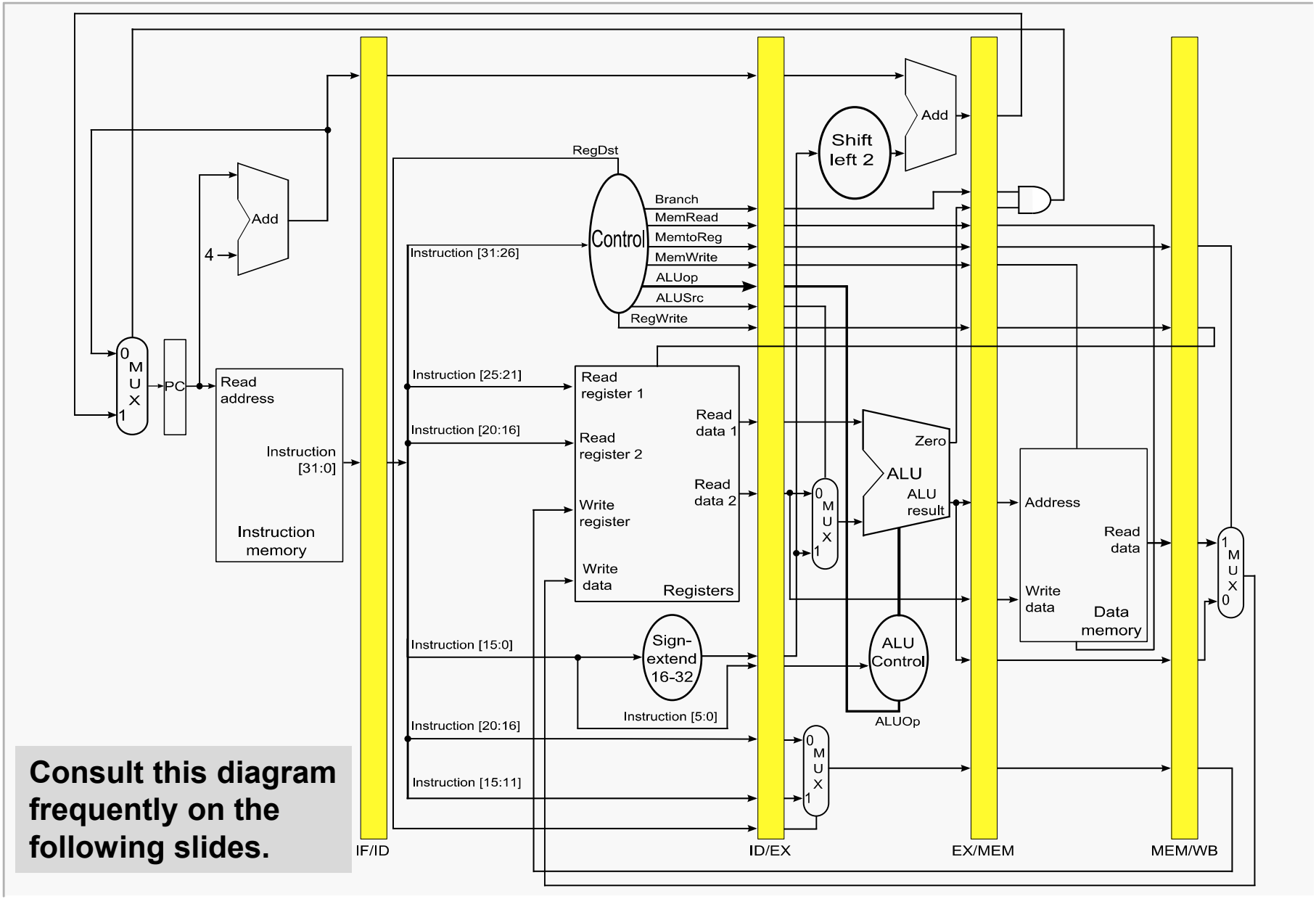
Pipelined Control with Interstage Buffers



Consult this diagram frequently on the following slides.

Pipelined Control Signals

Recall that we must ensure that each control signal "travels with" the instruction to which it applies.

The interstage buffers provide support for this synchronization.

We need to determine which signals each interstage buffers must store.

The key is to determine to which stage a signal must be passed, then make sure it reaches that state in synchronization with the correct instruction.

Analysis: ALUSrc

The Control module sets control signals during the Instruction Decode stage.

The ALUSrc signal must be applied to the multiplexor during the Execute stage, since that is when the corresponding instruction will need for the correct operand to be selected.

Suppose the instruction is fetched on clock cycle N, then:

cycle	actions
N + 1	instruction enters ID stage; ALUSrc is set
N + 2	instruction enters EX stage; ALUSrc is needed at multiplexor

So, we must store the ALUSrc signal in the ID/EX interstage buffer during clock cycle N + 1 when ALUSrc is set, and then read it from the buffer at the beginning of clock cycle N + 2 and pass it to the multiplexor.

Analysis: RegWrite

The Control module sets control signals during the Instruction Decode stage.

The RegWrite signal must be applied to the register file during the WriteBack stage, since that is when the data produced by the corresponding instruction will reach the register file.

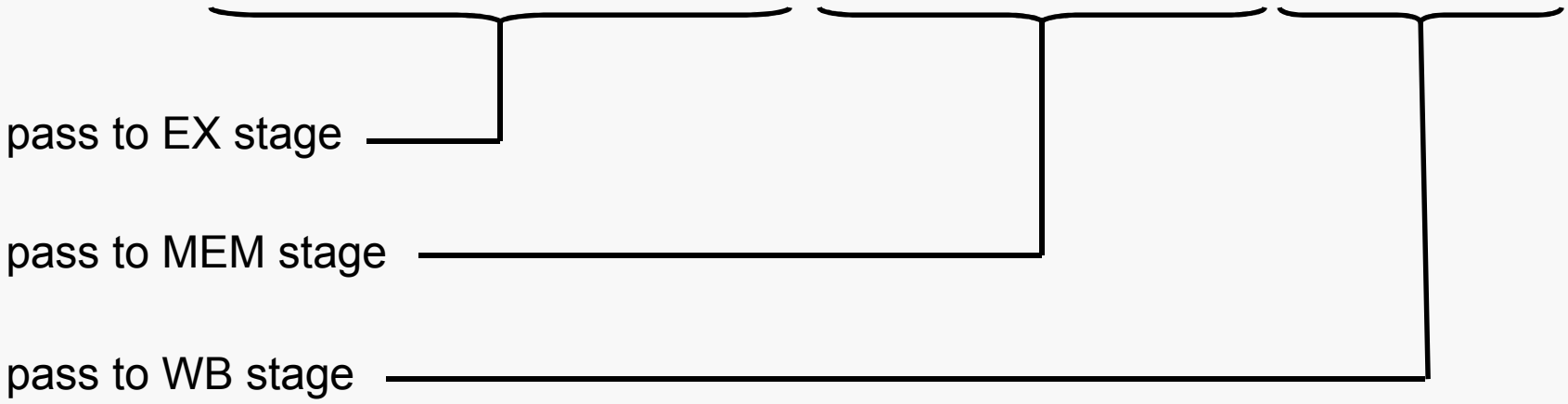
Suppose the instruction is fetched on clock cycle N, then:

cycle	actions
N + 1	instruction enters ID stage; RegWrite is set
N + 2	instruction enters EX stage; RegWrite is not needed yet
N + 3	instruction enters MEM stage; RegWrite is not needed yet
N + 4	instruction enters WB stage; data is ready to write; data and RegWrite are needed at the register file

So, we must pass the RegWrite signal to the ID/EX interstage buffer, then on to the EX/MEM interstage buffer, and then to the MEM/WB interstage buffer, and finally back to the register file.

Control Signals Grouped by Stages

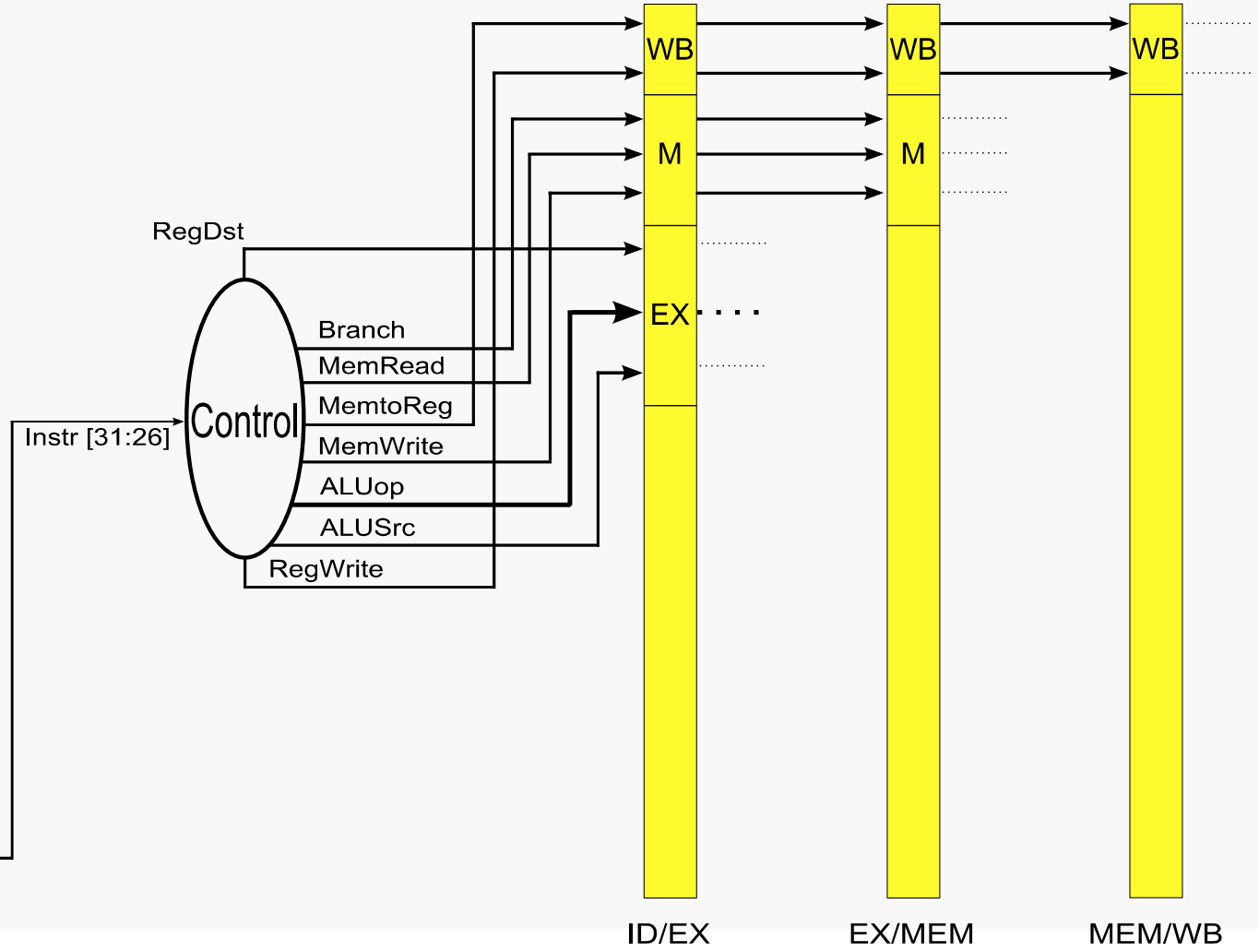
Instruction	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X



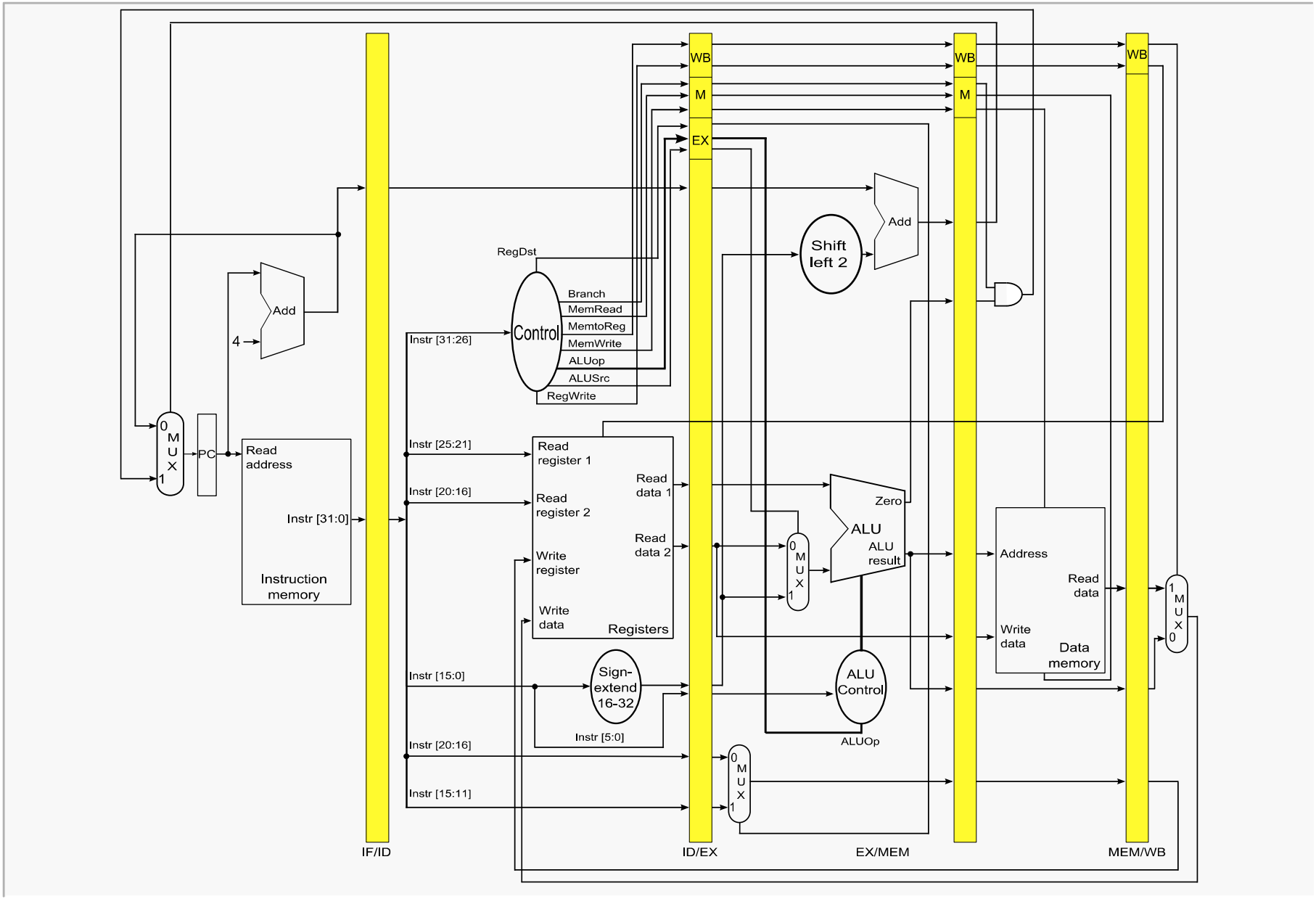
Control Signal Forwarding

Control signals derived from instruction opcode, as before.

Passed synchronously to the appropriate pipeline stage before being applied.



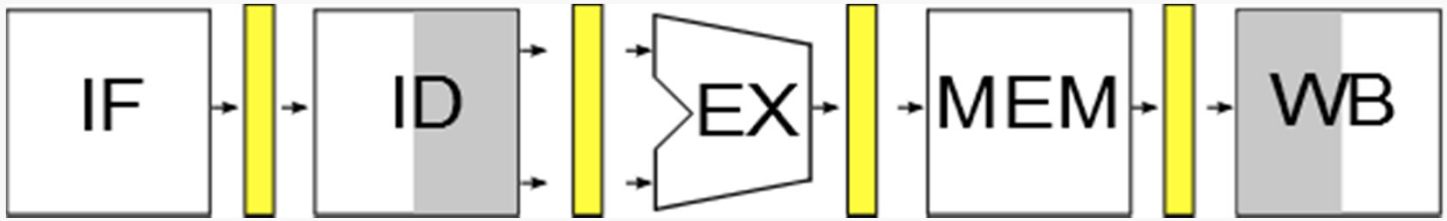
Pipelined Control Overview



Data Hazards in ALU Instructions

Consider this sequence:

```
sub  $2, $1, $3    # value for $2 known end of EX stage;
                   # stored in $2 in WB stage

and $12, $2, $5    # enters ID stage when sub enters EX;
                   # and needs $s2 when enters EX stage;
                   # sub is in MEM stage by then;
                   # $2 has not been written yet
```




```
Tick 0:   sub
Tick 1:   and       sub
Tick 2:             and       sub
Tick 3:                       and       sub
```

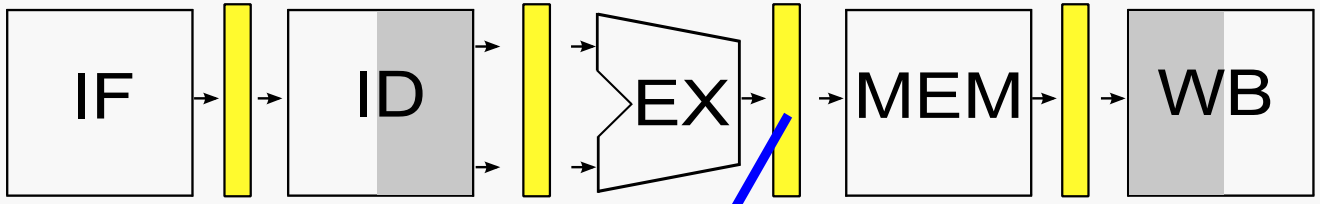
Data hazard?

Data Hazards in ALU Instructions

So this sequence leads to a data hazard involving `$2`:

```
sub  $2, $1, $3
and $12, $2, $5
```

Can we resolve the hazard simply by forwarding?



Tick 0:	sub				
Tick 1:	and	sub			
Tick 2:		and	sub		
Tick 3:			and	sub	

Yes!

But we must deliver the computed value at the right time; the next tick.

And, that value will be sitting in the EX/MEM interstage buffer.

Detecting the Hazard

On the one hand, this is obvious. The first instruction writes a value into a register that is subsequently used as input by the second instruction:

```
sub  $2, $1, $3
and $12, $2, $5
```

We must know the register numbers for both instructions in order to detect the hazard.

More precisely, we must know `rd` for the first instruction and both `rs` and `rt` for the second instruction.

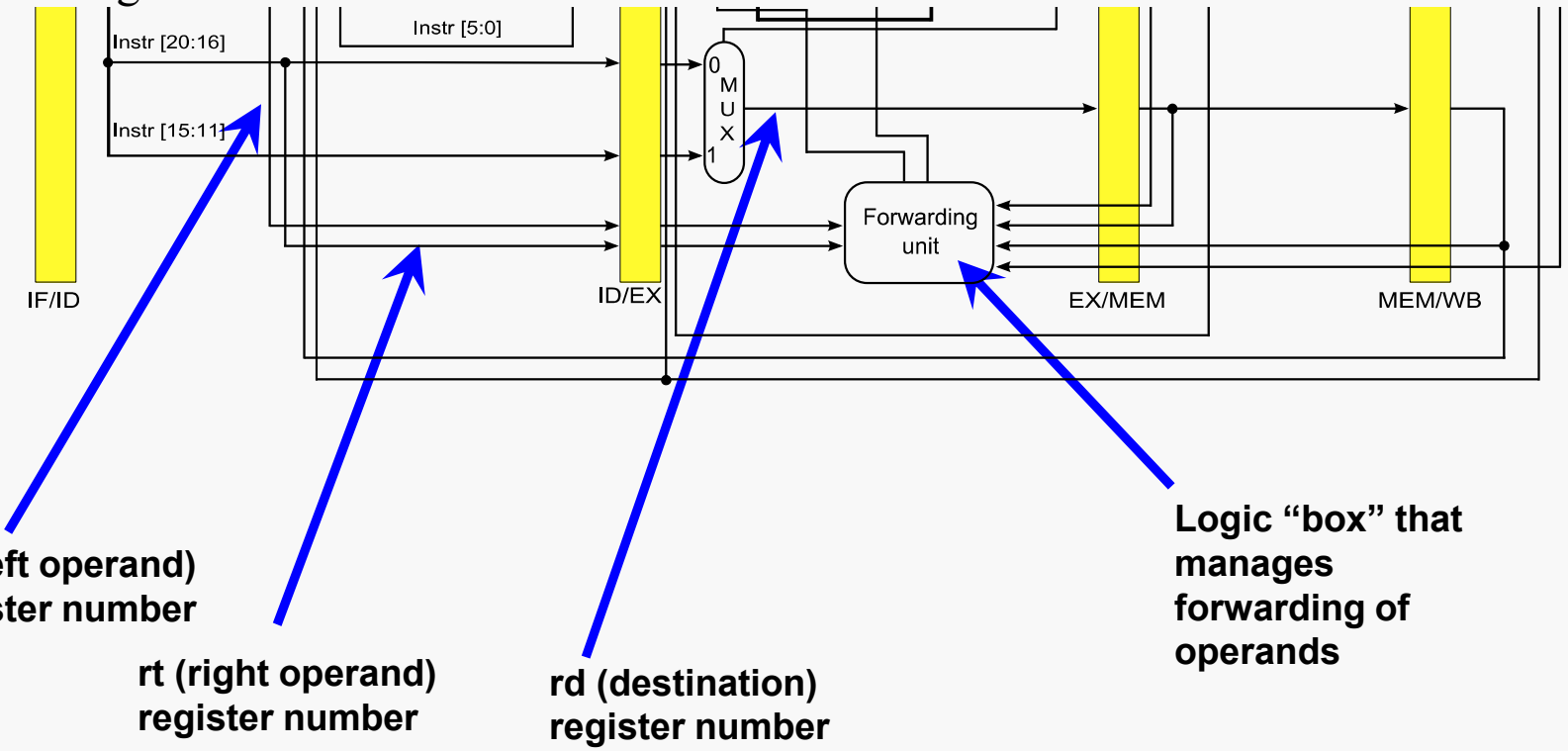
So, we must save those register numbers, via the interstage buffers.

Some notation will help us speak precisely about what's going on:

B.RegisterRX = register number for RX sitting in interstage pipeline buffer B

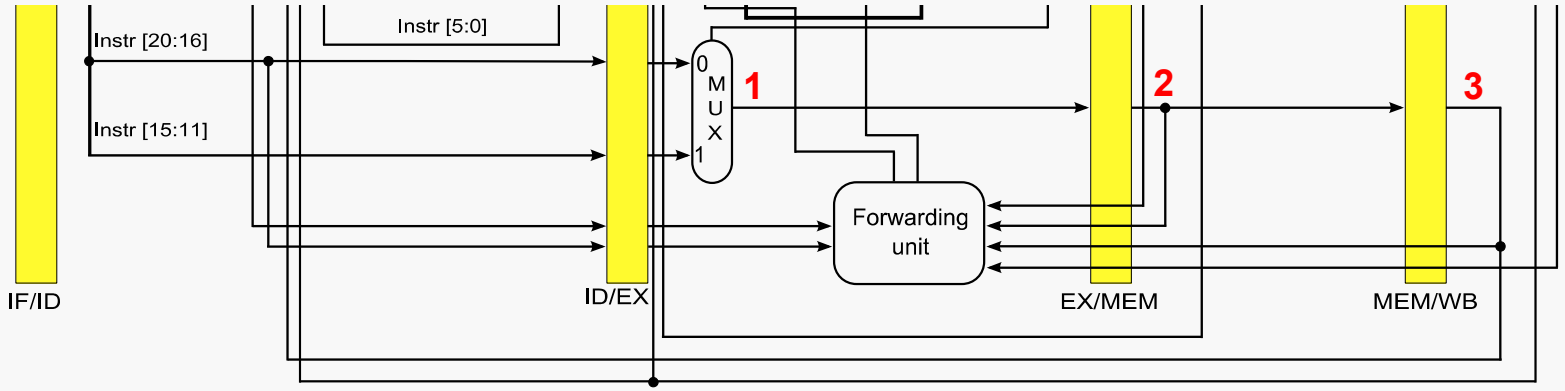
A Glance Ahead

Passing the register numbers:



A Glance Ahead

Passing the register numbers:



1: rd for instruction currently in EX stage

2: rd for instruction currently in MEM stage

3: rd for instruction currently in WB stage

They may all be different!

Detecting the Hazard

Now, for this sequence of instructions:

```
sub  $2, $1, $3
and $12, $2, $5
```

So, we detect the hazard because we see that:

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

Hence, we must forward the ALU output value from the EX/MEM interstage buffer to the `rs` input to the ALU.

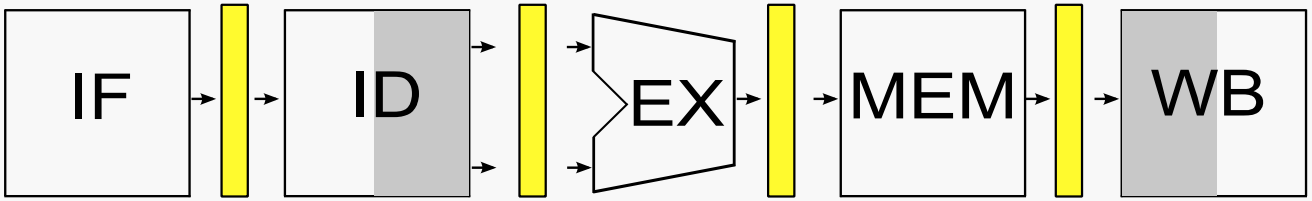
Apparently, we'll need to:

- pass (at least some) register numbers forward via the interstage buffers
- add a logic unit to compare those register numbers to detect hazards
- add data connections to support transferring data values being forwarded
- add some more selection logic (multiplexors)

Data Hazards in ALU Instructions

Now, consider this sequence:

```
sub  $2, $1, $3   # value for $2 known in EX stage
and $12, $2, $5   # enters ID stage when sub enters EX

or  $13, $6, $2   # enters ID stage when sub enters MEM;
                  # $2 has not been written yet
```



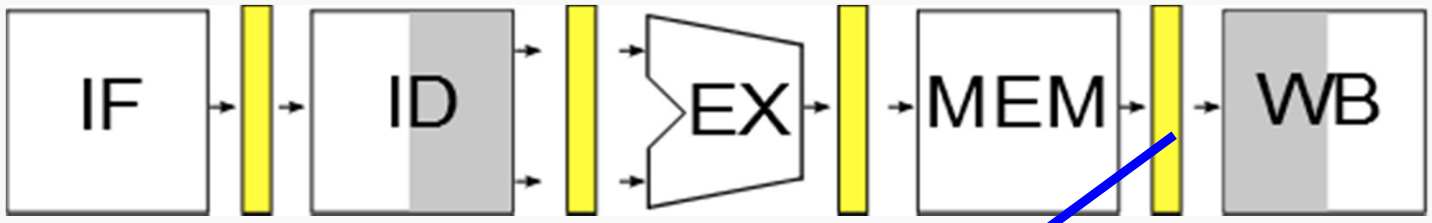

```
Tick 0:  sub
Tick 1:  and      sub
Tick 2:  or       and      sub
Tick 3:           or       and      sub
Tick 4:                    or       and      sub
```

Data hazard?

Data Hazards in ALU Instructions

Again, we have a data hazard:

```
sub  $2, $1, $3   # value for $2 known in EX stage
and $12, $2, $5   # enters ID stage when sub enters EX
or  $13, $6, $2   # enters ID stage when sub enters MEM;
```



Tick 0:	sub				
Tick 1:	and	sub			
Tick 2:	or	and	sub		
Tick 3:		or	and	sub	
Tick 4:			or	and	sub

Yes!

Now, we must deliver the computed value after a delay of one tick, from MEM/WB.

Detecting the Hazard

Again, we have a data hazard:

```
sub  $2, $1, $3
and $12, $2, $5
or  $13, $6, $2
```

So, we detect the hazard because we see that:

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

Hence, we must forward the ALU output value from the MEM/WB interstage buffer to the `rt*` input to the ALU.

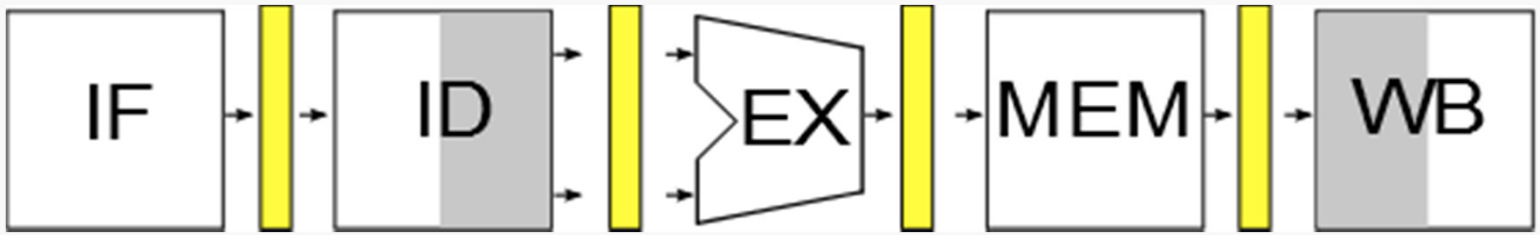
So… detecting data hazards is a multi-stage affair.

*** QTP: why does this one go to the rt input?**

Data Hazards in ALU Instructions

Now, consider this sequence:

```
sub  $2, $1, $3   # value for $2 known in EX stage;
and $12, $2, $5   # enters ID stage when sub enters EX;
or  $13, $6, $2   # enters ID stage when sub enters MEM;

add $14, $2, $2   # enters ID stage when sub enters WB;
                  # $2 has not been written yet, but. . .
```



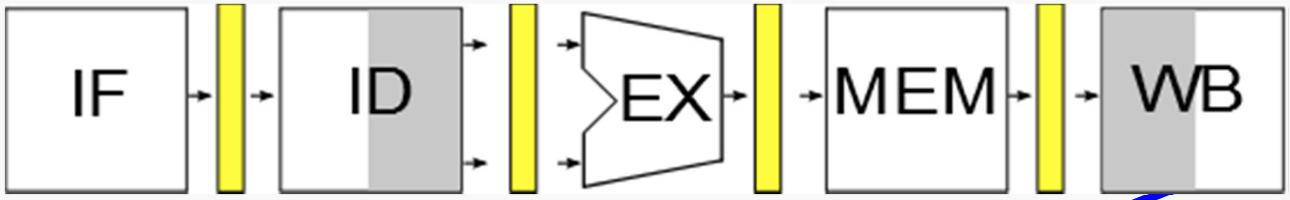
Tick 0:	sub				
Tick 1:	and	sub			
Tick 2:	or	and	sub		
Tick 3:	add	or	and	sub	
Tick 4:		add	or	and	sub

Data hazard?

Data Hazards in ALU Instructions

Now, there's almost a hazard… but not quite…

```
sub  $2, $1, $3   # value for $2 known in EX stage;
and $12, $2, $5   # enters ID stage when sub enters EX;
or  $13, $6, $2   # enters ID stage when sub enters MEM;
add $14, $2, $2   # enters ID stage when sub enters WB;
```



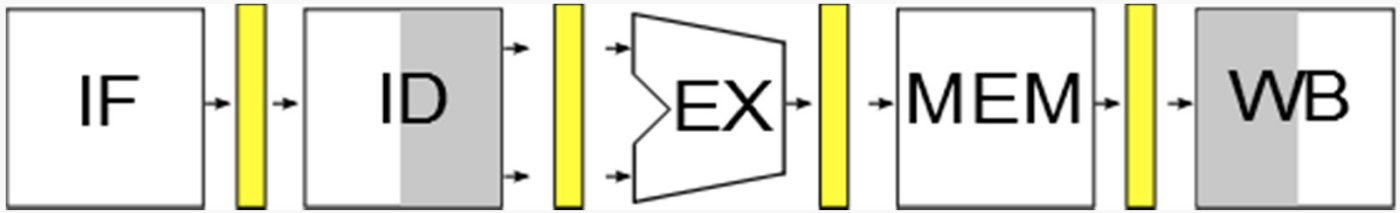
Tick 0:	sub				
Tick 1:	and	sub			
Tick 2:	or	and	sub		
Tick 3:	add	or	and	sub	
Tick 4:		add	or	and	sub

Now, we deliver the computed value to the register file in the first half of tick 4, and it's not read until the second half of that tick!

Data Hazards in ALU Instructions

Now, consider this sequence:

```
sub  $2, $1, $3   # value for $2 known in EX stage;
and $12, $2, $5   # enters ID stage when sub enters EX;
or  $13, $6, $2   # enters ID stage when sub enters MEM;
add $14, $2, $2   # enters ID stage when sub enters WB

sw  $15, 100($2)  # enters ID stage after sub is done
```



Tick 0:	sub				
Tick 1:	and	sub			
Tick 2:	or	and	sub		
Tick 3:	add	or	and	sub	
Tick 4:	sw	add	or	and	sub
Tick 5:		sw	add	or	and

Data hazard?

Detecting the Need to Forward

Here's what we (seem to) know so far:

ALU-related data hazards occur when

EX/MEM.RegisterRd = ID/EX.RegisterRs
EX/MEM.RegisterRd = ID/EX.RegisterRt

} Fwd from EX/MEM pipeline reg

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

} Fwd from MEM/WB pipeline reg

However, we have overlooked (at least) one thing…

Detecting the Need to Forward

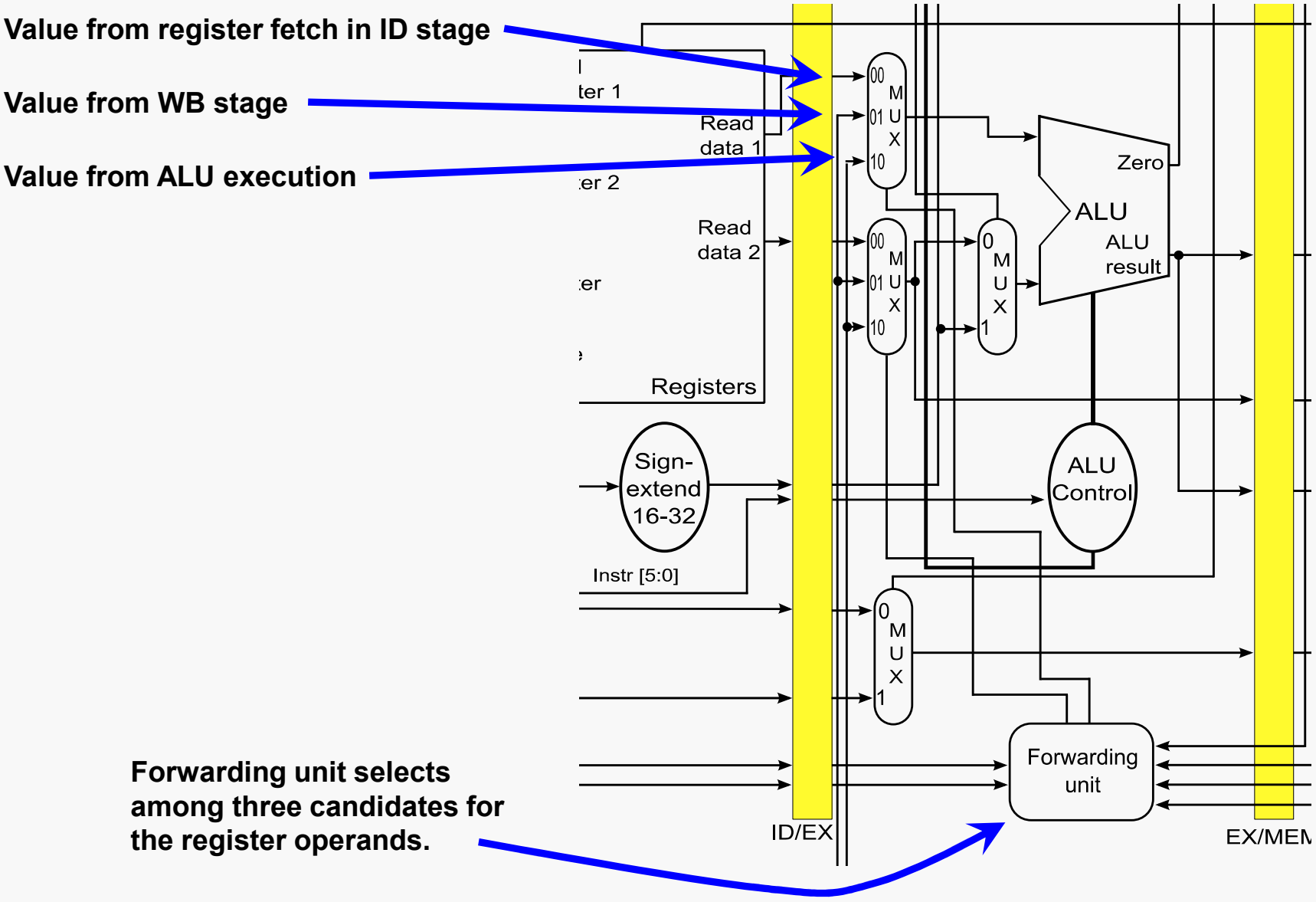
We don't need to forward unless the forwarding (earlier) instruction does actually write a value to a register:

EX/MEM.RegWrite == 1
MEM/WB.RegWrite == 1

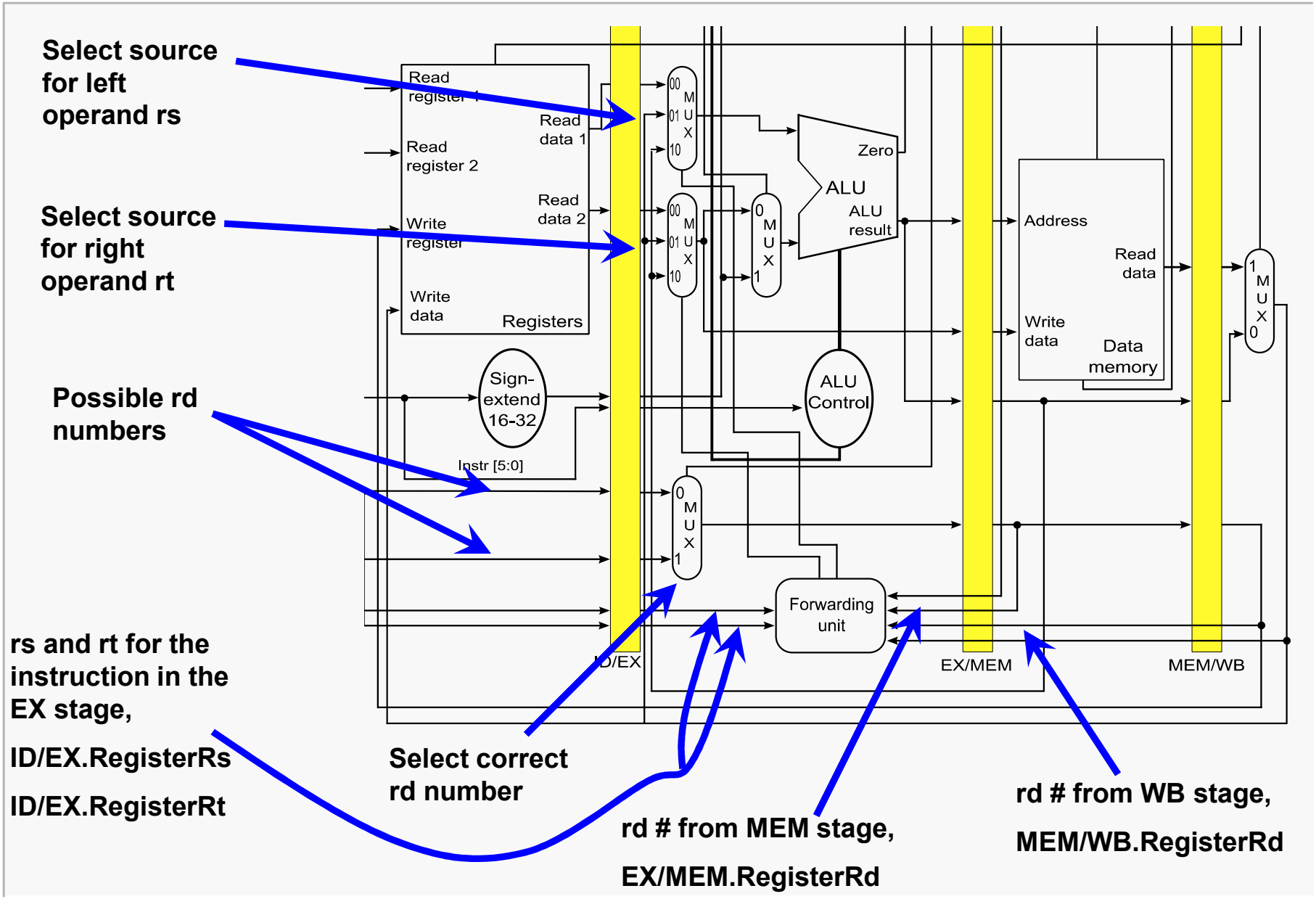
And we only forward if Rd for that instruction is not `$zero`:

EX/MEM.RegisterRd != 0
MEM/WB.RegisterRd != 0

Datapath Change: ALU Operand Selection



Forwarding Paths



Conditions for EX Hazard

Forwarding unit

```
If  ( EX/MEM.RegWrite          and
      EX/MEM.RegisterRd != 0 and
      EX/MEM.RegisterRd == ID/EX.RegisterRs )
then
      ForwardA = 10


If  ( EX/MEM.RegWrite          and
      EX/MEM.RegisterRd != 0  and
      EX/MEM.RegisterRd == ID/EX.RegisterRt )
then
      ForwardB = 10
```

QTP: could BOTH occur with respect to the same instruction?

Conditions for MEM Hazard

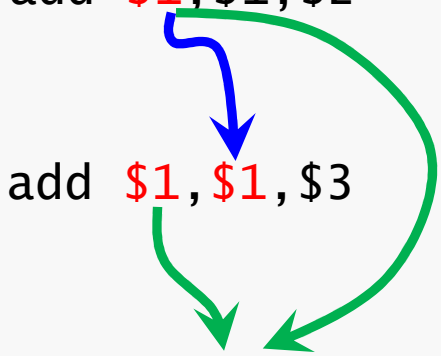

```
If  ( MEM/WB.RegWrite            and
      MEM/WB.RegisterRd != 0  and
      MEM/WB.RegisterRd == ID/EX.RegisterRs )
then
     ForwardA = 01


If  ( MEM/WB.RegWrite            and
      MEM/WB.RegisterRd != 0   and
      MEM/WB.RegisterRd == ID/EX.RegisterRt )
then
     ForwardB = 01
```

QTP: could BOTH an EX hazard and a MEM hazard occur with respect to the same instruction?

Double Data Hazard

Consider the sequence:

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



Both hazards occur… which value do we want to forward?

Revise MEM hazard condition:

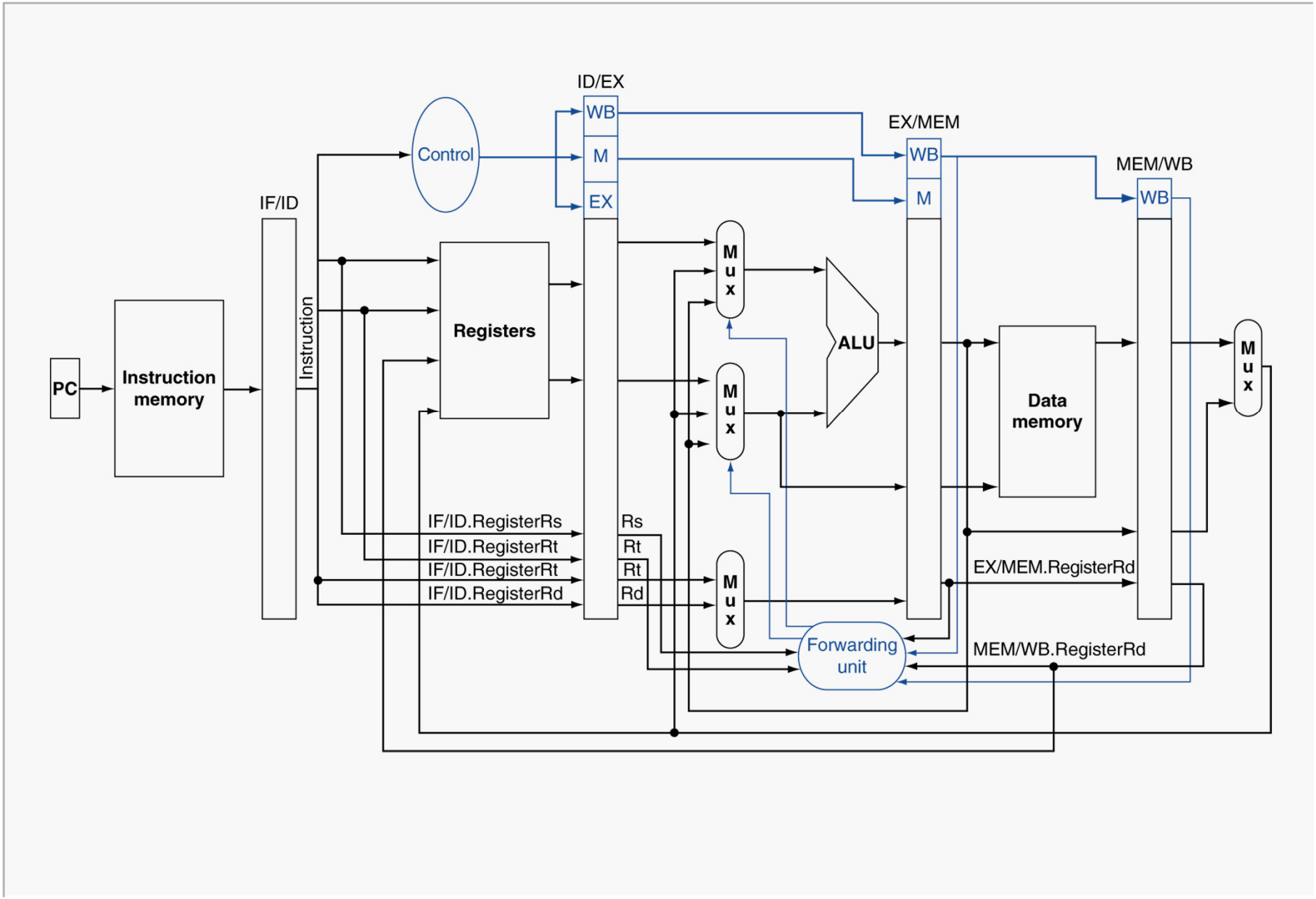
- Only forward if EX hazard condition is not true

Revised Conditions for MEM Hazard

Forwarding unit

```
If  ( MEM/WB.RegWrite and
      MEM/WB.RegisterRd != 0 and
      not ( EX/MEM.RegWrite and
            EX/MEM.RegisterRd != 0 and
            EX/MEM.RegisterRd == ID/EX.RegisterRs ) and
      MEM/WB.RegisterRd == ID/EX.RegisterRs )
then
      ForwardA = 01

If  ( MEM/WB.RegWrite and
      MEM/WB.RegisterRd != 0 and
      not ( EX/MEM.RegWrite and
            EX/MEM.RegisterRd != 0 and
            EX/MEM.RegisterRd == ID/EX.RegisterRt ) and
       MEM/WB.RegisterRd == ID/EX.RegisterRt )
then
       ForwardB = 01
```

Simplified Datapath with Forwarding



Unsimplified Datapath with Forwarding