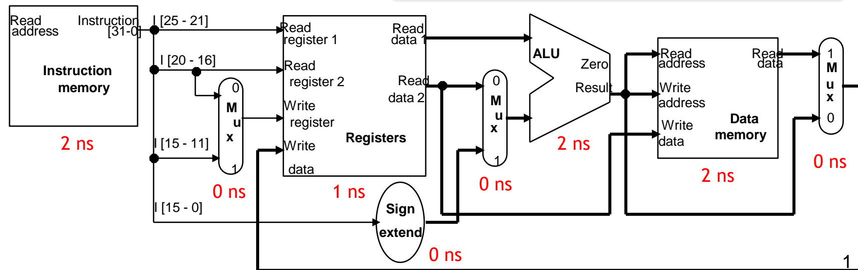
# Review

•Increment instructions like addi $t0, $t0, 1 have the potential to trash a register value in the single cycle design; why don't they?

• How long to lw $t0,-4($sp)?

| | |
|---|---|
| reading the instruction memory | ?ns |
| reading the base register $sp | ?ns |
| computing memory address $sp-4 | ?ns |
| reading the data memory | ?ns |
| storing data back to $t0 | ?ns |





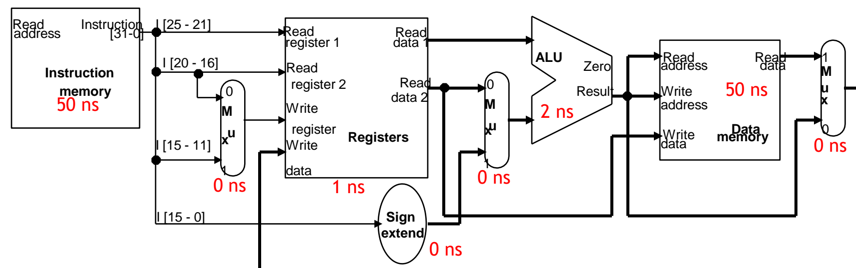

# It gets worse!

We've made very optimistic assumptions about memory latency:

Main memory accesses on modern machines is >50ns

For comparison, an ALU op on a typical machine ~0.3ns

Our worst case cycle (loads) includes 2 memory accesses: 50 + 1 + 2 + 50 + 1 = 104



## A multicycle approach

We've informally described instructions as executing in several steps

1. Instruction fetch and PC increment.
2. Reading sources from the register file.
3. Performing an ALU computation.
4. Reading or writing (data) memory.
5. Storing data back to the register file.

What if we made these stages *explicit* in the hardware design?

3

## Performance benefits

Each instruction can execute only the stages that are necessary.

- Arithmetic
- Load
- Store
- Branches

Proposed execution stages

1. Instruction fetch and PC increment
2. Reading sources from the register file
3. Performing an ALU computation
4. Reading or writing (data) memory
5. Storing data back to the register file

This would mean that instructions complete as soon as possible, instead of being limited by the slowest instruction

4

## The clock cycle

Things are simpler if we assume that each stage takes one clock cycle

- Instructions therefore require multiple clock cycles to execute
- But since each stage is fairly simple, the cycle time can be low

For the proposed execution stages below and the sample datapath delays shown earlier, each stage needs 2ns at most

- This accounts for the slowest devices, the ALU & data memory
- A 2ns clock cycle time corresponds to a 500MHz clock rate!

Proposed execution stages

1. Instruction fetch and PC increment
2. Reading sources from the register file
3. Performing an ALU computation
4. Reading or writing (data) memory
5. Storing data back to the register file

more than 50 times faster than single cycle

5

## Multicycle Design

- Consider the changes required in our design to turn it into a multicycle version.
- Topics
  - Hardware savings – adders
  - Restructuring design – adding muxes
  - Unifying the I- and D-memories – IorD mux
  - Carrying results over cycles – temporary registers
  - Register Write Signals
  - Final Design

7

## Cost Benefits Of Multi-cycle Design

As an added bonus, we can eliminate some of the hardware from the single-cycle datapath

- We will restrict ourselves to using each functional unit once per cycle, just like before
- But since instructions require multiple cycles, we could reuse some units in a *different* cycle during the execution of a single instruction

For example, we could use the same ALU:

- to increment the PC (first clock cycle), and
- for arithmetic operations (third clock cycle)

Proposed execution stages

1. Instruction fetch and PC increment
2. Reading sources from the register file
3. Performing an ALU computation
4. Reading or writing (data) memory
5. Storing data back to the register file

8

# Two extra adders

Our original single-cycle datapath had an ALU and two adders.

The arithmetic-logic unit had two responsibilities.

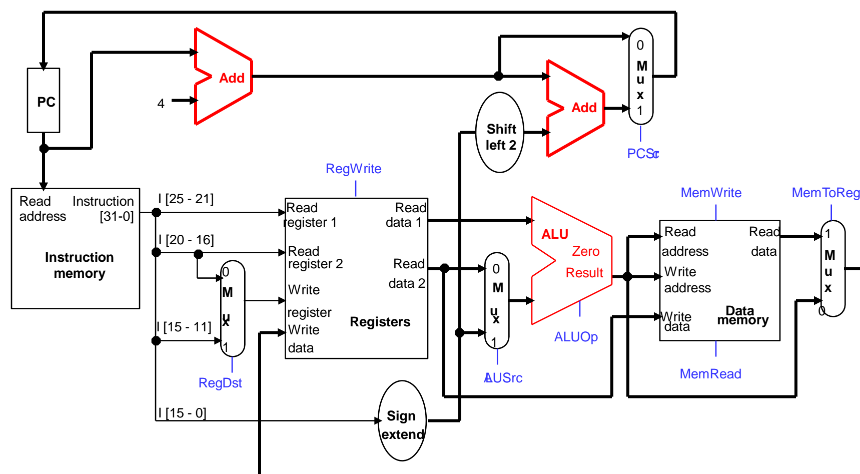
- Doing an operation on two registers for arithmetic instructions.
- Adding a register to a sign-extended constant, to compute effective addresses for lw and sw instructions.

One of the extra adders incremented the PC by computing PC + 4.

The other adder computed branch targets, by adding a sign-extended, shifted offset to (PC + 4).

9

# The extra adders of single-cycle




10

# Our new adder setup

We can eliminate *both* extra adders in a multicycle datapath, and use just one ALU, with suitable muxes

A 2-to-1 mux ALUSrcA sets the first ALU input to be the PC or a register

A 4-to-1 mux ALUSrcB selects the second ALU input :

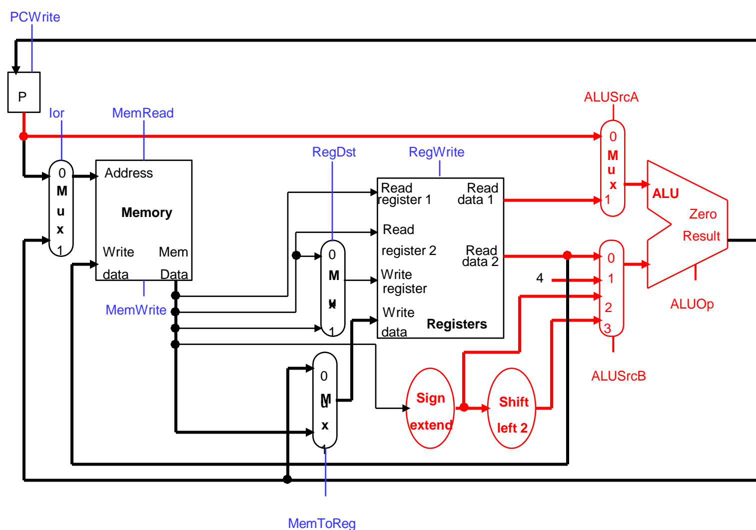
- the register file (for arithmetic operations),
- a constant 4 (to increment the PC),
- a sign-extended constant (for effective addresses), and
- a sign-extended and shifted constant (for branch targets).

This permits a single ALU to perform all of the necessary functions.

- Arithmetic operations on two register operands
- Incrementing the PC
- Computing effective addresses for lw and sw
- Adding a sign-extended, shifted offset to (PC + 4) for branches

11

# The multicycle adder setup highlighted




12

## Eliminating a memory

Similarly, we can get by with one unified memory, which will store *both* program instructions *and* data

This memory is used in the instruction fetch and data access stages, and an address could come from:

- the PC register (when we're fetching an instruction), or
- the ALU output (for the effective address of a lw or sw)

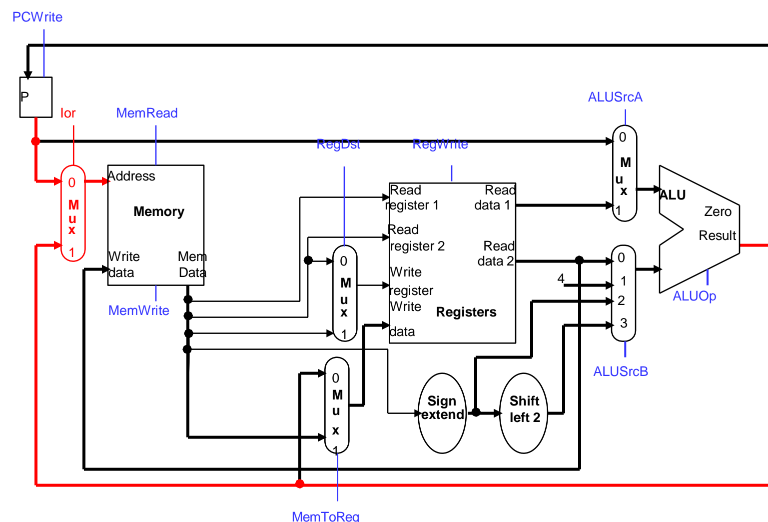
We add another 2-to-1 mux, IorD, to decide whether the memory is being accessed for instructions or data

Proposed execution stages

1. Instruction fetch and PC increment
2. Reading sources from the register file
3. Performing an ALU computation
4. Reading or writing (data) memory
5. Storing data back to the register file

13

## The new memory setup highlighted




14

# Intermediate registers

Sometimes we need the output of a functional unit in a later clock cycle during one instruction execution

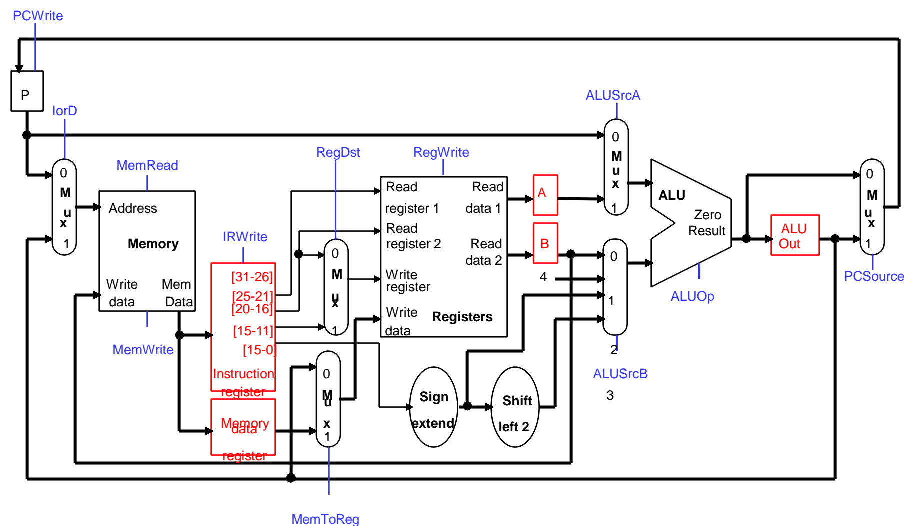
- The instruction word fetched in stage 1 determines the destination of the register write in stage 5
- The ALU result for an address computation in stage 3 is needed as the memory address for lw or sw in stage 4

These outputs will have to be stored in intermediate registers for future use. Otherwise they would probably be lost by the next clock cycle

- The instruction read in stage 1 is saved in instruction register
- Register file outputs from stage 2 are saved in registers A and B
- The ALU output will be stored in a register ALUOut
- Any data fetched from memory in stage 4 is kept in the memory data register, also called MDR



# The final multicycle datapath



## Register write control signals

We must add a few more control signals to the datapath

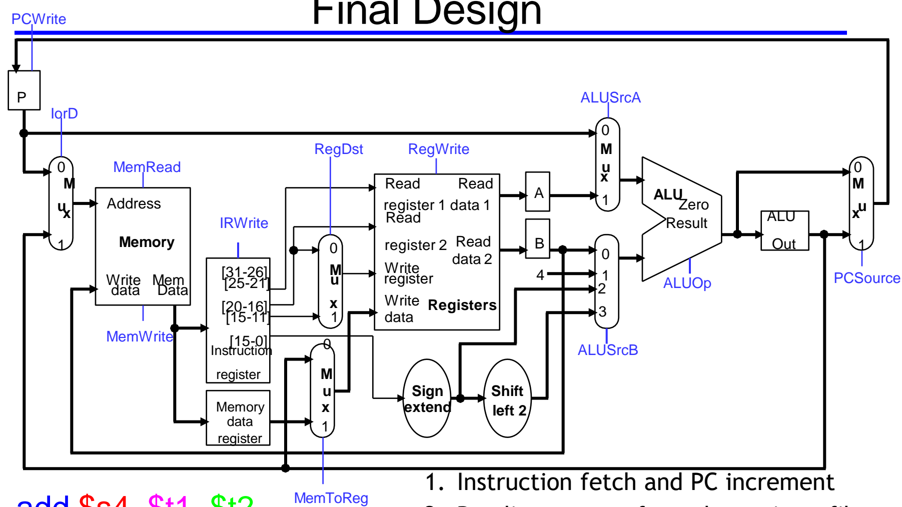
Since instructions now take a variable number of cycles to execute, we cannot update the PC on each cycle

- Instead, a PCWrite signal controls the loading of the PC
- The instruction register also has a write signal, IRWrite. We need to keep the instruction word for the duration of its execution, and must explicitly re-load the instruction register when needed

The other intermediate registers, MDR, A, B and ALUOut, will store data for only one clock cycle at most, and do not need write control signals

17

## Final Design



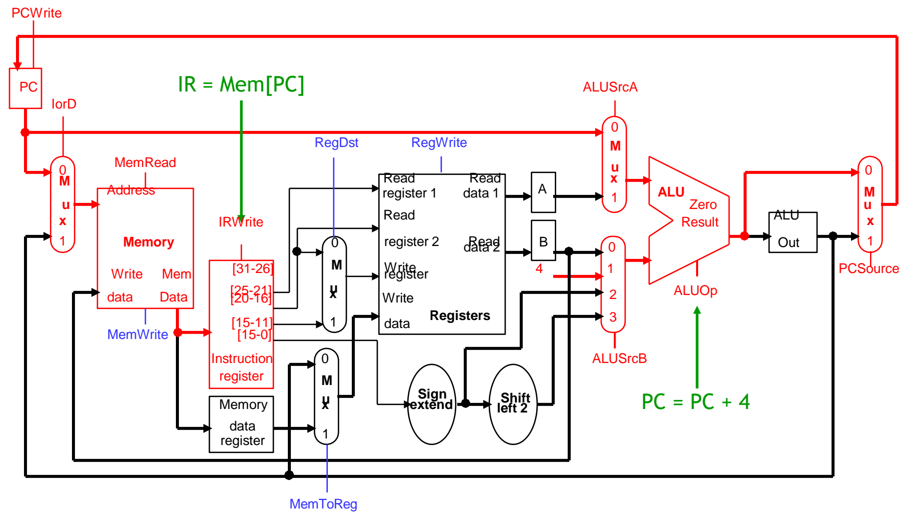

add $s4, $t1, $t2

lw $t0, -4($sp)

beq $at, $0, offset

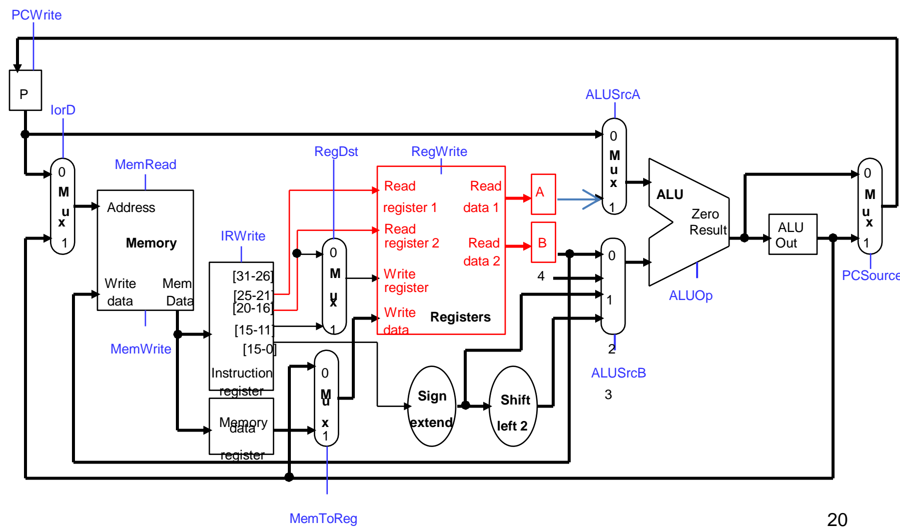
1. Instruction fetch and PC increment
2. Reading sources from the register file
3. Performing an ALU computation
4. Reading or writing (data) memory
5. Storing data back to the register file

# Stage 1: Instruction fetch & PC increment



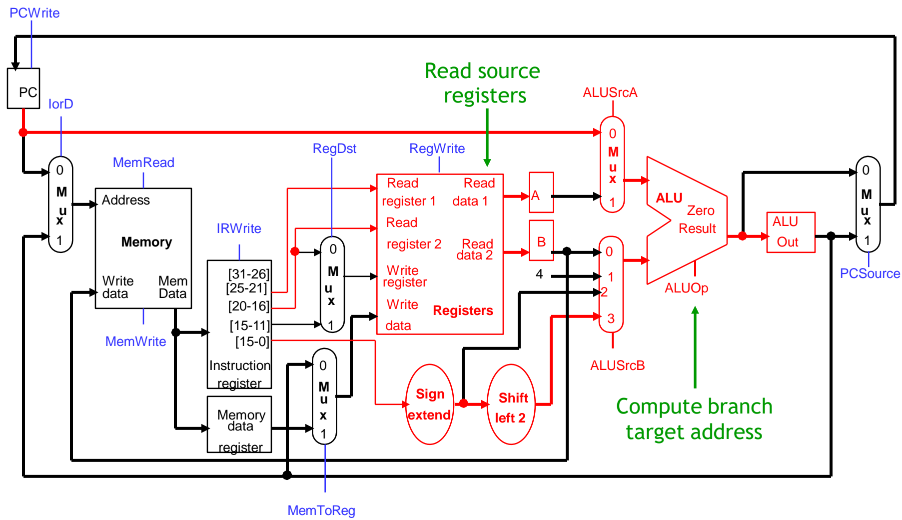

19

# Register File Read



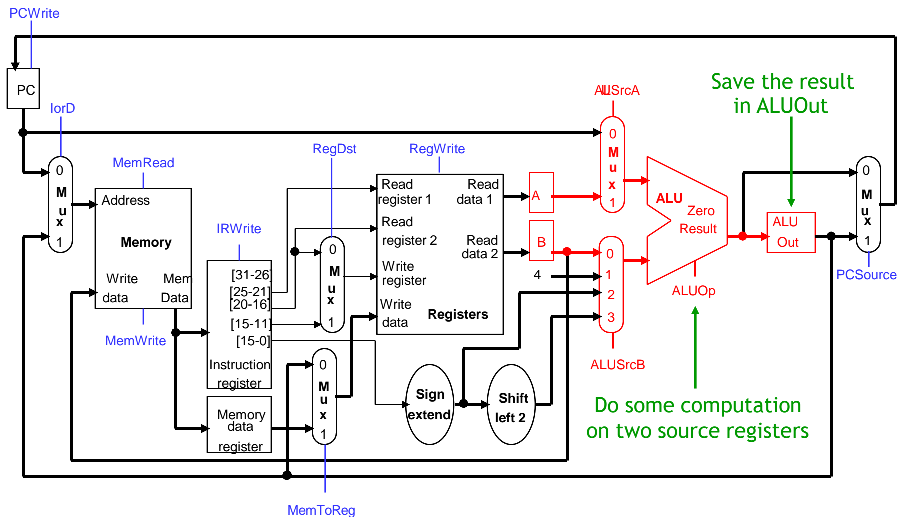

20

## Stage 2: Reg fetch & branch target

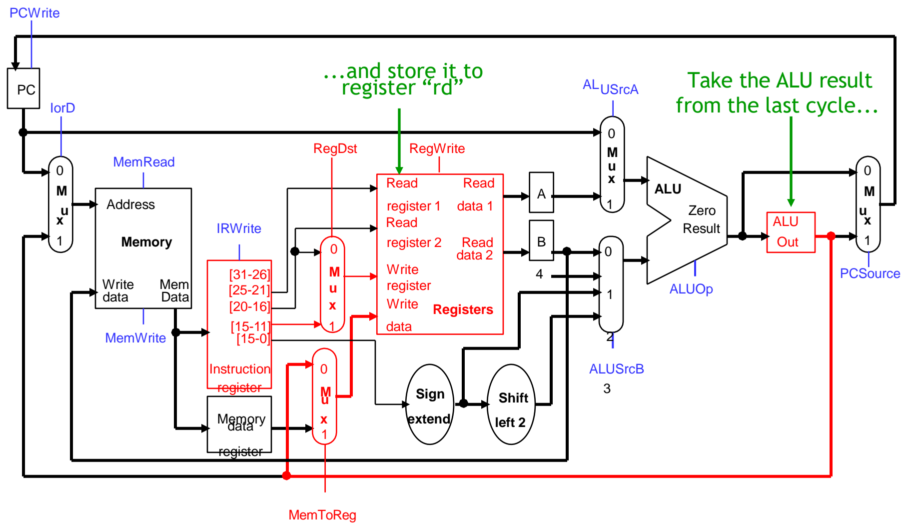


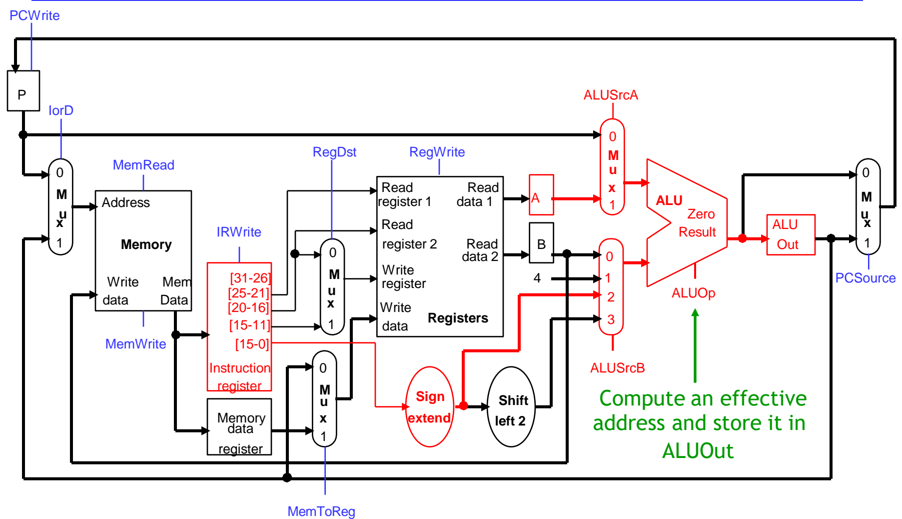

21

## Stage 3 (R-type): instruction execution

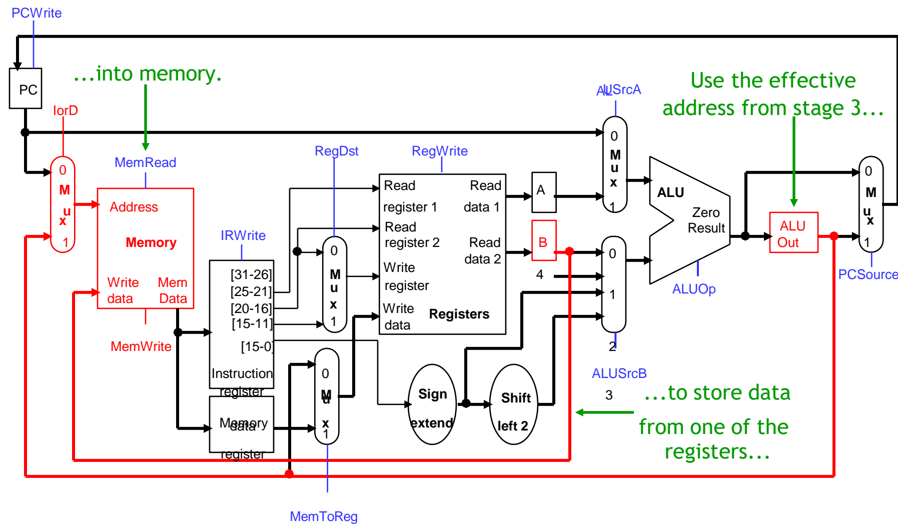



22


Stage 4 (R-type): write back
PCWrite
PC
IorD
MemRead
Address
Memory
Write data
Mem Data
MemWrite
IRWrite
[31-26]
[25-21]
[20-16]
[15-11]
[15-0]
Instruction register
Memory data register
RegDst
RegWrite
...and store it to register "rd"
Read register 1
Read data 1
Read register 2
Read data 2
Write register
Write data
Registers
A
B
Sign extend
Shift left 2
MemToReg
ALUSrcA
ALU
Zero
Result
ALUOp
ALUSrcB
ALU Out
Take the ALU result from the last cycle...
PCSource
23

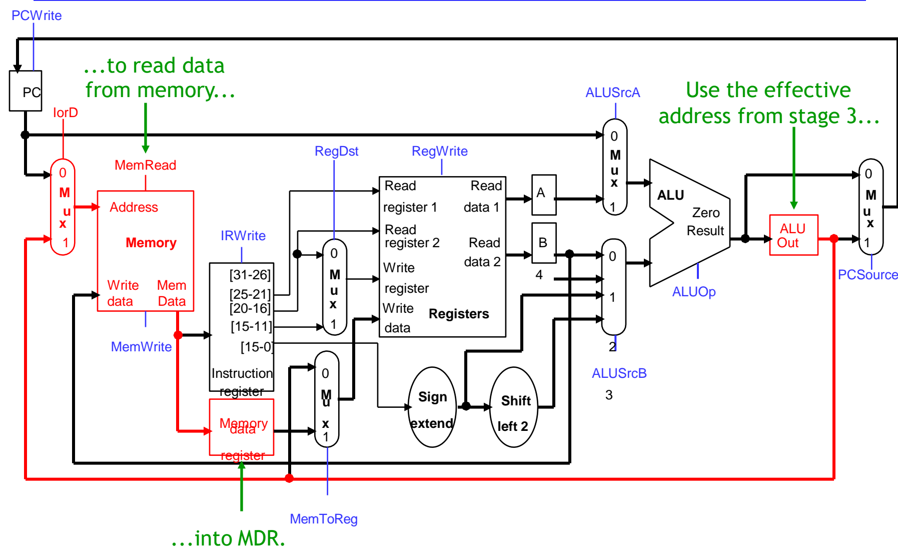

Stage 3 (sw): compute effective address
PCWrite
IorD
MemRead
Address
Memory
Write data
Mem Data
MemWrite
IRWrite
[31-26]
[25-21]
[20-16]
[15-11]
[15-0]
Instruction register
Memory data register
RegDst
RegWrite
Read register 1
Read data 1
Read register 2
Read data 2
Write register
Write data
Registers
A
B
Sign extend
Shift left 2
MemToReg
ALUSrcA
ALU
Zero
Result
ALUOp
ALUSrcB
ALU Out
PCSource
Compute an effective address and store it in ALUOut
24

# Stage 4 (sw): memory write



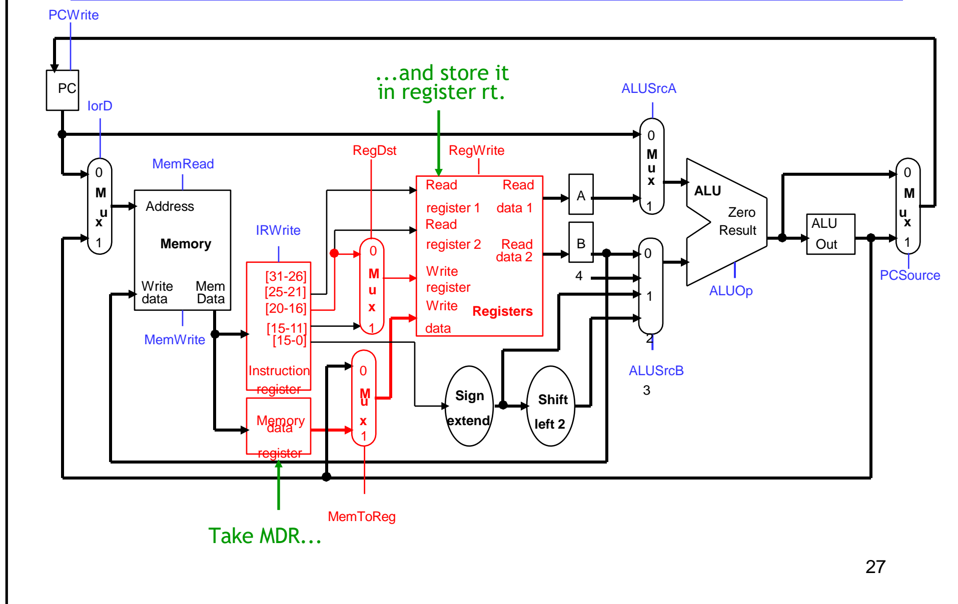

25

# Stage 4 (lw): memory read




26

## Stage 5 (lw): register write




## Summary

A single-cycle CPU has two main disadvantages

- The cycle time is limited by the worst case latency
- It requires more hardware than necessary

A multicycle processor splits instruction execution into several stages

- Instructions only execute as many stages as required
- Each stage is relatively simple, so the clock cycle time is reduced
- Functional units can be reused on different cycles

We made several modifications to the single-cycle datapath

- The two extra adders and one memory were removed
- Multiplexers were inserted so the ALU and memory can be used for different purposes in different execution stages
- New registers are needed to store intermediate results

28