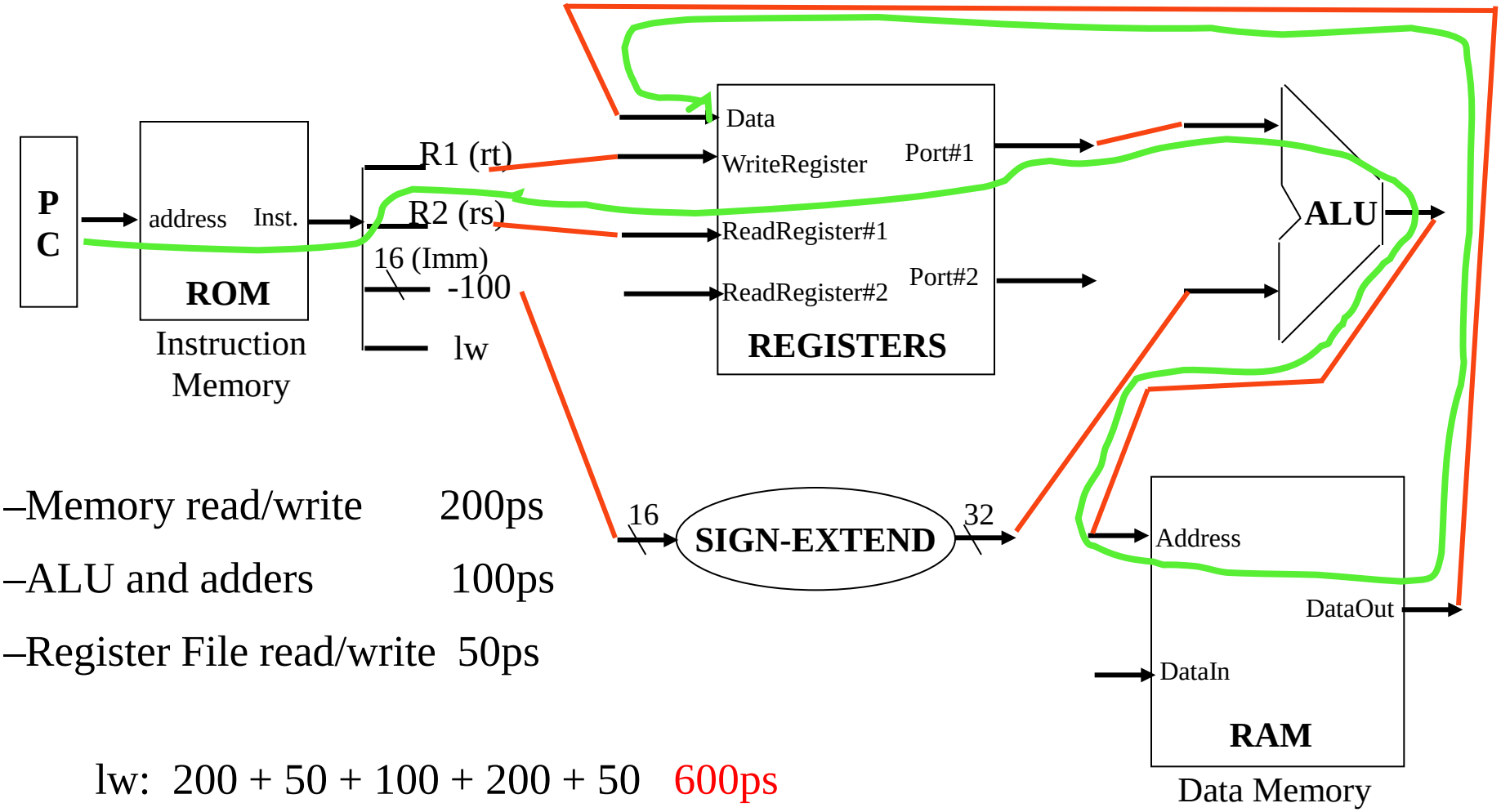
The Problem with Single-Cycle Processor Implementation: Performance

- Performance is limited by the slowest instruction
- Example: suppose we have the following delays
  - Memory read/write 200ps
  - ALU and adders 100ps
  - Register File read/write 50ps
- What is the critical path for each instruction?

What is the critical path for lw?



–Memory read/write 200ps

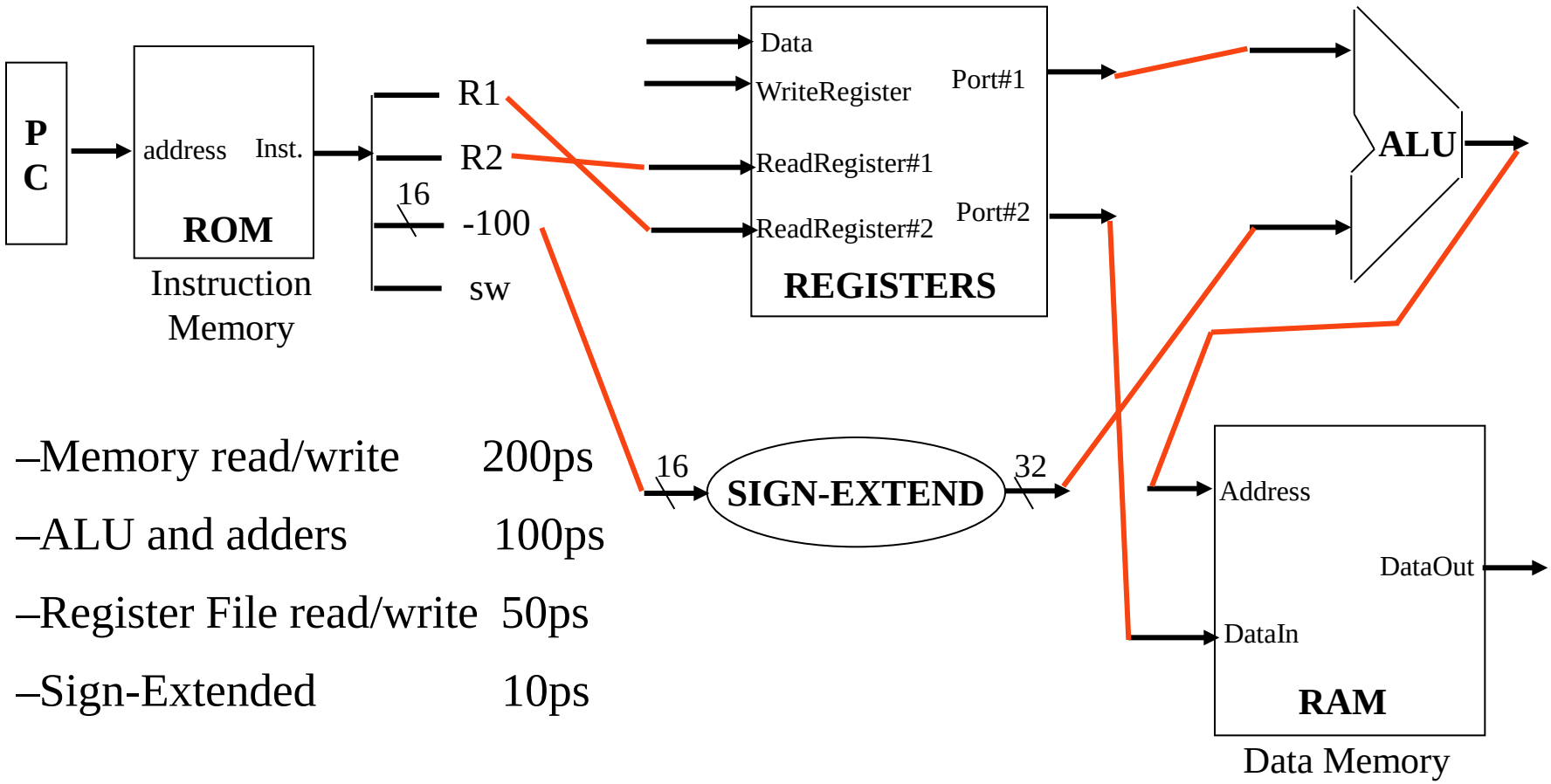
–ALU and adders 100ps

–Register File read/write 50ps

lw: 200 + 50 + 100 + 200 + 50 600ps

What is the critical path for sw?

sw R1, -100(R2)



–Memory read/write 200ps

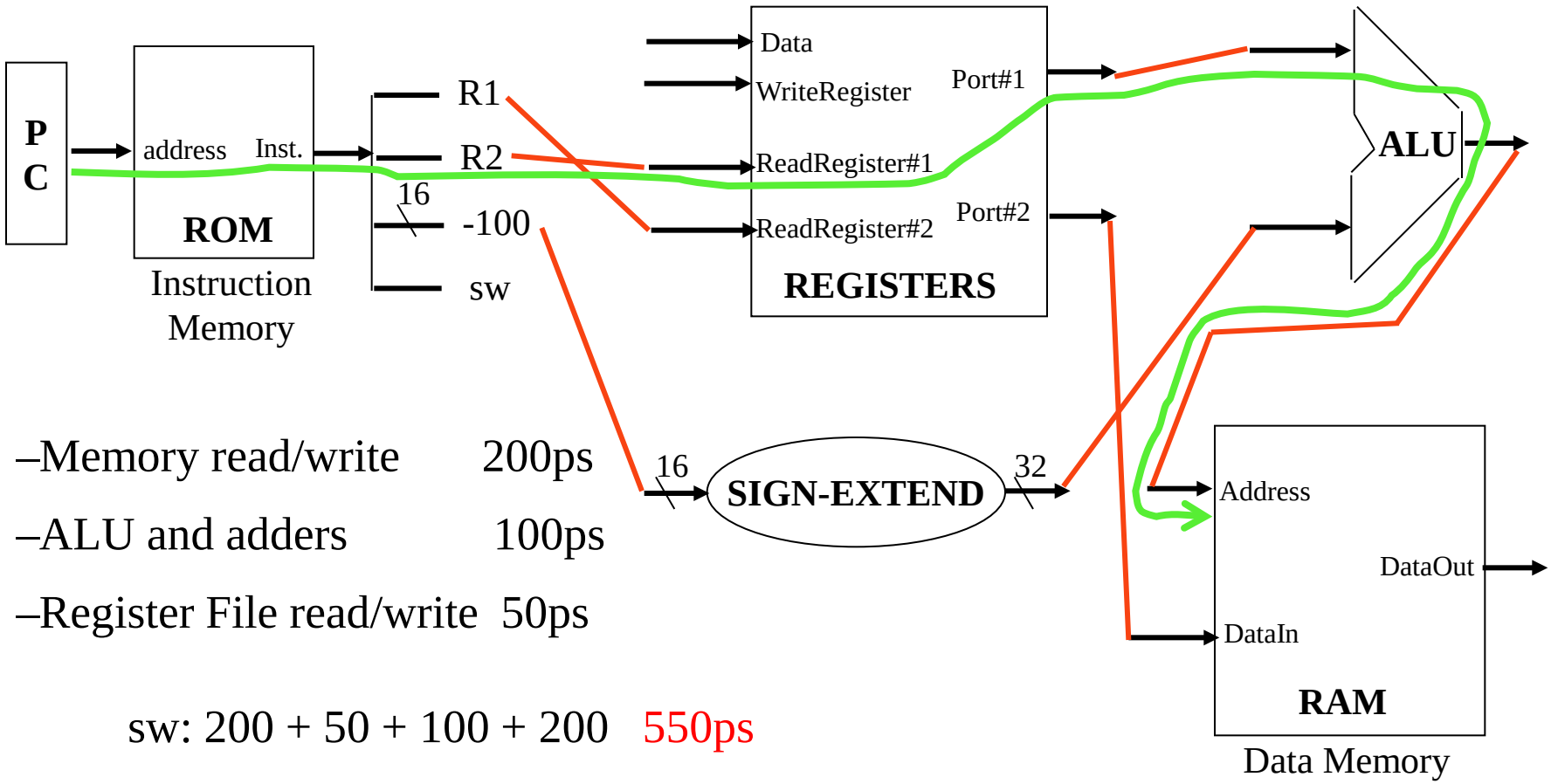
–ALU and adders 100ps

–Register File read/write 50ps

–Sign-Extended 10ps

What is the critical path for sw?

sw R1, -100(R2)



–Memory read/write 200ps

–ALU and adders 100ps

–Register File read/write 50ps

sw: 200 + 50 + 100 + 200 550ps

What is the critical path for each instruction?

– R-format	200 + 50 + 100 + 0 + 50	400ps
– Load word	200 + 50 + 100 + 200 + 50	600ps
– Store word	200 + 50 + 100 + 200	550ps
– Branch	200 + 50 + 100	350ps
– Jump	200	200ps

What is the implication?

Alternatives to Single-Cycle

- Multicycle Processor Implementation
  - Shorter clock cycle
  - Multiple clock cycles per instruction
  - Some instructions take more cycles then others
  - Less hardware required
- Pipelined Implementation
  - Overlap execution of instructions
  - Try to get short cycle times and low CPI
  - More hardware required … but also more performance!

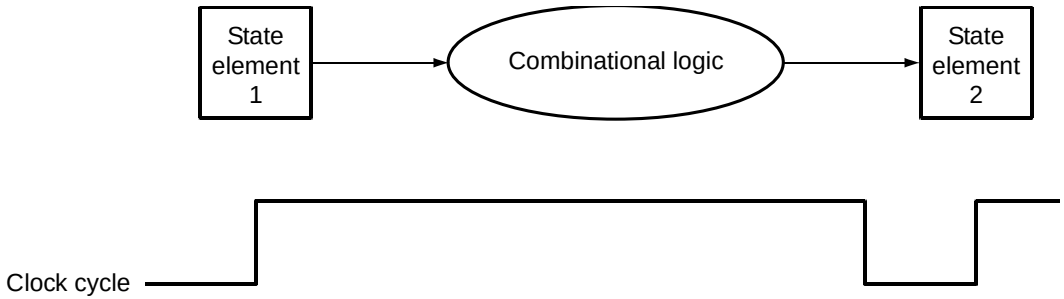
Summary

- Single-cycle control is purely combinational logic (*1-state FSM*)
- Complex logic requirements, like ALUs are often broken down in simpler components for a hierarchical design (2-level in this case)
- Once the dataflow for each instruction is understood, how to enable it through control points is straightforward.
- Logic synthesis tools can be very helpful in obtaining error-free logic (once the specs are right).

Processor: Multicycle Implementation

Our Simple Control Structure

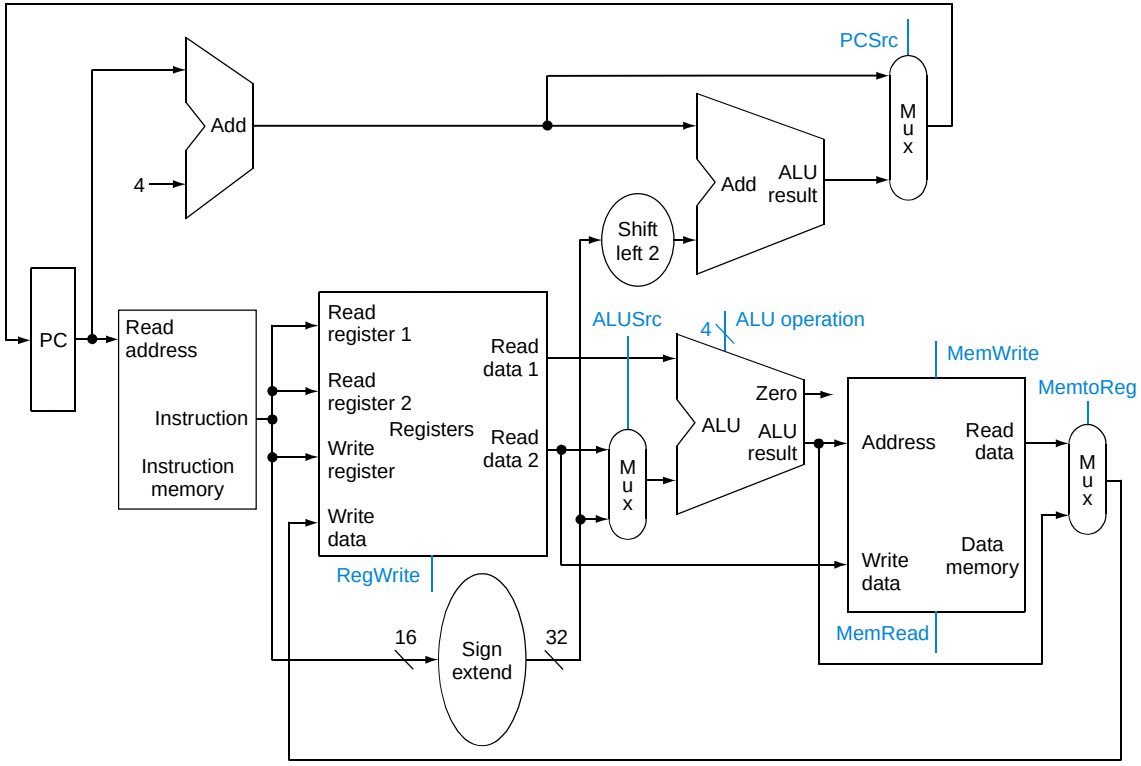
- All of the logic is combinational
- We wait for everything to settle down, and the right thing to be done
  - ALU might not produce “right answer” right away
  - We use write signals along with clock to determine when to write
- Cycle time determined by length of the longest path



We are ignoring some details like setup and hold times

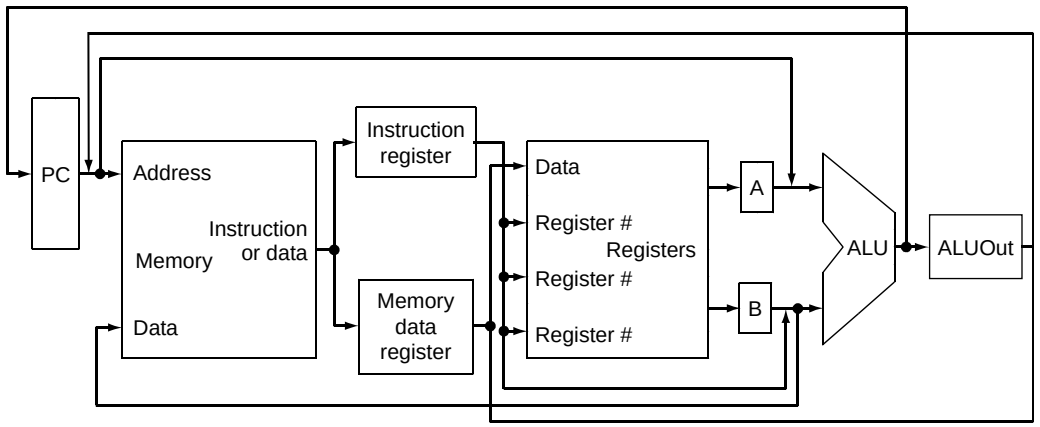
Single Cycle Implementation

- Calculate cycle time assuming negligible delays except:
  - –memory (200ps),
  - –ALU and adders (100ps),
  - –register file access (50ps)



Where we are headed

- Single Cycle Problems:
  - what if we had a more complicated instruction like floating point?
  - wasteful of area
- One Solution:
  - use a “smaller” cycle time
  - have different instructions take different numbers of cycles
  - a “multicycle” datapath:

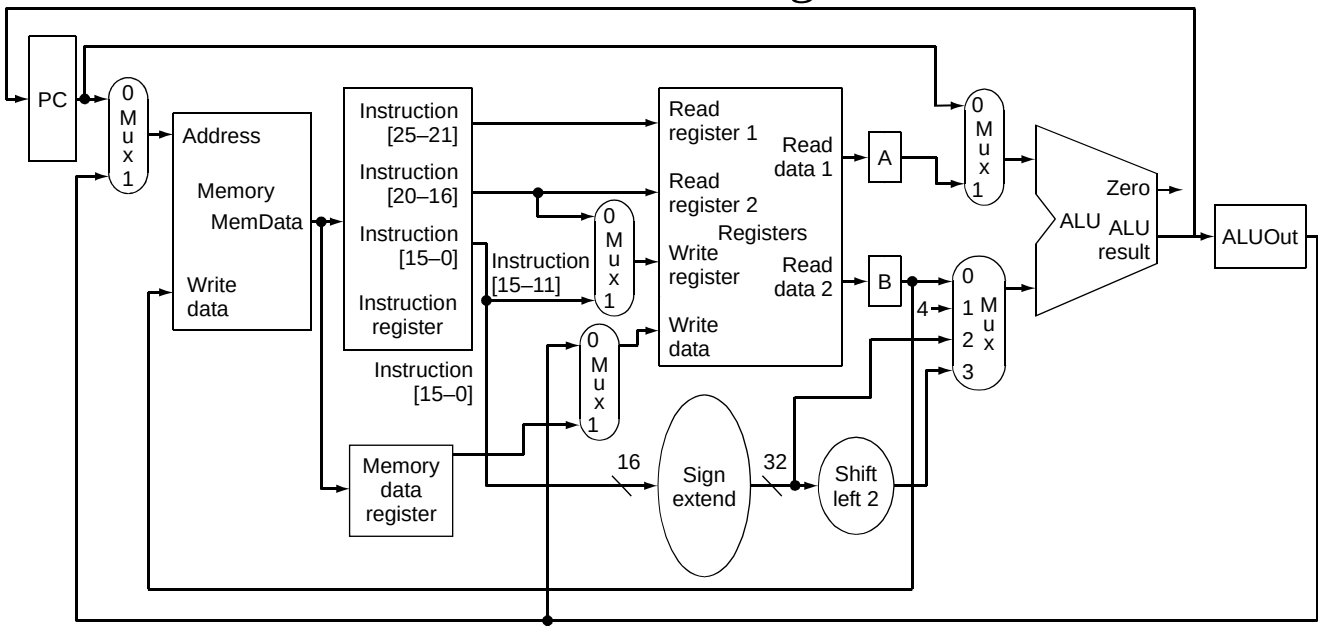


Multicycle Approach

- We will be reusing functional units
  - ALU used to compute address and to increment PC
  - Memory used for instruction and data
- Our control signals will not be determined directly by instruction
  - e.g., what should the ALU do for a “subtract” instruction?
- We’ll use a finite state machine for control

Multicycle Approach

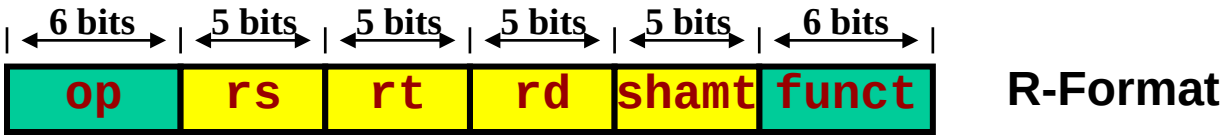
- Break up the instructions into steps, each step takes a cycle
  - balance the amount of work to be done
  - restrict each cycle to use only one major functional unit
- At the end of a cycle
  - store values for use in later cycles (easiest thing to do)
  - introduce additional “internal” registers



Instructions from ISA perspective

- Consider each instruction from perspective of ISA.
- Example:
  - The add instruction changes a register.
  - Register specified by bits 15:11 of instruction.
  - Instruction specified by the PC.
  - New value is the sum (“op”) of two registers.
  - Registers specified by bits 25:21 and 20:16 of the instruction

```
Reg[Memory[PC][15:11]] <=
Reg[Memory[PC][25:21]] op
                  Reg[Memory[PC][20:16]]
```



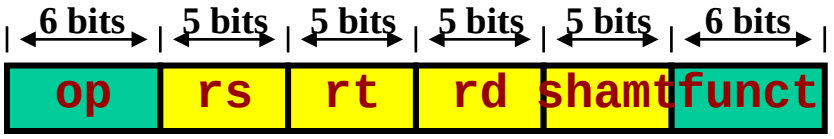
  - In order to accomplish this we must break up the instruction.

Breaking Down an Instruction

- ISA definition of arithmetic:

```
Reg[Memory[PC][15:11]] <= Reg[Memory[PC][25:21]]  op
                          Reg[Memory[PC][20:16]]
```

- Could break down to:
  - `IR <= Memory[PC]`
  - `A <= Reg[IR[25:21]]`
  - `B <= Reg[IR[20:16]]`
  - `ALUOut <= A op B`
  - `Reg[IR[15:11]] <= ALUOut`



R-Format

- We forgot an important part of the definition of arithmetic!
  - `PC <= PC + 4`

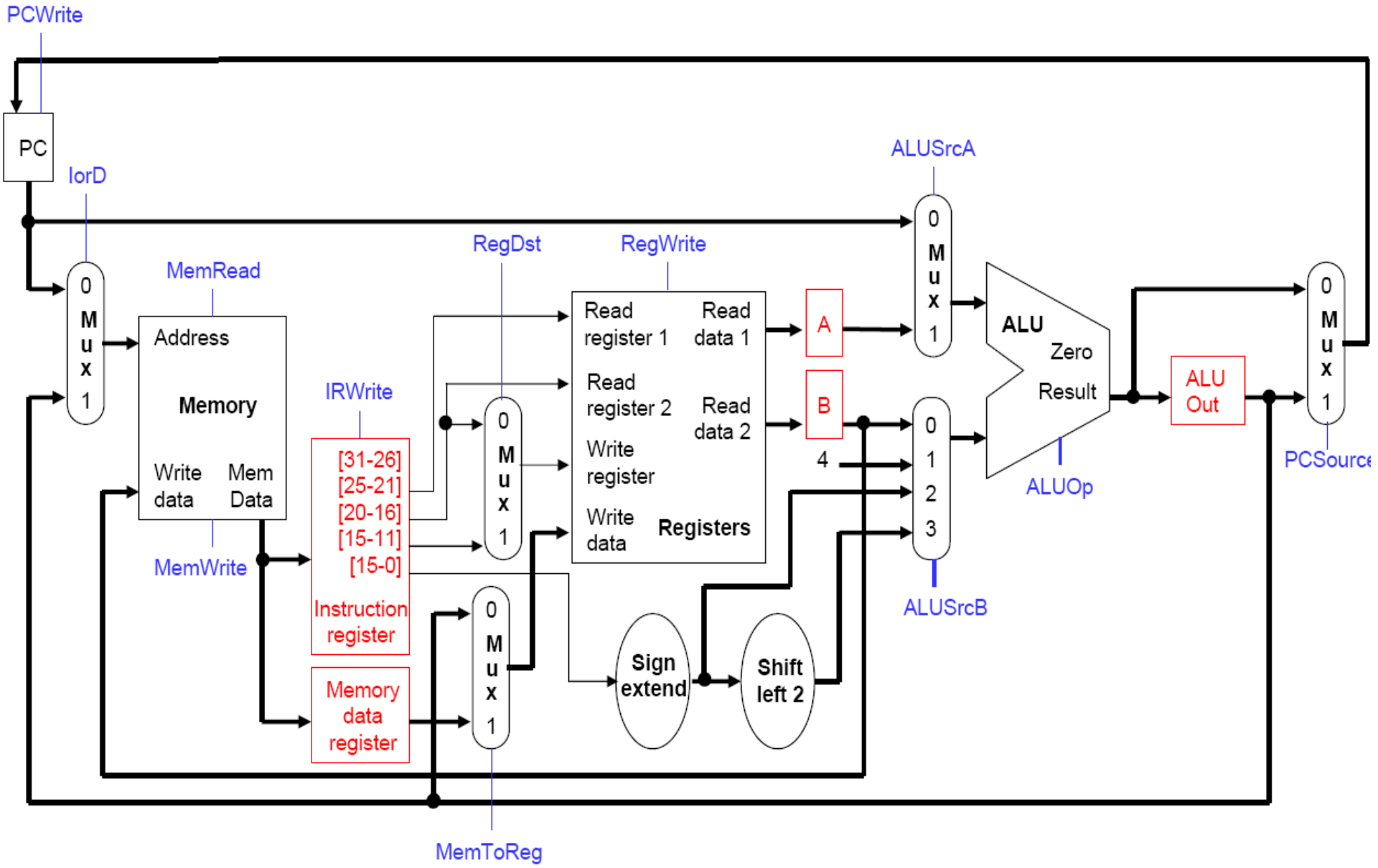
Idea behind multicycle approach

- We define each instruction from the ISA perspective (do this!)
- Break it down into steps following our rule that data flows through at most one major functional unit (e.g., balance work across steps)
- Introduce new registers as needed (e.g, A, B, ALUOut, MDR, etc.)
- Finally try and pack as much work into each step
  (avoid unnecessary cycles)
  while also trying to share steps where possible
  (minimizes control, helps to simplify solution)

Intermediate registers

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

Five Execution Steps

- Instruction Fetch
- Instruction Decode and Register Fetch
- Execution, Memory Address Computation, or Branch Completion
- Memory Access or R-type instruction completion
- Write-back step

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

Step 1: Instruction Fetch

- 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"

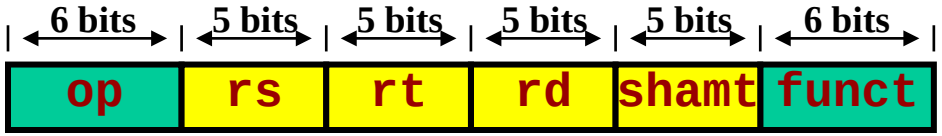
```
IR <= Memory[PC];
PC <= PC + 4;
```

Can we figure out the values of the control signals?

What is the advantage of updating the PC now?

Step 2: Instruction Decode and Register Fetch

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



R-Format

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

- We aren't setting any control lines based on the instruction type
  (we are busy "decoding" it in our control logic)

Step 3 (instruction dependent)

- ALU is performing one of three functions, based on instruction type
- Memory Reference:

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

- R-type:

```
ALUOut <= A op B;
```

- Branch:

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

Step 4 (R-type or memory-access)

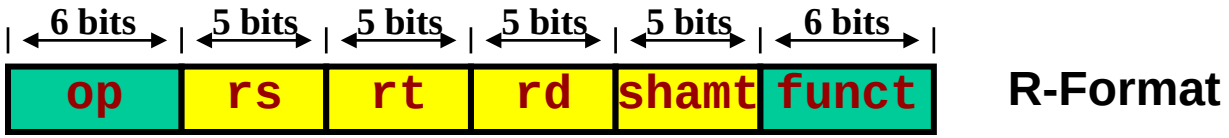
- Loads and stores access memory

```
MDR <= Memory[ALUOut];
       or
Memory[ALUOut] <= B;
```

- R-type instructions finish

```
Reg[IR[15:11]] <= ALUOut;
```

The write actually takes place at the end of the cycle on the edge

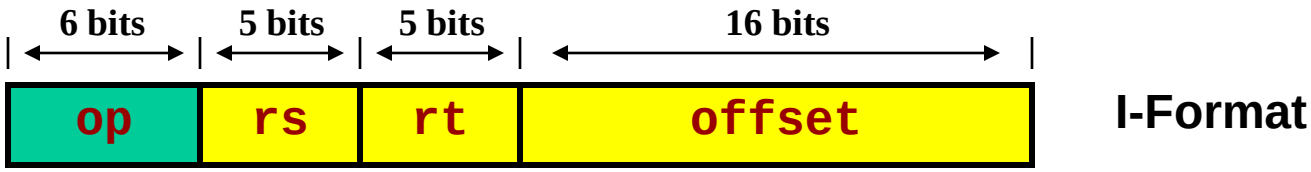


R-Format

Step 5 (Write-back step)

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

Which instruction needs this?



I-Format

Summary:

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR <= Memory[PC] PC <= PC + 4			
Instruction decode/register fetch	A <= Reg [IR[25:21]] B <= Reg [IR[20:16]] ALUOut <= PC + (sign-extend (IR[15:0]) << 2)			
Execution, address computation, branch/jump completion	ALUOut <= A op B	ALUOut <= A + sign-extend (IR[15:0])	If (A == B) PC <= ALUOut	PC <= {PC [31:28], (IR[25:0]],2'b00)}
Memory access or R-type completion	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B		
Memory read completion		Load: Reg[IR[20:16]] <= MDR		

FIGURE 5.30 Summary of the steps taken to execute any instruction class. Instructions take from three to five execution steps. The first two steps are independent of the instruction class. After these steps, an instruction takes from one to three more cycles to complete, depending on the instruction class. The empty entries for the Memory access step or the Memory read completion step indicate that the particular instruction class takes fewer cycles. In a multicycle implementation, a new instruction will be started as soon as the current instruction completes, so these cycles are not idle or wasted. As mentioned earlier, the register file actually reads every cycle, but as long as the IR does not change, the values read from the register file are identical. In particular, the value read into register B during the Instruction decode stage, for a branch or R-type instruction, is the same as the value stored into B during the Execution stage and then used in the Memory access stage for a store word instruction.

- 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.
- Next time, we'll look at controlling this datapath.