Rodrigo Becerril Ferreyra

Dr. Gevik Sardarbegians

CECS 440 Section 02

18 November 2021

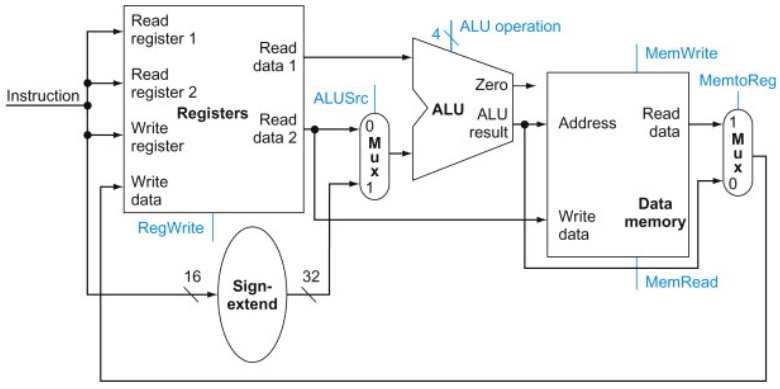
Exercise 4

Problem 4.1

For reference, the problem asks to use COD Figure 4.10; here is a screenshot of it.

Figure 4.3.4: The datapath for the memory instructions and the R-type instructions (COD Figure 4.10).

This example shows how a single datapath can be assembled from the pieces in COD Figures 4.7 (The two elements needed to implement R-format ALU operations ...) and 4.8 (The two units needed to implement loads and stores ...) by adding multiplexors. Two multiplexors are needed, as described in the example.



Feedback?

The instruction in question is `AND Rd, Rs1, Rs2`.

4.1.1: The control signals are all in blue: `RegWrite`, `ALUSrc`, `ALUop`(eration), `MemWrite`, `MemRead`, and `MemtoReg`. For an `AND` operation, `RegWrite` is `1`, because it is necessary to write `Rd` with the result of the operation. `ALUSrc` is `0`, because we will be reading from the data registers. The `ALUop` vector is `0000`, as specified in Chapter 4.4. `MemWrite` is

`0`, and `MemRead` is `0`, since we will not be using the memory for this operation. `MemtoReg` is also `0`, which means we will be bypassing the memory block completely.

4.1.2: Both the sign-extend block and the memory block do not play a part in this operation. The sign-extend block is used in load and store operations, which we are not using. The memory block is also only used in load and store operations.

4.1.3: The memory block does not produce an output in this situation. This is because `MemRead` is not asserted; therefore, no output generated by the memory block. The sign-extend block, being a combinational block, always produces an output; however, this output is not used in this operation because it is being "blocked" by a multiplexer.

Problem 4.3

For reference, here is the instruction mix in the problem.

R-type	I-type (non-lw)	Load	Store	Branch	Jump
24%	28%	25%	10%	11%	2%

4.3.1: The only instructions that interact with data memory are load and store instructions. Combined, they represent 35% of all instructions in the program, a fraction of 7/20.

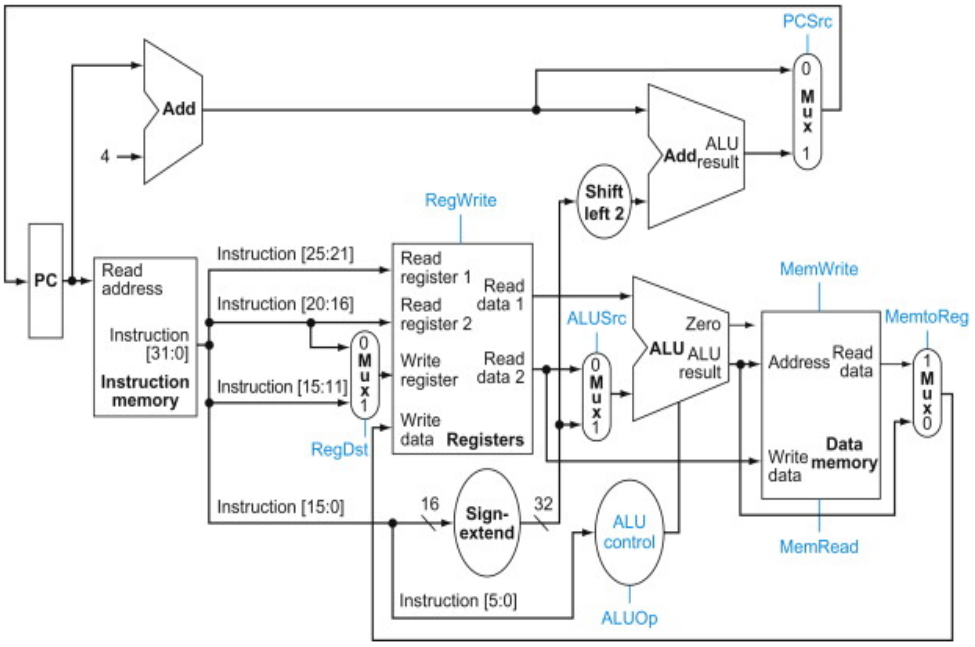
4.3.2: All instructions must be fetched from instruction memory, meaning that 100% of the instructions use instruction memory.

4.3.3: Only load word and store word assert the control signal `ALUSrc` (Chapter 4.4, COD Figure 4.18, see picture below), meaning that only they use the sign-extend block; this means that 35% or 7/20 of the program uses the sign-extend block.

Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

4.3.4: The sign-extend block is always working and outputting a value, due to the fact that it is a combinational block and not a sequential block. The reason its value is not used when it doesn't need to is that the multiplexer controlled by `ALUSrc` blocks its output (it selects either the output of a register on `0` or the output of the sign-extend block on `1`).

Problem 4.7



For reference, above is the datapath I am using to solve this problem (it can be found in Chapter 4.4, COD Figure 4.15).

4.7.1: An R-type instruction needs to fetch an instruction from memory (250 ps). The instruction goes through the mux controlled by `RegDst`, but because this happens simultaneously with other latencies, it is not counted (i.e., no other part immediately depends on or has to wait for the mux controlled by `RegDst`). Next, the data needs to be read from the register file (30 ps), go through the mux controlled by `ALUSrc` (25 ps), go through the ALU (200 ps), and go through the mux controlled by MemtoReg (25 ps). Lastly, the data is written back into the register (20 ps). This is a total of 550 ps.

4.7.2: A load word instruction must be fetched from the memory (250 ps) and go through the mux controlled by `RegDst` (which happens simultaneously, therefore is not counted). The

instruction also goes through the sign-extend block (50 ps) and go through the mux controlled by `ALUSrc` (25 ps). The desired register is read (30 ps) and its output is placed onto the input of the ALU. The result is added in the ALU (taking 200 ps). The ALU result is fed into the data memory block and read out (250 ps). The data finally goes through one more mux, which is the one controlled by `MemtoReg` (25 ps). Lastly, the data is written back into the register (20 ps). The whole process takes 850 ps.

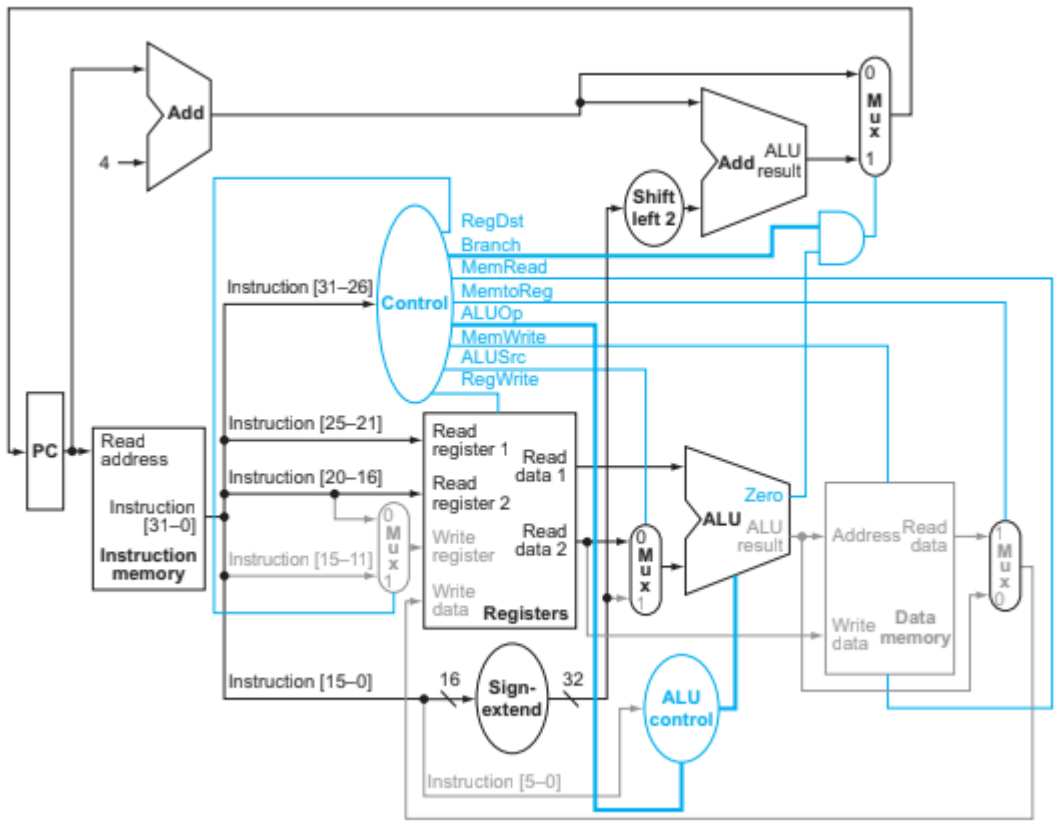
4.7.3: A store word instruction is similar to a load word instruction: it must first be read from instruction memory (250 ps); as before, the instruction goes through the mux controlled by `RegDst` simultaneously. The register needed gets read (30 ps) and its output is placed on the ALU input. The instruction also needs to go through the sign-extend block (50 ps) and the mux controlled by `ALUSrc` (25 ps). The ALU operation takes 200 ps to complete. After the result is calculated, the data is written to the data memory (20 ps). The total for the entire operation is 575 ps for everything.

4.7.4: An I-type non-load instruction (for example `addi`) must first be read from instruction memory (250 ps). The instruction goes through the mux controlled by `RegDst` simultaneously. Then, data from the desired register is read (30 ps) and placed onto the first input of the ALU. The instruction goes through the sign-extend block (50 ps) and the mux controlled by `ALUSrc` (25 ps). The ALU takes 200 ps to process the answer. The data then bypasses the memory block and passes through the mux controlled by `MemtoReg` (25 ps). The data is finally written into the desired register (20 ps). The total time taken is 600 ps.

4.7.5: The minimum clock period for this single-clock implementation is 850 ps. This is the amount of time it takes to perform a load from memory, and it is the longest instruction.

Problem 4.9

For reference, here is COD Figure 4.21 (which is asked for in the problem).



4.9.1: The clock cycle time is 850 ps without the improvement. This is the time required for a lw command as calculated in 4.7, and because we are using a single-cycle computer, this is the cycle time for the whole system. The clock cycle will be 1150 ps if the improvement is applied to this CPU. This is a 35.3% increase in cycle time.

4.9.2: Multiplying the 35.3% increase in time with the 5% decrease in commands, we get a 28.54% increase in total time needed to run the program. The program will be slower than before the “improvement.”

4.9.3: In order to break even, the cycle time must not rise over approximately 5% more than it is now. In concrete terms, the clock cycle must not rise over approximately 894 ps in order for the program to be faster than (or as fast as) it is without “improvements.”

Problem 4.13

4.13.1: I don’t think any new functional blocks are required for this task. We have the addition from the ALU and the sign-extend block for the immediate field. Nothing new is needed for this task.

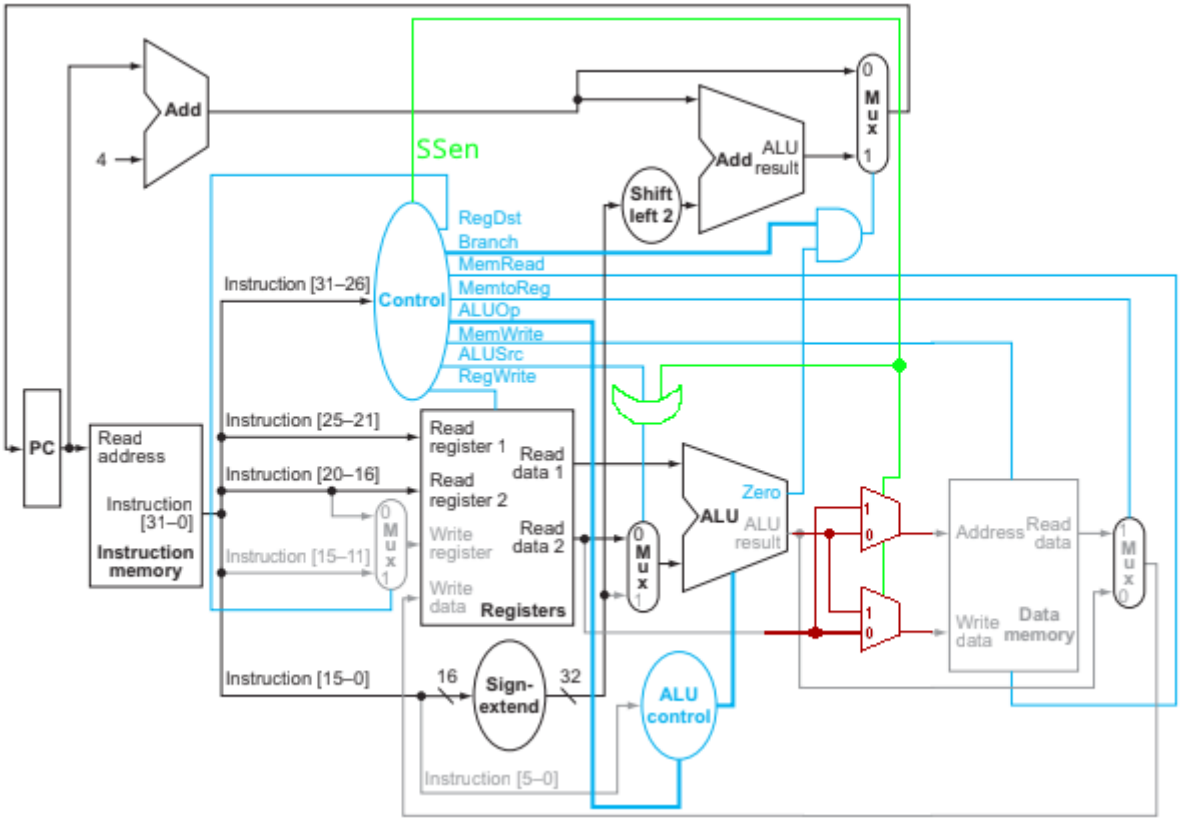
4.13.2: There are no modifications to be made in this situation; everything should work as intended with this new operation.

4.13.3: We need a new datapath for the address and the data. The data (the result of the summation between `rs` and `imm`) comes from the ALU output, but currently it only goes into the memory *address* input. We need it to go into the memory *data* input instead; this requires a multiplexer. Also, we need the `Read data 2` output from the registers to go into the memory *address* input (it currently only goes into the memory *data* input); this requires another mux. **Note that this design only works if `rt` is read from `Read data 2` and `rs` is read from `Read data 1`.** This is because `Read data 1` will be added to `imm`, and `Read data 2` will be the address. I understand that switching the order of the arguments is a little confusing, but this is necessary to avoid changing much of the pre-ALU design.

4.13.4: We only need one more control signal; let’s call it `SSen` (for “sum save enable”). It will be asserted when this new command is read and decoded. `SSen` is OR’d with `ALUSrc` in order to control the mux that selects the sign-extended immediate field instead of `Read data`

2 (which will be used for the address). SSen will feed into the select input for both muxes I described in 4.13.3.

4.13.5: Note: my added datapath is in red, and my added control path is in green.



Problem 4.15

4.15.1: From Problem 4.7, the latency of a load word operation is 850 ps. Subtracting any ALU operations from that, we reduce the time for a load operation to take place is 650 ps. Similarly, the time needed for a store operation is 575 ps; without ALU operations, that would only take 375 ps. The new clock cycle time would be 650 ps, which is a 23.5% decrease in time.

4.15.2: Assuming the original program distribution is 52% R-type and I-type (non-`lw`), 25% `lw`, 11% `sw`, and 12% `beq`, after the `lw`/`sw` split, the new distribution is 38.2% R- and I-type, 36.7% `lw`, 16.2% `sw`, and 8.8% `beq` (because the amount of `lw` and `sw` commands doubles). This means that there is a 36% total increase in the amount of commands. Multiplied with a 23.5% decrease in time, it means that by disallowing offsets in `lw` and `sw` commands, there is a 4.04% increase in the amount of time required to run the program. The program is (slightly) slower with the modification.

4.15.3: The primary factor is the amount of `lw`/`sw` commands in a program. Obviously, if you have a program with zero `lw`/`sw` commands, you will not experience any change. If your program consists of *only* `lw`/`sw` commands, you will double the number of instructions, and that will ruin your program's performance more than a small decrease in clock speed will help it.