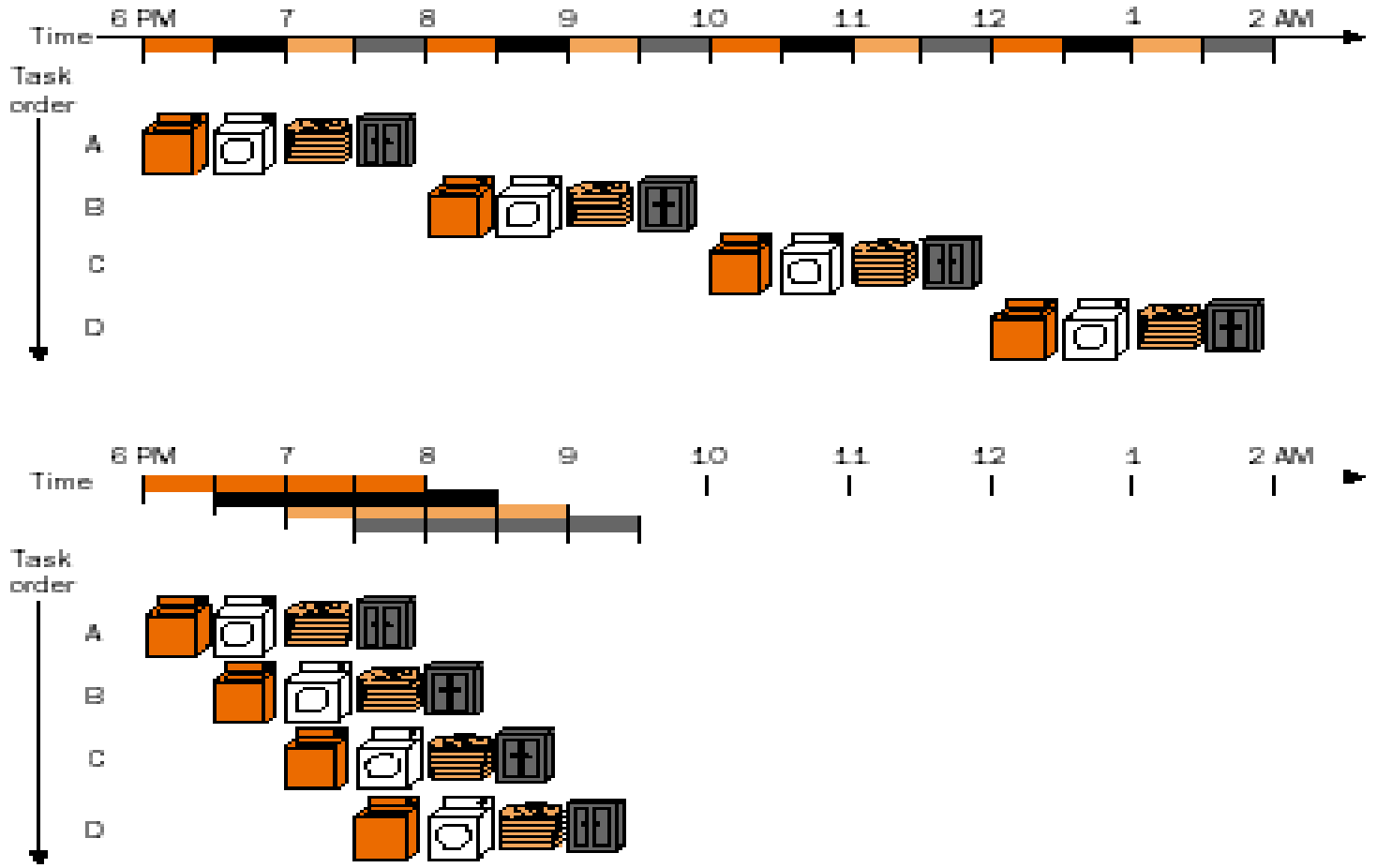
The Pipelined MIPS Processor

- **We complete our study of ALU architecture by investigating an approach providing even higher performance for the MIPS CPU.**
- **We first saw how the MIPS CPU performance could be improved by converting the so-called single-cycle CPU to a multi-cycle design.**
  - **In the multi-cycle approach, instead of using a single clock cycle for the whole instruction, the clock is accelerated, and instructions execute in phases over several clock cycles.**
  - **Each instruction phase takes one clock cycle.**
  - **This means that as each instruction executes, only one section of the CPU will be active per clock cycle -- the one executing that phase of the instruction.**
- **This suggests that perhaps we might redesign the CPU slightly so that every CPU section can operate independently on an instruction at the same time.**

The "Laundry Example"

- As an introduction to the concept of pipelining, Patterson and Hennessy use the example of doing one's laundry.
- Most people have – or have access to – a washer and dryer.
- Assume that you need to wash several washer loads of clothing.
- Would anyone divide the clothing into washer loads and then wash, dry, fold and put away the first load before starting the second?
- No, if you were washing clothes, you would finish washing the first load, put it in the dryer, and start the second load washing.
- If there were more loads to wash, you would begin to fold and put away finished clothing while the later loads were washing and drying.
- We can see this schematically on the next slide.

Graphical Example of the Laundry Cycle



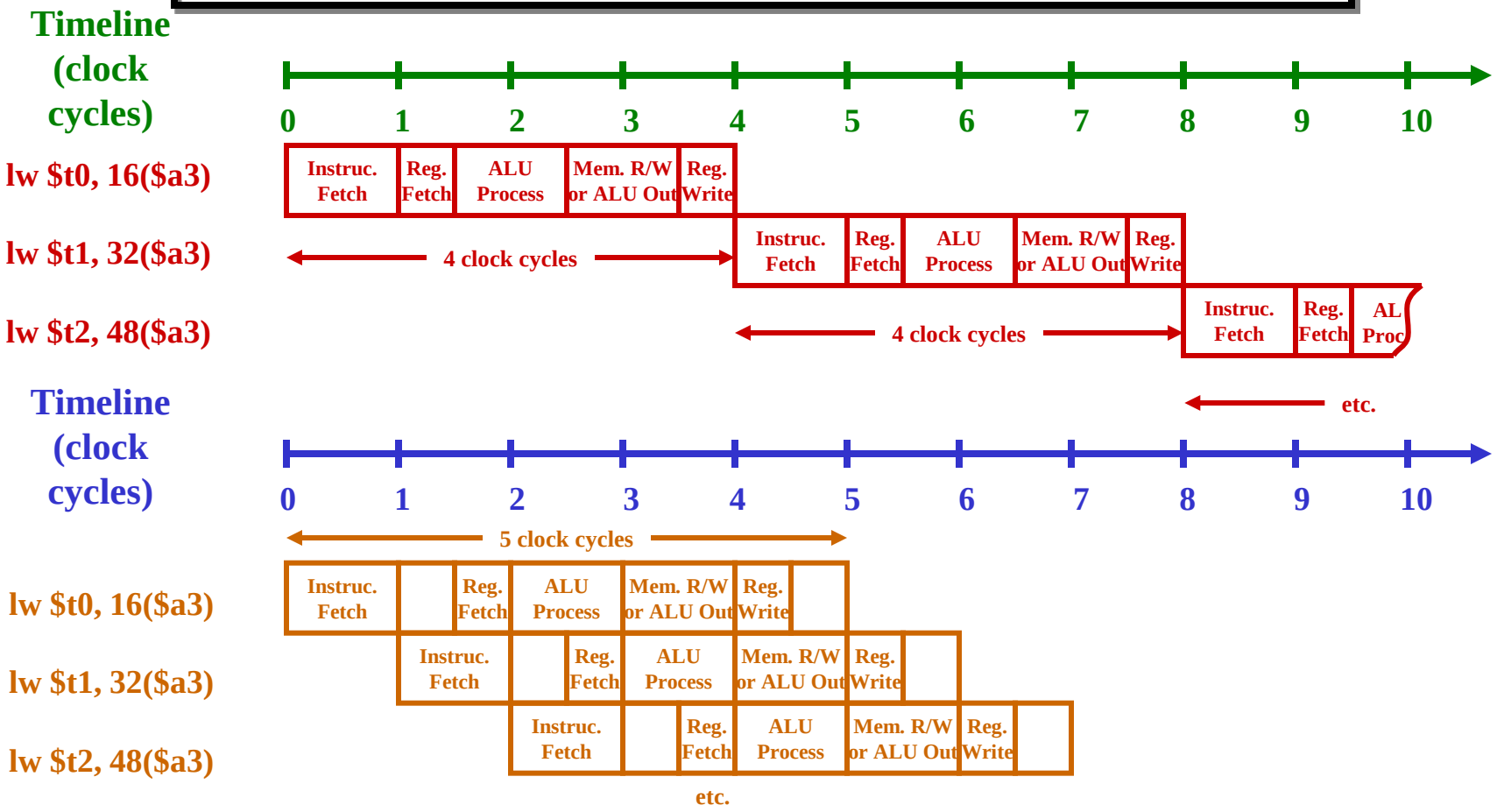
The "Pipeline" Processor

- **Patterson and Hennessy applied this "simultaneous wash-dry-fold-put away concept" to the single-cycle computer model.**
- **The idea was to "wash, dry, fold, and put away" instructions simultaneously so that the instruction throughput – the number of clock cycles per instructions – could be dramatically decreased.**
- **In the case of the single cycle model, one instruction is done per clock cycle, but the clock must be as slow as the slowest instruction.**
- **In the multi-cycle implementation, the clock runs faster, instructions takes 3-5 cycles, but only one instruction is processed at a time.**
- **What if, each time the clock ticked, we could process an instruction in each section of the multicycle processor? Then we could process several instructions simultaneously, approaching the goal of completing an instruction every clock cycle.**

Pipeline Architecture

- **A pipelined computer executes instructions concurrently.**
- **Hardware units are organized into stages:**
  - **Execution in each stage takes exactly 1 clock period.**
  - **Stages are separated by pipeline registers that preserve and pass partial results to the next stage.**
- **As noted earlier, performance = complexity + cost. The pipeline approach brings additional expense plus its own set of problems and complications, called hazards, which we will also study.**

Sequential Versus Pipelined Execution

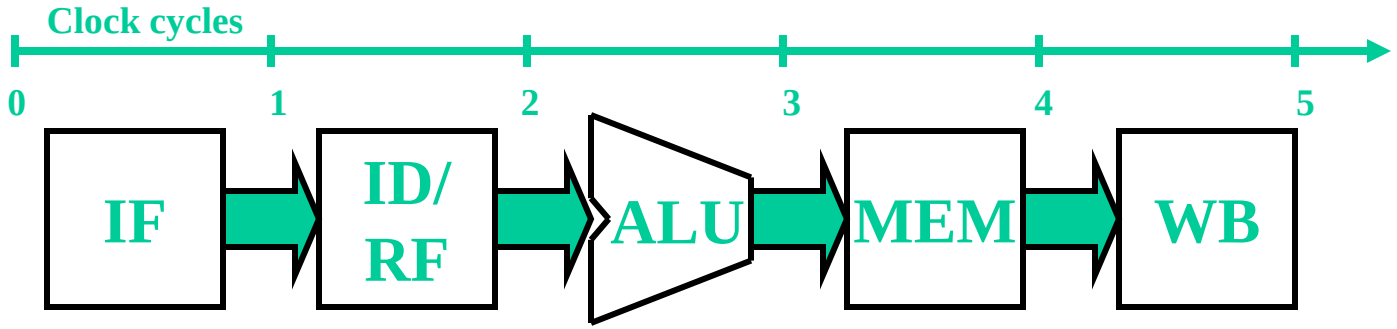


Speed Advantage of the Pipeline

- The multicycle, serial processor that we studied last lecture can execute n instructions in ns clock periods, or $ET_S = ns$, where ET is the execution time and s is the number of stages.
- A pipelined processor with s stages can execute n instructions in $ET_P = s + (n - 1)$ clock periods.
- The ideal pipeline speedup depends on the number of stages, and can be greater for more stages (hence Intel's choice of a 20-stage pipeline for the current P-IV).
- Thus the speed advantage of pipeline over multicycle can be defined as:

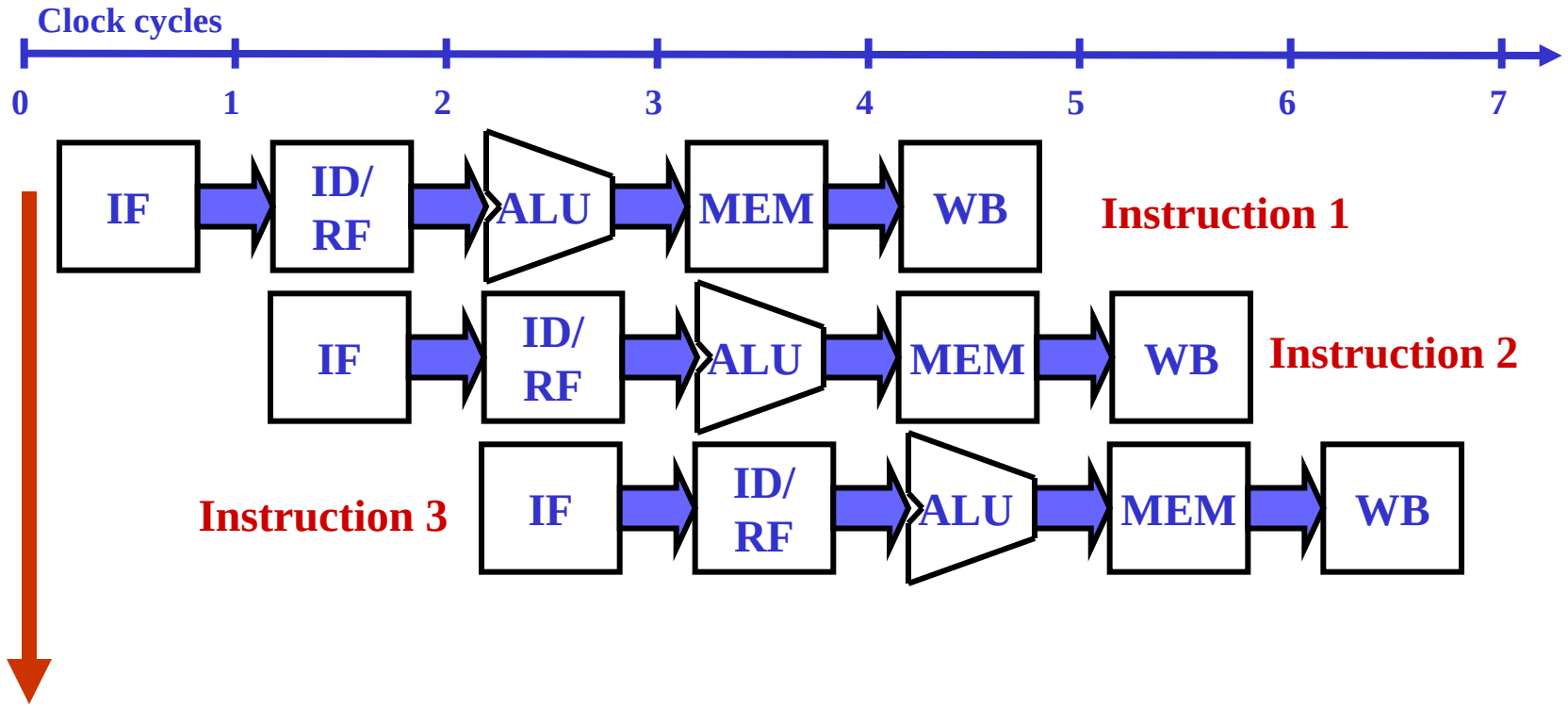
$$S_P = \frac{ET_s}{ET_P} = \frac{ns}{s+(n-1)} \xrightarrow{n>>s} s$$

Pipeline Stages

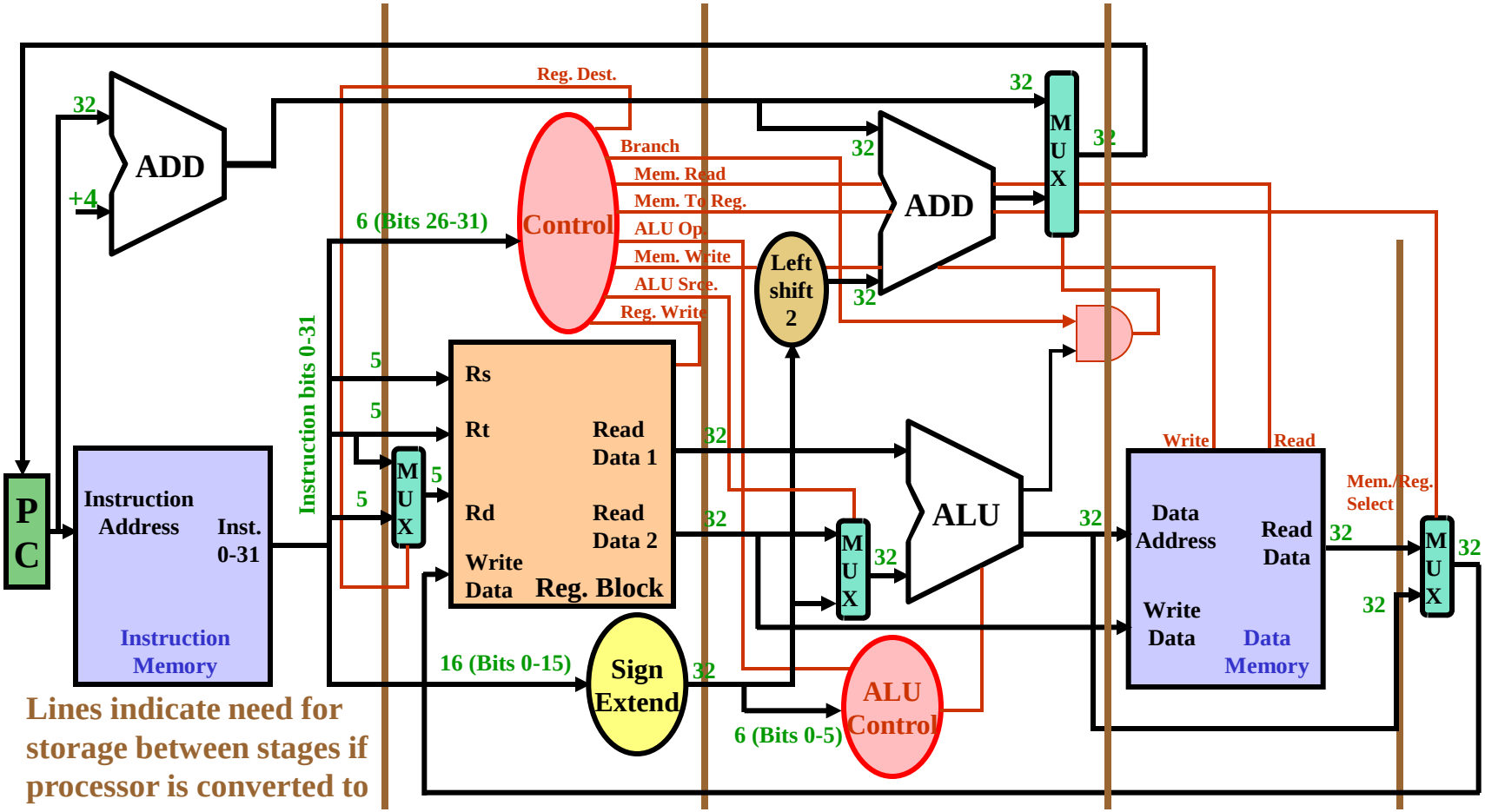


- **The MIPS R2000 pipeline processor is divided into five processing stages:**
  1. **Instruction fetch (IF)**
  2. **Instruction decode (ID) and register fetch (RF)**
  3. **ALU instruction execution (ALU) – ALU processing, branch condition evaluation, memory address computation, etc. This is also referred to as execution (EX)**
  4. **Memory access (MEM)**
  5. **Write back (WB) to register file**

Overlapped Pipeline Execution

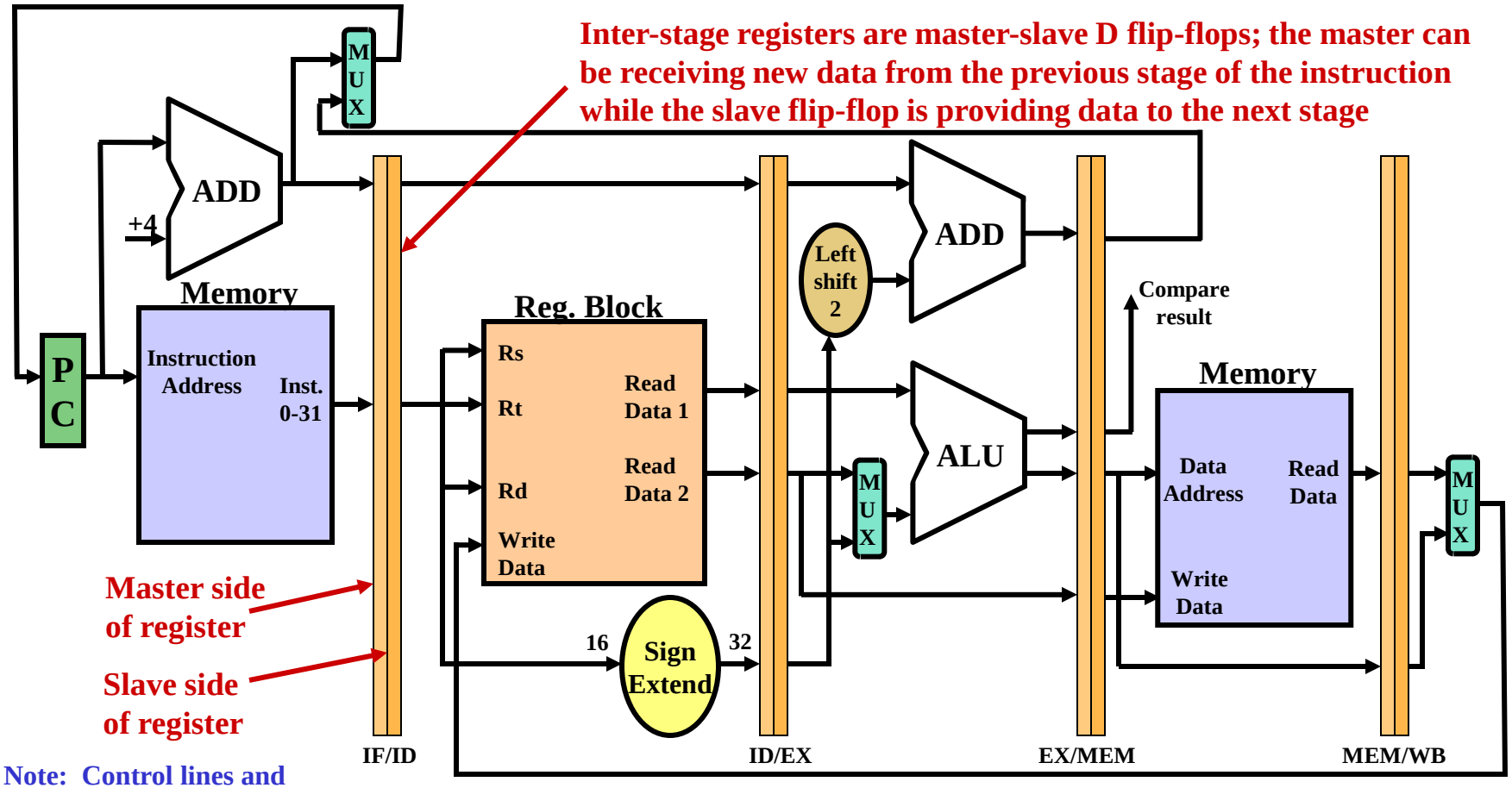


Single-Cycle Datapath



Lines indicate need for storage between stages if processor is converted to pipeline

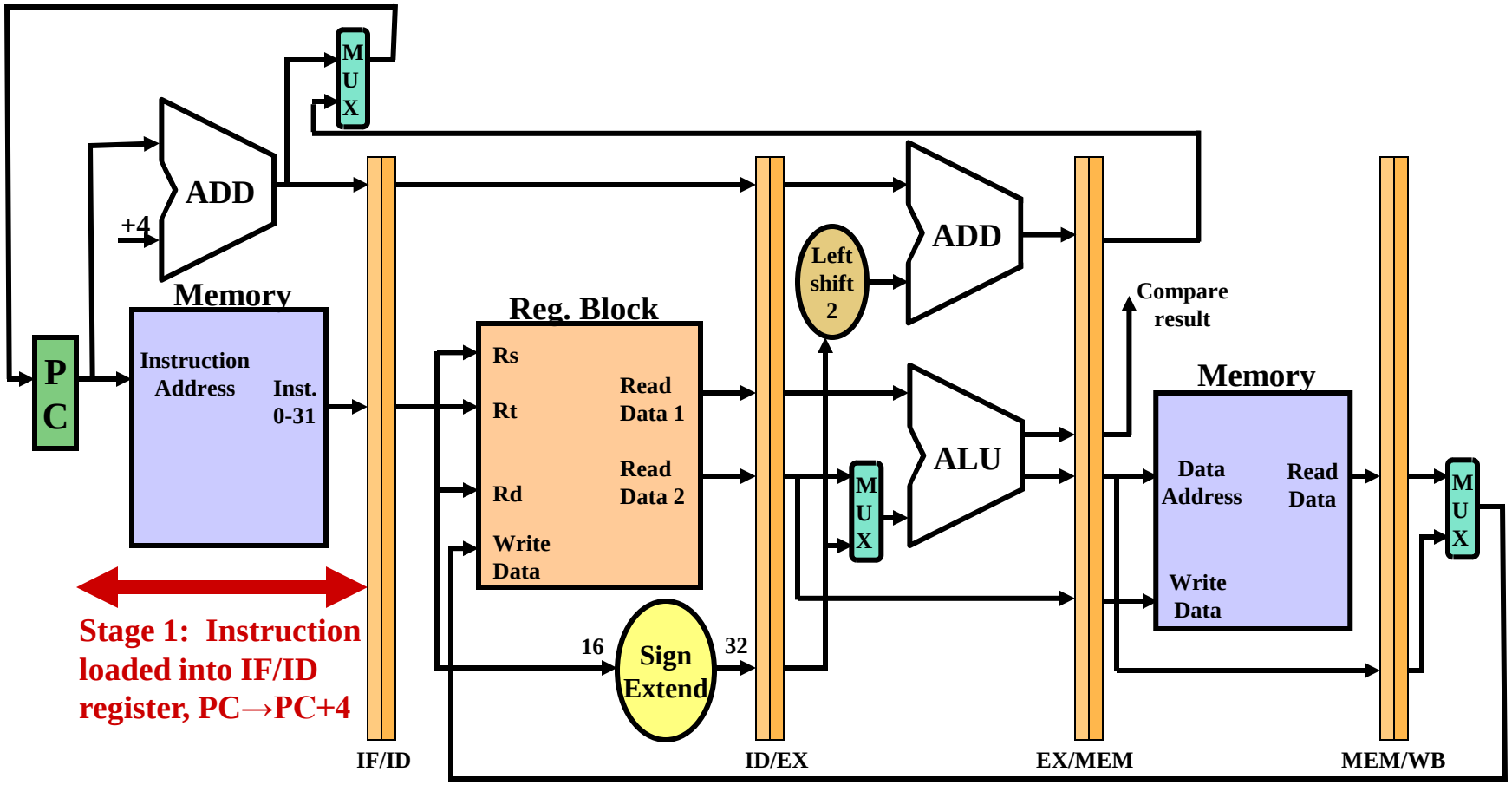
Single-Cycle Datapath with Pipeline Registers



Note: Control lines and logic not shown for clarity

After David A. Patterson and John L. Hennessy, *Computer Organization and Design*, 2nd Edition

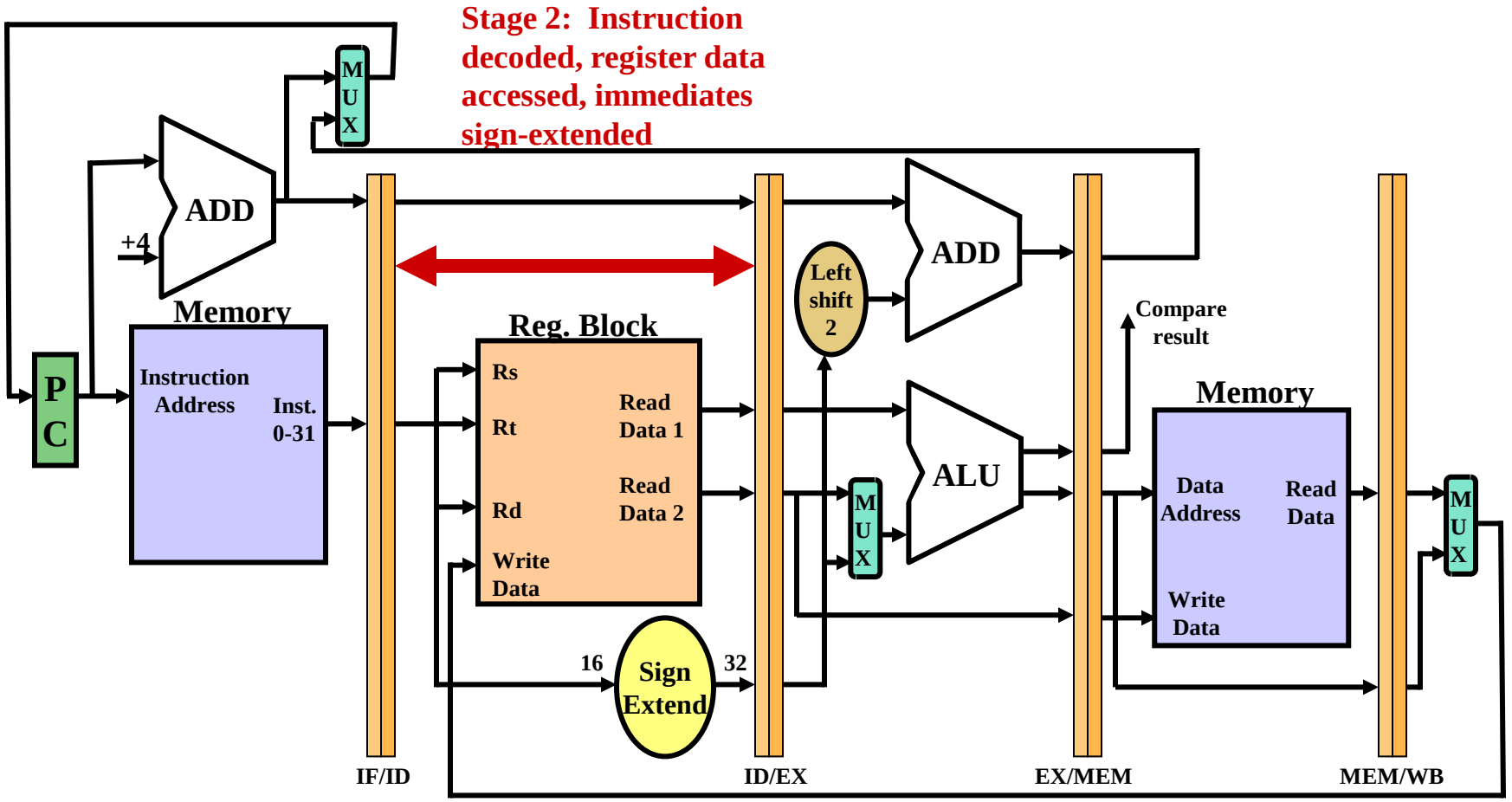
Instruction Process Through Pipeline (1)



Stage 1: Instruction loaded into IF/ID register, PC→PC+4

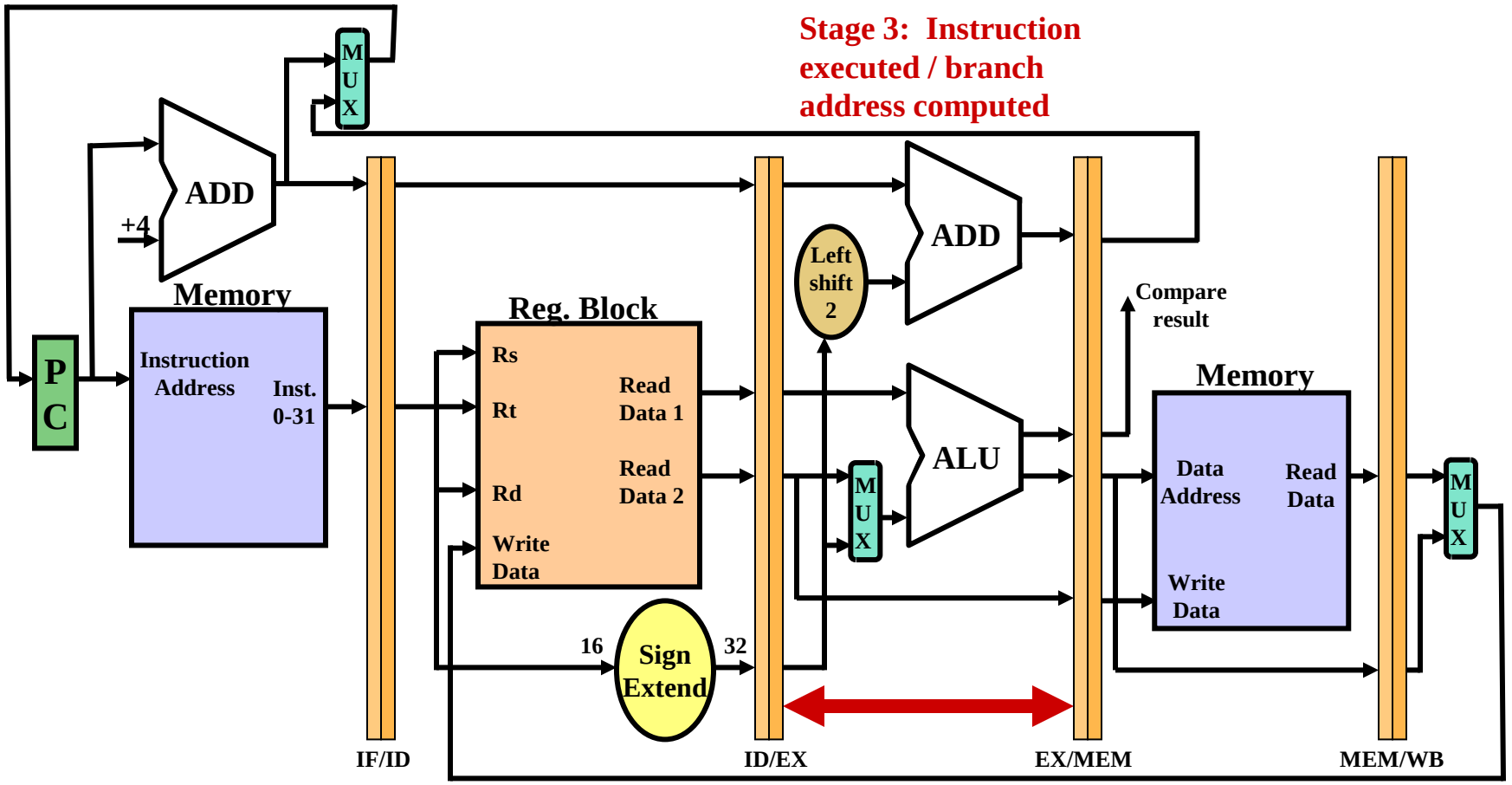
After David A. Patterson and John L. Hennessy, *Computer Organization and Design*, 2nd Edition

Instruction Process Through Pipeline (2)



After David A. Patterson and John L. Hennessy, *Computer Organization and Design*, 2nd Edition

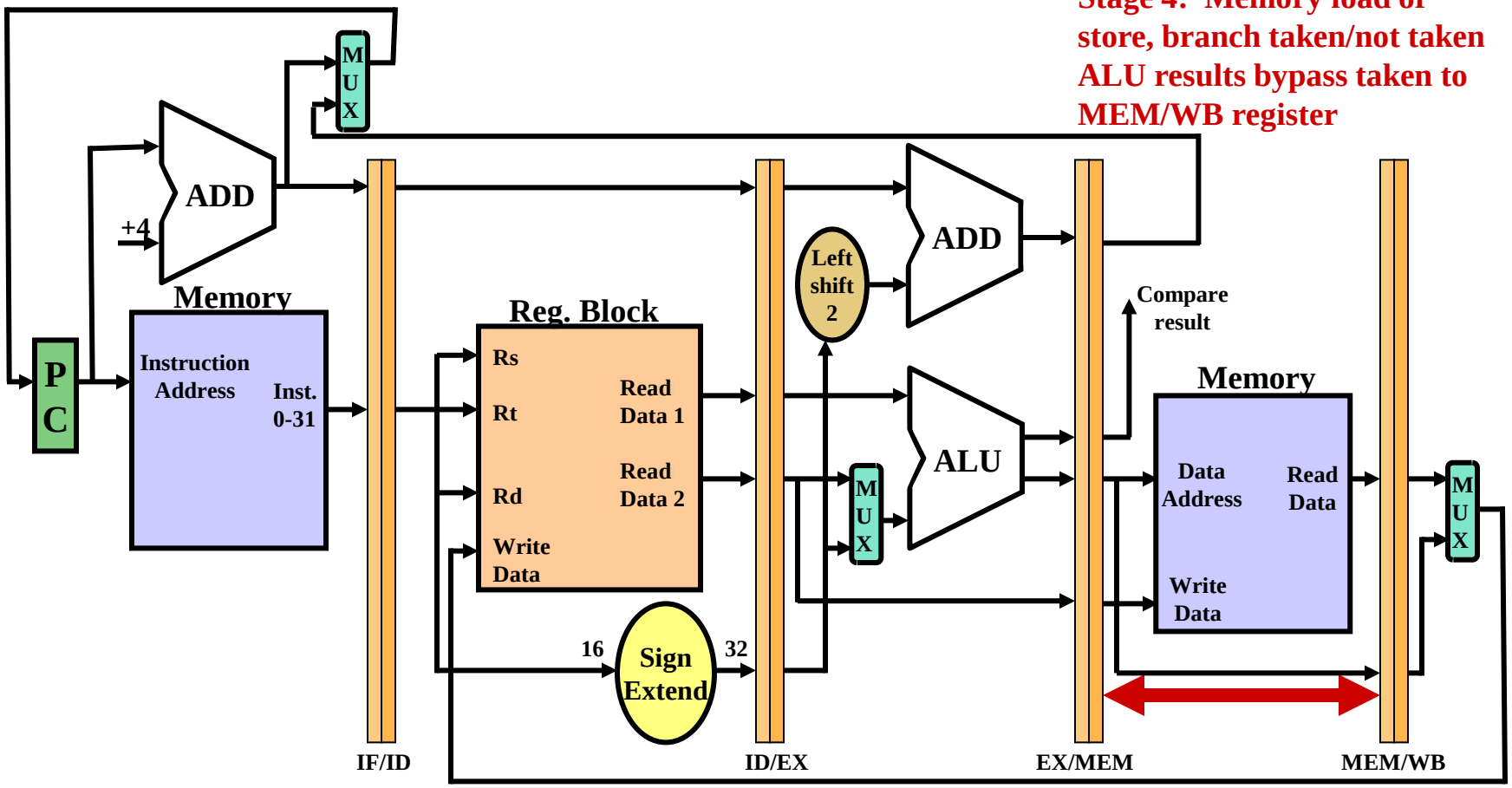
Instruction Process Through Pipeline (3)



After David A. Patterson and John L. Hennessy, *Computer Organization and Design*, 2nd Edition

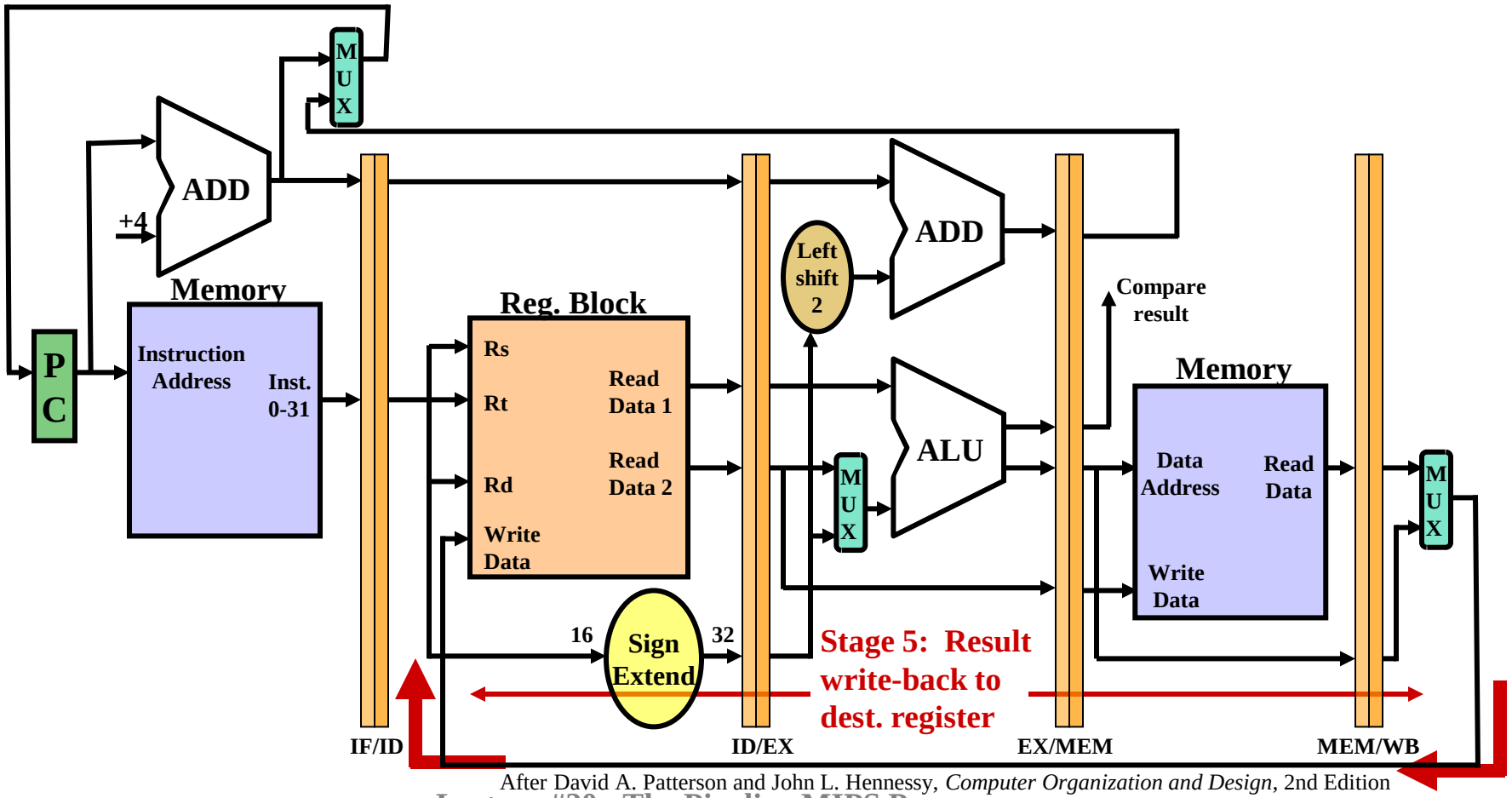
Instruction Process Through Pipeline (4)

Stage 4: Memory load or store, branch taken/not taken ALU results bypass taken to MEM/WB register



After David A. Patterson and John L. Hennessy, *Computer Organization and Design*, 2nd Edition

Instruction Process Through Pipeline (5)

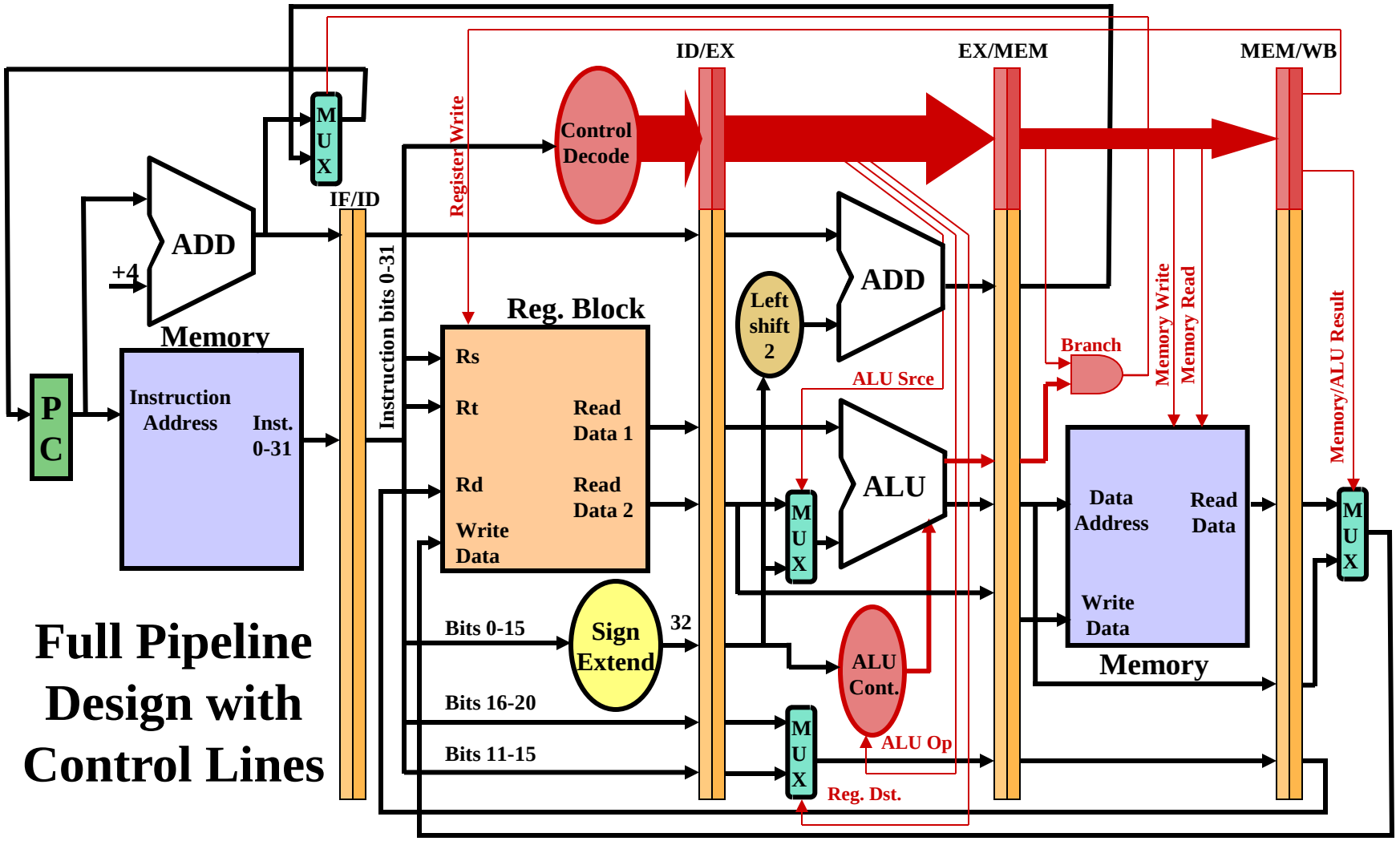


After David A. Patterson and John L. Hennessy, *Computer Organization and Design*, 2nd Edition

Adding Control

- **Control information must be carried along as a part of the instruction, since this information is required at different stages of the pipeline.**
- **This can be done by adding more inter-stage storage register bits to forward control data yet to be used.**
- **The result is very large inter-stage registers. For example, the storage capacity required between the instruction decode and ALU execution stages (ID/EX register) is more than 120 bits.**
- **The resulting processor with full control functionality is shown on the next slide**

Full Pipeline Design with Control Lines

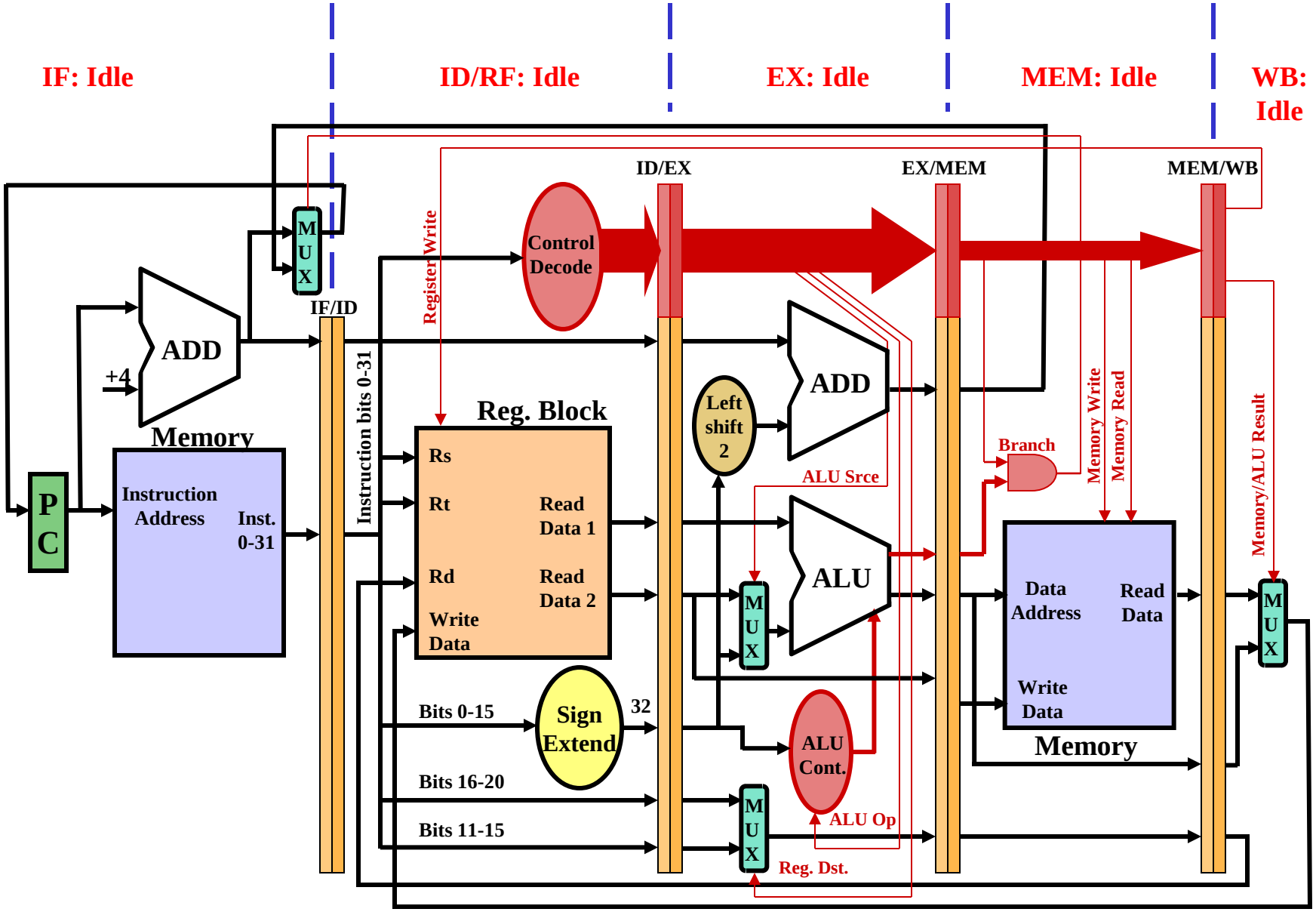


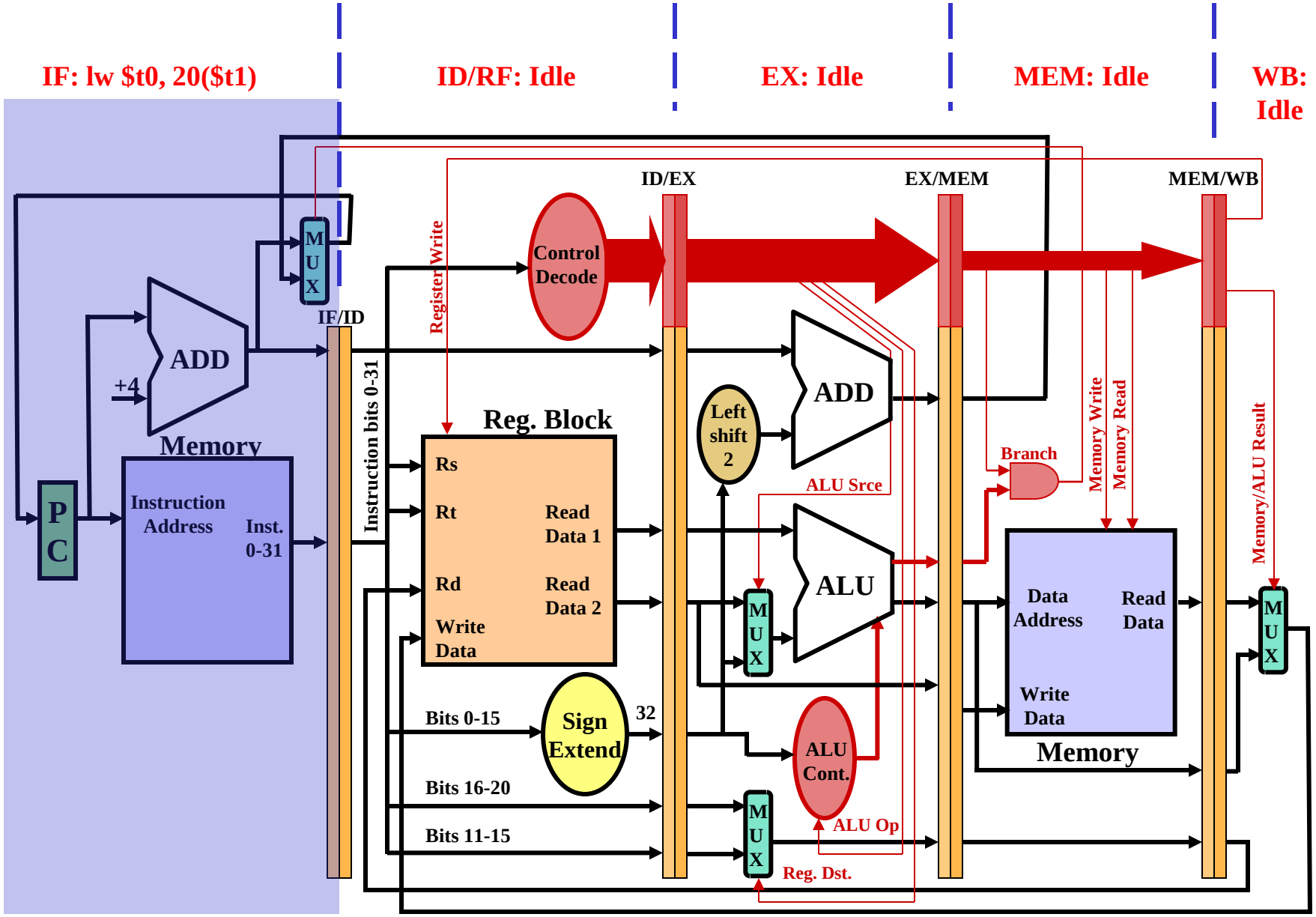
The Pipeline in Action

- **The following instruction sequence from the P&H text illustrates the pipeline in action.**

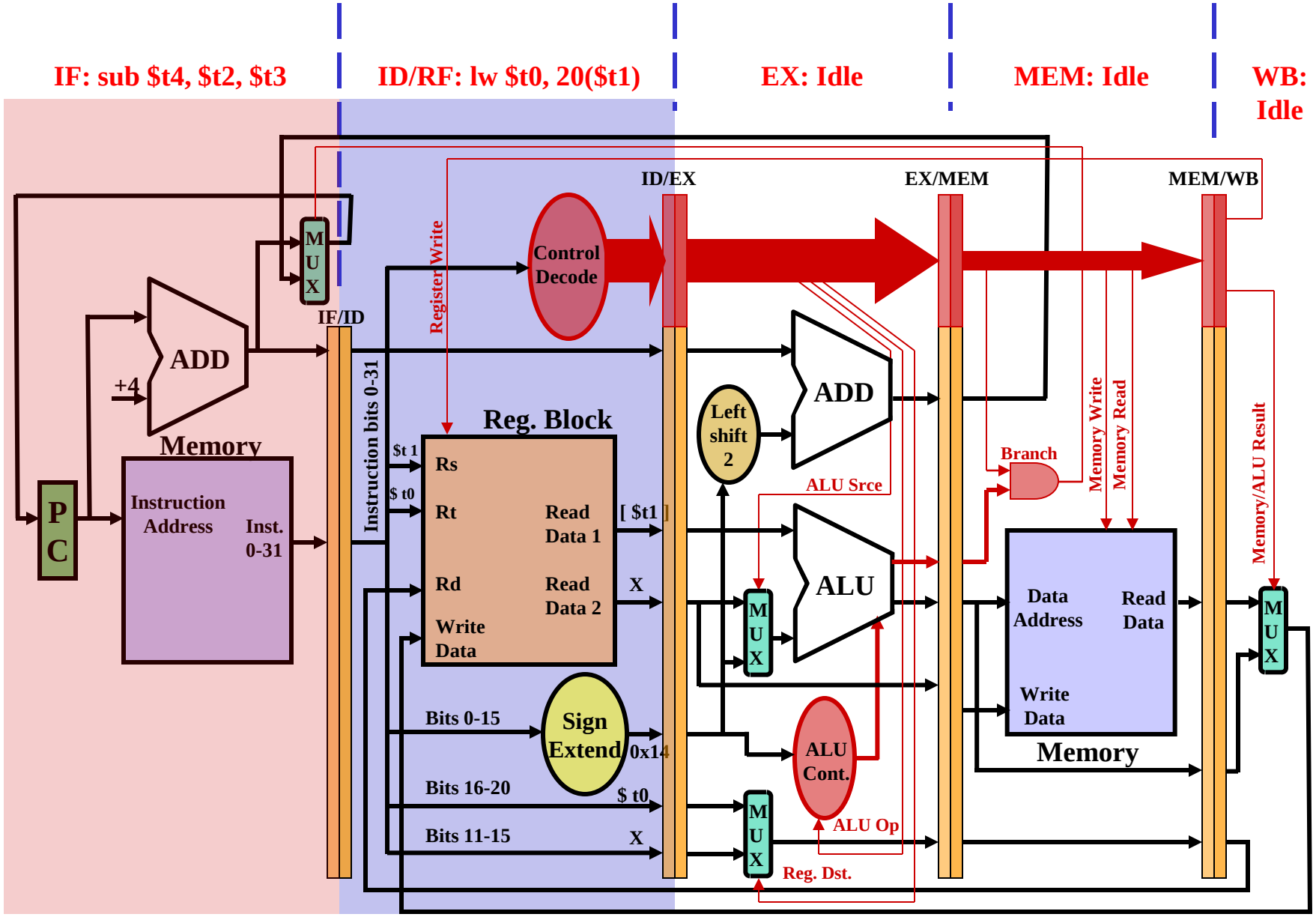
```
lw $t0, 20($t1)
sub $t4, $t2, $t3
and $t7, $t5, $t6
or $s2, $s1, $s0
add $s5, $s3, $s4
```

- **Note that registers are identified by letter id's, not numbers. This is counter to P&H, which uses the number id's. In reading, P&H, remember the number-letter id conversions, e.g., $8-15=$t0-t7, $16-21 = $s0-s5.**

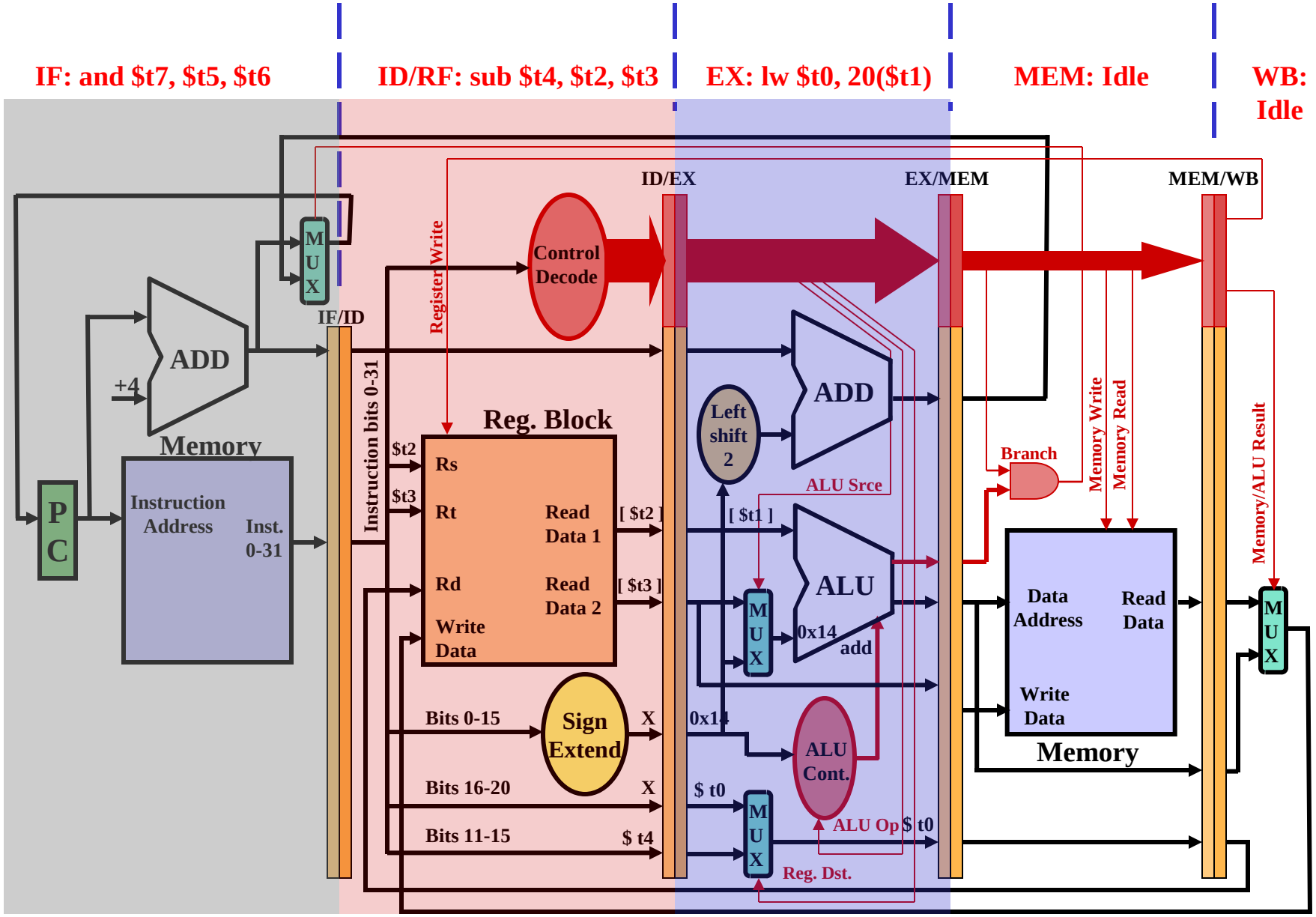

IF: Idle
ID/RF: Idle
EX: Idle
MEM: Idle
WB: Idle
ID/EX
EX/MEM
MEM/WB
Register Write
Control Decode
M U X
IF/ID
ADD
+4
Memory
P C
Instruction Address
Inst. 0-31
Instruction bits 0-31
Reg. Block
Rs
Rt
Rd
Write Data
Read Data 1
Read Data 2
Left shift 2
ADD
ALU Srce
ALU
Branch
Memory Write
Memory Read
Memory/ALU Result
Data Address
Read Data
Write Data
Memory
Bits 0-15
Sign Extend
32
Bits 16-20
Bits 11-15
ALU Cont.
ALU Op
Reg. Dst.



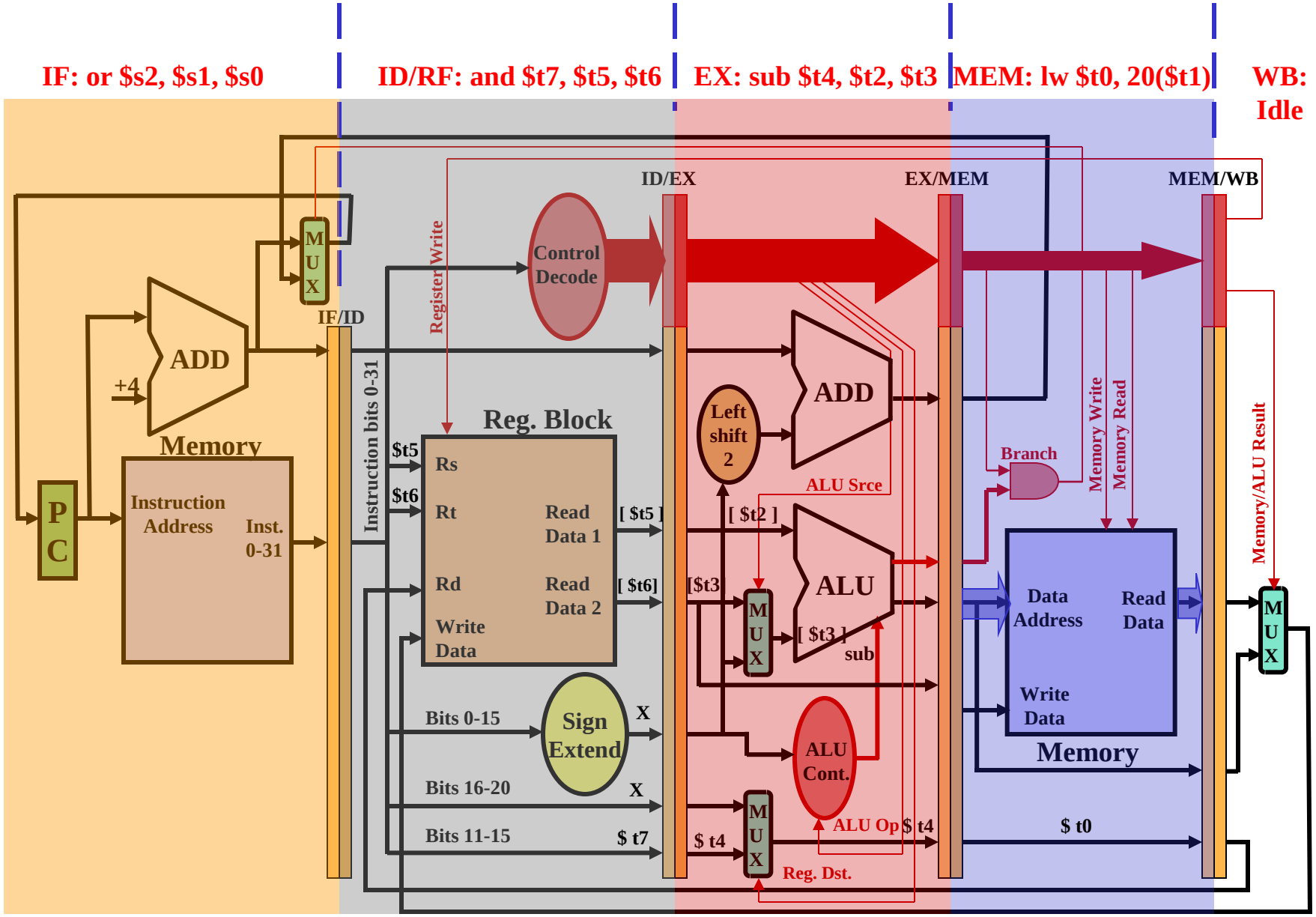
After David A. Patterson and John L. Hennessy, *Computer Organization and Design*, 2nd Edition



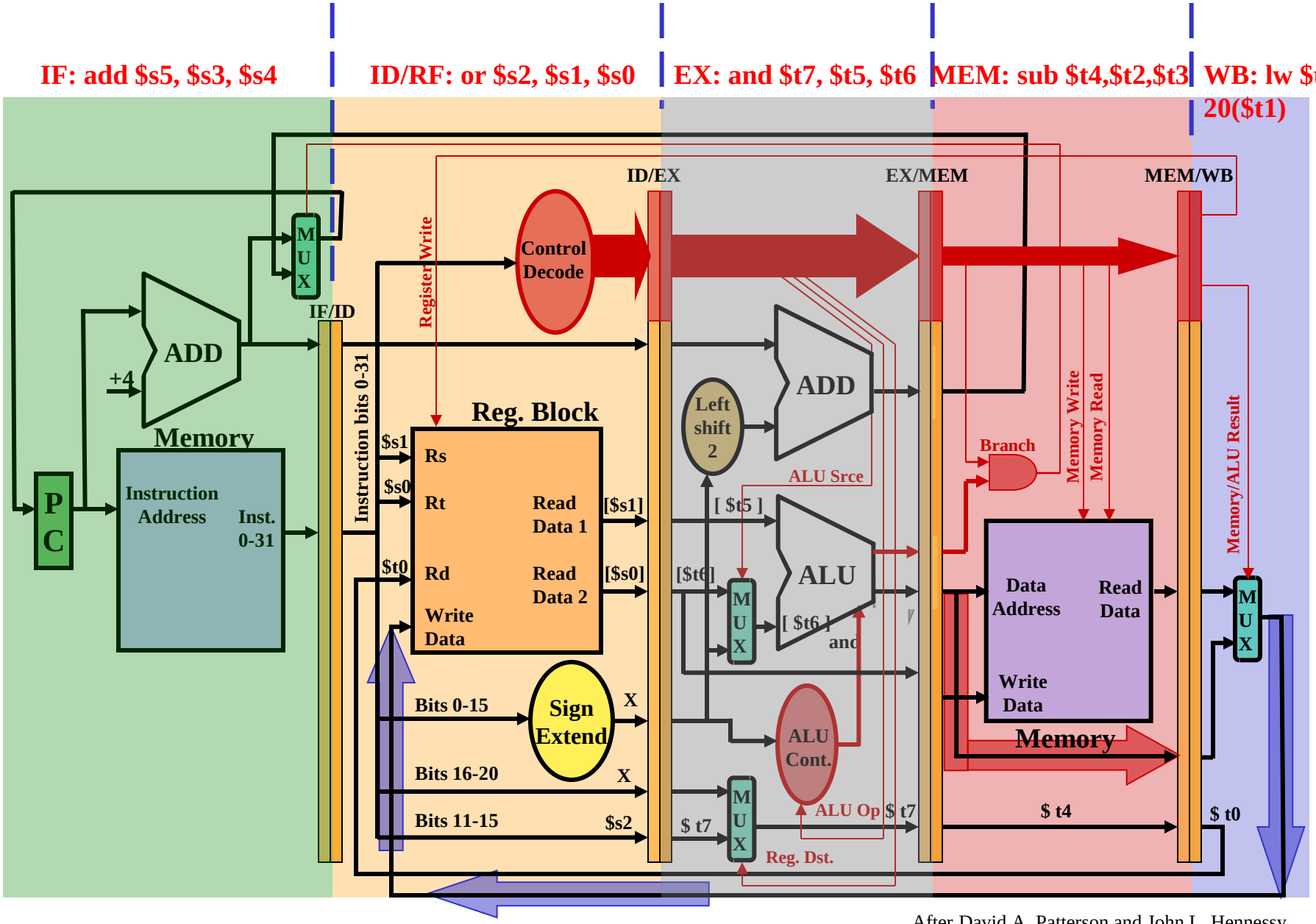
After David A. Patterson and John L. Hennessy, *Computer Organization and Design*, 2nd Edition



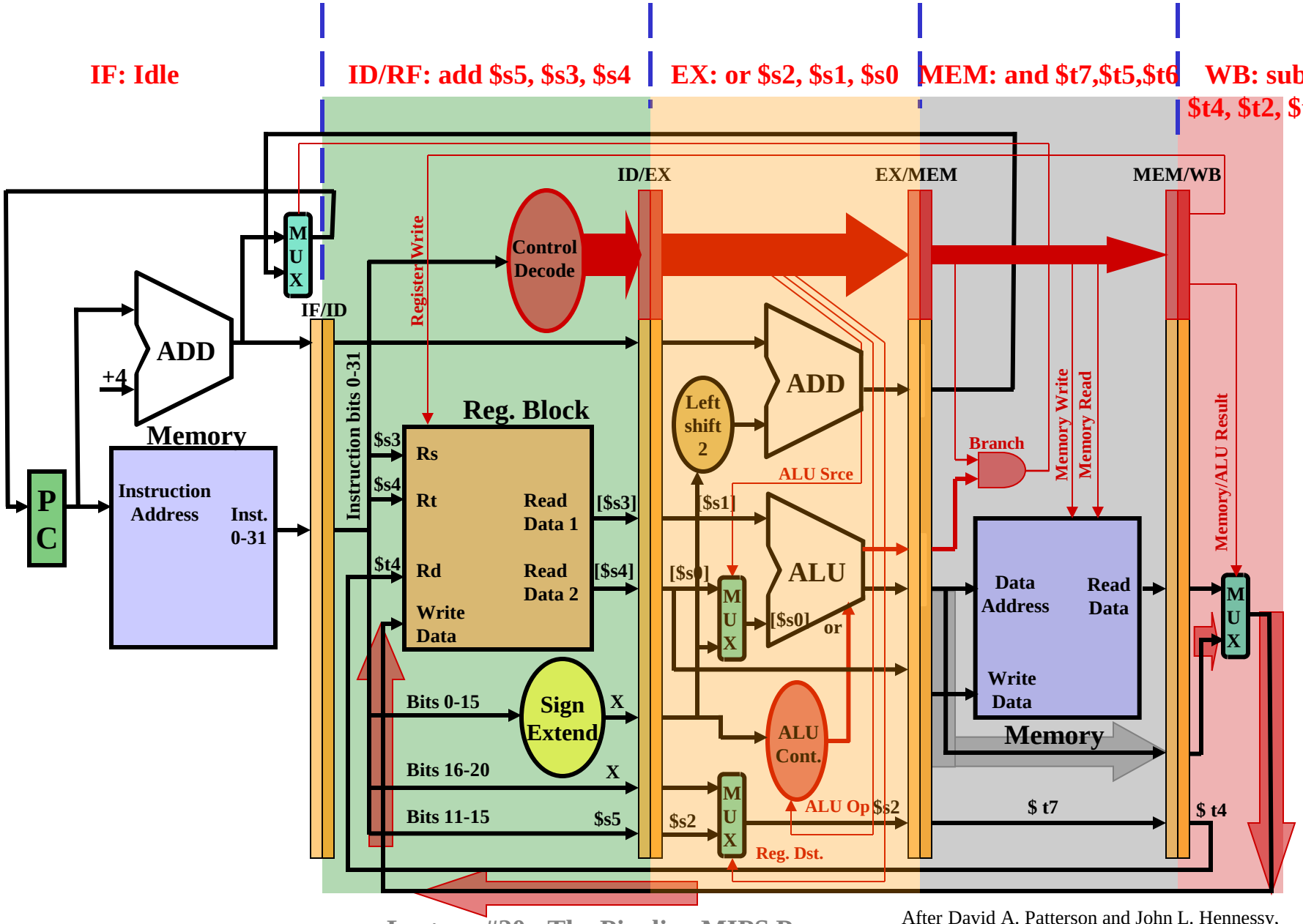
After David A. Patterson and John L. Hennessy, *Computer Organization and Design*, 2nd Edition

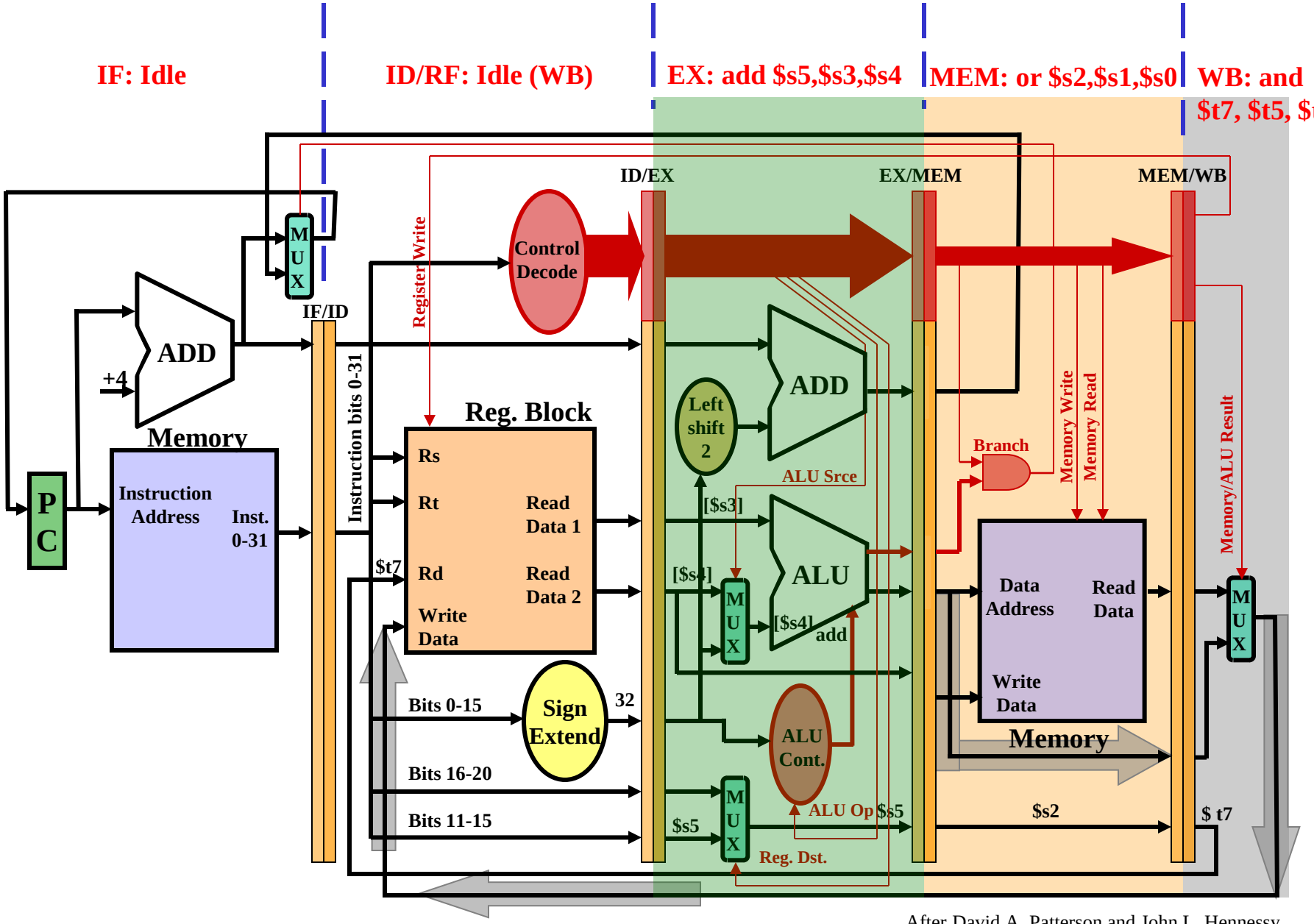


After David A. Patterson and John L. Hennessy, *Computer Organization and Design*, 2nd Edition

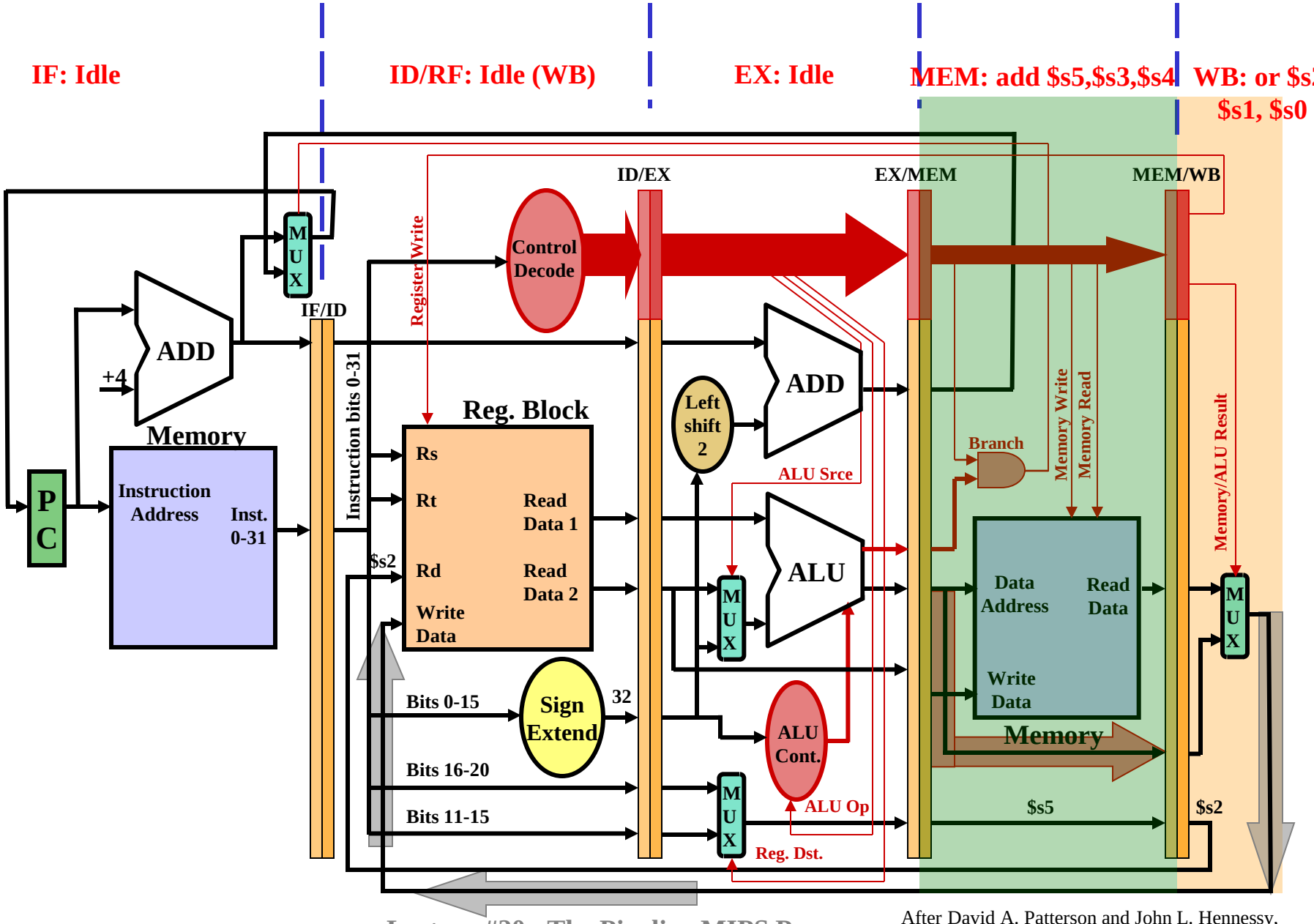


After David A. Patterson and John L. Hennessy, *Computer Organization and Design*, 2nd Edition

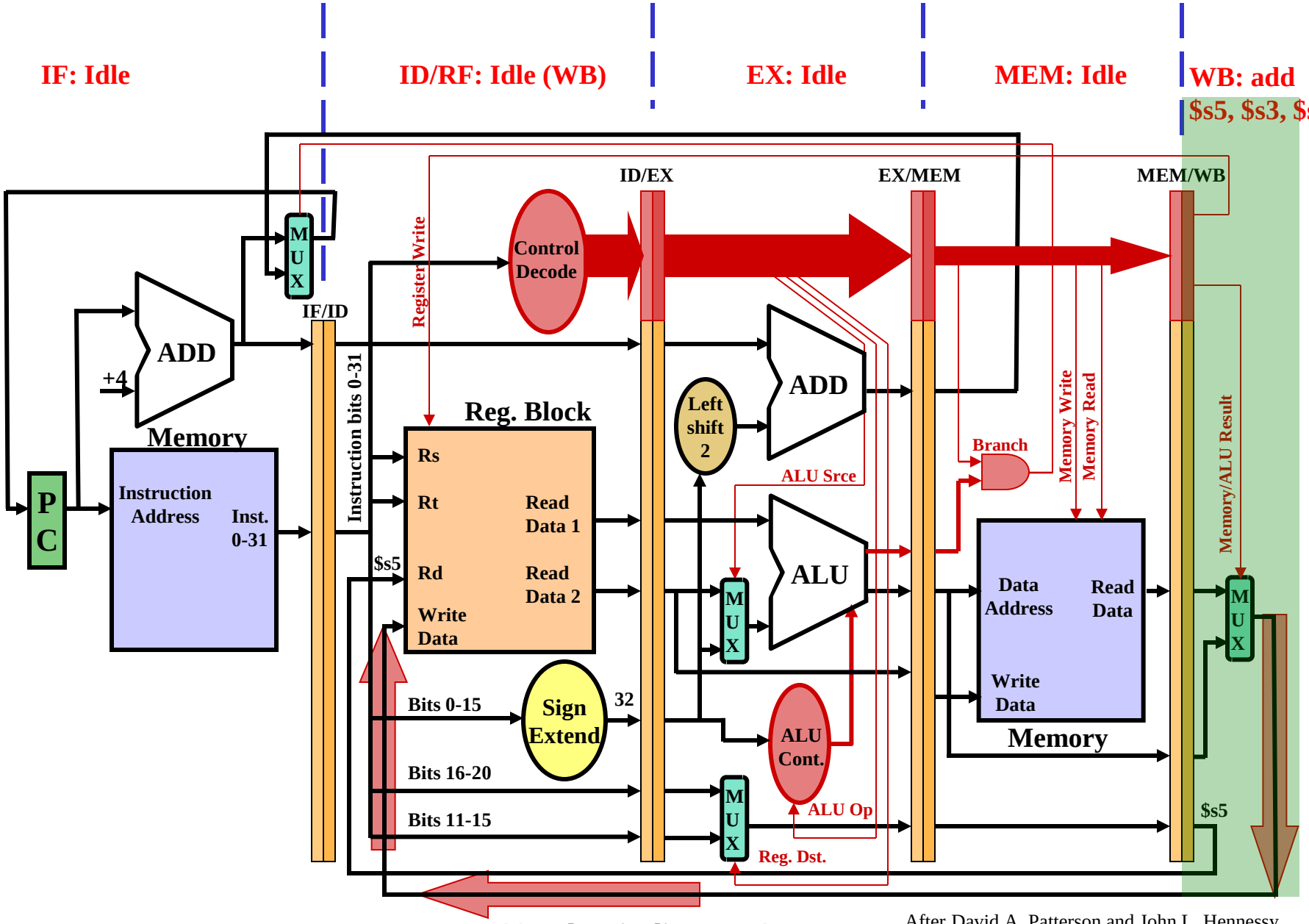

IF: Idle
ID/RF: add $s5, $s3, $s4
EX: or $s2, $s1, $s0
MEM: and $t7,$t5,$t6
WB: sub $t4, $t2, $t3
ID/EX
EX/MEM
MEM/WB
M U X
Register Write
Control Decode
IF/ID
ADD
+4
Memory
P C
Instruction Address
Inst. 0-31
Instruction bits 0-31
Reg. Block
$s3
$s4
$t4
Rs
Rt
Rd
Write Data
Read Data 1
Read Data 2
[$s3]
[$s4]
Left shift 2
ADD
[$s1]
[$s0]
ALU Srce
ALU
[$s0]
or
Branch
Memory Write
Memory Read
Memory/ALU Result
Data Address
Read Data
Write Data
Memory
Bits 0-15
Sign Extend
X
Bits 16-20
X
Bits 11-15
$s5
ALU Cont.
ALU Op
$s2
$s2
$ t7
$ t4
Reg. Dst.



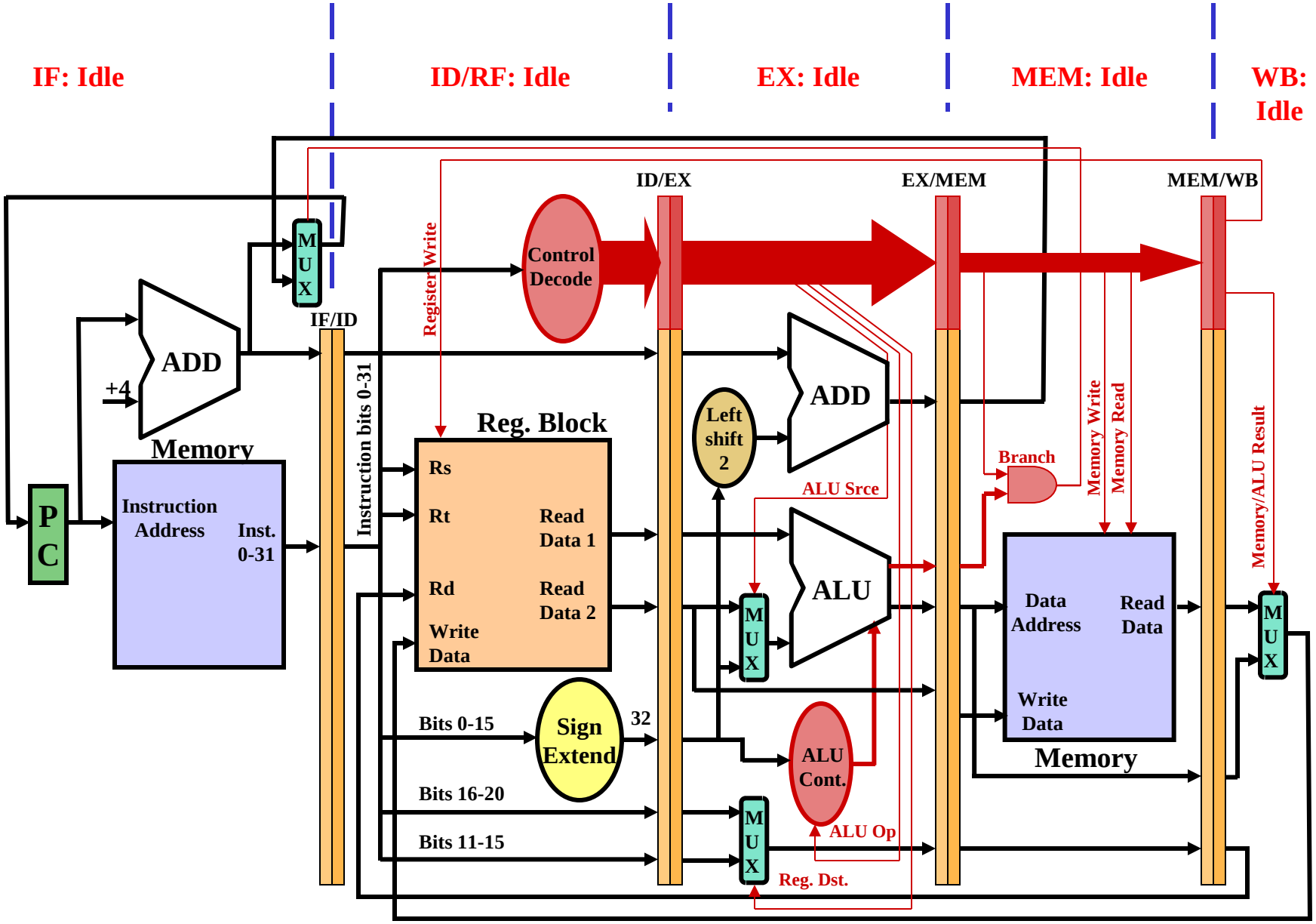
After David A. Patterson and John L. Hennessy, *Computer Organization and Design*, 2nd Edition



After David A. Patterson and John L. Hennessy, *Computer Organization and Design*, 2nd Edition



After David A. Patterson and John L. Hennessy, *Computer Organization and Design*, 2nd Edition


IF: Idle
ID/RF: Idle
EX: Idle
MEM: Idle
WB: Idle
ID/EX
EX/MEM
MEM/WB
M U X
Register Write
Control Decode
IF/ID
ADD
+4
Memory
P C
Instruction Address
Inst. 0-31
Instruction bits 0-31
Reg. Block
Rs
Rt
Rd
Write Data
Read Data 1
Read Data 2
Left shift 2
ADD
ALU Srce
Branch
Memory Write
Memory Read
Memory/ALU Result
ALU
Data Address
Read Data
Write Data
Memory
Bits 0-15
Sign Extend
32
Bits 16-20
Bits 11-15
ALU Cont.
ALU Op
Reg. Dst.

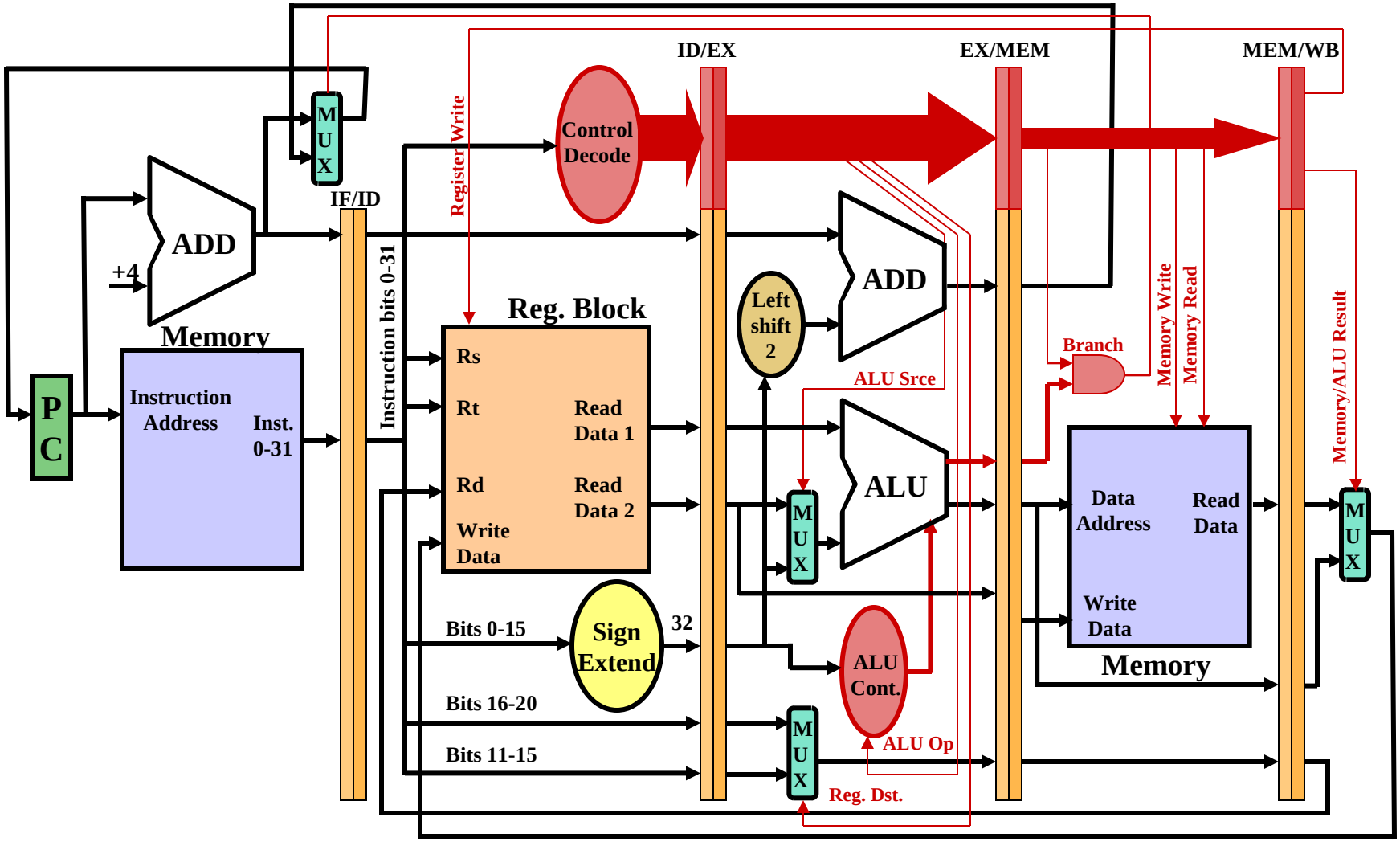
Pipeline Processor Operation Summary

- **Pipelining replaces the "single-cycle" processor with a row of five "mini-processors," each capable of completing one part of each instruction.**
- **A new instruction is started every clock cycle.**
- **Inter-process registers store instruction information (data, write register, branch conditions) between cycles so that the entire "instruction envelope" is passed between the pipeline stages.**
- **When the pipeline is filled with instructions, an instruction completes every clock cycle.**

Exercise 1

- **On the diagram on the next page, identify the following:**
  1. **Highlight all the control lines that must be active during a load word instruction.**
  2. **As in our exercise in Lecture 20, identify the decoder locations.**
  3. **The ID/EX Register interface stores the most bits of any of the pipeline section interfaces. Approximately how many bits is that, according to the diagram?**

Print out a copy of this diagram and bring to class.



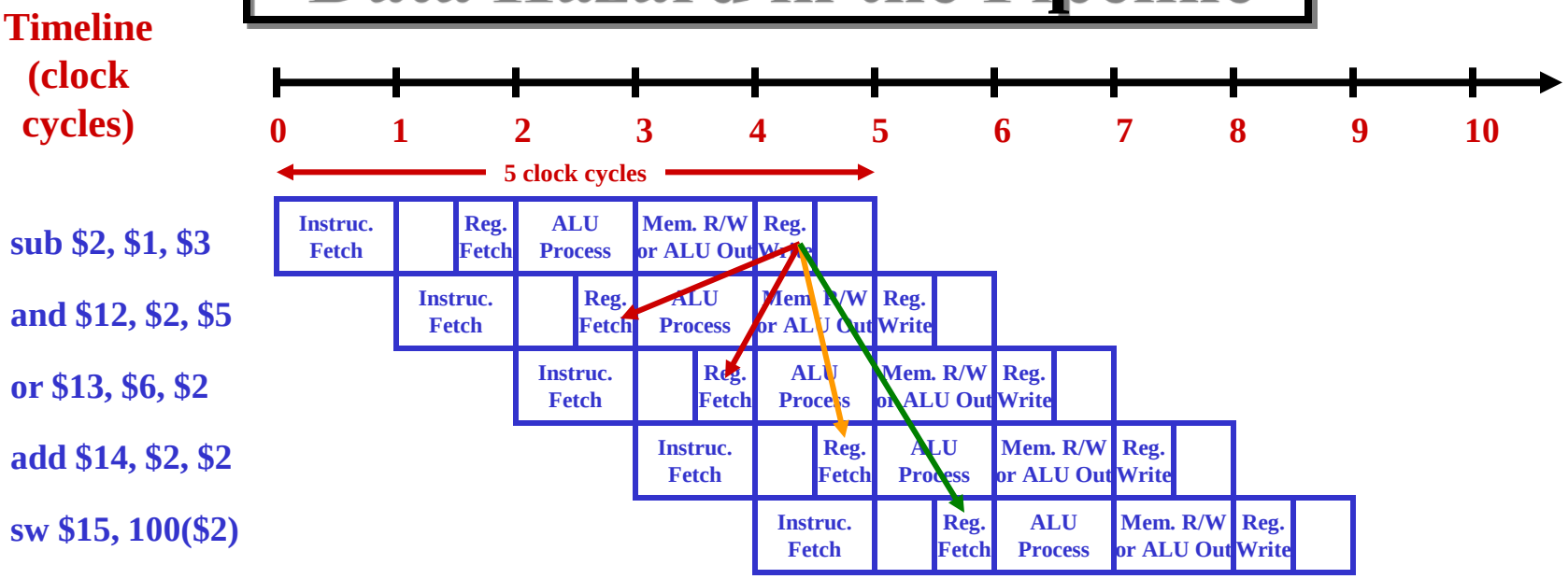
Hazards

- **Hazards occur because data required for executing the current instruction may not be available.**
- **An instruction in the "register fetch" cycle may need data from a register whose value will be changed by an instruction "downstream" but still in process in the pipeline (in the ALU, memory/memory bypass or writeback cycle).**
- **Thus an "upstream" instruction could access a register and get incorrect data because the register data has not yet been updated by a "downstream" instruction.**

Hazards (2)

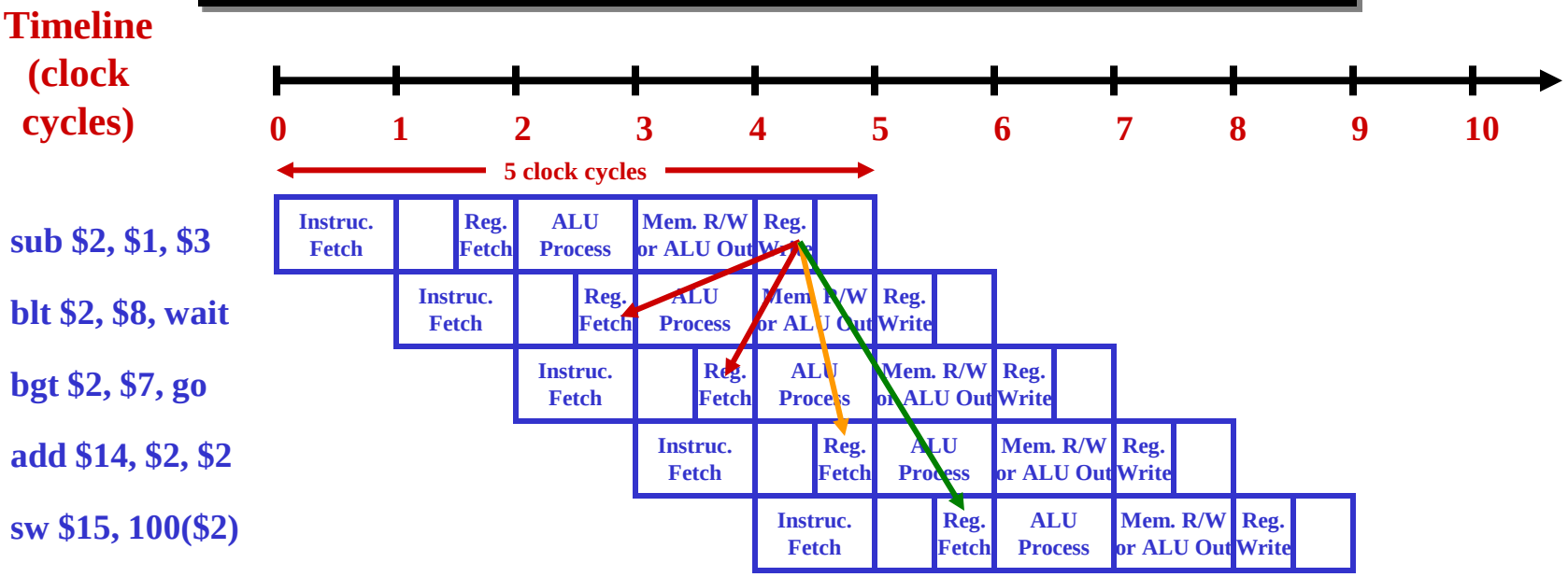
- There are two types of hazards, data hazards, and control hazards.
- Both occur because an instruction in the ID/RF stage of the MIPS pipeline needs register data that will be shortly updated by instructions in the EX or MEM/Bypass, or WB stage.
- Data hazards occur when an instruction needs register contents for an arithmetic/ logical/memory instruction.
- Control hazards occur when a branch instruction is pending and the data necessary to initiate/bypass the branch is not yet available in the same sort of scenario.

Data Hazard in the Pipeline



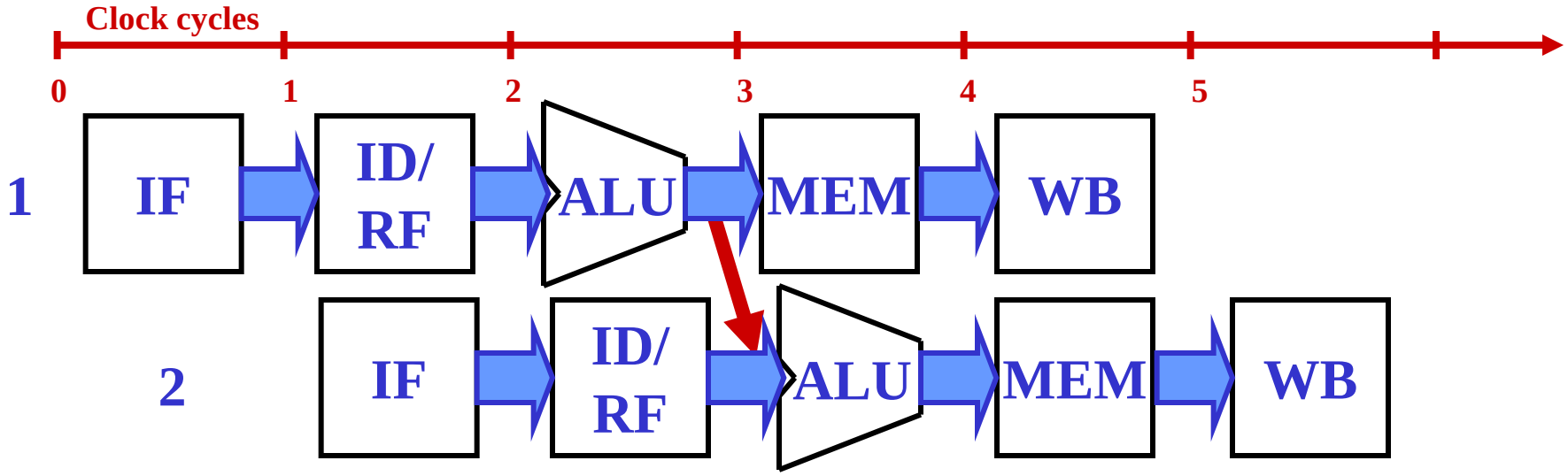
- **In the instruction sequence above, the last four instructions require data from $2, which is changed in the first instruction.**
- **The $2 data will not be rewritten until cycle 4, so the AND and OR (2nd and 3rd instructions) will fetch incorrect data from $2.**
- **Even the add may not get the correct information (sw is okay).**

Control Hazards in the Pipeline



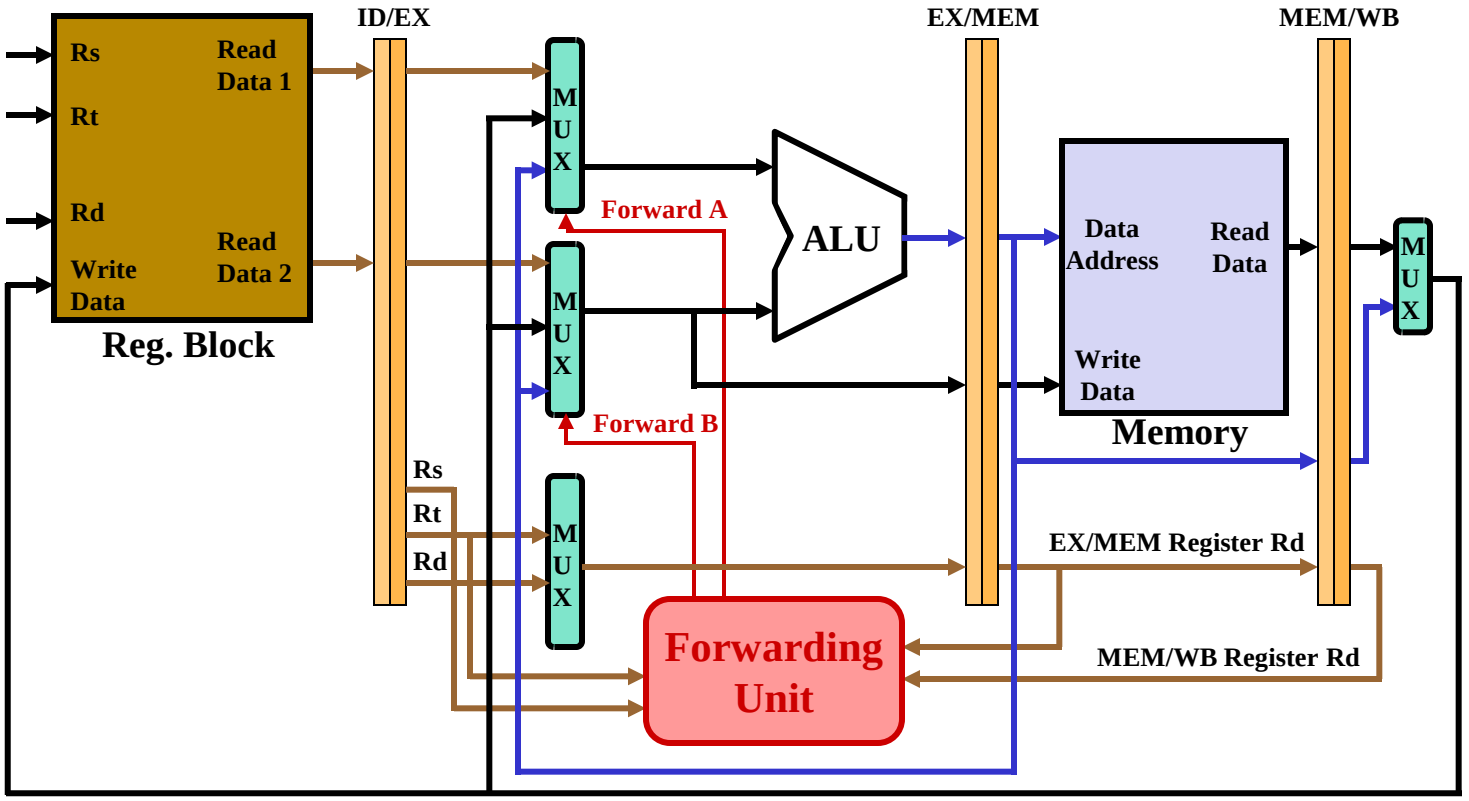
- Here the problem is changed, with two branch instructions added.
- Neither branch instruction may be executed correctly, once again because the new $2 data will not be ready.
- This wrong data could cause an incorrect branch.

Forwarding as a Solution to Data Hazards



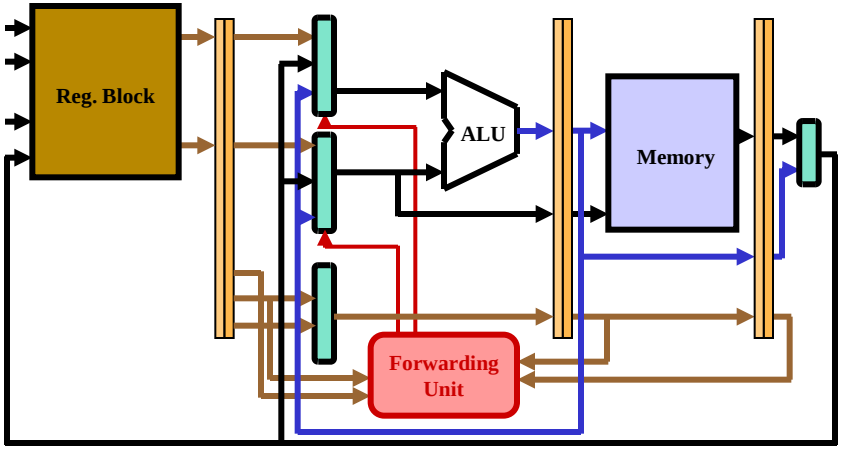
- One solution to the problem of data hazards is forwarding.
- Forwarding uses the fact that although instruction 2 needs register data two clock cycles before instruction 1 enters the WB stage, that data is already available as the output of the ALU.
- If a mechanism were available, instruction 1 could forward required register data after its ALU cycle to the ID/RF cycle of instruction 2.

Forwarding Unit in the Pipeline



After David A. Patterson and John L. Hennessy, *Computer Organization and Design*, 2nd Edition

Forwarding Unit Operation

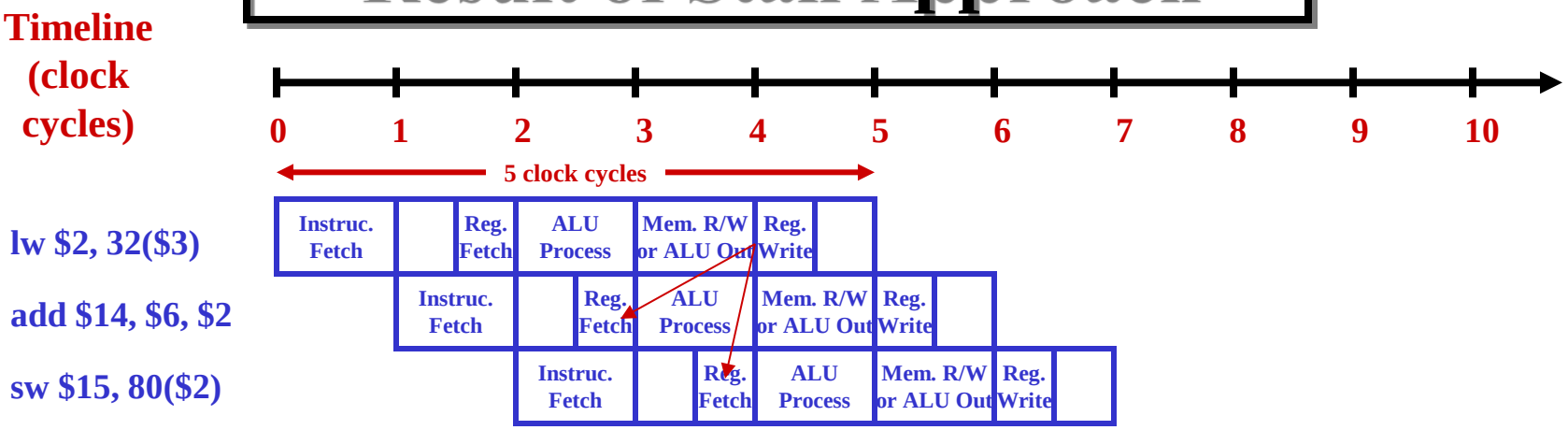


- The forwarding unit samples register id's in the EX/MEM and MEM/WB registers to determine if source registers in the ID/RF cycle are the same.
- If so, source register data is replaced by pipeline (as yet unwritten) data by the forwarding unit.
- The correct information is thus processed and the instruction can proceed to correct execution.

Stalls

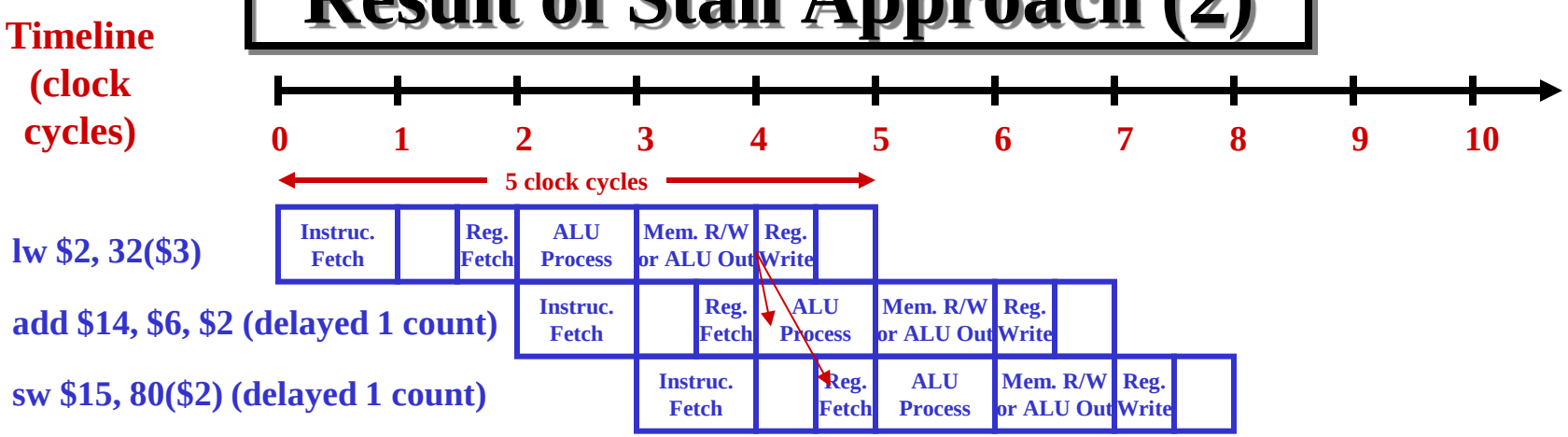
- **Forwarding will not always solve the problems of data hazards.**
- **For example, suppose an add instruction follows a load word (lw), and the add involves the register that receives the memory data.**
- **In this case, forwarding will not work.**
- **The reason is that the data must be read from memory, and so it will not be available until the end of the MEM cycle. Thus the required data is not available for a forward, and the add instruction. if it proceeds, will process the wrong data.**
- **A solution to this problem is the stall.**
- **A stall halts the instruction awaiting data, while the key instruction (a lw in this case) proceeds to the end of the MEM cycle, after which the desired data is available to the add.**

Result of Stall Approach



- **Consider the 3 instructions above, the last two depending on the lw.**
- **$2 contents will be available at the beginning of the WB stage in the first instruction, but not before.**
- **A solution is to let the lw proceed down the pipe, while the add and sw instructions hold place for one cycle.**

Result of Stall Approach (2)



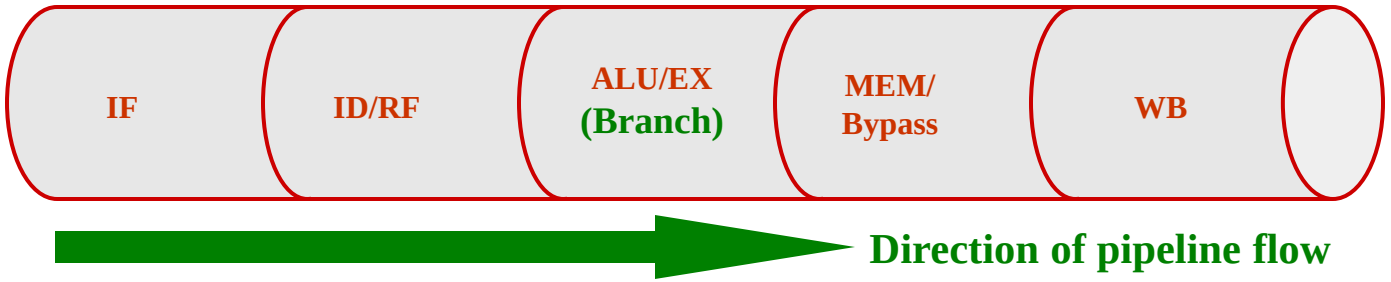
- With the delay, the lw result feeds the ALU input stage of the add instruction, and the fetch stage of the sw.
- Note that forwarding in still required (this time from the MEM/WB interface, not the ALU output).
- However, in addition to forwarding, instructions following a lw must also be delayed for one clock cycle.

Other Problems With Branches

- A remaining problem is what to do about instructions following a branch. Even assuming forwarding and stalls, the branch/no branch decision is not made until the third stage. This means that in the MIPS pipeline, two following instructions will enter the pipe before the branch/no branch decision is made. What if:
  - The following instructions were for the case of "branch taken" and the branch was not taken.
  - The following instructions were for "branch not taken" and it was taken.
- In either case, the wrong instructions are in the pipe and they must be eliminated ("flushed"). How can this problem be prevented?
- A few approaches to the problem are shown in the following slides.

Control Hazard Approaches (1)

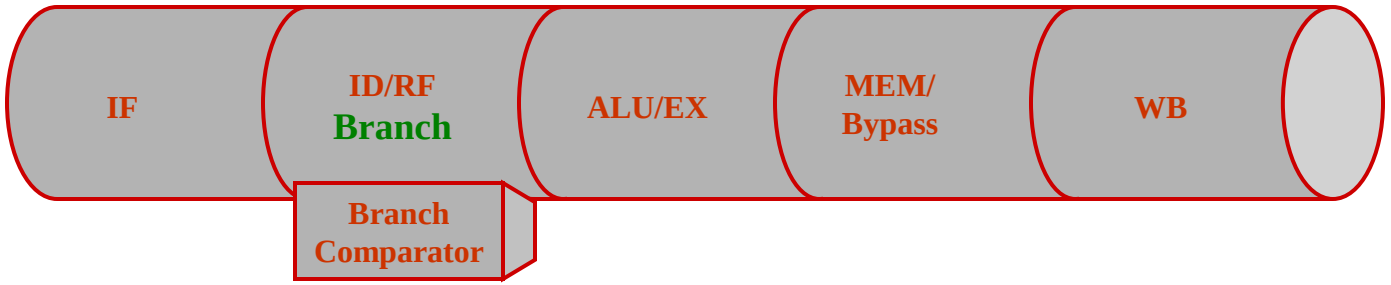
MIPS R-2000 Pipeline Processor



- **One approach is to always assume the branch is (or is not) taken:**
  - **Say we assume the branch is never taken. Then if the instruction in ALU/EX is a branch, the instructions in IF and ID/RF will be those in the "not taken" program line (branch determination is made in ALU/EX).**
  - **It this assumption is correct, the pipeline will continue to flow without delay.**
  - **When the branch is taken, instructions in IF and ID/RF must be "flushed," usually by changing the "op" code of those instructions to a "nop" and letting them proceed to the end of the pipe.**
  - **Clearly, a 2-clock time delay is involved here, and it would be worse for longer pipelines (P-IV pipeline ~ 20 stages).**

Control Hazard Approaches (2)

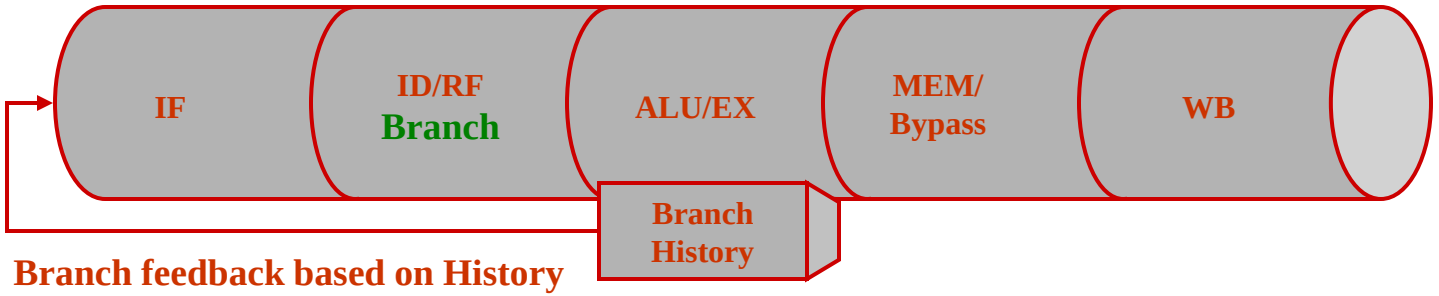
MIPS R-2000 Pipeline Processor



- **Reducing the cost of taking the branch:**
  - In this case, a branch assumption is still made (taken or not taken).
  - The difference is that since register contents (and/or immediates) are identified in the ID/RF stage, a comparator can be added there to do the branch/no-branch determination.
  - With the branch determination made in this early stage, only one instruction must be flushed, in the IF stage (only a 1-instruction delay).

Control Hazard Approaches (3)

MIPS R-2000 Pipeline Processor



- **Dynamic branch prediction based on recent branch history:**
  - In this approach, an indicator bit (0/1) gives the last branch condition.
  - The next branch can be made according to the bit setting.
  - This is useful in highly repetitive loops, which may continue for a long time until a substantial number of calculations are complete.
  - Some schemes use 2 bits and do not change the prediction until the predictor is wrong twice, after which the alternate behavior is chosen.
  - In either case, incorrect predictions will still be made, but hopefully not as often.

Exercise 2

1. **Explain forwarding in your own words.**
2. **Why doesn't forwarding always work? How can this problem be solved?**
3. **Why could 2-bit dynamic branch prediction work to ensure about a 1% error rate in branch prediction in a subroutine that loops about 100 times before completion? Hint: Assume that the subroutine is called frequently, and that it always executes 100 or more loop traversals before returning to the calling program.**

Summary

- The pipeline approach to CPU design provides a significant speed increase over a single-cycle design, up to several hundred percent.
- The improvement is so dramatic that today, all general-purpose processor units (now usually clustered in groups of 2, 4, 6, 8 or more CPU's) are designed using a pipeline approach.
- However, this performance improvement must be paid for with increased (1) price, and (2) complexity. Pipelines introduce processing problems, including:
  - Hazards
  - Incorrect branch prediction
- The increased price and complexity come about in the hardware approaches to solving these and other similar problems.