

Goals

- Understand how the 5-stage MIPS pipeline works
 - See examples of how architecture impacts ISA design
 - Understand how the pipeline affects performance
- Understand hazards and how to avoid them
 - Structural hazards
 - Data hazards
 - Control hazards

Processor Design in Two Acts

Act I: A single-cycle CPU

Foreshadowing

- Act I: A Single-cycle Processor
 - Simplest design – Not how many real machines work (maybe some deeply embedded processors)
 - Figure out the basic parts; what it takes to execute instructions
- Act II: A Pipelined Processor
 - This is how many real machines work
 - Exploit parallelism by executing multiple instructions at once.

Basic Steps for Execution

- Fetch an instruction from the instruction store
- Decode it
 - What does this instruction do?
- Gather inputs
 - From the register file
 - From memory
- Perform the operation
- Write back the outputs
 - To register file or memory
- Determine the next instruction to execute

The Processor Design Algorithm

- Once you have an ISA...
- Design/Draw the datapath
 - Identify and instantiate the hardware for your architectural state
 - Foreach instruction
 - Simulate the instruction
 - Add and connect the datapath elements it requires
 - Is it workable? If not, fix it.
- Design the control
 - Foreach instruction
 - Simulate the instruction
 - What control lines do you need?
 - How will you compute their value?
 - Modify control accordingly
 - Is it workable? If not, fix it.
- You've already done much of this in 141L.

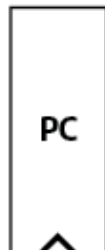
- Arithmetic; R-Type
 - $Inst = Mem[PC]$
 - $REG[rd] = REG[rs] \text{ op } REG[rt]$
 - $PC = PC + 4$

bits	31:26	25:21	20:16	15:11	10:6	5:0
name	op	rs	rt	rd	shamt	funct
# bits	6	5	5	5	5	6

- Arithmetic; R-Type

- $\text{Inst} = \text{Mem}[\text{PC}]$
- $\text{REG}[\text{rd}] = \text{REG}[\text{rs}] \text{ op } \text{REG}[\text{rt}]$
- $\text{PC} = \text{PC} + 4$

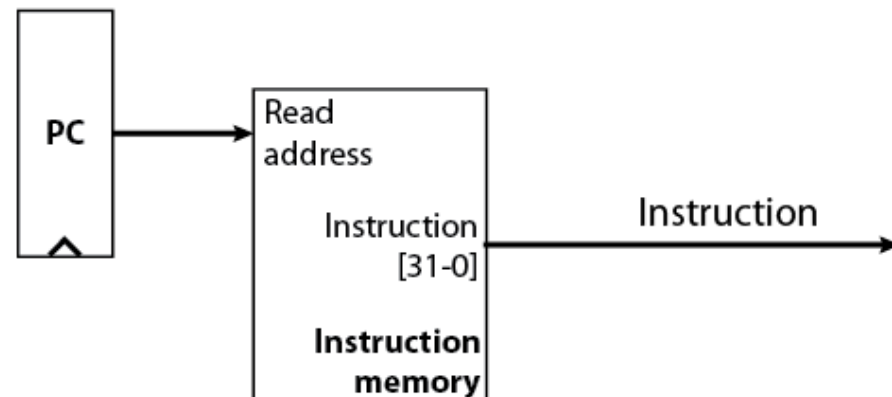
bits	31:26	25:21	20:16	15:11	10:6	5:0
name	op	rs	rt	rd	shamt	funct
# bits	6	5	5	5	5	6



- Arithmetic; R-Type

- $\text{Inst} = \text{Mem}[\text{PC}]$
- $\text{REG}[\text{rd}] = \text{REG}[\text{rs}] \text{ op } \text{REG}[\text{rt}]$
- $\text{PC} = \text{PC} + 4$

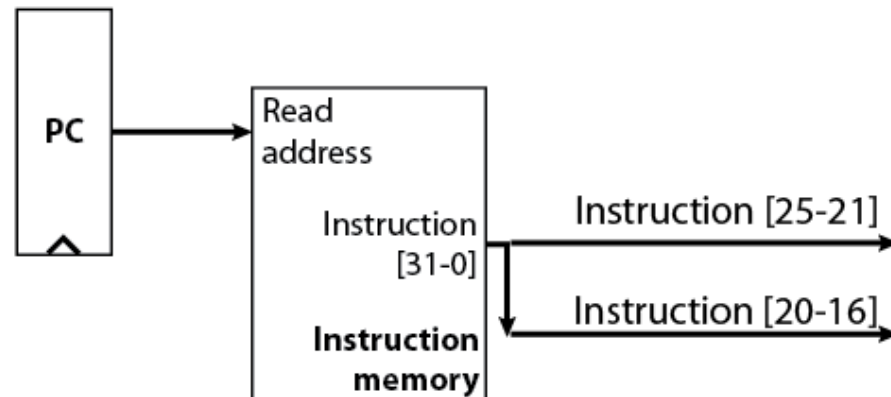
bits	31:26	25:21	20:16	15:11	10:6	5:0
name	op	rs	rt	rd	shamt	funct
# bits	6	5	5	5	5	6



- Arithmetic; R-Type

- $Inst = Mem[PC]$
- $REG[rd] = REG[rs] \text{ op } REG[rt]$
- $PC = PC + 4$

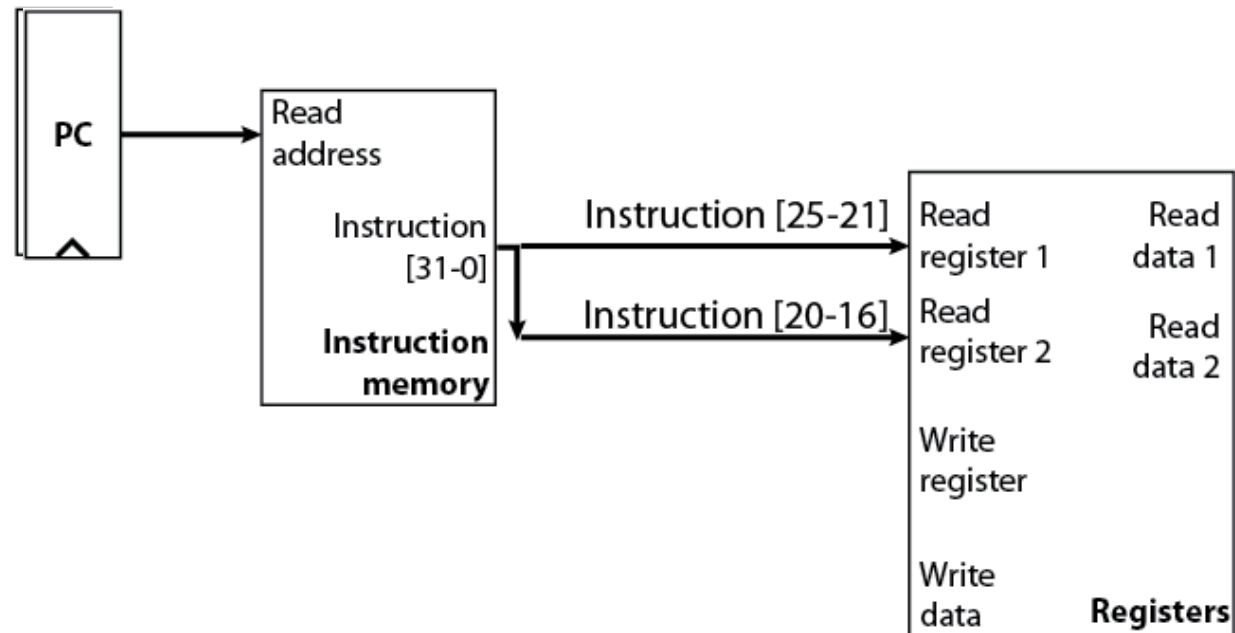
bits	31:26	25:21	20:16	15:11	10:6	5:0
name	op	rs	rt	rd	shamt	funct
# bits	6	5	5	5	5	6



- Arithmetic; R-Type

- $Inst = Mem[PC]$
- $REG[rd] = REG[rs] \text{ op } REG[rt]$
- $PC = PC + 4$

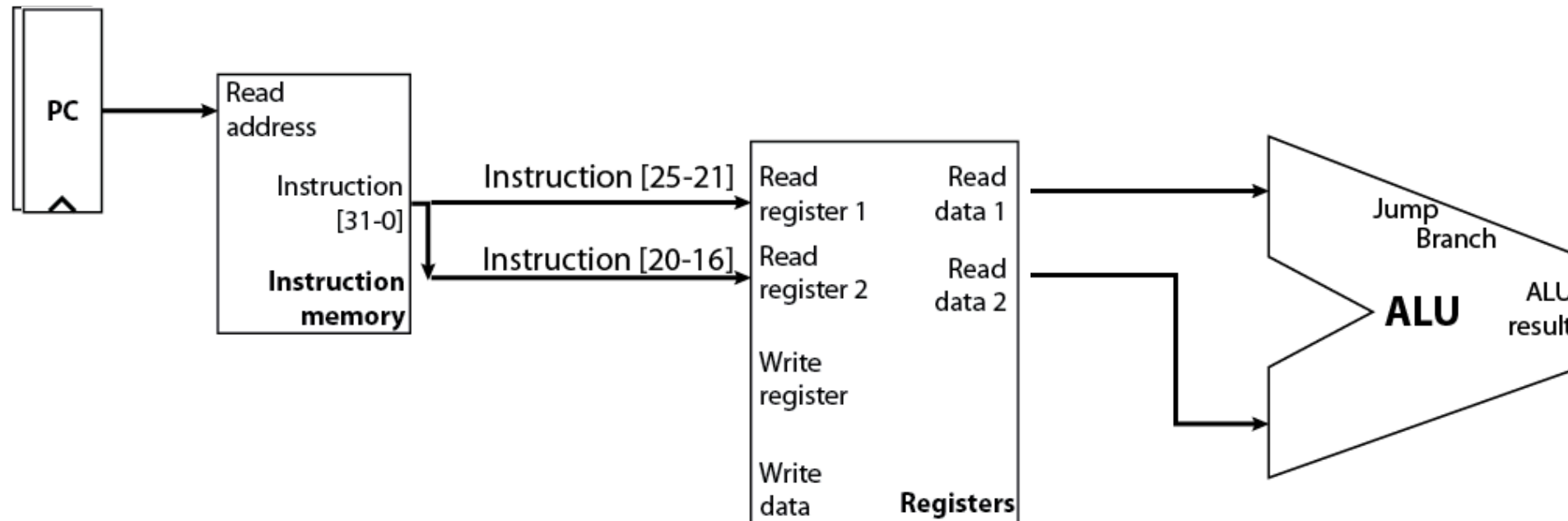
bits	31:26	25:21	20:16	15:11	10:6	5:0
name	op	rs	rt	rd	shamt	funct
# bits	6	5	5	5	5	6



- Arithmetic; R-Type

- $Inst = Mem[PC]$
- $REG[rd] = REG[rs] \text{ op } REG[rt]$
- $PC = PC + 4$

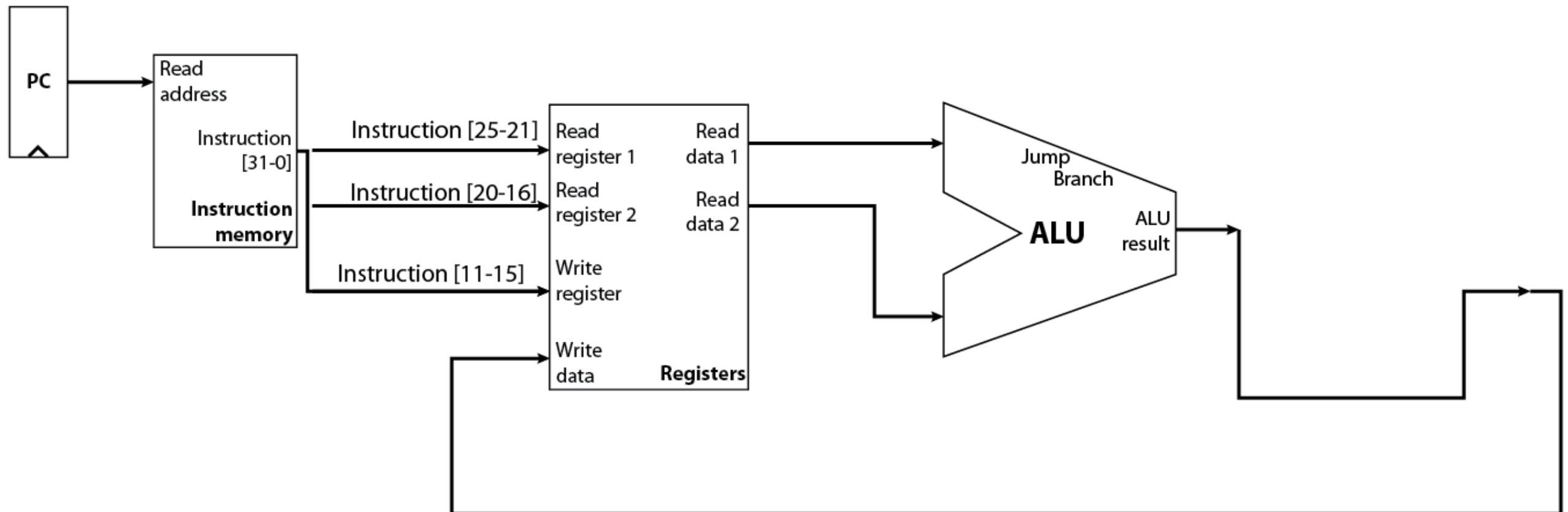
bits	31:26	25:21	20:16	15:11	10:6	5:0
name	op	rs	rt	rd	shamt	funct
# bits	6	5	5	5	5	6



- Arithmetic; R-Type

- $Inst = Mem[PC]$
- $REG[rd] = REG[rs] \text{ op } REG[rt]$
- $PC = PC + 4$

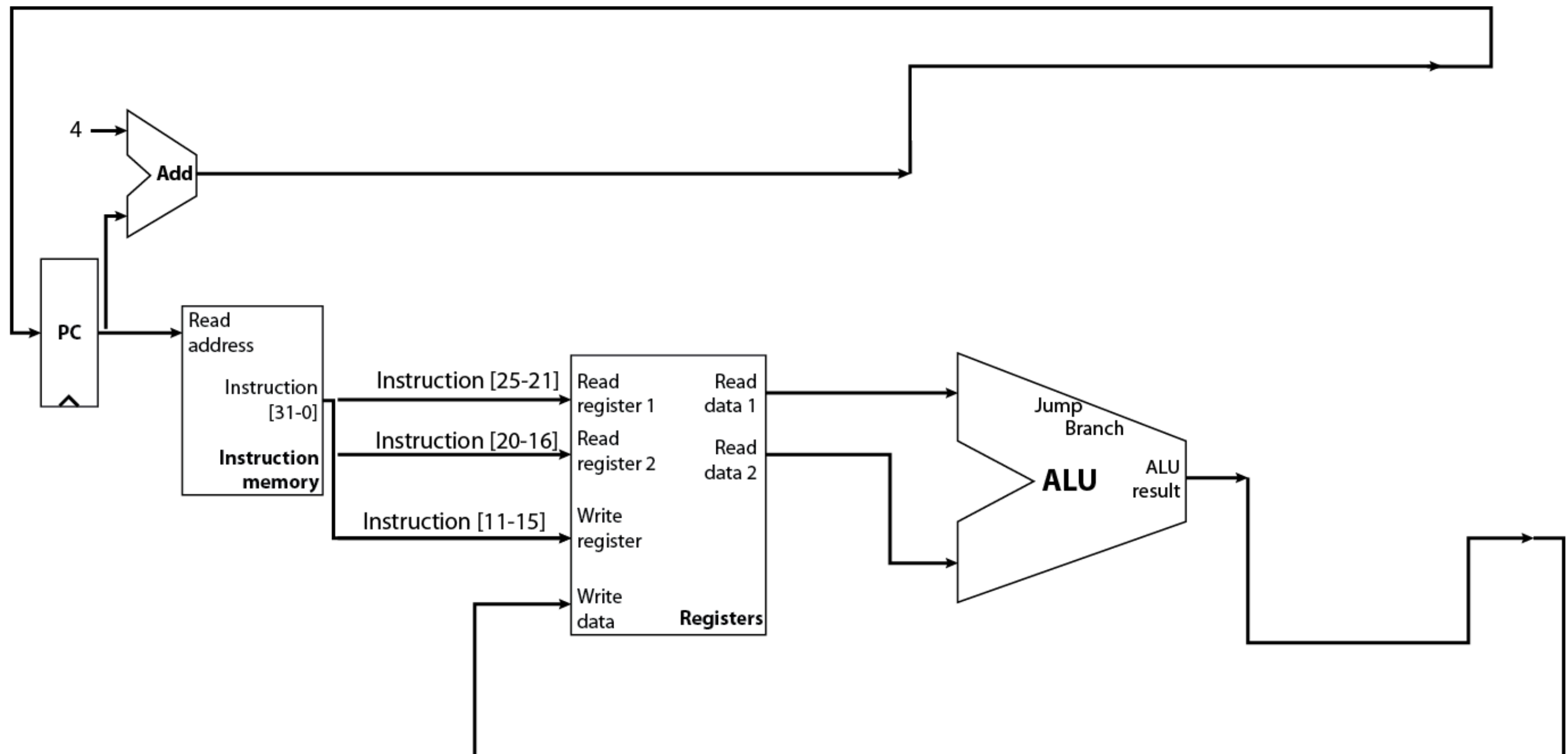
bits	31:26	25:21	20:16	15:11	10:6	5:0
name	op	rs	rt	rd	shamt	funct
# bits	6	5	5	5	5	6



- Arithmetic; R-Type

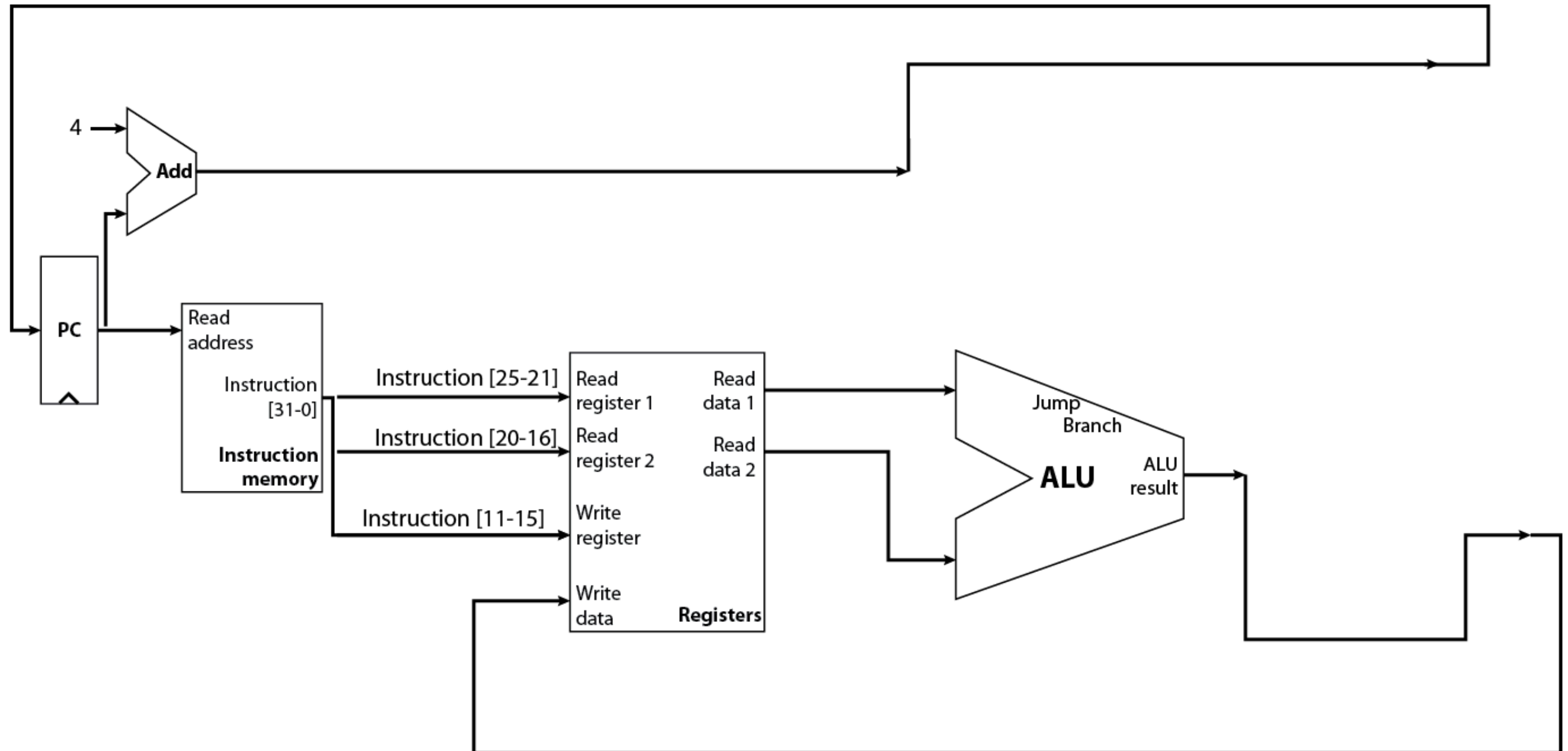
- $Inst = Mem[PC]$
- $REG[rd] = REG[rs] \text{ op } REG[rt]$
- $PC = PC + 4$

bits	31:26	25:21	20:16	15:11	10:6	5:0
name	op	rs	rt	rd	shamt	funct
# bits	6	5	5	5	5	6



- ADDI; I-Type
 - $PC = PC + 4$
 - $REG[rd] = REG[rs] \text{ op } \text{SignExtImm}$

bits	31:26	25:21	20:16	15:0
name	op	rs	rt	imm
# bits	6	5	5	16

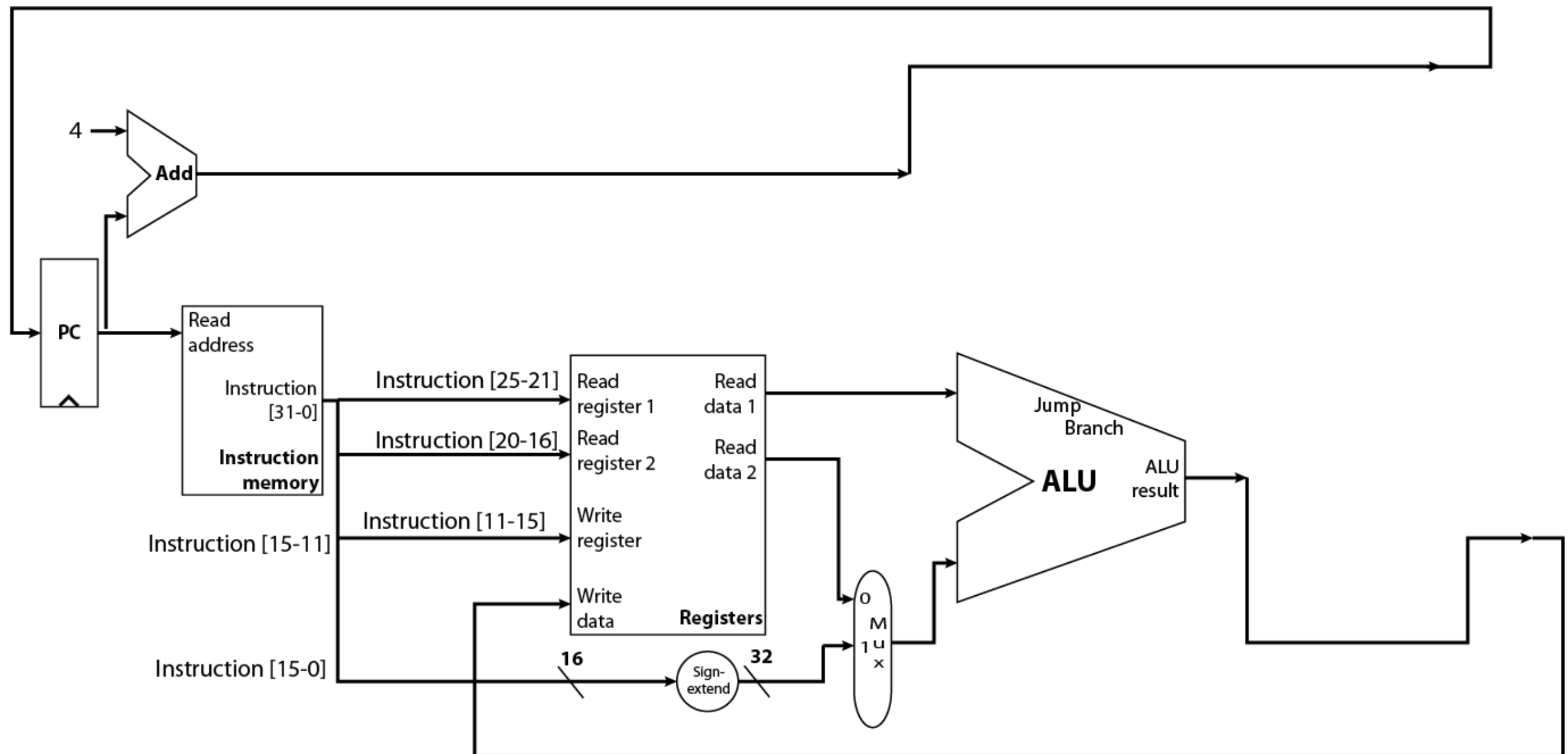


- ADDI; I-Type

- $PC = PC + 4$

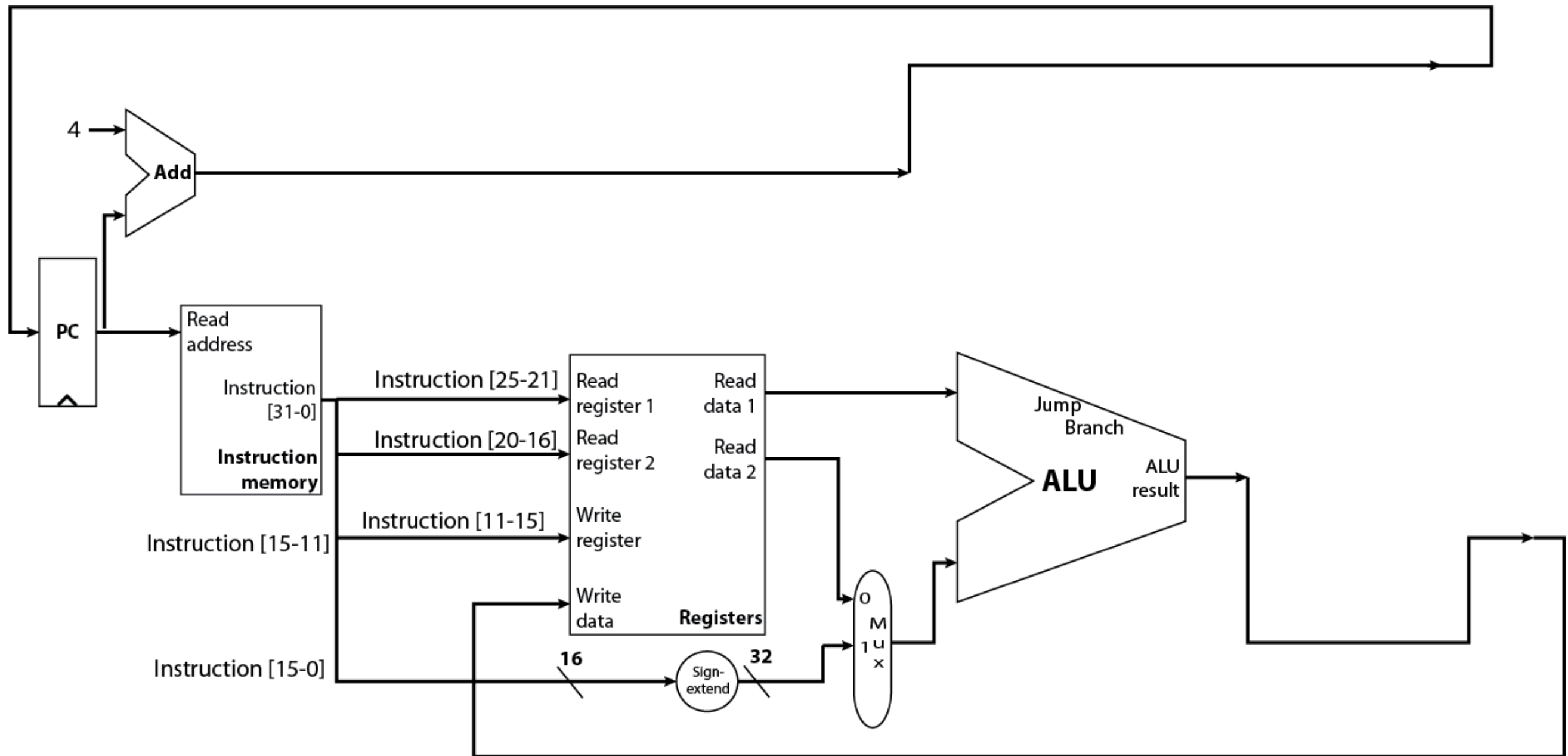
- $REG[rd] = REG[rs] \text{ op } \text{SignExtImm}$

bits	31:26	25:21	20:16	15:0
name	op	rs	rt	imm
# bits	6	5	5	16



- Load Word
 - $PC = PC + 4$
 - $REG[rt] = MEM[signextendImm + REG[rs]]$

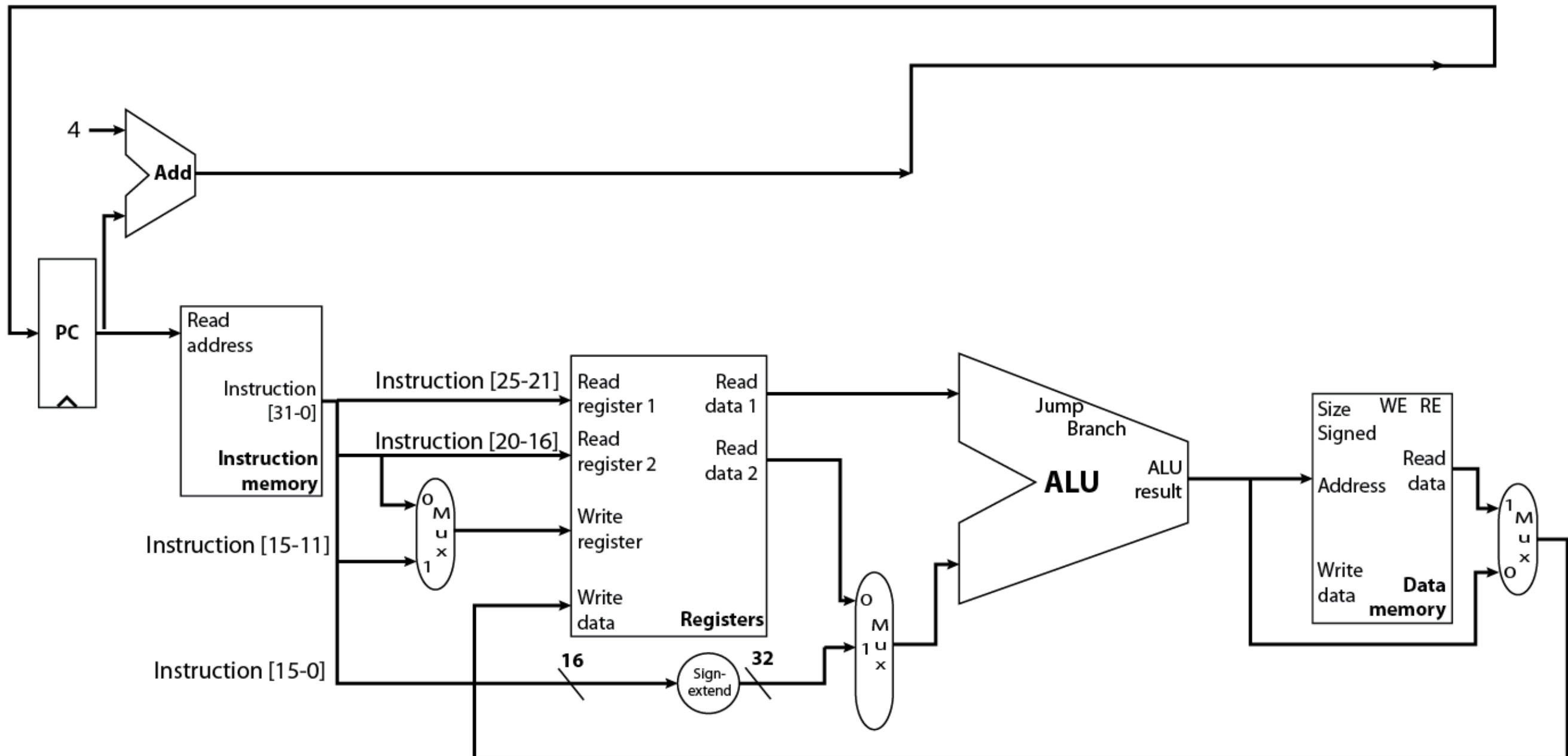
bits	31:26	25:21	20:16	15:0
name	op	rs	rt	immediate
# bits	6	5	5	16



• Load Word

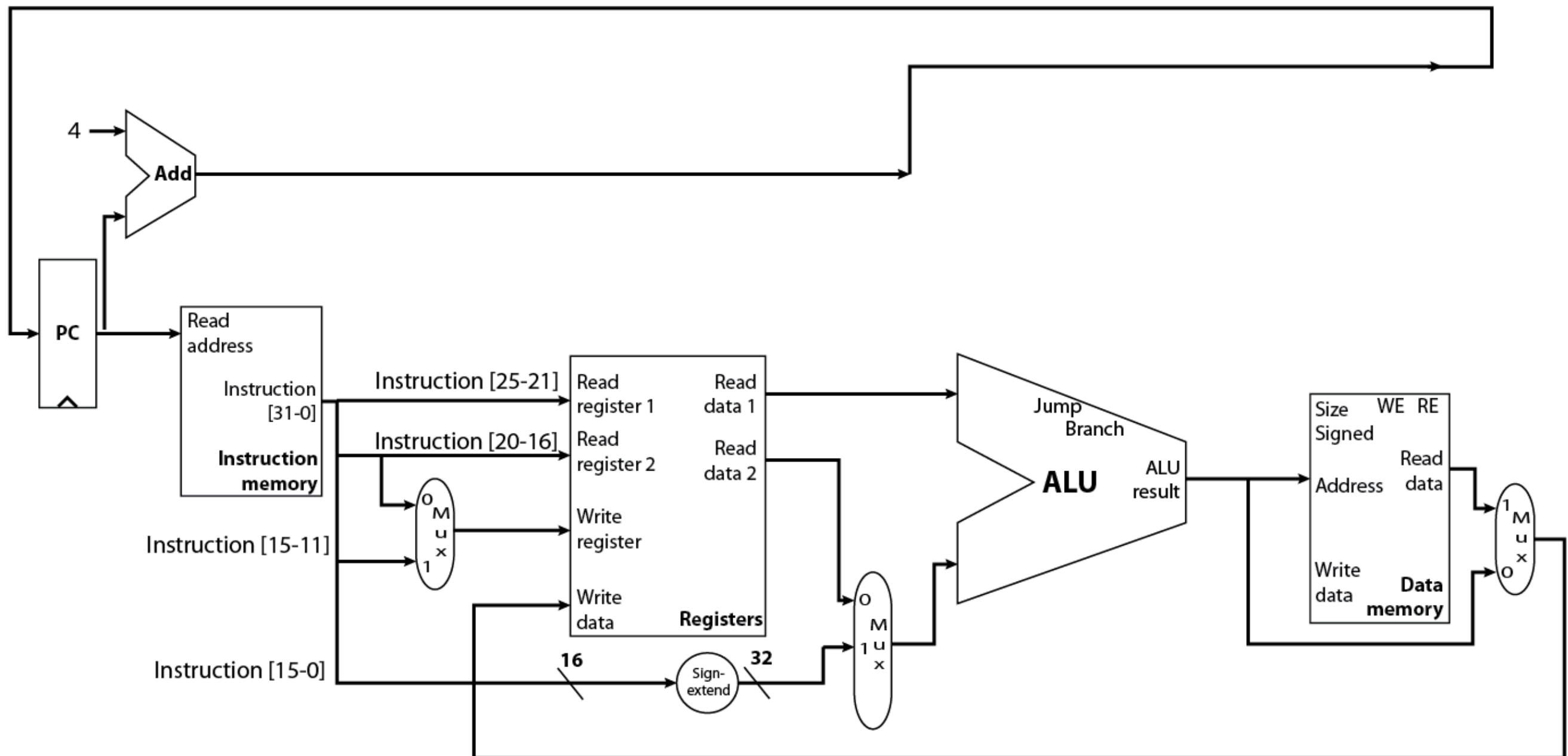
- $PC = PC + 4$
- $REG[rt] = MEM_{signextendImm} + REG[rs]$

bits	31:26	25:21	20:16	15:0
name	op	rs	rt	immediate
# bits	6	5	5	16



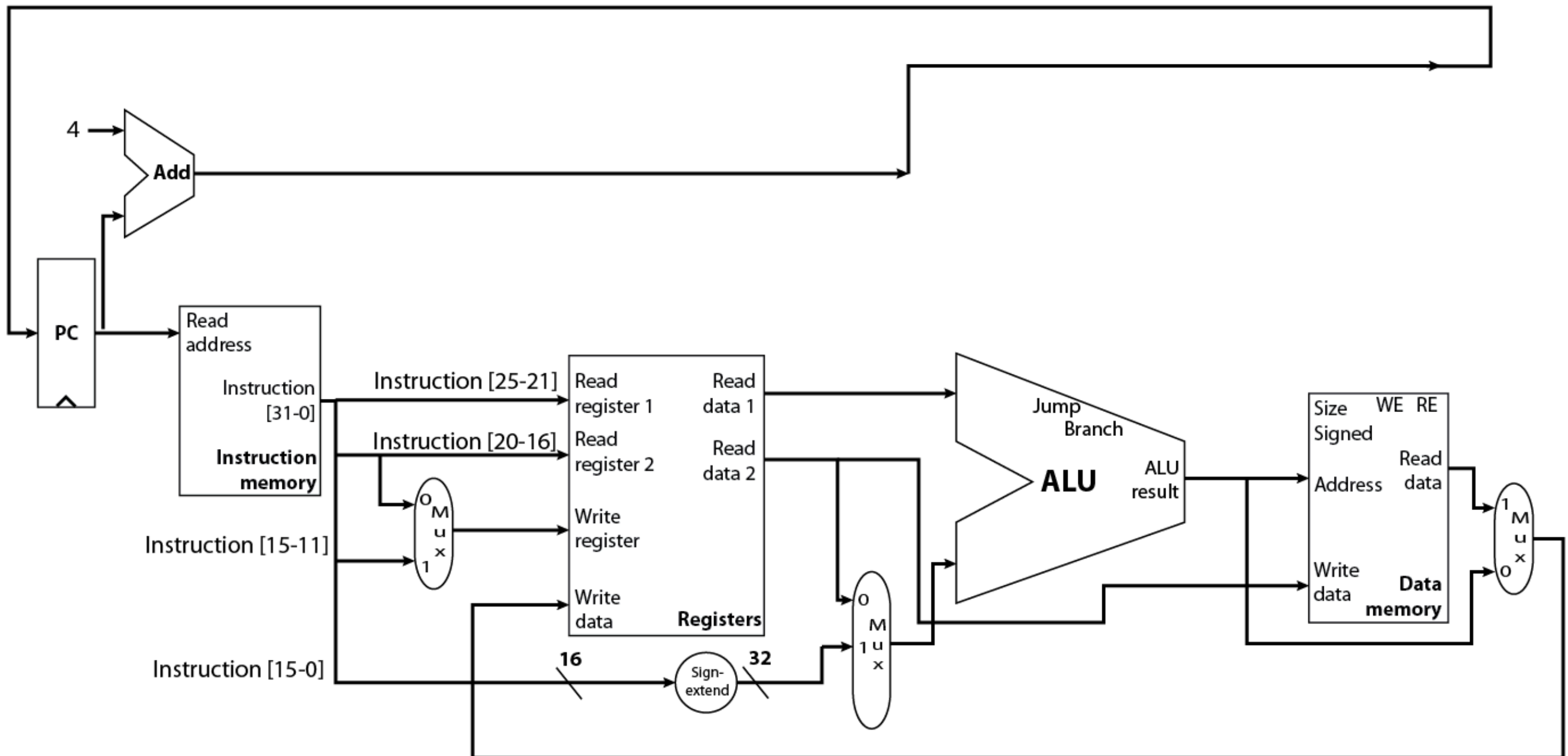
- Store Word
 - $PC = PC + 4$
 - $MEM[\text{signextendImm} + REG[rs]] = REG[rt]$

bits	31:26	25:21	20:16	15:0
name	op	rs	rt	immediate
# bits	6	5	5	16



- Store Word
 - $PC = PC + 4$
 - $MEM[\text{signextendImm} + REG[rs]] = REG[rt]$

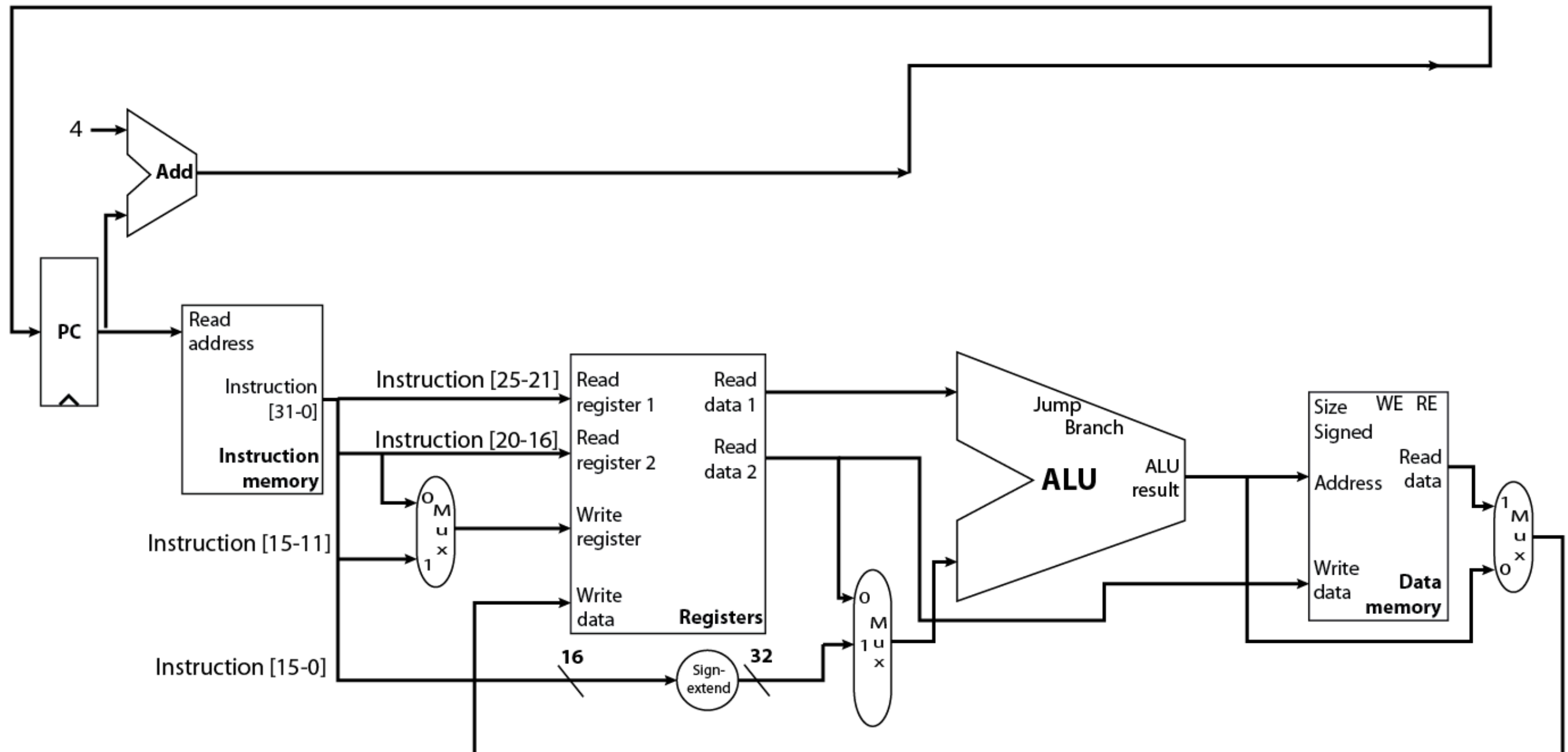
bits	31:26	25:21	20:16	15:0
name	op	rs	rt	immediate
# bits	6	5	5	16



• Branch-equal; I-Type

- $PC = (REG[rs] == REG[rt]) ? PC + 4 + SignExtImmediate * 4 : PC + 4;$

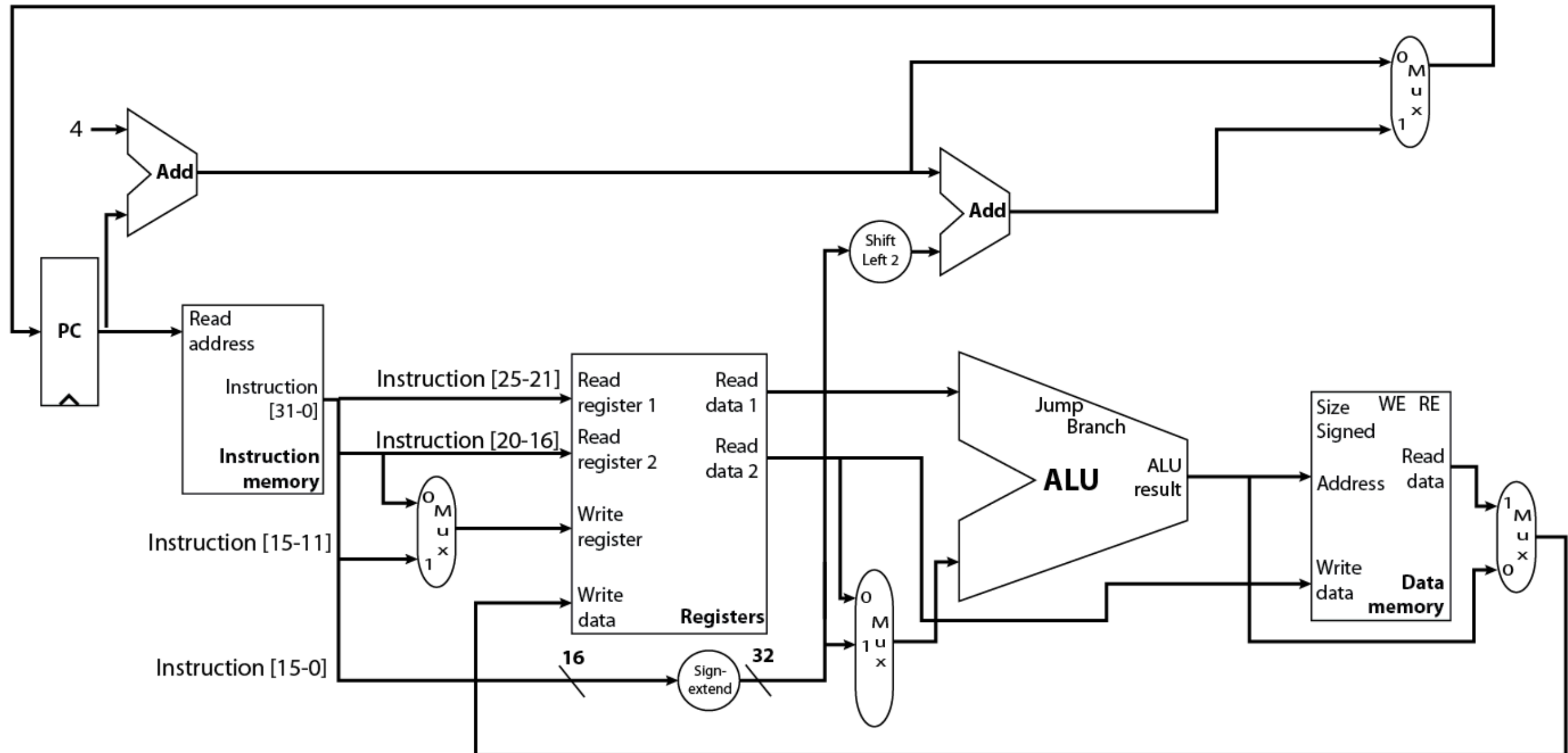
bits	31:26	25:21	20:16	15:0
name	op	rs	rt	displacement
# bits	6	5	5	16



• Branch-equal; I-Type

- $PC = (REG[rs] == REG[rt]) ? PC + 4 + \text{SignExtImmediate} * 4 : PC + 4;$

bits	31:26	25:21	20:16	15:0
name	op	rs	rt	displacement
# bits	6	5	5	16



A Single-cycle Processor

- Performance refresher
- $ET = IC * CPI * CT$
- Single cycle $\Rightarrow CPI == 1$; That sounds great
- Unfortunately, Single cycle $\Rightarrow CT$ is large
 - Even RISC instructions take quite a bite of effort to execute
 - This is a lot to do in one cycle

Our Hardware is Mostly Idle

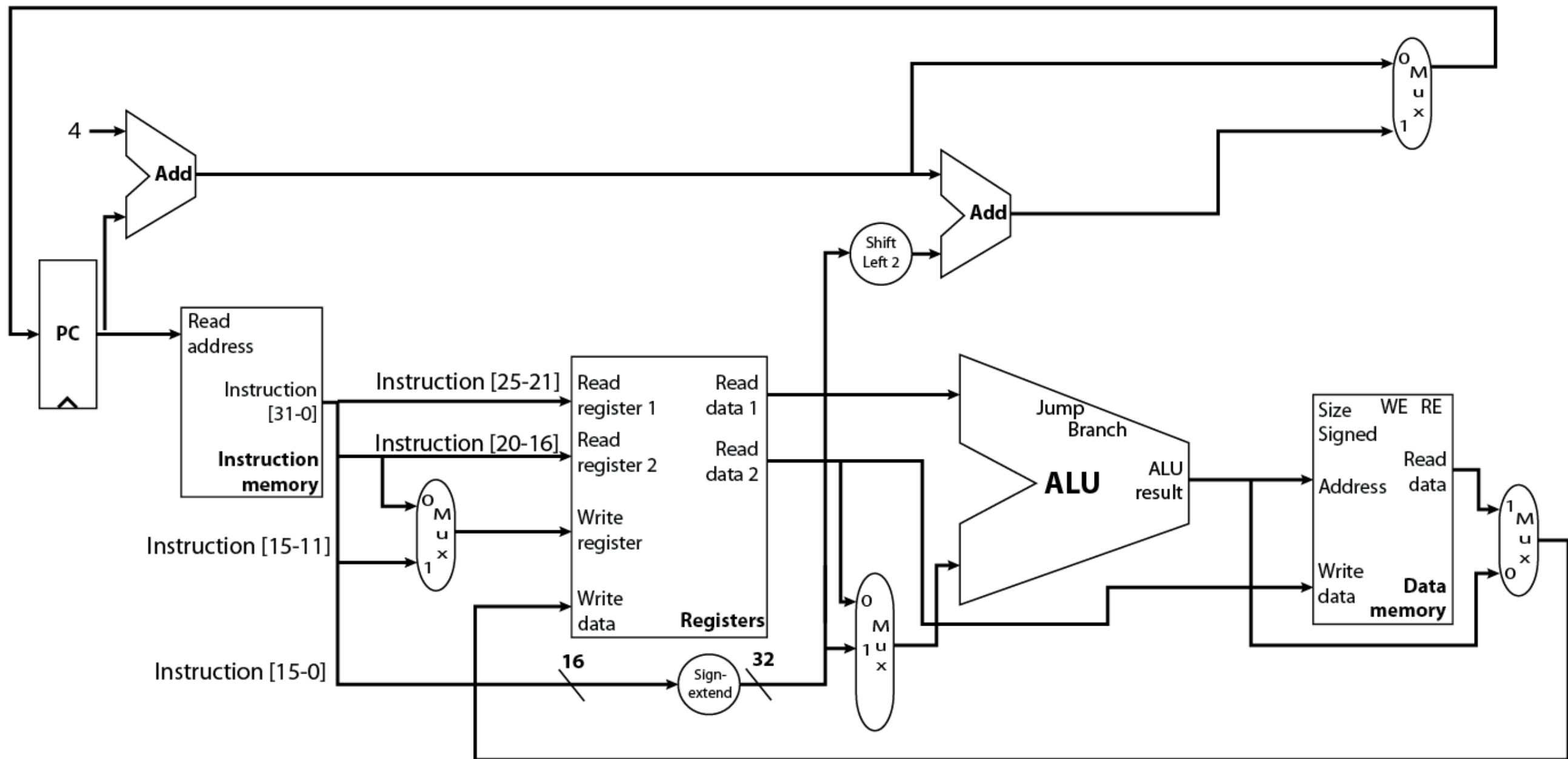
Cycle time = 15 ns

Slowest module (alu) is ~6ns

Our Hardware is Mostly Idle

Cycle time = 15 ns

Slowest module (alu) is ~6ns

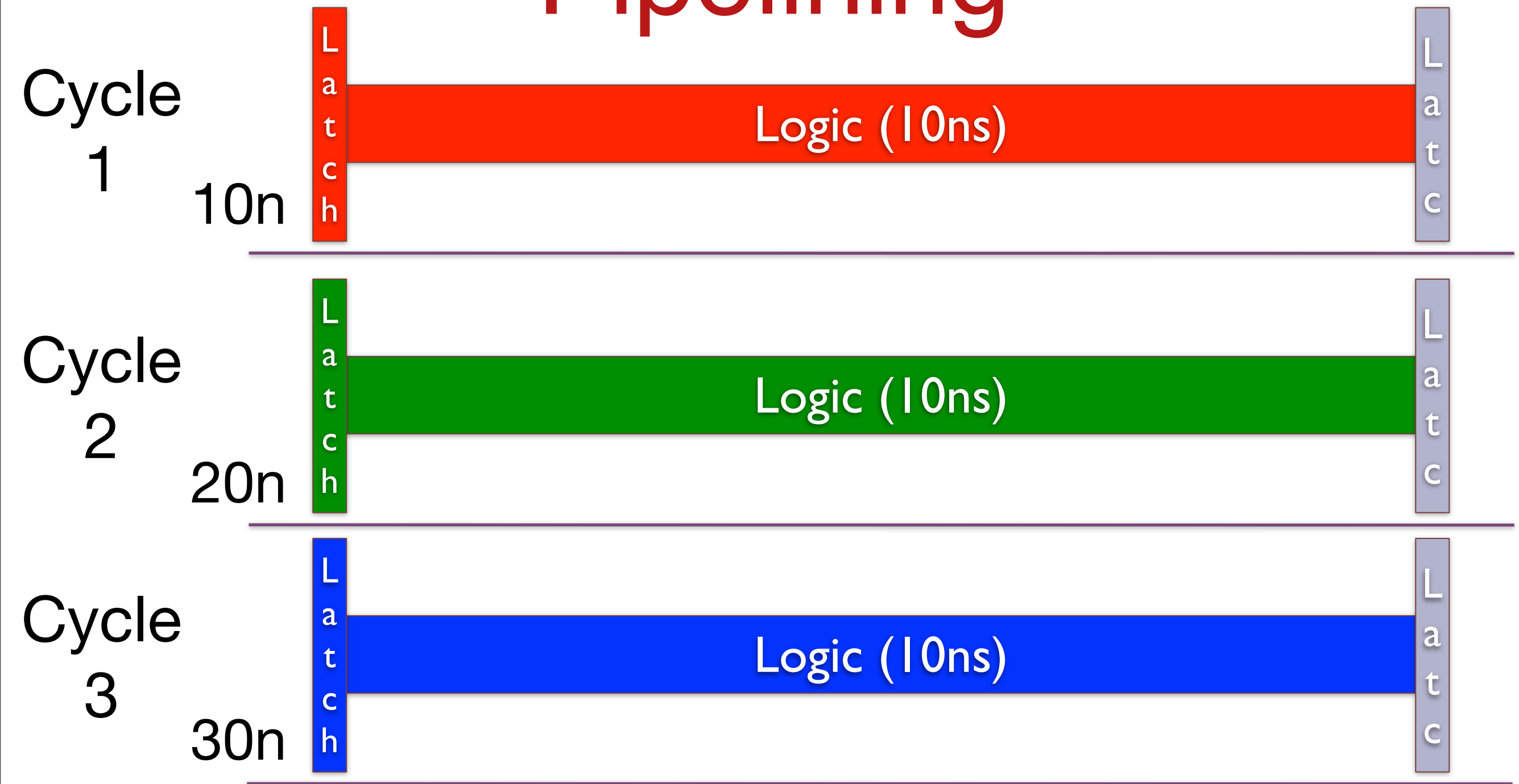


Processor Design in Two Acts

Act II: A pipelined CPU

Pipelining Review

Pipelining

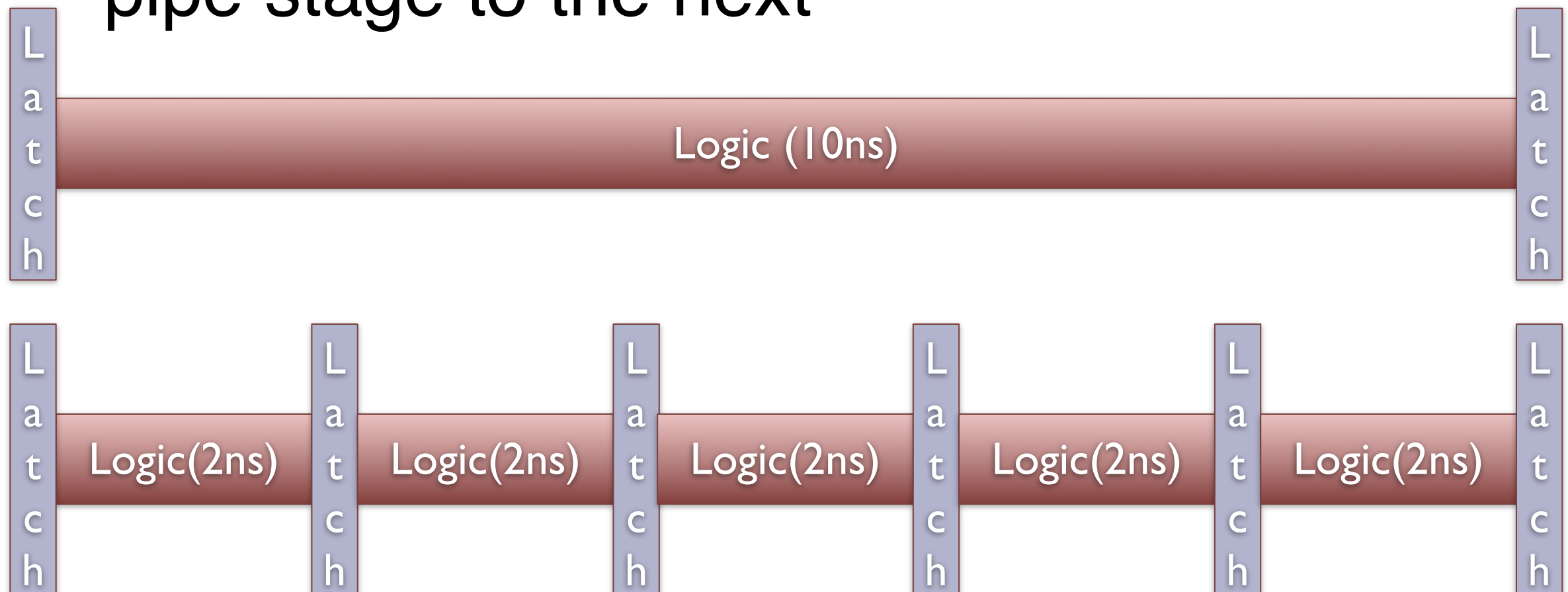


What's the throughput?

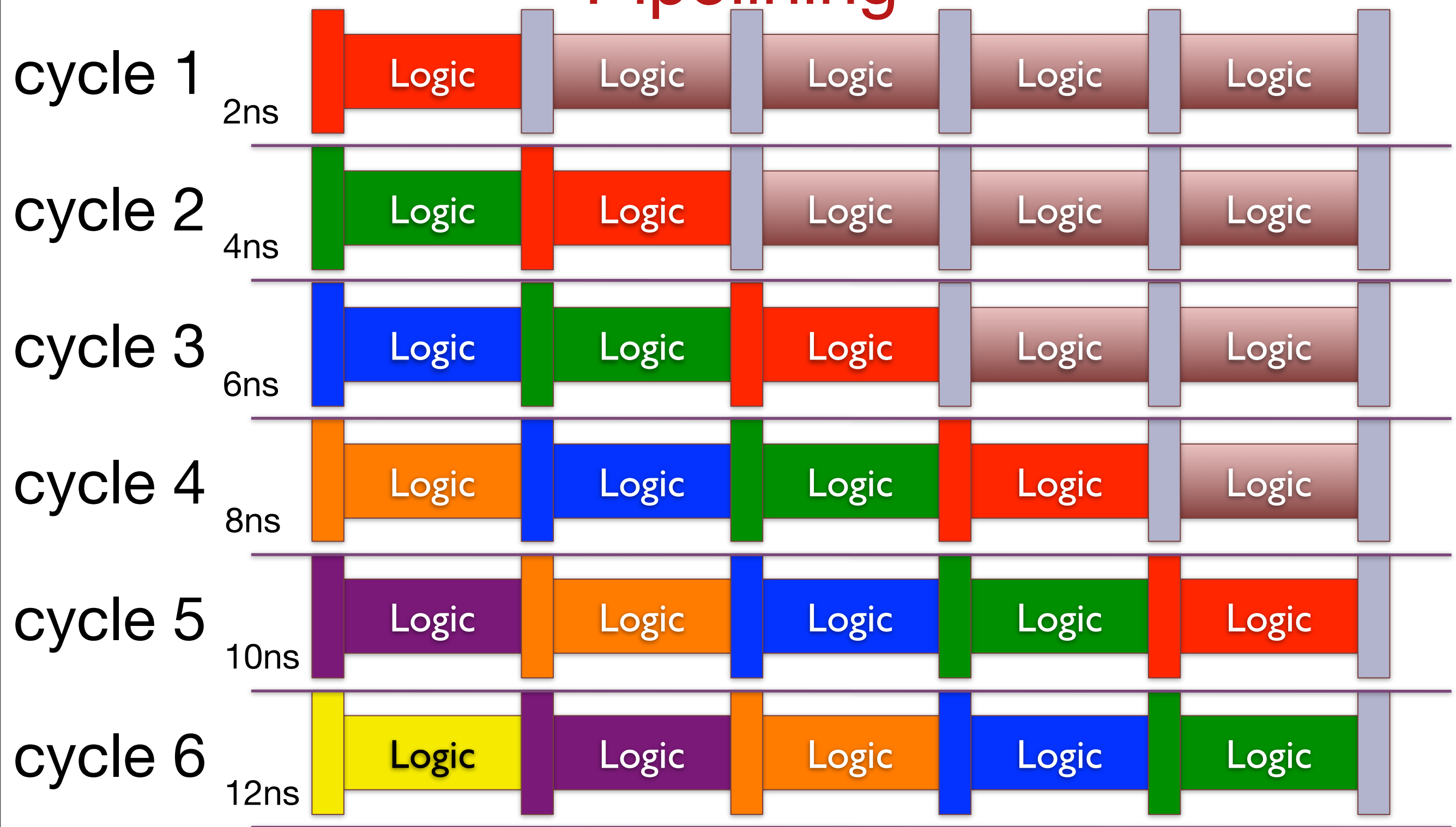
What's the latency for one unit of work?

Pipelining

- Break up the logic with latches into “pipeline stages”
- Each stage can act on different data
- Latches hold the inputs to their stage
- Every clock cycle data transfers from one pipe stage to the next



Pipelining

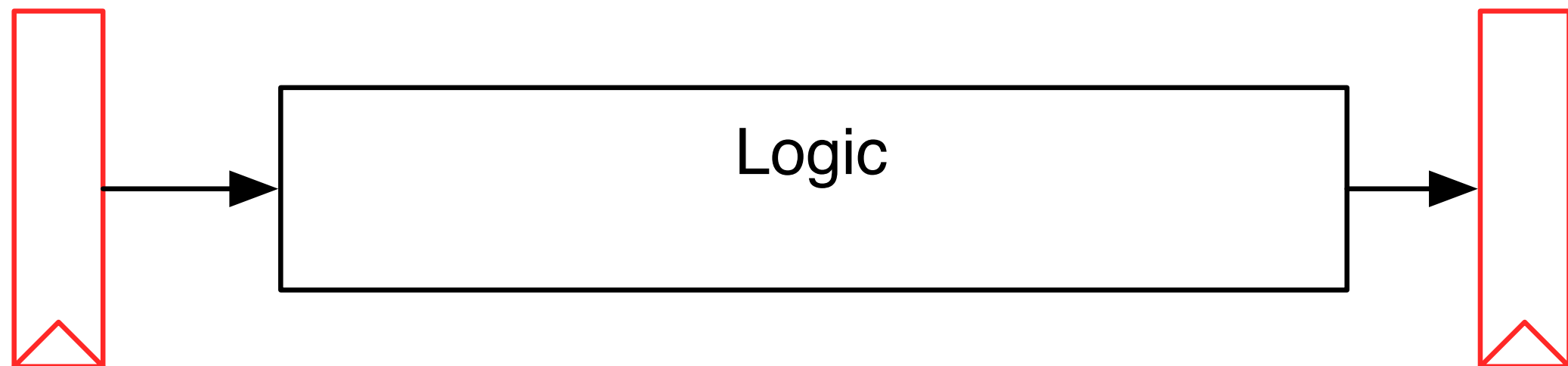


What's the latency for one unit of work?

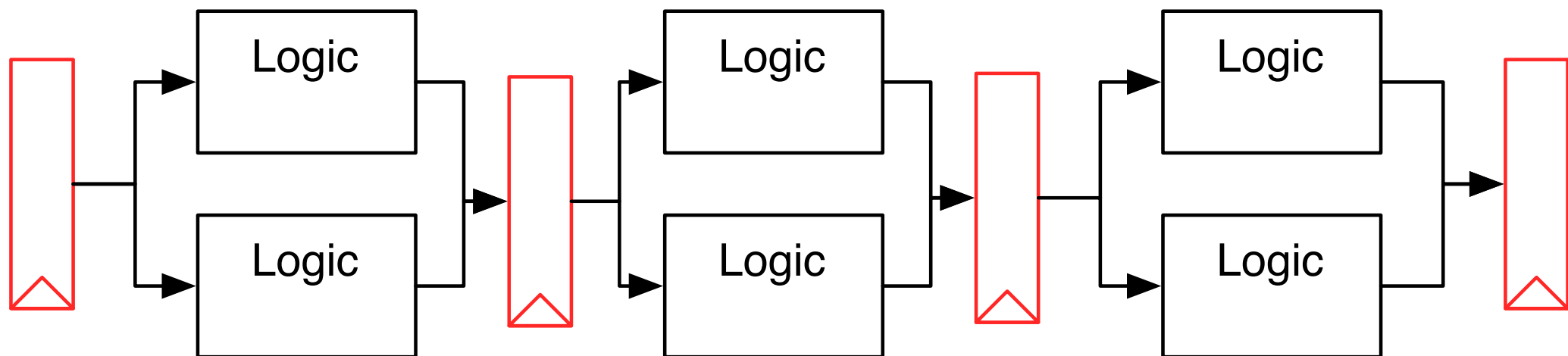
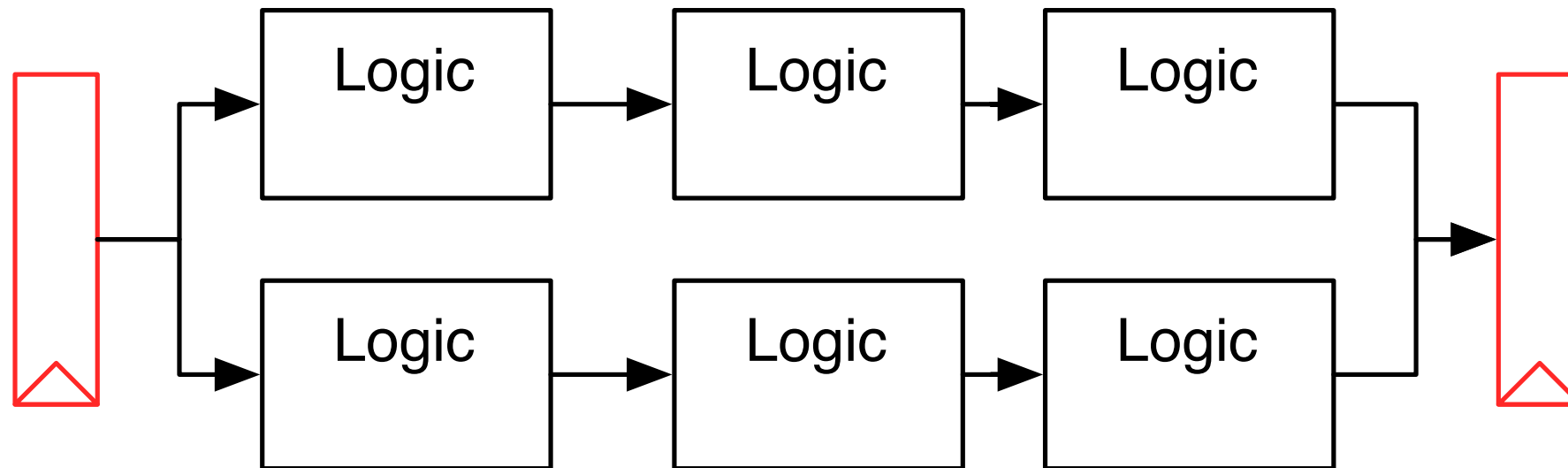
What's the throughput?

Critical path review

- Critical path is the longest possible delay between two registers in a design.
- The critical path sets the cycle time, since the cycle time must be long enough for a signal to traverse the critical path.
- Lengthening or shortening non-critical paths does not change performance
- Ideally, all paths are about the same length

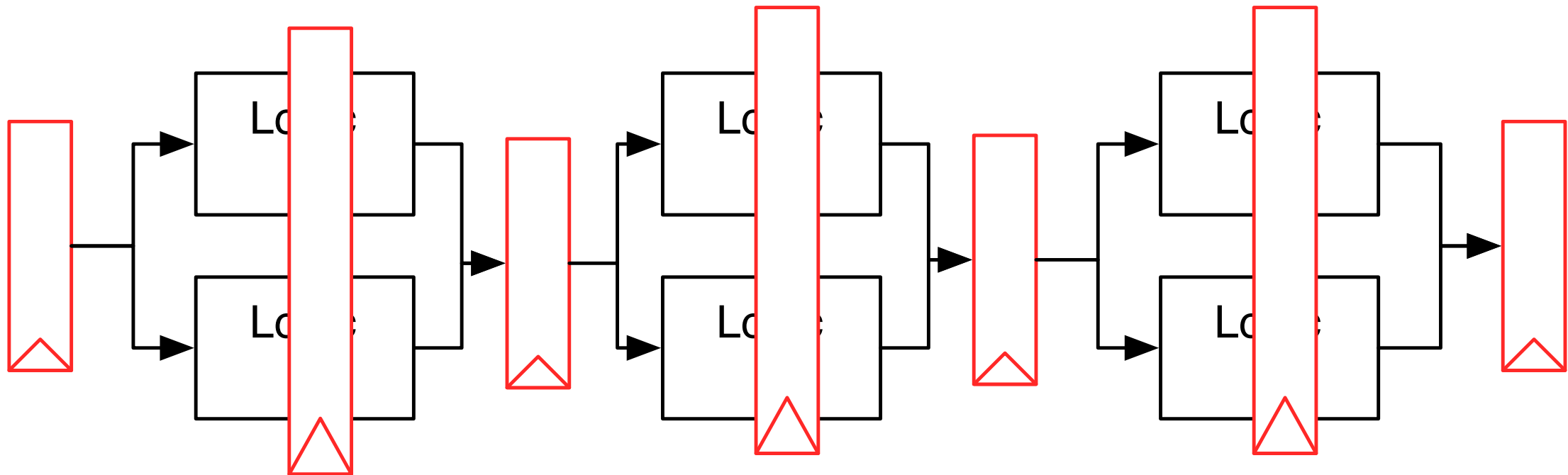


Pipelining and Logic



- Hopefully, critical path reduced by 1/3

Limitations of Pipelining

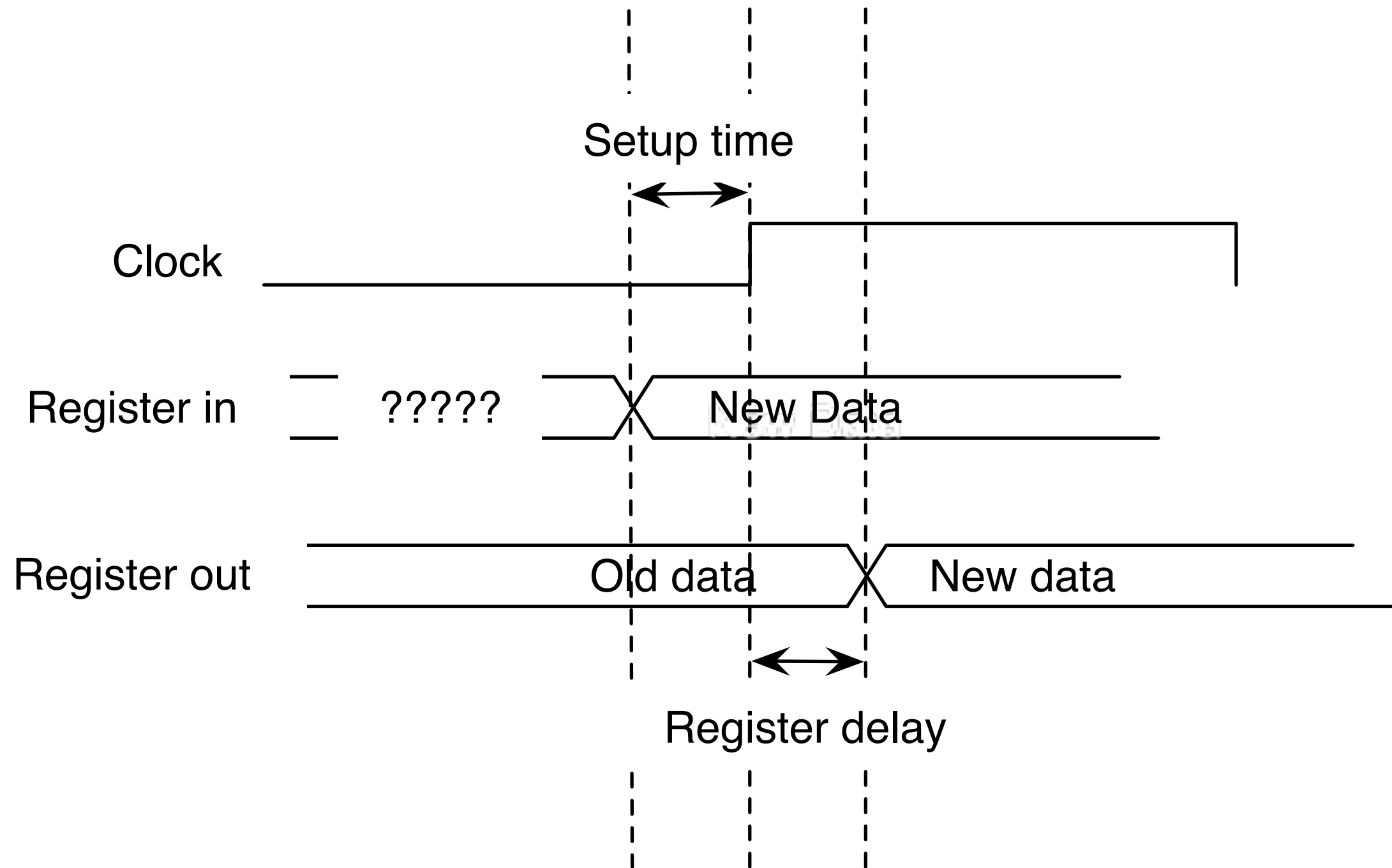


- You cannot pipeline forever
 - Some logic cannot be pipelined arbitrarily -- Memories
 - Some logic is inconvenient to pipeline.
 - How do you insert a register in the middle of an multiplier?
- Registers have a cost
 - They cost area -- choose “narrow points” in the logic
 - They cost energy -- latches don't do any useful work
 - They cost time
 - Extra logic delay
 - Set-up and hold times.
- Pipelining may not affect the critical path as you expect

Pipelining Overhead

- Logic Delay (LD) -- How long does the logic take (i.e., the useful part)
- Set up time (ST) -- How long before the clock edge do the inputs to a register need be ready?
 - Relatively short -- 0.036 ns on our FPGAs
- Register delay (RD) -- Delay through the internals of the register.
 - Longer -- 1.5 ns for our FPGAs.
 - Much, much shorter for RAW CMOS.

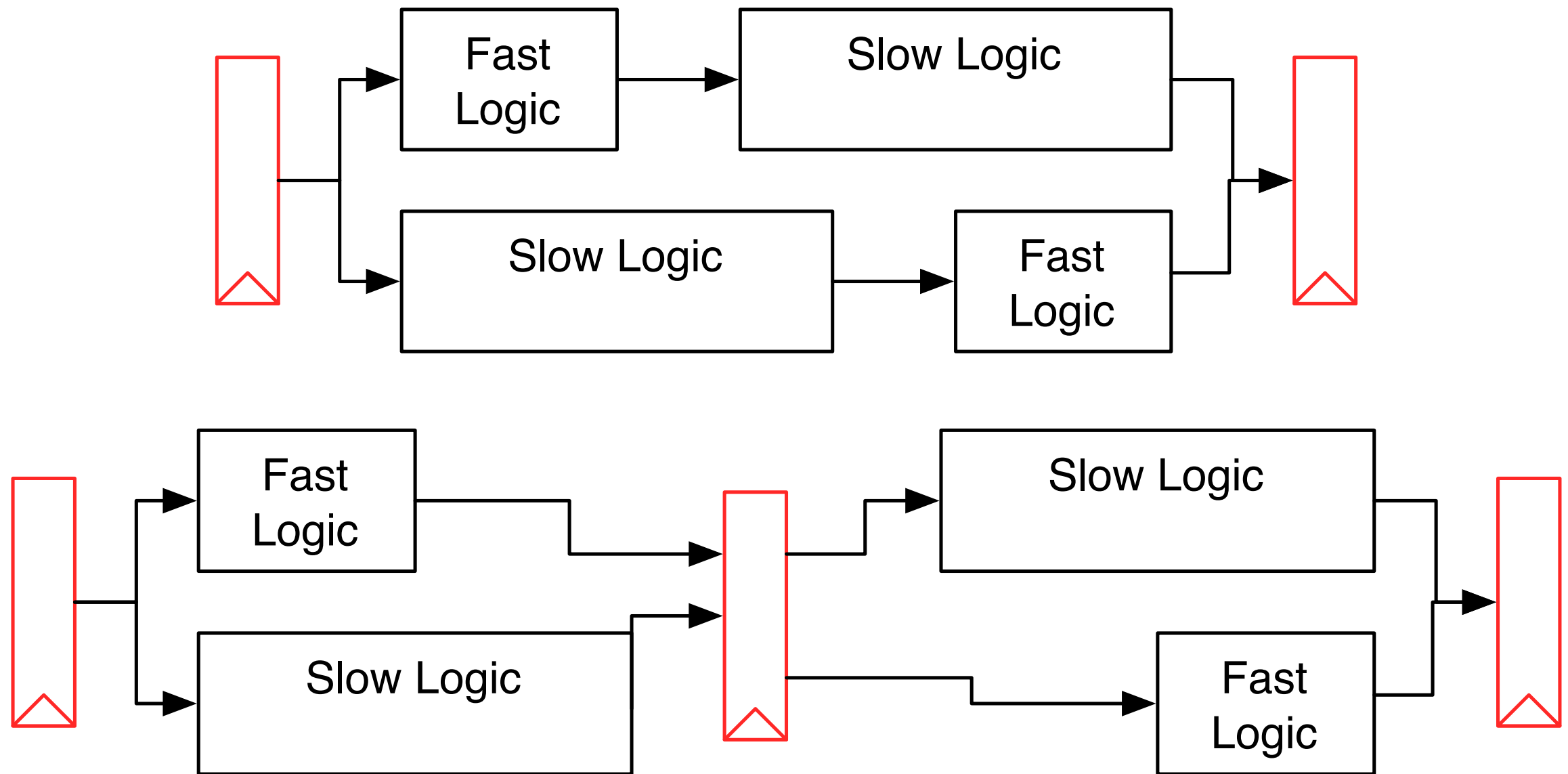
Pipelining Overhead



Pipelining Overhead

- Logic Delay (LD) -- How long does the logic take (i.e., the useful part)
- Set up time (ST) -- How long before the clock edge do the inputs to a register need be ready?
- Register delay (RD) -- Delay through the internals of the register.
- CT_{base} -- cycle time before pipelining
 - $CT_{base} = LD + ST + RD.$
- CT_{pipe} -- cycle time after pipelining N times
 - $CT_{pipe} = ST + RD + LD/N$
 - Total time = $N*ST + N*RD + LD$

Pipelining Difficulties



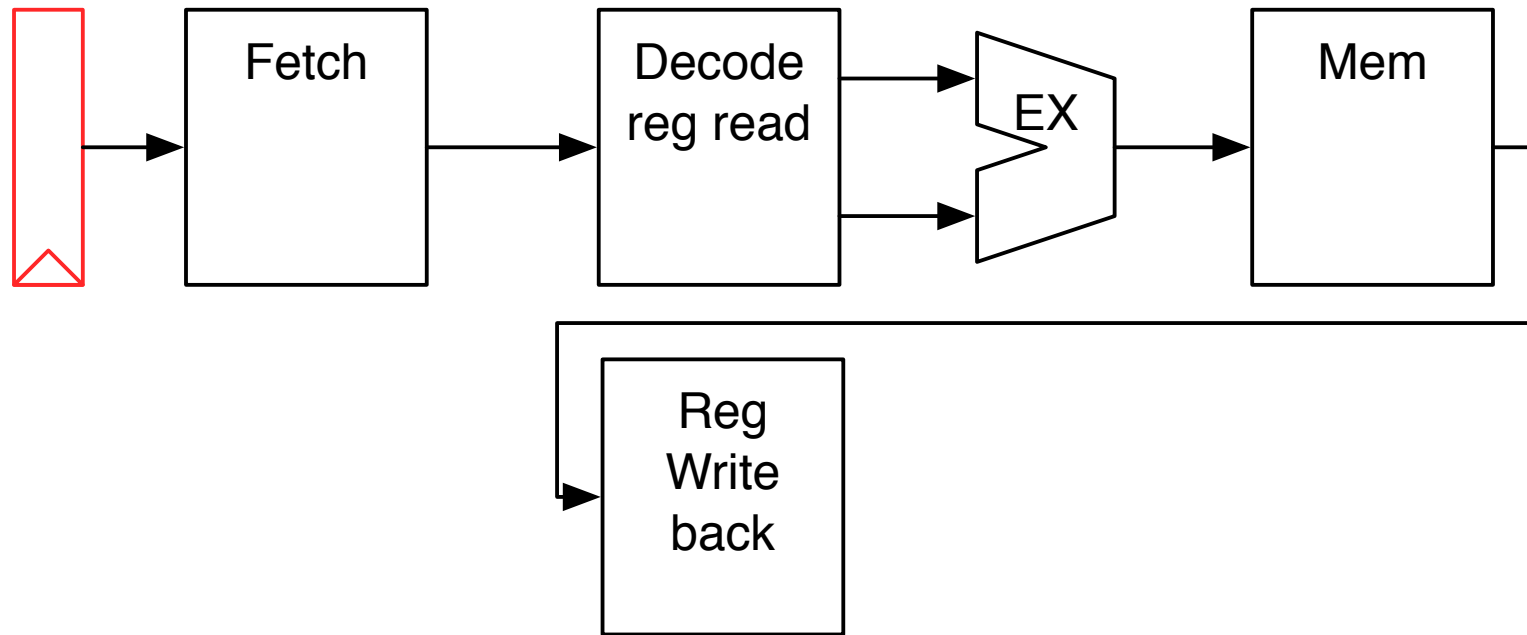
- You can't always pipeline how you would like

How to pipeline a processor

- Break each instruction into pieces -- remember the basic algorithm for execution
 - Fetch
 - Decode
 - Collect arguments
 - Execute
 - Write back results
 - Compute next PC
- The “classic 5-stage MIPS pipeline”
 - Fetch -- read the instruction
 - Decode -- decode and read from the register file
 - Execute -- Perform arithmetic ops and address calculations
 - Memory -- access data memory.
 - Write back-- Store results in the register file.

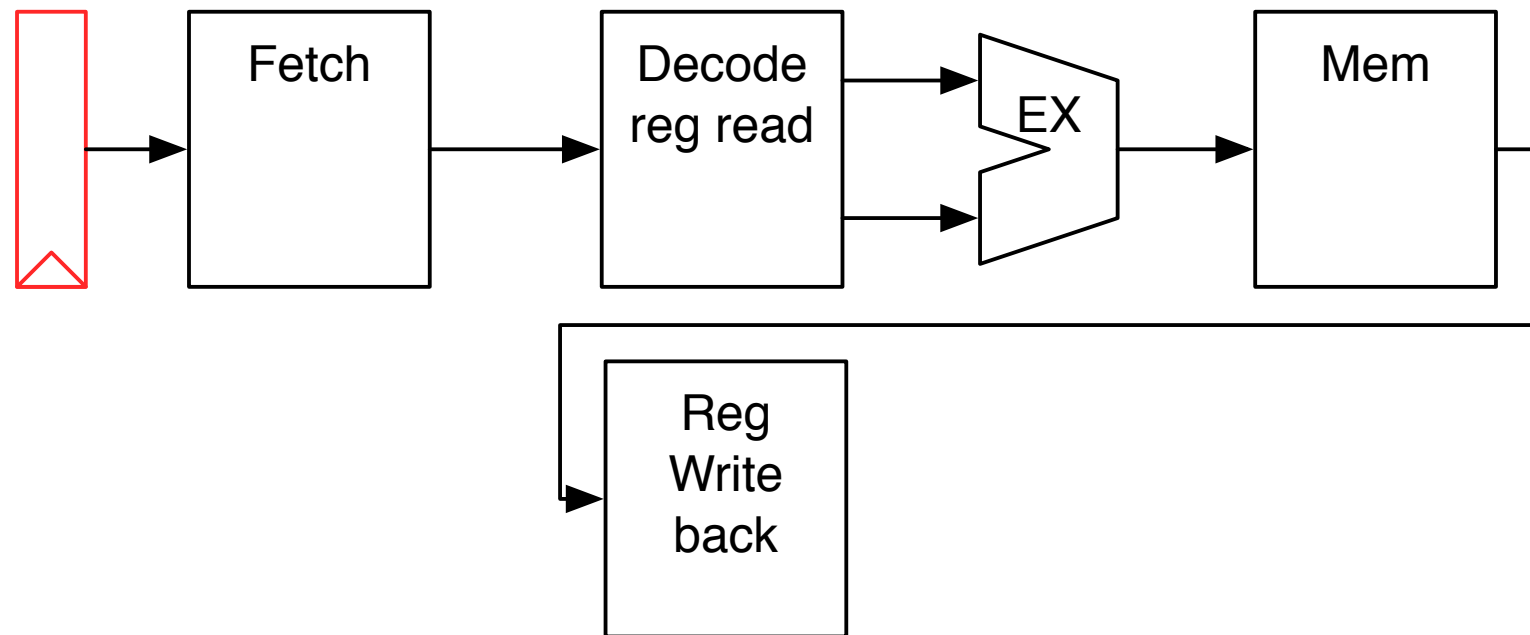
Pipelining a processor

Pipelining a processor

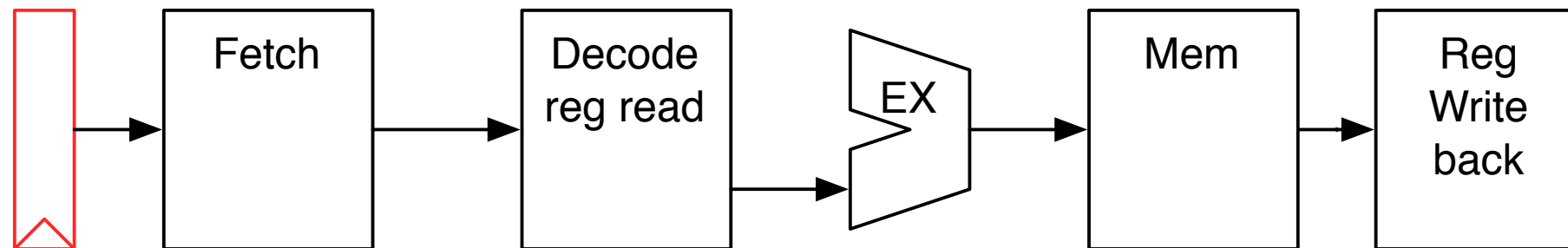


Reality

Pipelining a processor

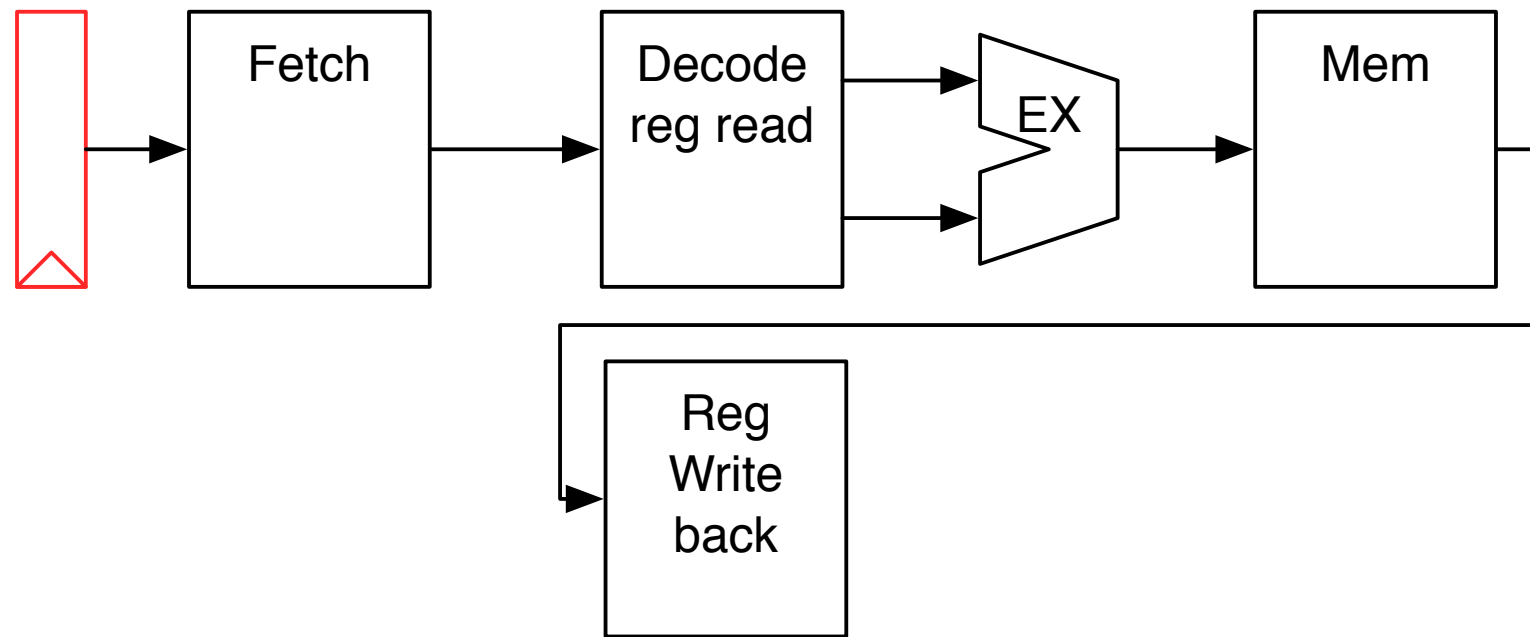


Reality

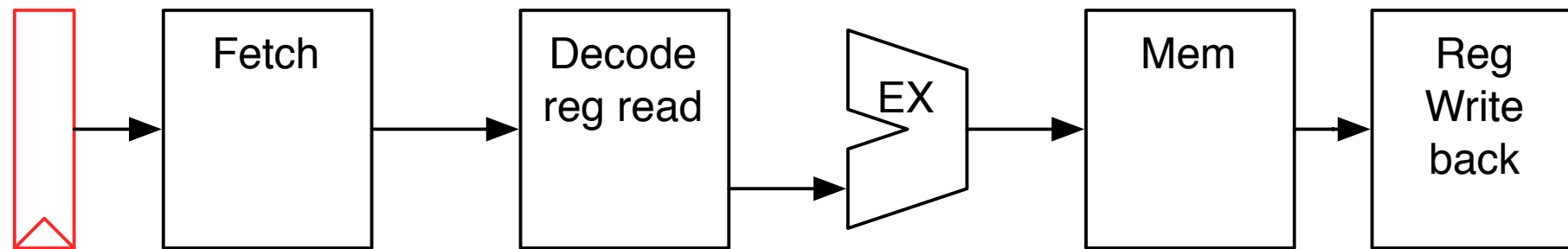


Easier to draw

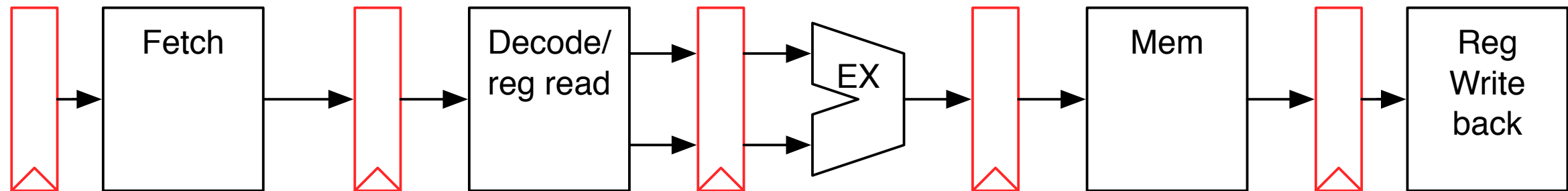
Pipelining a processor



Reality

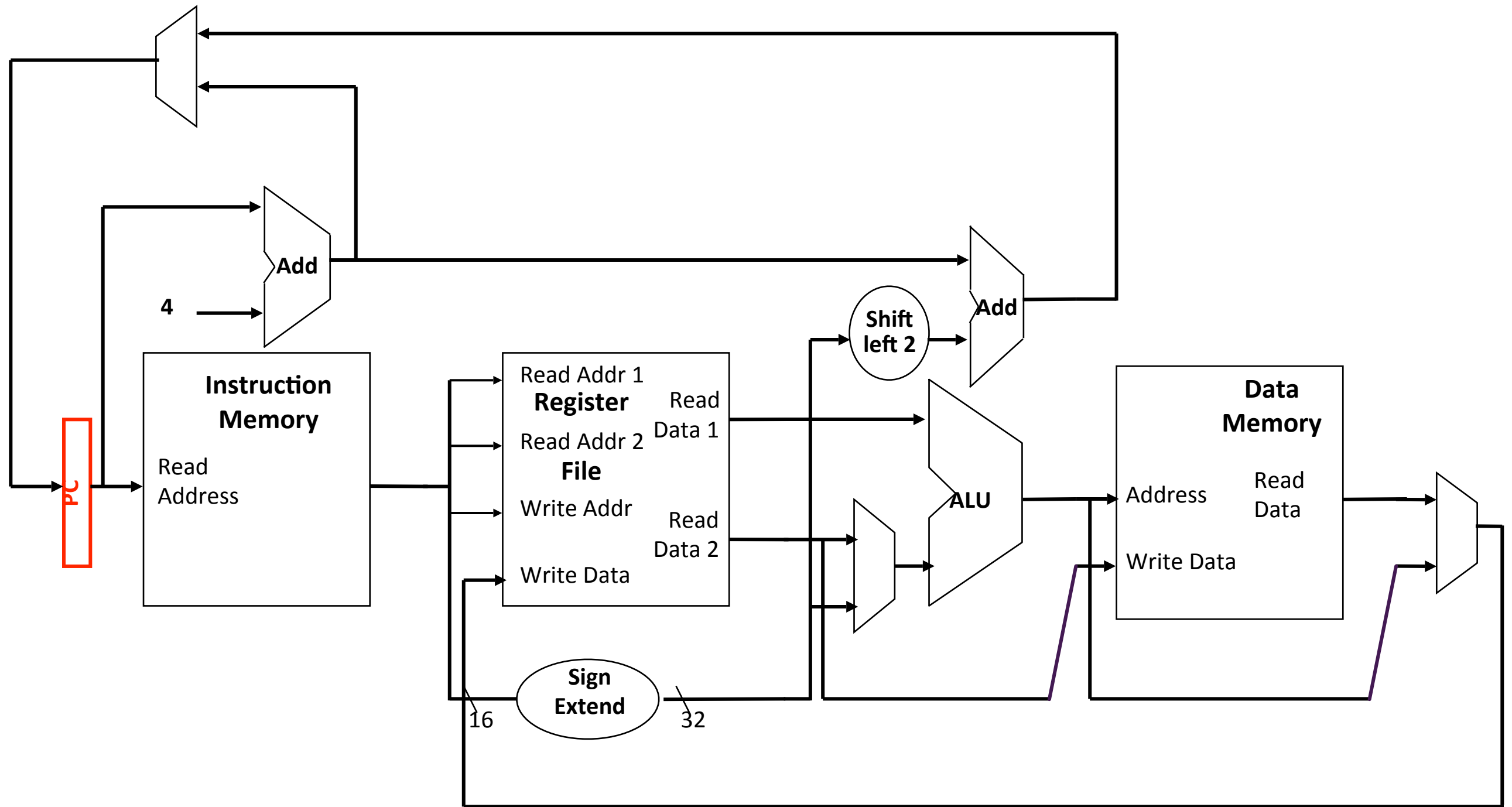


Easier to draw

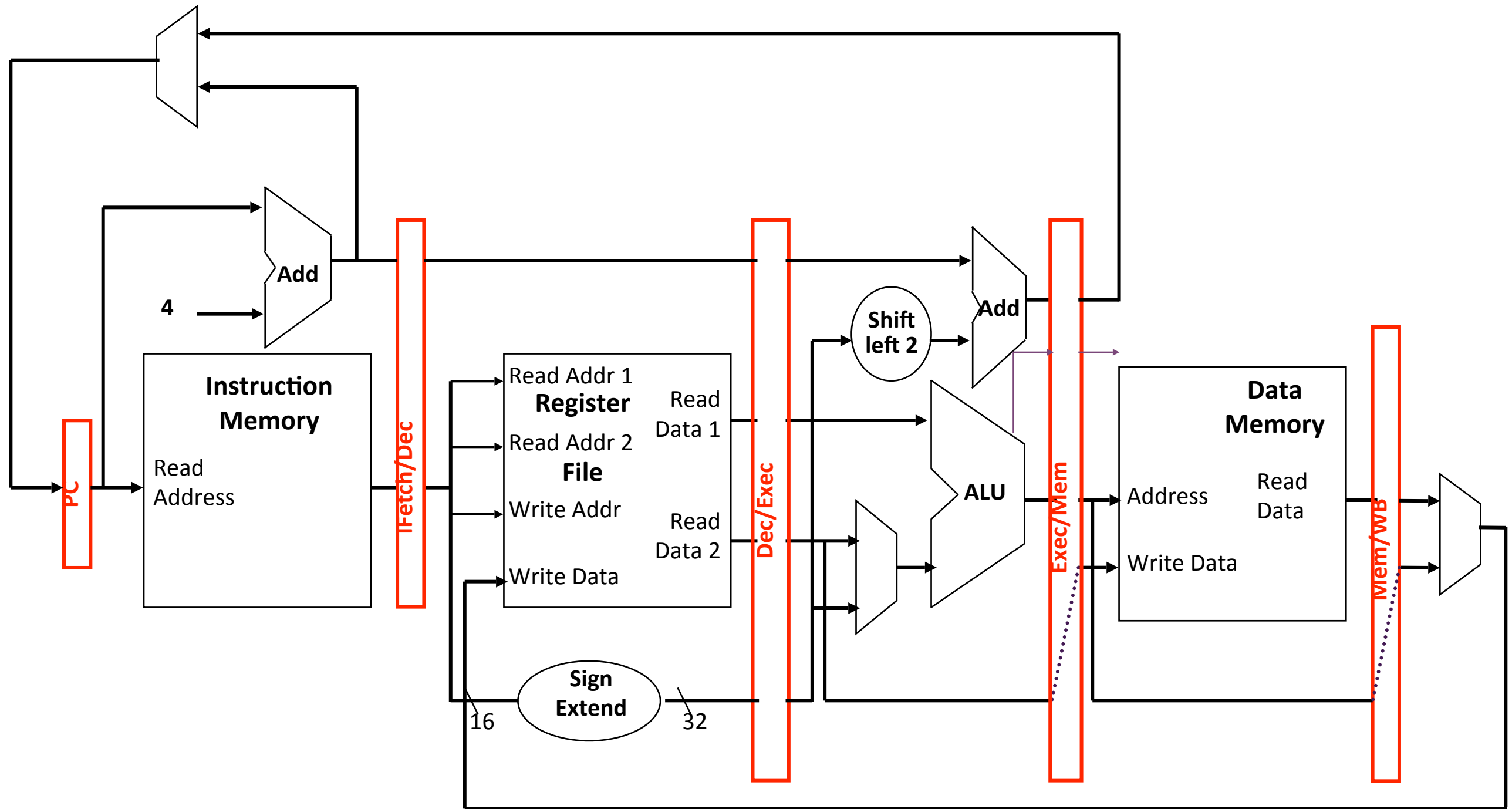


Pipelined

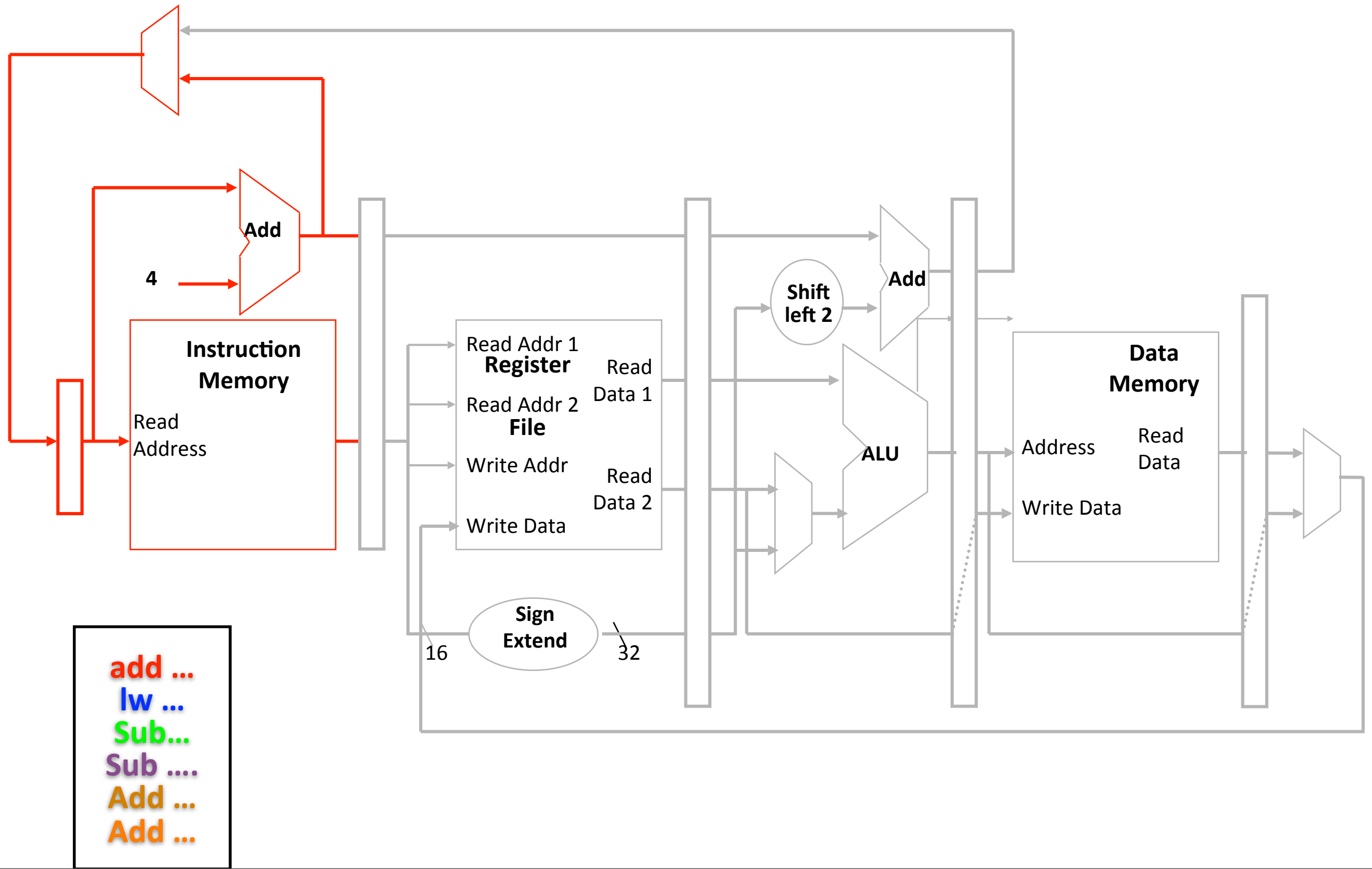
Pipelined Datapath



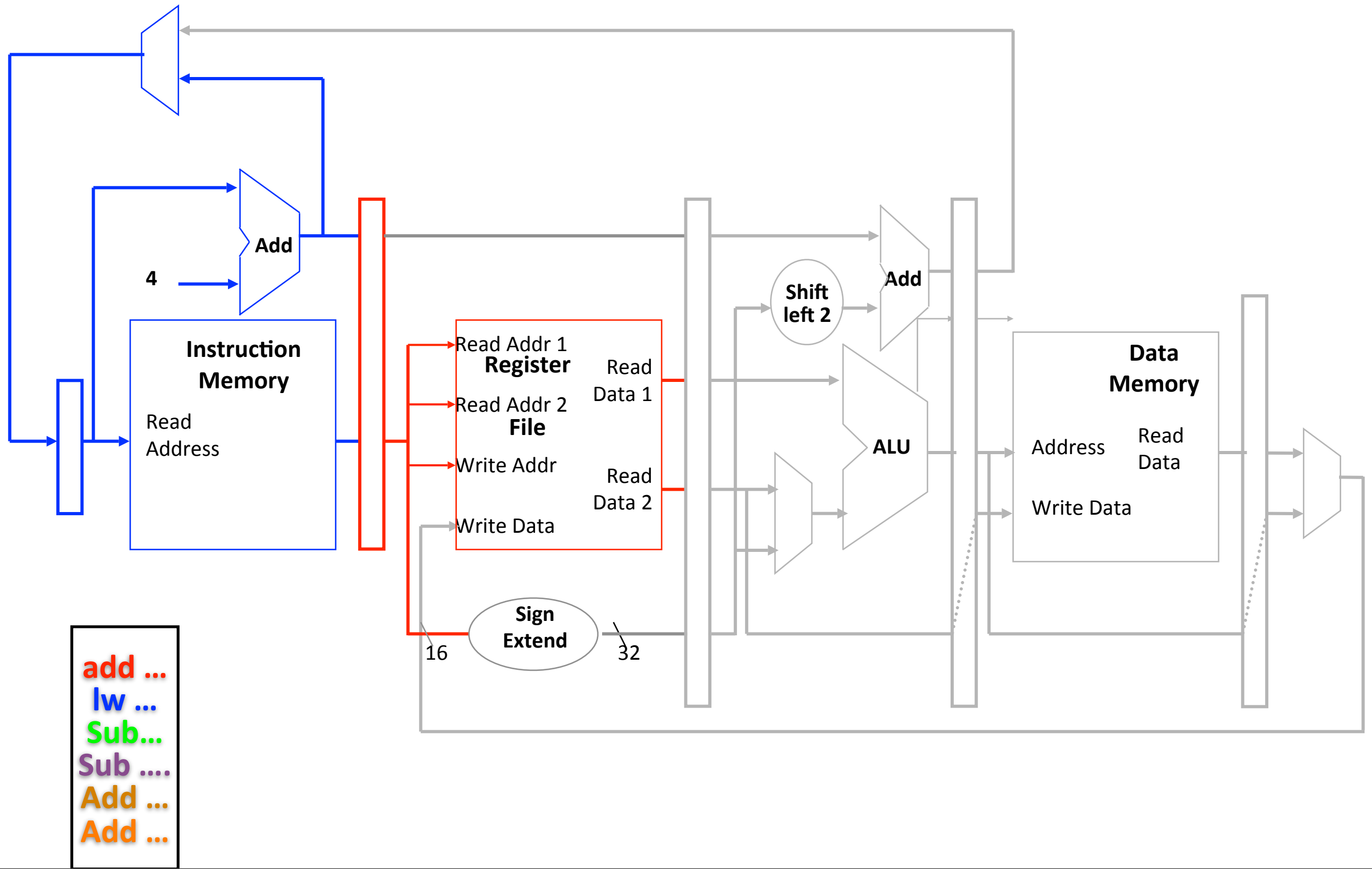
Pipelined Datapath



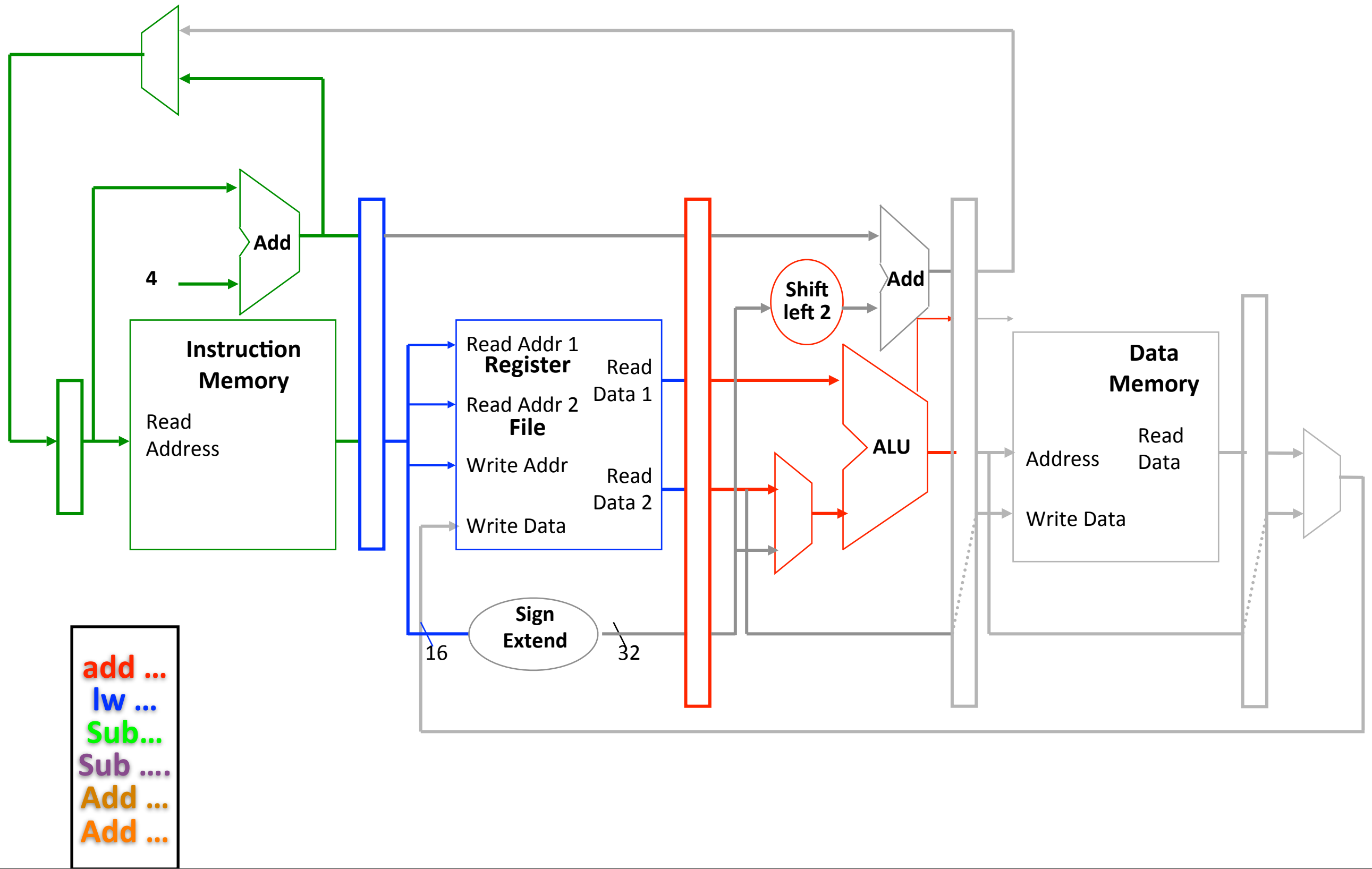
Pipelined Datapath



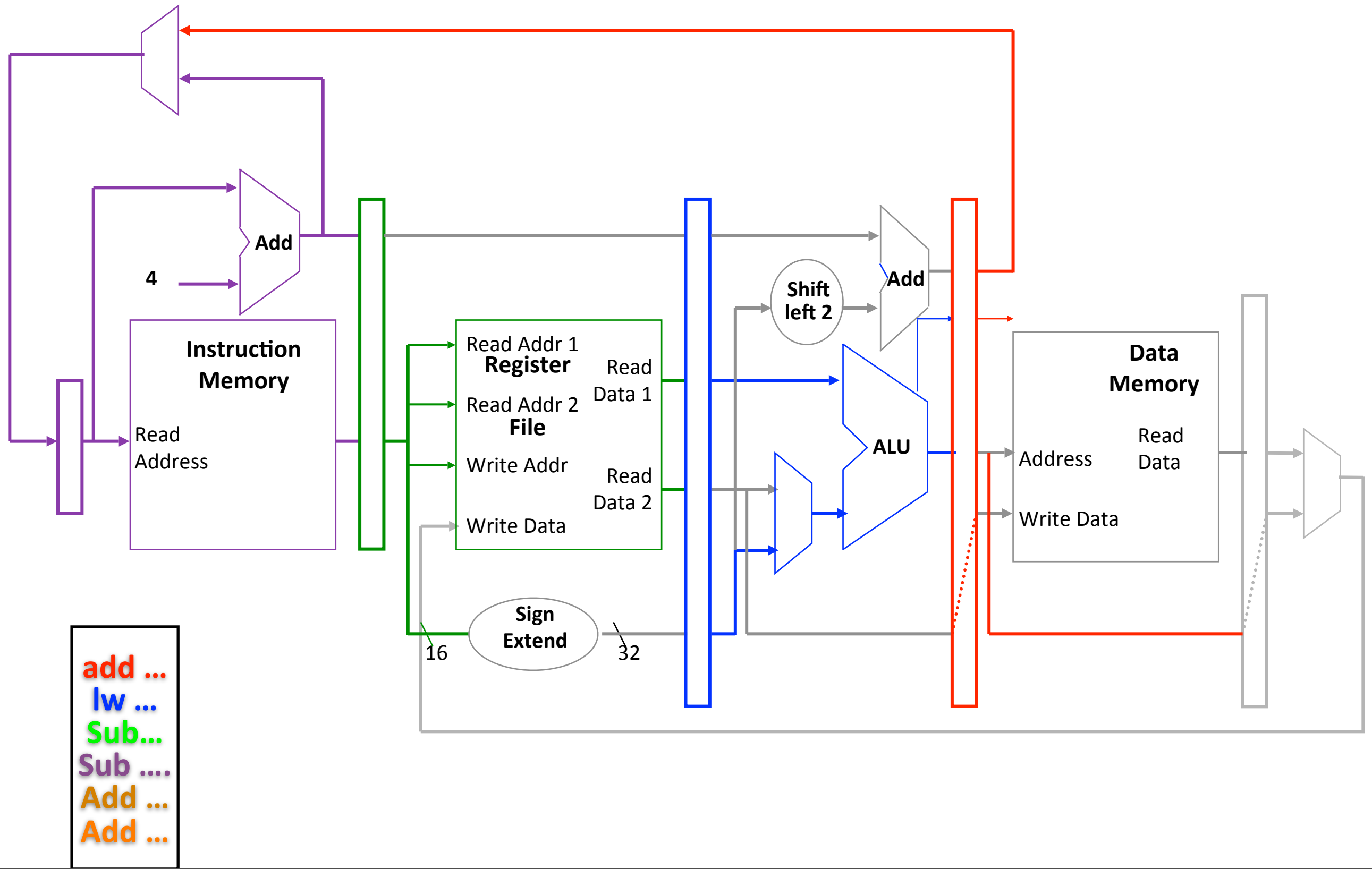
Pipelined Datapath



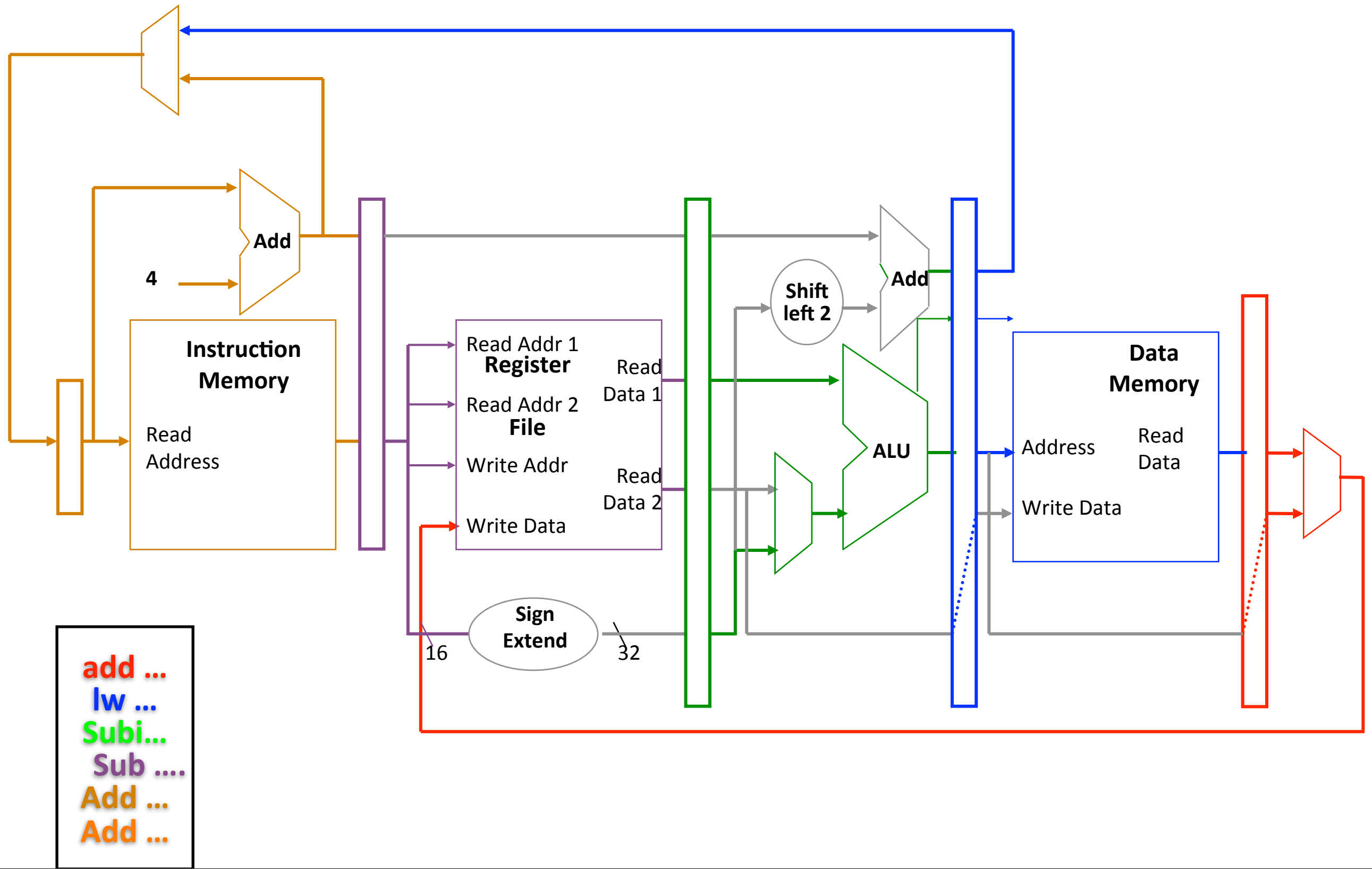
Pipelined Datapath



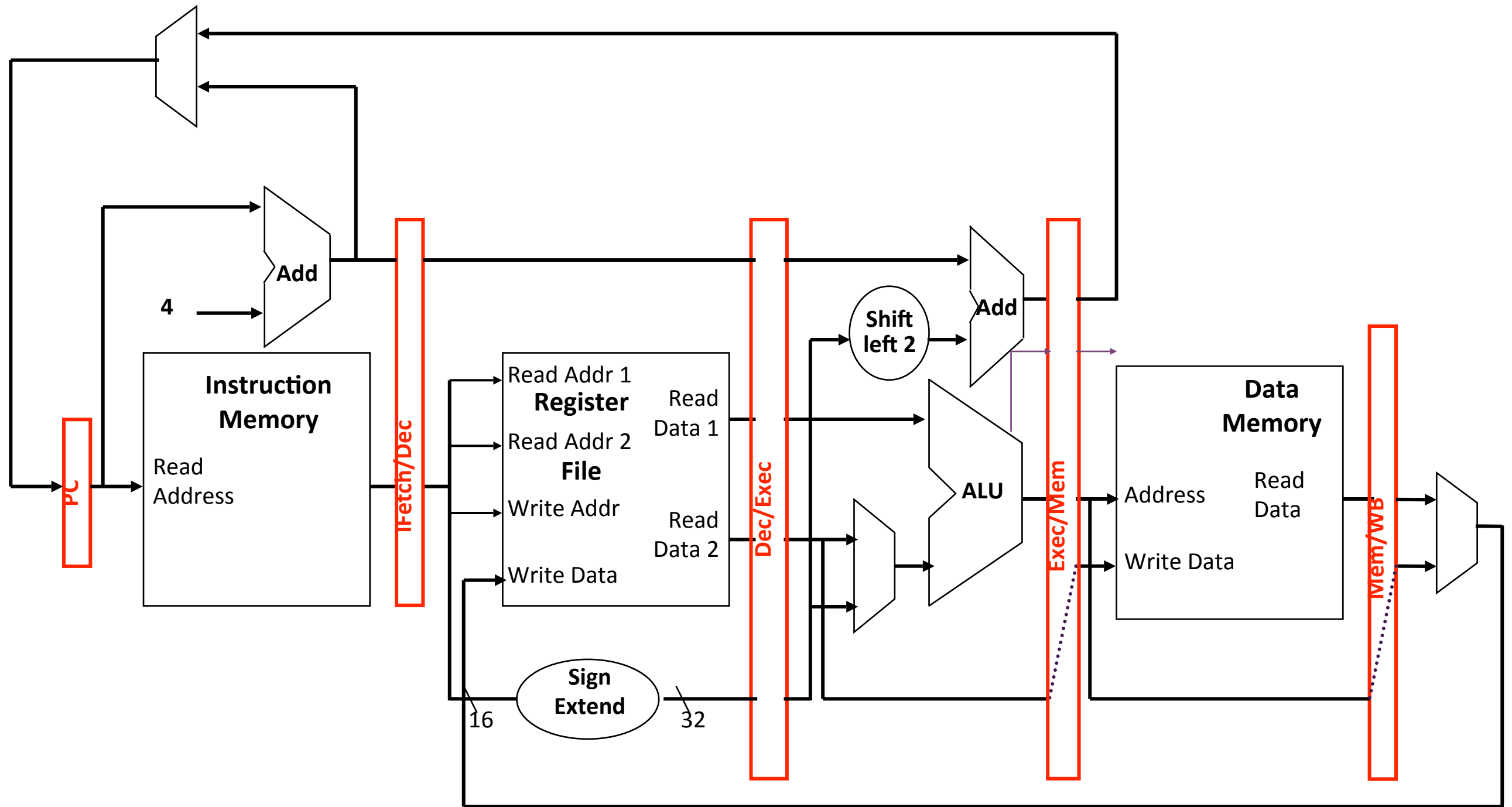
Pipelined Datapath



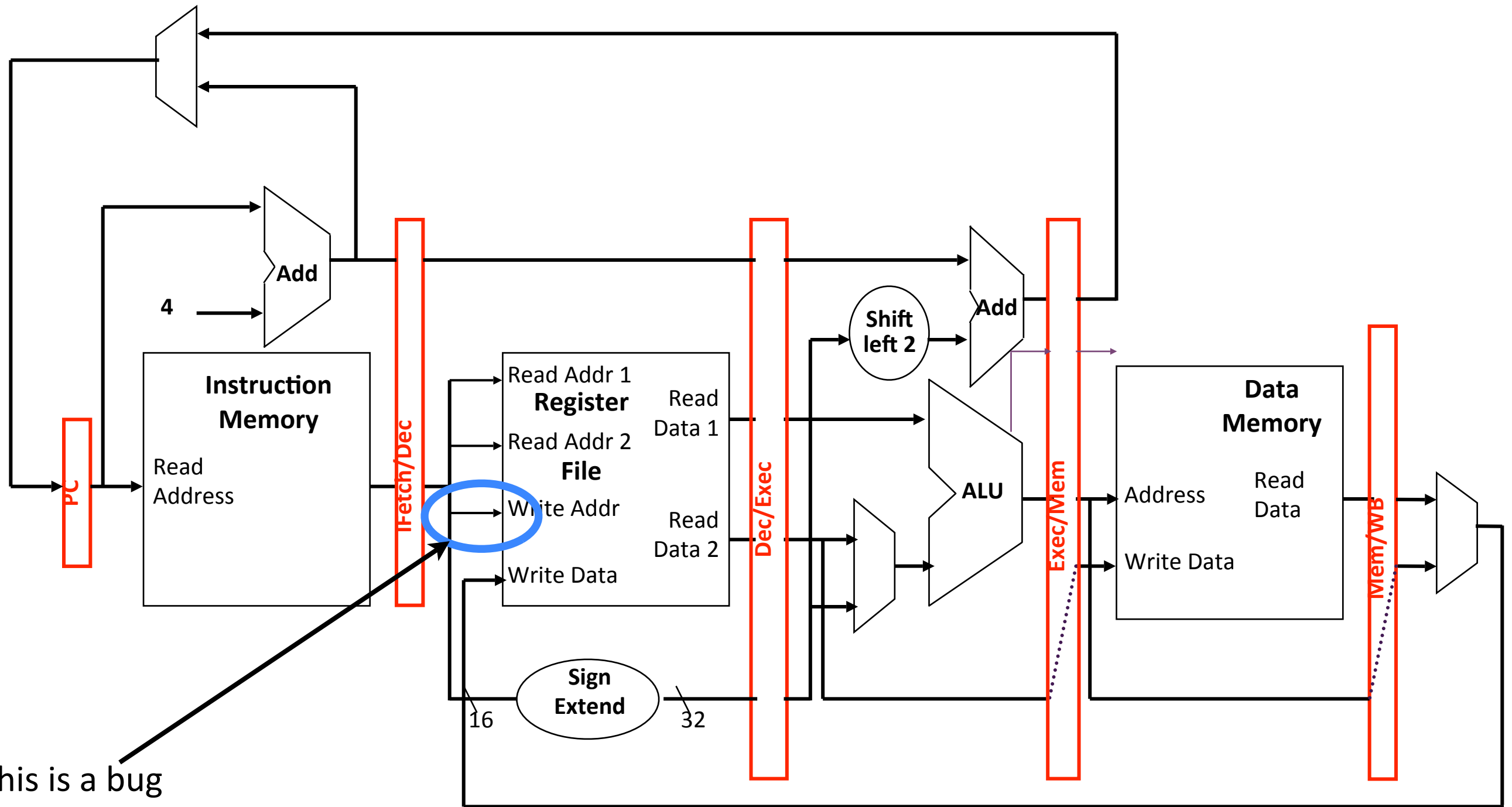
Pipelined Datapath



Pipelined Datapath

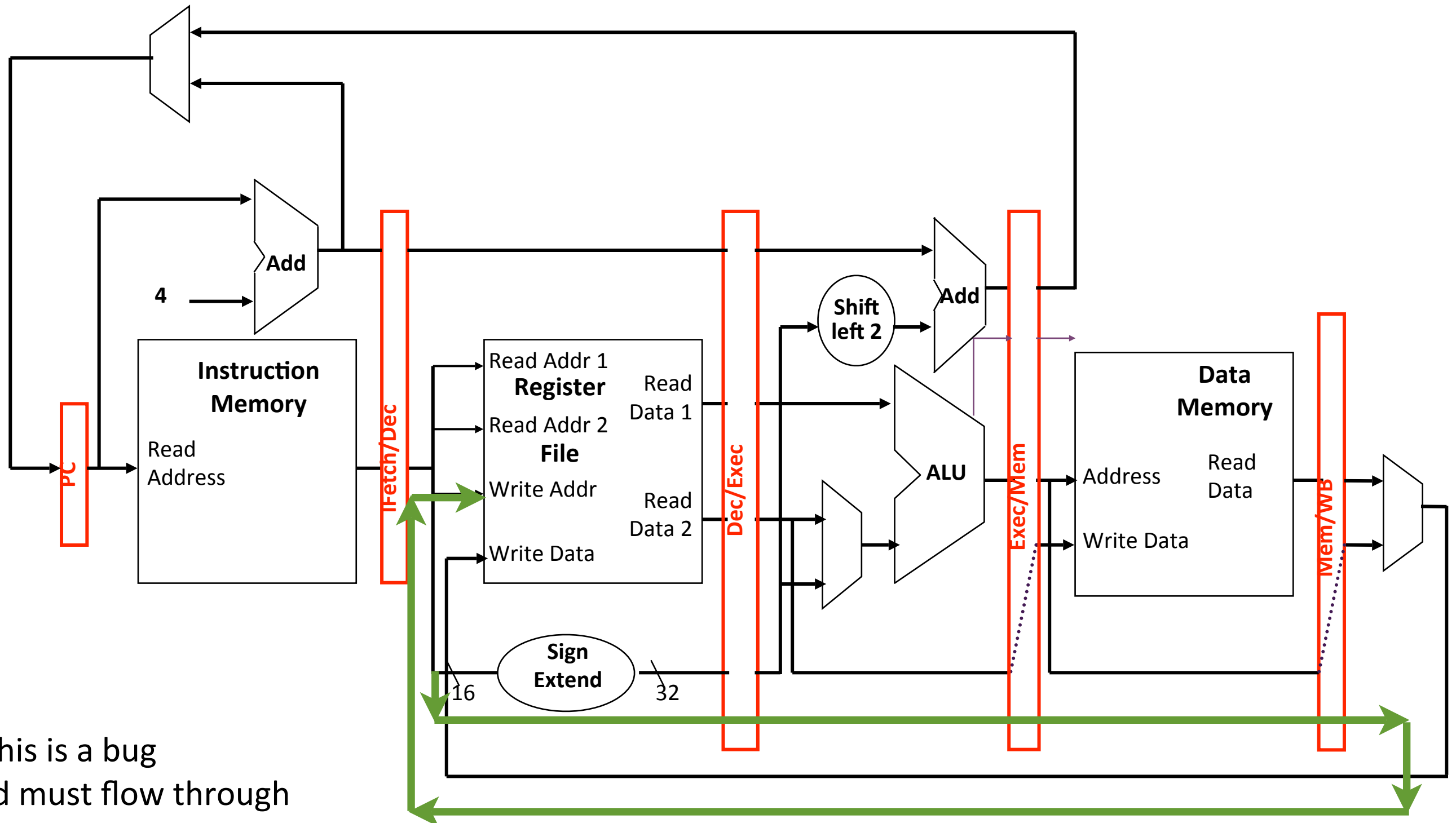


Pipelined Datapath



This is a bug
rd must flow through
the pipeline with the instruction.
This signal needs to come from the WB stage.

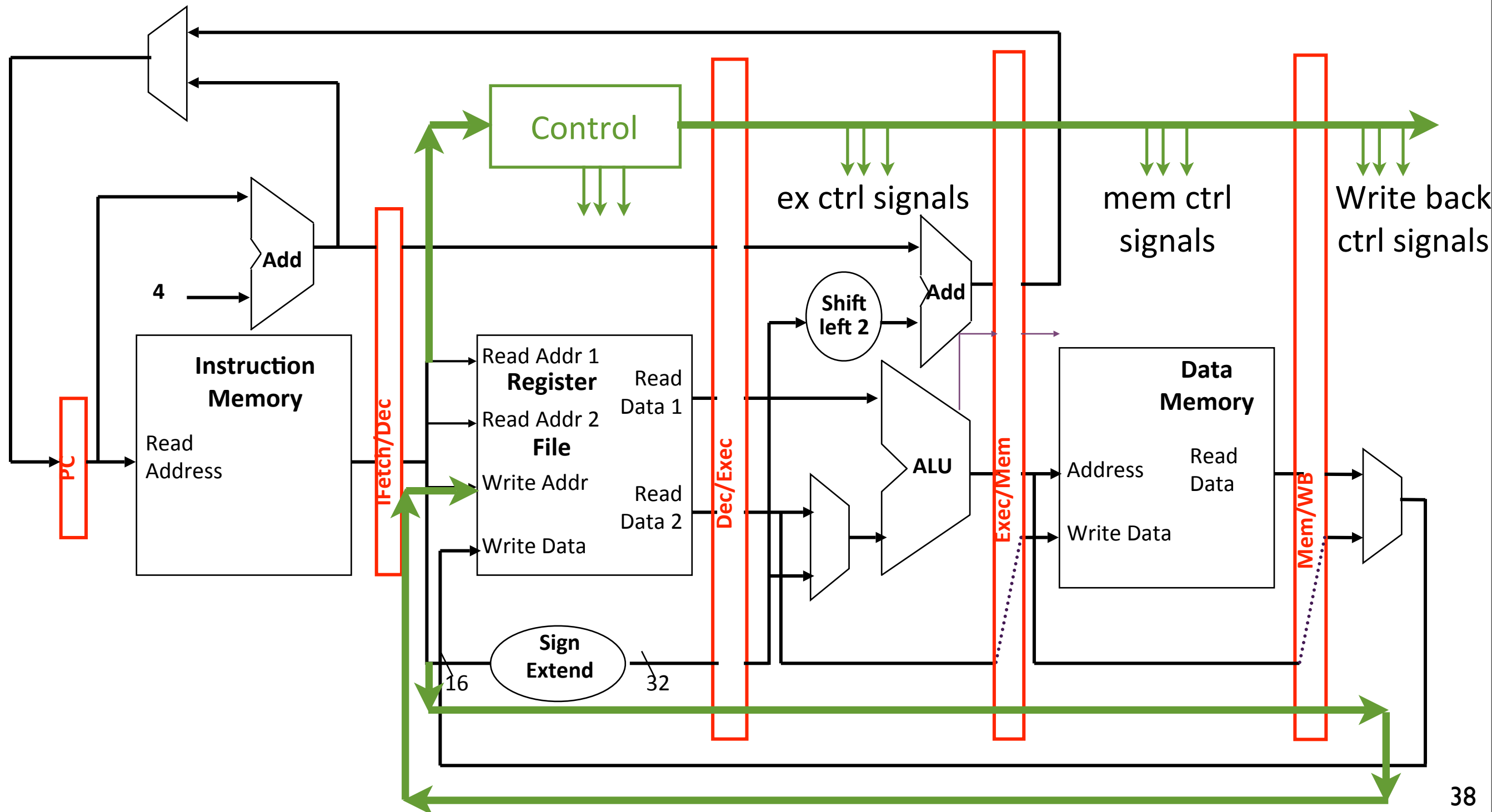
Pipelined Datapath



This is a bug
rd must flow through
the pipeline with the instruction.
This signal needs to come from the WB stage.

Pipelined Control

- Control lives in decode stage, signals flow down the pipe with the instructions they correspond to

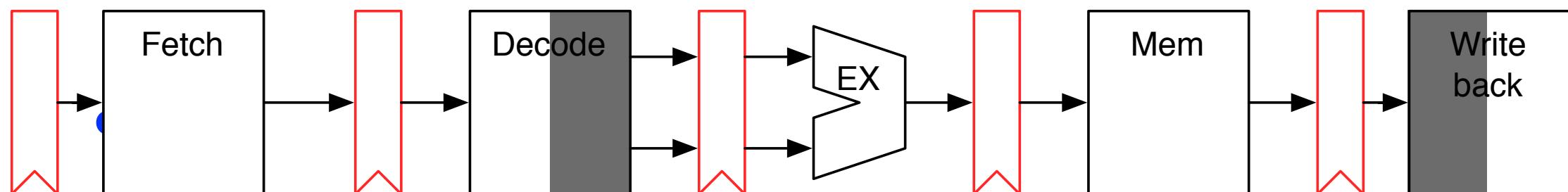


Impact of Pipelining

- $L = IC * CPI * CT$
- Break the processor into P pipe stages
 - $CT_{new} = CT/P$
 - $CPI_{new} = CPI_{old}$
 - CPI is an average: Cycles/instructions
 - The latency of *one instruction* is P cycles
 - The average $CPI = 1$
 - $IC_{new} = IC_{old}$
- Total speedup should be 5x!
 - Except for the overhead of the pipeline registers
 - And the realities of logic design...

A Structural Hazard

- Both the decode and write back stage have to access the register file.
- There is only one registers file. A structural hazard!!
- Solution: Write early, read late
 - Writes occur at the clock edge and complete long before the end of the cycle
 - This leave enough time for the outputs to settle for the reads.
- Hazard avoided!



Pipelining is Tricky

- Simple pipelining is easy
 - If the data flows in one direction only
 - If the stages are independent
 - In fact the tool can do this automatically via “retiming” (If you are curious, experiment with this in Quartus).
- Not so, for processors.
 - Branch instructions affect the next PC -- backward flow
 - Instructions need values computed by previous instructions -- not independent

Not just tricky, Hazardous!

- Hazards are situations where pipelining does not work as elegantly as we would like
- Three kinds
 - Structural hazards -- we have run out of a hardware resource.
 - Data hazards -- an input is not available on the cycle it is needed.
 - Control hazards -- the next instruction is not known.
- Dealing with hazards increases complexity or decreases performance (or both)
- Dealing efficiently with hazards is much of what makes processor design hard.
 - That, and the Quartus tools ;-)

Hazards: Key Points

- Hazards cause imperfect pipelining
 - They prevent us from achieving $CPI = 1$
 - They are generally caused by “counter flow” data dependences in the pipeline
- Three kinds
 - Structural -- contention for hardware resources
 - Data -- a data value is not available when/where it is needed.
 - Control -- the next instruction to execute is not known.
- Two ways to deal with hazards
 - Removal -- add hardware and/or complexity to work around the hazard so it does not occur
 - Stall -- Hold up the execution of new instructions. Let the older instructions finish, so the hazard will clear.

Data Dependences

- A data dependence occurs whenever one instruction needs a value produced by another.
 - Register values
 - Also memory accesses (more on this later)

```
add $s0, $t0, $t1
```

```
sub $t2, $s0, $t3
```

```
add $t3, $s0, $t4
```

```
add $t3, $t2, $t4
```

Data Dependences

- A data dependence occurs whenever one instruction needs a value produced by another.
 - Register values
 - Also memory accesses (more on this later)

```
add $s0, $t0, $t1
```

```
sub $t2, $s0, $t3
```

```
add $t3, $s0, $t4
```

```
add $t3, $t2, $t4
```

Data Dependences

- A data dependence occurs whenever one instruction needs a value produced by another.
 - Register values
 - Also memory accesses (more on this later)

```
add $s0, $t0, $t1
```

```
sub $t2, $s0, $t3
```

```
add $t3, $s0, $t4
```

```
add $t3, $t2, $t4
```

Data Dependences

- A data dependence occurs whenever one instruction needs a value produced by another.
 - Register values
 - Also memory accesses (more on this later)

add \$s0, \$t0, \$t1

sub \$t2, \$s0, \$t3

add \$t3, \$s0, \$t4

add \$t3, \$t2, \$t4

Data Dependences

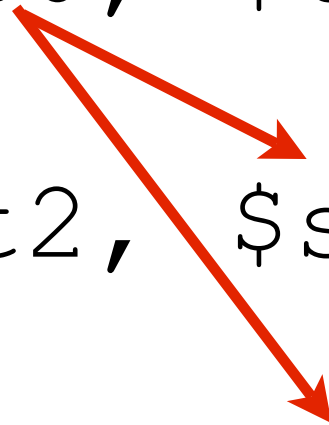
- A data dependence occurs whenever one instruction needs a value produced by another.
 - Register values
 - Also memory accesses (more on this later)

add \$s0, \$t0, \$t1

sub \$t2, \$s0, \$t3

add \$t3, \$s0, \$t4

add \$t3, \$t2, \$t4



Data Dependences

- A data dependence occurs whenever one instruction needs a value produced by another.
 - Register values
 - Also memory accesses (more on this later)

```
graph TD; I1[add $s0, $t0, $t1] --> I2[sub $t2, $s0, $t3]; I1 --> I3[add $t3, $s0, $t4]; I2 --> I4[add $t3, $t2, $t4];
```

add \$s0, \$t0, \$t1

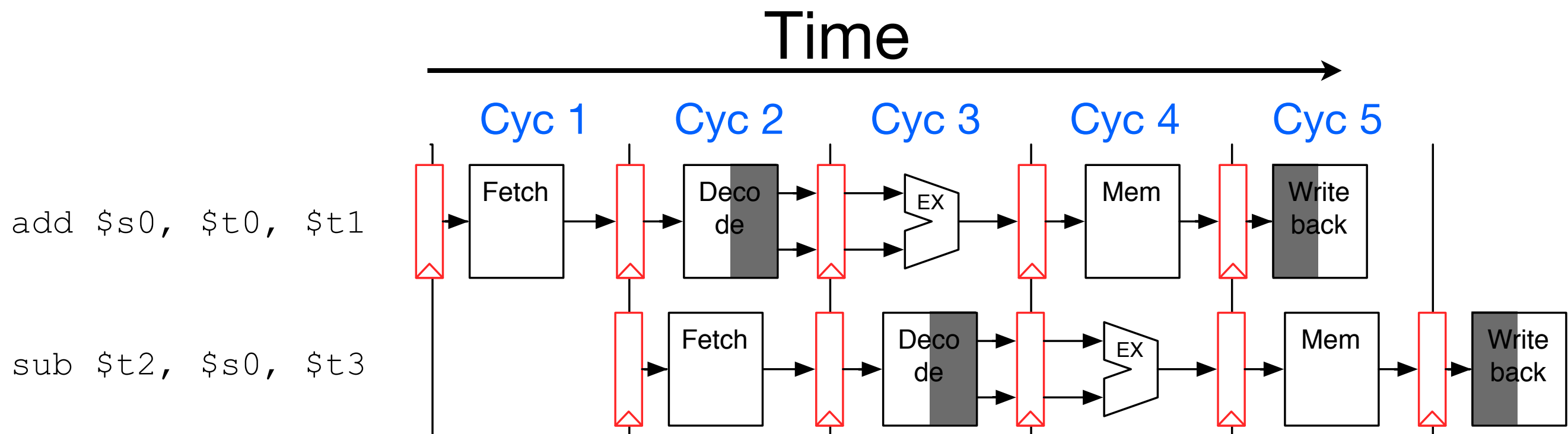
sub \$t2, \$s0, \$t3

add \$t3, \$s0, \$t4

add \$t3, \$t2, \$t4

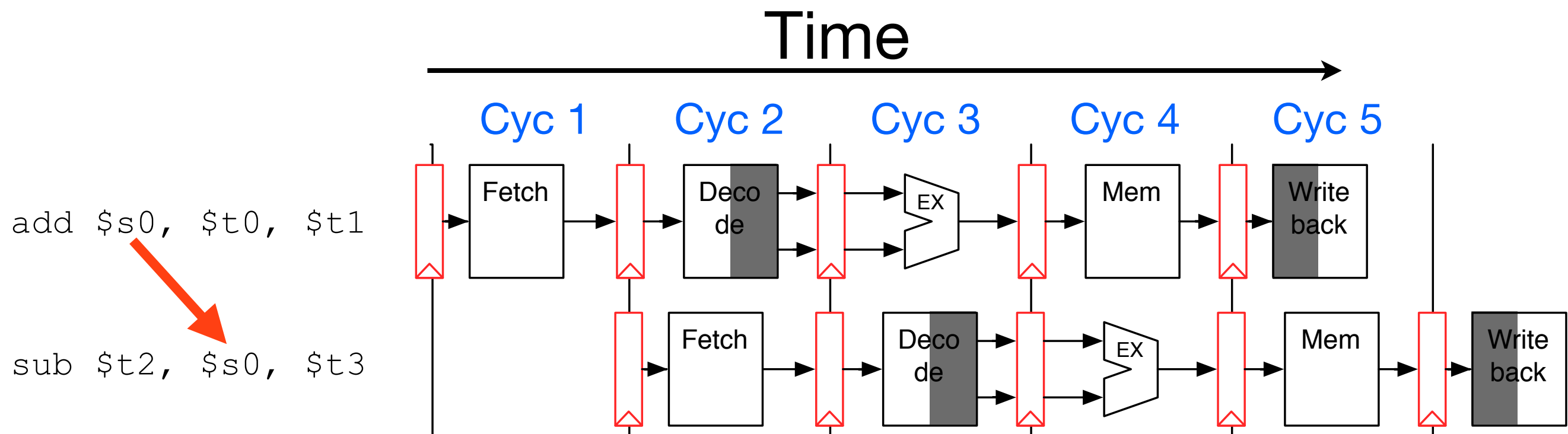
Dependences in the pipeline

- In our simple pipeline, these instructions cause a data hazard



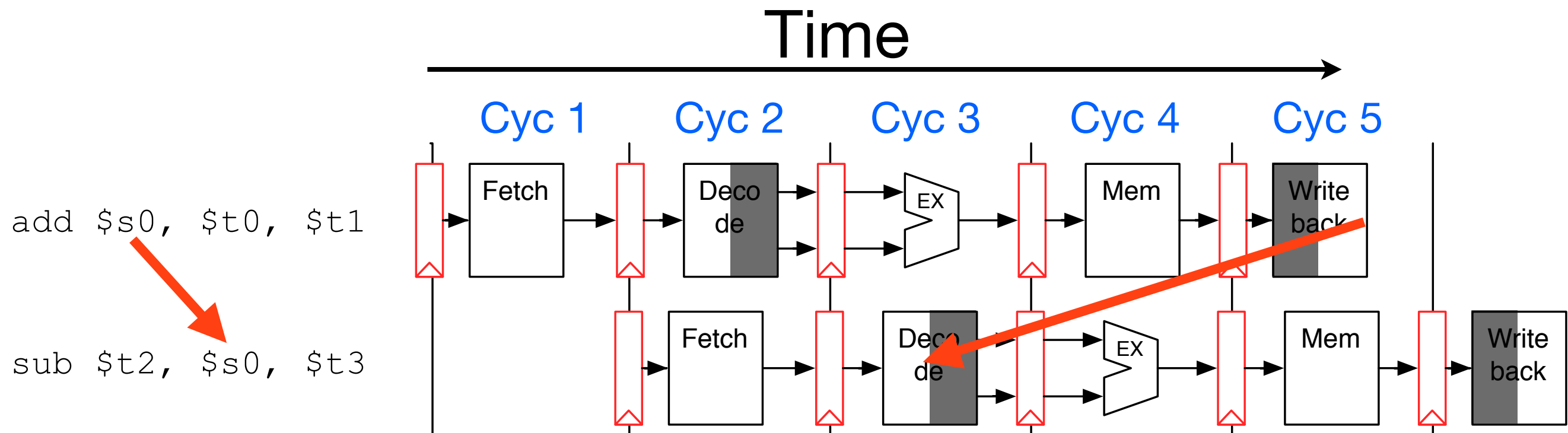
Dependences in the pipeline

- In our simple pipeline, these instructions cause a data hazard



Dependences in the pipeline

- In our simple pipeline, these instructions cause a data hazard

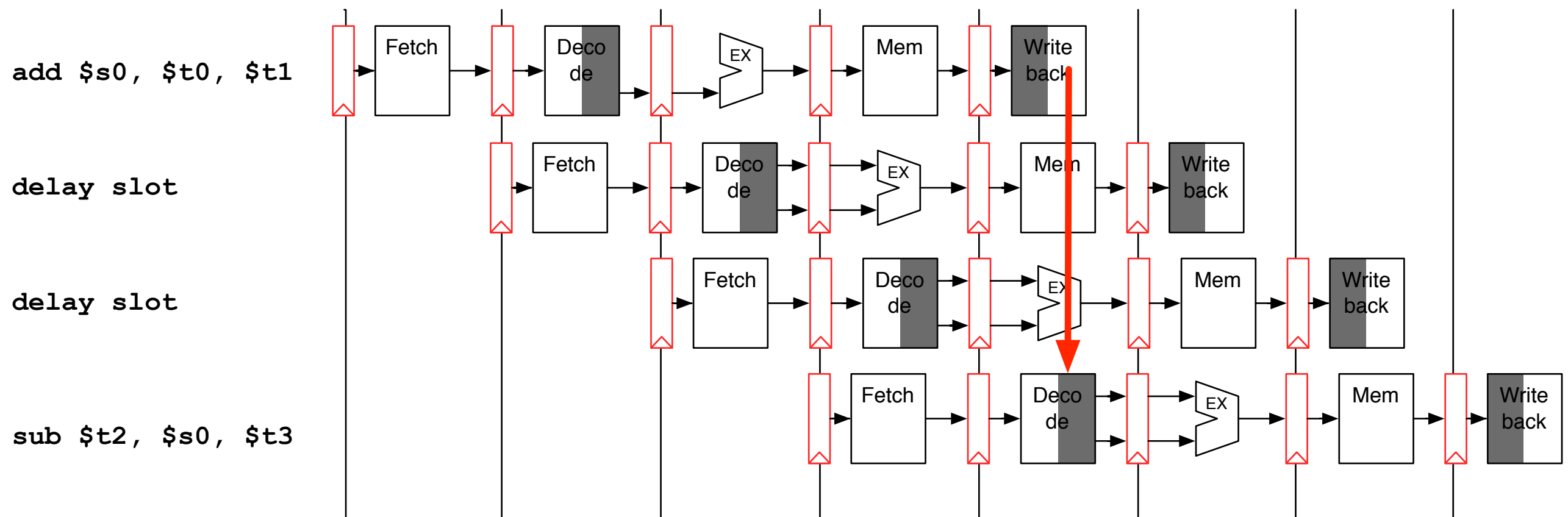


How can we fix it?

- Ideas?

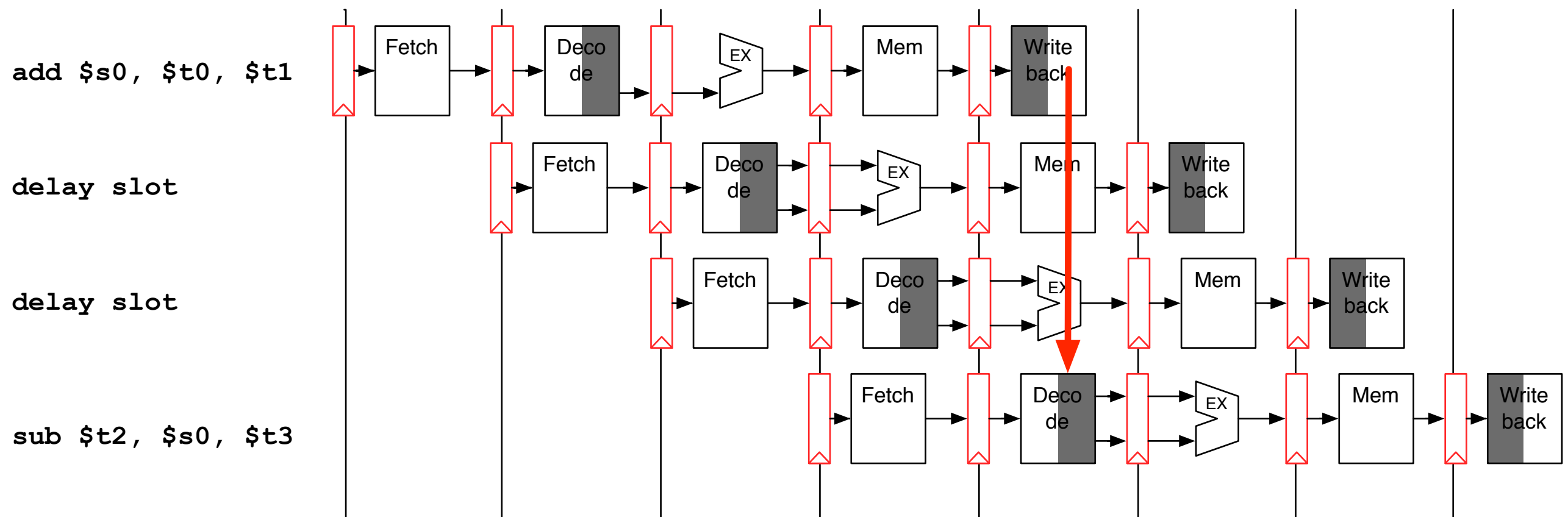
Solution 1: Make the compiler deal with it.

- Expose hazards to the big A architecture
 - A result is available N instructions after the instruction that generates it.
 - In the meantime, the register file has the old value.
 - This is called “a register delay slot”
- What is N? Can it change?



Solution 1: Make the compiler deal with it.

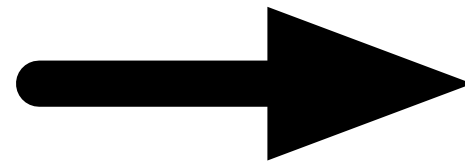
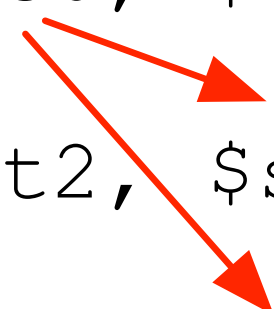
- Expose hazards to the big A architecture
 - A result is available N instructions after the instruction that generates it.
 - In the meantime, the register file has the old value.
 - This is called “a register delay slot”
- What is N? Can it change? **N = 2, for our design**



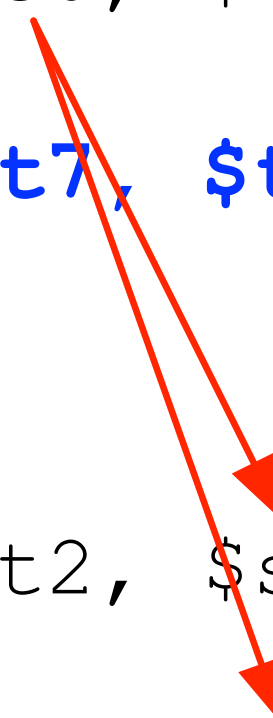
Compiling for delay slots

- The compiler must fill the delay slots
- Ideally, with useful instructions, but nops will work too.

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
add $t3, $s0, $t4
and $t7, $t5, $t4
```

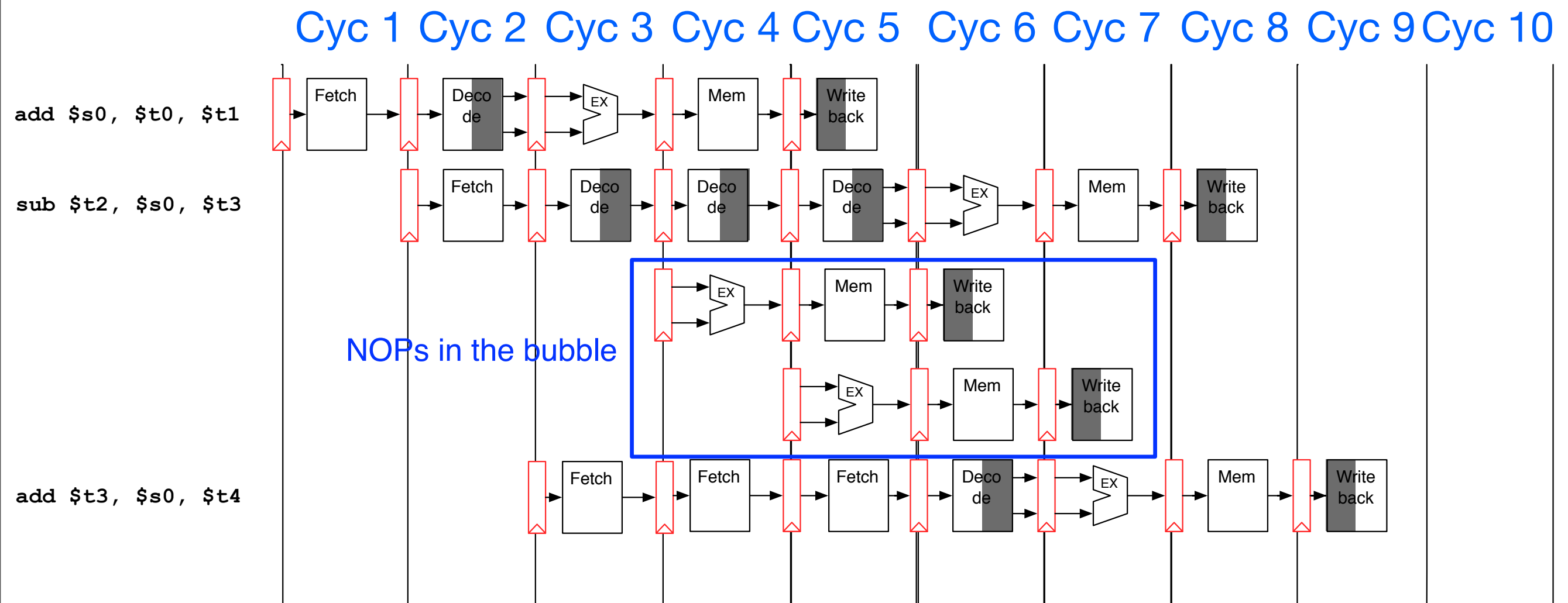


```
add $s0, $t0, $t1
and $t7, $t5, $t4
nop
sub $t2, $s0, $t3
add $t3, $s0, $t4
```



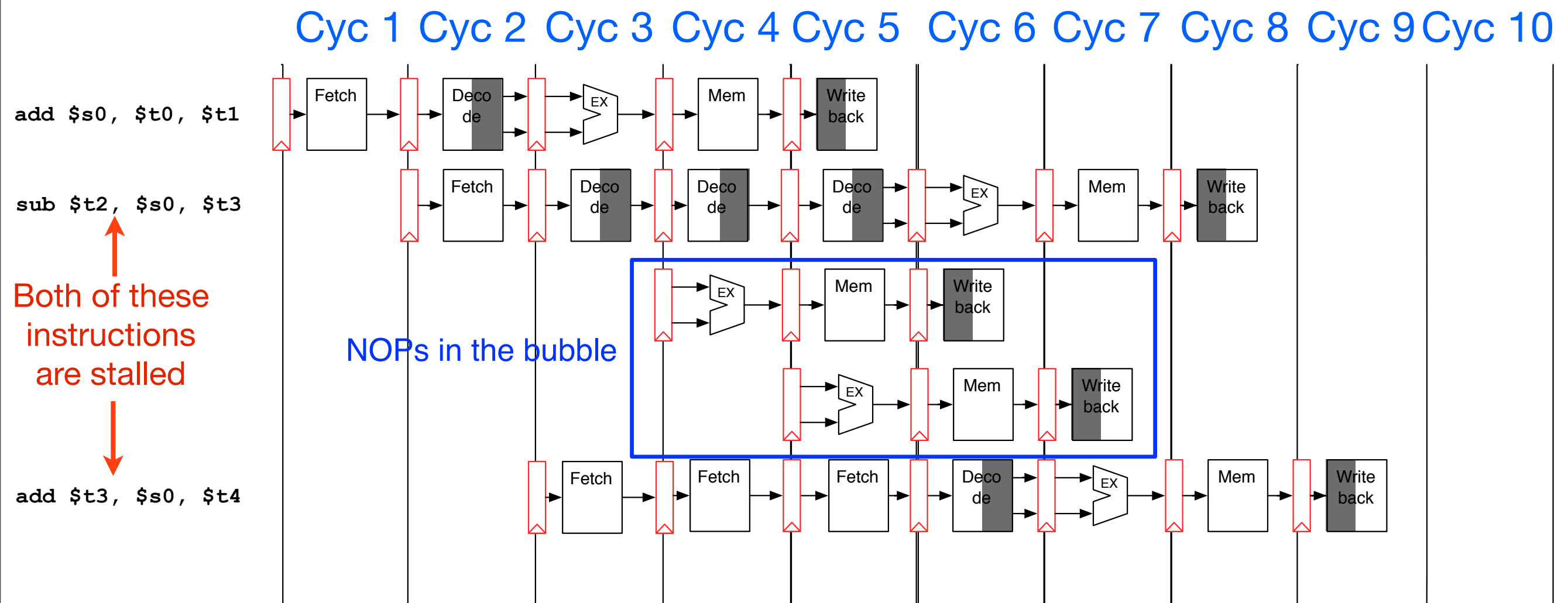
Solution 2: Stall

- When you need a value that is not ready, “stall”
 - Suspend the execution of the executing instruction
 - and those that follow.
 - This introduces a pipeline “bubble.”
- A bubble is a lack of work to do, it propagates through the pipeline like nop instructions



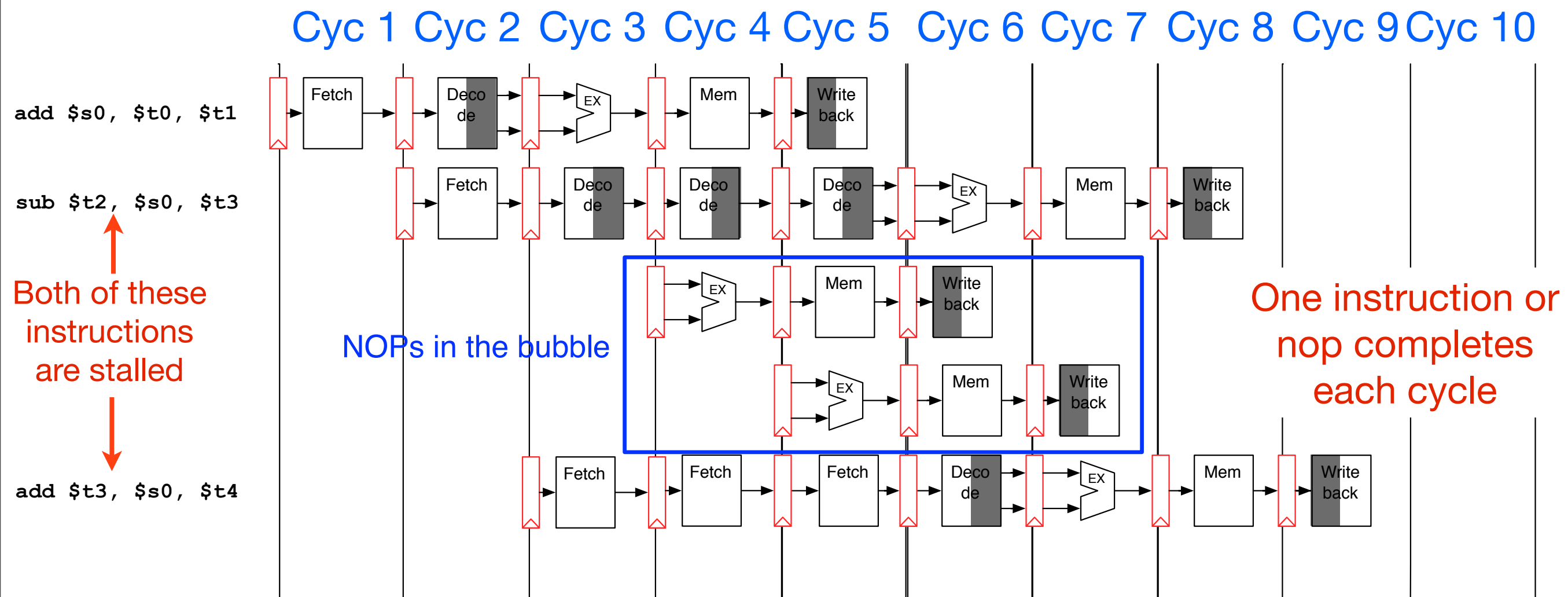
Solution 2: Stall

- When you need a value that is not ready, “stall”
 - Suspend the execution of the executing instruction
 - and those that follow.
 - This introduces a pipeline “bubble.”
- A bubble is a lack of work to do, it propagates through the pipeline like nop instructions



Solution 2: Stall

- When you need a value that is not ready, “stall”
 - Suspend the execution of the executing instruction
 - and those that follow.
 - This introduces a pipeline “bubble.”
- A bubble is a lack of work to do, it propagates through the pipeline like nop instructions



Stalling the pipeline

- Freeze all pipeline stages before the stage where the hazard occurred.
 - Disable the PC update
 - Disable the pipeline registers
- This is equivalent to inserting into the pipeline when a hazard exists
 - Insert nop control bits at stalled stage (decode in our example)
 - How is this solution still potentially “better” than relying on the compiler?

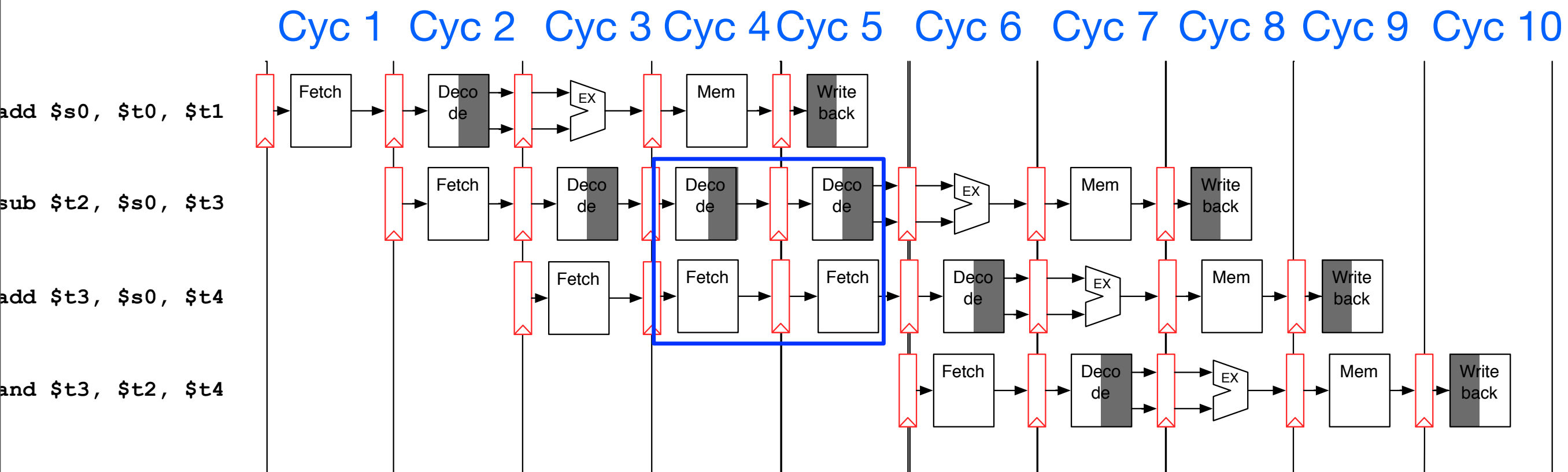
Stalling the pipeline

- Freeze all pipeline stages before the stage where the hazard occurred.
 - Disable the PC update
 - Disable the pipeline registers
- This is equivalent to inserting into the pipeline when a hazard exists
 - Insert nop control bits at stalled stage (decode in our example)
 - How is this solution still potentially “better” than relying on the compiler?

The compiler can still act like there are delay slots to avoid stalls.
Implementation details are not exposed in the ISA

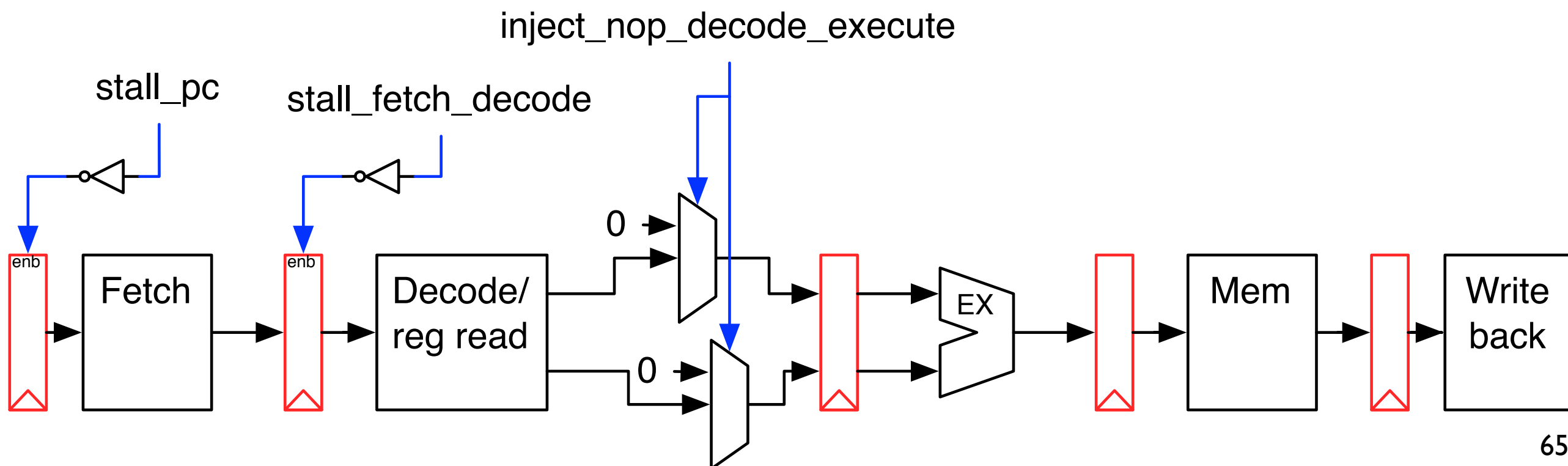
Calculating CPI for Stalls

- In this case, the bubble lasts for 2 cycles.
 - As a result, in cycle (6 and 7), no instruction completes.
 - What happens to CPI?
- In the absence of stalls, CPI is one, since one instruction completes per cycle
 - If an instruction stalls for N cycles, it's CPI goes up by N



Hardware for Stalling

- Turn off the enables on the earlier pipeline stages
 - The earlier stages will keep processing the same instruction over and over.
 - No new instructions get fetched.
- Insert control and data values corresponding to a nop into the “downstream” pipeline register.
 - This will create the bubble.
 - The nops will flow downstream, doing nothing.
- When the stall is over, re-enable the pipeline registers
 - The instructions in the “upstream” stages will start moving again.
 - New instructions will start entering the pipeline again.



The Impact of Stalling On Performance

- $ET = I * CPI * CT$
- I and CT are constant
- What is the impact of stalling on CPI ?
- What do we need to know to figure it out?

The Impact of Stalling On Performance

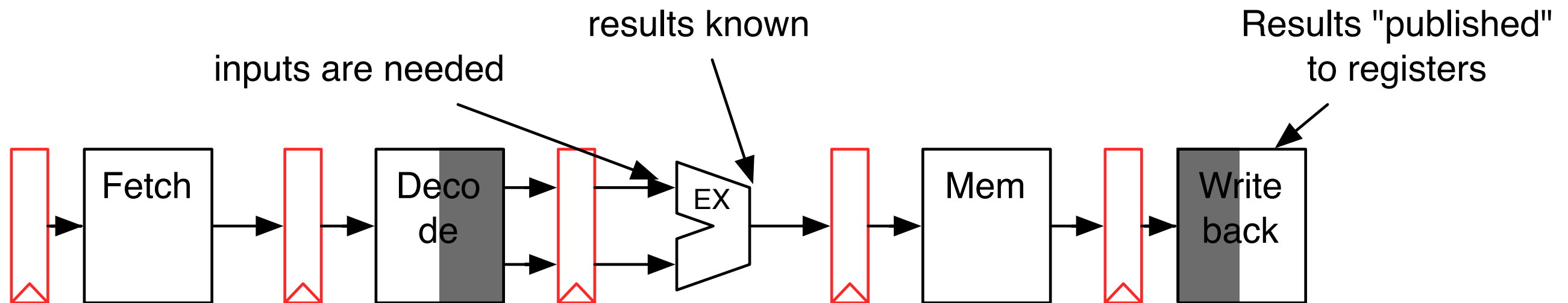
- $ET = I * CPI * CT$
- I and CT are constant
- What is the impact of stalling on CPI ?
- Fraction of instructions that stall: 30%
- Baseline $CPI = 1$
- Stall $CPI = 1 + 2 = 3$
- New $CPI =$

The Impact of Stalling On Performance

- $ET = I * CPI * CT$
- I and CT are constant
- What is the impact of stalling on CPI ?
- Fraction of instructions that stall: 30%
- Baseline $CPI = 1$
- Stall $CPI = 1 + 2 = 3$
- New $CPI =$
 $0.3 * 3 + 0.7 * 1 = 1.6$

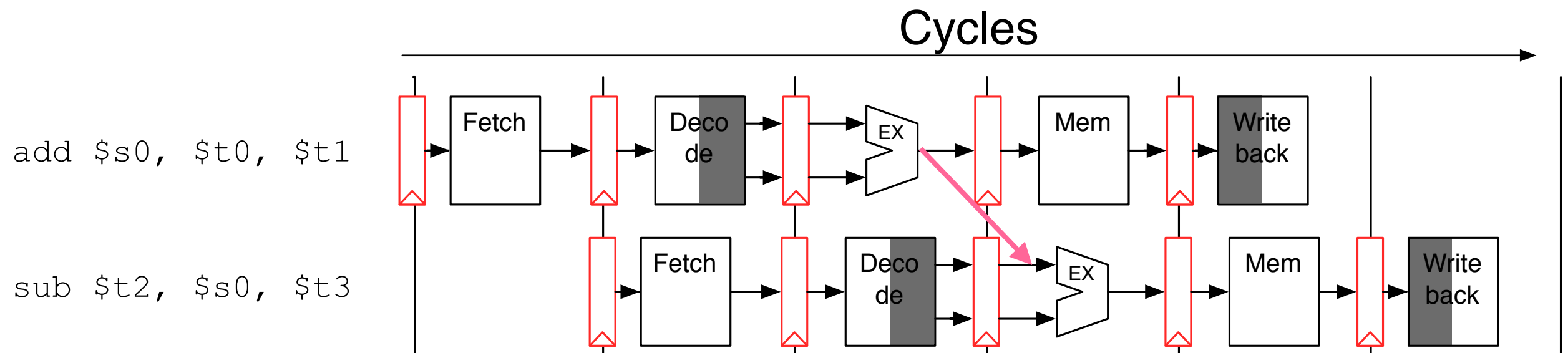
Solution 3: Bypassing/

- Data values are computed in Ex and Mem but “publicized in write back”
- The data exists! We should use it.



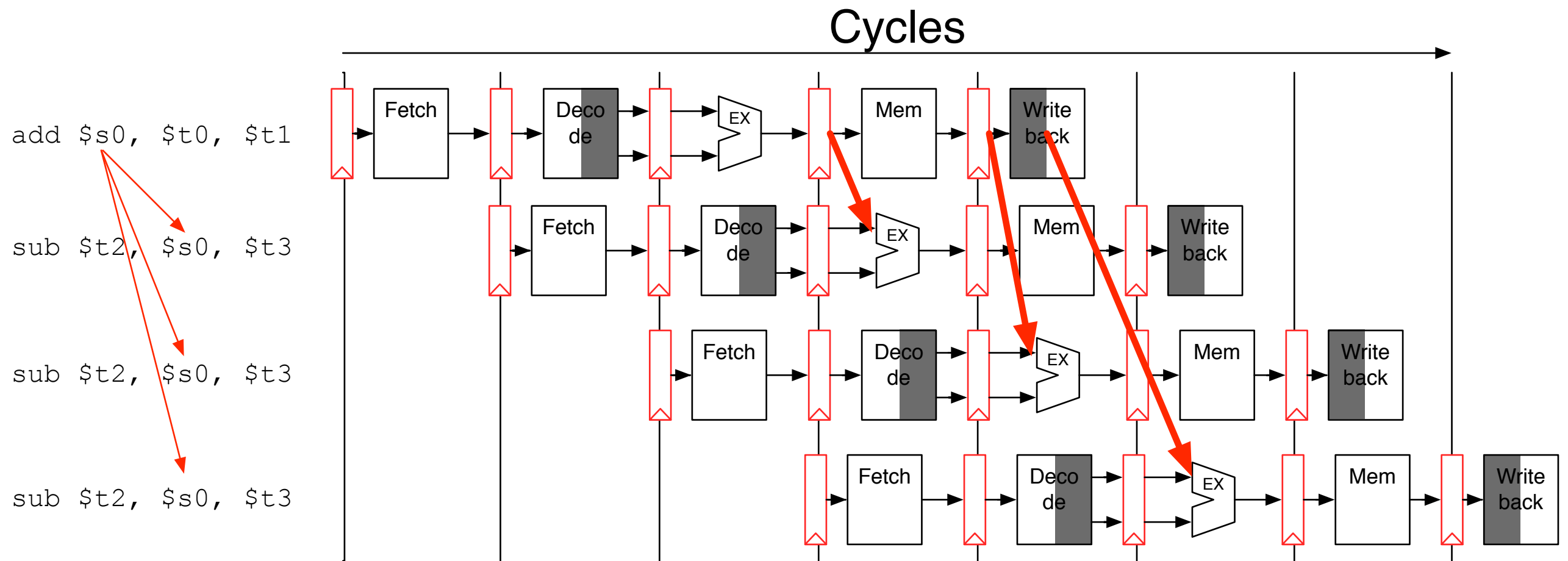
Bypassing or Forwarding

- Take the values, where ever they are

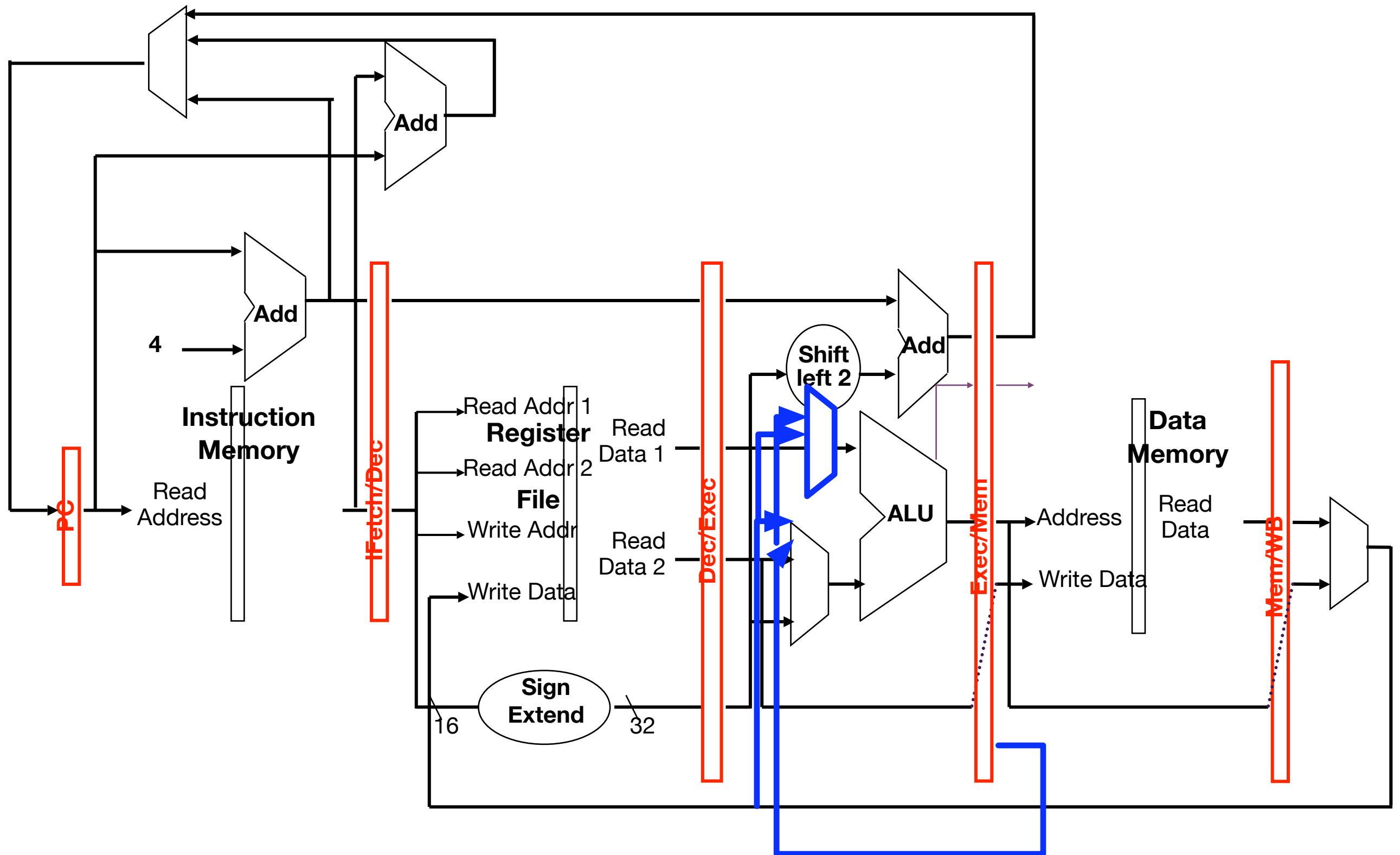


•

Forwarding Paths



Forwarding in Hardware

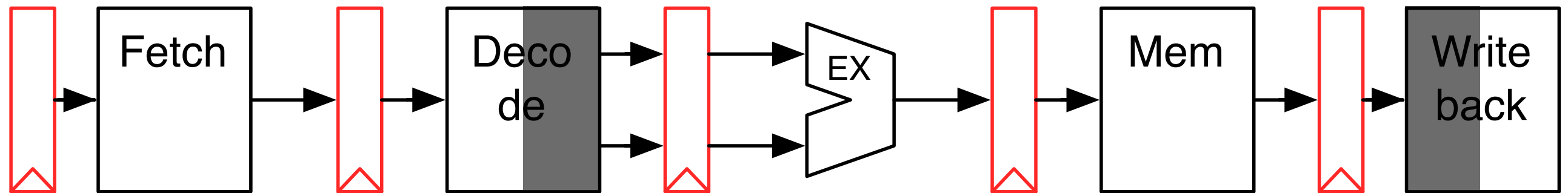


Hardware Cost of Forwarding

- In our pipeline, adding forwarding required relatively little hardware.
- For deeper pipelines it gets much more expensive
 - Roughly: $\text{ALU} * \text{pipe_stages}$ you need to forward over
 - Some modern processor have multiple ALUs (4-5)
 - And deeper pipelines (4-5 stages of to forward across)
- Not all forwarding paths need to be supported.
 - If a path does not exist, the processor will need to stall.

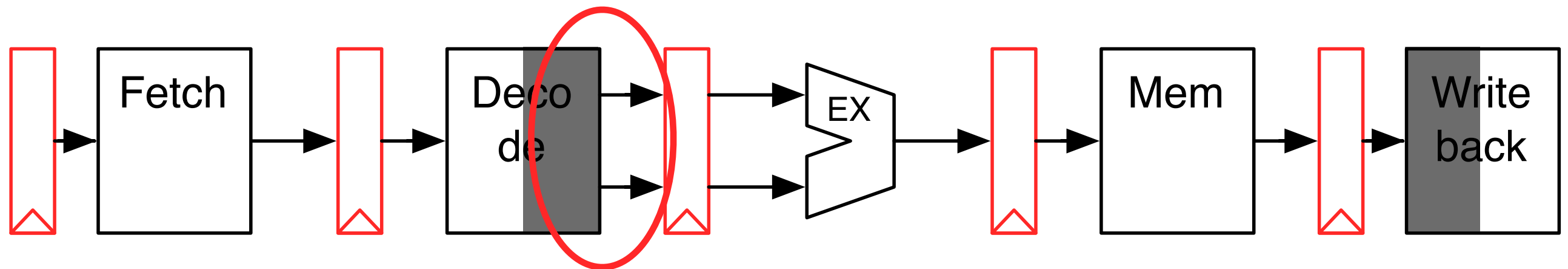
Computing the PC Normally

- Non-branch instruction
 - $PC = PC + 4$
- When is PC ready?



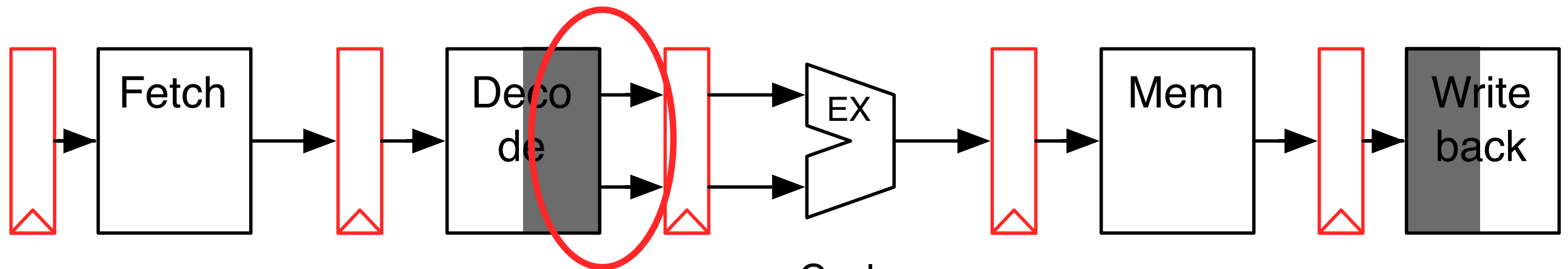
Computing the PC Normally

- Non-branch instruction
 - $PC = PC + 4$
- When is PC ready?

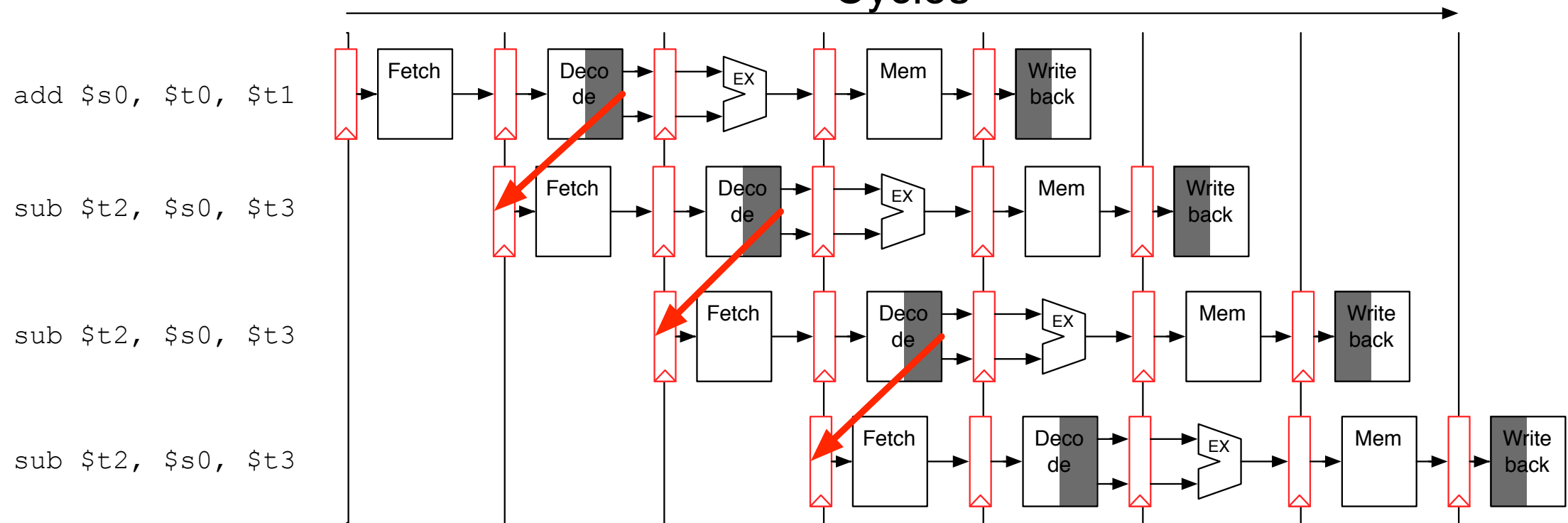


Computing the PC Normally

- Non-branch instruction
 - $PC = PC + 4$
- When is PC ready?



Cycles

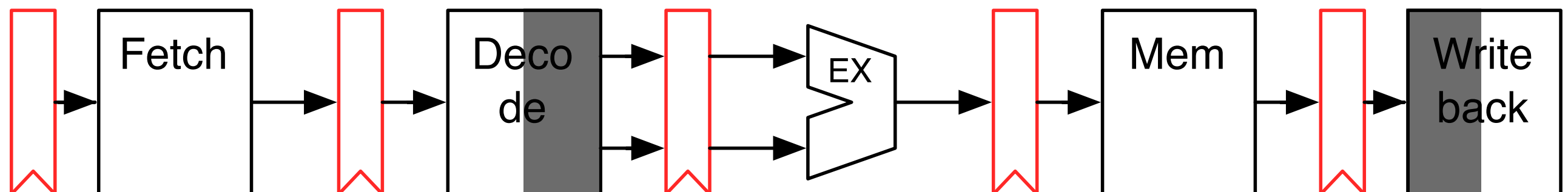


Fixing the Ubiquitous Control Hazard

- We need to know if an instruction is a branch in the fetch stage!
- How can we accomplish this?

Solution 1: Partially decode the instruction in fetch. You just need to know if it's a branch, a jump, or something else.

Solution 2: We'll discuss later.

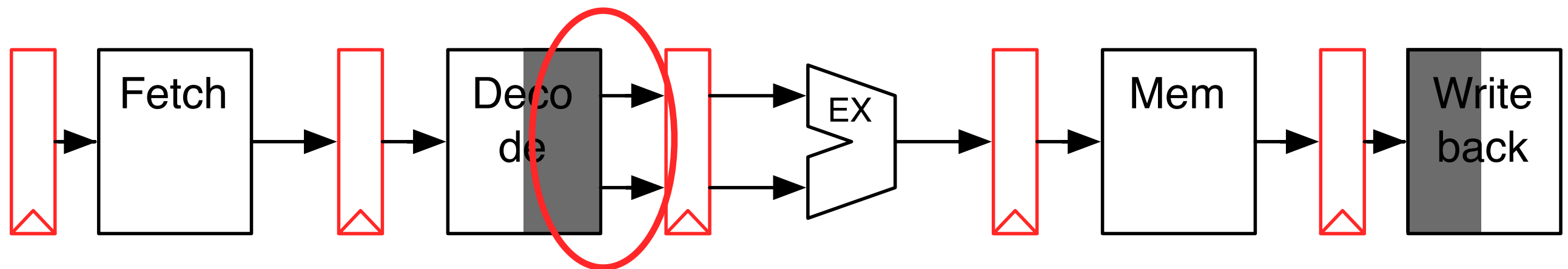


Fixing the Ubiquitous Control Hazard

- We need to know if an instruction is a branch in the fetch stage!
- How can we accomplish this?

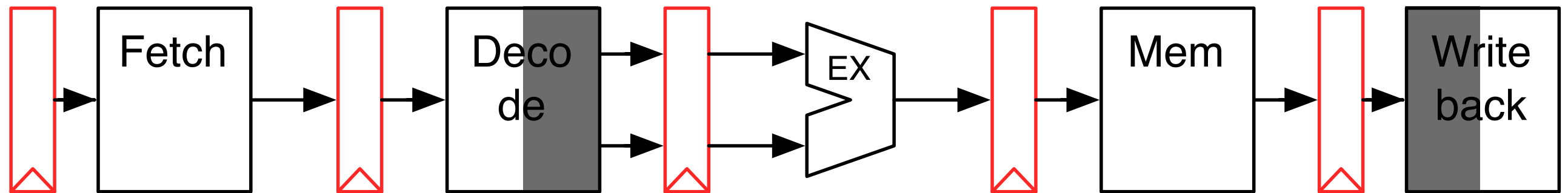
Solution 1: Partially decode the instruction in fetch. You just need to know if it's a branch, a jump, or something else.

Solution 2: We'll discuss later.



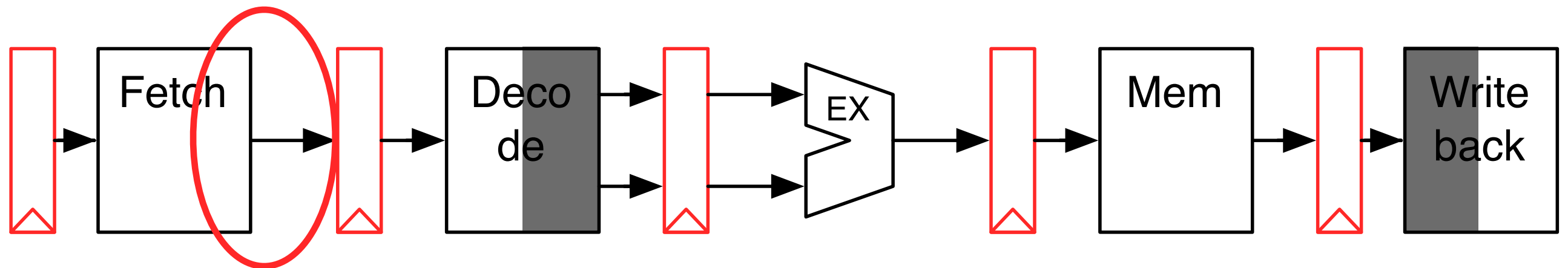
Computing the PC Normally

- Pre-decode in the fetch unit.
 - $PC = PC + 4$
- The PC is ready for the next fetch cycle.



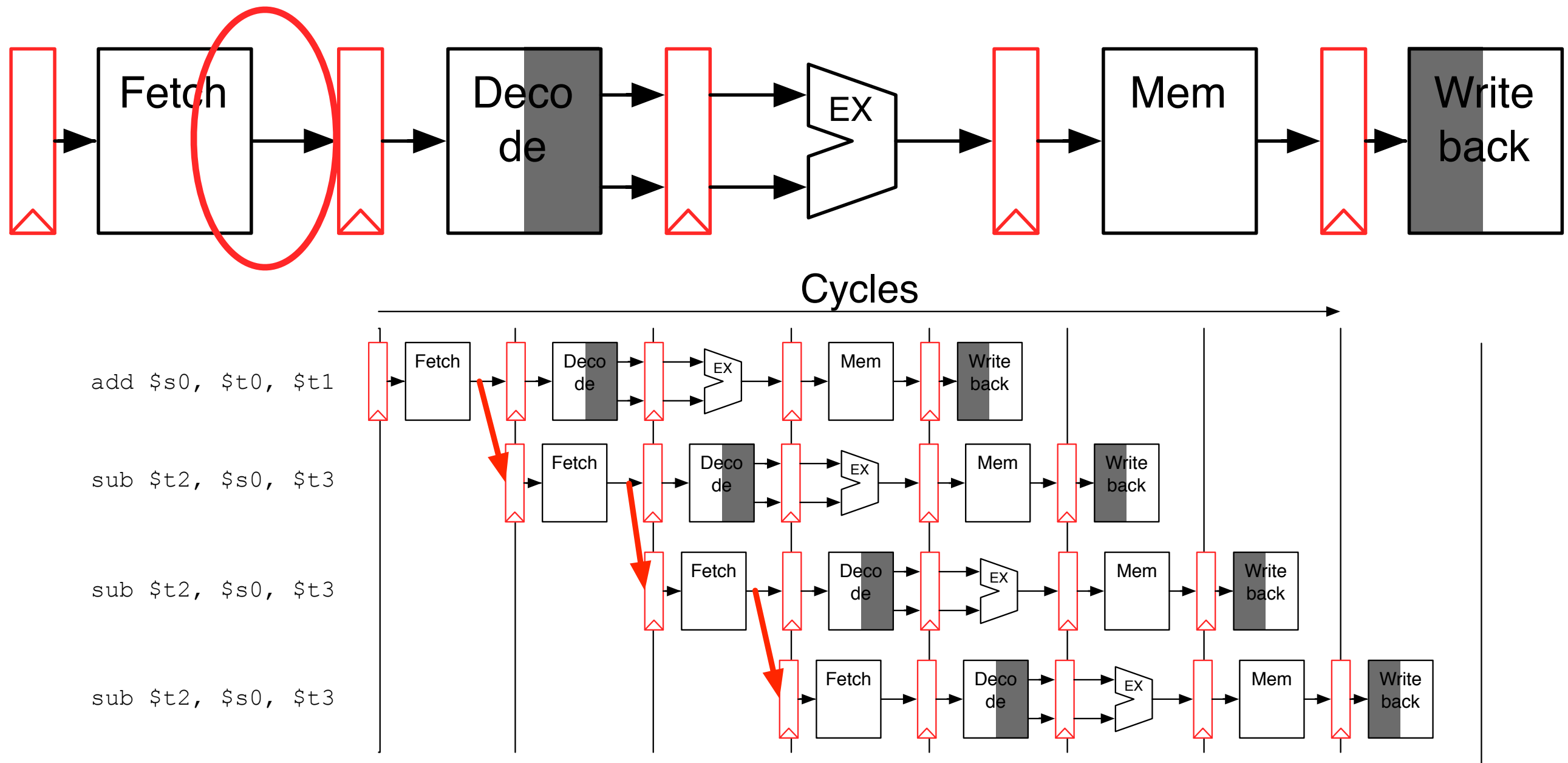
Computing the PC Normally

- Pre-decode in the fetch unit.
 - $PC = PC + 4$
- The PC is ready for the next fetch cycle.



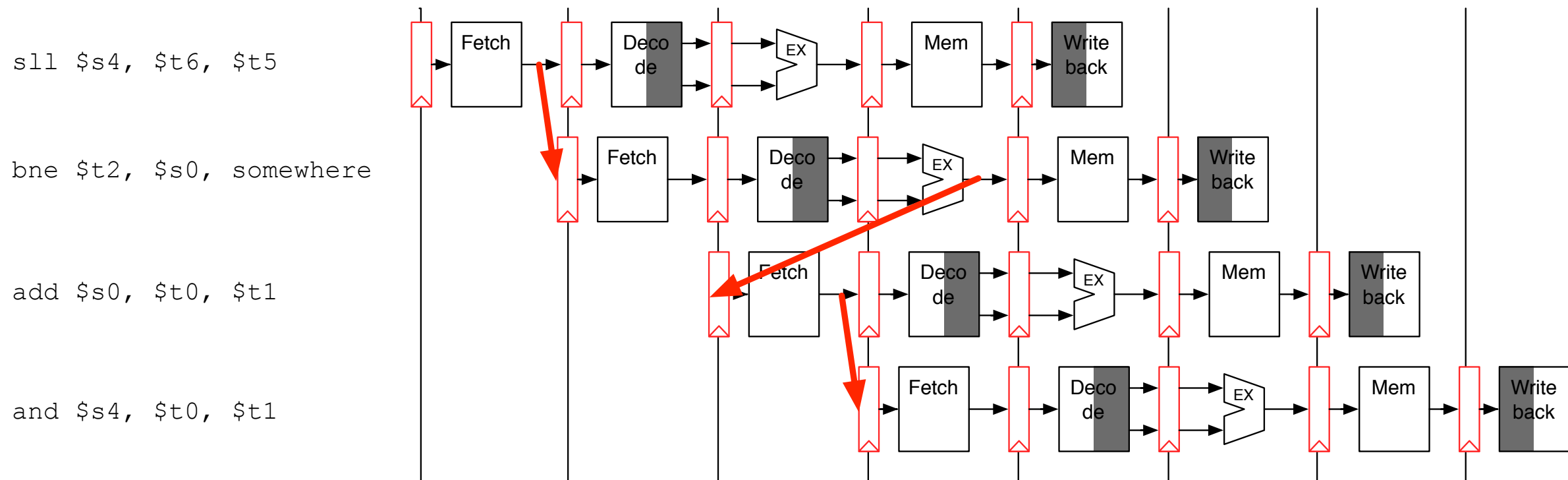
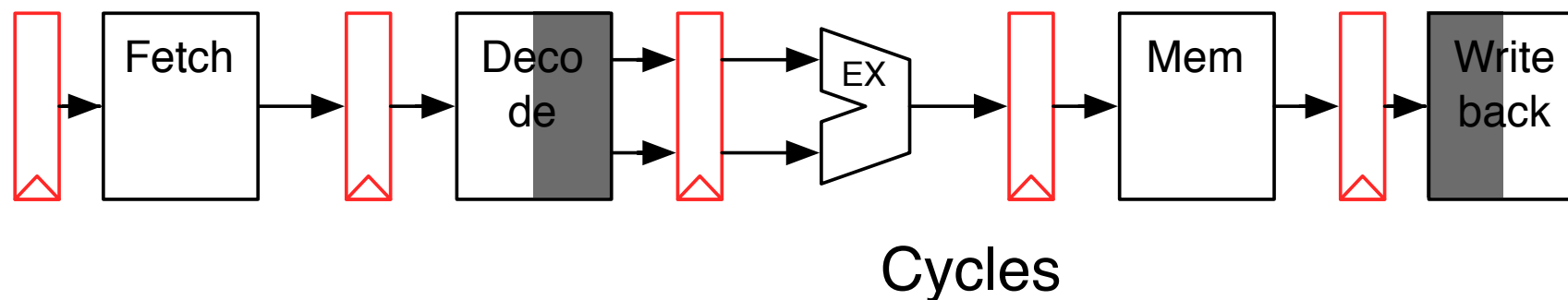
Computing the PC Normally

- Pre-decode in the fetch unit.
 - $PC = PC + 4$
- The PC is ready for the next fetch cycle.



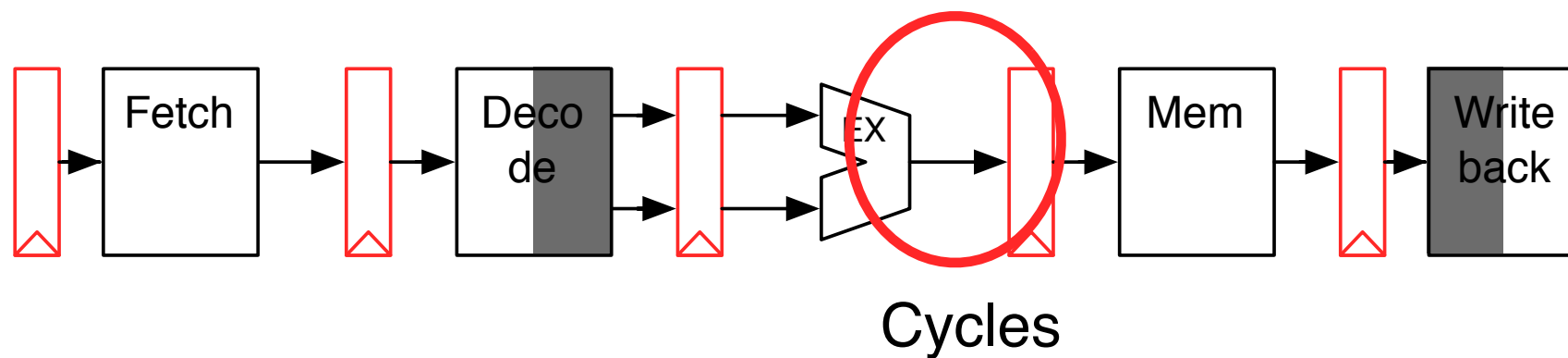
Computing the PC for Branches

- Branch instructions
 - `bne $s1, $s2, offset`
 - `if ($s1 != $s2) { PC = PC + offset } else { PC = PC + 4; }`
- When is the value ready?



Computing the PC for Branches

- Branch instructions
 - `bne $s1, $s2, offset`
 - `if ($s1 != $s2) { PC = PC + offset } else { PC = PC + 4; }`
- When is the value ready?

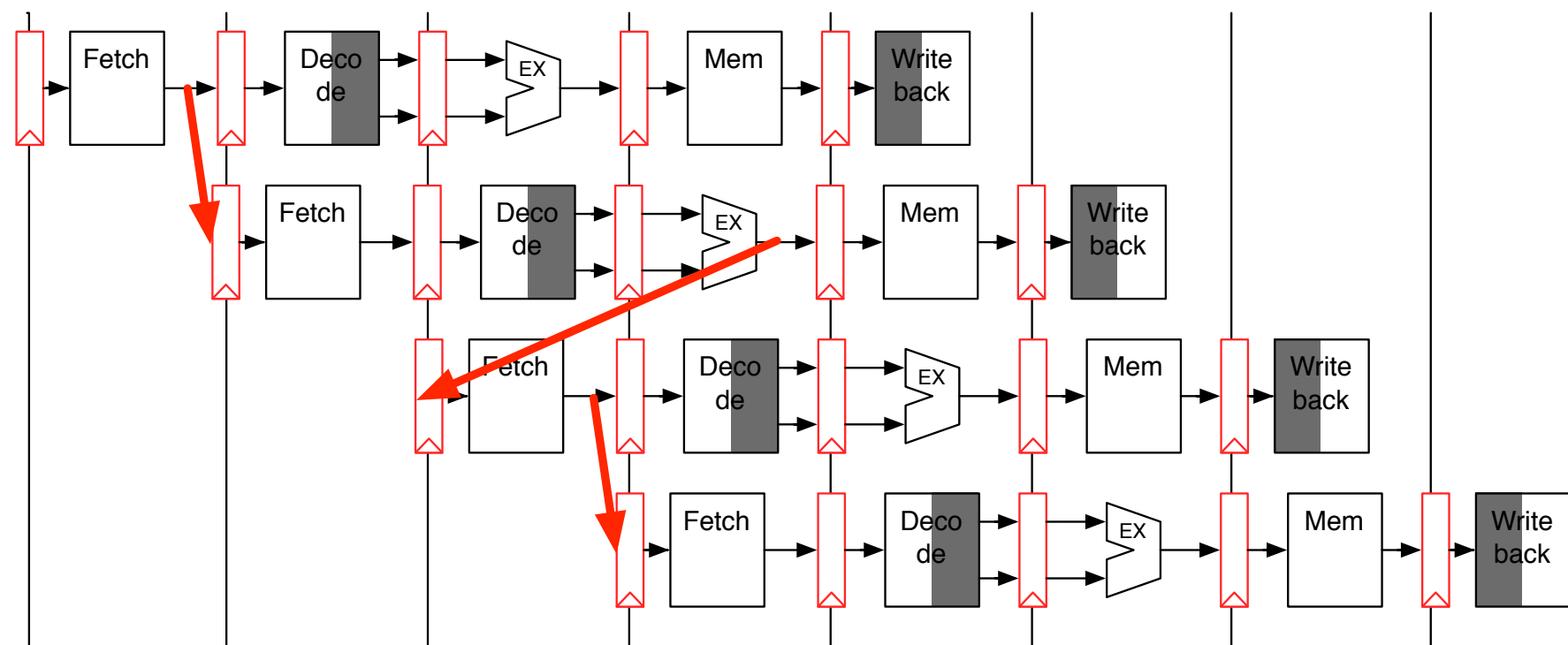


`sll $s4, $t6, $t5`

`bne $t2, $s0, somewhere`

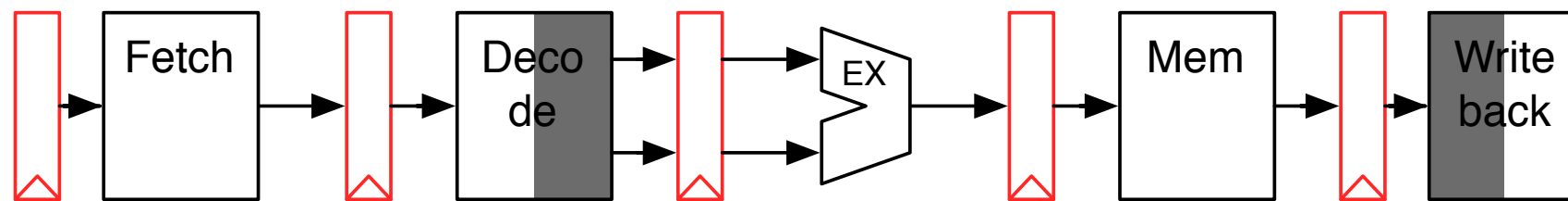
`add $s0, $t0, $t1`

`and $s4, $t0, $t1`



Computing the PC for Jumps

- Jump instructions
 - jr \$s1 -- jump register
 - PC = \$s1
- When is the value ready?

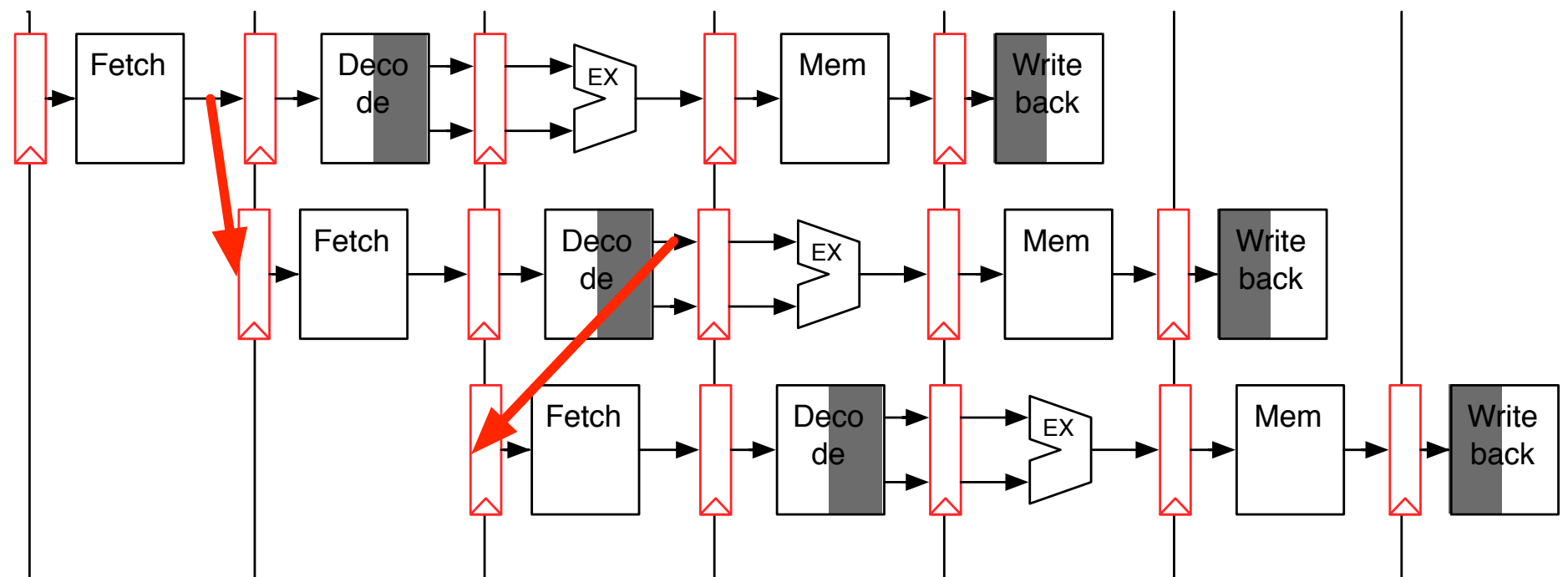


Cycles

sll \$s4, \$t6, \$t5

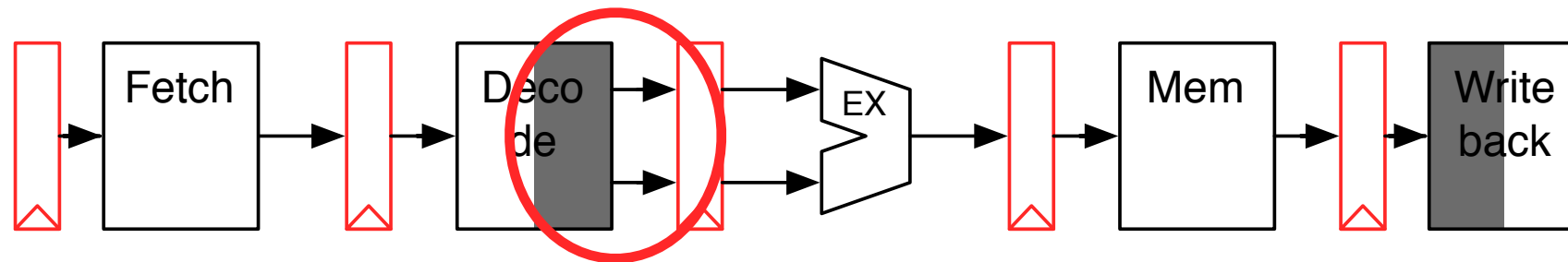
jr \$s4

add \$s0, \$t0, \$t1



Computing the PC for Jumps

- Jump instructions
 - jr \$s1 -- jump register
 - PC = \$s1
- When is the value ready?

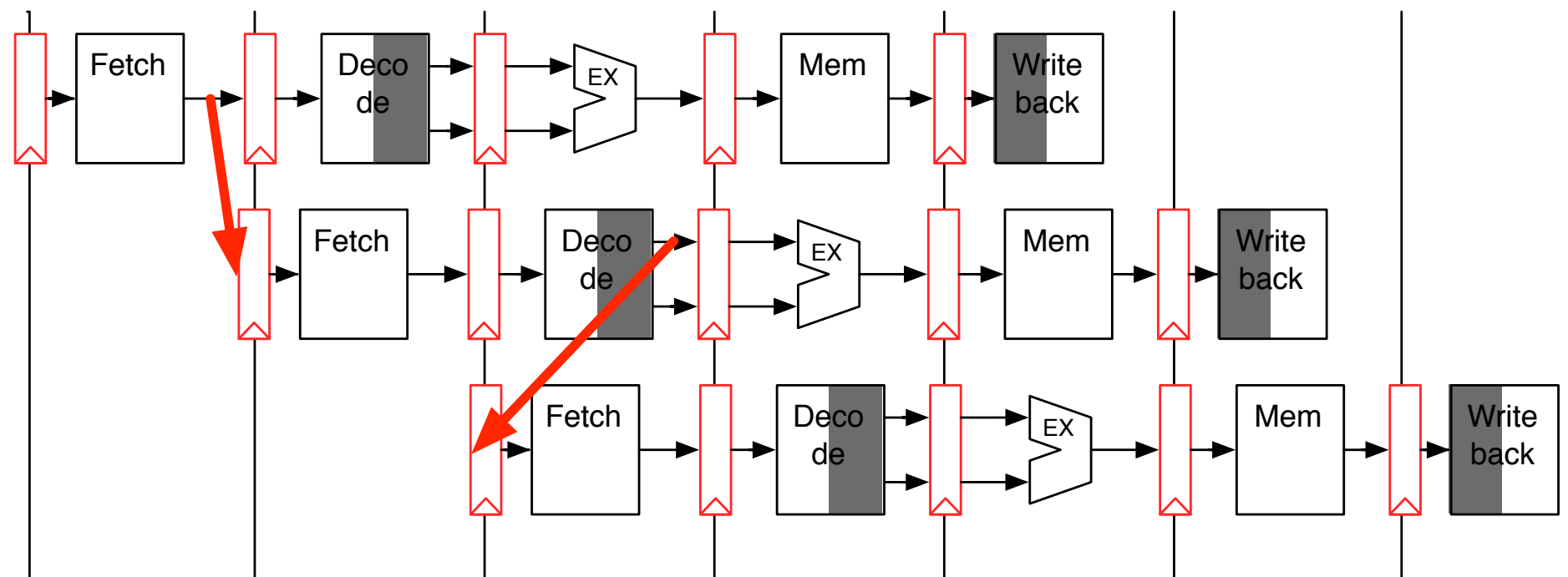


Cycles

sll \$s4, \$t6, \$t5

jr \$s4

add \$s0, \$t0, \$t1



Simple “static” Prediction

- “static” means before run time
- Many prediction schemes are possible
- Predict taken
 - Pros?
- Predict not-taken
 - Pros?
- Backward taken/Forward not taken
 - The best of both worlds!
 - Most loops have have a backward branch at the bottom, those will predict taken
 - Others (non-loop) branches will be not-taken.

Simple “static” Prediction

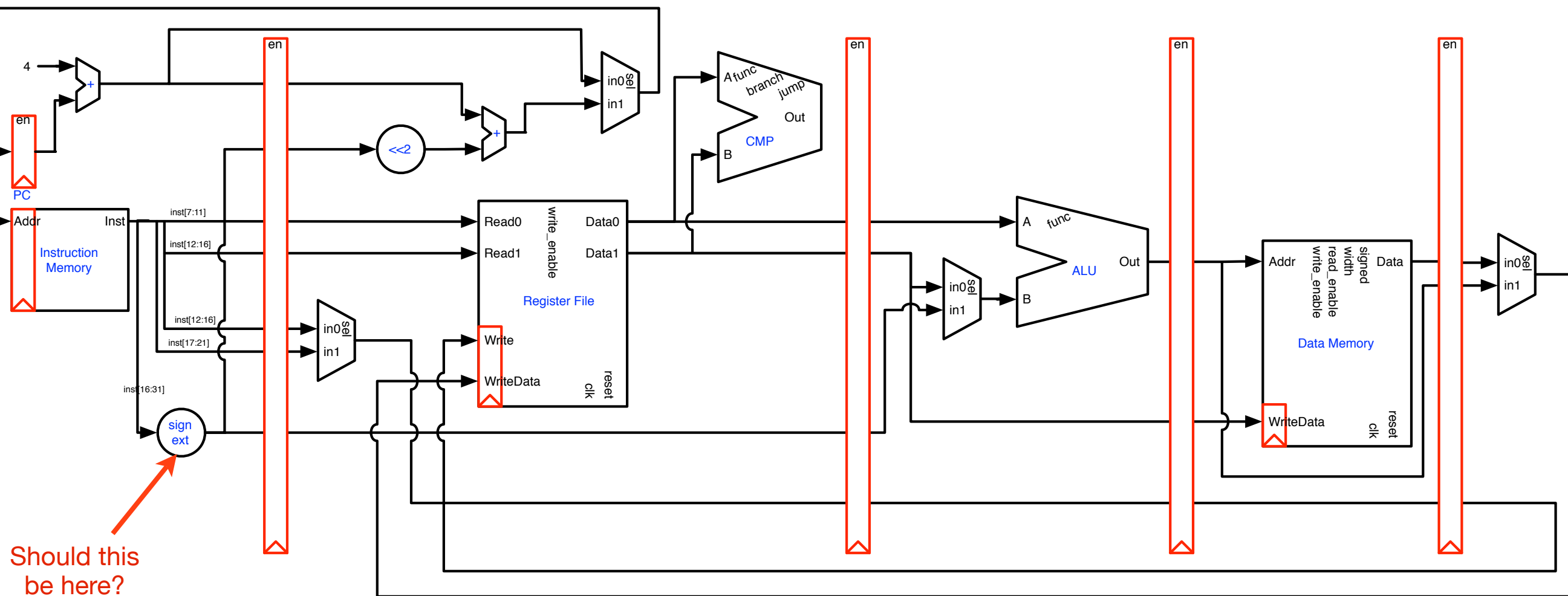
- “static” means before run time
- Many prediction schemes are possible
- Predict taken
 - Pros? Loops are commons
- Predict not-taken
 - Pros?
- Backward taken/Forward not taken
 - The best of both worlds!
 - Most loops have have a backward branch at the bottom, those will predict taken
 - Others (non-loop) branches will be not-taken.

Simple “static” Prediction

- “static” means before run time
- Many prediction schemes are possible
- Predict taken
 - Pros? Loops are commons
- Predict not-taken
 - Pros? Not all branches are for loops.
- Backward taken/Forward not taken
 - The best of both worlds!
 - Most loops have have a backward branch at the bottom, those will predict taken
 - Others (non-loop) branches will be not-taken.

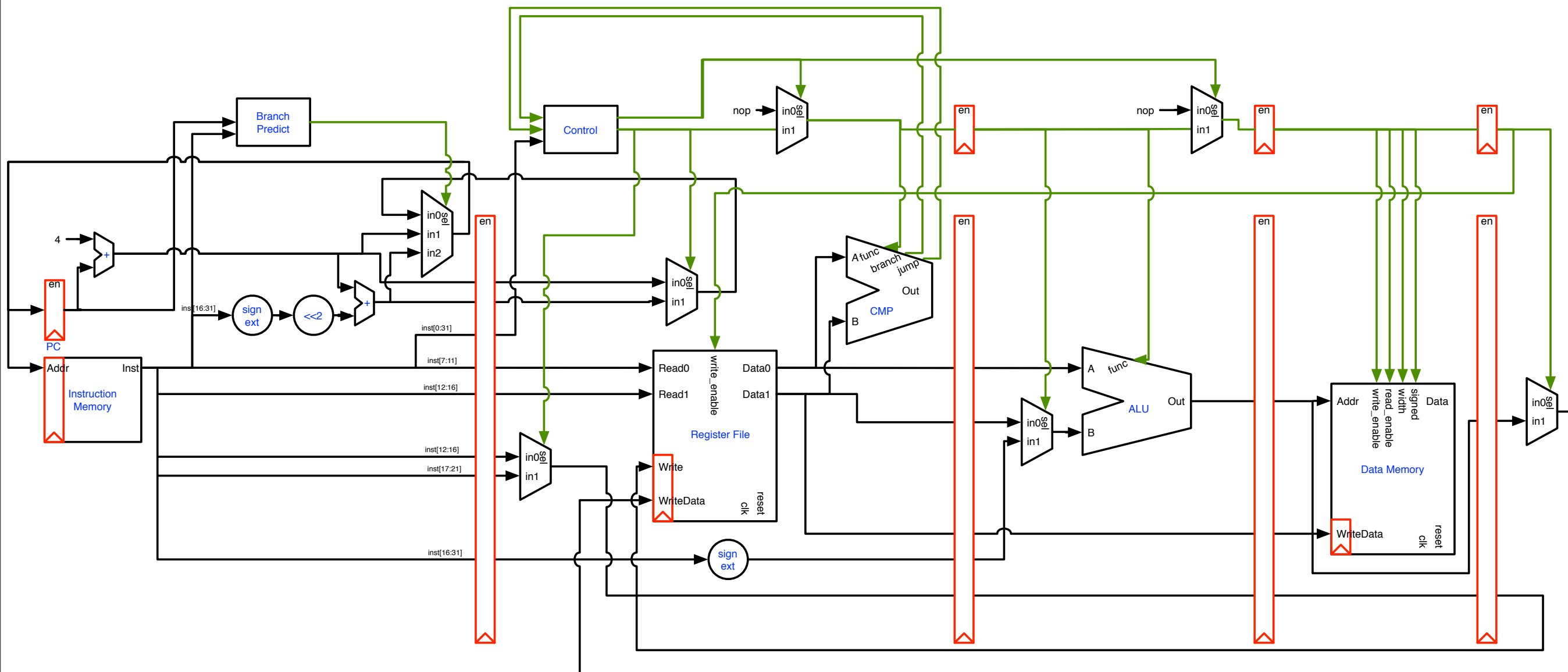
Basic Pipeline Recap

- The PC is required in Fetch
- For branches, it's not know till *decode*.



Branches only, one delay slot, simplified ISA, no control

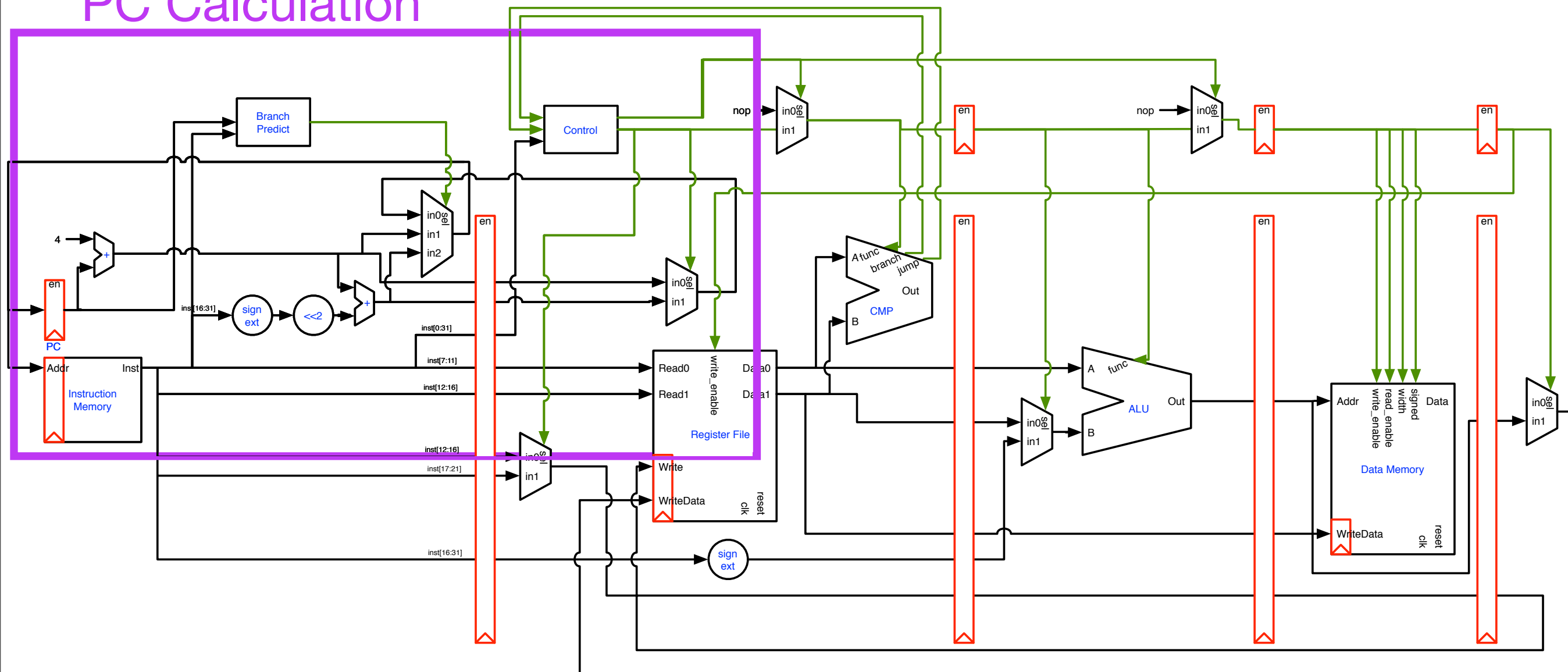
Supporting Speculation



- Predict: Compute all possible next PCs in fetch. Choose one.
 - The correct next PC is known in decode
- Flush as needed: Replace “wrong path” instructions with no-ops.

Supporting Speculation

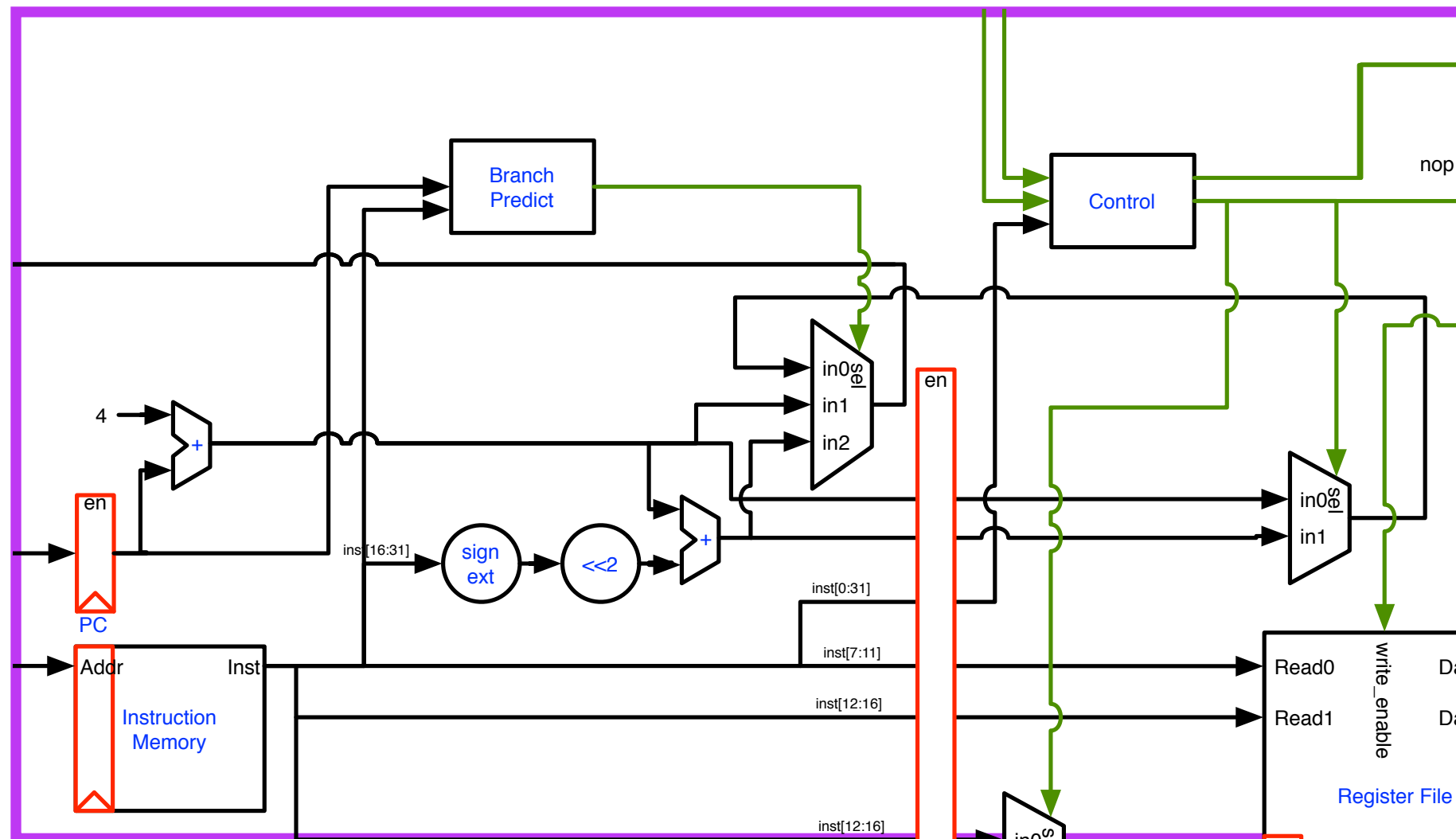
PC Calculation



- Predict: Compute all possible next PCs in fetch. Choose one.
 - The correct next PC is known in decode
- Flush as needed: Replace “wrong path” instructions with no-ops.

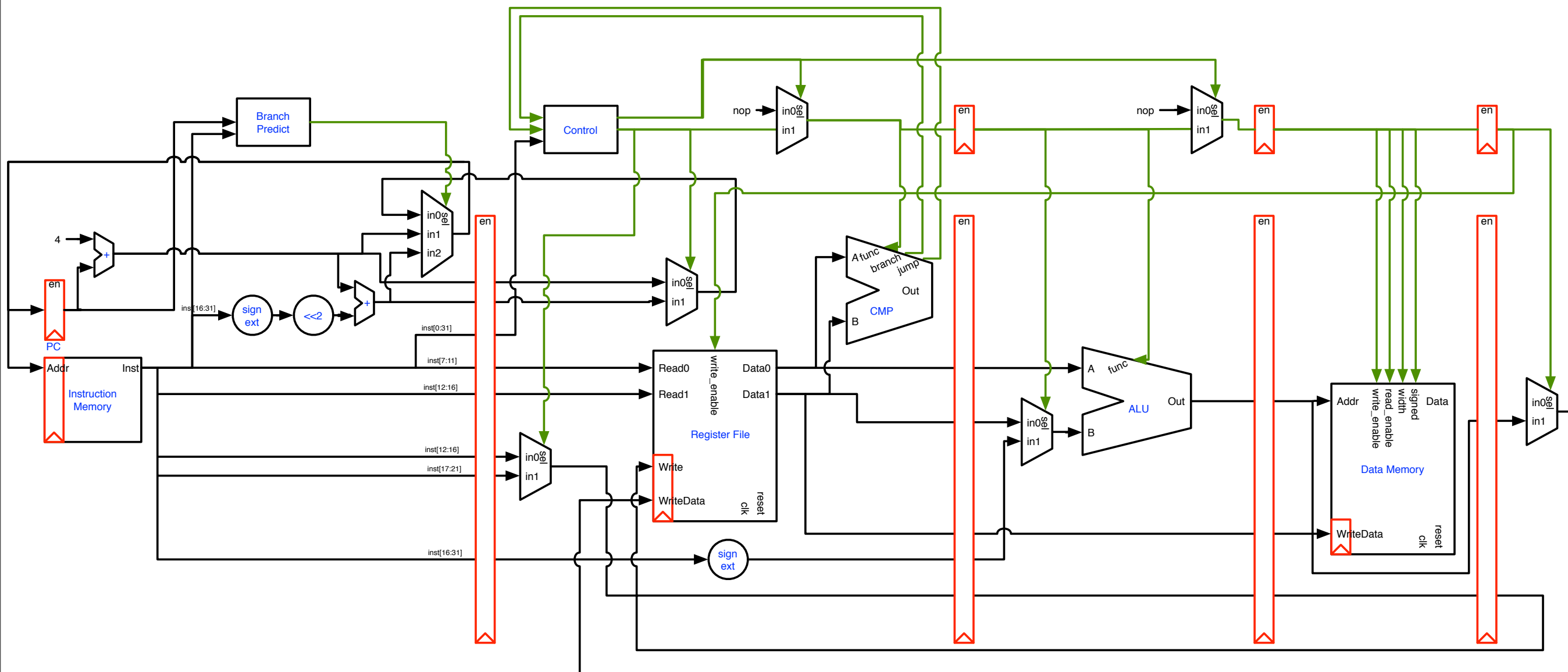
Supporting Speculation

PC Calculation



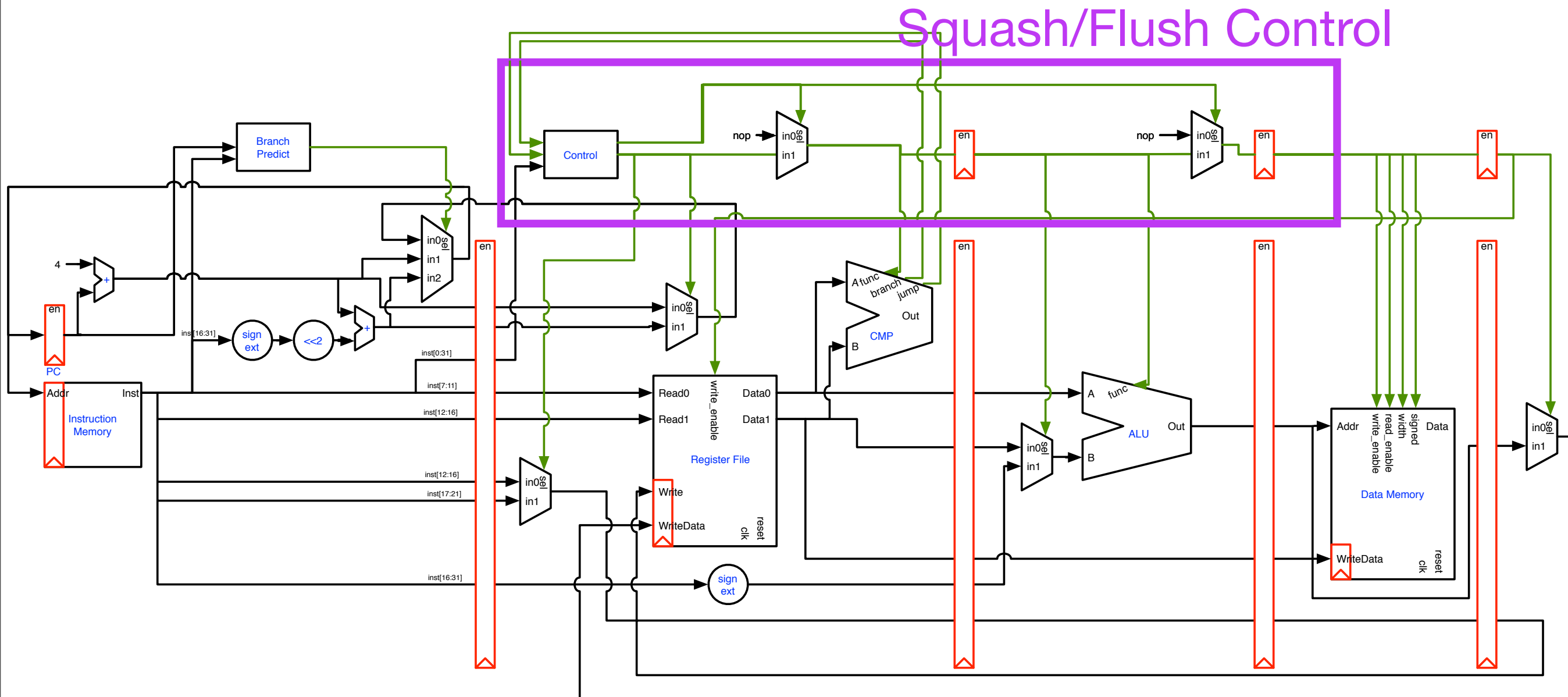
- Predict: Compute all possible next PCs in fetch. Choose one.
 - The correct next PC is known in decode
- Flush as needed: Replace “wrong path” instructions with no-ops.

Supporting Speculation



- Predict: Compute all possible next PCs in fetch. Choose one.
 - The correct next PC is known in decode
- Flush as needed: Replace “wrong path” instructions with no-ops.

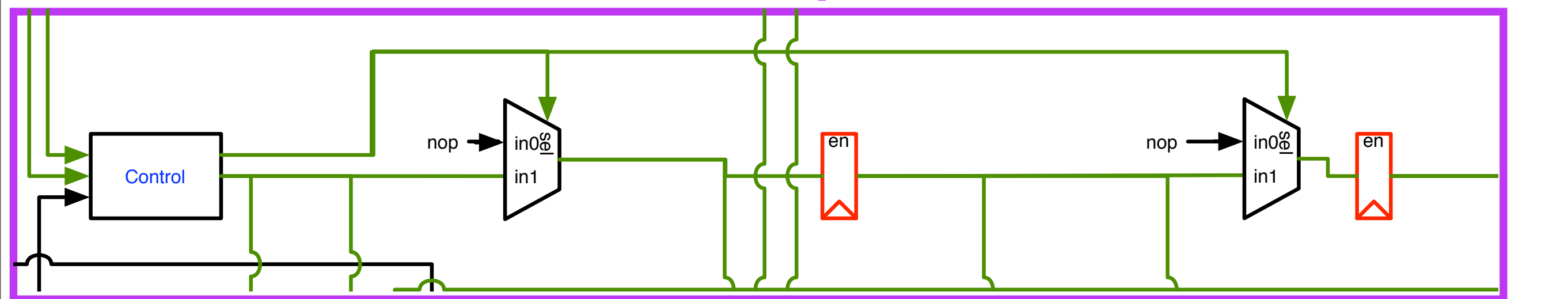
Supporting Speculation



- Predict: Compute all possible next PCs in fetch. Choose one.
 - The correct next PC is known in decode
- Flush as needed: Replace “wrong path” instructions with no-ops.

Supporting Speculation

Squash/Flush Control



- Predict: Compute all possible next PCs in fetch. Choose one.
 - The correct next PC is known in decode
- Flush as needed: Replace “wrong path” instructions with no-ops.