CSSE232
Computer Architecture I

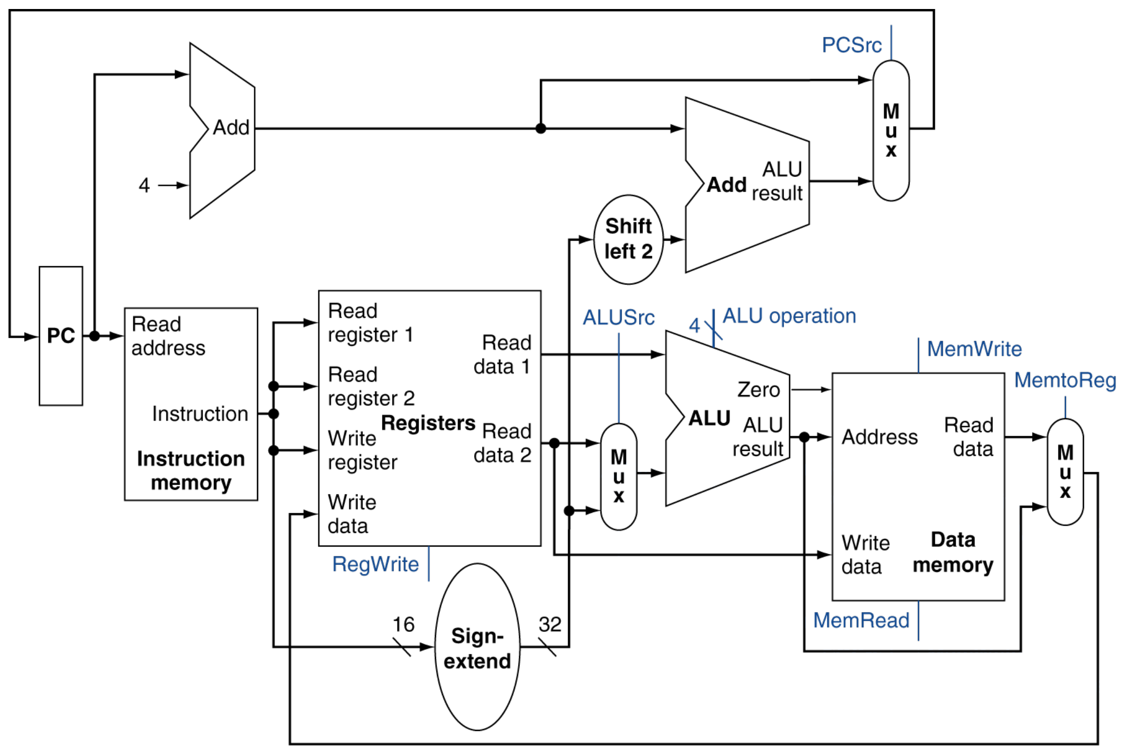
Multicycle Datapath

Class Status

- Next 3 days : Multicycle datapath
  - Reading
    - Multicycle datapath is not in the book!

How long do instructions take?

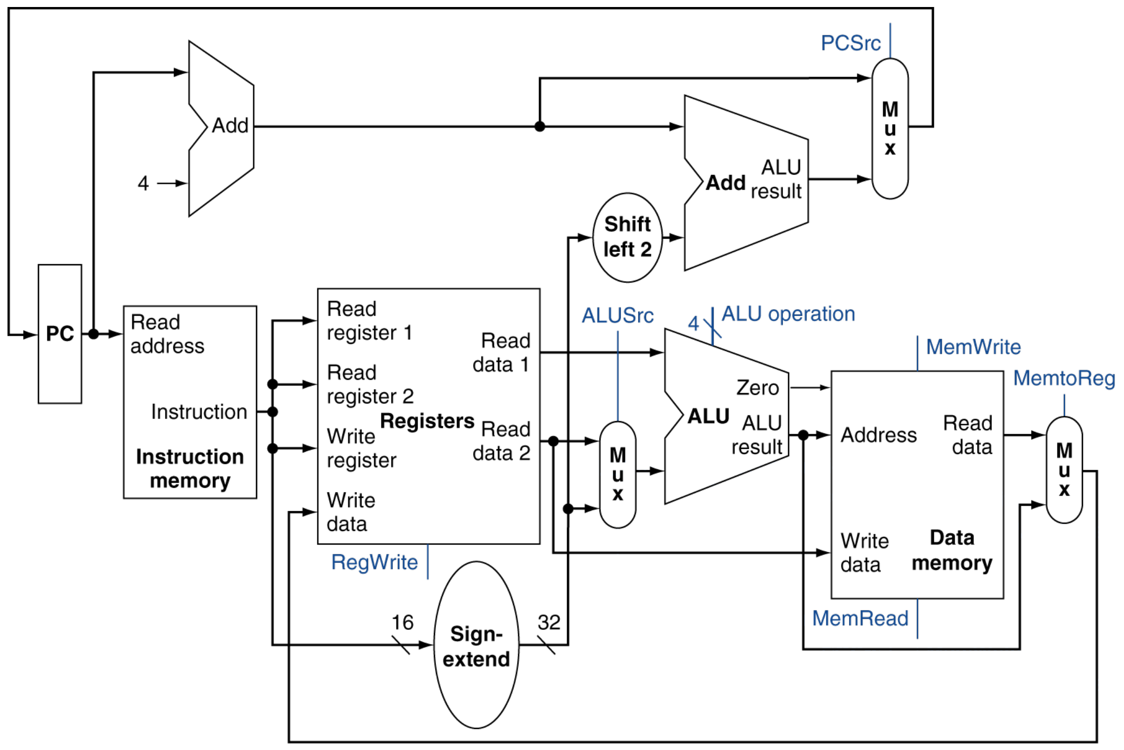
- ALU 2ns
- Mem 2ns
- Reg File 1ns
- Everything else is free!



How long do instructions take?

- ALU 2ns
- Mem 2ns
- Reg File 1ns
- Everything else is free!

- R-type? Branch? Store? Load?



Outline

- Problems with single-cycle
- Steps
  - IF, ID, EXEC, MEM, WB
- RTL (Register Transfer Language)
  - Describe processor actions

Single Cycle Design Problems

- What are they?

Single Cycle Design Problems

- Fixed period clock
  - Every instruction takes one clock cycle
  - All cycles same length
  - Cycle time set by longest instruction path (lw)
    - Some instructions *could* run faster
    - What if we have very long instructions? (floating point)
- Many adders
  - Duplicate hardware is a waste?

Improving Single Cycle

- Reduce hardware use?

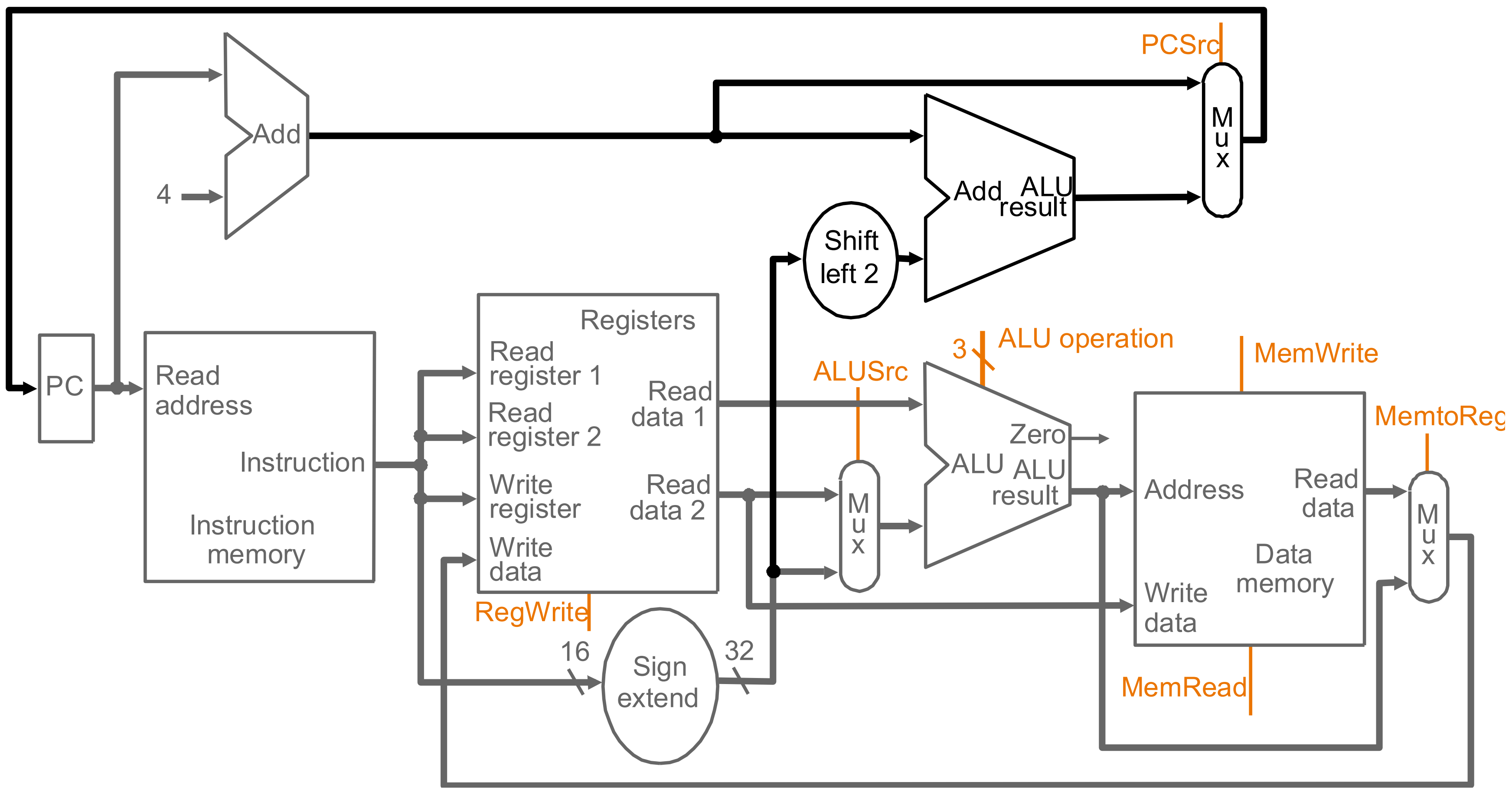
- Reduce time use?

Improving Single Cycle

- Reduce hardware use?
  - Very hard, all parts are needed
- Reduce time use?
  - Variable length clock
    - Longer for lw, shorter for branch
    - Has been done, hard to do in practice
  - Shorter clock cycles
    - Break work into small steps
    - Continue instruction if not finished after step

Multicycle

- Break instructions into steps
  - Each step takes one cycle
  - Each step needs approximately the same time
  - Each step uses one piece of hardware
- Save instruction data between steps
  - Save partial work at end of cycle
  - Next cycle, continue instruction



ALU 2ns Mem 2ns Reg 1ns

Breaking instructions into steps

- Our goal is to break up the instructions into *steps* so that
  - each step takes one clock cycle
  - the amount of work to be done in each step/cycle is about equal
  - each cycle uses at most once each major functional unit so that such units do not have to be replicated
  - functional units can be shared between different cycles within one instruction
- Data at end of one cycle to be used in next *must be stored* !!

Breaking instructions into steps

- We break instructions into the following *potential* execution steps – not all instructions require all the steps – each step takes one clock cycle
    1. Instruction fetch and PC increment (**IF**)
    2. Instruction decode and register fetch (**ID**)
    3. Execution, memory address computation, or branch completion (**EX**)
    4. Memory access or R-type instruction completion (**MEM**)
    5. Memory read completion (**WB**)

- Each MIPS instruction takes from 3 – 5 cycles (steps)

Step 1: Instruction Fetch & PC Increment (**IF**)

- Use PC to get instruction and put it in the instruction register.
  Increment the PC by 4 and put the result back in the PC.

- Can be described succinctly using *RTL* (*Register-Transfer Language*):

```
IR = Memory[PC];
 PC = PC + 4;
```

Step 2: Instruction Decode and Register Fetch (**ID**)

- Read registers rs and rt in case we need them.
  Compute the branch address in case the instruction is a branch.

- RTL:

```
A = Reg[IR[25-21]];
B = Reg[IR[20-16]];
ALUOut = PC + (sign-extend(IR[15-0]) << 2);
```

Step 3: Execution, Address Computation or Branch Completion (**EX**)

- ALU performs one of four functions *<u>depending</u>* on instruction type
  - memory reference:
    `ALUOut = A + sign-extend(IR[15-0]);`
  - R-type:
    `ALUOut = A op B;`
  - branch (instruction *completes*):
    `if (A==B) PC = ALUOut;`
  - jump (instruction *completes*):
    `PC = PC[31-28] || (IR(25-0) << 2)`

Note that the PC is written twice!!

Step 4: Memory access or R-type Instruction Completion (**MEM**)

- Again *depending* on instruction type:
- Loads and stores access memory
  - load

    `MDR = Memory[ALUOut];`
  - store (instruction *completes*)

    `Memory[ALUOut] = B;`

- R-type (instructions *completes*)

  `Reg[IR[15-11]] = ALUOut;`

Step 5: Memory Read Completion (**WB**)

- Again <u>*depending*</u> on instruction type:
- Load writes back (instruction *completes*)

```
Reg[IR[20-16]]= MDR;
```

Important: There is no reason from a datapath (or control) point of view that Step 5 cannot be eliminated by performing

```
Reg[IR[20-16]]= Memory[ALUOut];
```

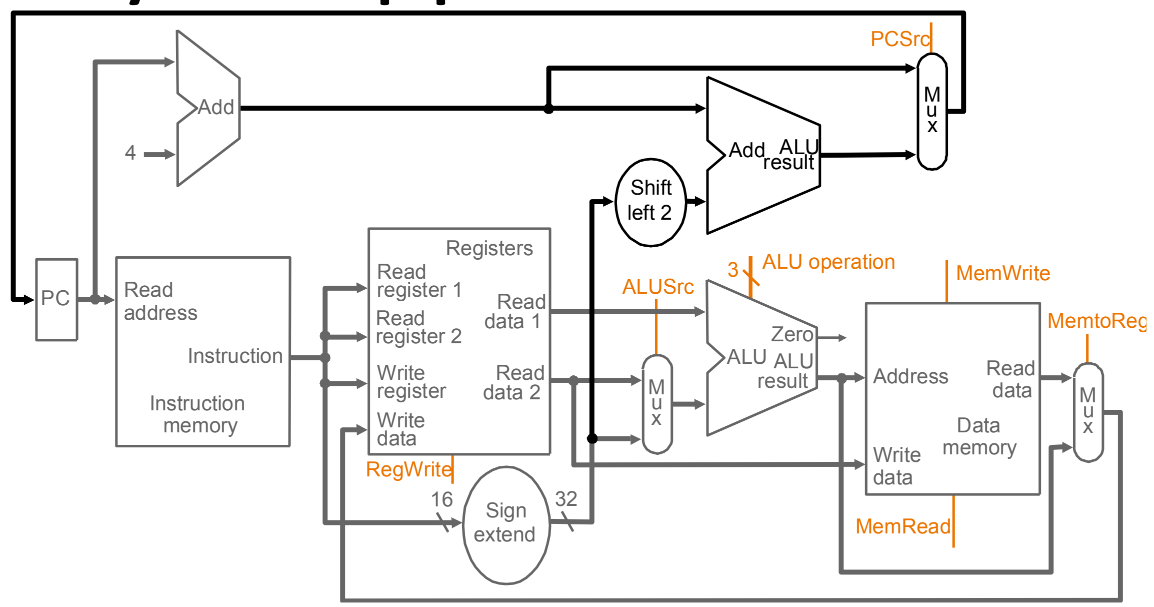
for loads in Step 4. This would eliminate the MDR as well.

The reason this is not done is that, to keep steps balanced in length, the design restriction is to allow each step to contain ***at most*** one ALU operation, or one register access, or one memory access.

Summary of Instruction Execution

Step	Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
1: IF	Instruction fetch	IR = Memory[PC] PC = PC + 4			
2: ID	Instruction decode/register fetch	A = Reg [IR[25-21]] B = Reg [IR[20-16]] ALUOut = PC + (sign-extend (IR[15-0]) << 2)			
3: EX	Execution, address computation, branch/ jump completion	ALUOut = A op B	ALUOut = A + sign-extend (IR[15-0])	if (A ==B) then PC = ALUOut	PC = PC [31-28] II (IR[25-0]<<2)
4: MEM	Memory access or R-type completion	Reg [IR[15-11]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B		
5: WB	Memory read completion		Load: Reg[IR[20-16]] = MDR		

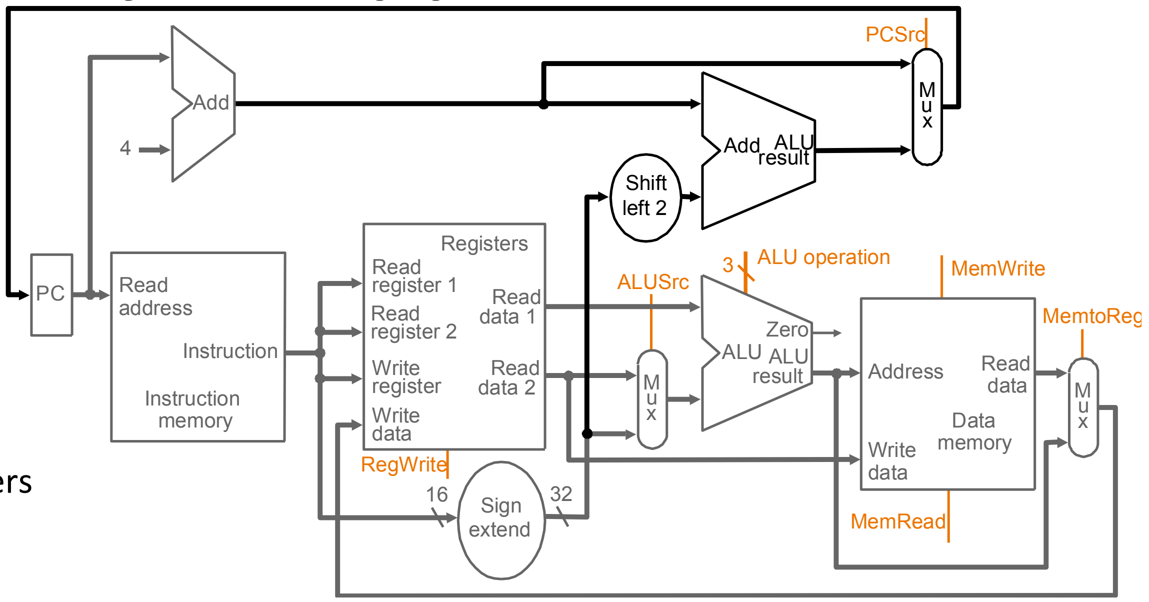
Multicycle Approach



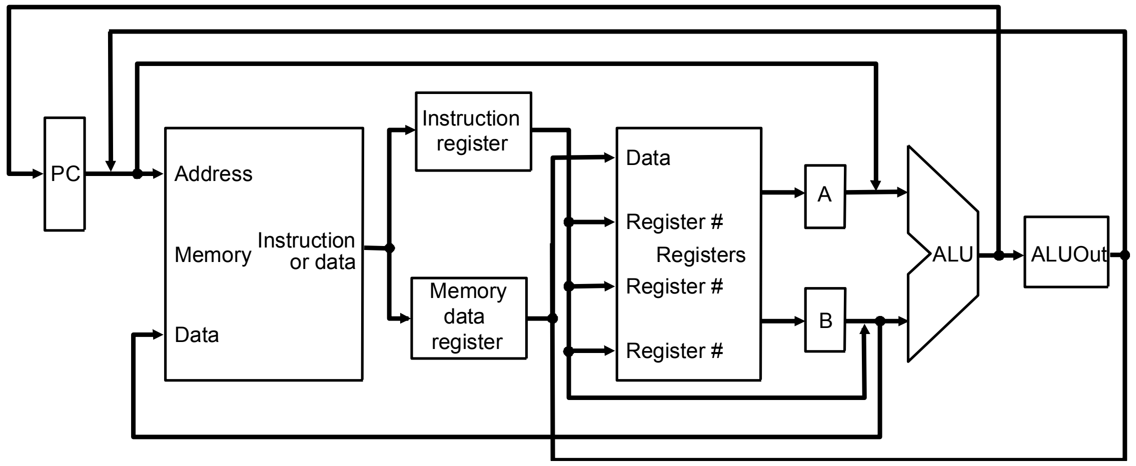
Single-cycle datapath

Multicycle Approach

- Note particularities of multicycle vs. single-diagrams
  - single memory for data and instructions
  - single ALU, no extra adders
  - extra registers to hold data between clock cycles

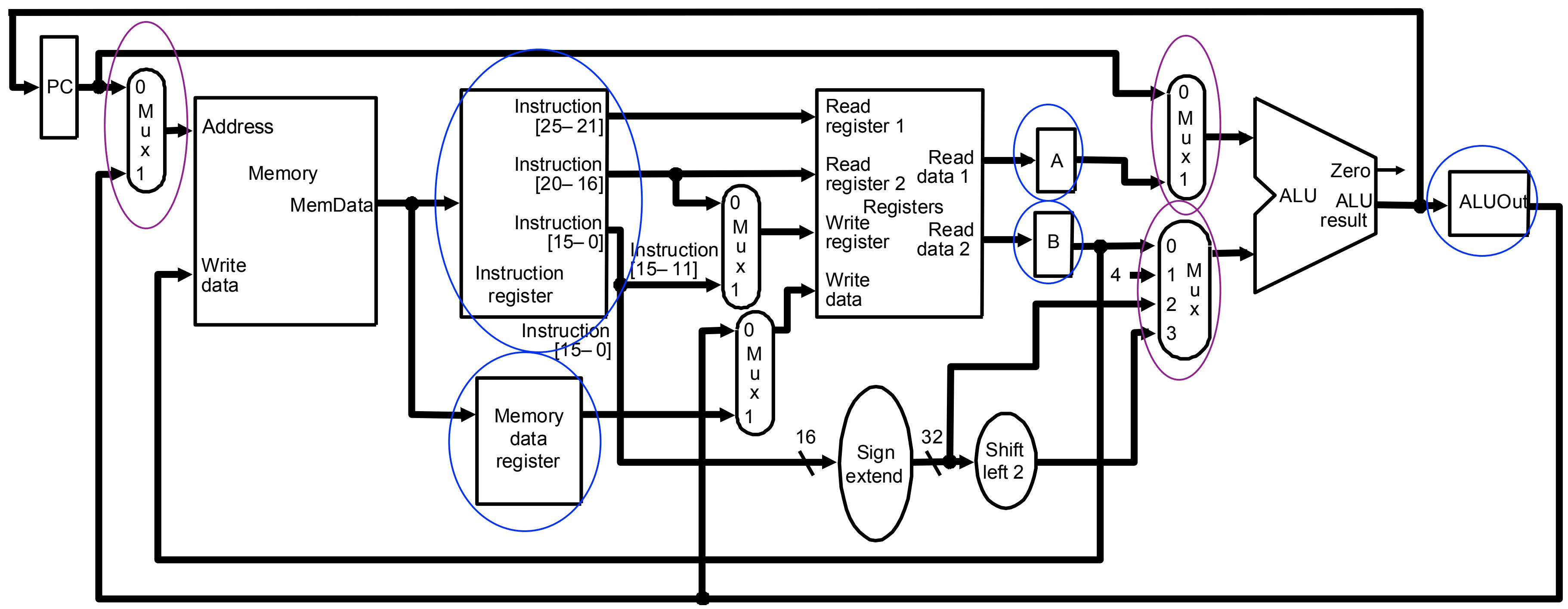


Single-cycle datapath



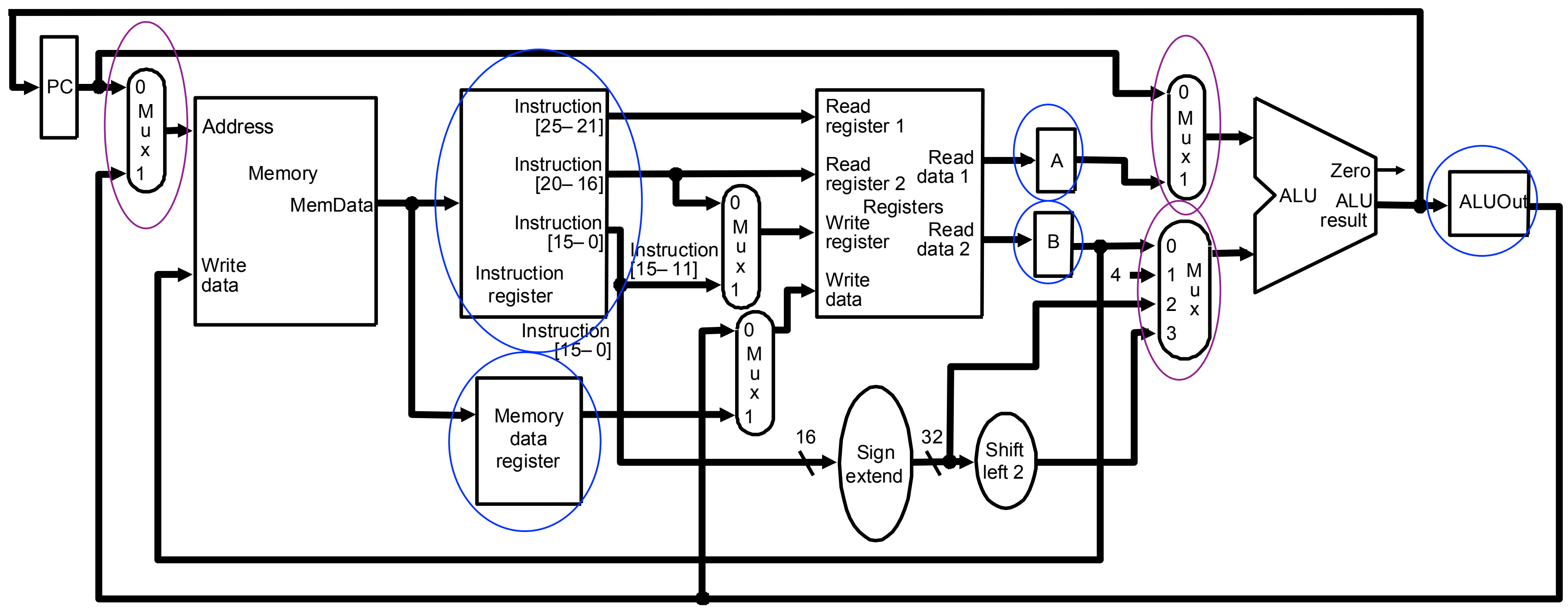
Multicycle datapath (high-level view)

Multicycle Datapath



Basic multicycle MIPS datapath handles R-type instructions and load/stores: new internal registers and new multiplexors in ovals.

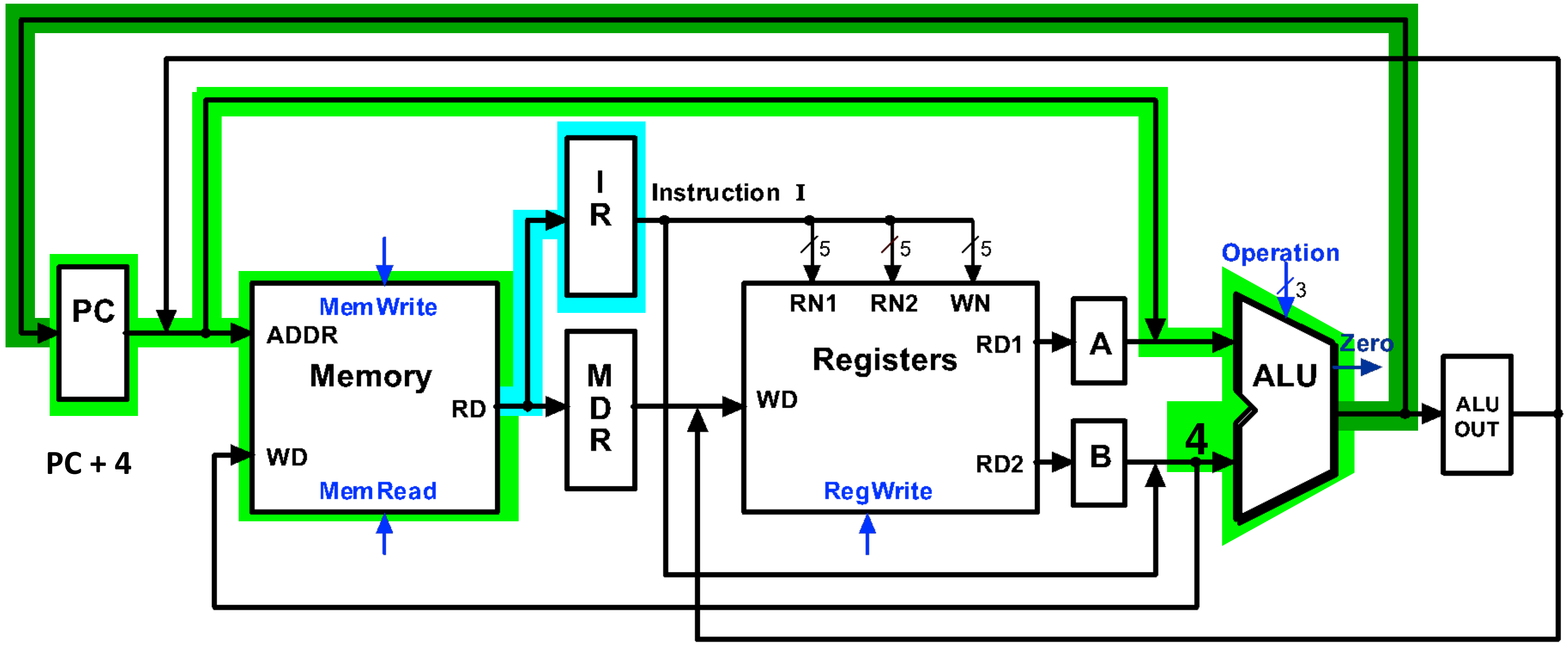
Multicycle Datapath



Basic multicycle MIPS datapath handles R-type instructions and load/stores: new internal registers and new multiplexors in ovals.

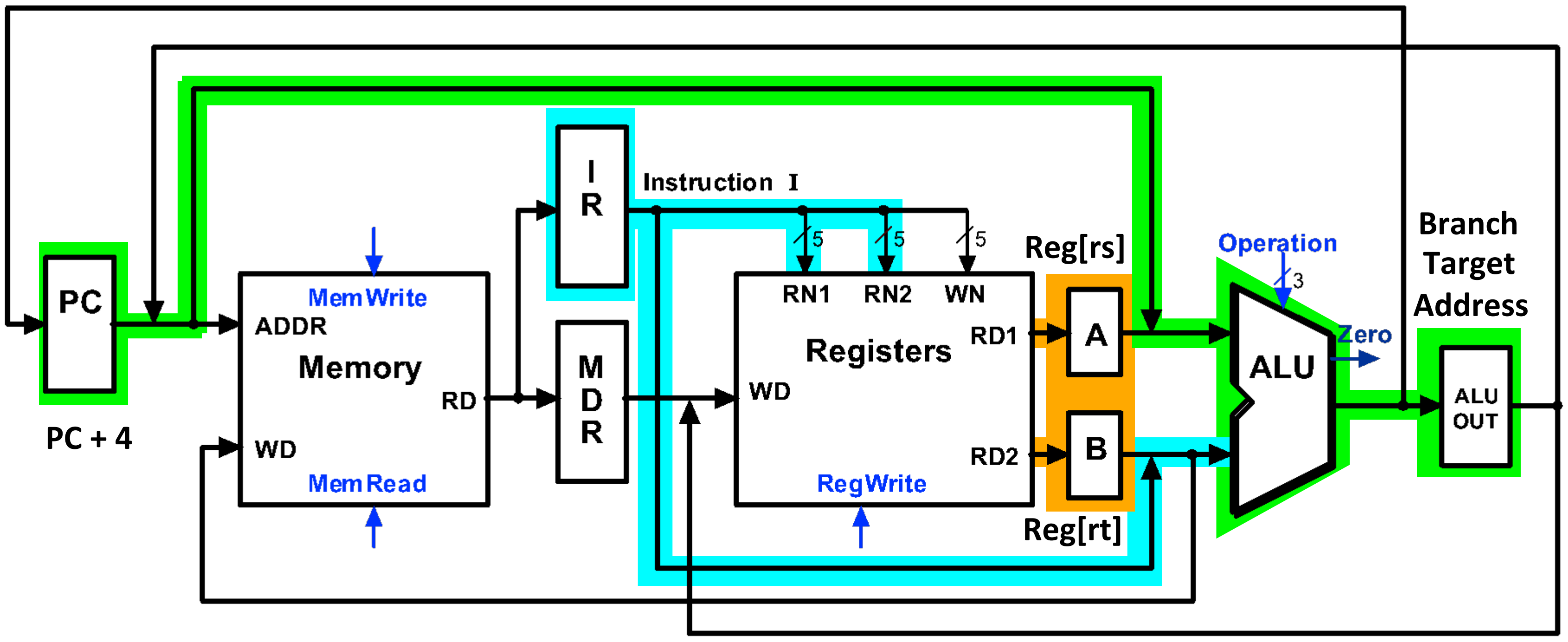
Multicycle Execution Step (1): Instruction Fetch

```
IR = Memory[PC];
PC = PC + 4;
```



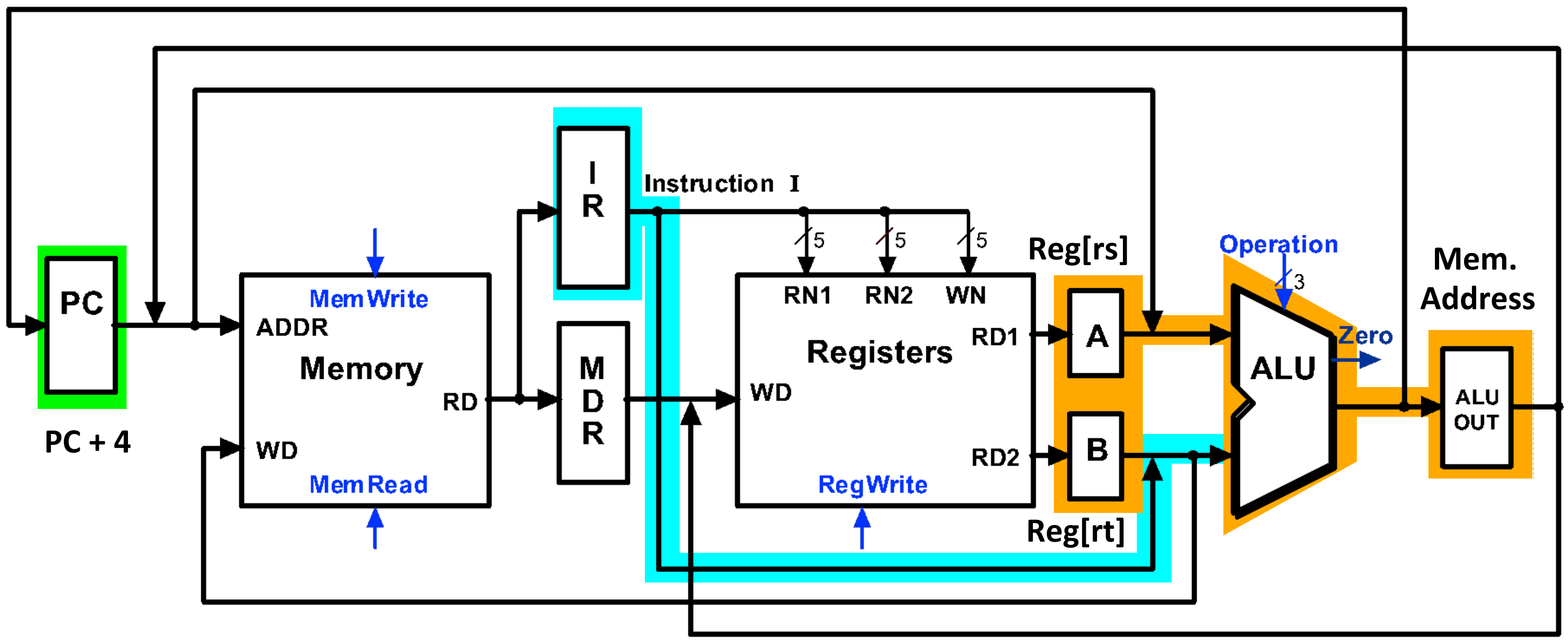
Multicycle Execution Step (2): Instruction Decode & Register Fetch

```
A = Reg[IR[25-21]];      (A = Reg[rs])
B = Reg[IR[20-15]];      (B = Reg[rt])
ALUOut = (PC + sign-extend(IR[15-0]) << 2)
```



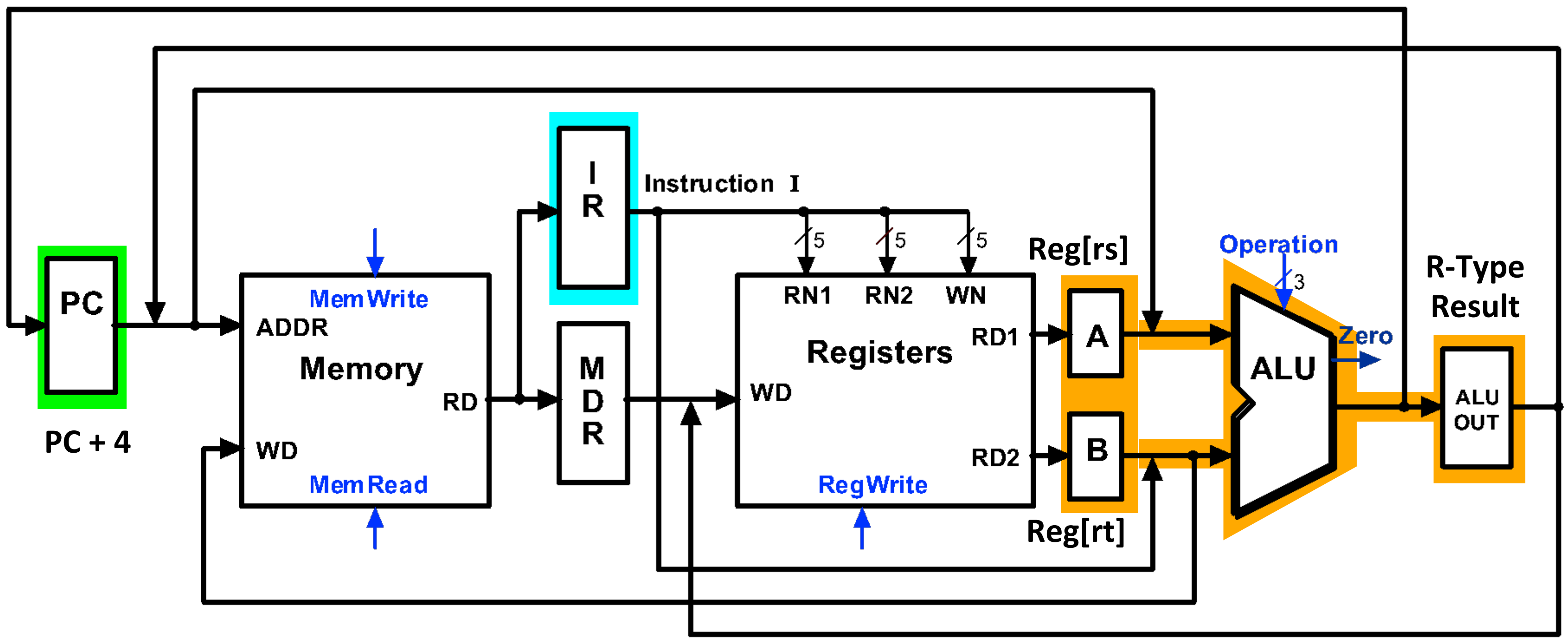
Multicycle Execution Step (3): Memory Reference Instructions

```
ALUOut = A + sign-extend(IR[15-0]);
```



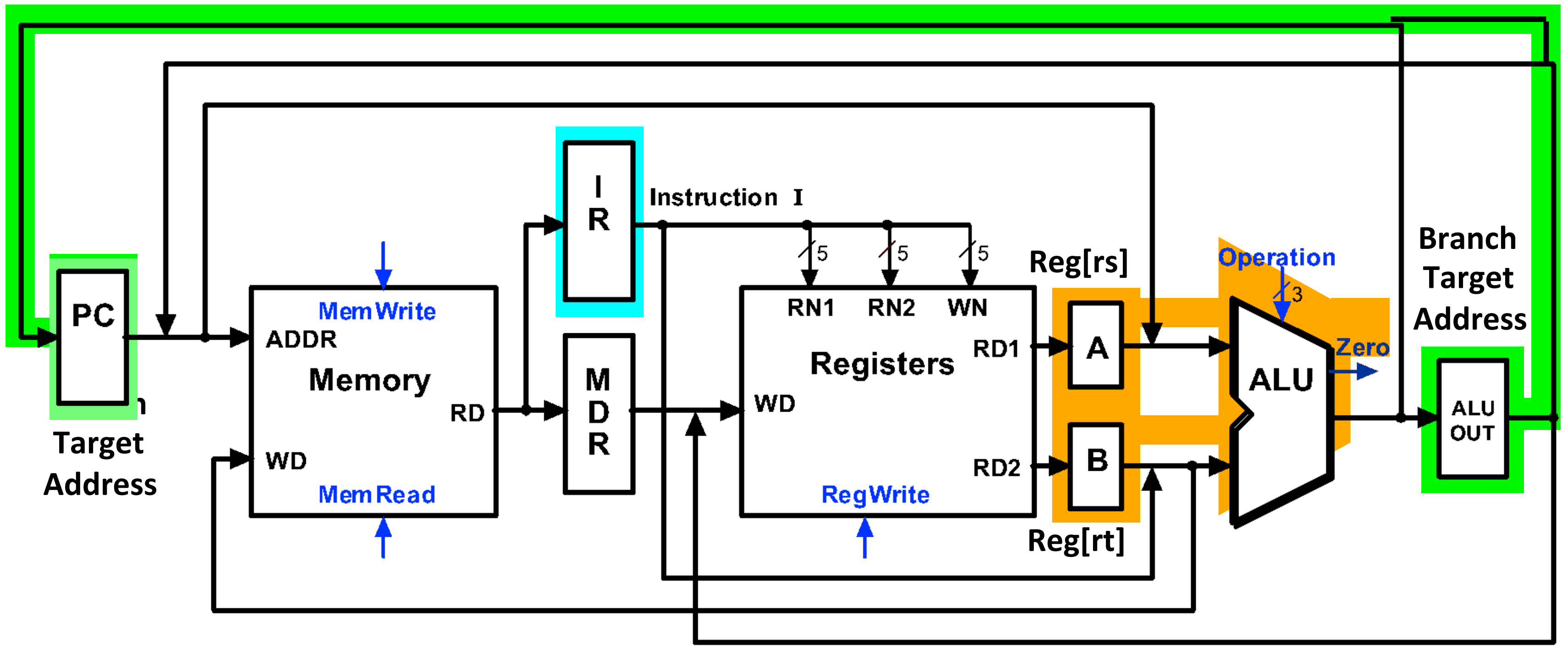
Multicycle Execution Step (3): ALU Instruction (R-Type)

```
ALUOut = A op B
```



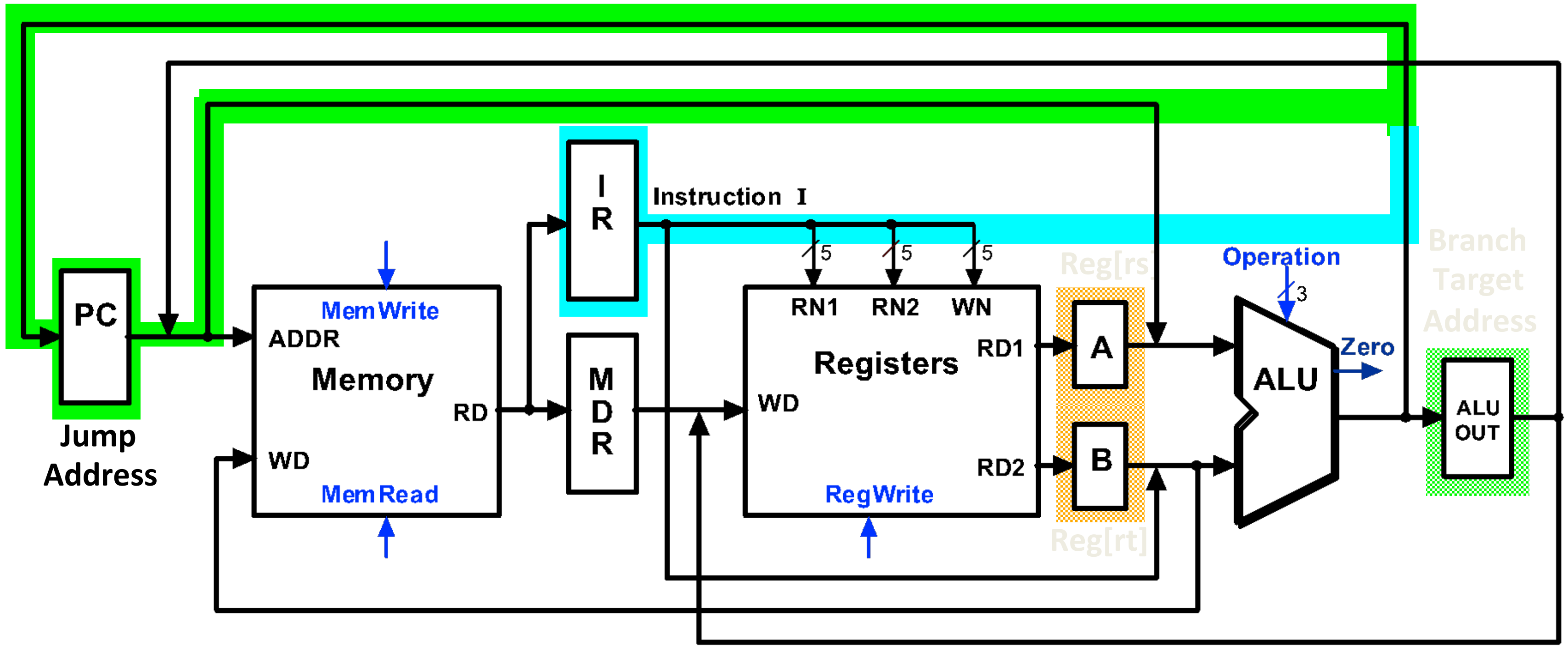
Multicycle Execution Step (3): Branch Instructions

```
if (A == B) PC = ALUOut;
```



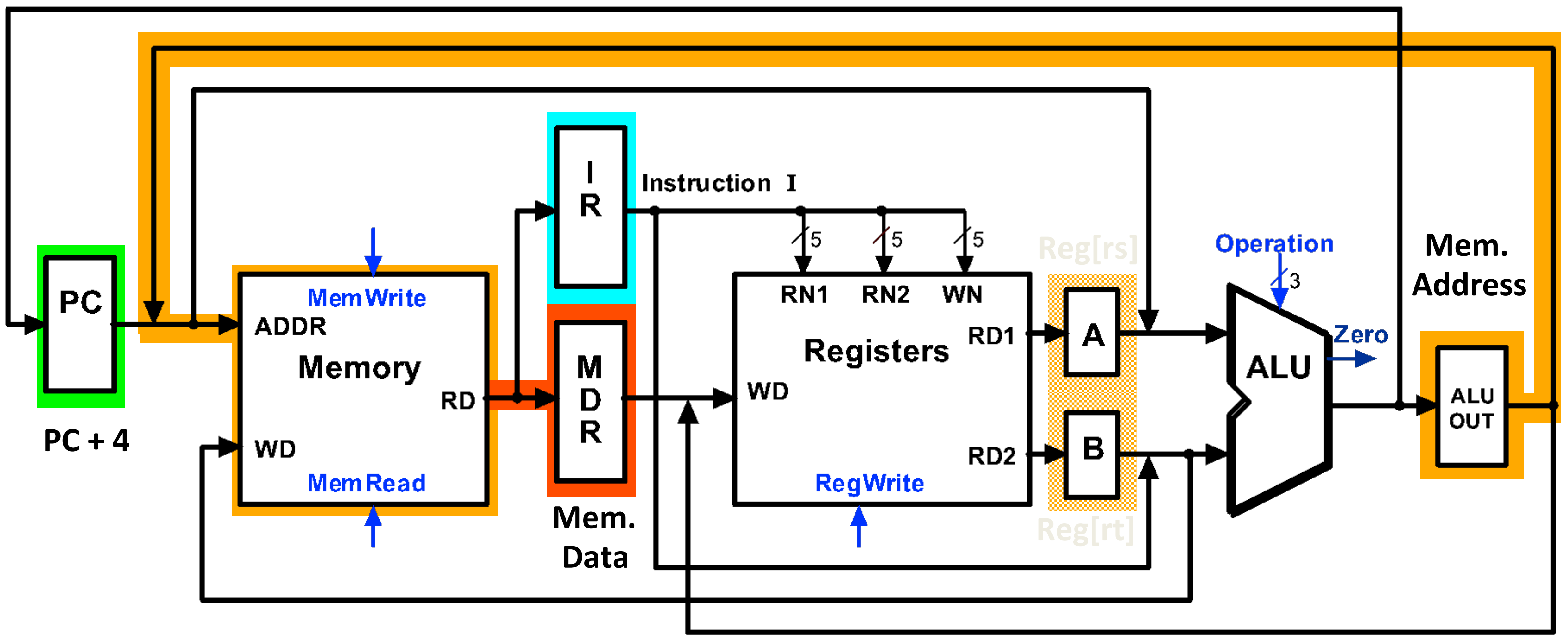
Multicycle Execution Step (3): Jump Instruction

```
PC = PC[31-28] concat (IR[25-0] << 2)
```



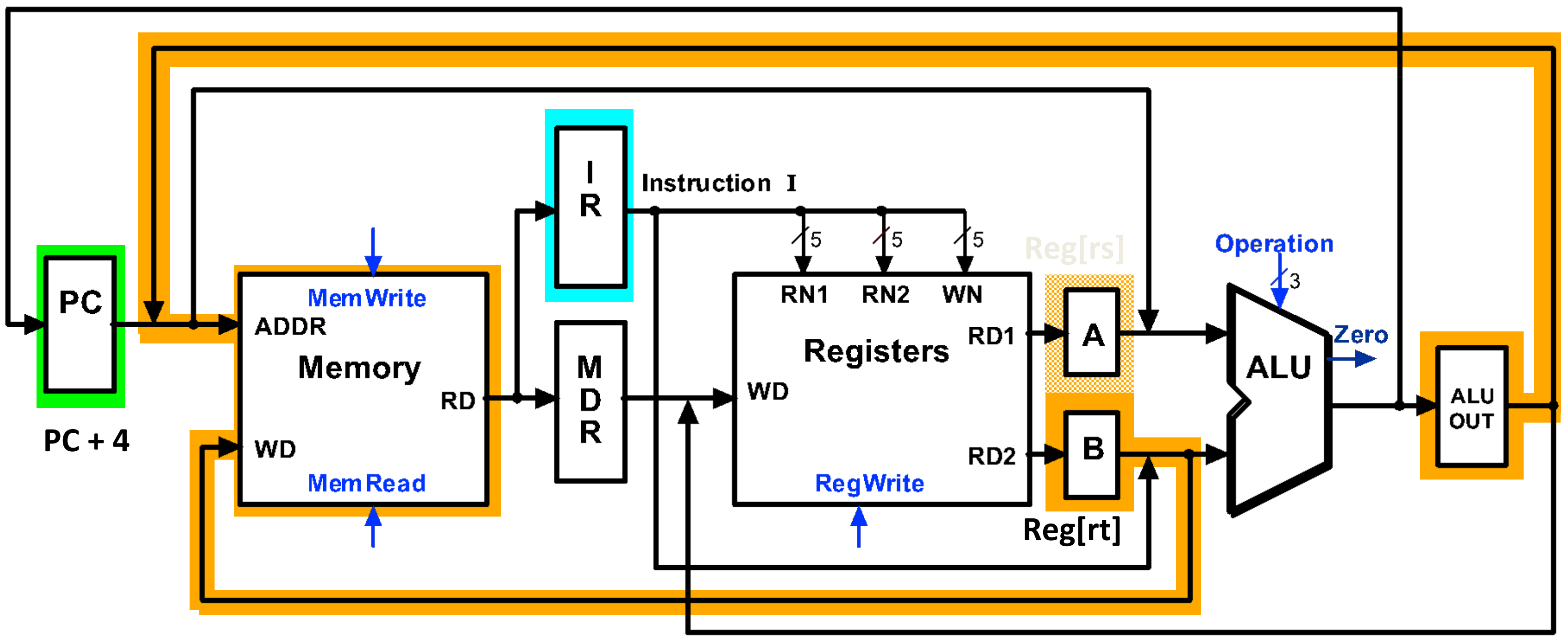
Multicycle Execution Step (4): Memory Access - Read (`lw`)

```
MDR = Memory[ALUOut];
```



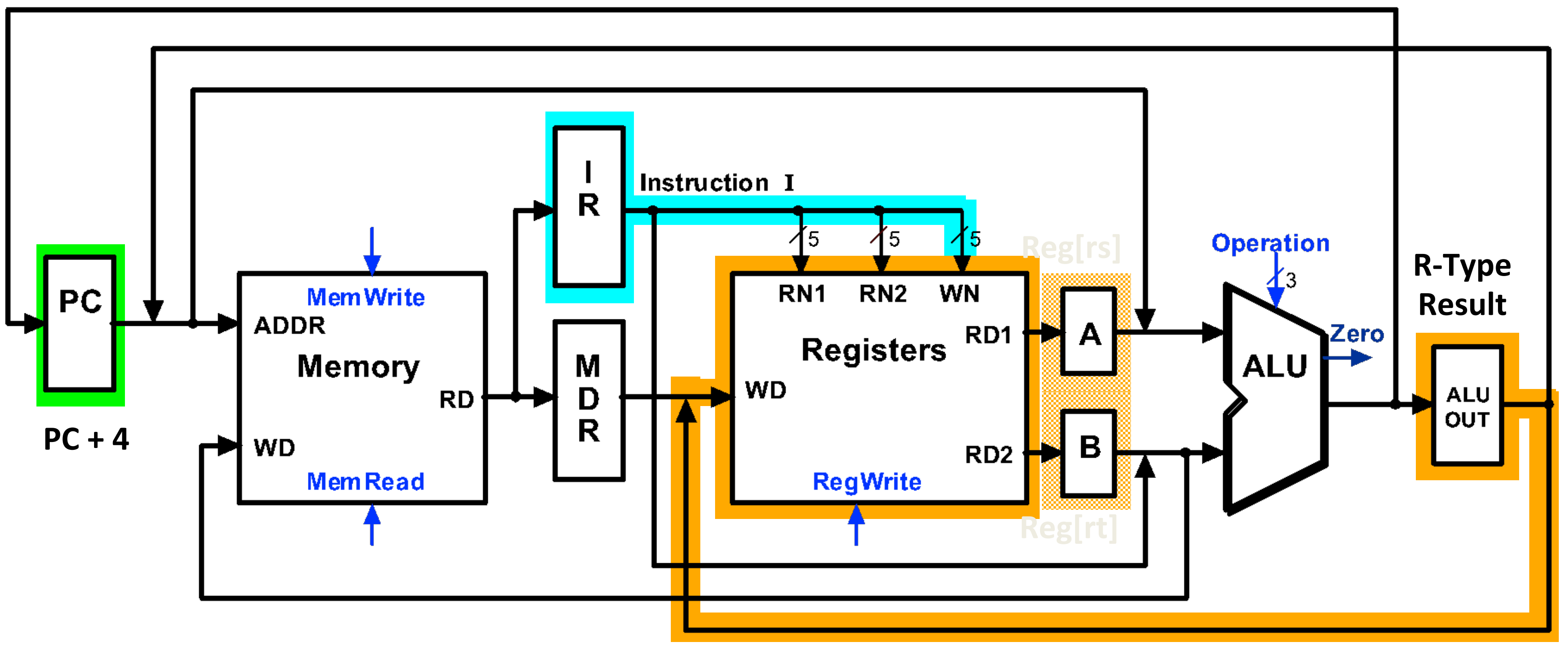
Multicycle Execution Step (4): Memory Access - Write (`sw`)

```
Memory[ALUOut] = B;
```



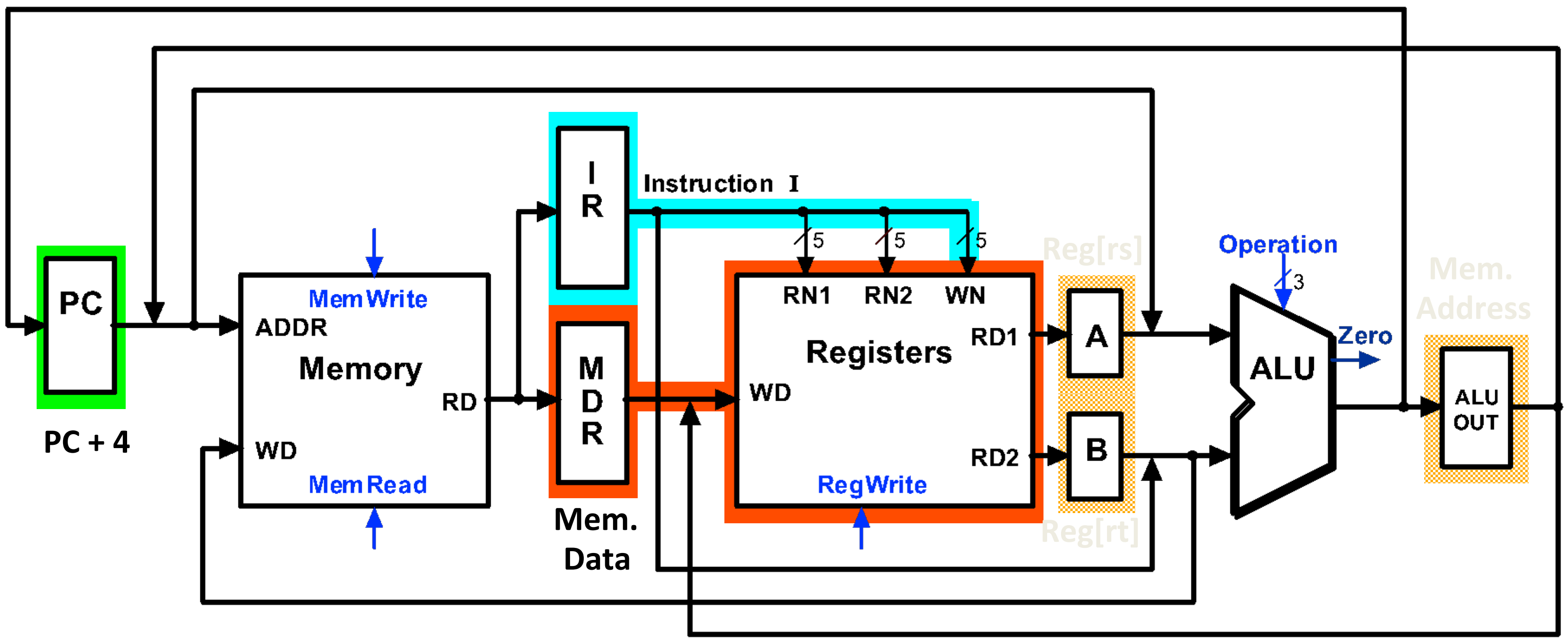
Multicycle Execution Step (4): ALU Instruction (R-Type)

```
Reg[IR[15:11]] = ALUOUT
```



Multicycle Execution Step (5): Memory Read Completion (`lw`)

```
Reg[IR[20-16]] = MDR;
```



Review and Questions

- Problems with single-cycle
- Steps
- RTL