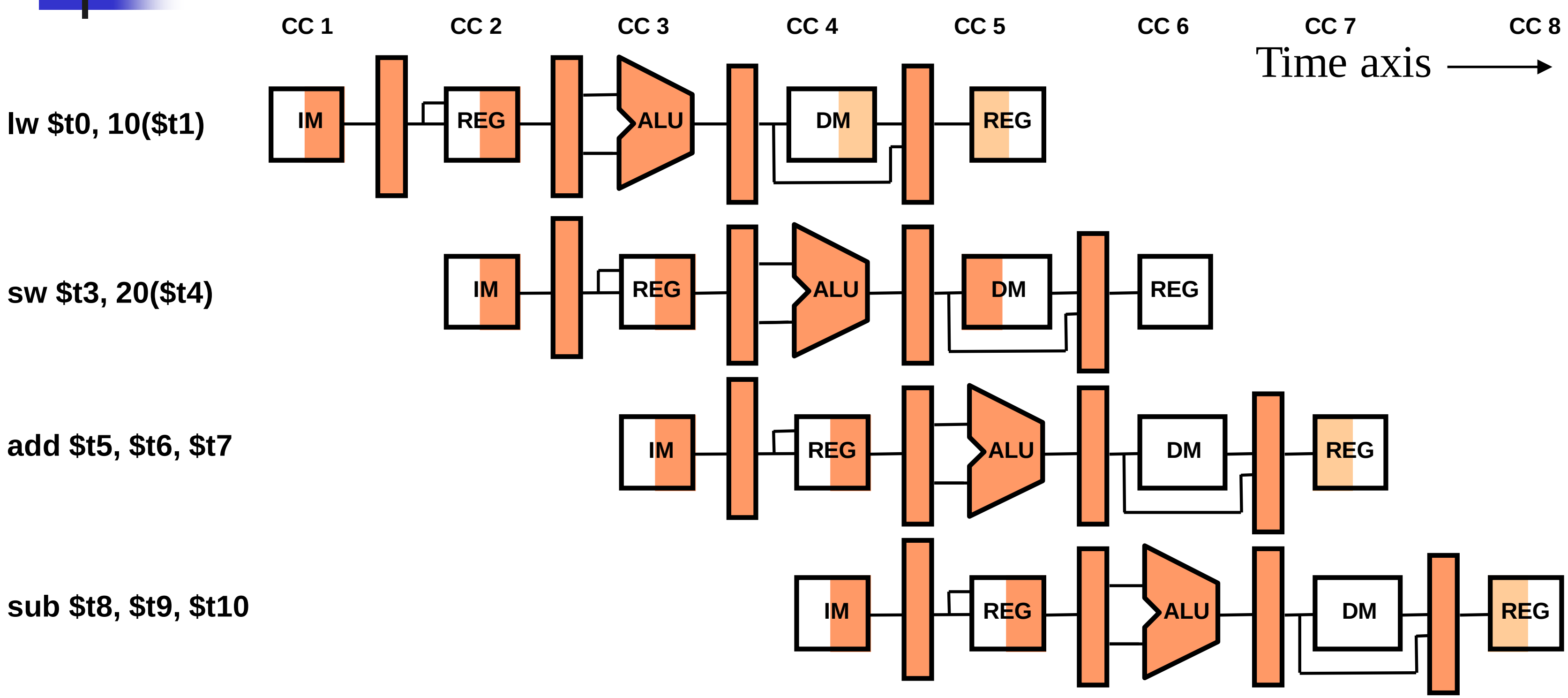
Single-Clock-Cycle Diagram: Clock Cycle 7



Single-Clock-Cycle Diagram: Clock Cycle 8



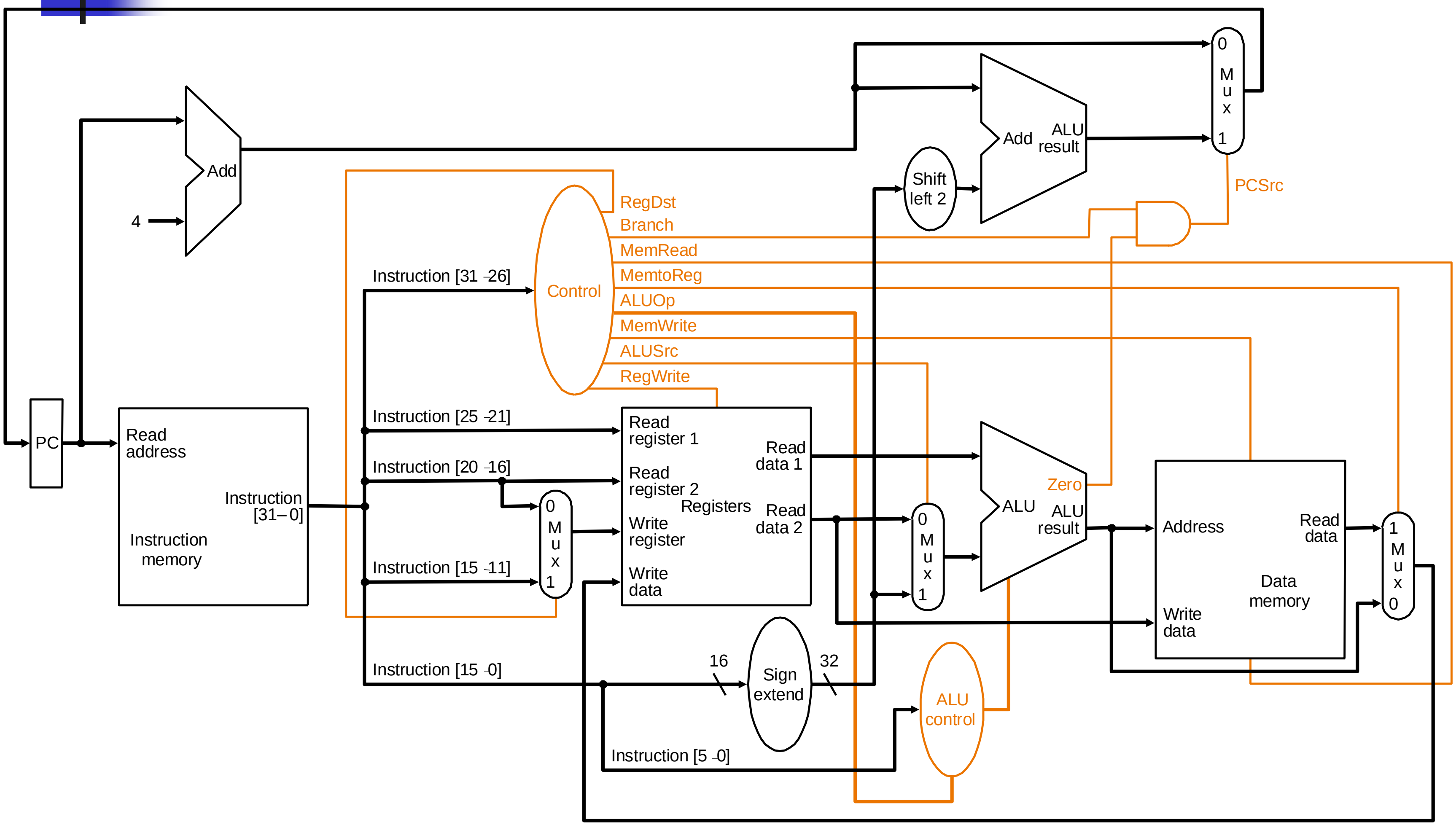
Alternative View – Multiple-Clock-Cycle Diagram



Notes

- One significant difference in the execution of an R-type instruction between multicycle and pipelined implementations:
  - register write-back for the R-type instruction is the 5th (the last write-back) pipeline stage vs. the 4th stage for the multicycle implementation. *Why?*
  - think of *structural hazards* when writing to the register file…
- Worth repeating: the *essential difference* between the pipeline and multicycle implementations is the insertion of pipeline registers to *decouple the 5 stages*
- The CPI of an *ideal pipeline* (no stalls) is 1. *Why?*
- The RaVi Architecture Visualization Project of Dortmund U. has pipeline simulations – see link in our Additional Resources page
- As we develop control for the pipeline keep in mind that the text *does not consider* `jump` – should not be too hard to implement!

Recall Single-Cycle Control – the Datapath



Recall Single-Cycle – ALU Control

Instruction opcode	AluOp	Instruction operation	Funct Field	Desired ALU action	ALU control input
LW	00	load word	xxxxxx	add	010
SW	00	store word	xxxxxx	add	010
Branch eq	01	branch eq	xxxxxx	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	and	000
R-type	10	OR	100101	or	001
R-type	10	set on less	101010	set on less	111

ALUOp		**Funct field**						**Operation**
ALUOp1	**ALUOp0**	**F5**	**F4**	**F3**	**F2**	**F1**	**F0**	
0	0	X	X	X	X	X	X	010
0	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

Truth table for ALU control bits

Recall Single-Cycle – Control Signals

Effect of control bits

Signal Name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20-16)	The register destination number for the Write register comes from the rd field (bits 15-11)
RegWrite	None	The register on the Write register input is written with the value on the Write data input
AlLUSrc	The second ALU operand comes from the second register file output (Read data 2)	The second ALU operand is the sign-extended, lower 16 bits of the instruction
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4	The PC is replaced by the output of the adder that computes the branch target
MemRead	None	Data memory contents designated by the address input are put on the first Read data output
MemWrite	None	Data memory contents designated by the address input are replaced by the value of the Write data input
MemtoReg	The value fed to the register Write data input comes from the ALU	The value fed to the register Write data input comes from the data memory

Deter-mining control bits

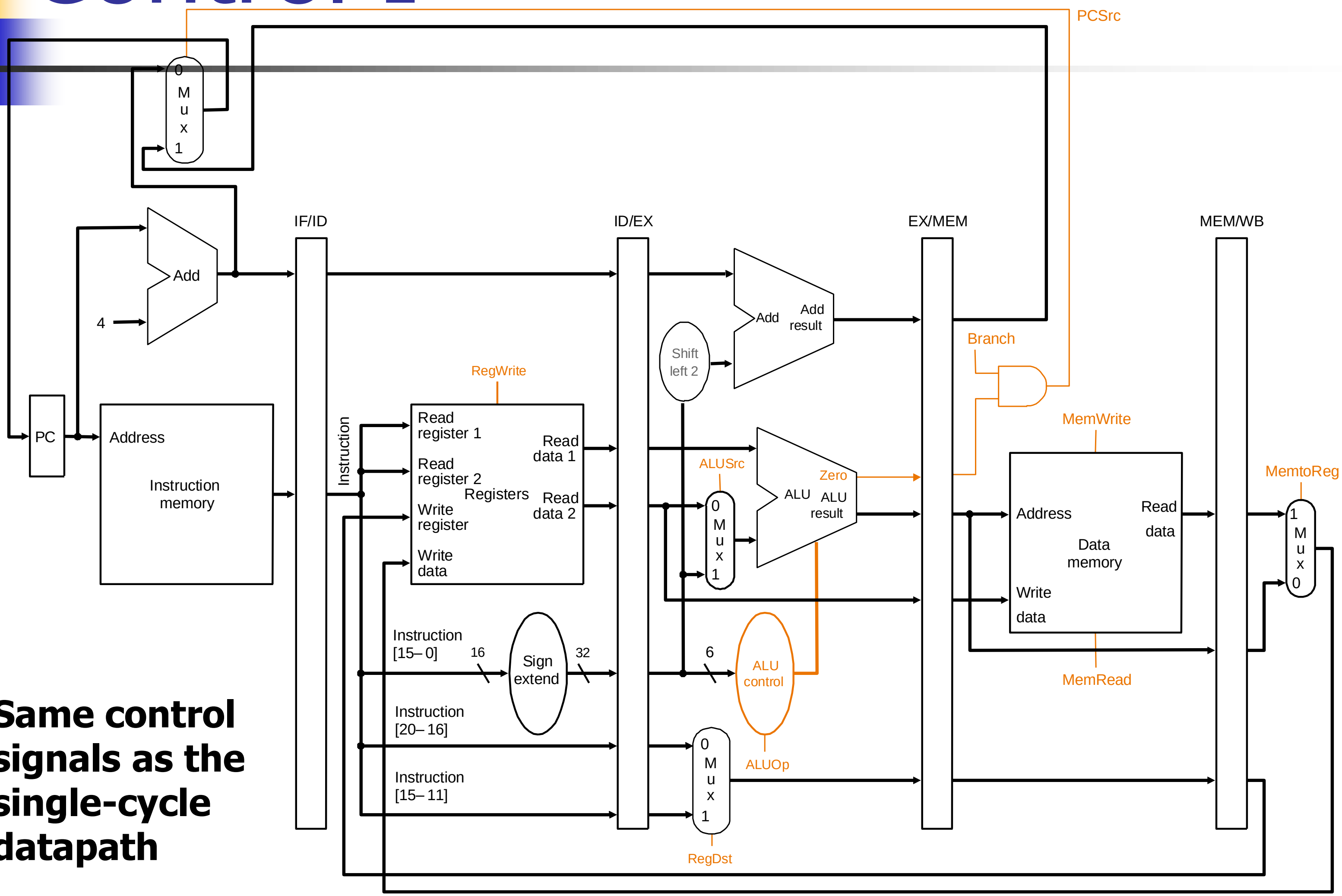
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X	0	0	1	0	0	0
`beq`	X	0	X	0	0	0	1	0	1

Pipeline Control

- Initial design – *motivated by single-cycle datapath control* – use the *same* control signals
- Observe:
  - No separate write signal for the PC as it is written every cycle
  - No separate write signals for the pipeline registers as they are written every cycle
  - *No separate read signal for instruction memory* as it is read every clock cycle
  - *No separate read signal for register file* as it is read every clock cycle
- Need to *set control signals during each pipeline stage*
- Since control signals are associated with components active during a single pipeline stage, can *group control lines into five groups according to pipeline stage*

Will be modified by hazard detection unit!!

Pipelined Datapath with Control I



Same control signals as the single-cycle datapath

Pipeline Control Signals

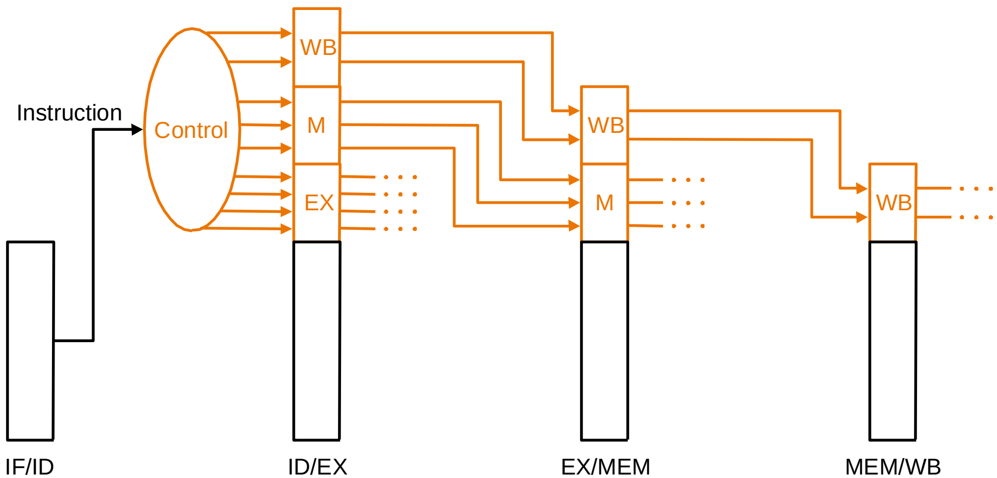
- There are five stages in the pipeline
  - *instruction fetch / PC increment*
  - *instruction decode / register fetch*
  - *execution / address calculation*
  - *memory access*
  - *write back*

Nothing to control as instruction memory read and PC write are always enabled

Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
`lw`	0	0	0	1	0	1	0	1	1
`sw`	X	0	0	1	0	0	1	0	X
`beq`	X	0	1	0	1	0	0	0	X

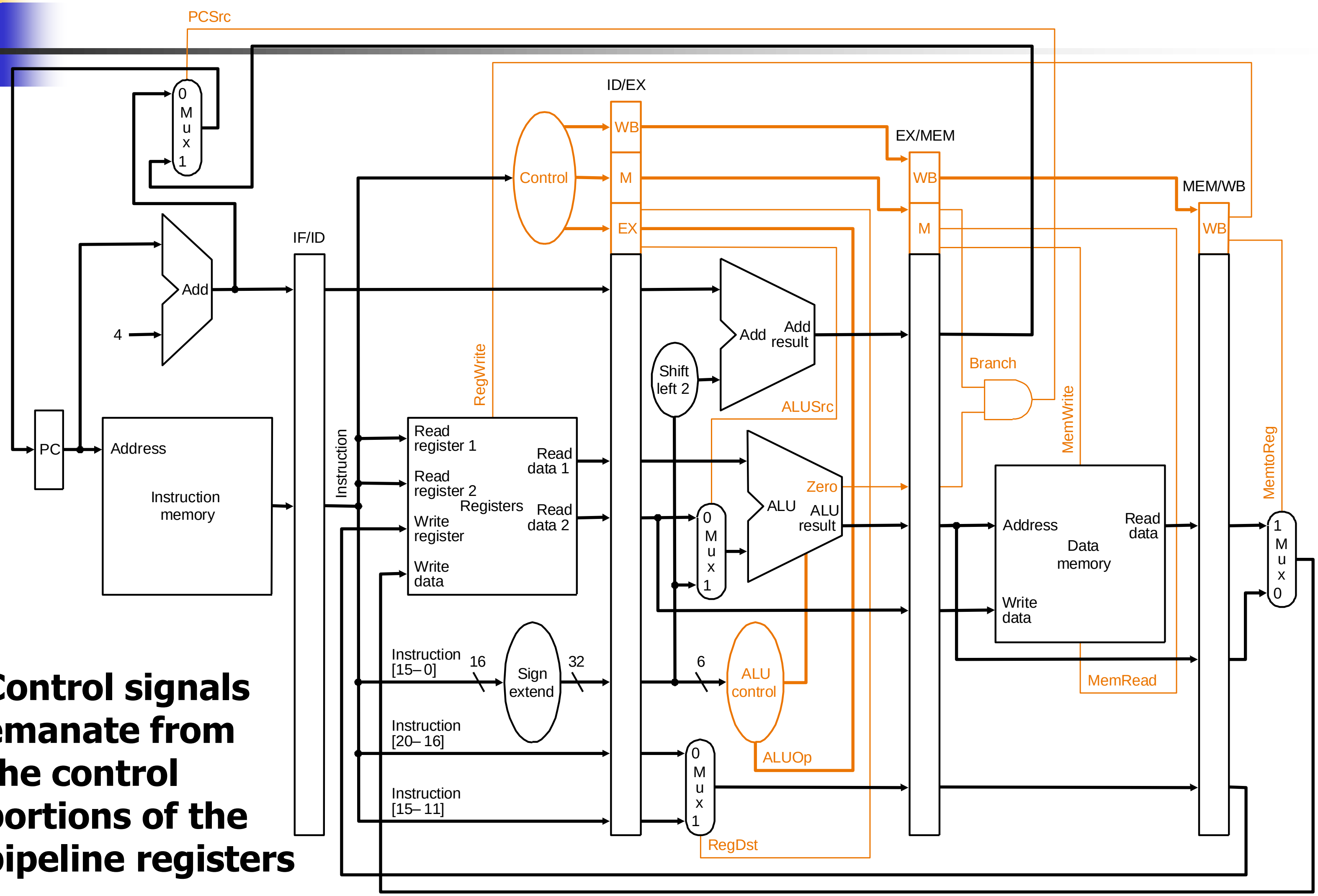
Pipeline Control Implementation

- *Pass control signals along just like the data* – extend each pipeline register to hold needed control bits for succeeding stages



- *Note*: The 6-bit *funct field* of the instruction required in the EX stage to generate ALU control can be retrieved as the 6 least significant bits of the immediate field which is sign-extended and passed from the IF/ID register to the ID/EX register

Pipelined Datapath with Control II

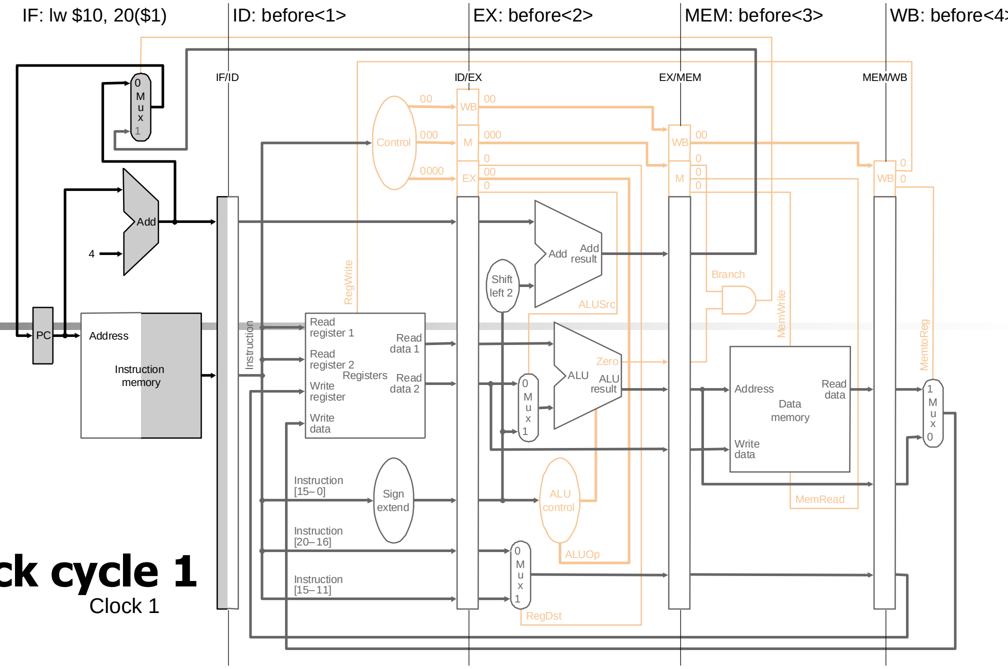


Control signals emanate from the control portions of the pipeline registers

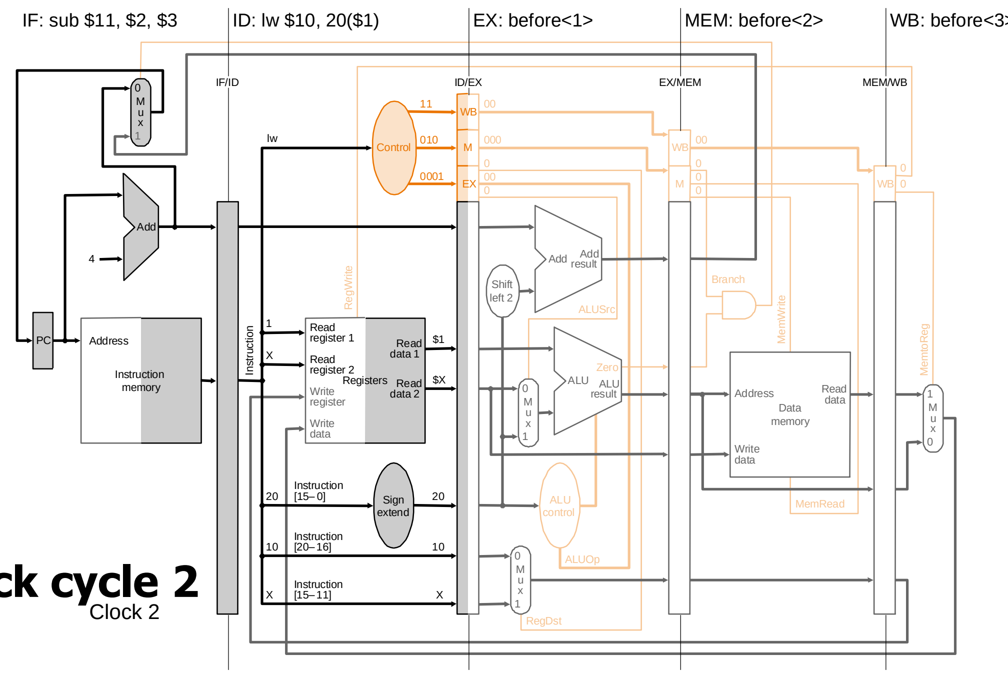
Pipelined Execution and Control

- Instruction sequence:

```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $7
or  $13, $6, $7
add $14, $8, $9
```



Clock cycle 1



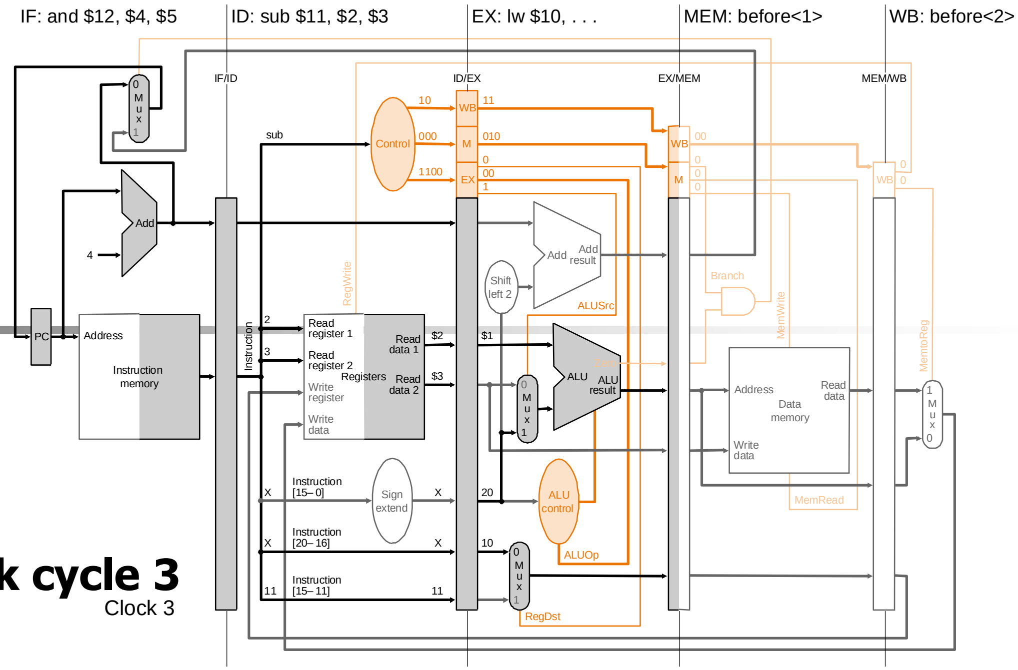
Clock cycle 2

Label "before<i>" means i th instruction before `lw`

Pipelined Execution and Control

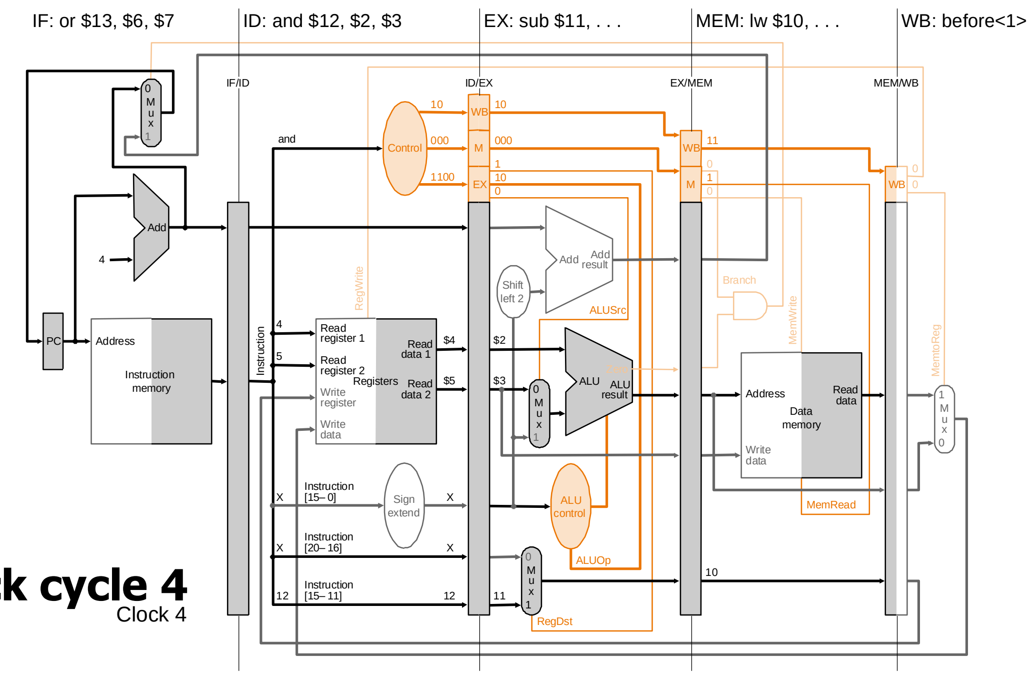
- Instruction sequence:

```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $7
or  $13, $6, $7
add $14, $8, $9
```



Clock cycle 3

Clock 3



Clock cycle 4

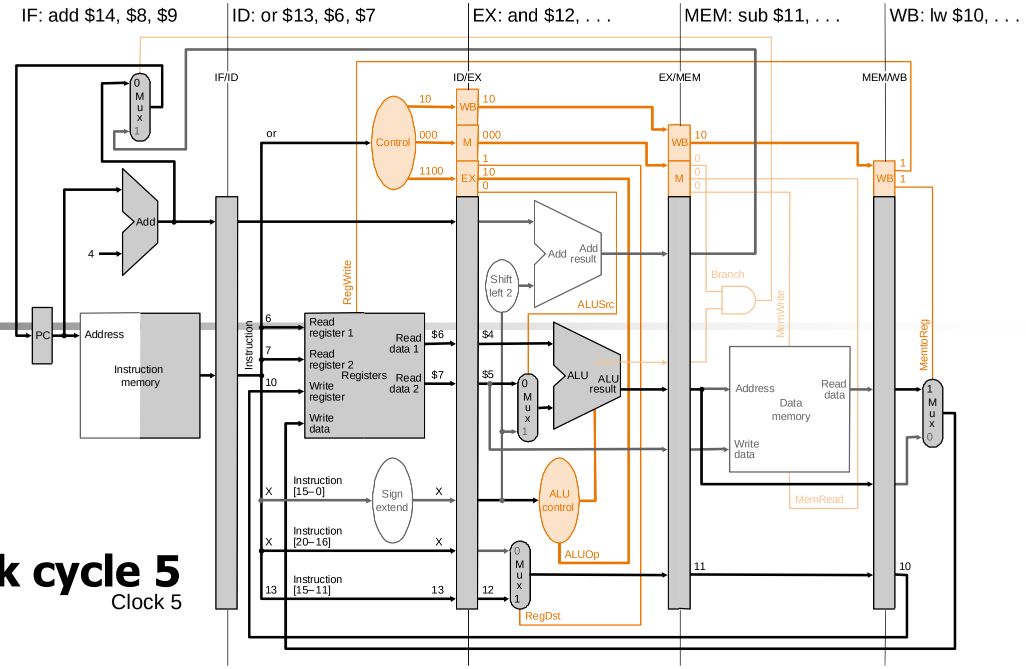
Clock 4

Pipelined Execution and Control

- Instruction sequence:

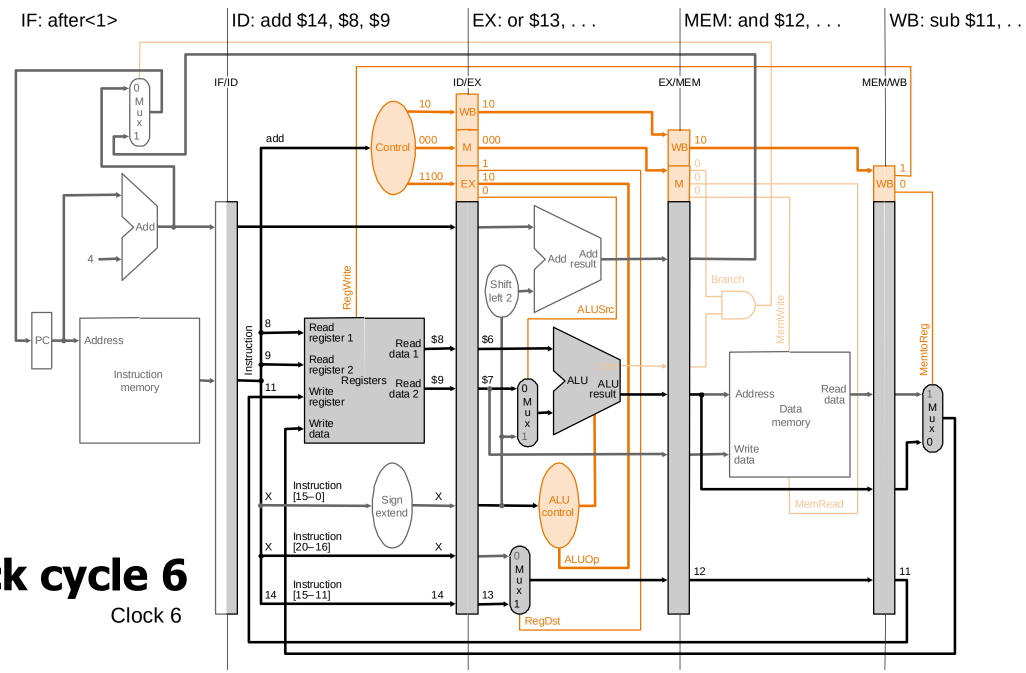
```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $7
or  $13, $6, $7
add $14, $8, $9
```

Label “after<i>” means i th instruction after `add`



Clock cycle 5

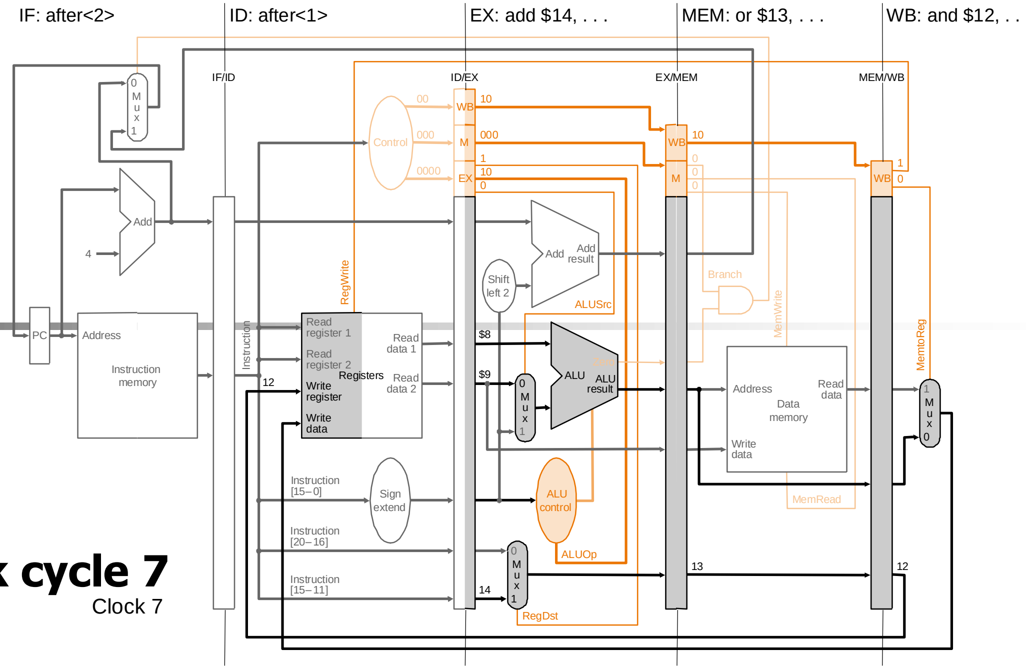
Clock 5



Clock cycle 6

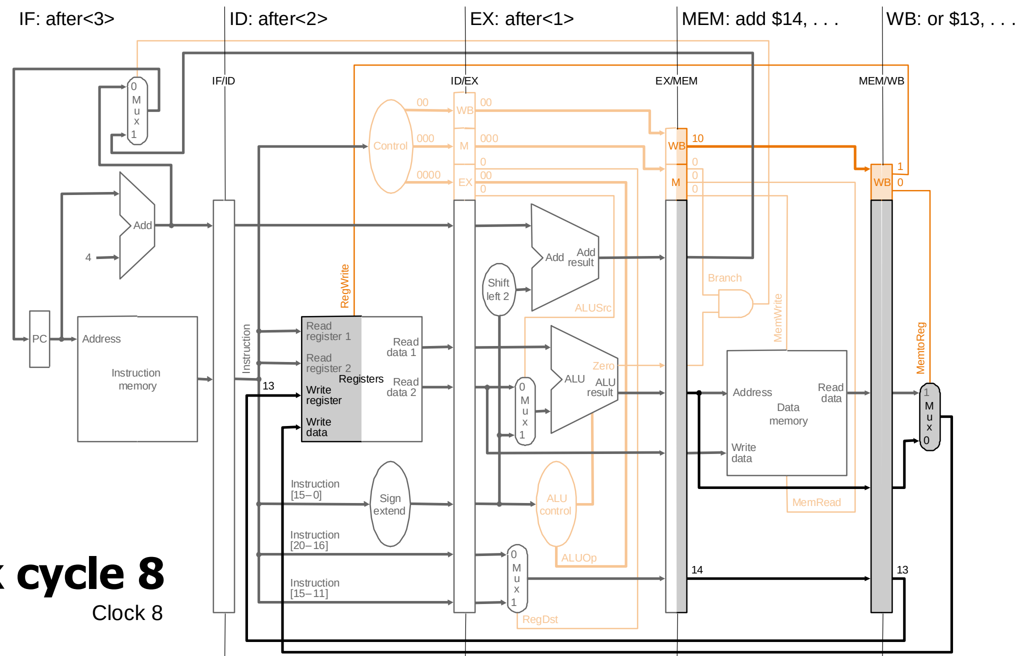
Clock 6

Pipelined Execution and Control



Clock cycle 7

Clock 7



Clock cycle 8

Clock 8

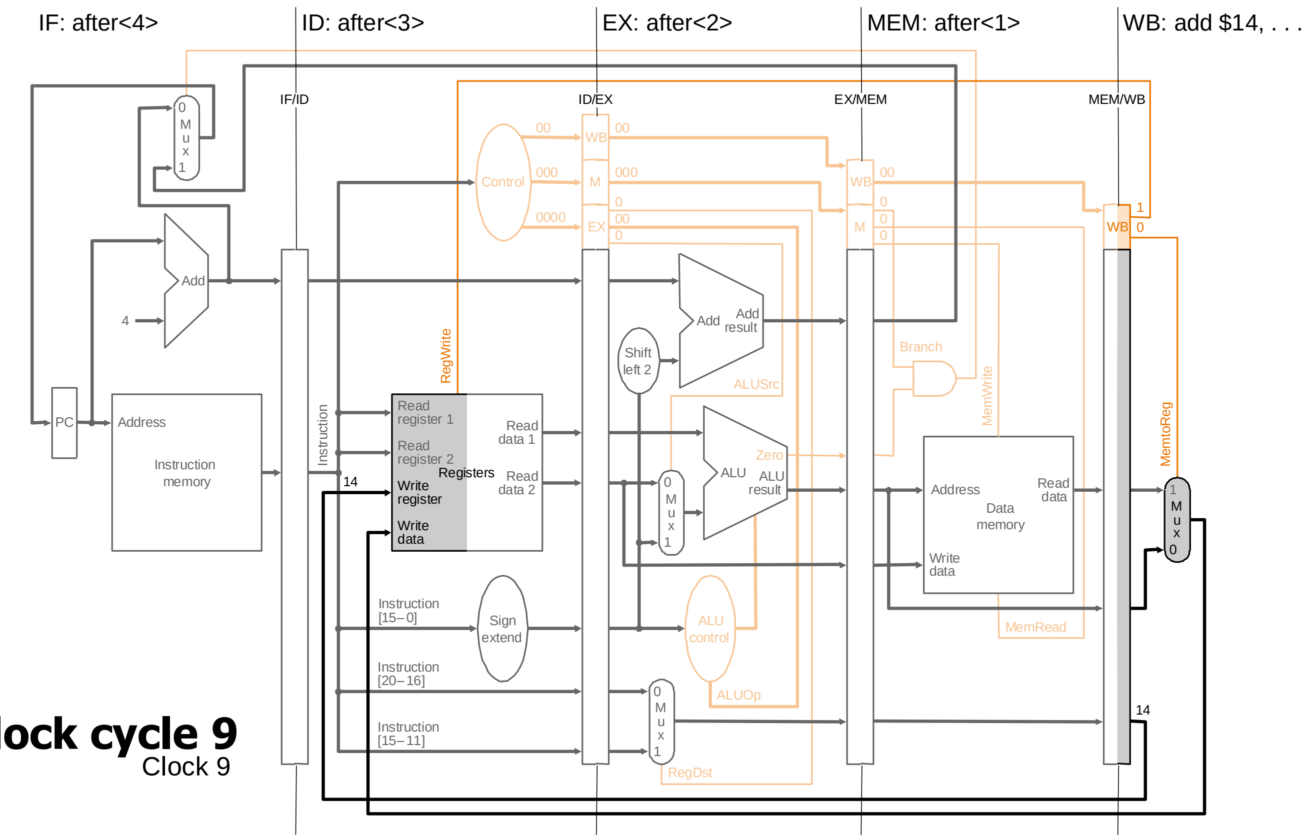
- Instruction sequence:

```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $7
or  $13, $6, $7
add $14, $8, $9
```

Pipelined Execution and Control

- Instruction sequence:

```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $7
or  $13, $6, $7
add $14, $8, $9
```



Clock cycle 9

Clock 9

Revisiting Hazards

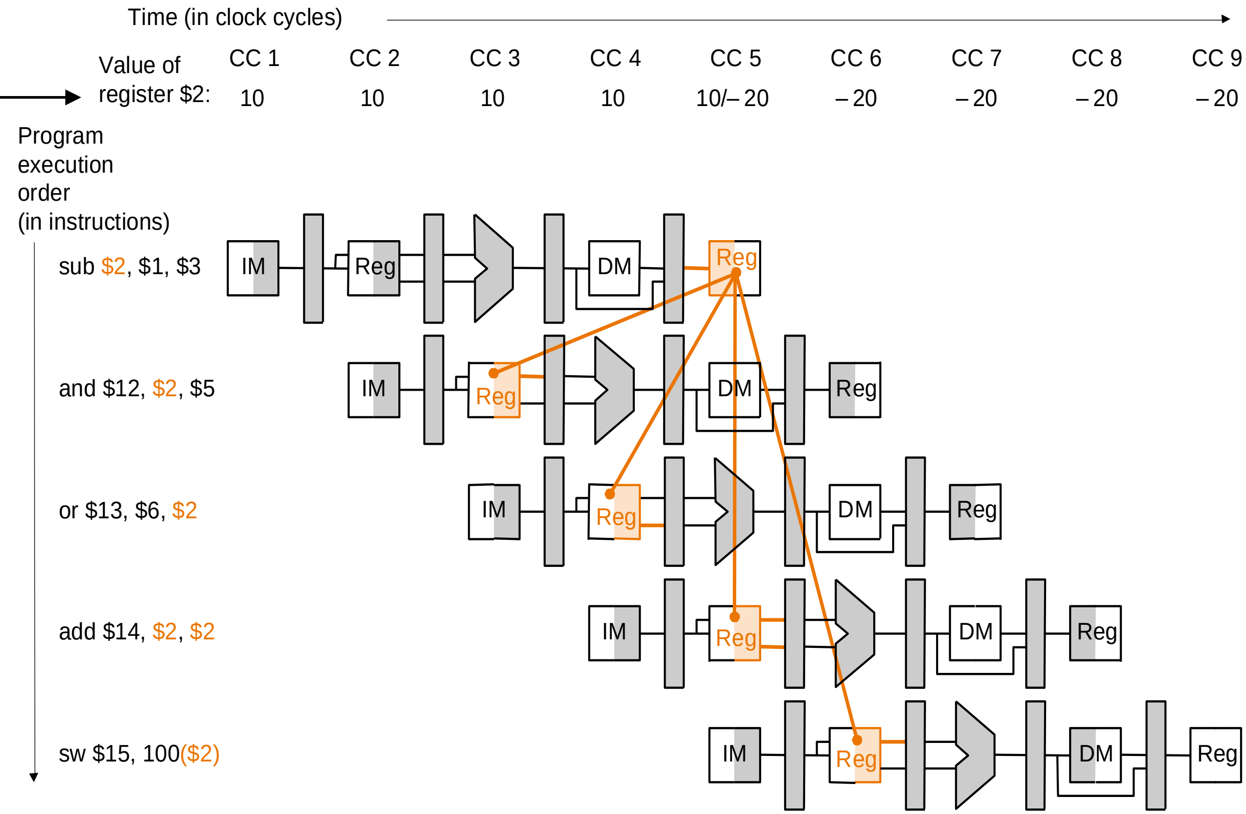
- So far our datapath and control have ignored hazards
- We shall revisit *data hazards* and *control hazards* and enhance our datapath and control to handle them in *hardware*…

Data Hazards and Forwarding

- Problem with starting an instruction before previous are finished:
  - data dependencies that <u>go backward in time</u> – called *data hazards*

$2 = 10 before `sub`;
$2 = -20 after `sub`

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



Software Solution

- Have compiler guarantee *never* any data hazards!
  - by *rearranging instructions to insert independent instructions between instructions* that would otherwise have a data hazard between them,
  - or, if such rearrangement is not possible, *insert* `nop`s

```
sub   $2,  $1, $3
lw    $10, 40($3)
slt   $5, $6, $7
and   $12, $2, $5
or    $13, $6, $2
add   $14, $2, $2
sw    $15, 100($2)
```

or

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

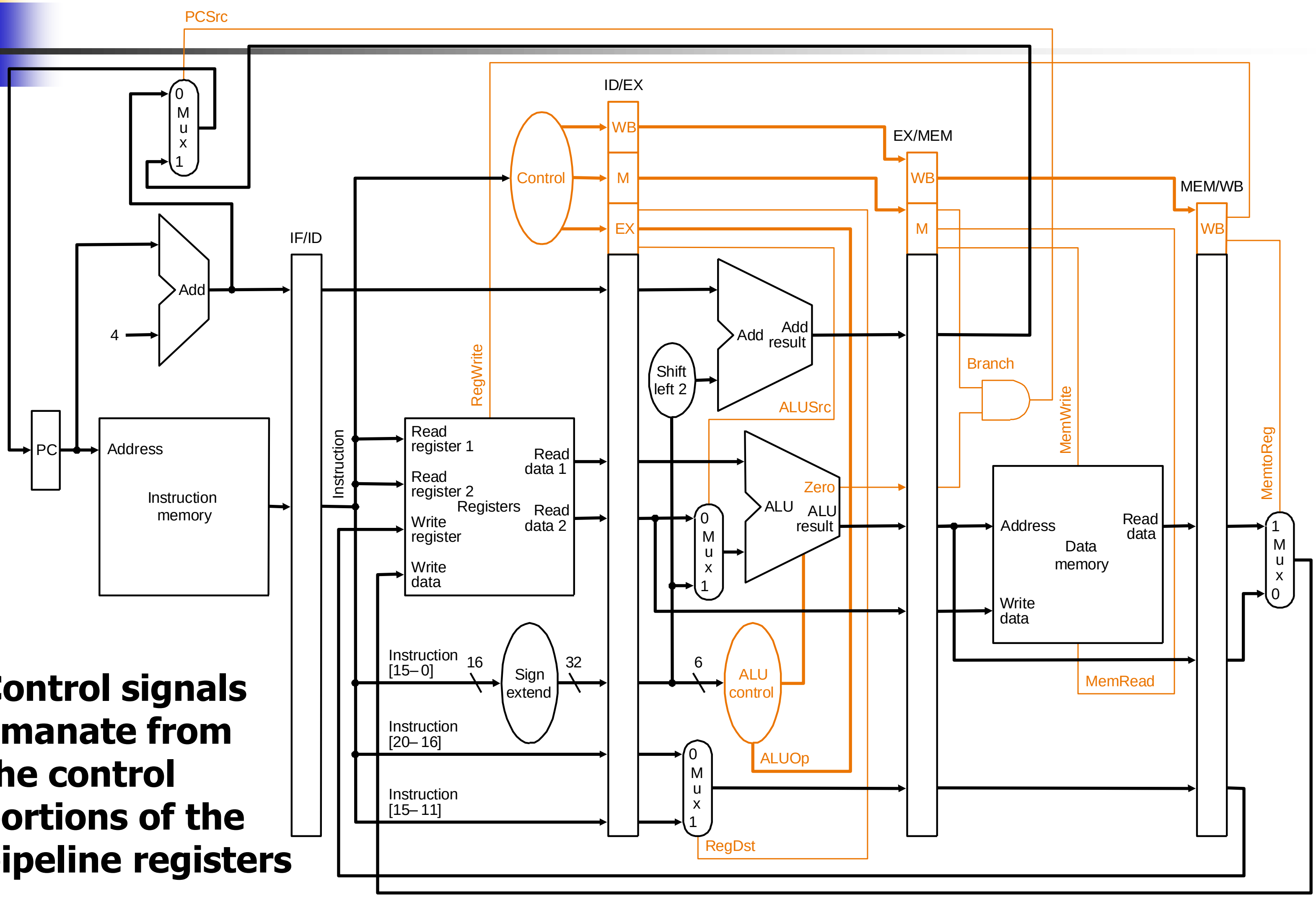
- Such compiler solutions may not always be possible, and `nop`s slow the machine down

MIPS: `nop` = "no operation" = 00…0 (32bits) = `sll $0, $0, 0`

Hardware Solution: Forwarding

- Idea: *use intermediate data*, do not wait for result to be finally written to the destination register. Two steps:
  1. *Detect* data hazard
  2. *Forward* intermediate data to resolve hazard

Pipelined Datapath with Control II (as before)



Control signals emanate from the control portions of the pipeline registers

Hazard Detection

- Hazard conditions:

1a. EX/MEM.RegisterRd = ID/EX.RegisterRs

1b. EX/MEM.RegisterRd = ID/EX.RegisterRt

2a. MEM/WB.RegisterRd = ID/EX.RegisterRs

2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

  - Eg., in the earlier example, first hazard between `sub $2, $1, $3` and `and $12, $2, $5` is detected when the `and` is in EX stage and the `sub` is in MEM stage because
    - `EX/MEM.RegisterRd = ID/EX.RegisterRs = $2 (1a)`

- Whether to forward also depends on:
  - *if the later instruction is going to write a register* – if *not*, no need to forward, even if there is register number match as in conditions above
  - *if the destination register of the later instruction is $0* – in which case there is no need to forward value ($0 is always 0 and never overwritten)

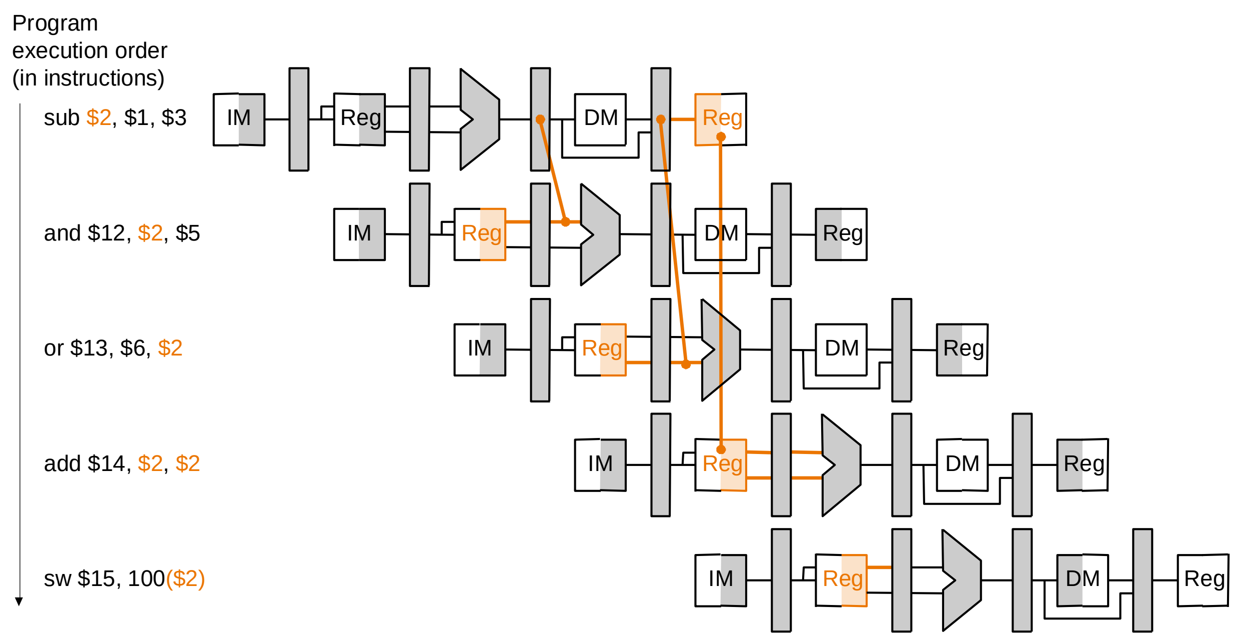
Data Forwarding

- Plan:
  - *allow inputs to the ALU not just from ID/EX, but also later pipeline registers*, and
  - *use multiplexors and control signals to choose appropriate inputs* to ALU

Time (in clock cycles) →

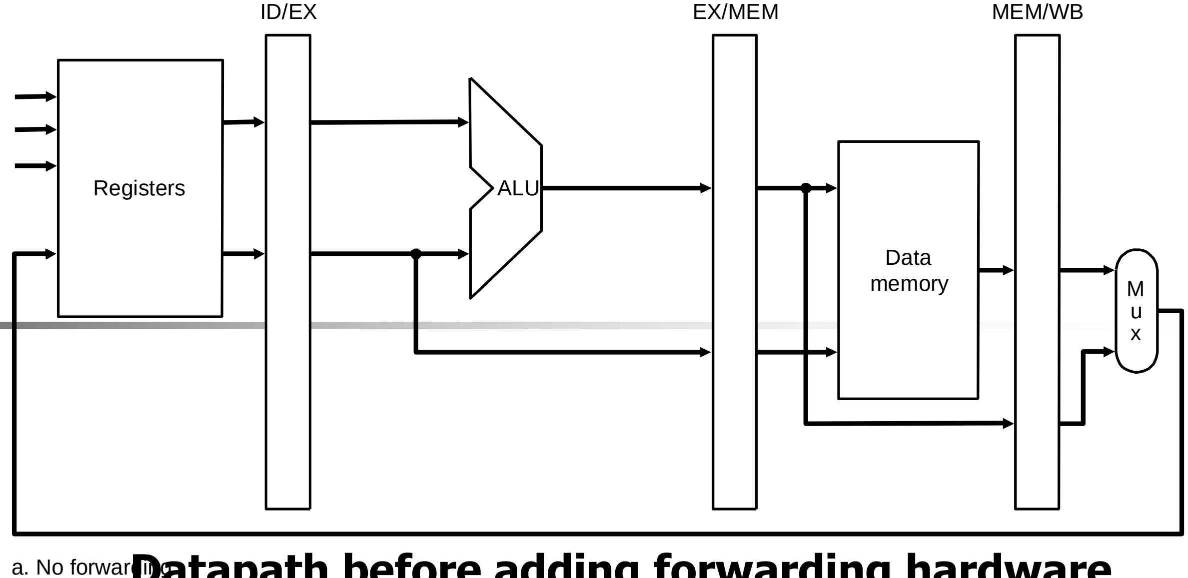
	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register $2 :	10	10	10	10	10/– 20	– 20	– 20	– 20	– 20
Value of EX/MEM :	X	X	X	– 20	X	X	X	X	X
Value of MEM/WB :	X	X	X	X	– 20	X	X	X	X

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

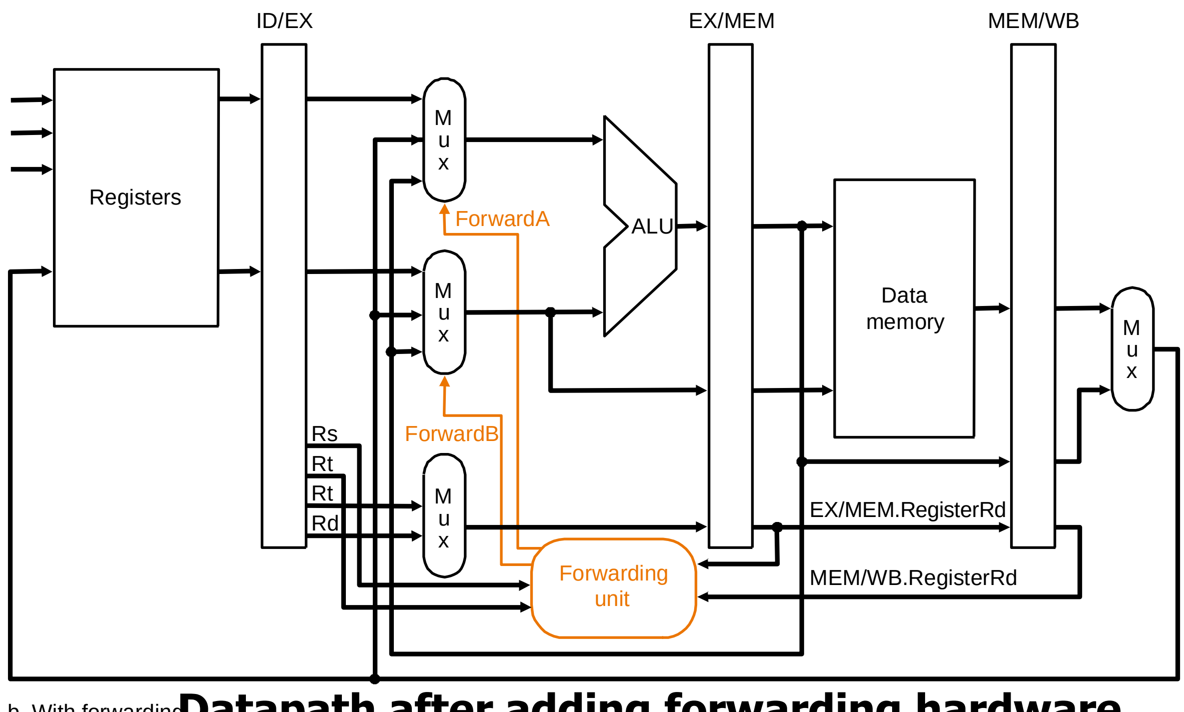


Dependencies between pipelines move forward in time

Forwarding Hardware



Datapath before adding forwarding hardware



Datapath after adding forwarding hardware

Forwarding Hardware: Multiplexor Control

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from prior ALU result
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result
ForwardB = 00	ID/EX	The second ALU operand comes from the register file
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from prior ALU result
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result

Depending on the selection in the rightmost multiplexor (see datapath with control diagram)

Data Hazard: Detection and Forwarding

- Forwarding unit determines multiplexor control according to the following rules:

1. **EX hazard**

if (EX/MEM.RegWrite // if there is a write…
and (EX/MEM.RegisterRd $\neq$ 0) // to a non-\$0 register…
and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) // which matches, then…
ForwardA = 10

if (EX/MEM.RegWrite // if there is a write…
and (EX/MEM.RegisterRd $\neq$ 0) // to a non-\$0 register…
and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) // which matches, then…
ForwardB = 10

Data Hazard: Detection and Forwarding

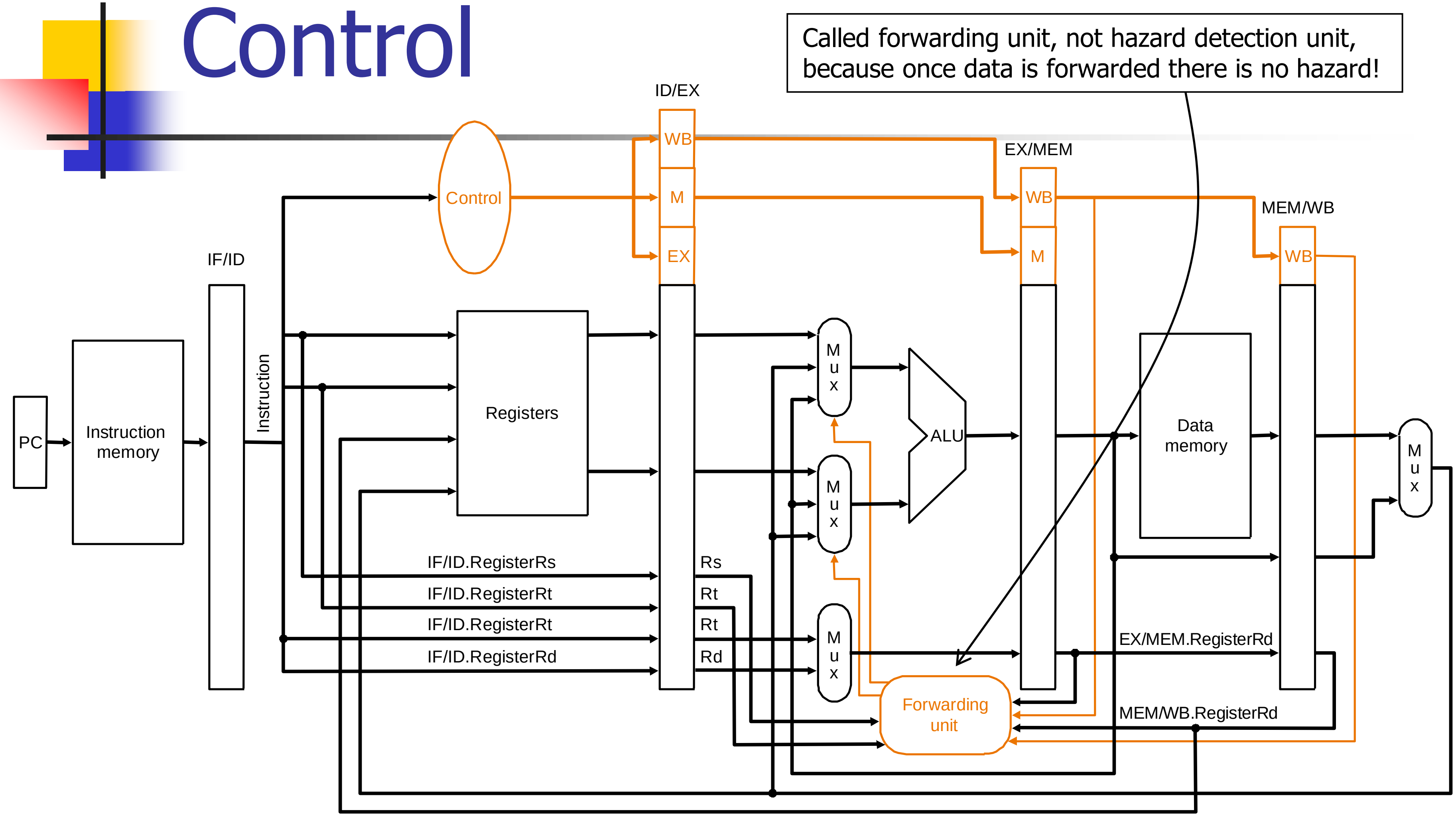
2. **MEM hazard**

```
if (       MEM/WB.RegWrite                          // if there is a write…
   and ( MEM/WB.RegisterRd ≠ 0 )                    // to a non-$0 register…
   and ( EX/MEM.RegisterRd ≠ ID/EX.RegisterRs )     // and not already a register match
                                                    // with earlier pipeline register…
   and ( MEM/WB.RegisterRd = ID/EX.RegisterRs ) )   // but match with later pipeline
                                                       register, then…
ForwardA = 01

if (       MEM/WB.RegWrite                          // if there is a write…
   and ( MEM/WB.RegisterRd ≠ 0 )                    // to a non-$0 register…
   and ( EX/MEM.RegisterRd ≠ ID/EX.RegisterRt )     // and not already a register match
                                                    // with earlier pipeline register…
   and ( MEM/WB.RegisterRd = ID/EX.RegisterRt ) )   // but match with later pipeline
                                                       register, then…
ForwardB = 01
```

This check is necessary, e.g., for sequences such as add $1, $1, $2; add $1, $1, $3; add $1, $1, $4; (array summing?), where an earlier pipeline (EX/MEM) register has more recent data

Forwarding Hardware with Control



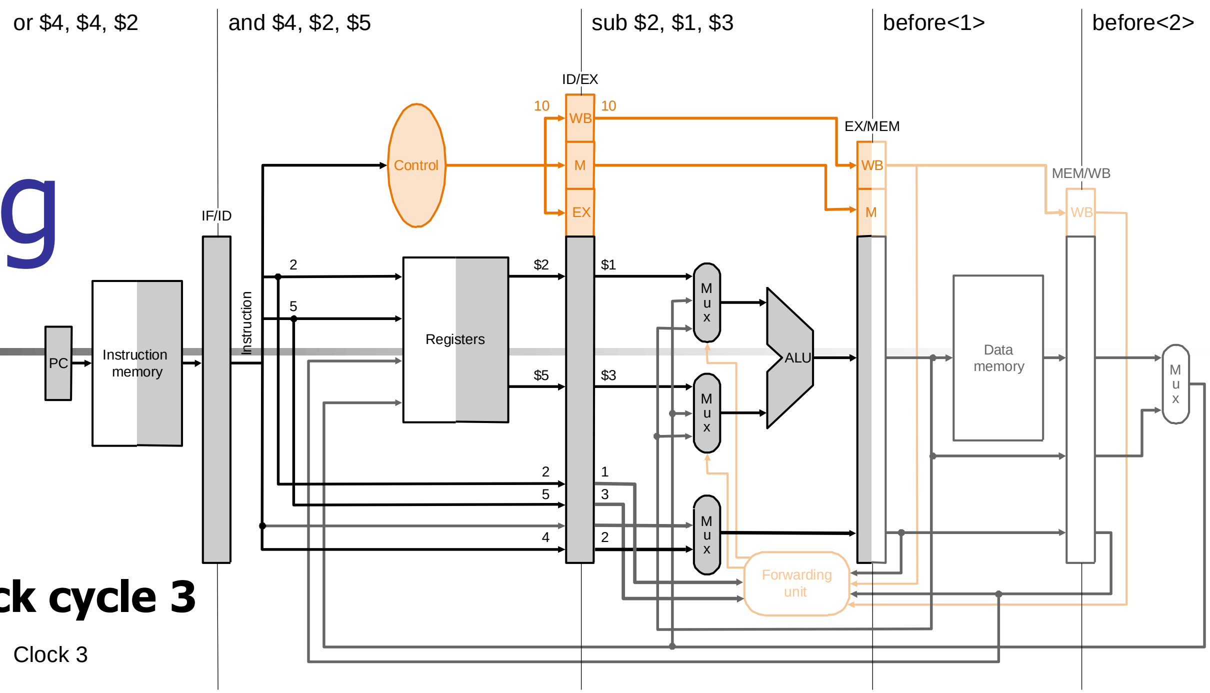
Datapath with forwarding hardware and control wires – certain details, e.g., branching hardware, are omitted to simplify the drawing

Note: so far we have only handled forwarding to R-type instructions…!

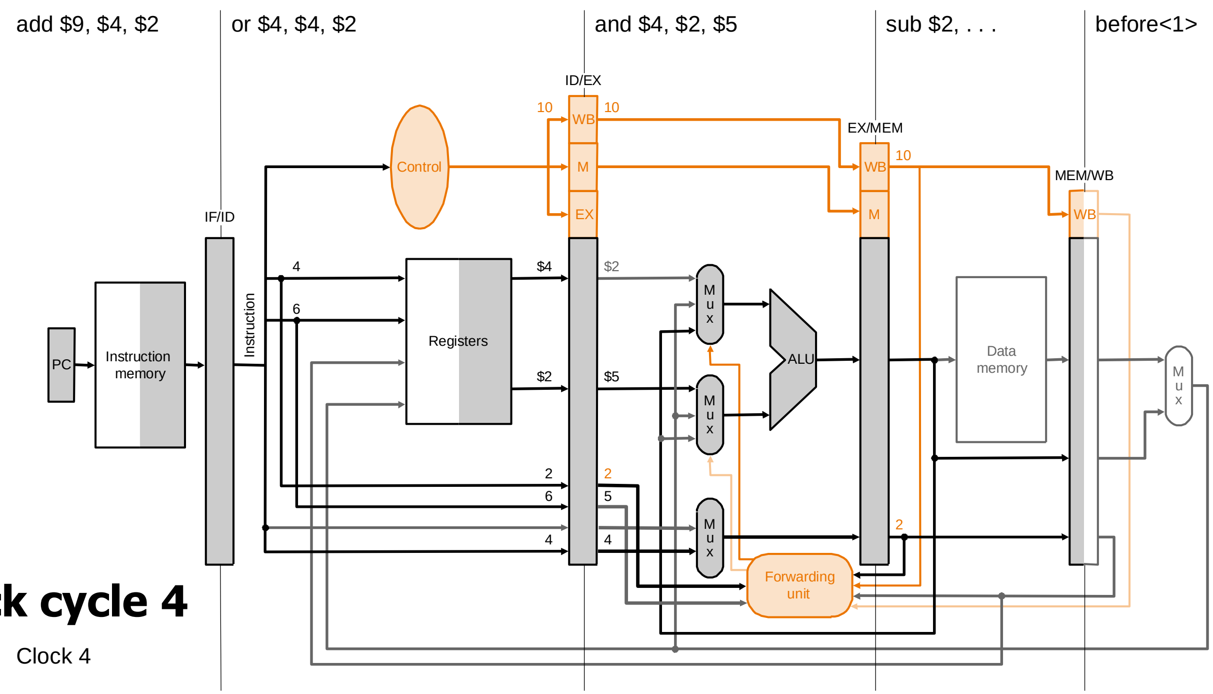
Forwarding

- Execution example:

```
sub $2, $1, $3
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```



Clock cycle 3



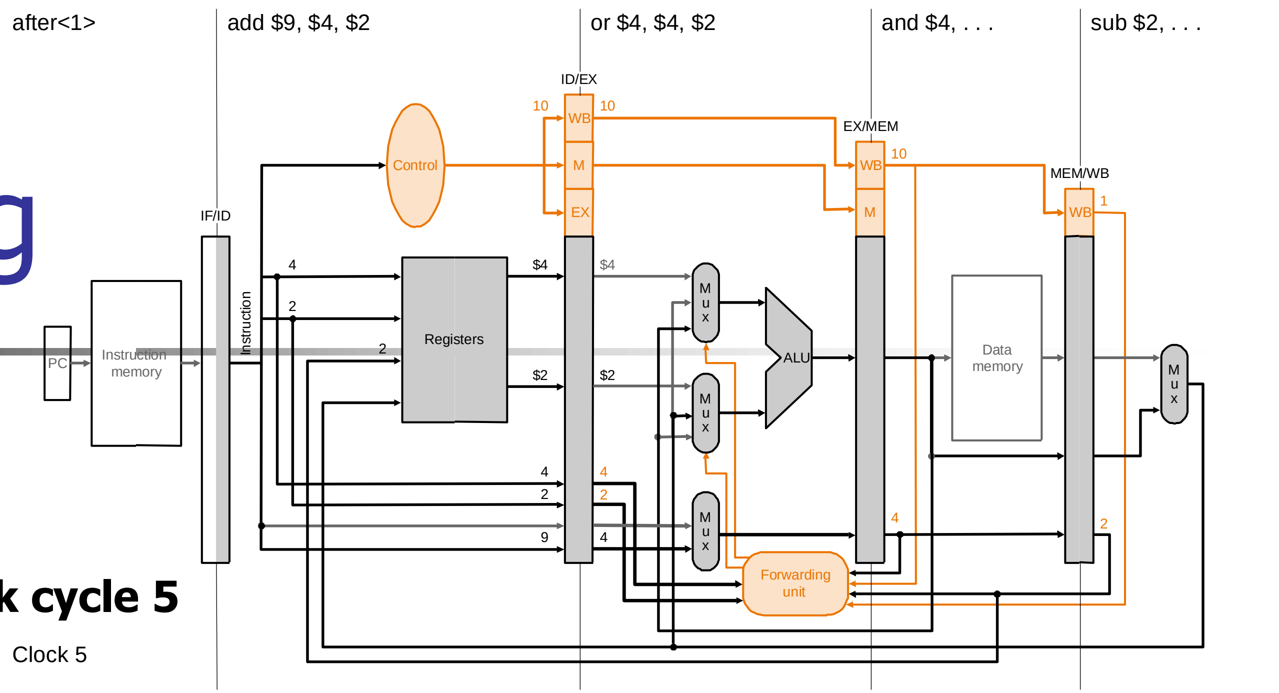
Clock cycle 4

Forwarding

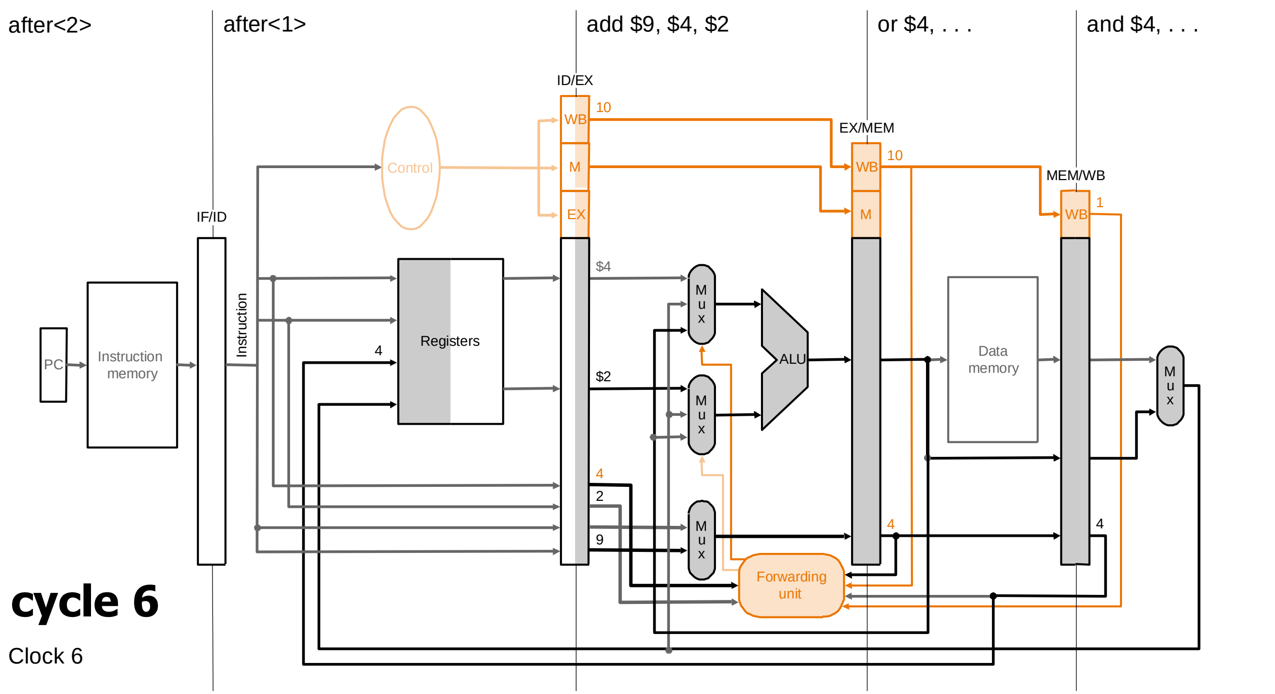
- Execution example (cont.):

```
sub $2, $1, $3
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```

Clock cycle 5



Clock cycle 6

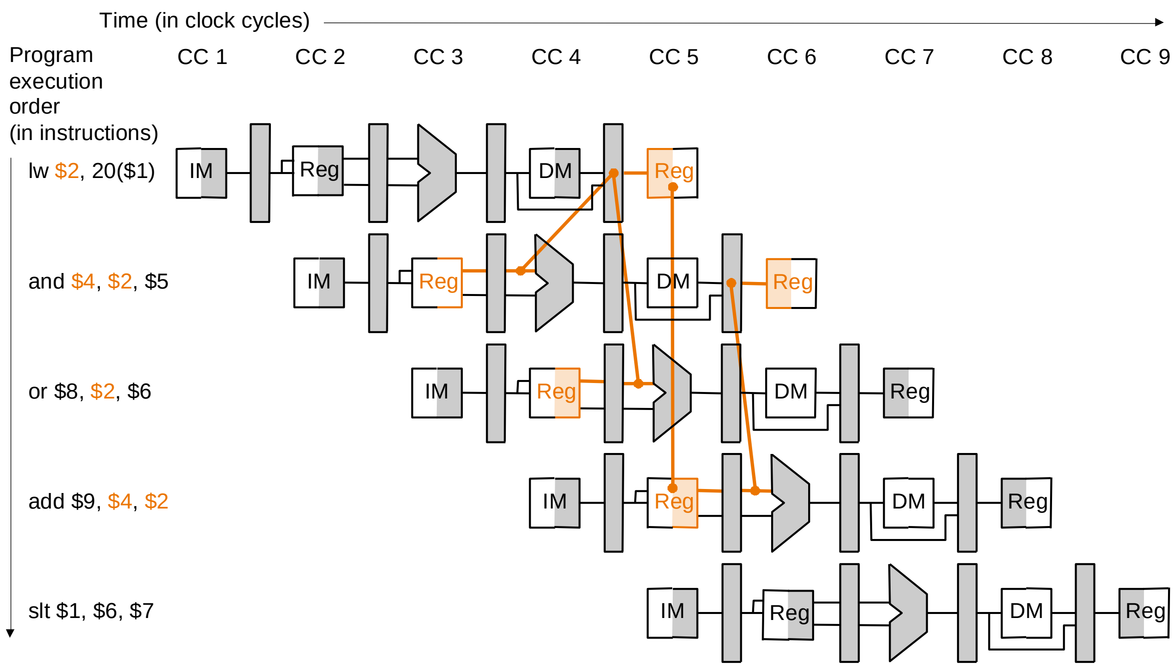


Data Hazards and Stalls

- Load word can still cause a hazard:
  - an instruction tries to read a register following a load instruction that writes to the same register

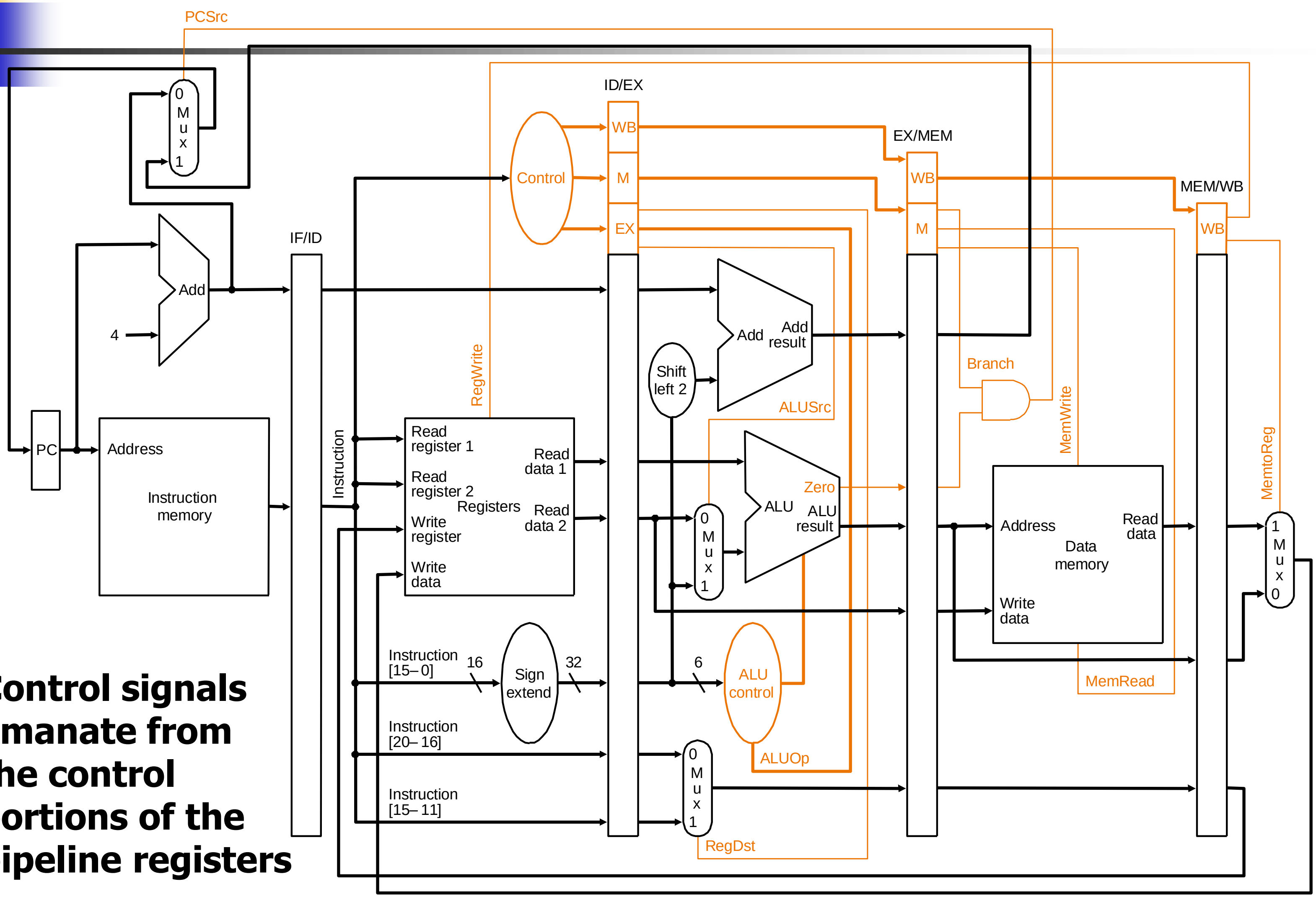
```
lw  $2, 20($1)
and $4, $2, $5
or  $8, $2, $6
add $9, $4, $2
Slt $1, $6, $7
```

As even a pipeline dependency goes backward in time forwarding will not solve the hazard



- therefore, we need a *hazard detection unit* to *stall* the pipeline after the load instruction

Pipelined Datapath with Control II (as before)



Control signals emanate from the control portions of the pipeline registers

Hazard Detection Logic to Stall

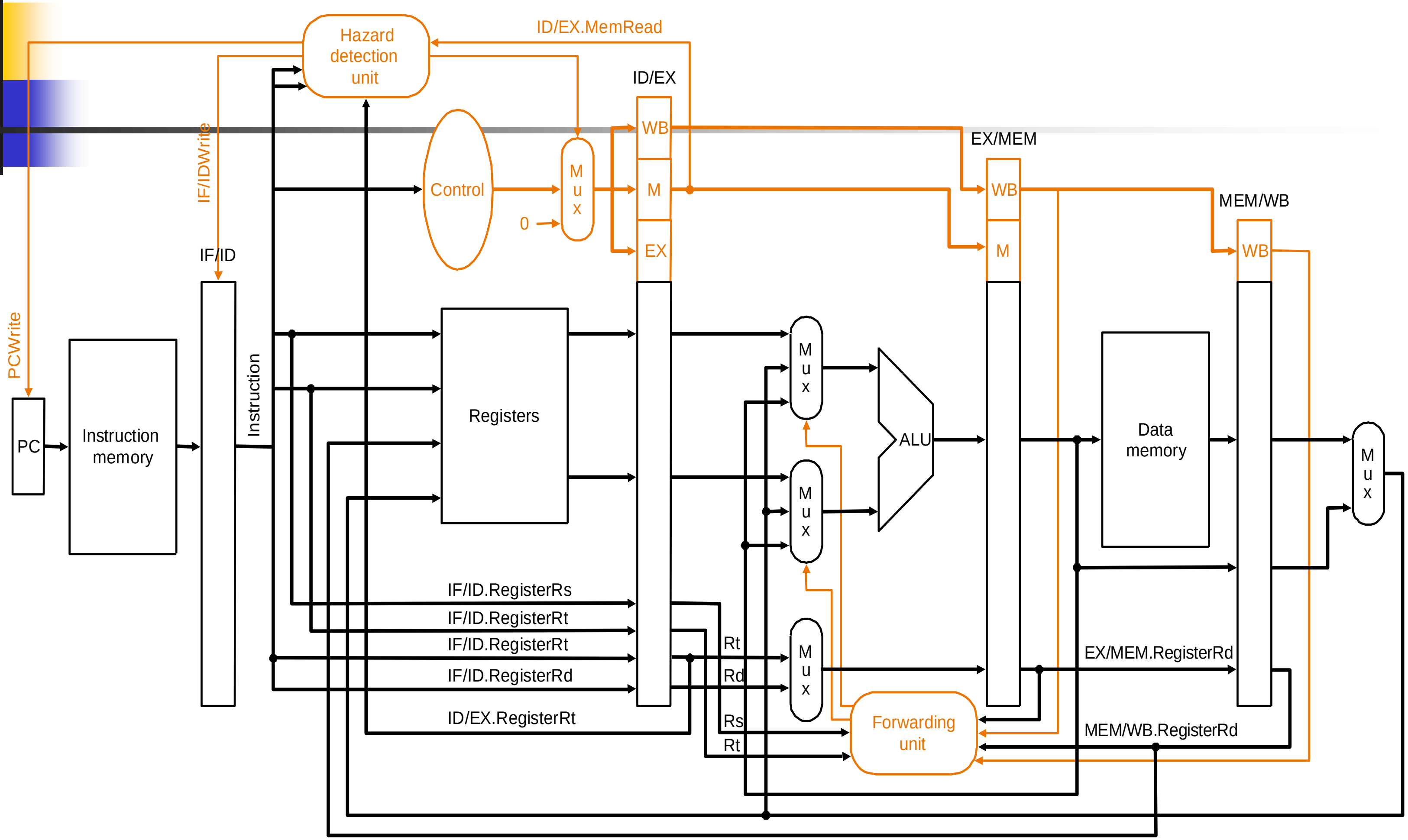
- Hazard detection unit implements the following check if to stall

```
if ( ID/EX.MemRead                                // if the instruction in the EX stage is a load…
    and ( ( ID/EX.RegisterRt = IF/ID.RegisterRs )        // and the destination register
      or  ( ID/EX.RegisterRt = IF/ID.RegisterRt ) ) )  // matches either source register
                                                  // of the instruction in the ID stage, then…
stall the pipeline
```

Mechanics of Stalling

- If the check to stall verifies, then the *pipeline needs to stall only 1 clock cycle* after the load as after that the forwarding unit can resolve the dependency
- What the hardware does to stall the pipeline 1 cycle:
  - *does not let the IF/ID register change* (*disable write*!) – this will cause the instruction in the ID stage to repeat, i.e., *stall*
  - therefore, the instruction, just behind, in the IF stage must be stalled as well – so hardware *does not let the PC change* (*disable write*!) – this will cause the instruction in the IF stage to repeat, i.e., *stall*
  - *changes all the EX, MEM and WB control fields in the ID/EX pipeline register to 0*, so effectively the instruction just behind the load becomes a `nop` – a *bubble* is said to have been inserted into the pipeline
    - note that we cannot turn that instruction into an `nop` by 0ing all the bits in the instruction itself – recall `nop` = 00…0 (32 bits) – because it has already been decoded and control signals generated

Hazard Detection Unit

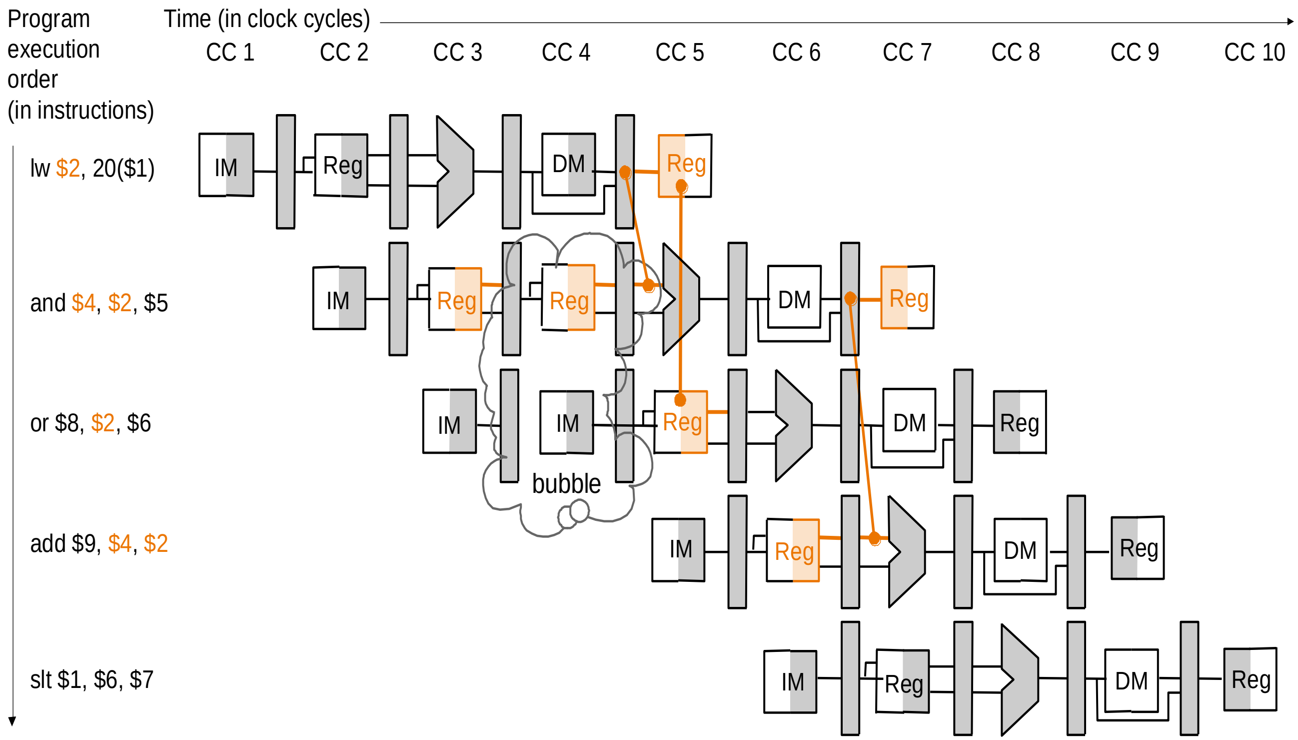


Datapath with forwarding hardware, the hazard detection unit and controls wires – certain details, e.g., branching hardware are omitted to simplify the drawing

Stalling Resolves a Hazard

- Same instruction sequence as before for which forwarding by itself could not resolve the hazard:

```
lw   $2, 20($1)
and $4, $2, $5
or   $8, $2, $6
add $9, $4, $2
Slt $1, $6, $7
```

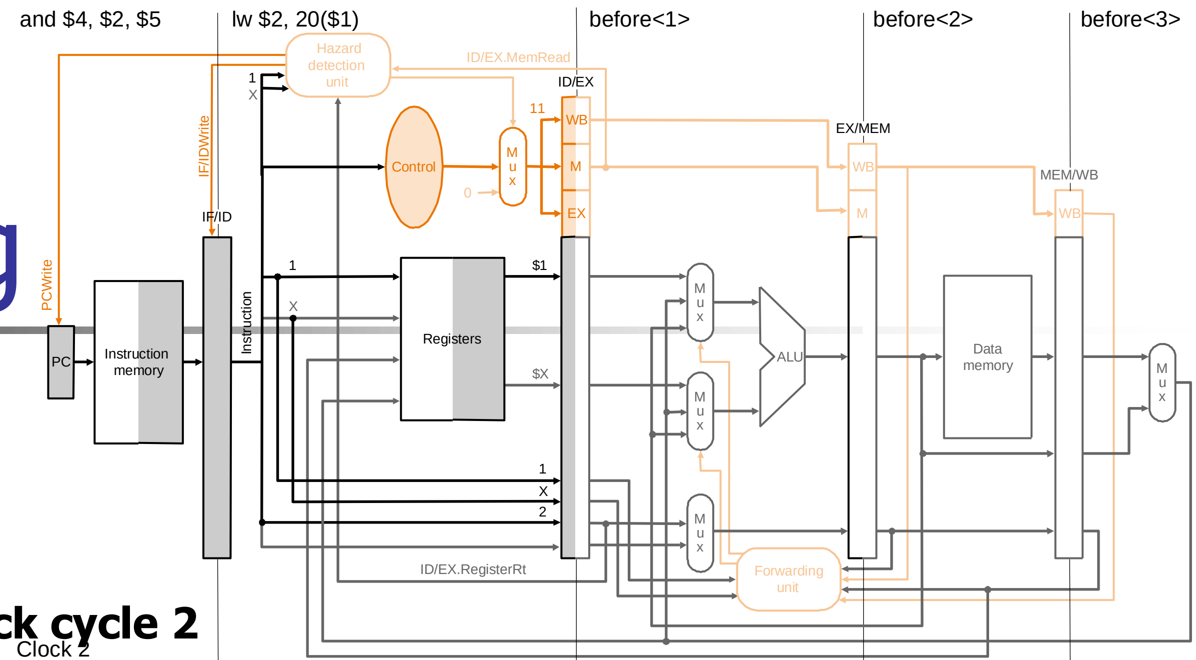


Hazard detection unit inserts a 1-cycle bubble in the pipeline, after which all pipeline register dependencies go forward so then the forwarding unit can handle them and there are no more hazards

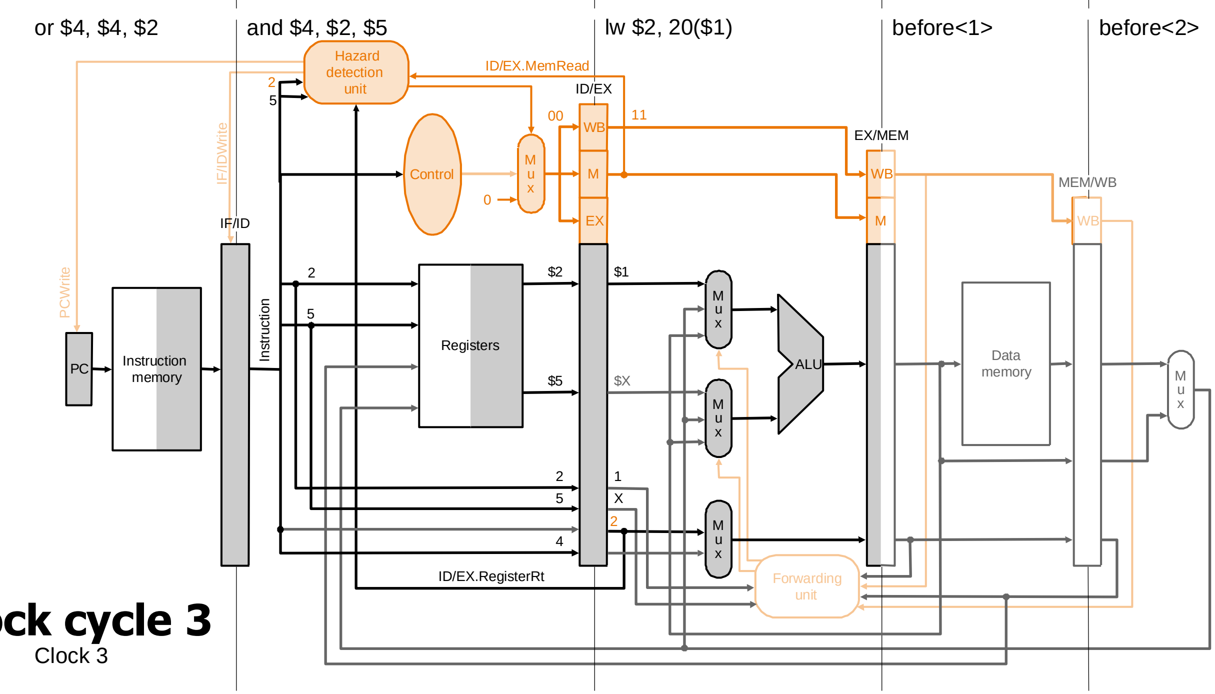
Stalling

- Execution example:

```
lw  $2, 20($1)
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```



Clock cycle 2

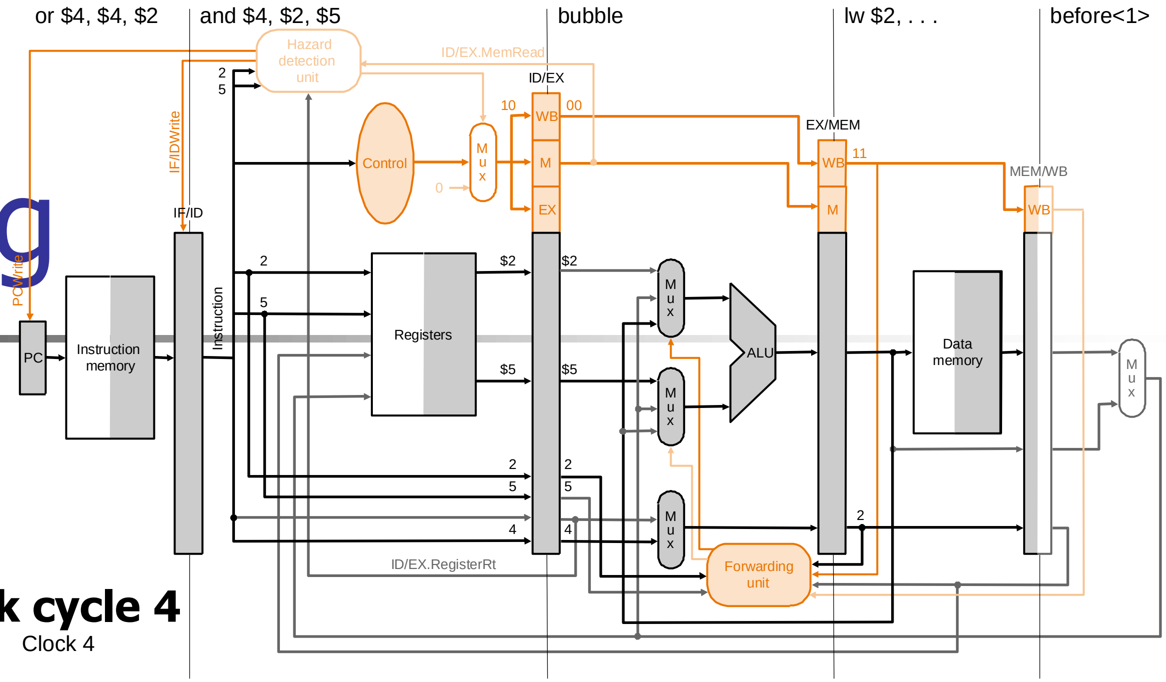


Clock cycle 3

Stalling

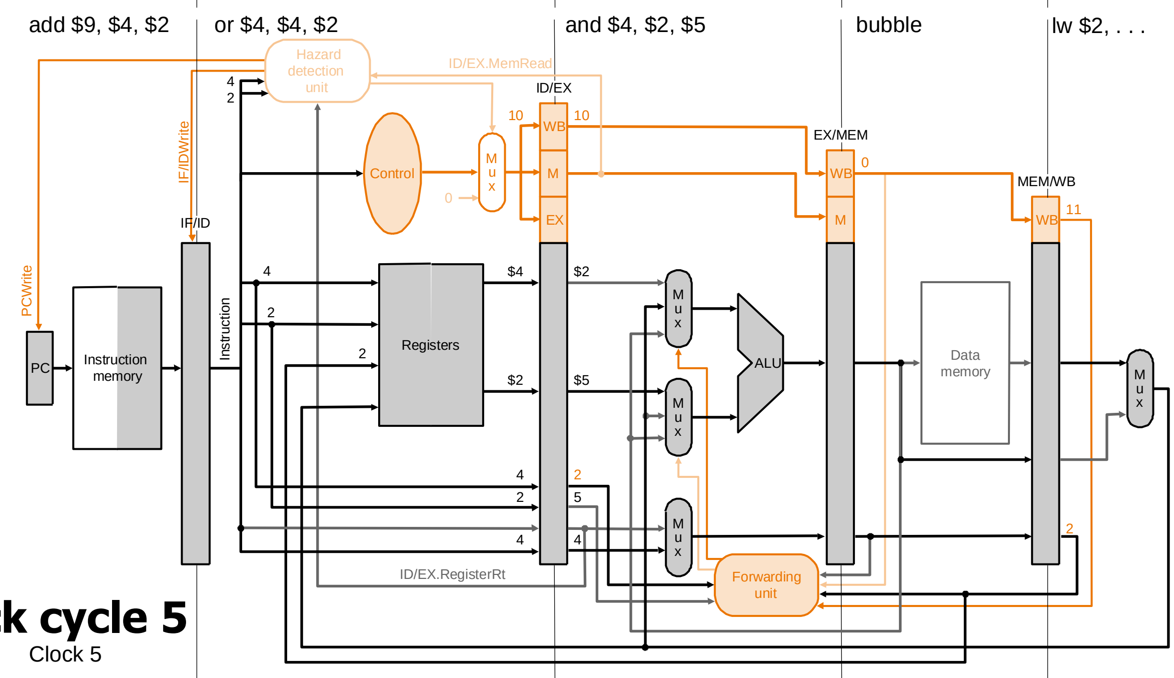
- Execution example (cont.):

```
lw  $2, 20($1)
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```



Clock cycle 4

Clock 4



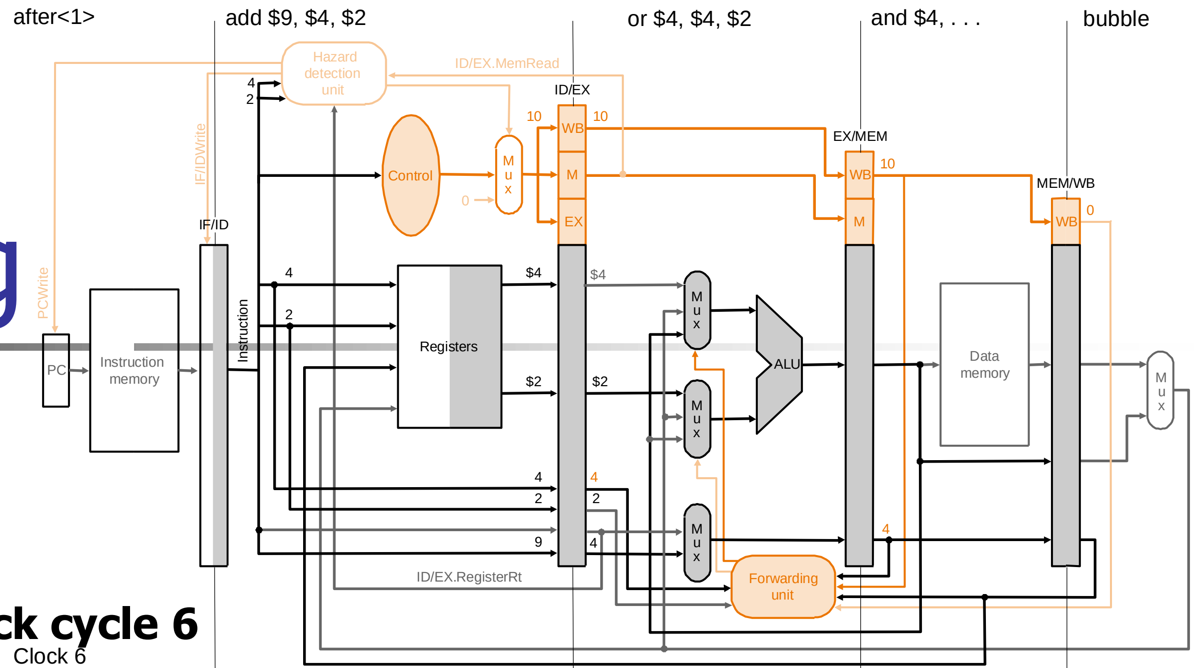
Clock cycle 5

Clock 5

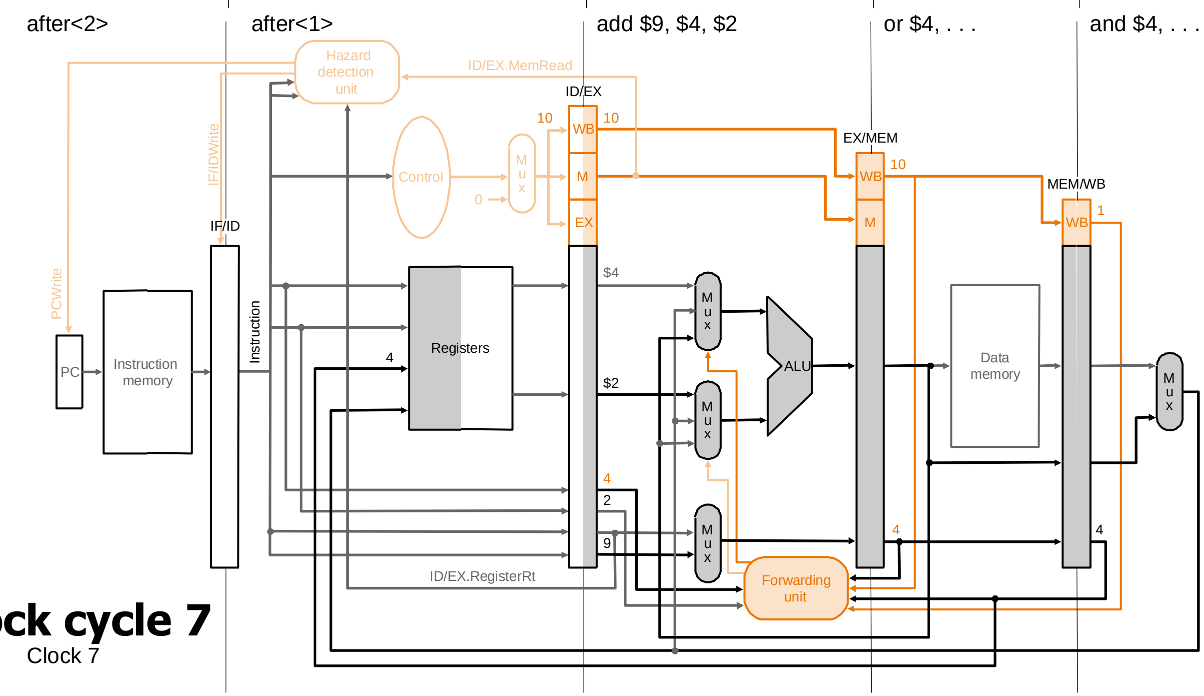
Stalling

- Execution example (cont.):

```
lw  $2, 20($1)
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```



Clock cycle 6



Clock cycle 7

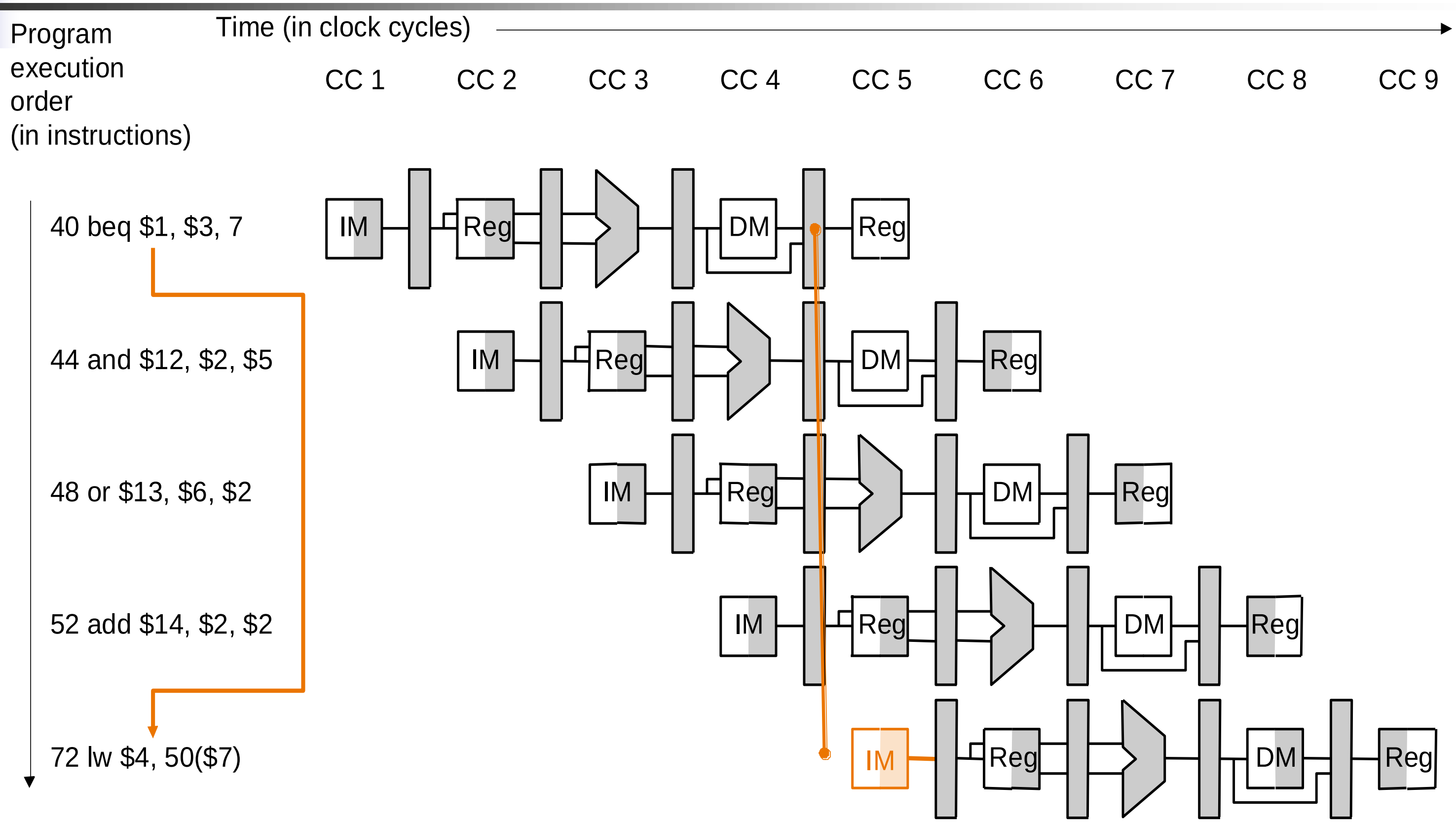
Control (or Branch) Hazards

- Problem with branches in the pipeline we have so far is that the *branch decision is not made till the MEM stage* – so *what instructions, if at all, should we insert into the pipeline following the branch instructions*?

- Possible solution: *stall* the pipeline till branch decision is known
  - not efficient, slow the pipeline significantly!

- Another solution: *predict* the branch outcome
  - e.g., always predict *branch-not-taken – continue with next sequential instructions*
  - if the prediction is wrong have to *flush* the pipeline behind the branch – discard instructions already fetched or decoded – and *continue execution at the branch target*

Predicting Branch-not-taken: Misprediction delay



The outcome of branch taken (prediction wrong) is decided only when `beq` is in the MEM stage, so the following three sequential instructions already in the pipeline have to be flushed and execution resumes at `lw`

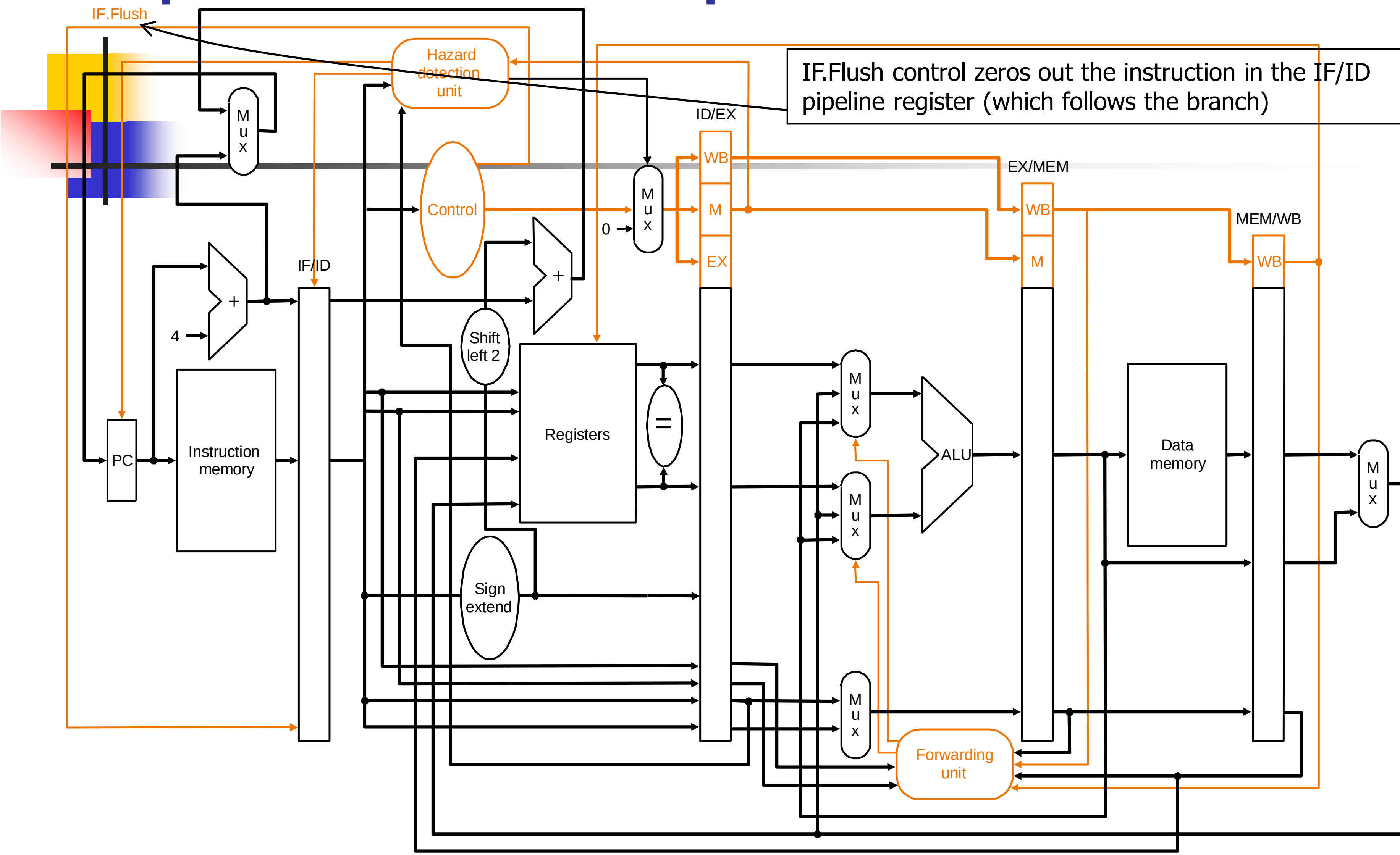
Optimizing the Pipeline to Reduce Branch Delay

- *Move the branch decision* from the MEM stage (as in our current pipeline) *earlier to the ID stage*
  - *calculating the branch target address* involves moving the branch adder from the MEM stage to the ID stage – inputs to this adder, the PC value and the immediate fields are already available in the IF/ID pipeline register
  - *calculating the branch decision* is efficiently done, e.g., for equality test, by XORing respective bits and then ORing all the results and inverting, rather than using the ALU to subtract and then test for zero (when there is a carry delay)
    - with the more efficient equality test we can put it in the ID stage without significantly lengthening this stage – remember an objective of pipeline design is to keep pipeline stages balanced
  - *we must correspondingly make additions to the forwarding and hazard detection units* to forward to or stall the branch at the ID stage in case the branch decision depends on an earlier result

Flushing on Misprediction

- Same strategy as for stalling on load-use data hazard…
- Zero out all the control values (or the instruction itself) in pipeline registers for the instructions following the branch that are already in the pipeline – effectively turning them into `nop`s – so they are flushed
  - in the optimized pipeline, with branch decision made in the ID stage, we have to flush only one instruction in the IF stage – the branch delay penalty is then only one clock cycle

Optimized Datapath for Branch



Branch decision is moved from the MEM stage to the ID stage – simplified drawing not showing enhancements to the forwarding and hazard detection units

Pipelined Branch

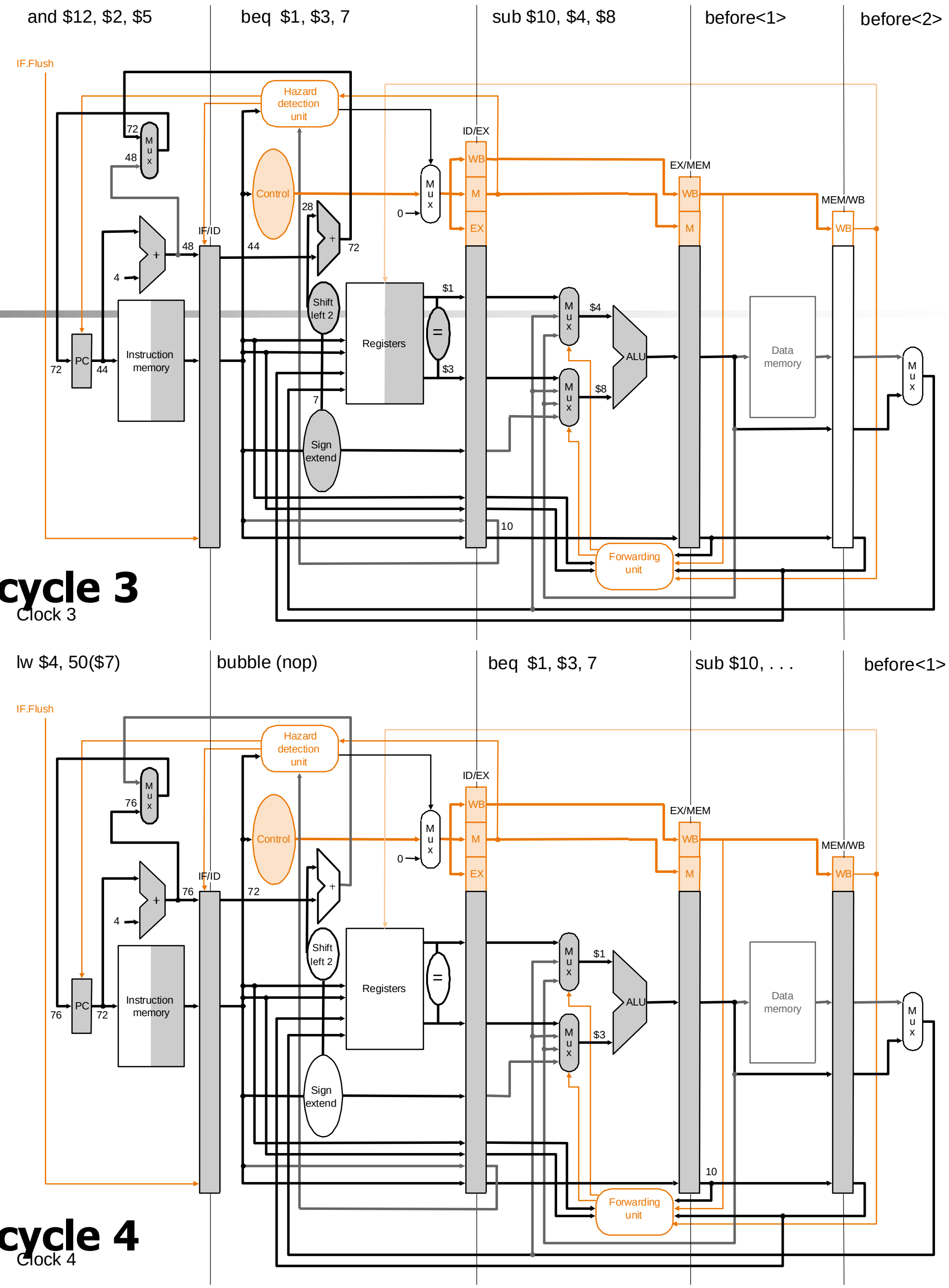
- Execution example:

```
36 sub $10, $4, $8
40 beq $1,  $3,  7
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)
```

Optimized pipeline with only one bubble as a result of the taken branch

Clock cycle 3

Clock cycle 4



Simple Example: Comparing Performance

- *Compare performance for single-cycle, multicycle, and pipelined datapaths using the gcc instruction mix*
  - assume 2 ns for memory access, 2 ns for ALU operation, 1 ns for register read or write
  - assume gcc instruction mix 23% loads, 13% stores, 19% branches, 2% jumps, 43% ALU
  - for pipelined execution assume
    - 50% of the loads are followed immediately by an instruction that uses the result of the load
    - 25% of branches are mispredicted
    - branch delay on misprediction is 1 clock cycle
    - jumps always incur 1 clock cycle delay so their average time is 2 clock cycles

Simple Example: Comparing Performance

- *Single-cycle* (p. 373): average instruction time 8 ns
- *Multicycle* (p. 397): average instruction time 8.04 ns
- *Pipelined*:
  - loads use 1 cc (clock cycle) when no load-use dependency and 2 cc when there is dependency – given 50% of loads are followed by dependency the average cc per load is 1.5
  - stores use 1 cc each
  - branches use 1 cc when predicted correctly and 2 cc when not – given 25% misprediction average cc per branch is 1.25
  - jumps use 2 cc each
  - ALU instructions use 1 cc each
  - therefore, average CPI is

$1.5 \times 23\% + 1 \times 13\% + 1.25 \times 19\% + 2 \times 2\% + 1 \times 43\% = 1.18$

  - therefore, average instruction time is $1.18 \times 2 = 2.36$ ns