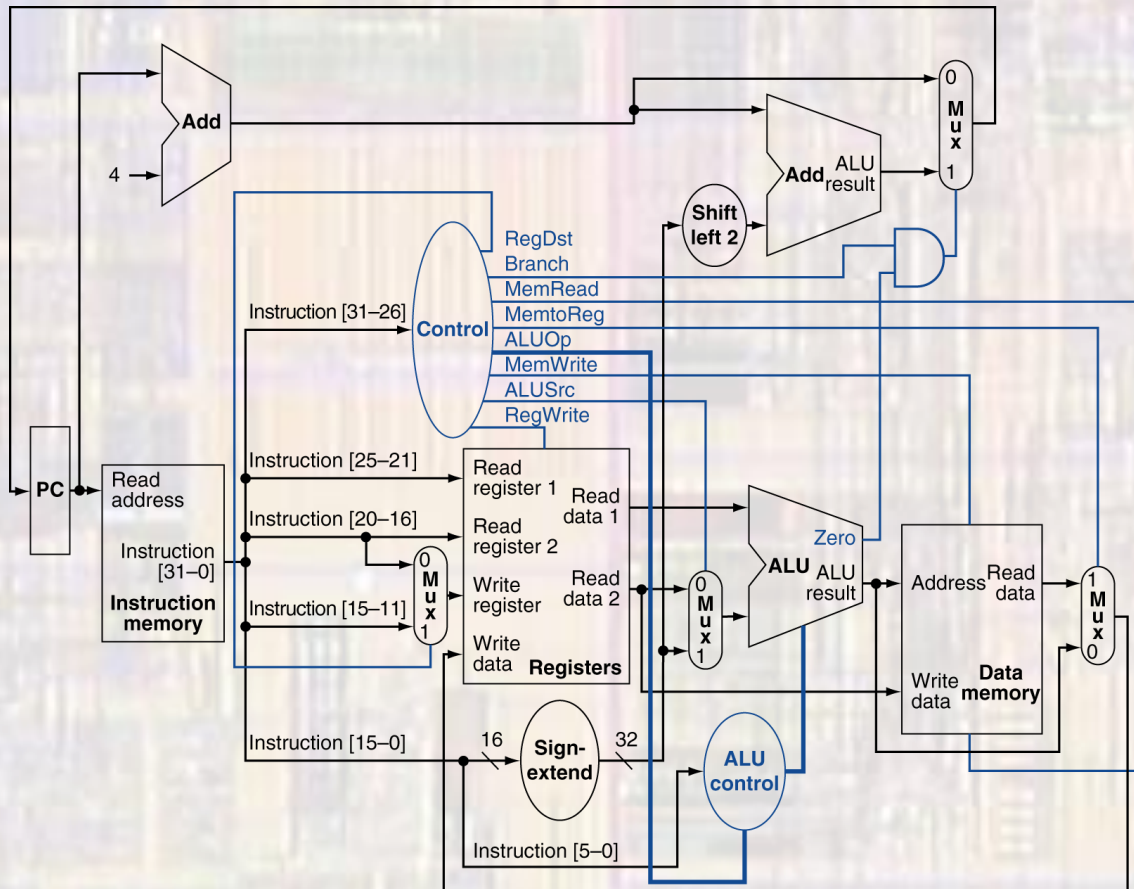


Processor Architecture Pipeline

Last modified 5/4/20

Pipelining

- Simple Datapath



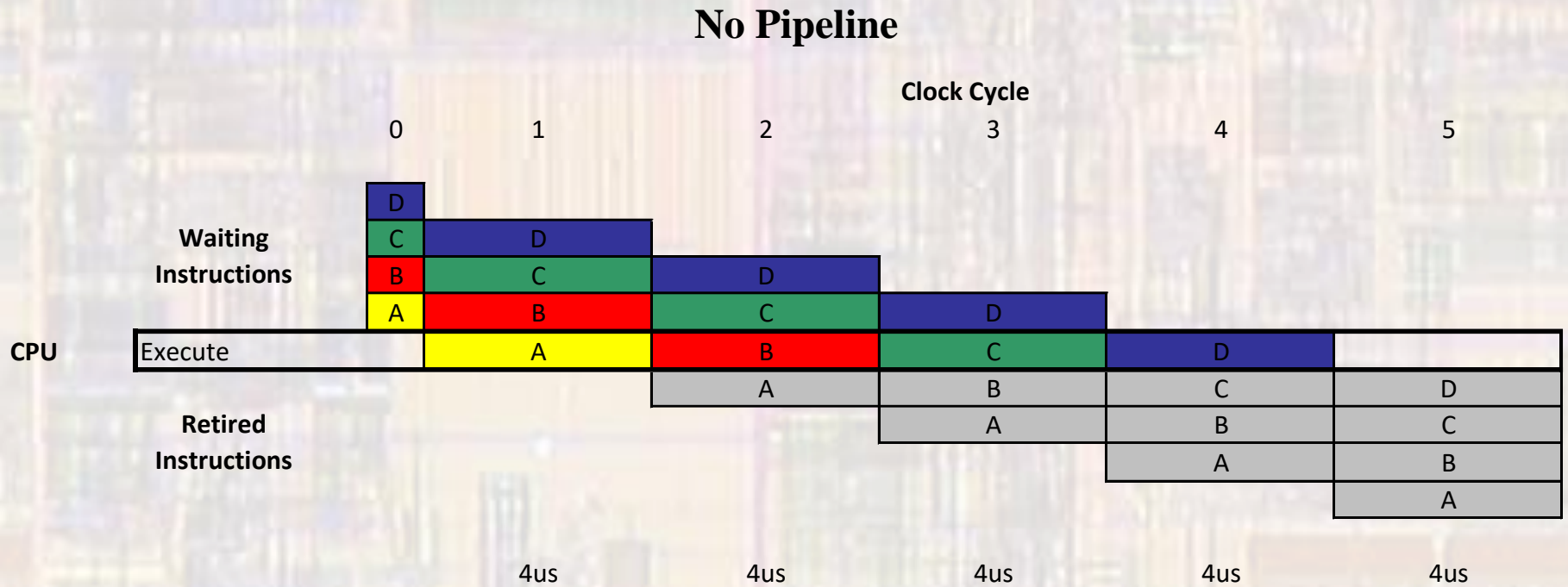
Pipelining

- 5 Stages of Instruction Execution
 - Fetch (IF)
 - Decode / Register Access (ID)
 - Execute (EX)
 - Memory Access (MEM)
 - Write Back (WB)

Pipeline these at 1 stage each

Pipelining

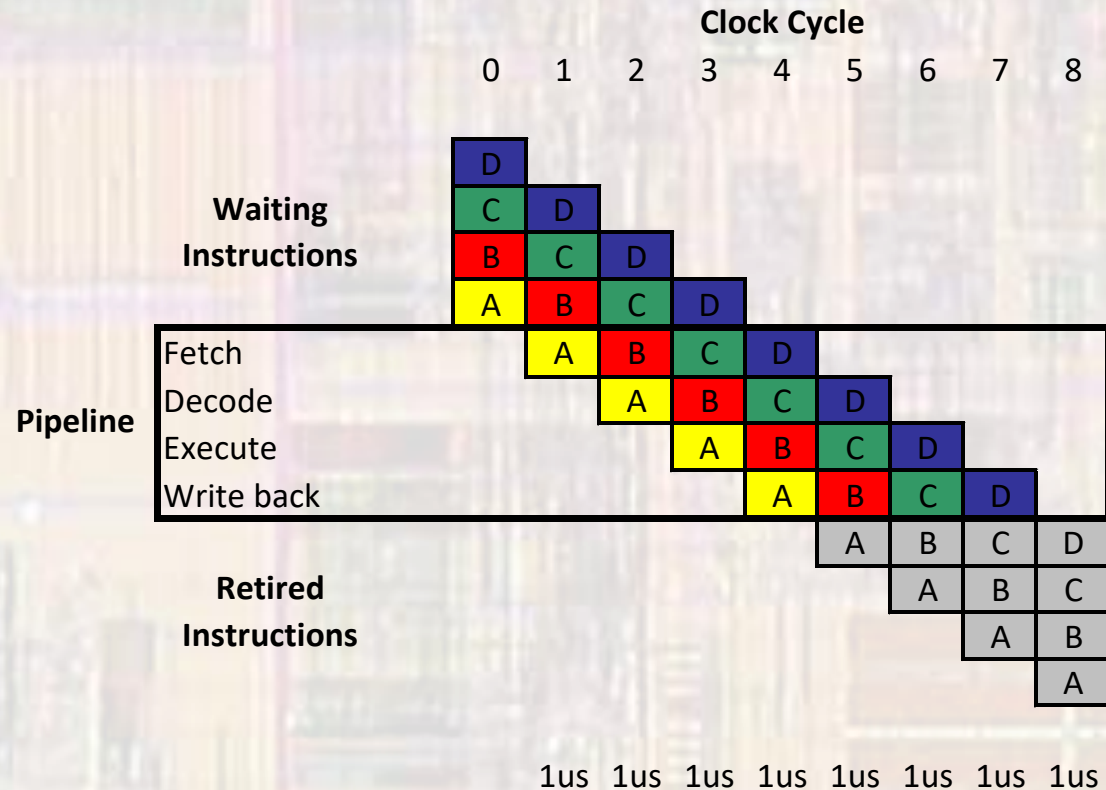
- Pipelining



Execute = fetch instruction, decode, execute, write back

Pipelining

- Pipelining
 - Break complex tasks into smaller chunks
 - Start the next instruction as soon as each subtask is complete



Pipelining

- Pipeline Performance

- Pipelining does not reduce the time to execute an instruction
 - In fact – it usually increases the instruction execution time
- Pipelining does increase the instruction throughput

Time	1000					1000					1000				
IF/ID/EX/MEM/WB	1					2					3				

Time	200	200	200	200	200	200	200	200	200	200	200	200	200	200	200
IF	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ID		1	2	3	4	5	6	7	8	9	10	11	12	13	14
EX			1	2	3	4	5	6	7	8	9	10	11	12	13
MEM				1	2	3	4	5	6	7	8	9	10	11	12
WB					1	2	3	4	5	6	7	8	9	10	11

Pipelining

- Pipeline Performance
 - Non-pipelined
 - 1M Instructions $\rightarrow 1 \times 10^9$ units of time
 - Pipelined (5 stage)
 - 1M Instructions $\rightarrow 2 \times 10^8 + 5 \times 200 \approx 2 \times 10^8$ units of time
- Overall throughput improvement of 5x

Pipelining

- Pipeline Performance
 - Non-pipelined
 - 1M Instructions $\rightarrow 1 \times 10^9$ units of time
 - Pipelined (5 stage w/20% penalty per stage)
 - 1M Instructions $\rightarrow 2.4 \times 10^8 + 5 * 2240 \approx 2.4 \times 10^8$ units of time
 - Overall throughput improvement of 4.2x

Pipelining

- Pipeline Performance

- Pipeline stages typically do not all take the same amount of time

Stage	IF	ID/RR	EX	MEM	WB
Delay	200ps	100ps	200ps	200ps	100ps

- Non-pipelined instruction throughput = 1 inst / 800ps
- Pipelined (5 stage) instruction throughput = 1 inst / 200ps
- Overall throughput improvement of 4x

Pipelining

- Pipeline Performance
 - Not all instruction need to use all the pipeline stages

Instruction	IF	ID/RR	EX	MEM	WB
ADD	X	X	X		X
OR	X	X	X		X
LW	X	X	X	X	X
SW	X	X	X	X	
BEQ	X	X	X		

Pipelining

- MIPS Pipeline Considerations
 - All instructions are 32-bits
 - Easier to fetch and decode in one cycle
 - Few and regular instruction formats
 - R, I, J
 - Can decode and read registers in one step - **why?**
 - Load/store addressing
 - Can calculate address in 3rd stage, access memory in 4th stage
 - Alignment of memory operands
 - Memory access takes only one cycle

Pipelining

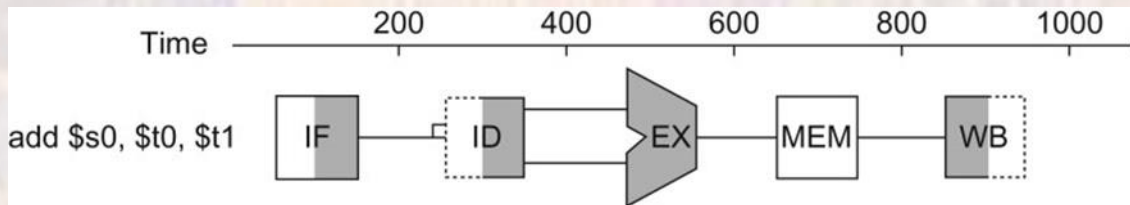
- Pipeline Operation
 - The program memory, register file and data memory can each be read or written
 - We will use the following convention
 - Writes occur in the first half of the clock cycle
 - Reads occur in the second half of the clock cycle

How would we implement this?

Pipelining

- Pipeline Operation

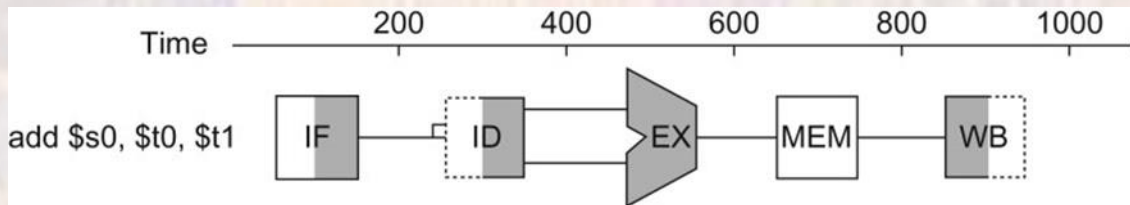
WB actually occurs in the first half of the next clock cycle



- In this example
 - Reads are done from program memory and the register file
 - Write is done on the register file
 - The data memory is not used
 - The ALU executes

Pipelining

- Pipeline Operation
- What about this operation should concern us?



- The ID (register read) and the WB access the same resource
 - This creates a potential for conflicts

Pipelining

- Pipeline Hazards
 - Hazards are conditions where the next instruction cannot perform its assigned pipeline action in the next clock cycle
 - 3 types
 - Structural
 - Data
 - Control

Pipelining

- Structural Hazards
 - These hazards result from a **resource** conflict
 - Classic case is Harvard vs. vonNeuman memory architectures
 - vonNeuman architectures share a single memory for program and data
 - A lw or sw command requires access to data memory to load or store the data value
 - It would not be possible to fetch the appropriate instruction during this clock cycle since the memory would be in use
 - The IF would be stalled and a “bubble” would be created in the pipeline

Pipelining

- Structural Hazards
 - vonNeuman memory architecture

Time	200	200	200	200	200	200	200	200	200	200	200	200	200	200	200
IF	LW	2	3	Stall	4	5	6	7	8	9	10	11	12	13	14
ID		LW	2	3	bubble	4	5	6	7	8	9	10	11	12	13
EX			LW	2	3	bubble	4	5	6	7	8	9	10	11	12
MEM				LW	2	3	bubble	4	5	6	7	8	9	10	11
WB					LW	2	3	bubble	4	5	6	7	8	9	10

data memory access prevents a concurrent instruction fetch

Pipelining

- Structural Hazards
 - MIPS implementation is designed to avoid structural hazards

Pipelining

- Data Hazards
 - These hazards result from a **dependence** of one instruction on another instruction still in the pipeline
 - Consider the following code snippet

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

- The value of \$s0 is needed to perform the subtraction

Pipelining

- Data Hazards

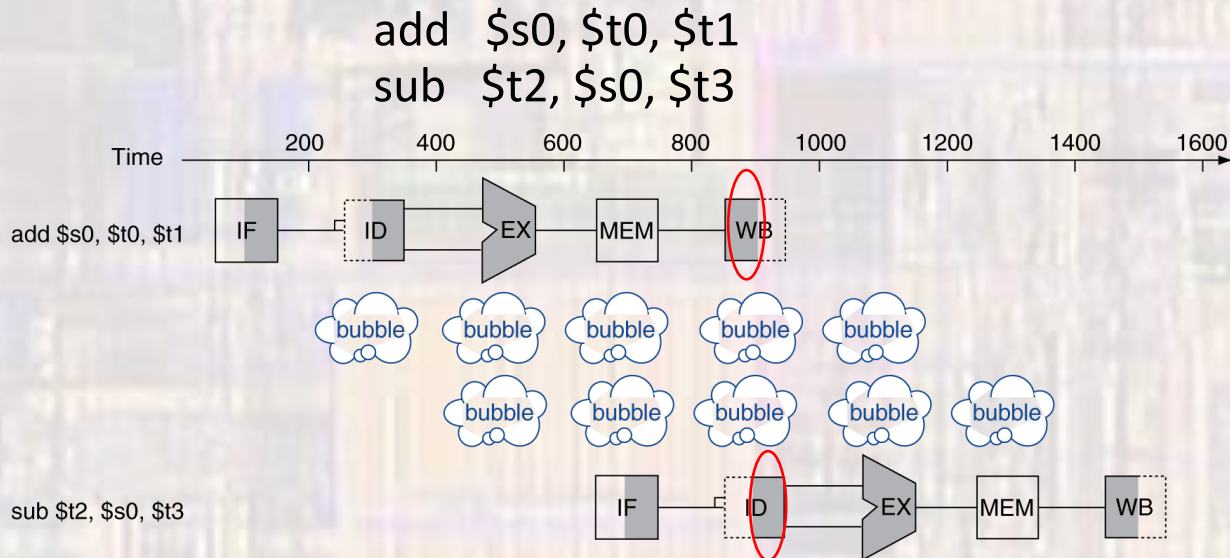
```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```

Time	200	200	200	200	200	200	200	200	200	200	200	200	200	200	200
IF	add	sub	sub	sub	3	4	5	6	7	8	9	10	11	12	13
ID		add	stall	stall	sub	3	4	5	6	7	8	9	10	11	12
EX			add	bubble	bubble	sub	3	4	5	6	7	8	9	10	11
MEM				add	bubble	bubble	sub	3	4	5	6	7	8	9	10
WB					add	bubble	bubble	sub	3	4	5	6	7	8	9

- 2 clock cycle bubbles are created
- It would be 3 bubbles – except we can take advantage of our convention
 - writes occur in the first half of the clock cycle
 - reads occur in the second half of the clock cycle
 - the WB occurs during the same clock cycle as the register read

Pipelining

- Data Hazards



- 2 clock cycle bubbles are created
- It would be 3 bubbles – except we can take advantage of our convention
 - writes occur in the first half of the clock cycle
 - reads occur in the second half of the clock cycle
 - the WB occurs during the same clock cycle as the register read

Pipelining


- Data Hazards

- In many cases the compiler can avoid a data hazard

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
or  $s2, $t0, $t1
and $s3, $t0, $t3
add $s4, $t1, $t3
```

```
add $s0, $t0, $t1
or  $s2, $t0, $t1
and $s3, $t0, $t3
add $s4, $t1, $t3
sub $t2, $s0, $t3
```

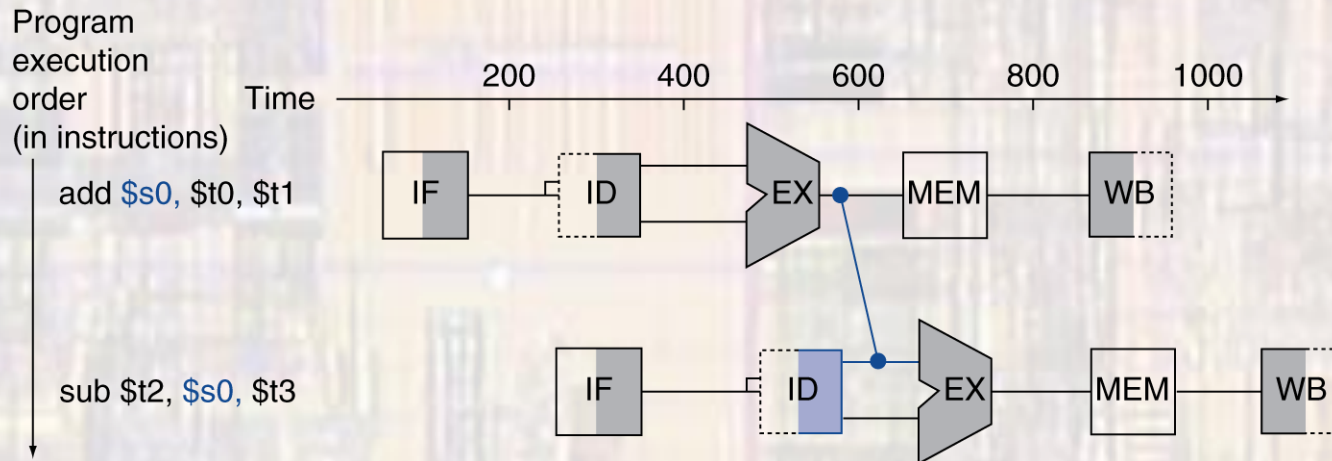
re-order the instruction to remove the hazard condition



Pipelining

- Data Hazards

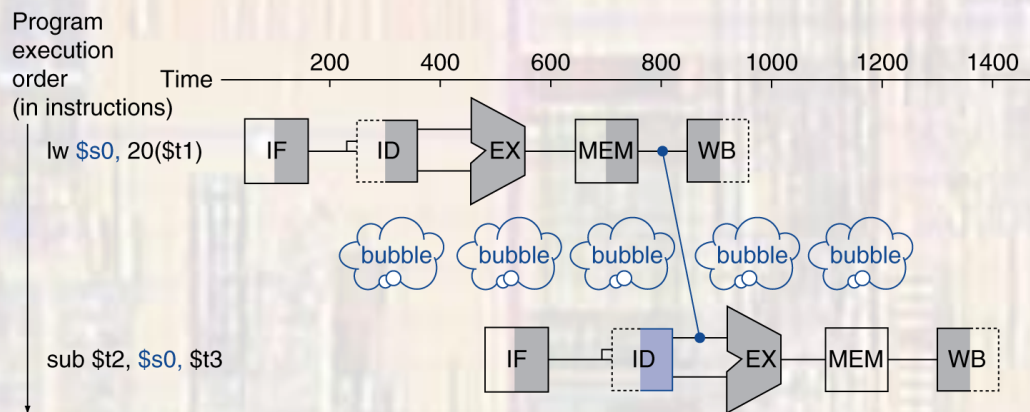
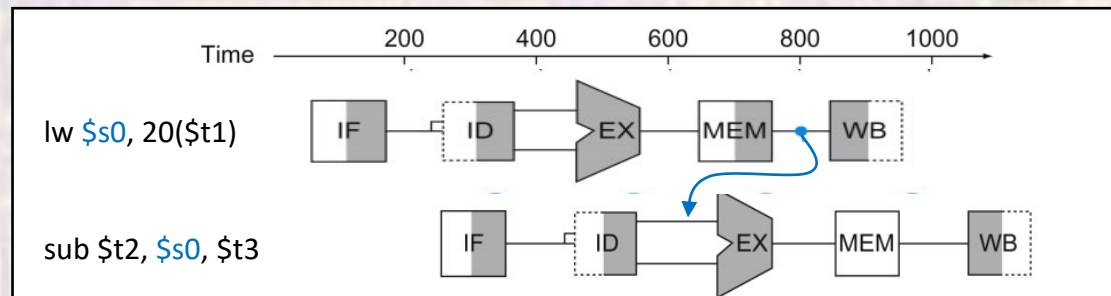
- Hardware can also be used to avoid data hazards
 - called forwarding or bypassing
 - provide the needed data as soon as it is valid
 - requires extra circuitry



Pipelining

- Data Hazards

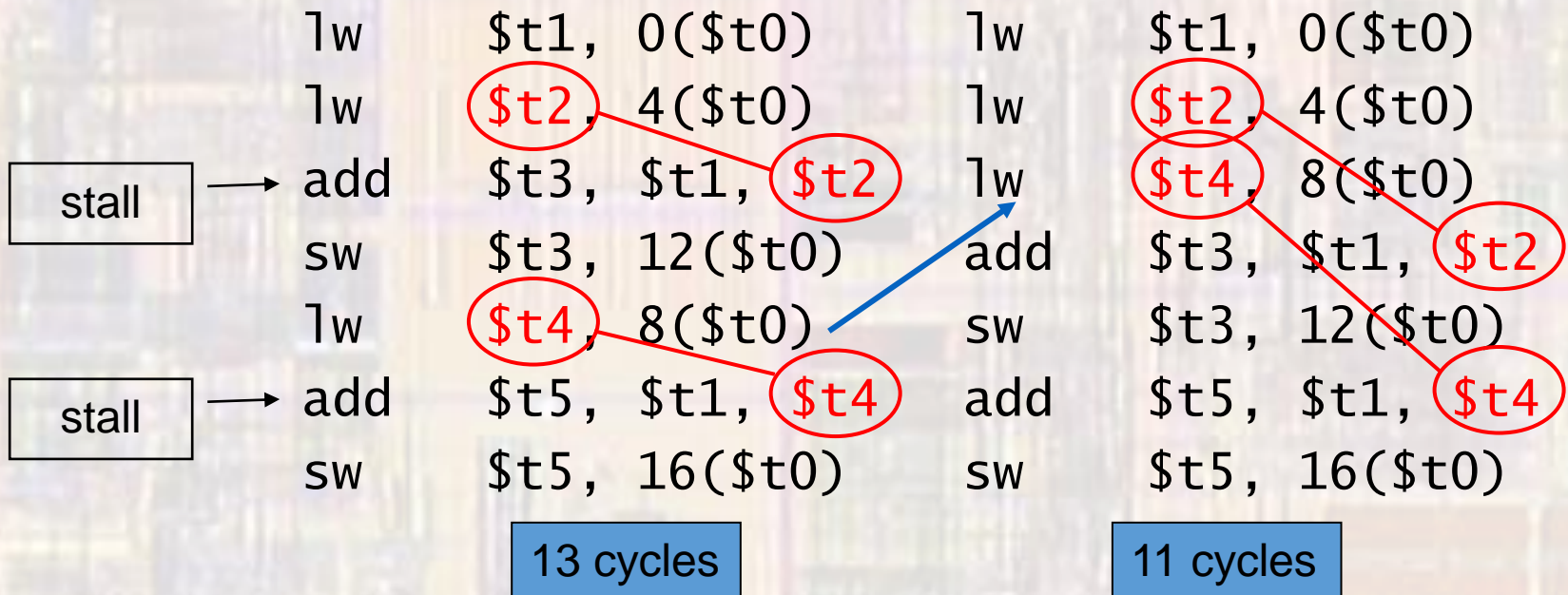
- Hardware cannot avoid all data hazards
 - cannot go backwards in time !



Pipelining

- Data Hazards

- Forwarding plus compiler optimizations can avoid additional data hazards



Pipelining

- Control Hazards
 - These hazards result from making a **decision** while other instructions continue to progress through the pipeline
 - Branch instructions are the most common example
 - don't know whether to load the next instruction or not
 - three approaches
 - stall
 - predict
 - delay

Pipelining

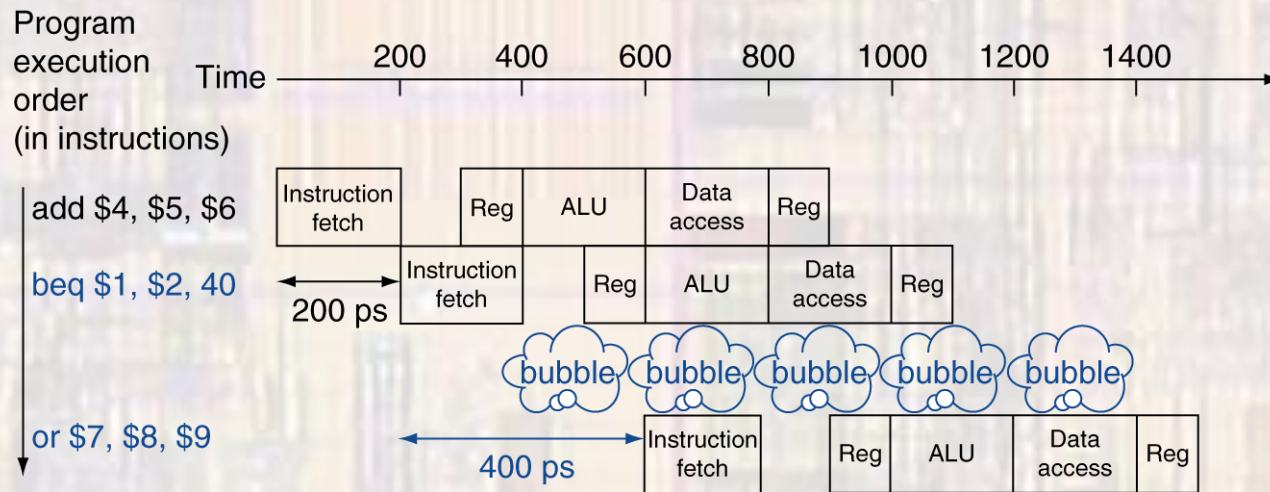
- Control Hazards - stall
 - Do not load the next instruction into the pipeline

Time	200	200	200	200	200	200	200	200	200	200	200	200	200	200	200
IF	add	beq	3	3	8	9	10	11	12	13	14	15	16	17	18
ID		add	beq	stall	stall	8	9	10	11	12	13	14	15	16	17
EX			add	beq	bubble	bubble	8	9	10	11	12	13	14	15	16
MEM				add	beq	bubble	bubble	8	9	10	11	12	13	14	15
WB					add	beq	bubble	bubble	8	9	10	11	12	13	14

- during decode – know you have a branch
- during execute – know if taking branch or not
 - PC will be updated
- Next cycle – fetch the next instruction based on PC value

Pipelining

- Control Hazards - stall
 - Even if you add circuitry to detect the branch and update the PC all during the decode – can't avoid a stall

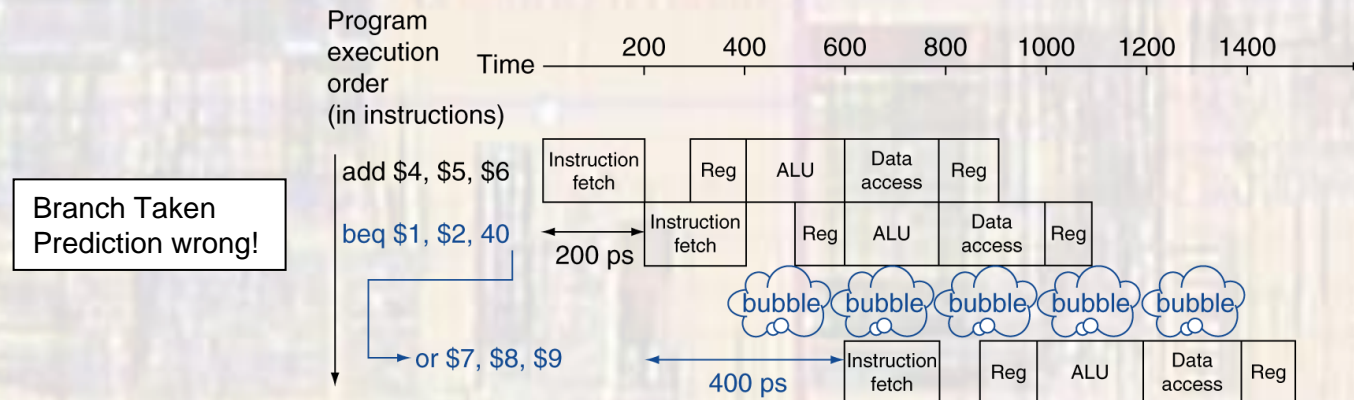
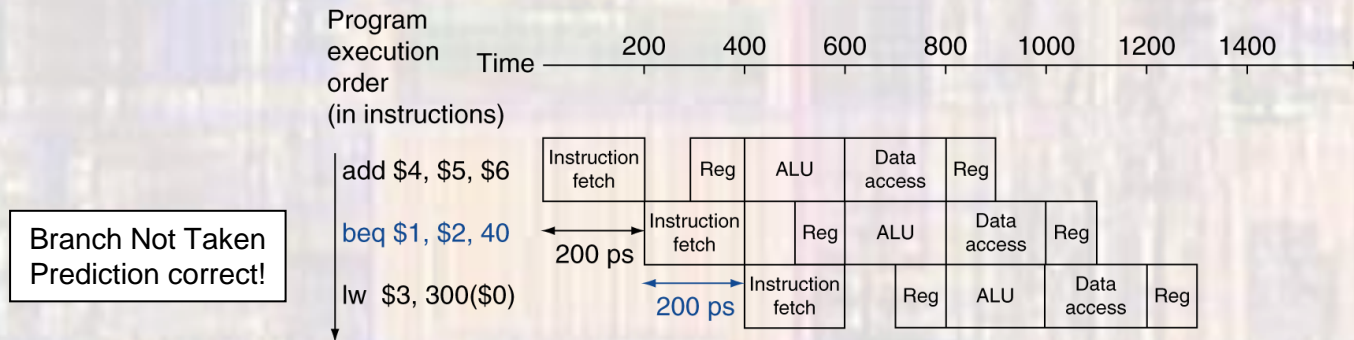


Pipelining

- Control Hazards - predict
 - Many algorithms
 - Simplest – assume branch will not be taken
 - no penalty if correct
 - stall only when wrong

Pipelining

- Control Hazards – predict
 - Predict branch not taken



Pipelining

- Control Hazards - predict
 - Static Branch Prediction
 - Predict backward branches - taken
 - Predict forward branches – not taken
 - Looping code
 - executes the loop 100 times
 - jumps out of the loop 1 time
 - Dynamic Branch Prediction
 - Keep track of recent branch behavior (for each branch)
 - Assume recent behavior will continue
 - When wrong – clear history and start over
 - Hardware intensive

Pipelining

- Control Hazards - delay
 - Delayed Decision
 - Pipeline always executes the instruction immediately after the branch
 - The branch then executes (only 1 cycle delay allowed)
 - Requires the next instruction to be **independent** of the branch decision
 - Compiler is designed to set this up

Pipelining

- Control Hazards - delay

- Delayed Decision (assume HW to limit bubble to 1 cycle)

```

add $t0,$t1,$t2
beq $t1,$t2,-30
...
    
```

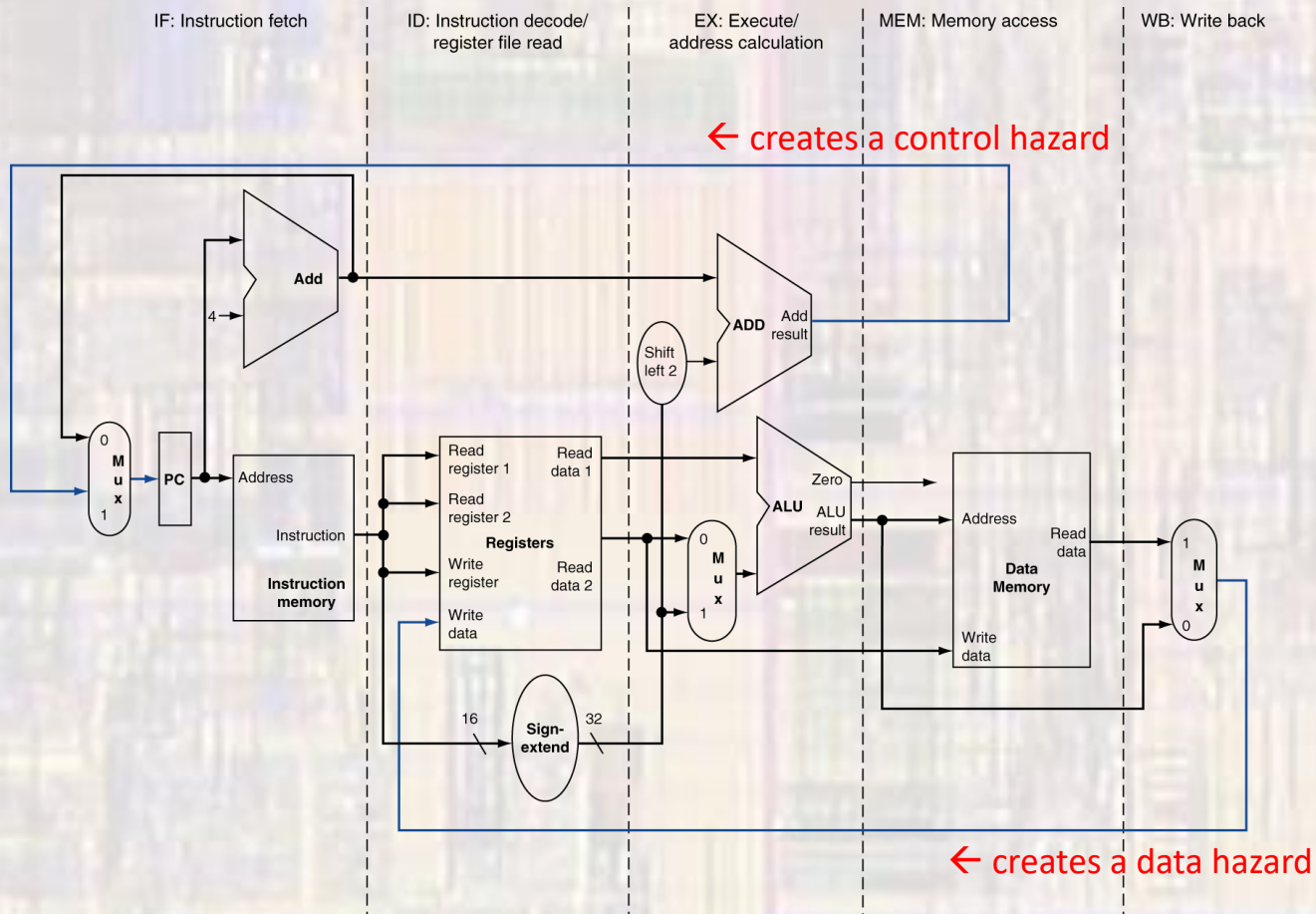
Time	200	200	200	200	200	200	200	200	200	200	200	200	200	200	200
IF	add	beq	3	8	9	10	11	12	13	14	15	16	17	18	19
ID		add	beq	stall	8	9	10	11	12	13	14	15	16	17	18
EX			add	beq	bubble	8	9	10	11	12	13	14	15	16	17
MEM				add	beq	bubble	8	9	10	11	12	13	14	15	16
WB					add	beq	bubble	8	9	10	11	12	13	14	15

re-order

Time	200	200	200	200	200	200	200	200	200	200	200	200	200	200	200
IF	beq	add	8	9	10	11	12	13	14	15	16	17	18	19	20
ID		beq	add	8	9	10	11	12	13	14	15	16	17	18	19
EX			beq	add	8	9	10	11	12	13	14	15	16	17	18
MEM				beq	add	8	9	10	11	12	13	14	15	16	17
WB					beq	add	8	9	10	11	12	13	14	15	16

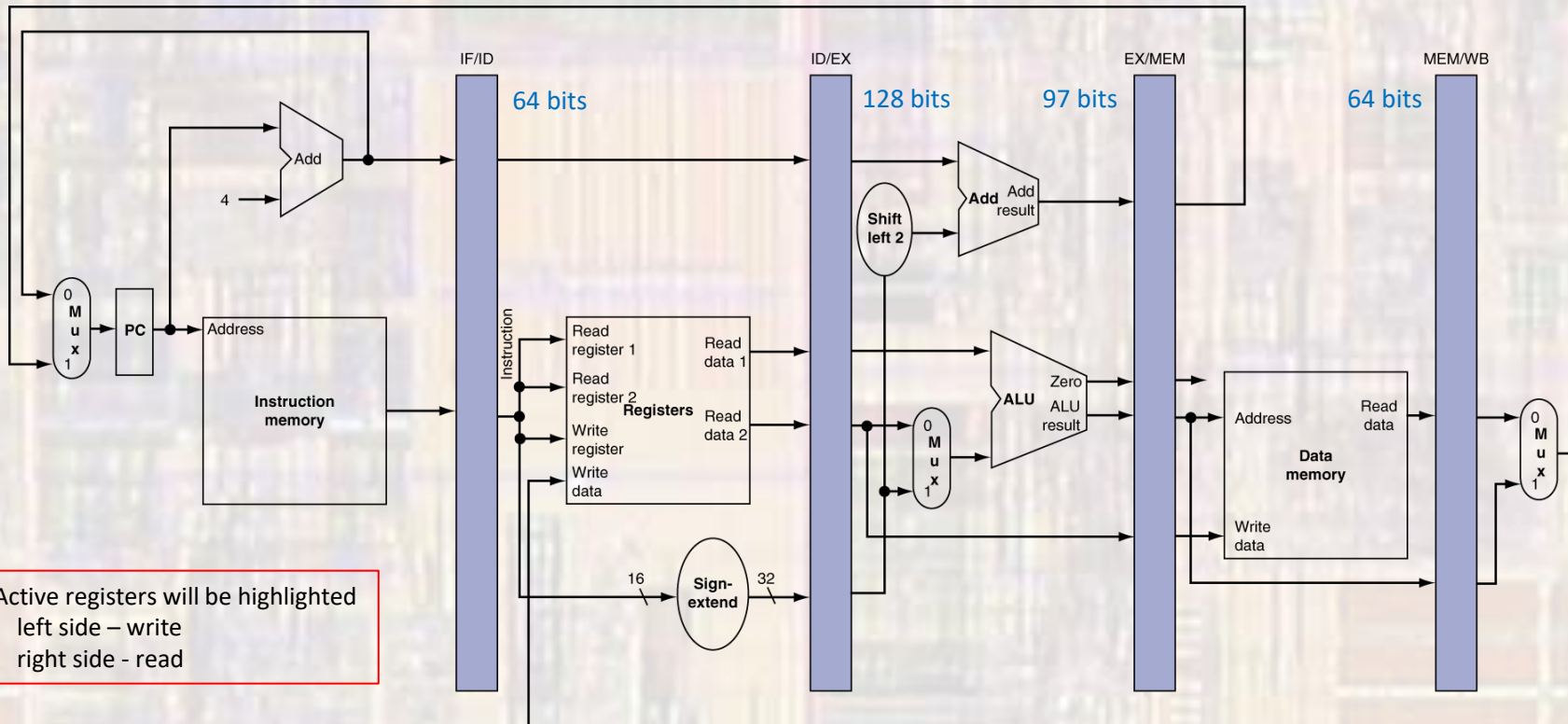
Pipelining

- Mapping the datapath to a pipeline



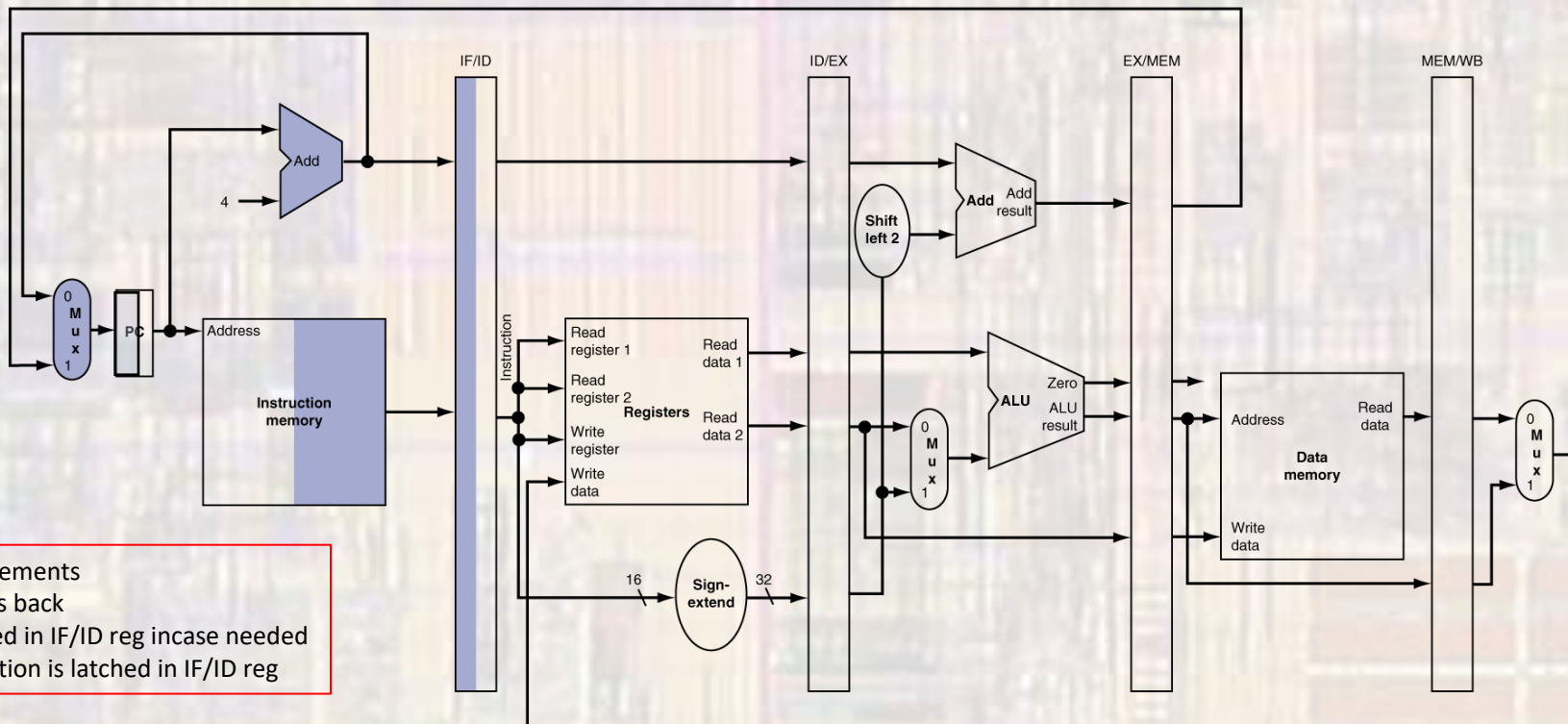
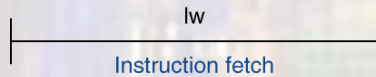
Pipelining

- Mapping the datapath to a pipeline
 - Registers are required to hold intermediate values between stages



Pipelining

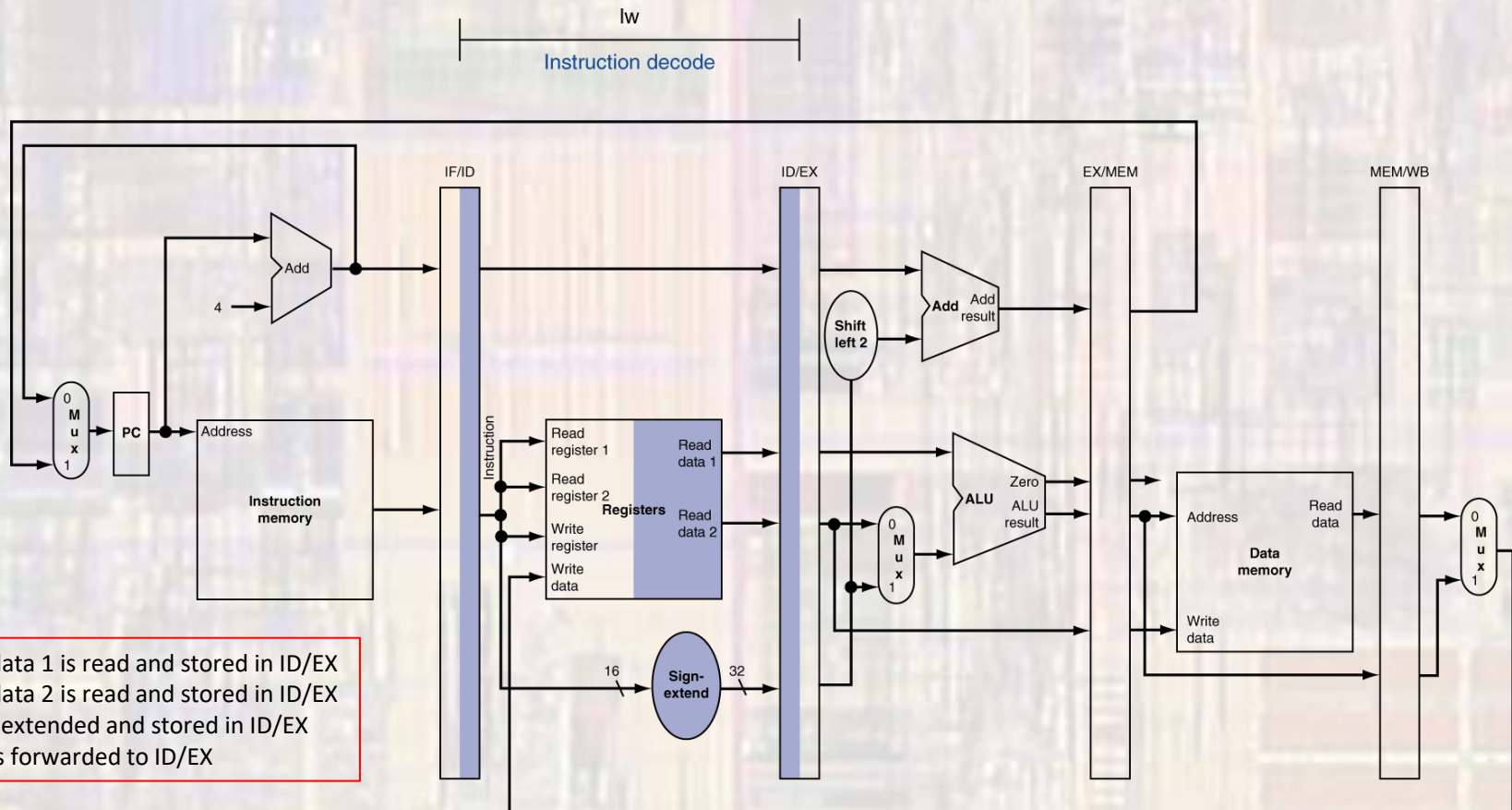
- Mapping the datapath to a pipeline
 - lw instruction - IF



PC increments
feeds back
stored in IF/ID reg incase needed
Instruction is latched in IF/ID reg

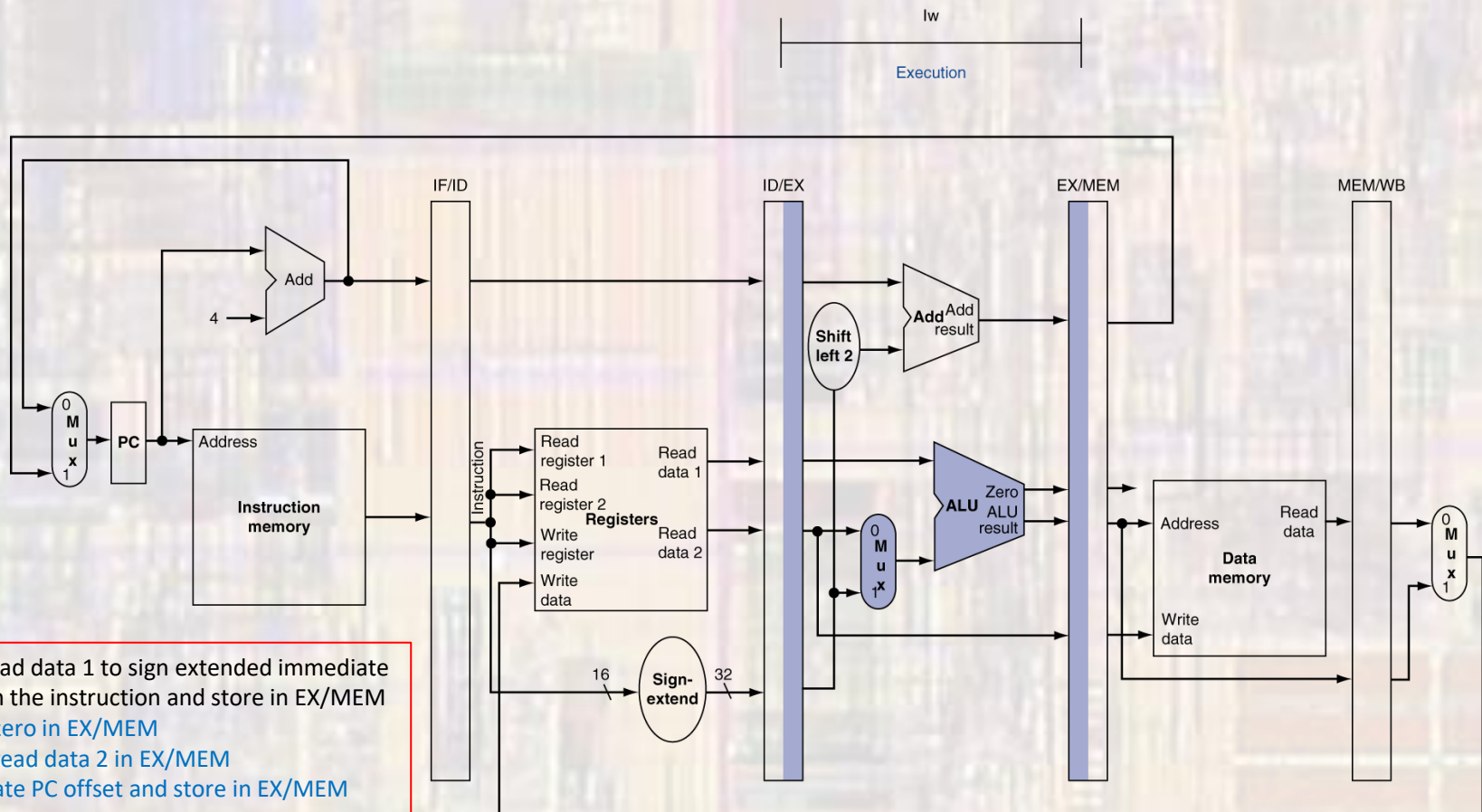
Pipelining

- Mapping the datapath to a pipeline
- lw instruction – ID (instruction decode and register read)



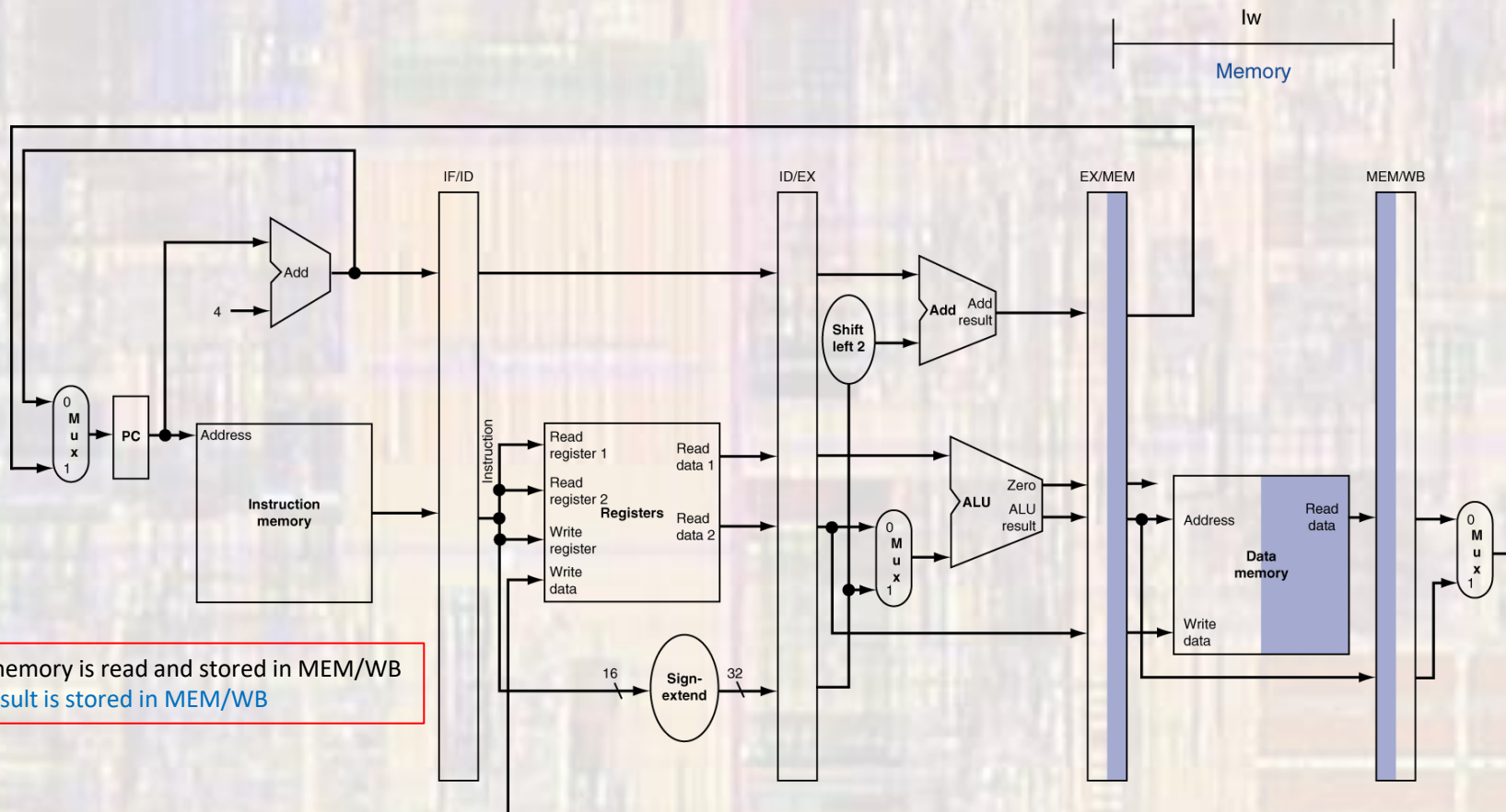
Pipelining

- Mapping the datapath to a pipeline
- lw instruction – EX



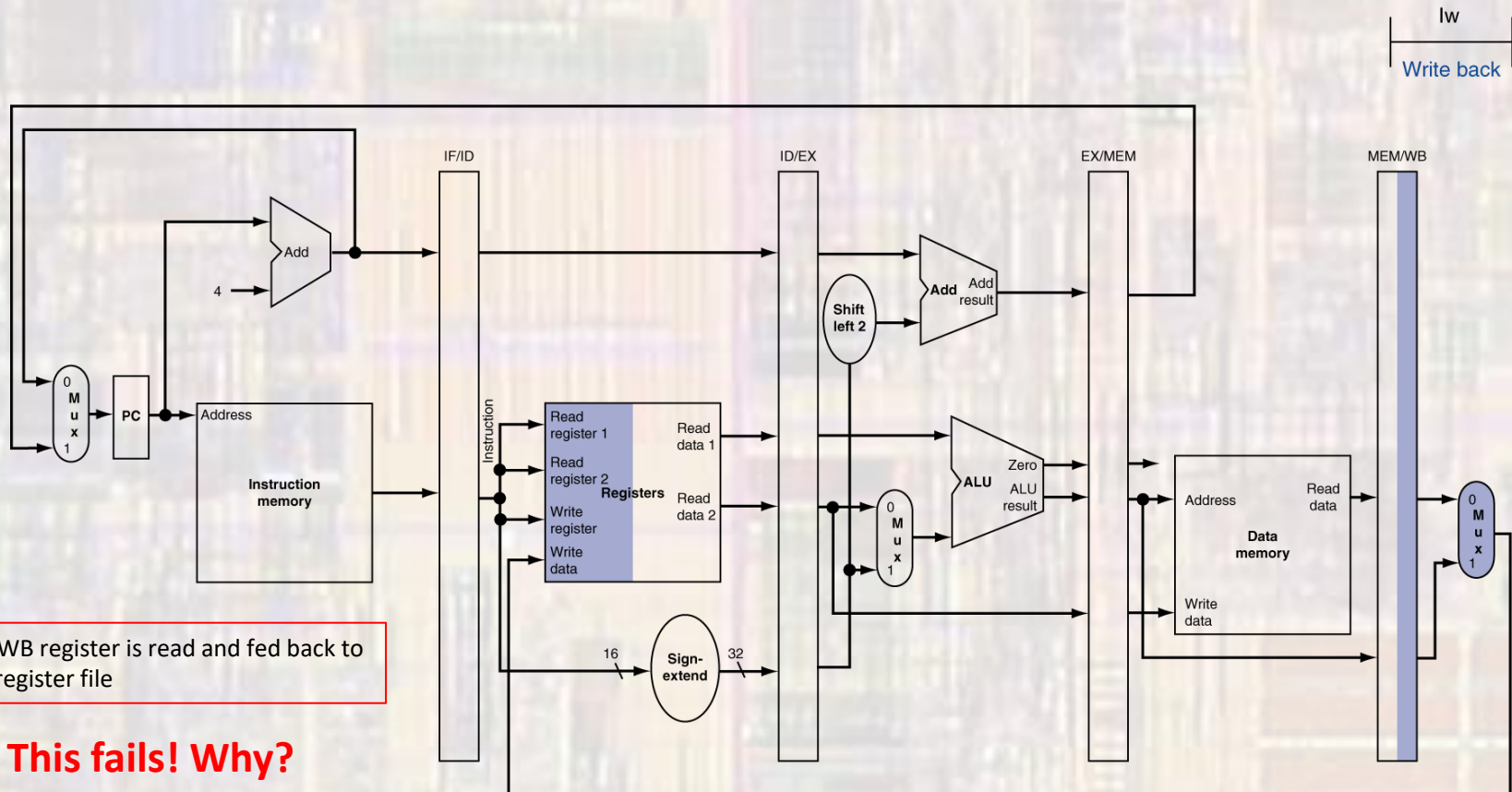
Pipelining

- Mapping the datapath to a pipeline
 - lw instruction – MEM



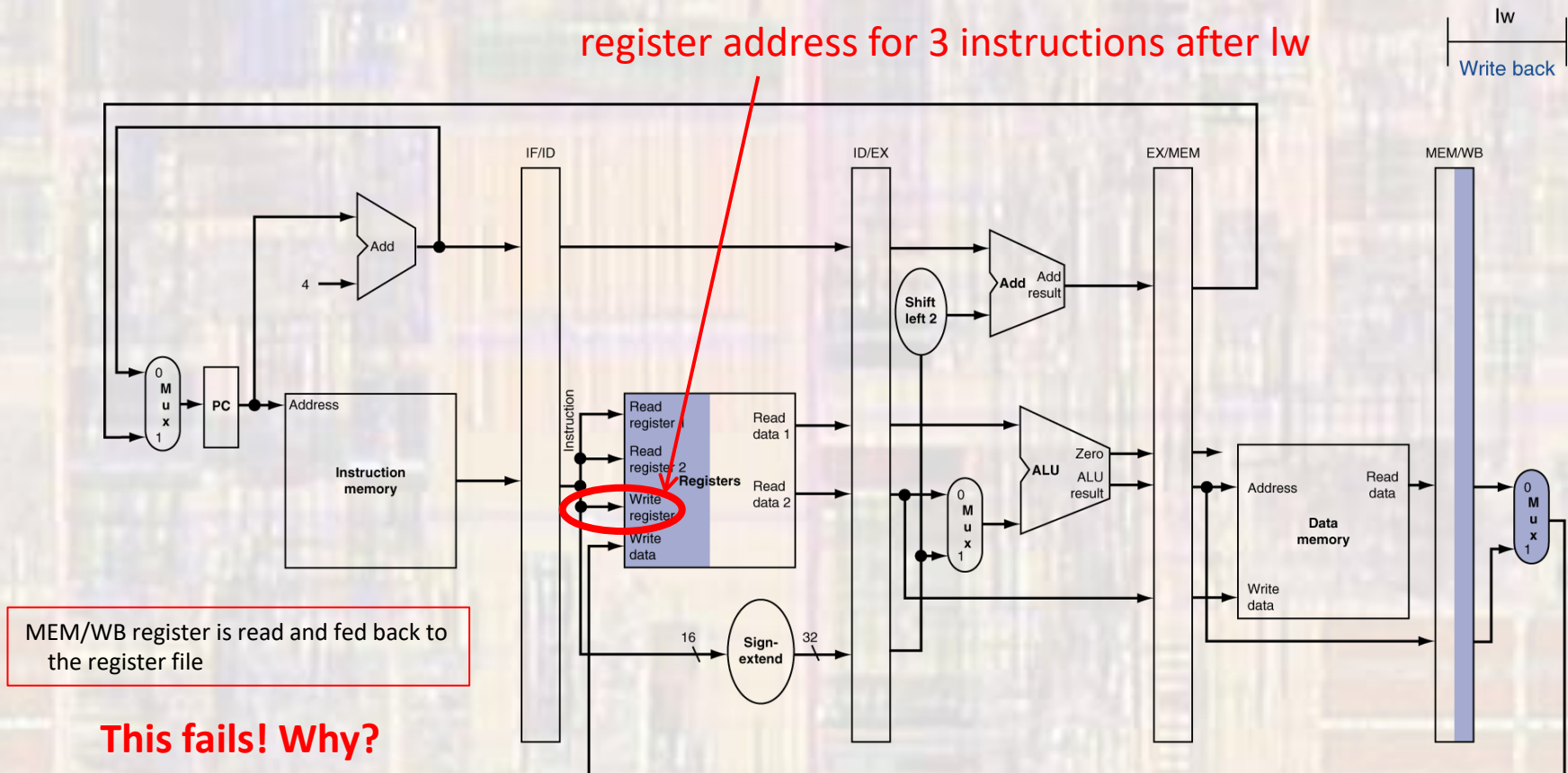
Pipelining

- Mapping the datapath to a pipeline
 - lw instruction – WB



Pipelining

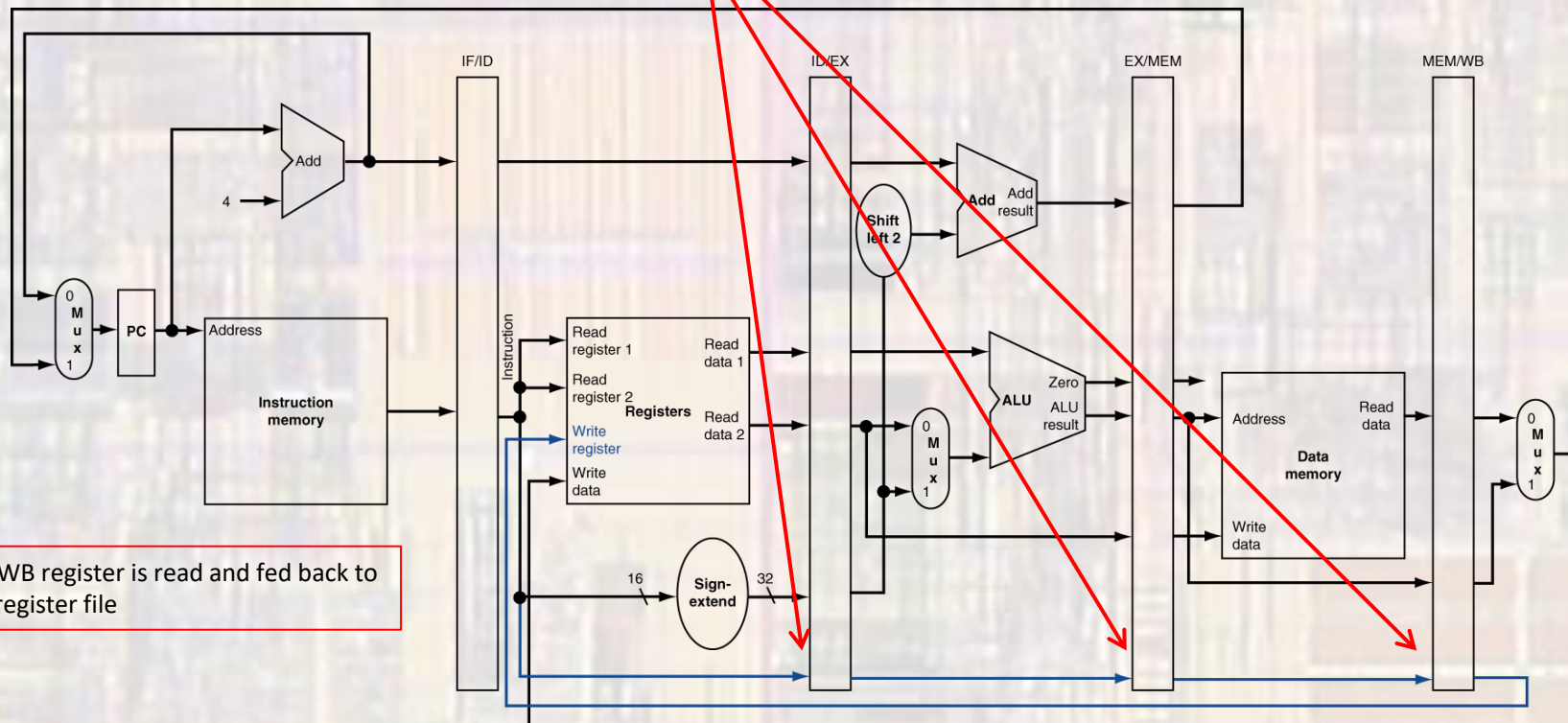
- Mapping the datapath to a a pipeline
 - lw instruction – WB



Pipelining

- Mapping the datapath to a pipeline
- lw instruction – WB

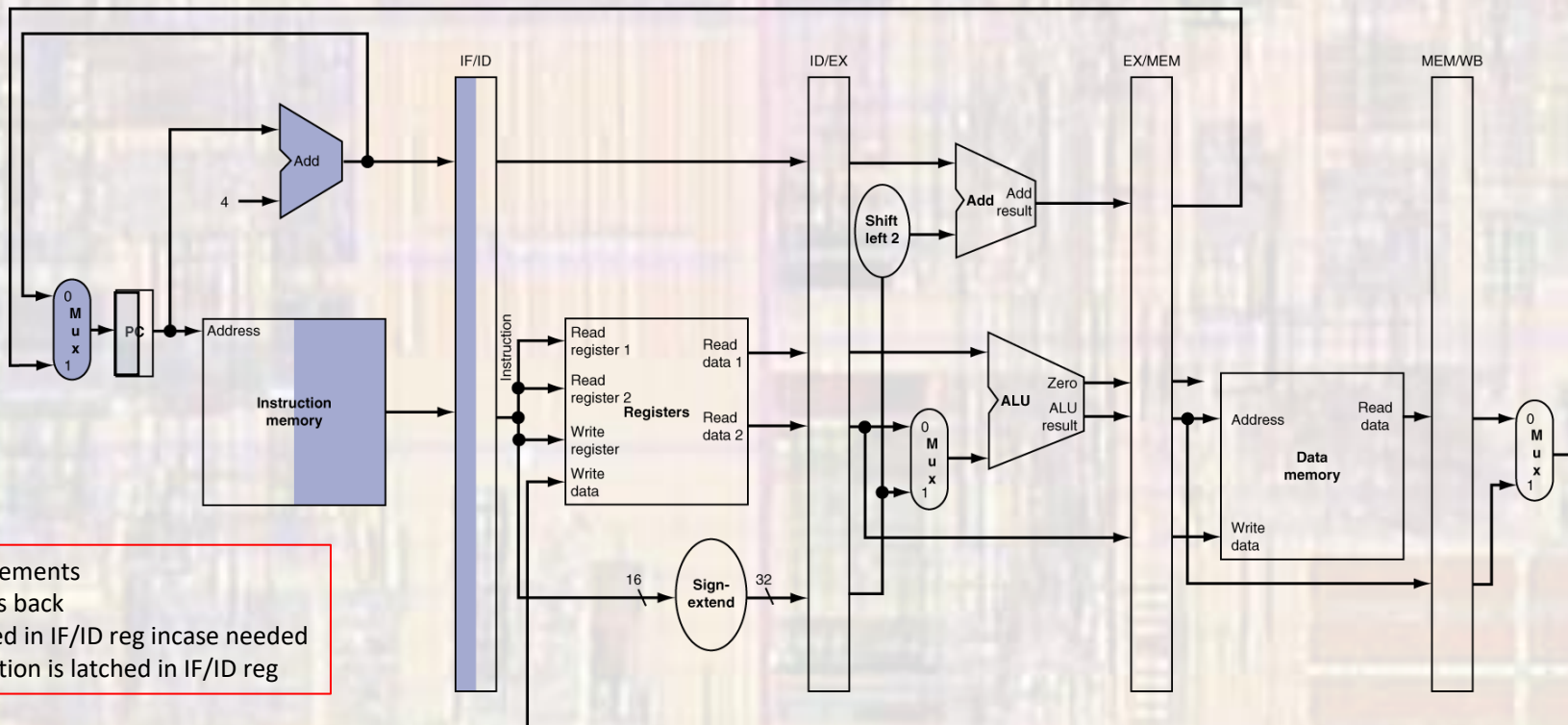
add write register value to ID/EX, EX/MEM, MEM/WB



MEM/WB register is read and fed back to the register file

Pipelining

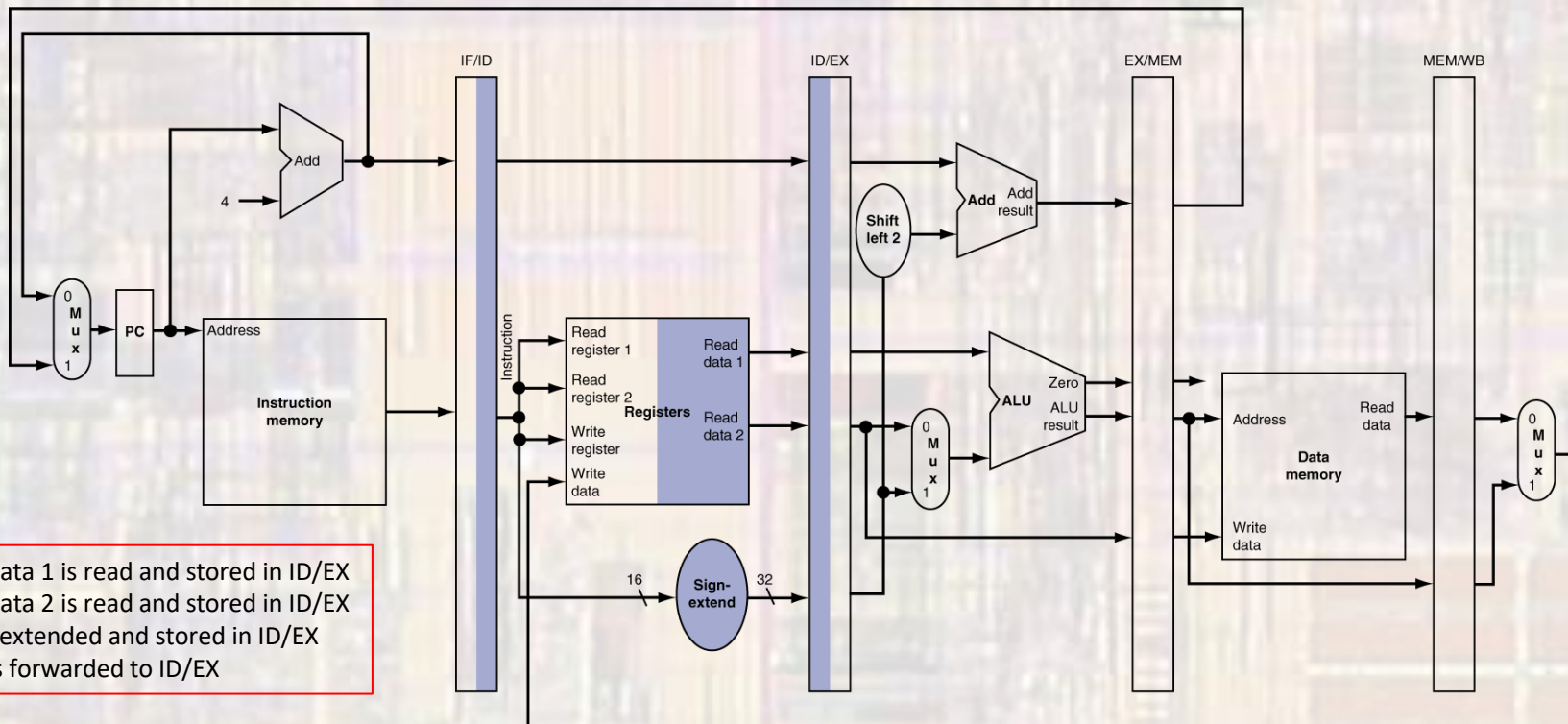
- Mapping the datapath to a pipeline
 - sw instruction - IF



PC increments
feeds back
stored in IF/ID reg incase needed
Instruction is latched in IF/ID reg

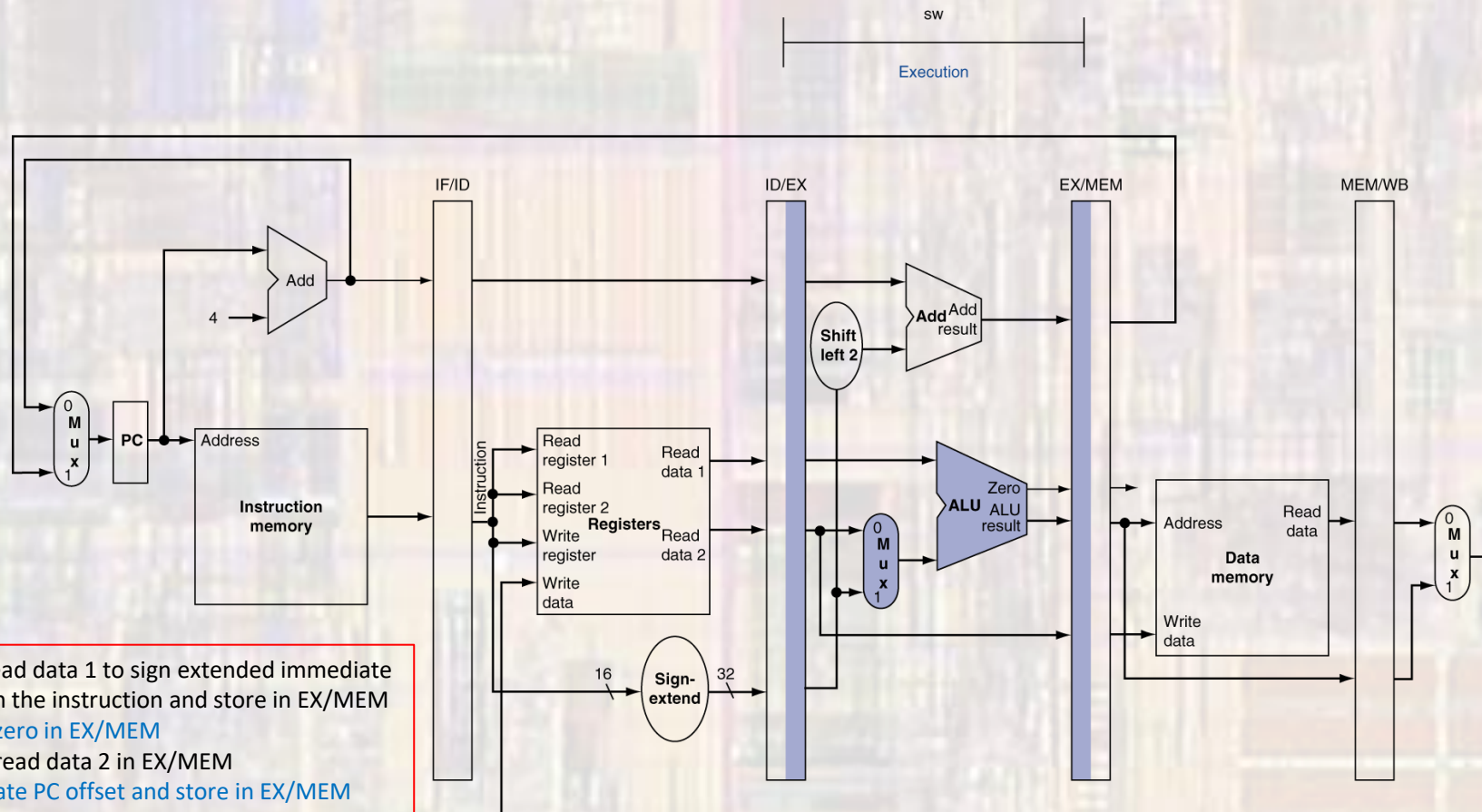
Pipelining

- Mapping the datapath to a pipeline
 - sw instruction – ID (instruction decode and register read)



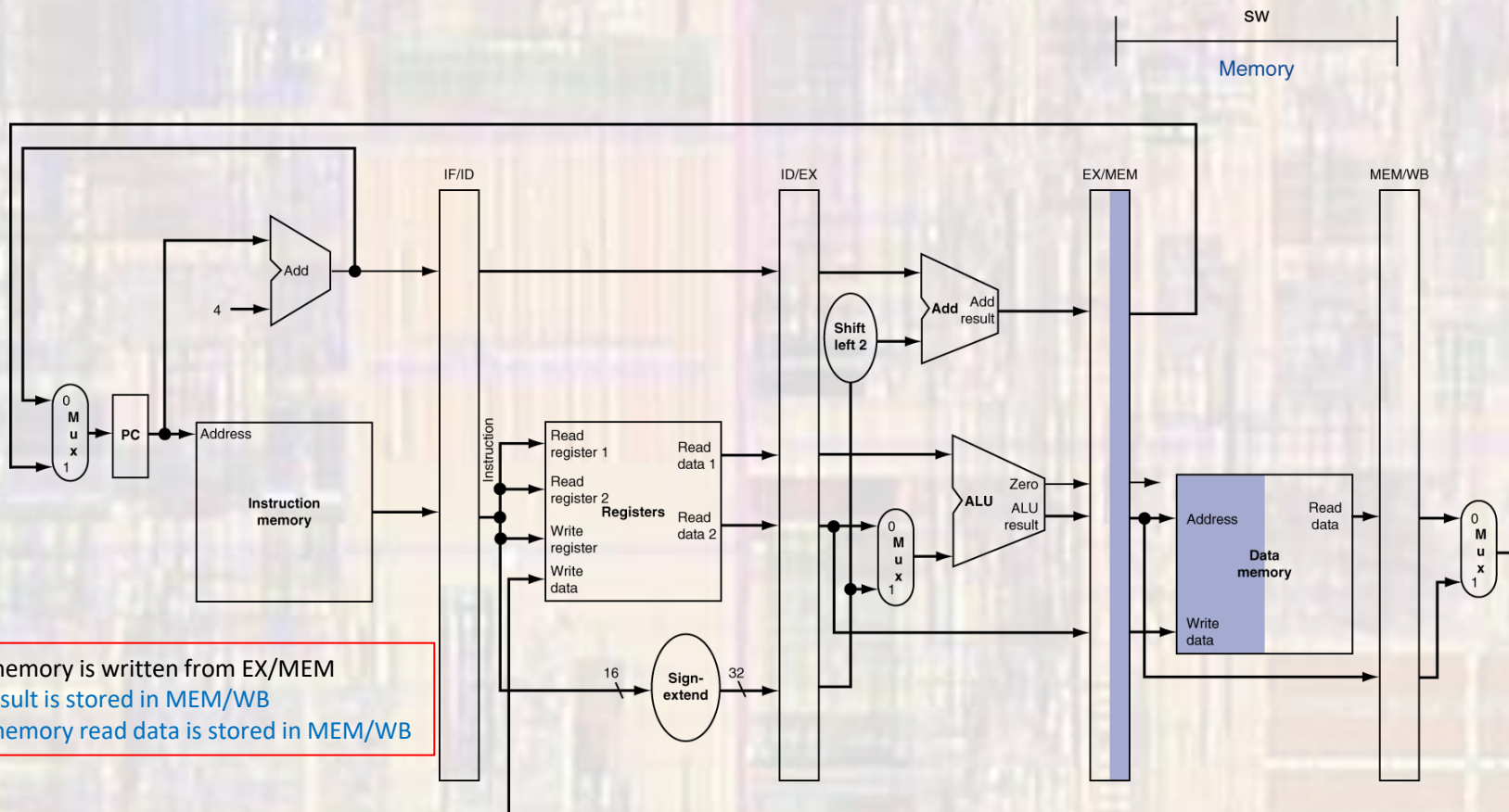
Pipelining

- Mapping the datapath to a pipeline
 - sw instruction – EX



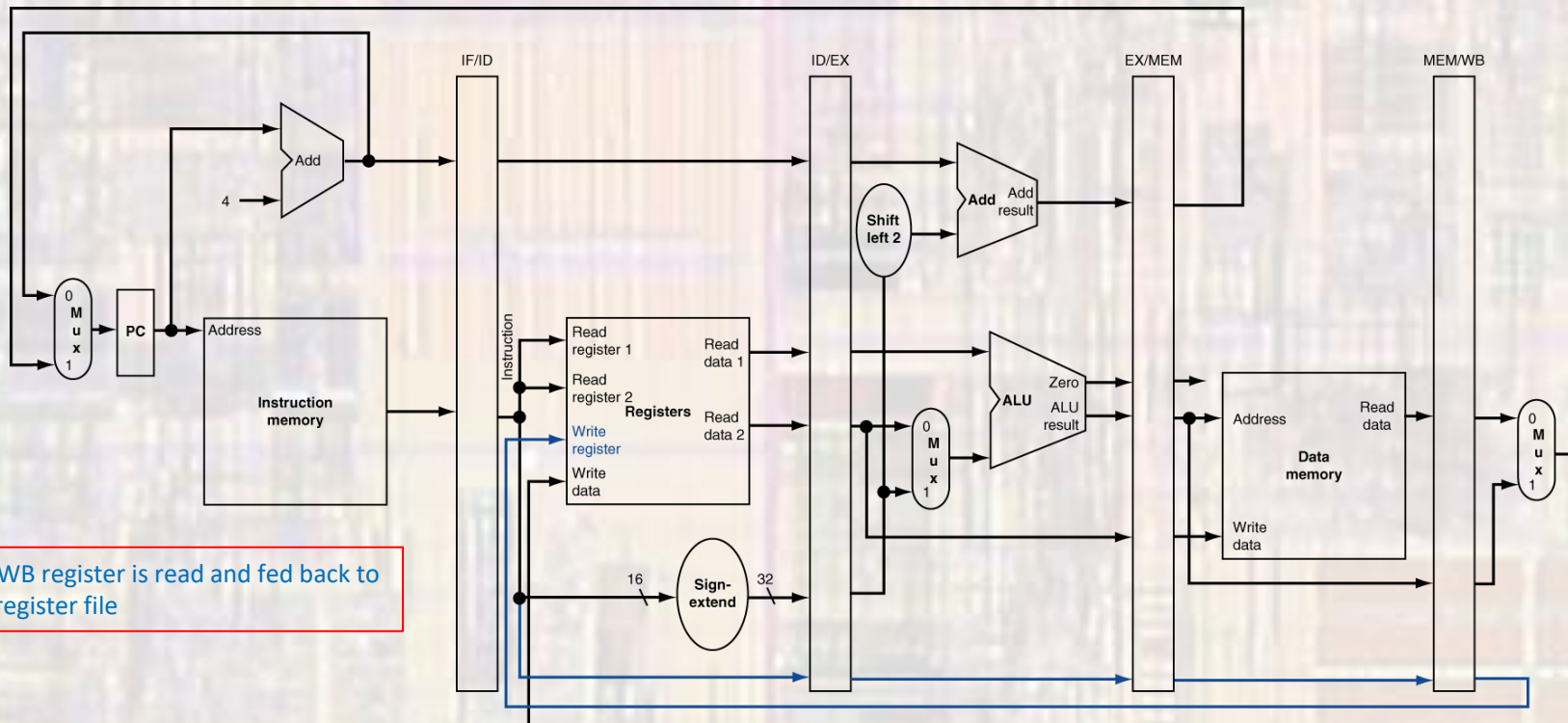
Pipelining

- Mapping the datapath to a pipeline
 - sw instruction – MEM



Pipelining

- Mapping the datapath to a pipeline
 - sw instruction – WB



Pipelining

- Pipeline Control
 - Many more control signals than we show
 - IF – all control lines operate the same way for all instructions
 - PC is read
 - Program Memory is read
 - PC is updated
 - ID - all control lines operate the same way for all instructions
 - Instruction is decoded
 - Registers are read

Pipelining

- Pipeline Control

- EX – executes or calculates an address
 - *RegDst* – choose between 2nd or 3rd register field for WB
 - *ALUOp* – L/S, Branch, or R-type
 - *ALUSrc* – selects Read Data 2 or sign extended immediate
- These are generated in the ID stage but used in the EX stage
 - Must pass them forward through the ID/EX register
- MEM – R/W to memory and selects the offset branch value
 - *MemRead* , *MemWrite* – memory read / write
 - *Branch* – combined with “zero” selects the offset branch to feed back to the PC
- These are generated in the ID stage but used in the MEM stage
 - Must pass them forward through the ID/EX register and the EX/MEM register

Pipelining

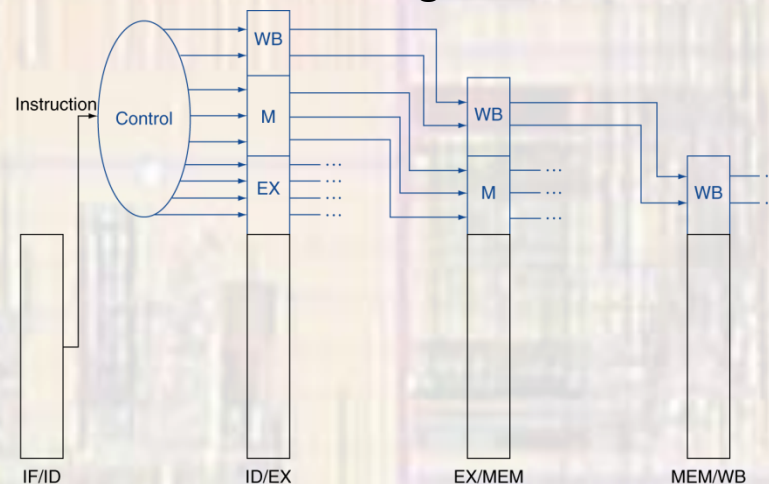
- Pipeline Control

- WB – chooses what to write back

- *RegWrite* – enables a write to the register file
- *MemtoReg* – choose between ALU output or memory output to feed back to the register file

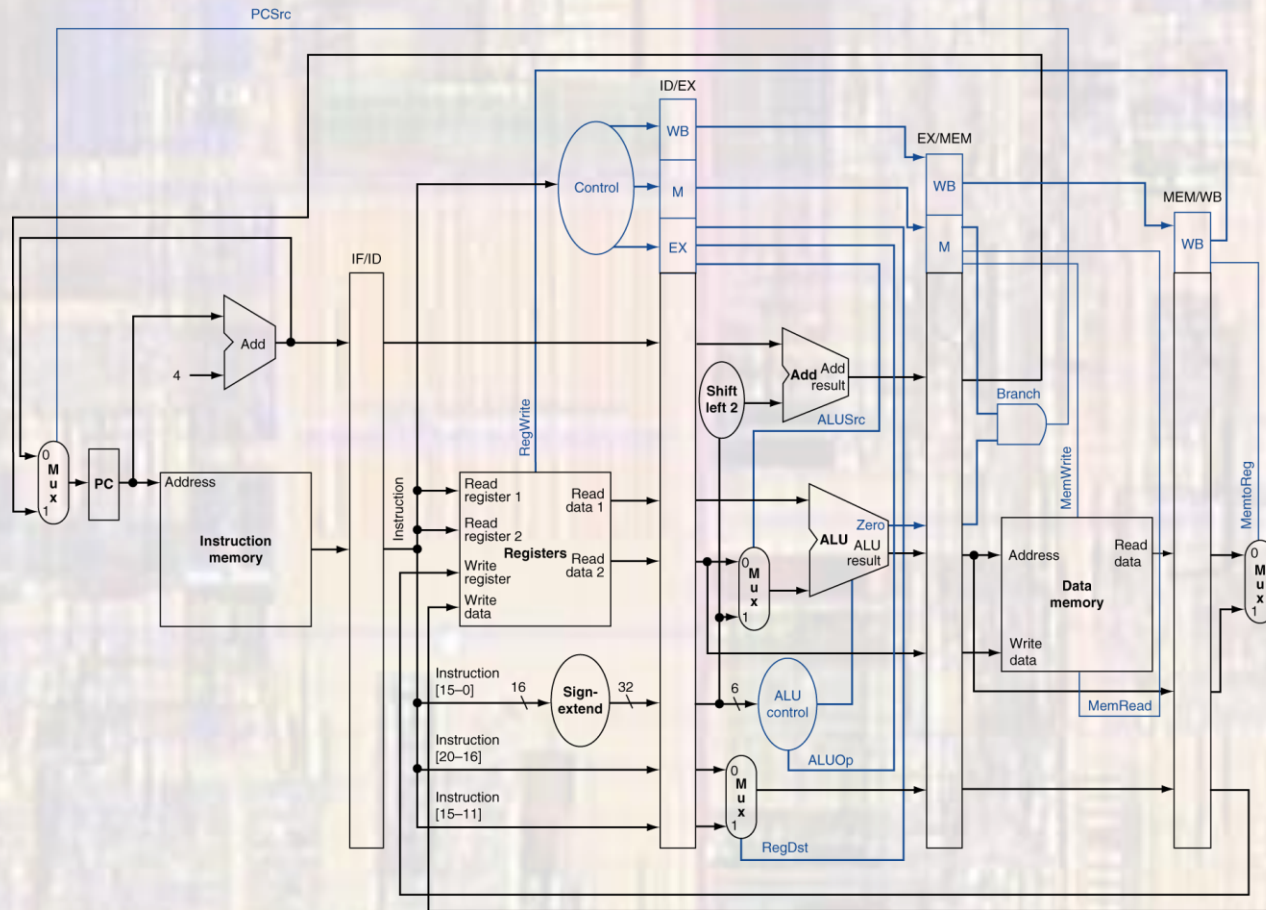
- These are generated in the ID stage but used in the MEM stage

- Must pass them forward through the ID/EX, EX/MEM and MEM/WB registers



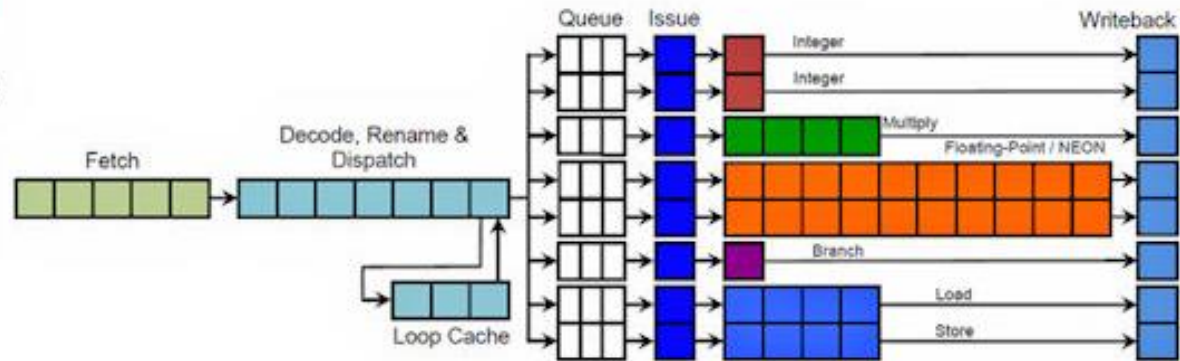
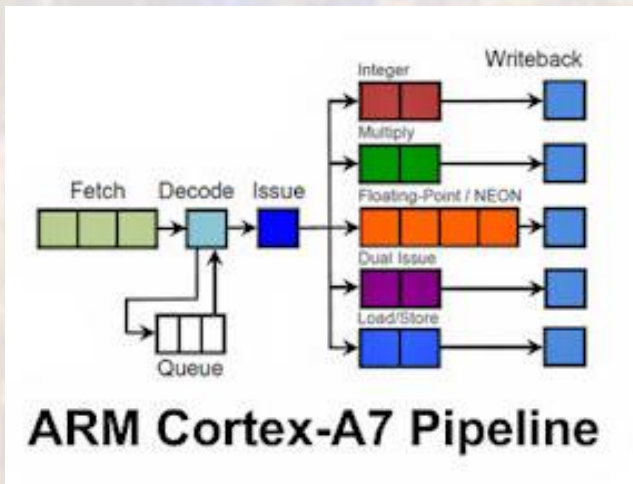
Pipelining

- Pipeline Control



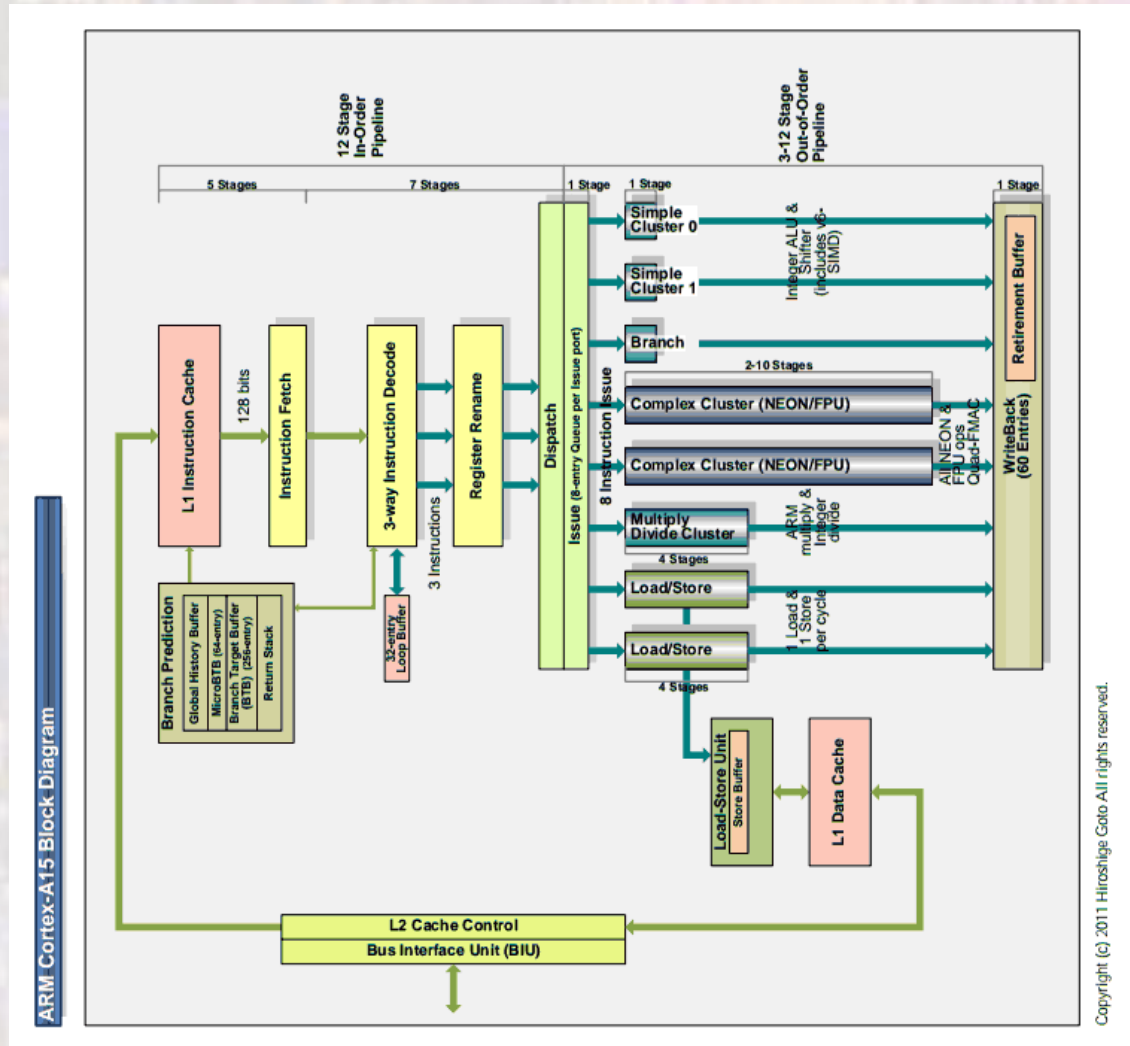
Pipelining

- Superscalar
 - Parallelism at the micro-architecture level



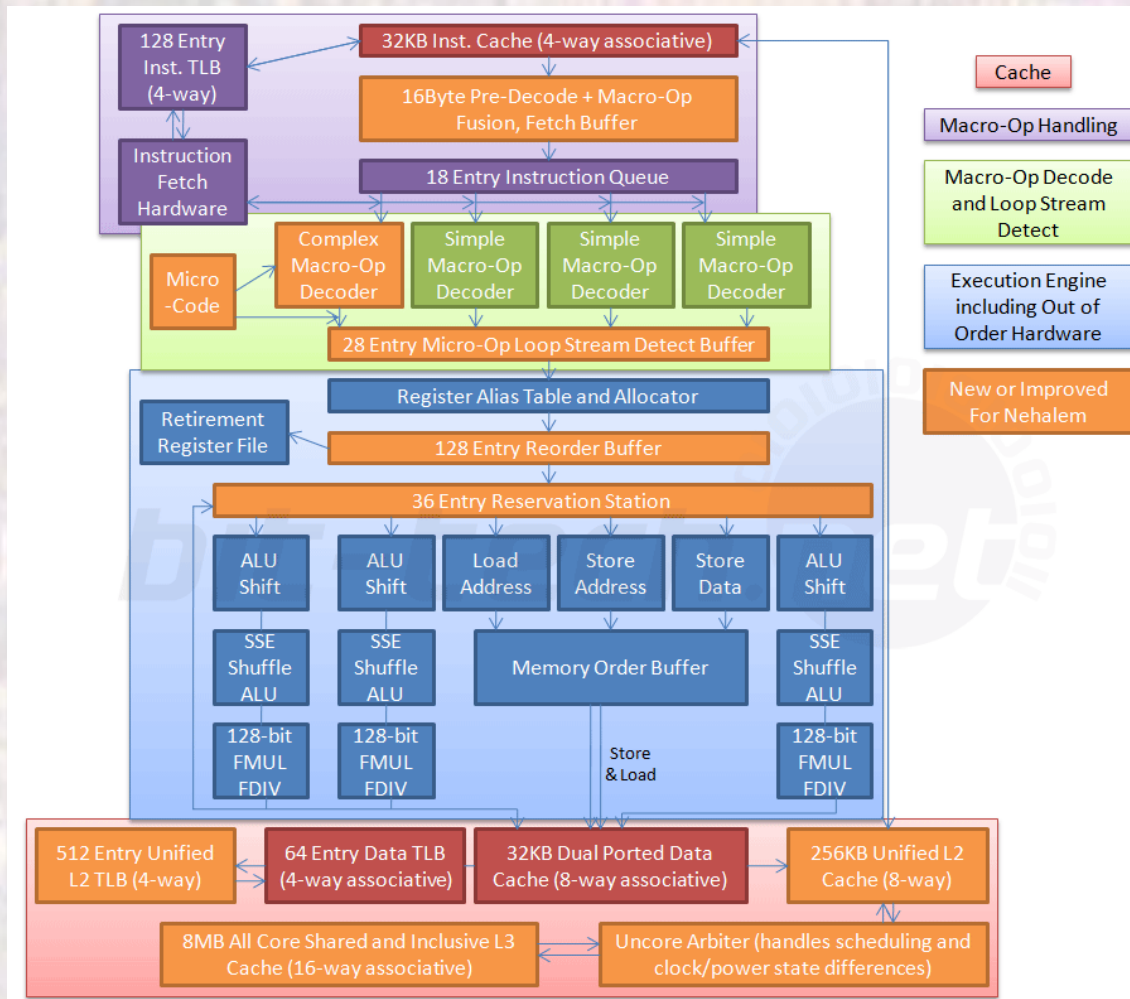
Pipelining

- Processor Architecture



Pipelining

- Processor Architecture



Pipelining

- Modern Example

