11. Basic processor design – introduction to pipelining

EECS 370 – Introduction to Computer Organization

Fall 2013

Profs. Valeria Bertacco, Robert Dick, and Satish Narayanasamy

EECS Department
University of Michigan in Ann Arbor, USA

Announcements

- Project 2
  - Due 25 Oct.
- Competition
  - Due 28 Oct.
  - Win a Raspberry Pi.
- Homework 4
  - Due 24 Oct.

The problem with single-cycle datapaths

1 ns – Register read/write time

2 ns – ALU/adder

2 ns – memory access

0 ns – MUX, PC access, sign extend, ROM

Instr.	Get instr.	Read reg.	ALU	Mem.	Write reg.	Total
add	2 ns	1 ns	2 ns	0 ns	1 ns	6 ns
beq	2 ns	1 ns	2 ns	0 ns	0 ns	5 ns
sw	2 ns	1 ns	2 ns	2 ns	0 ns	7 ns
lw	2 ns	1 ns	2 ns	2 ns	1 ns	8 ns

Review: execution time

Assume 100 instructions executed

- 25% of instructions are loads,
- 10% of instructions are stores,
- 45% of instructions are adds, and
- 20% of instructions are branches.

Single-cycle execution:

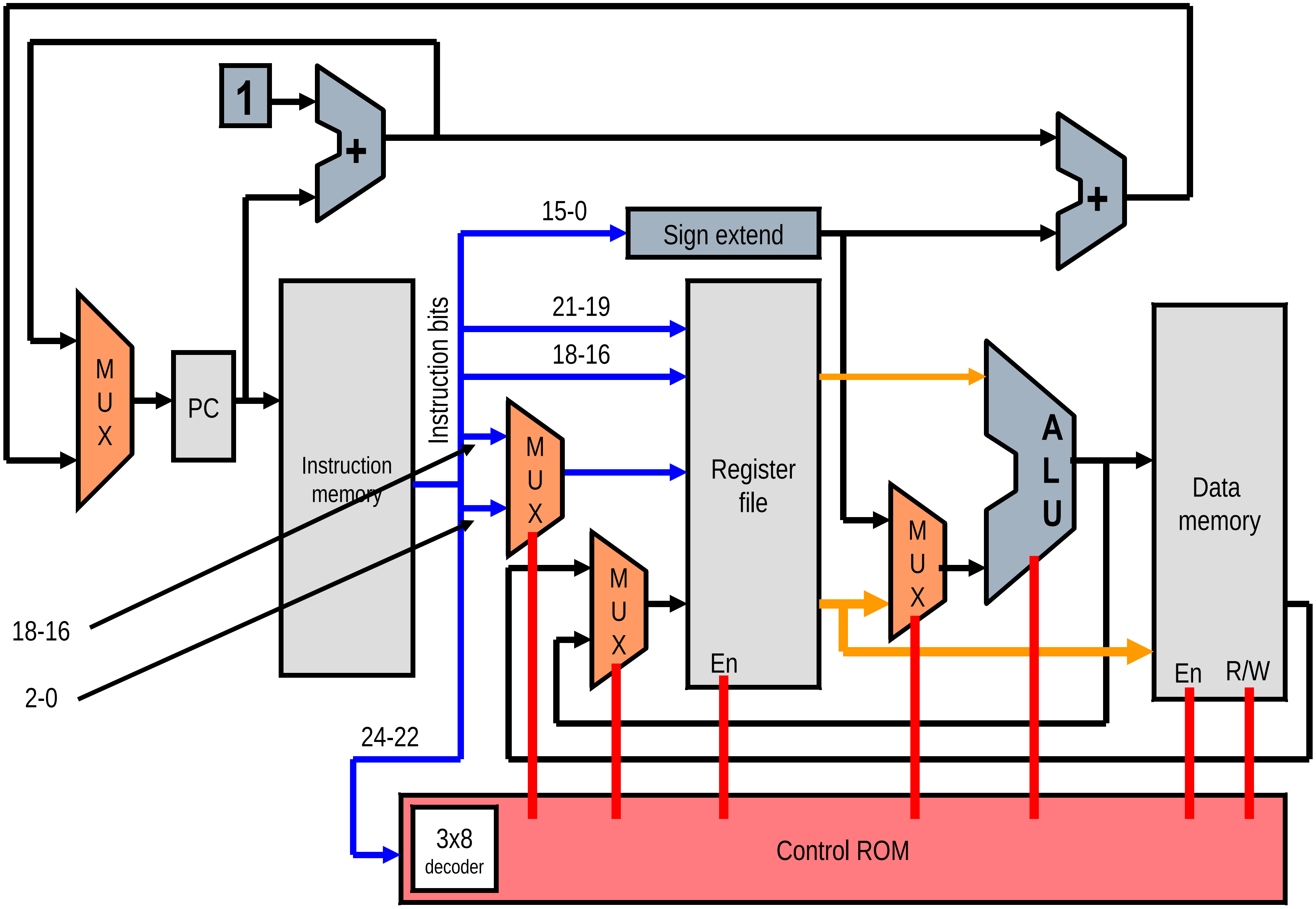
- 100 · 8ns = <u>**800**</u> ns

Optimal execution:

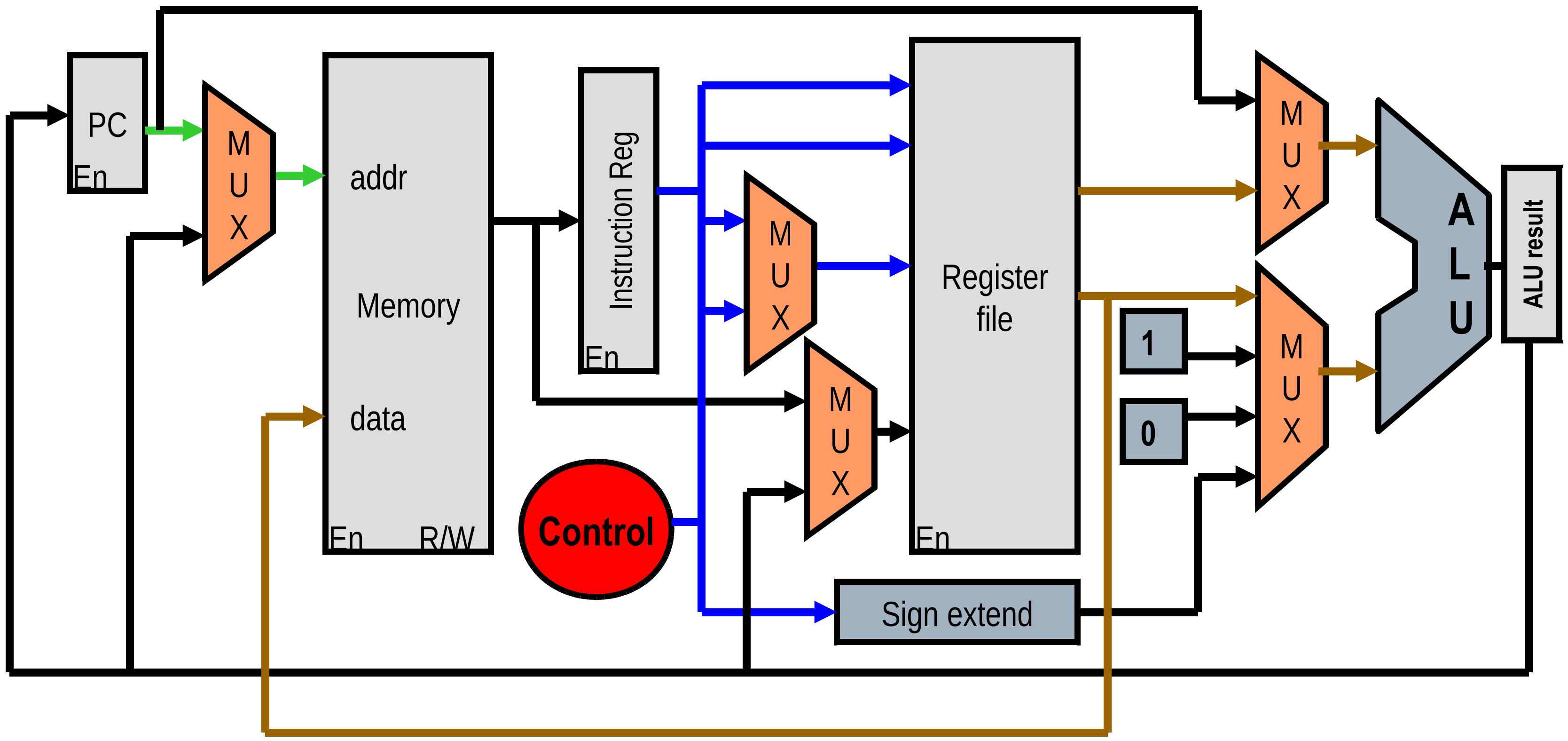
- 25·8ns + 10·7ns + 45·6ns + 20·5ns = <u>**640**</u> ns

In reality, overhead to support multicycle.

Review: single-cycle LC2Kx datapath



Review: multi-cycle LC2Kx datapath



Review: single-cycle and multi-cycle performance

Given:

- 2 ns – Register read/write time
- 1 ns – ALU/adder
- 2 ns – memory access
- 0 ns – MUX, PC access, sign extend, ROM

For program with 100 instructions:

- 25 lw, 10 sw, 45 add, 20 beq

For SC and MC datapaths:

- What is the cycle time
- How many cycles to execute?

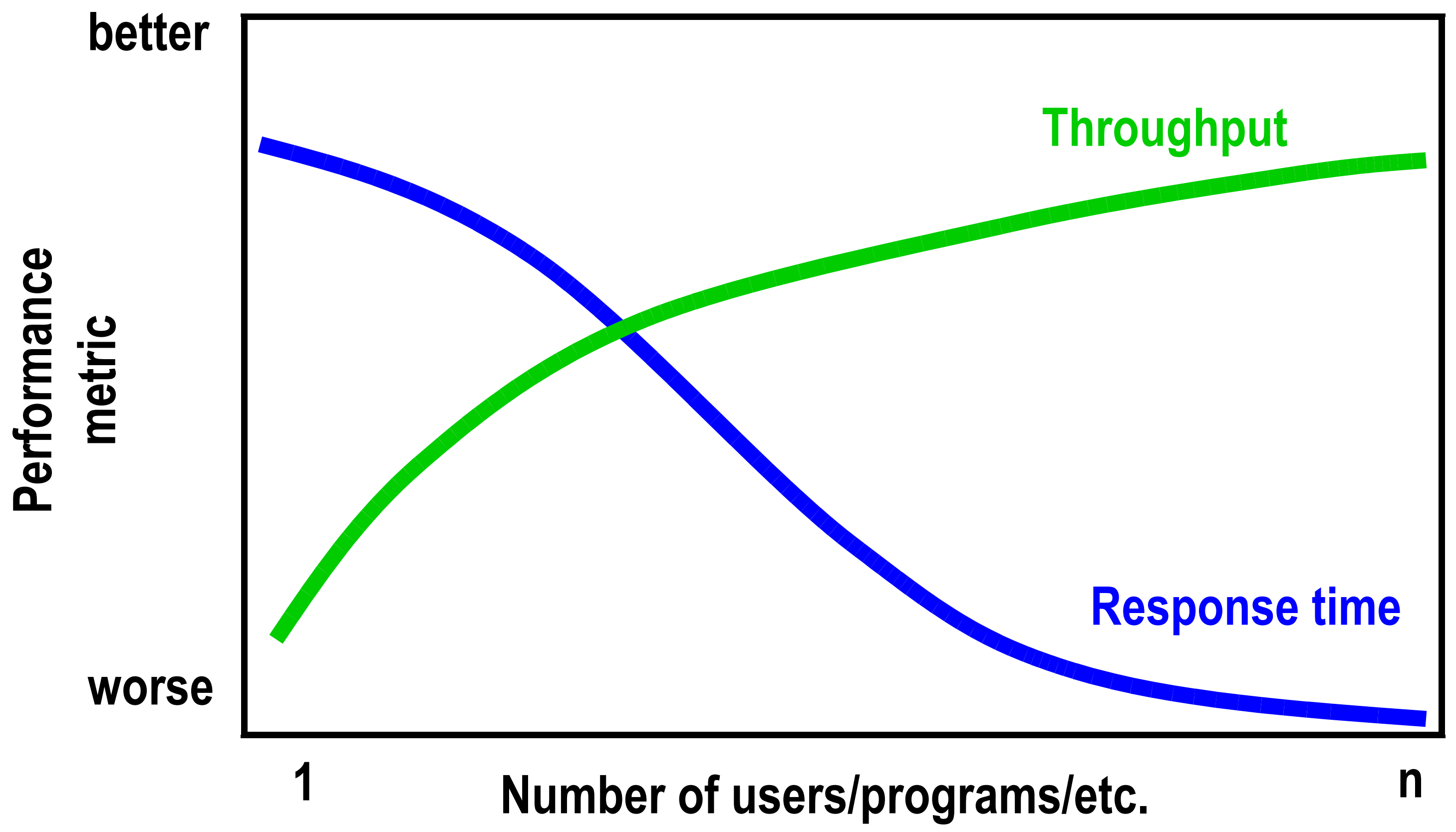
Performance metrics

1. **Response time**: From job submission to job completion, for individual job.
   - When is my job done (time)?
   - How long will this program/instruction take?

1. **Throughput**: In the steady state, what rate are jobs completed at?
   - How many jobs can be finished in two hours (when individual jobs are short relative to two hours)?
   - How many programs/instructions complete per hour?
   - Improved relatively easily by using multiprocessors.

Relating response time to throughput

More throughput ↔ higher response time (*Little's law*)
Works when individual jobs are short relative to response time.



Performance metrics – execution time

- Response time for a program is its **execution time.**

Execution time (for an application):

= total instructions executed x CPI x clock period

- **Called the "Iron Law" of performance**

- CPI = avg number of clock cycles per instruction *for an application.*

- For multi-cycle processor implementations we need
  - Cycles necessary for each type of instruction.
  - Mix of instructions executed in the application (dynamic instruction execution profile).
  - Cache state can also influence this.

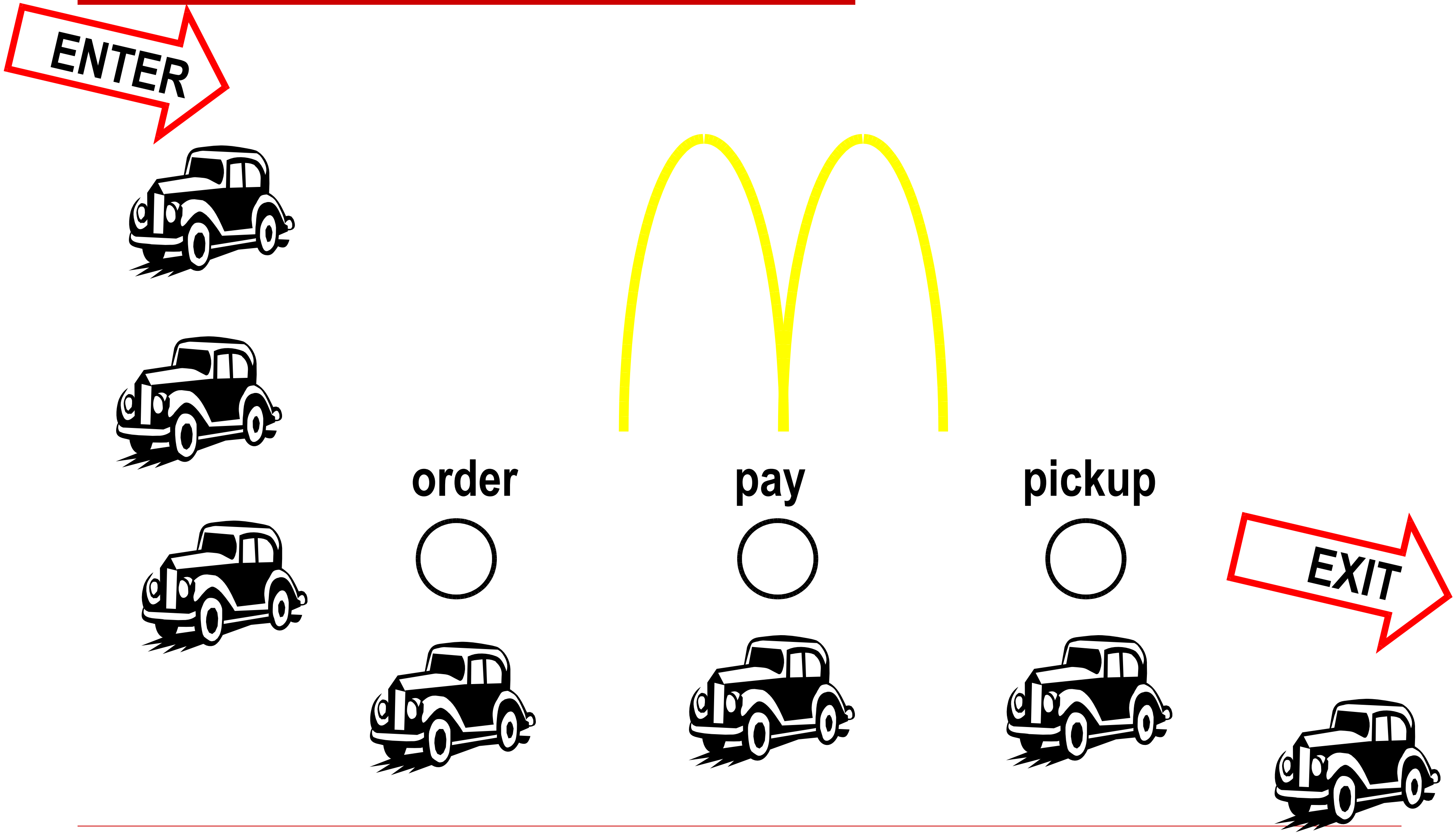
What are we building to?

- Single-cycle processor implementations
  - CPI = ?
  - clock period = ?
- Multi-cycle processor implementations
  - CPI = ?
  - clock period = ?
- Next step: improve CPI without much hurting clock period
  - Work on different parts of multiple instructions at the same time.

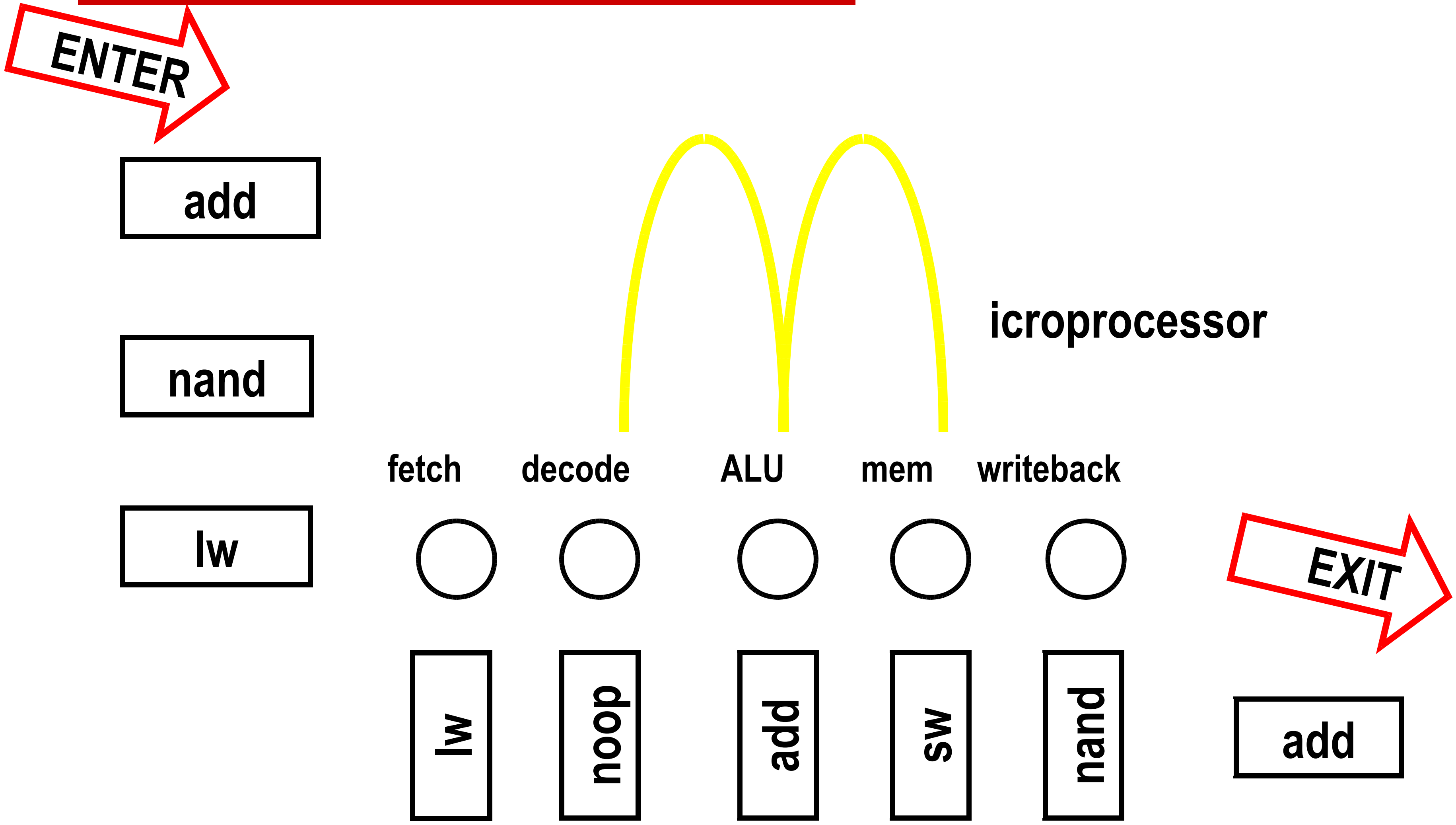
Pipelining

- Want to execute an instruction?
  - Build a processor (multi-cycle)
  - Find instructions
  - Line up instructions (1, 2, 3, …)
  - Overlap execution
    - Cycle #1: Fetch 1
    - Cycle #2: Decode 1 Fetch 2
    - Cycle #3: ALU 1 Decode 2 Fetch 3
    - . . . . . .
  - This is called pipelining instruction execution.
  - Used extensively for the first time on IBM 360 (1960s).
  - CPI approaches 1.

Pipelining



Pipelining


ENTER
add
nand
lw
icroprocessor
fetch
decode
ALU
mem
writeback
EXIT
lw
noop
add
sw
nand
add

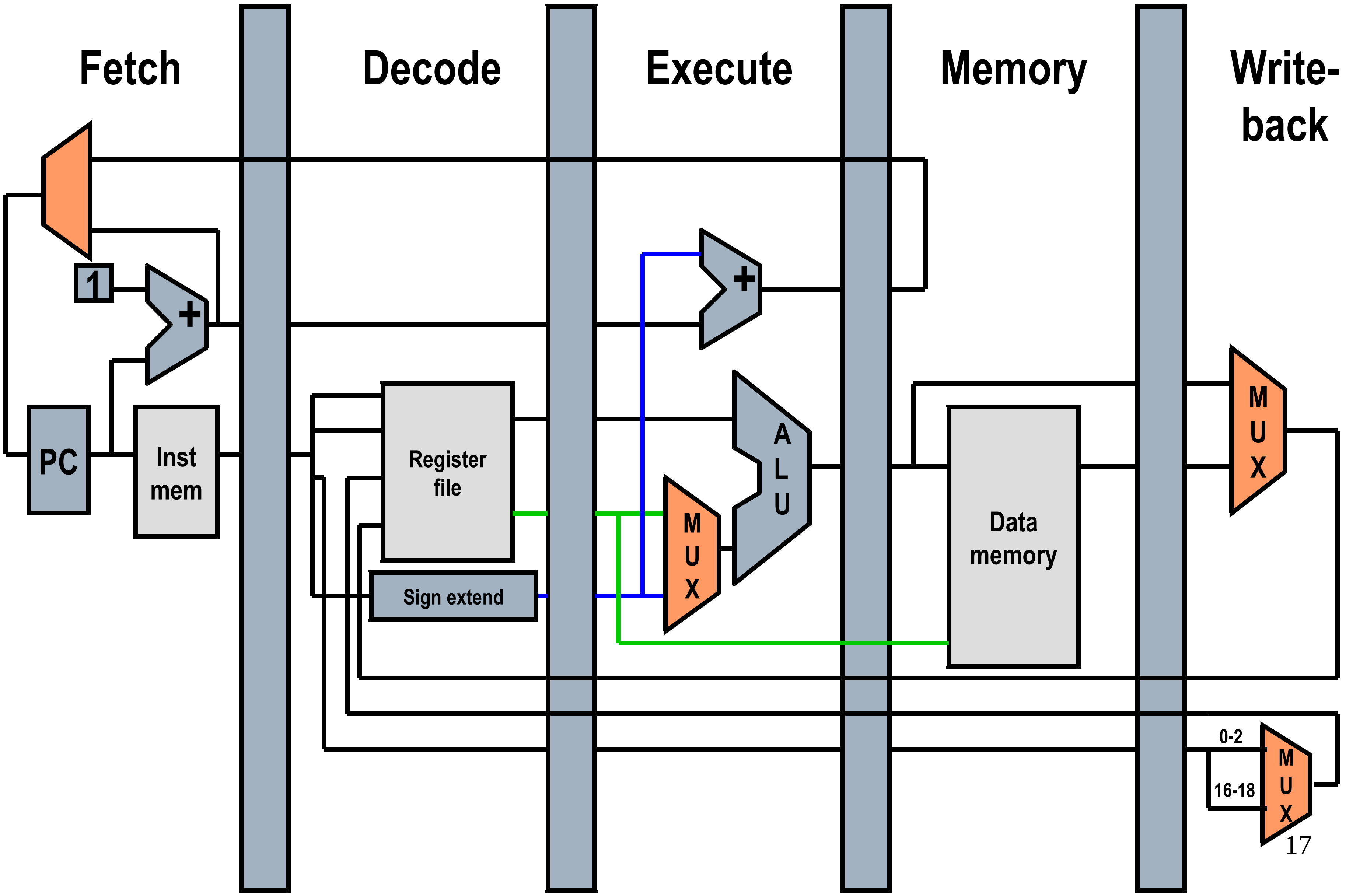
Pipelining trends

- Execute numerous instructions at the same time.
  - Pipelining: 12-20+ cycles.
  - Multiple pipelines.
- Pentium
  - 2 pipelines, 5 cycles each (10 instructions “in flight”).
- Pentium Pro/II/III
  - Roughly 3 pipelines, 12 cycles deep.
  - Instructions can execute out of their original program order.
- Pentium IV
  - 4 pipelines, 20 cycles deep.

Pipelined implementation of LC2Kx

- Break the execution of the instruction into cycles.
  - Similar to the multi-cycle datapath.
- Design a separate datapath stage for the execution performed during each cycle.
  - Build pipeline registers to communicate between the stages.

Pipelined datapath

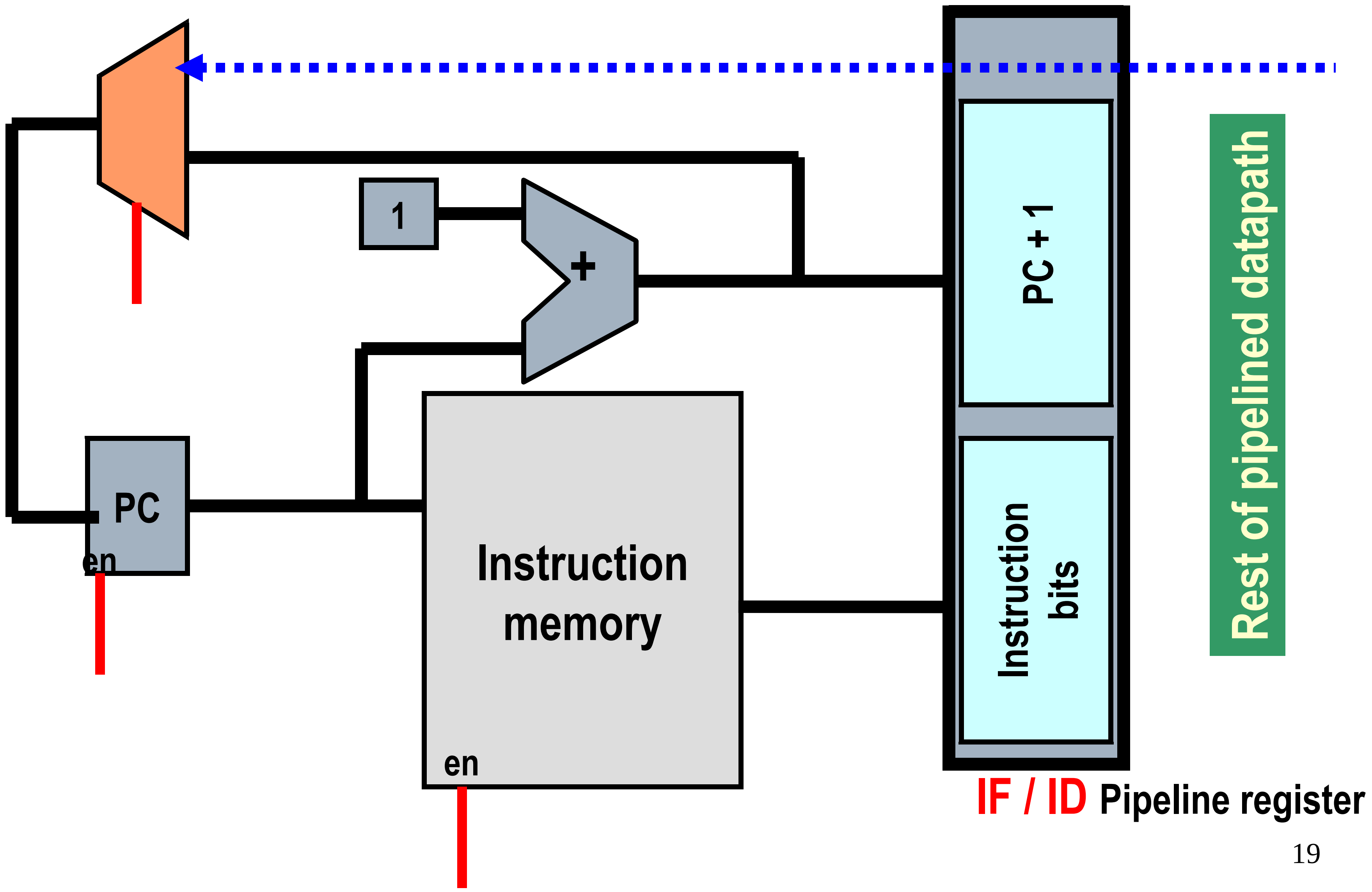


Stage 1: fetch

- Design a datapath that can fetch an instruction from memory every cycle.
  - Use PC to index memory to read instruction.
  - Increment the PC (assume no branches for now).

- Write everything needed to complete execution to the pipeline register (IF/ID)
  - The next stage will read this pipeline register.
  - Note that pipeline register must be edge-triggered.

Pipeline datapath – fetch stage



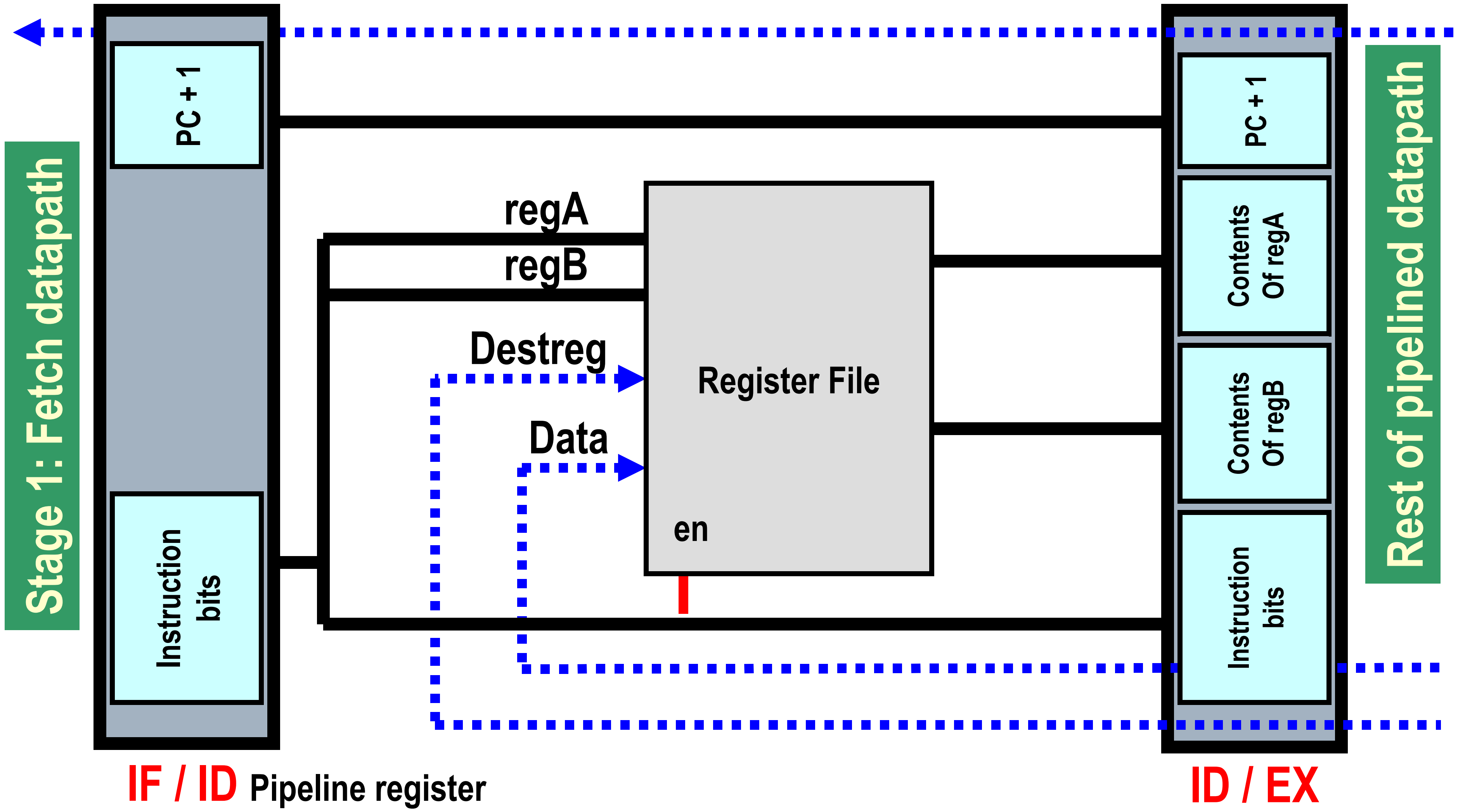
IF / ID Pipeline register

Stage 2: decode

- Design a datapath that reads the IF/ID pipeline register, decodes instruction and reads register file (specified by regA and regB of instruction bits).
  - Decode is easy, just pass on the opcode and let later stages figure out their own control signals for the instruction.

- Write everything needed to complete execution to the pipeline register (ID/EX).
  - Pass on the offset field and both destination register specifiers (or simply pass on the whole instruction!).
  - Including PC+1 even though decode didn’t use it.

Pipeline datapath – decode stage

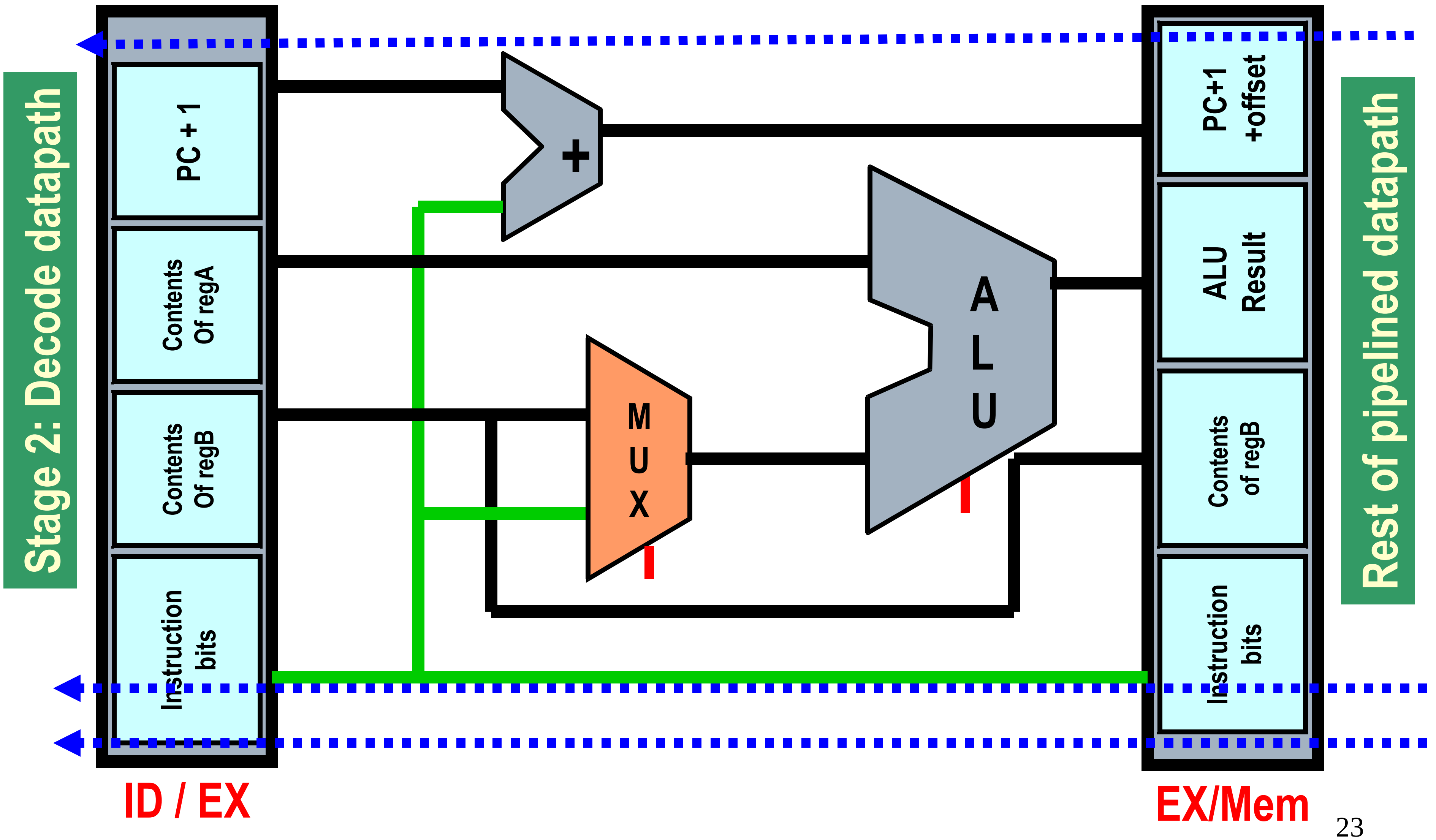


Stage 3: execute

- Design a datapath that performs the proper ALU operation for the instruction specified and the values present in the ID/EX pipeline register.
  - The inputs are the contents of regA and either the contents of regB or the offset field on the instruction.
  - Also, calculate PC+1+offset in case this is a branch.

- Write everything needed to complete execution to the pipeline register (EX/Mem).
  - ALU result, contents of regB and PC+1+offset.
  - Instruction bits for opcode and destReg specifiers.
  - Result from comparison of regA and regB contents.

Pipeline datapath – execute stage

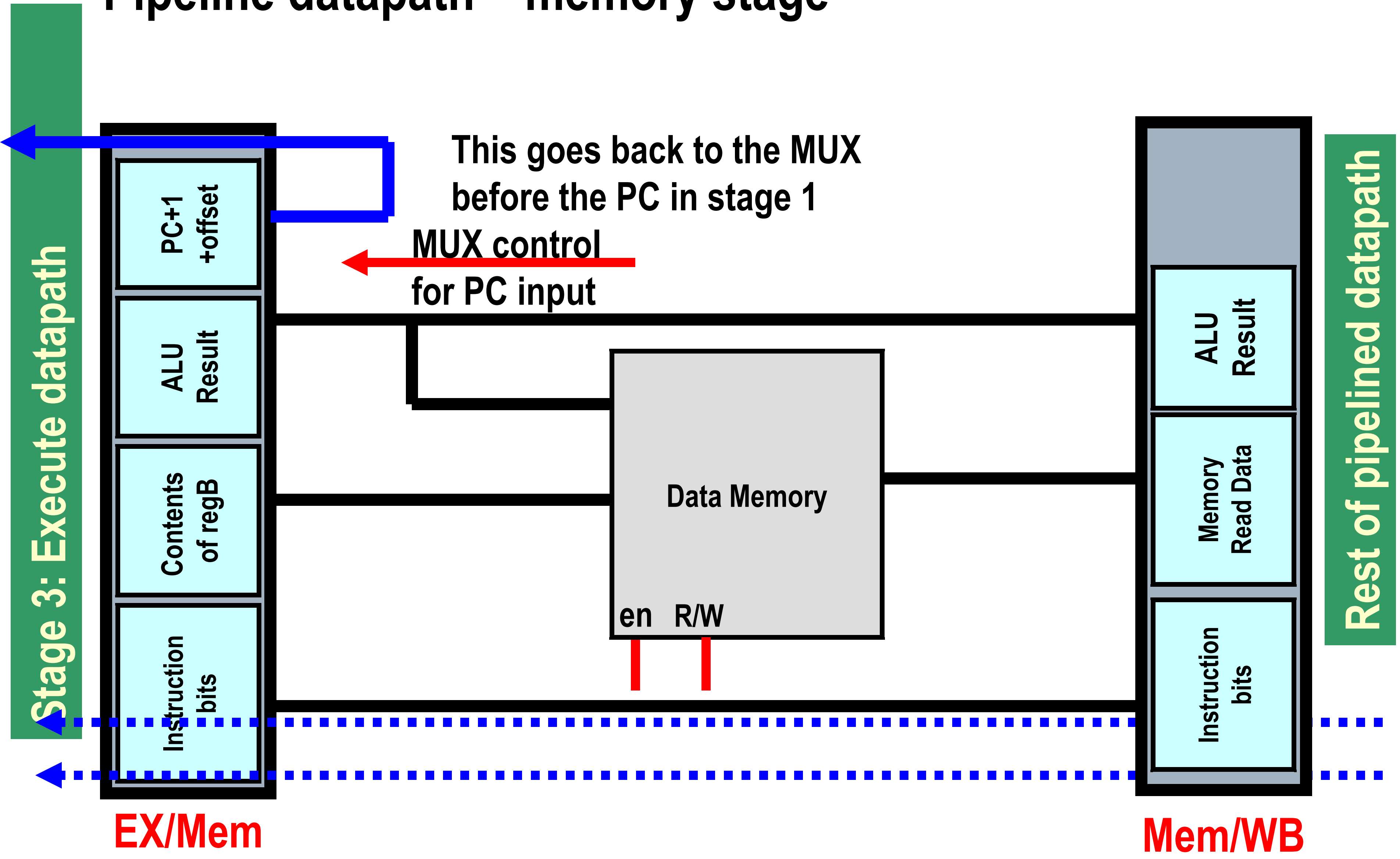


Stage 4: memory operation

- Design a datapath that performs the proper memory operation for the instruction specified and the values present in the EX/Mem pipeline register.
  - ALU result contains address for **ld** and **st** instructions.
  - Opcode bits control memory R/W and enable signals.

- Write everything needed to complete execution to the pipeline register (Mem/WB).
  - ALU result and MemData.
  - Instruction bits for opcode and destReg specifiers.

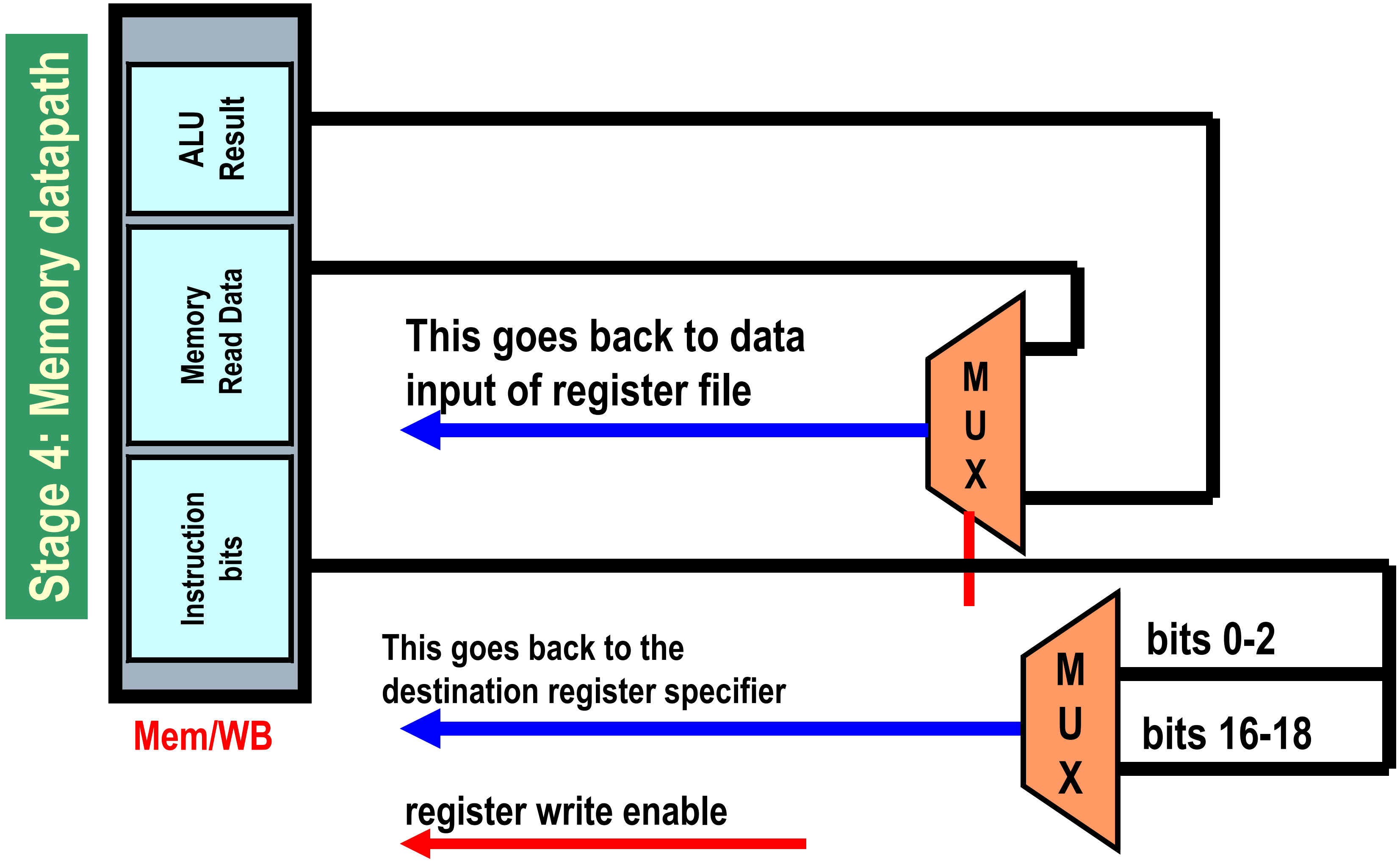
Pipeline datapath – memory stage



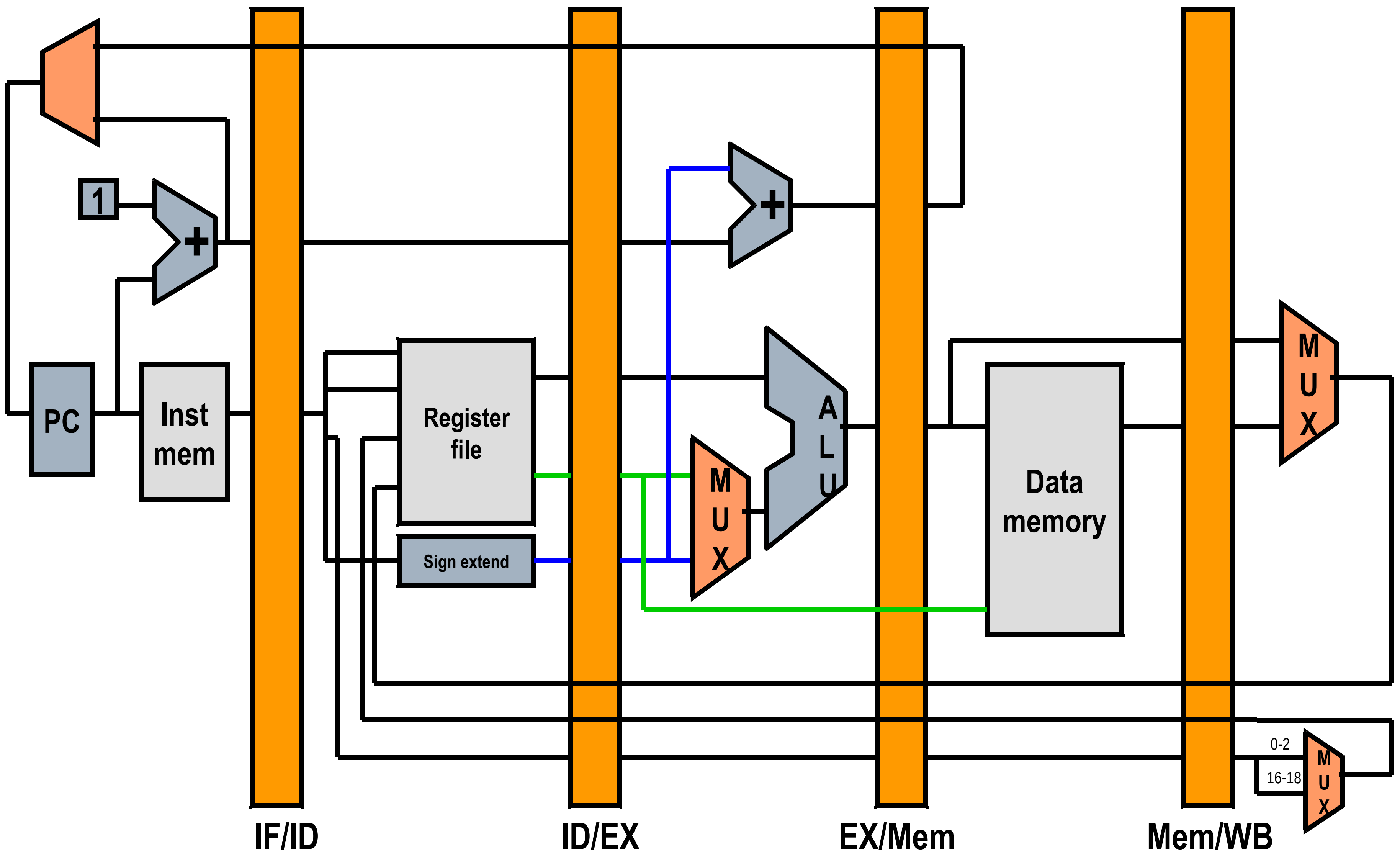
Stage 5: write back

- Design a datapath that completes the execution of this instruction, writing to the register file if required.
  - Write MemData to destReg for ld instruction.
  - Write ALU result to destReg for add or nand instructions.
  - Opcode bits also control register write enable signal.

Pipeline datapath – writeback stage



Interaction among stages

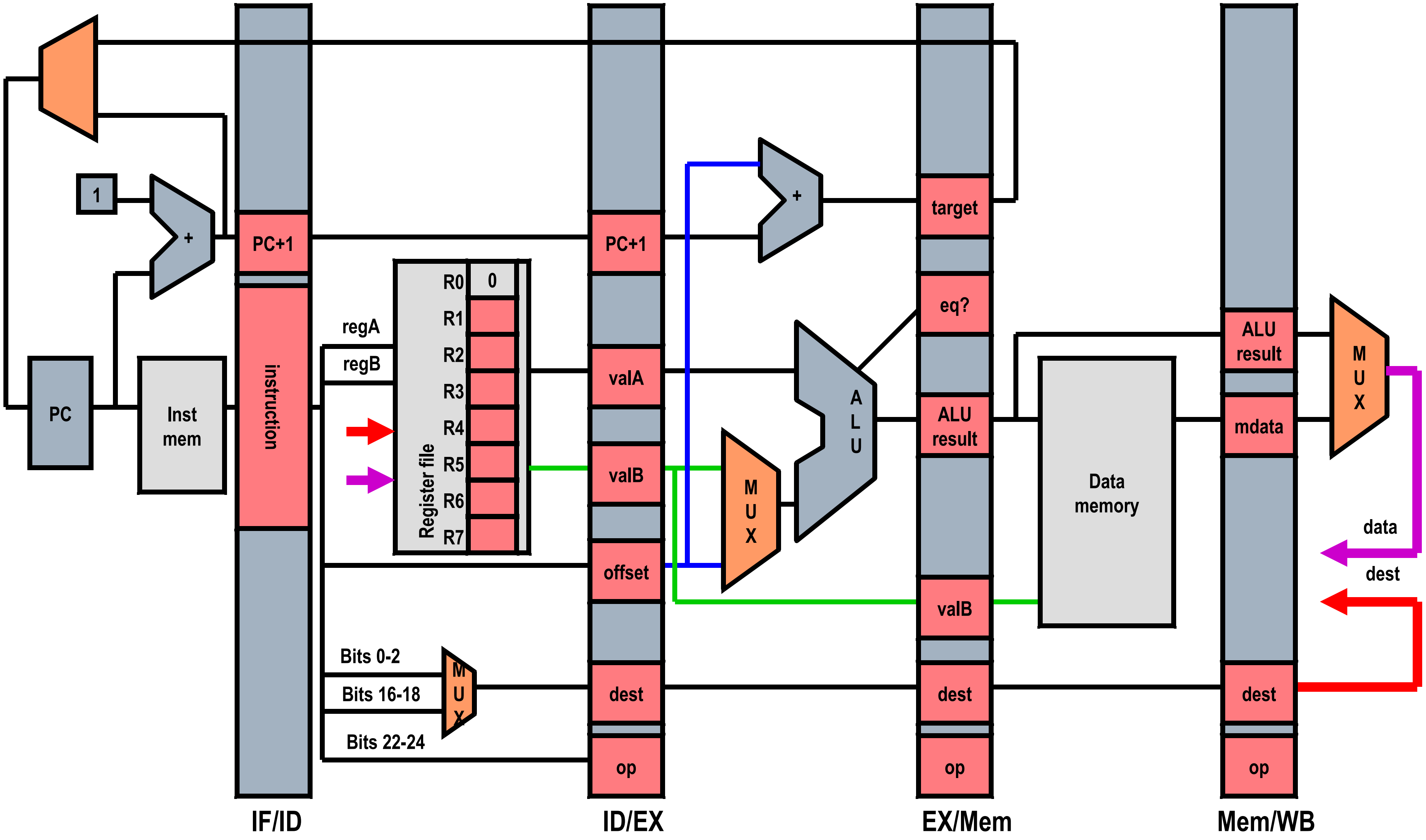


Sample code (simple)

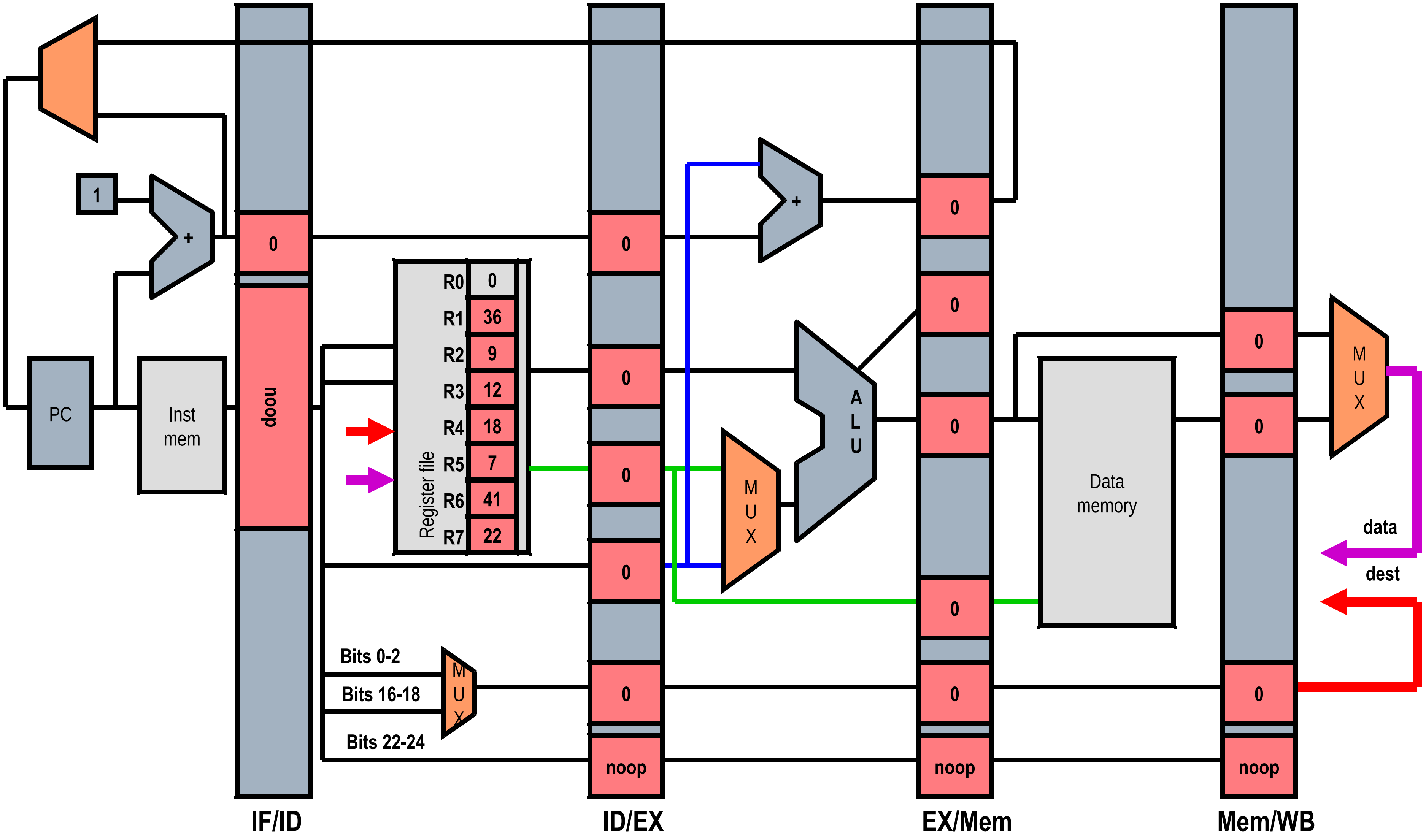
Let’s run the following code on pipelined LC2K2x

```
add   1  2  3    ;  reg 3 = reg 1 + reg 2
nand  4  5  6    ;  reg 6 = ~(reg 4 & reg 5)
lw    2  4  20   ;  reg 4 =  Mem[reg2+20]
add   2  5  5    ;  reg 5 = reg 2 + reg 5
sw    3  7  10   ;  Mem[reg3+10] = reg 7
```

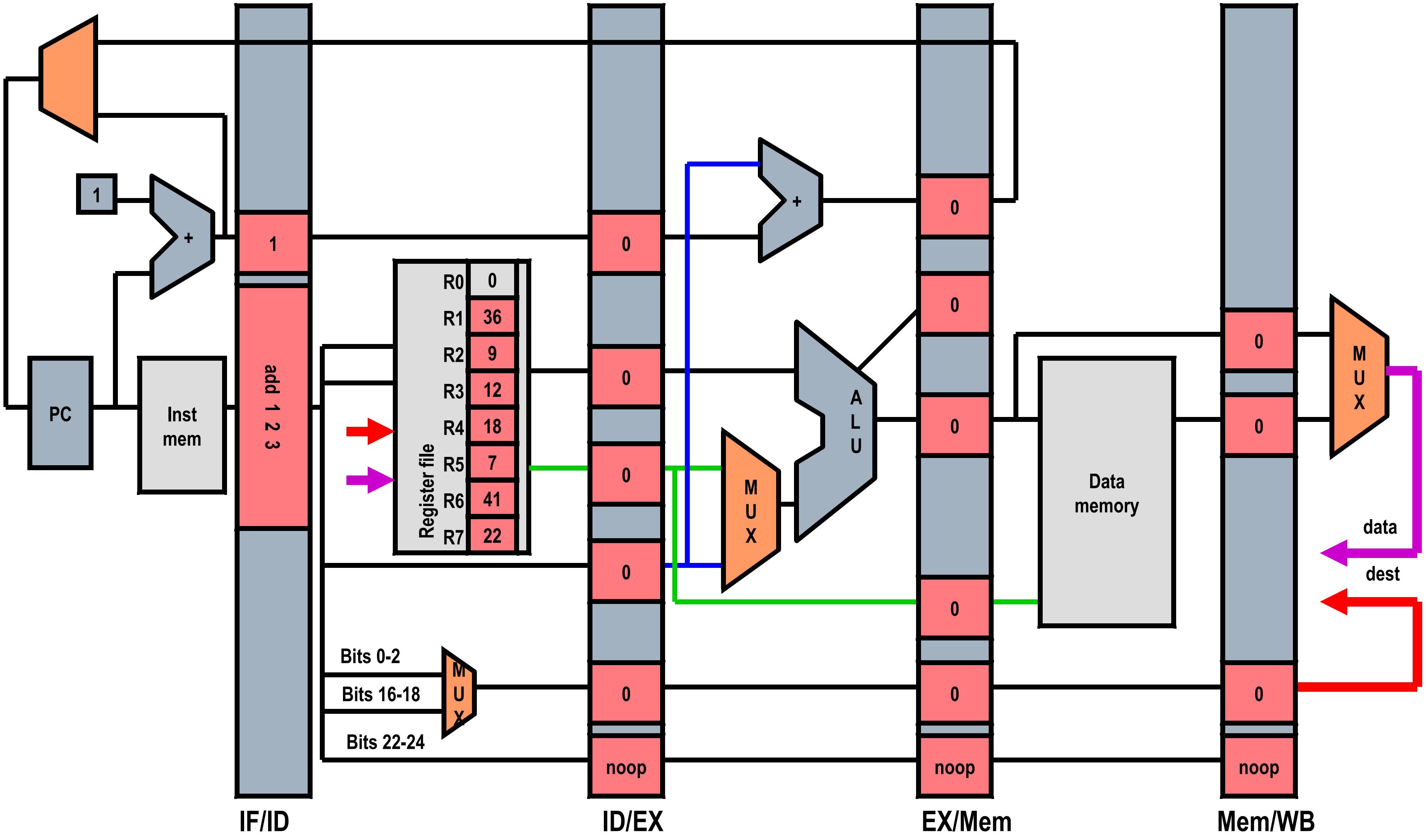
Pipelined datapath



Time 0 - initial state

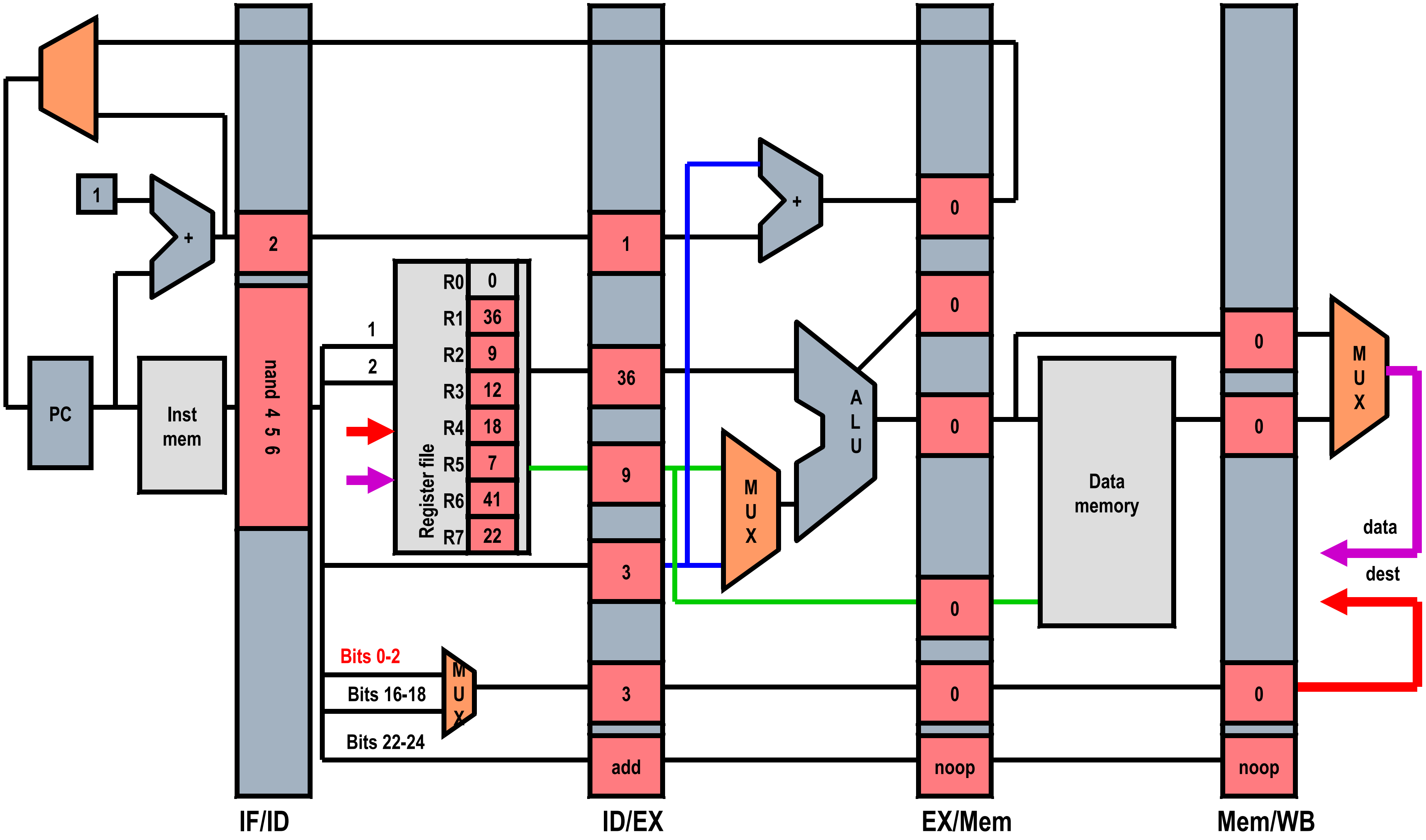


Time 1 - fetch: add 1 2 3



add 1 2 3

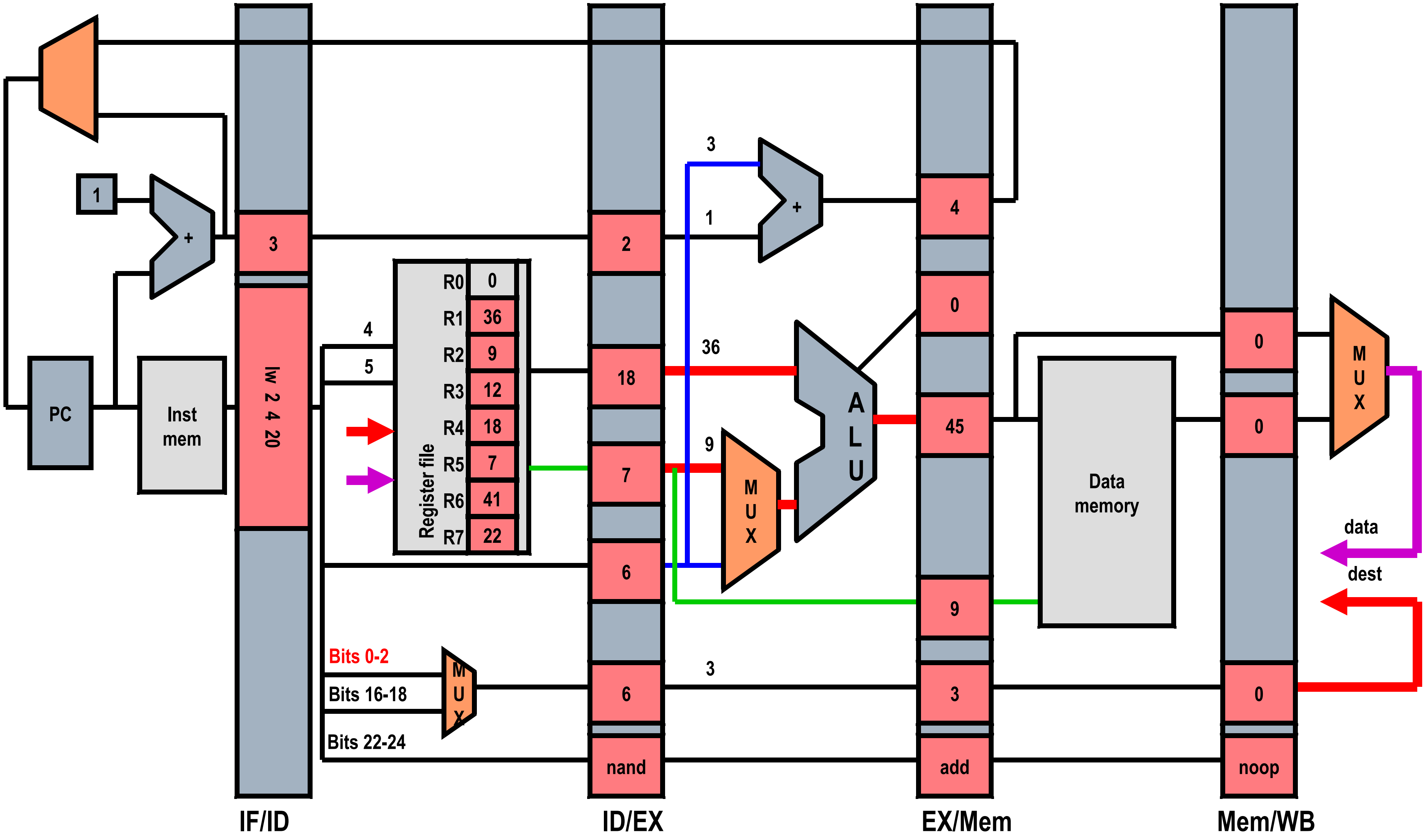
Time 2 - fetch: nand 4 5 6



nand 4 5 6

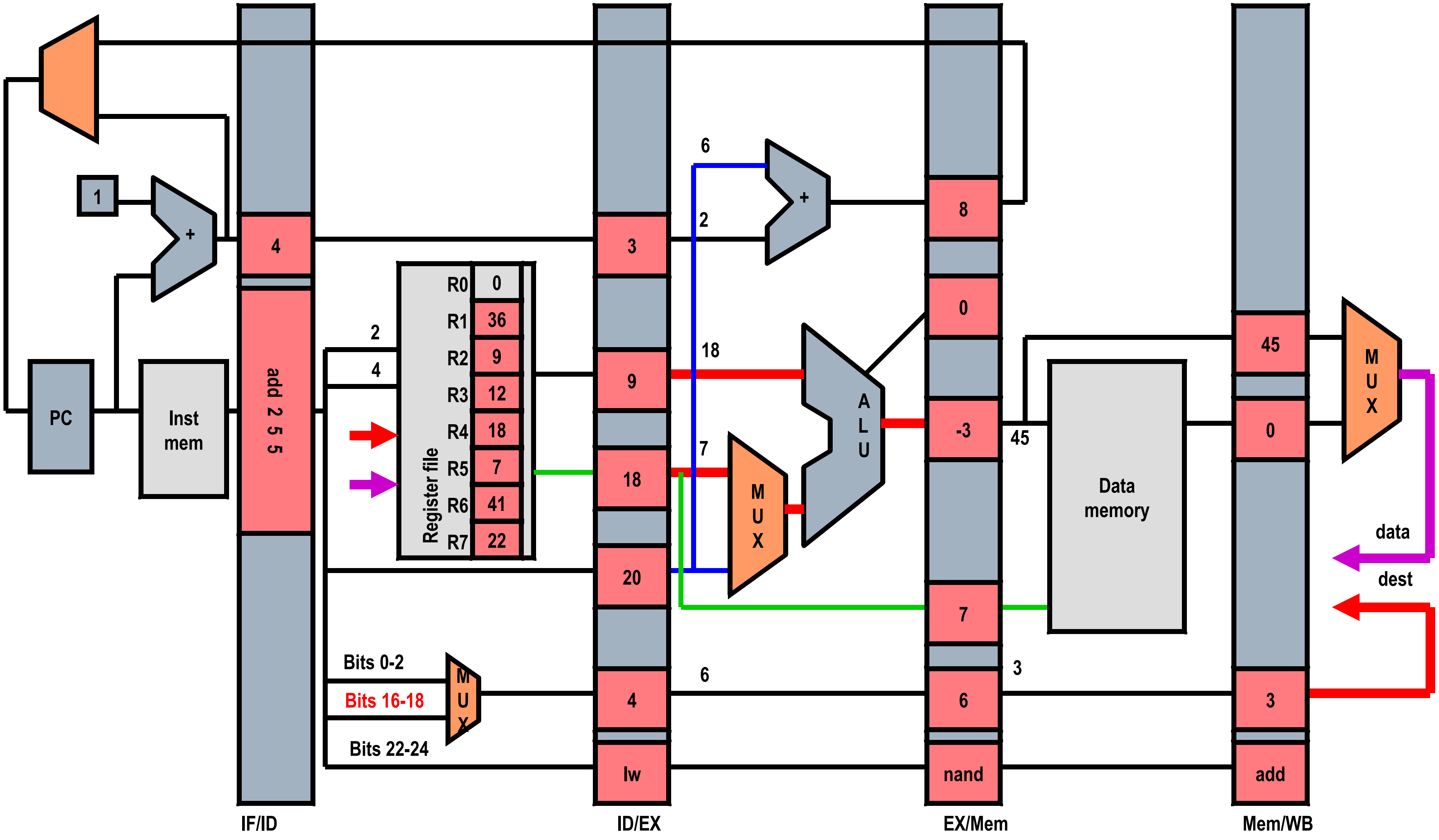
add 1 2 3

Time 3 - fetch: lw 2 4 20



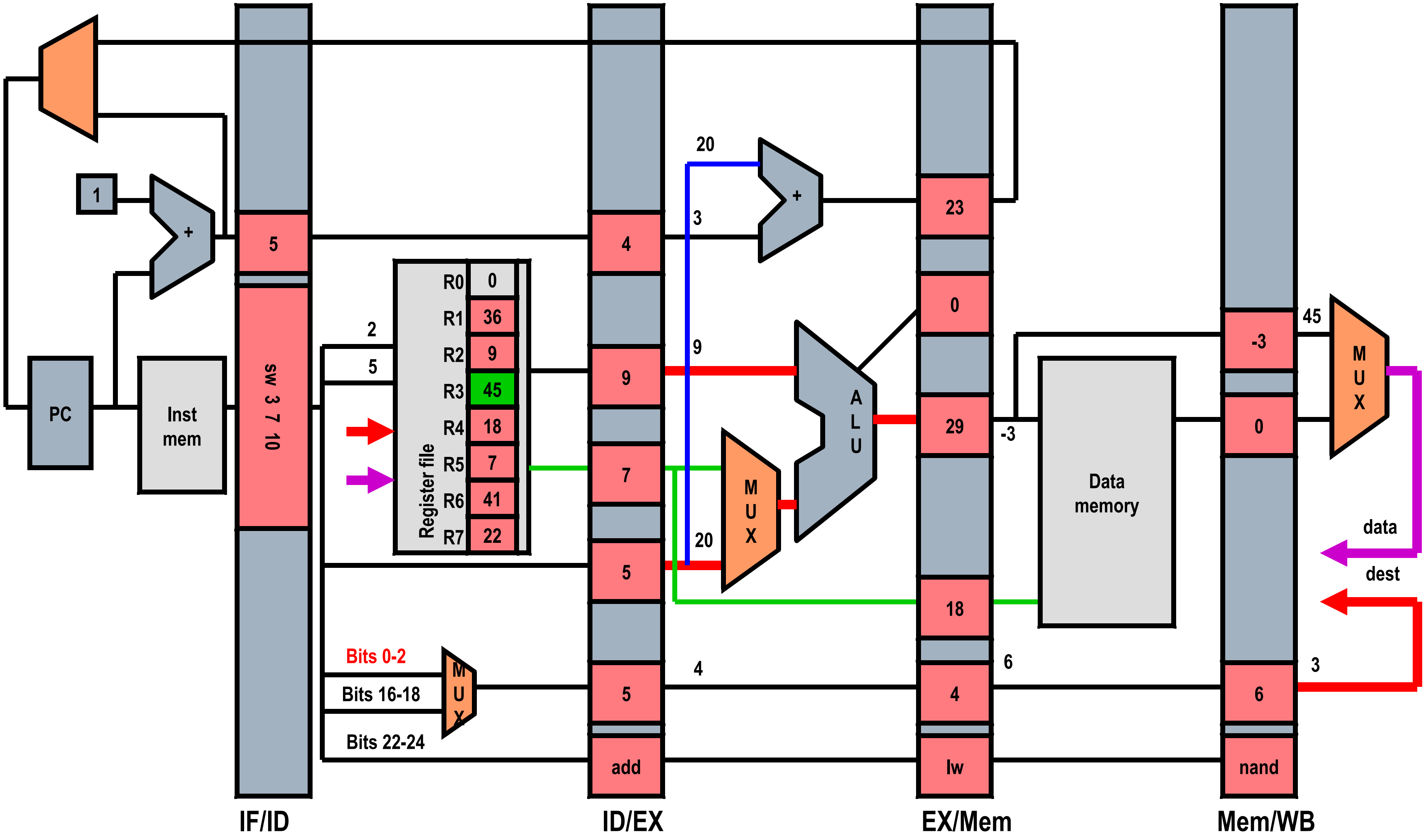
lw 2 4 20 nand 4 5 6 add 1 2 3

Time 4 - fetch: add 2 5 5



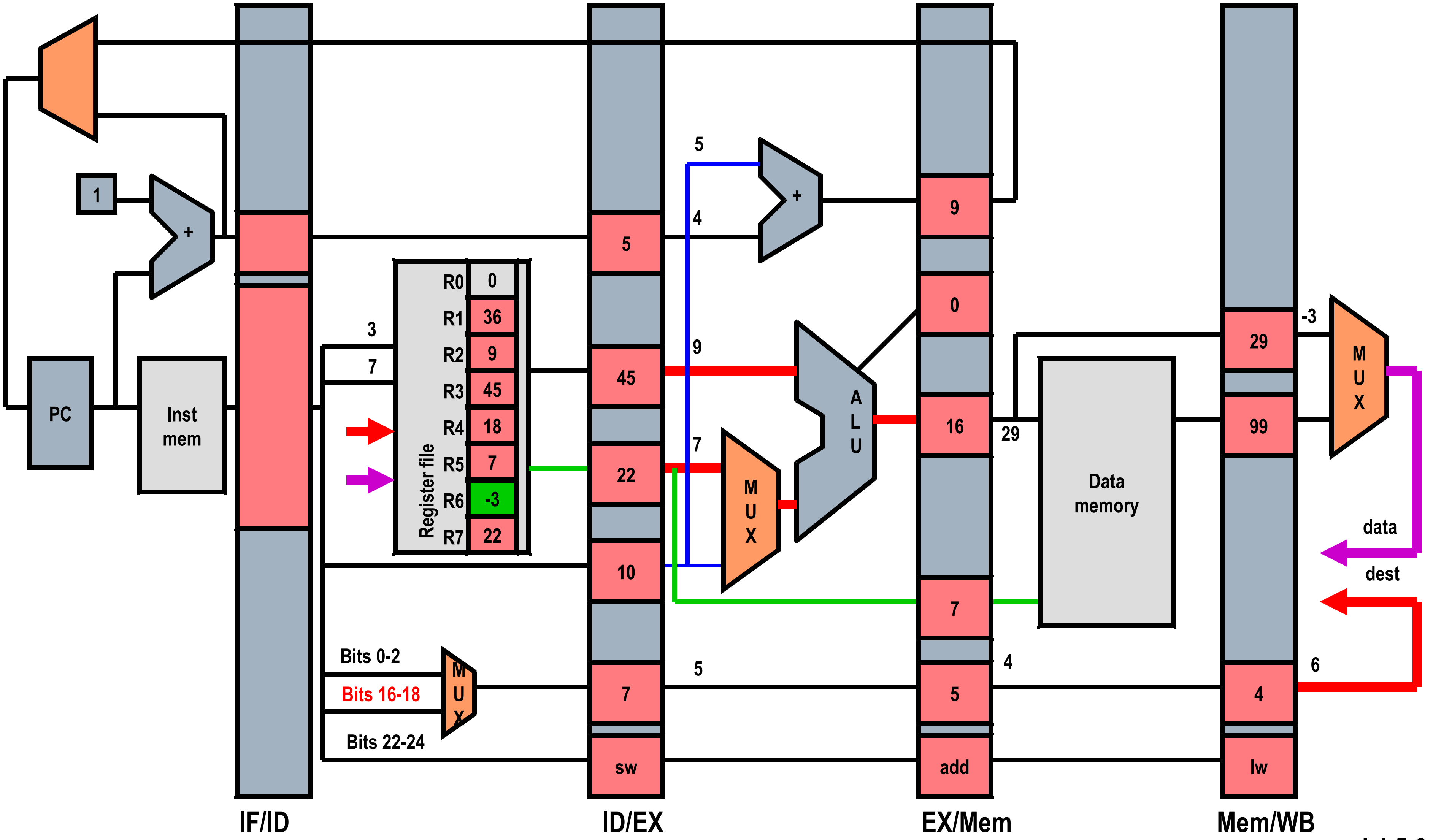
add 2 5 5 lw 2 4 20 nand 4 5 6 add 1 2 3

Time 5 - fetch: sw 3 7 10



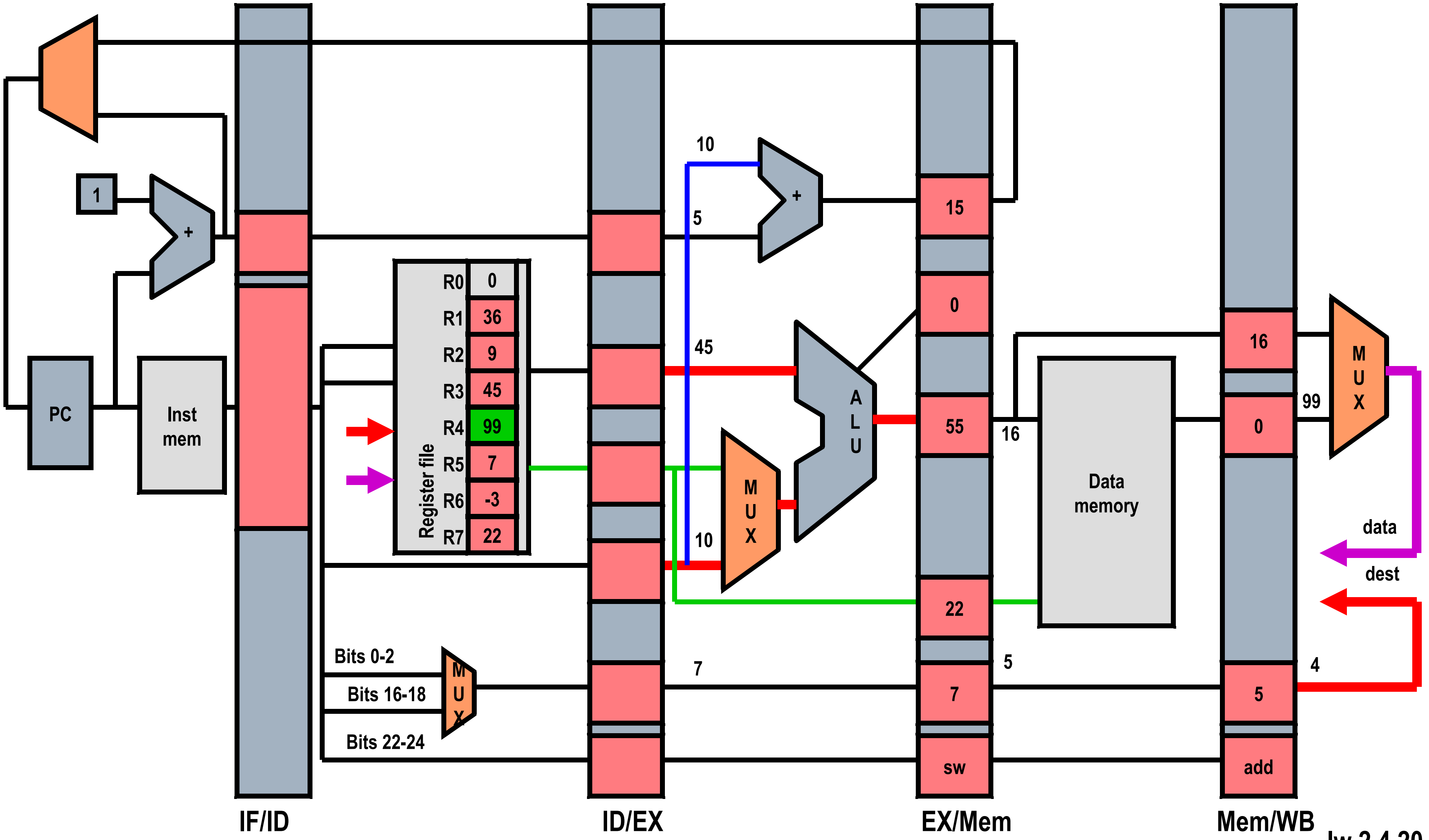
sw 3 7 10 add 2 5 5 lw 2 4 20 nand 4 5 6 add 1 2 3

Time 6 – no more instructions



sw 3 7 10 add 2 5 5 lw 2 4 20 nand 4 5 6

Time 7 – no more instructions

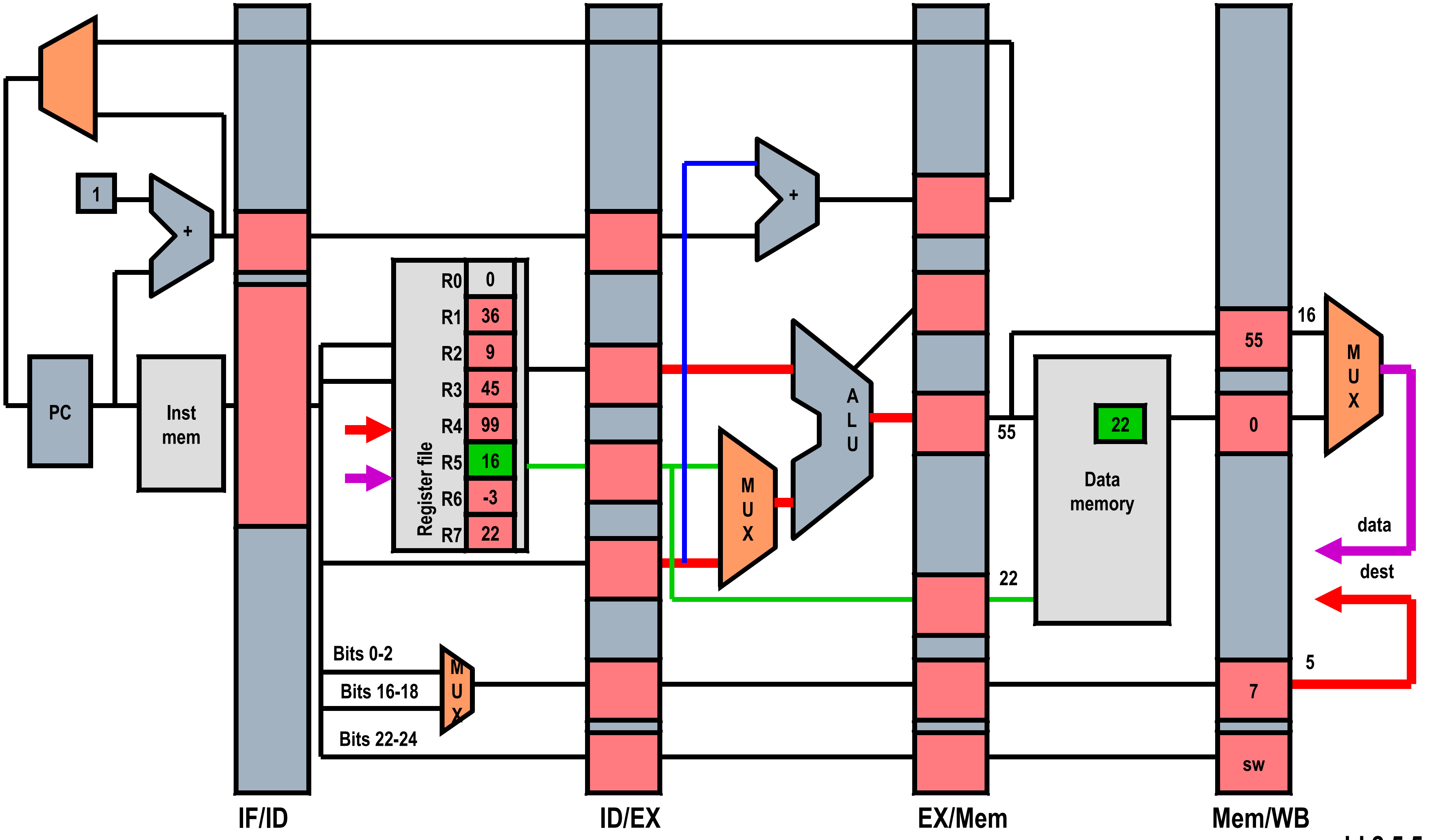


sw 3 7 10

add 2 5 5

lw 2 4 20

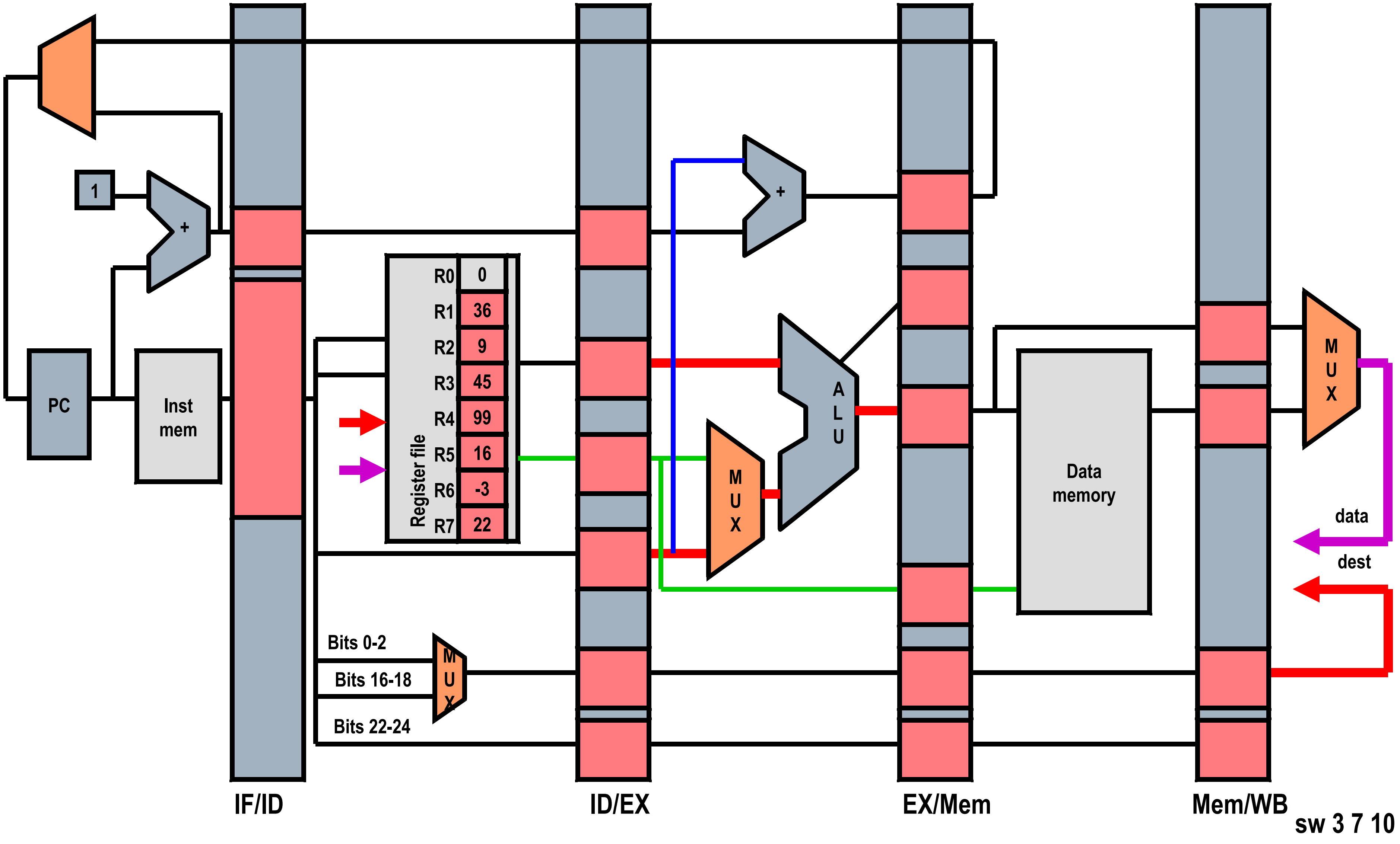
Time 8 – no more instructions



sw 3 7 10

add 2 5 5

Time 9 – no more instructions



Time graphs (pipeline trace)

Cycle:	1	2	3	4	5	6	7	8	9
add	Fetch	decode	execute	memory	writebk				
nand		Fetch	decode	execute	memory	writebk			
lw			Fetch	decode	execute	memory	writebk		
add				Fetch	decode	execute	memory	writebk	
sw					Fetch	decode	execute	memory	writebk

A vertical slice reports the entire activity of the pipeline at time 5.

What can go wrong?

Data hazards: since register reads occur in stage 2 and register writes occur in stage 5, it is possible to read the wrong value if it is about to be written.

Control hazards: A branch instruction may change the PC, but not until stage 4. What do we fetch before that?

Exceptions: How do you handle exceptions in a pipelined processor with 5 instructions in flight?

Next Lecture: data hazards.

Next time

- Hazards