Homework 3: Solution
EE 371 / CS 330 / CE 321: Computer Architecture

1. **Pipeline Basics [5 marks]**
In this exercise, we examine how pipelining affects the clock cycle time of the processor. Problems in this exercise assume that individual stages of the datapath have the following latencies:

IF	ID	EX	MEM	WB
200ps	300ps	150ps	350ps	250ps

Table 1: Pipeline Stage Times

Also, assume that instructions executed by the processor are broken down as follows:

R-type	beq	ld	sd	Total
40%	15%	20%	25%	100%

Table 2: Instruction Distribution

1. What is the clock cycle time in a pipelined and non-pipelined processor? [1 mark]

Solution:

Pipelined = **350ps** (worst case)
Non-pipelined = $200 + 300 + 150 + 350 + 250 =$ **1250ps**

2. What is the total latency of an `ld` instruction in a pipelined and non-pipelined processor? [1 mark]

Solution:

Pipelined = 350 * 5 = **1750ps**
Non-pipelined = **1250ps**
Pipelining uses more cycle time since one stage would now take 350 ps each.

3. If we can split one stage of the pipelined datapath into two new stages, each with half the latency of the original stage, which stage would you split and what is the new clock cycle time of the processor? [1 mark]

Solution:

We can split the **MEM** stage, since it has the highest latency. This will reduce the clock cycle time to 300 ps.

4. Assuming there are no stalls or hazards, what is the utilization of the data memory? [1 mark]

Solution:

Data memory is used for ld and sd instructions. Therefore, the utilization of the data memory is:

$$20 + 25 = \mathbf{45\%}$$

5. Assuming there are no stalls or hazards, what is the utilization of the write-register port of the "Registers" unit? [1 mark]

Solution:

Write register port for the register unit is used for ld and R-type instructions. Therefore, the utilization is:

$$40 + 20 = \mathbf{60\%}$$

2. **Pipeline Hazard Basics [5 marks]**
A hypothetical processor has 9 stages of a pipeline as shown in table below. The first row in the table below shows the pipeline stage number, second row gives the name of each stage, and third row gives the delay of each stage in Nano-seconds. The name of each stage describes the task performed by it. Each stage takes 1 cycle to execute. This processor stores all the register contents in a compressed fashion. After fetching the operands the operands are first decompressed, and before saving the results in register file, the results are first compressed.

1	2	3	4	5	6	7	8	9
Instruction Fetch	Instruction Decode	Operand Fetch	Decompress Operands	Instruction Execute 1	Instruction Execute 2	Memory Access	Compress results	Register Write back
1ns	1.4ns	1.1ns	2.2ns	2.3ns	2.3ns	1.3ns	2.2ns	1.2ns

1. How many cycles are required to implement/execute one instruction on this pipeline? [0.5 marks]

Solution:

Since each stage takes 1 cycle and there are 9 stages, a total of **9 cycles** are needed to execute one complete instruction.

2. How many cycles are required to execute 17 instructions on this pipeline? Assume that no stall cycles occur during the execution of all instructions. [0.5 marks]

Solution:

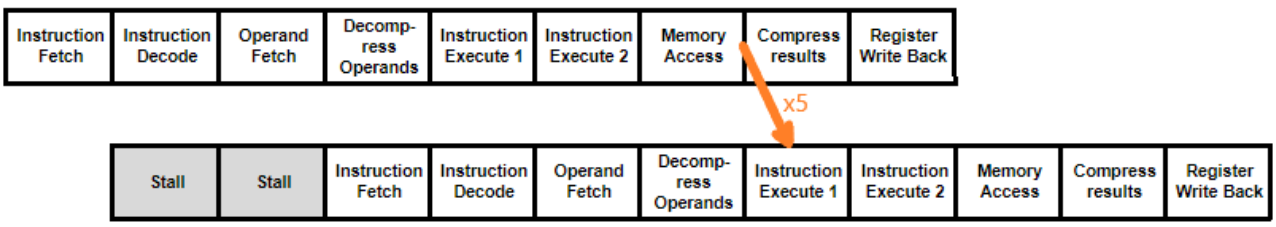
A total of **25 cycles** would be needed to execute 17 instructions. This was calculated using the fact that the first instruction takes 9 cycles and the remaining 16 need one cycle on top of the last executed instruction. Thus, ((9 + (1 * 16)) = 25) cycles are needed.

3. Assume that all necessary bypass circuitry is implemented in this 9 stage pipeline. How many cycles will the pipeline stall during the execution of the given two instructions? [0.5 marks]

```
x5 = Load from memory
x2 = x5 + x7
```

Solution:

2 NOPs or stalls would be needed such that when instruction x5 is in stage 8, instruction x2 is in stage 5. Due to pipelining and bypass circuitry, the value of x5 would be directly sent to the ALU of instruction x2. Forwarding would be done as follows:



4. Assume that no bypass circuitry is implemented in this 9 stage pipeline. How many cycles will the pipeline stall during the execution of the given two instructions? [0.5 marks]

```
x5 = x1 + x3
x2 = x5 + x7
```

Solution:

Assuming that operand fetch is the stage where data is read from the registers

A total of **6** stalls would be needed so that when instruction x5 writes the value for x5 back in the memory, instruction x2 can read it without conflict.
However, if writing to registers (register write back) and reading from registers (operand fetch) can take place in the same cycle, then a total of **5** stalls would be needed.

5. How much total time (in ns) is required to execute one entire instruction if the nine stages were not pipelined? [0.5 marks]

Solution:

15 ns
A load instruction uses all 9 of the stages so the cycle time of any instruction would be defined based on the cycle time for a load instruction which is 15ns.

6. What is the delay of 1 cycle (in ns) when all the nine stages are pipelined? [0.5 marks]

Solution:

The delay of 1 cycle would be **2.3ns** which is given by the slowest stage ie stage 5/6.

7. What is the total time (in ns) required to execute one entire instruction if the nine stages are pipelined? [1 marks]

Solution:

In a pipelined version, the cycle time for each stage would be the same ie it would 2.3ns per stage. In that case, the total time needed for one instruction would be **20.7ns** (2.3 * 9).

8. For each set of code given below, mention if forwarding circuit can avoid all the stalls in the code: [1 mark]

```
a. add  x1, x2, x3
   add  x6, x7, x8
   add  x4, x1, x5

b. ld  x1,0(x3)
   add  x2, x1, x3
```

Solution:

For the first instruction, stalls **can** be avoided using forwarding.
For the second set of instructions, the number of stalls can be reduced from 5 to 2 but they **cannot** be avoided.

3. **Pipeline Diagram [5 marks]**

Consider the following loop.

```
LOOP: ld x11, 8(x13)
      ld x10, 0(x13)
      add x12, x10, x11
      addi x13, x13, -16
      bne x12, x0, LOOP
```

Assume that perfect branch prediction is used (no stalls due to control hazards), that the pipeline has full forwarding support, and that branches are resolved in the EX (as opposed to the ID) stage.

1. Show a pipeline execution diagram for the first two iterations of this loop. [2.5 marks]

Solution:
... indicates a stall/bubble.
I have numbered the lines of the code and will use this numbering in the pipeline execution diagram.

1. LOOP: ld x11, 8(x13)
2. ld x10, 0(x13)
3. add x12, x10, x11
4. addi x13, x13, -16
5. bne x12, x0, LOOP

1	IF	ID	EX	ME	|	WB										
2		IF	ID	EX	|	ME	WB									
3			IF	ID	|	...	EX	ME	WB							
4				IF	|	...	ID	EX	ME	WB						
5					|	...	IF	ID	EX	ME	WB					
1								IF	ID	EX	ME	WB				
2									IF	ID	EX	ME	WB			
3										IF	ID	...	EX	|	ME	WB
4											IF	...	ID	|	EX	ME	WB
5															IF	ID	EX	ME	WB

2. Mark pipeline stages that do not perform useful work. How often while the pipeline is full do we have a cycle in which all five pipeline stages are doing useful work? [2.5 marks]

Solution:
! indicates a stage that does not do useful work.

1	IF	ID	EX	ME	|	WB										
2		IF	ID	EX	|	ME	WB									
3			IF	ID	|	...	EX	ME!	WB							
4				IF	|	...	ID	EX	ME!	WB						
5					|	...	IF	ID	EX	ME!	WB!					
1								IF	ID	EX	ME	WB				
2									IF	ID	EX	ME	WB			
3										IF	ID	...	EX	|	ME!	WB
4											IF	...	ID	|	EX	ME!	WB
5															IF	ID	EX	ME!	WB!

In a clock cycle, a pipeline stage is not performing useful work if it is being stalled or if an instruction at a particular stage is not doing any useful work.

4. **Pipeline Design Optimization for Memory Access [7 marks]**

`ld` is the instruction with the longest latency on the RISC-V implementation of single-cycle non-pipelined processor discussed in class. If we modified ld and sd so that there was no offset (i.e., the address to be loaded from/stored to must be calculated and placed in rs1 before calling ld/sd), then no instruction would use both the ALU and Data memory. This would allow us to reduce the clock cycle time. However, it would also increase the number of instructions, because many ld and sd instructions would need to be replaced with ld/add or sd/add combinations.

1. What would the new clock cycle time be for the non-pipelined processor? [1 mark]

Solution:

We will assume the latencies provided in Question 2.
The new clock cycle time would be 1100 ps since now the the ALU and Data Memory will operate in parallel. This means we can remove the time for faster of the two i.e. the time for ALU (150ps).

$$1250 - 150 = 1100\text{ps}$$

2. In the non-pipelined version, would a program with the instruction mix presented in Question 1 run faster or slower on this new CPU? By how much? (For simplicity, assume every ld and sd instruction is replaced with a sequence of two instructions.) [1 mark]

Solution:

Every instruction was initially taking $1250 \times n$ ps, where n is the number of instructions. The same program would have

$$\frac{40 + 15 + (20 \times 2) + (25 \times 2)}{100} = 1.45 \times n \text{ instructions}$$

when it is compiled for the modified processor. Therefore the time taken by the new processor is

$$1100 \times 1.45 \times n = 1595 \cdot n\text{ps}$$

We can say that program with the instruction mix presented in question runs **slower** on this new CPU. The speed up can be calculated by

$$\frac{1250}{1595} = \mathbf{0.783}$$

3. What is the primary factor that influences whether a program will run faster or slower on the new CPU? [1 mark]

Solution:

The time for the program to run depends on the number of ld/sd instructions. Also, a program whose loads and stores tend to be to only a few different address may also run faster on the modified CPU.

4. Do you consider the original CPU a better overall design, or do you consider the new CPU a better overall design? Why? [1 mark]

Solution:

I consider the original design better since it is faster compared to the modified one in case of a single cycle processor. Nevertheless, if we consider the pipelined processor, there is no change the modification has no effect on the clock cycle time.

5. As a result of the change, the MEM and EX stages of the pipelined version of the processor can be overlapped and the pipeline has only four stages. How will the reduction in pipeline depth affect the cycle time? [1 mark]

Solution:

There will be no affect on the clock cycle time since we are not making any changes to the stage with the highest latency

6. How might this change improve the performance of the pipeline? [1 mark]

Solution:

Running EX and MEM stages in paralles can effectively reduce the number of stalls as it will eliminate the need for a cycle or NOPs in case the program has a load instrcution followed by the instruction that use the resultant loaded register value

7. How might this change degrade the performance of the pipeline? [1 mark]

Solution:

As discussed in previous parts we are replacing the ld instruction with addi and ld and sd instruction with addi and sd. This will increase the number of instructions. This will consequently degrade the performance of the pipeline.

5. Pipeline Design for New Instructions [50 marks]

Consider the following instructions that are not found in the RISC-V architecture:

i. Load Word Register
lwr rd, rs2(rs1) // Reg[rd] = Mem[Reg[rs1] + Reg[rs2]]

ii. Add 3 operands
add3 rd, rs1, rs2, rs3 // Reg[rd] = Reg[rs1] + Reg[rs2] + Reg[rs3]

iii. Add to Memory
addm rd, Offset(rs) // Reg[rd] = Reg[rd] + Mem[Offset + Reg[rs]]

iv. Branch Equal to Memory
beqm rs1, offset(rs2), rs3 // if(Reg[rs1]==Mem[Offset+Reg[rs2]]) PC = PC + Reg[rs3]

v. Store Word and Increment
swinc rs2, offset(rs1) // Mem[Reg[rs1] + offset]= Reg[rs2], Reg[rs1] = Reg[rs1] + 4

We want to modify the RISC-V processor to support the above instructions. For parts (b) and (d) below, you can use a printed version of the figures in the book over which you can draw your suggested modifications (no need to draw the entire diagram from scratch). For each of the above instructions, do the following:

(A) Suggest if any of the existing instruction formats is a good choice to encode the new instruction. If not, then propose a new instruction format.

(B) Modify the datapath and control signals of the single-cycle RISC-V processor (Figure 4.17 of the book) to execute the new instruction using the instruction format suggested in part (a). Use the minimal amount of additional hardware and clock cycles/ control states. Remember when adding new instructions, don't break the operation of the standard ones.

(C) Discuss the effect of the modification in part (b) on the latency of single-cycled non-pipelined CPU

(D) Discuss if the suggested modification in part (b) should be handled by increasing/decreasing the number of pipelining stages that were discussed in class. Draw a pipelined version of the new processor similar to Figure 4.49 of the book.

(E) Disucss if any new types of data hazards are introduced due to the new instruction? If yes, can they be mitigated through forwarding? Use a mutli-cycle pipeline diagram like Figure 4.51 of the book to illustrate the new forwarding paths.

(F) Discuss, based on the above analysis, why the new instruction was not made part of the RISC-V architecture.

1. **Load Word Register**

Solution:

(A) **Suggest if any of the existing instruction formats is a good choice to encode the new instruction. If not, then propose a new instruction format.**

The format of an R-type instruction can be used to encode Load Word Register instruction

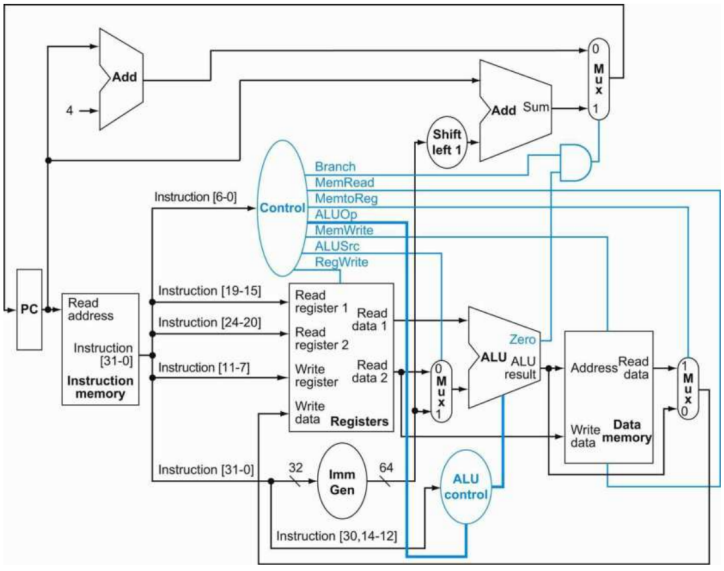
funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

(B) **Modify the datapath and control signals of the single-cycle RISC-V processor (Figure 4.17 of the book) to execute the new instruction using the instruction format suggested in part (A). Use the minimal amount of additional hardware and clock cycles/control states. Remember when adding new instructions, don't break the operation of the standard ones.**

The basic structure of the processor stays the same. ALU takes the first input from ReadData1 and the second from ReadData2. The computed sum is then given to the DataMemory as the address of the data to be loaded. This data is then passed on to the MUX and then to writedata is the Register block where the data is written in rd. The values of the control lines would be as follows:

- Branch = 0
- MemRead = 1
- MemtoReg = 1
- ALUOP = 00
- MemWrite = 0
- ALUSrc = 0
- RegWrite = 1

The processor after the addition of this instruction would look the same:



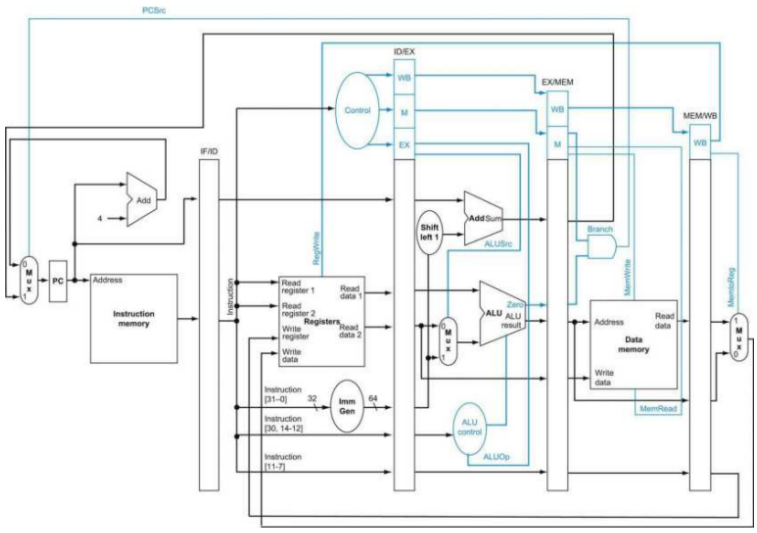
(C) **Discuss the effect of the modification in part (B) on the latency of single-cycle non- pipelined CPU**

No major components were added only the values for the control signals changed so there would be no change in the latency.

(D) **Discuss if the suggested modification in part (B) should be handled by increasing/decreasing the number of pipelining stages that were discussed in class. Draw a pipelined version of the**

new processor similar to Figure 4.49 of the book.

No change is required in the pipelined version.
The pipelined version would look like the one given below:



(E) **Discuss if any new types of data hazards are introduced due to the new instruction? If yes, can they be mitigated through forwarding? Use a multi-cycle pipeline diagram like Figure 4.51 of the book to illustrate the new forwarding paths.**

No new data hazards would occur.

(F) **Discuss, based on the above analysis, why the new instruction was not made part of the RISC-V architecture.**

The new instruction actually gives a benefit when the index of an array is stored in a variable (which is commonly the case in loops), this new instruction would allow us to save the effort of having to add it to the base register before 'ld' instruction. Hence instruction count in this case is going down, and performance is improving.
Although, the argument against that can be that instead of array index based implementation, we can replace the loop implementation with a pointer based implementation, where the need to perform that addition goes away.
So, since the use case for index variable can be taken care of at compiler level, they probably deemed it unnecessary to add an instruction to their RISC ("reduced" instruction set computer) architecture, whose design objective is to minimize the number of instructions in the architecture.

2. **Add 3 operands**

Solution:

(A) **Suggest if any of the existing instruction formats is a good choice to encode the new instruction. If not, then propose a new instruction format.**

There is no existing instruction format for this instruction. A possible new instruction format can be:

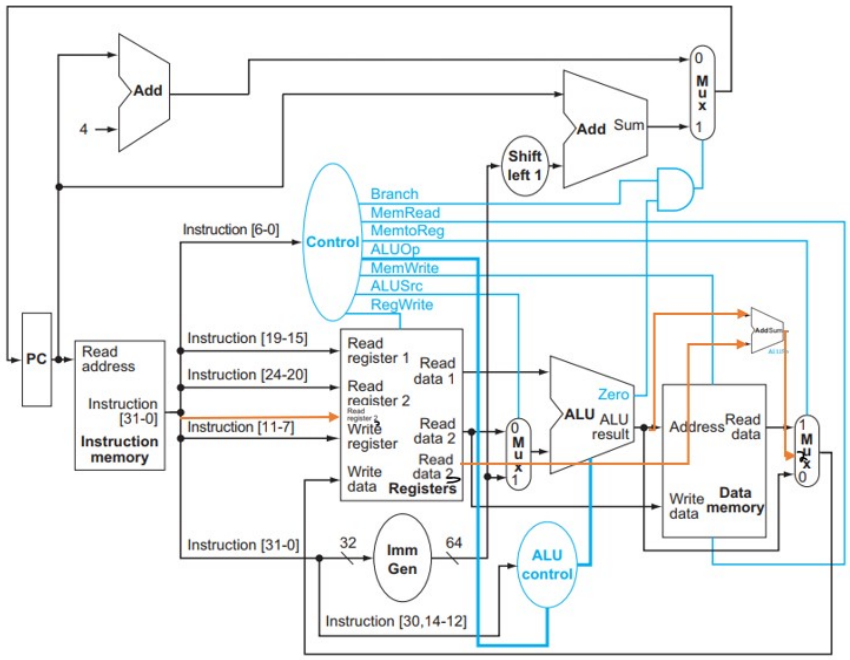
funct2	rs3	rs2	rs1	funct3	rd	opcode
2 bits	5 bits	5 bits	5 bits	3 bits	5 bits	7 bits

(B) **Modify the datapath and control signals of the single-cycle RISC-V processor (Figure 4.17 of the book) to execute the new instruction using the instruction format suggested in part (A).**

Use the minimal amount of additional hardware and clock cycles/control states. Remember when adding new instructions, don't break the operation of the standard ones.

One additional adder is added and we increase the register reading port from 2 to 3 in the register file. Since for this instruction, the output of the adder that we added has to be written in register file, we have to increase the number of bits of the control signal MemtoReg. MemtoReg now chose between three option: Writing the output of AlU to registerfile, writing the output of the Read Data port of data memory and writing the output of the adder that we have added particularly for this instruction. The basic working of the processor, however, stays unchanged.

- Branch = 0
- MemRead = 0
- MemtoReg = 10
- ALUOP = 00
- MemWrite = 0
- ALUSrc = 0
- RegWrite = 1

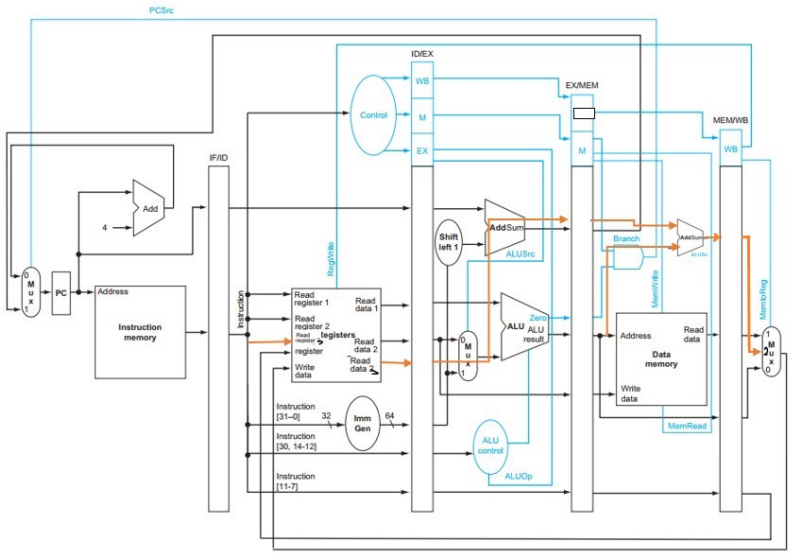


(C) **Discuss the effect of the modification in part (B) on the latency of single-cycle non- pipelined CPU**

The latency will not increase due to increase in the number of reading ports of the register file. This is because reading is done in parallel.
However, there is an additional adder now. Since the adder is in parallel with memory access the worst case latency is still the same.

(D) **Discuss if the suggested modification in part (B) should be handled by increasing/decreasing the number of pipelining stages that were discussed in class. Draw a pipelined version of the new processor similar to Figure 4.49 of the book.**

Even though we adding an adder, the addition can be done in parallel with MEM stage. Hence, there will be no new pipeline stage

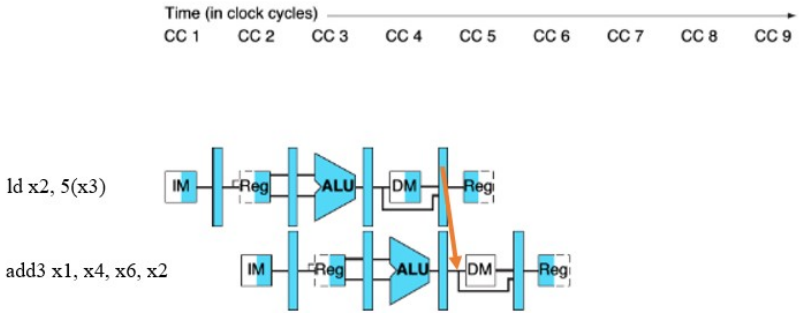


(E) **Discuss if any new types of data hazards are introduced due to the new instruction? If yes, can they be mitigated through forwarding? Use a multi-cycle pipeline diagram like Figure 4.51 of the book to illustrate the new forwarding paths.**

Lets take this arbitrary set of instructions that highlight the new data hazards that might occour:

ld x2, 5(x3)
add3 x1, x4, x6, x2

We can call this data hazard MEM/WB to ALU 3rd arg (or, more precisely, 1st arg of the new Adder) where third operand of add3 is dependent on destination register of a preceeding ld instruction. This hazard can be mitigated using a forwarding path between the MEM/WB pipeline register to the input of the new adder which is located in the MEM stage (shown below).



(F) **Discuss, based on the above analysis, why the new instruction was not made part of the RISC-V architecture.**

Adding significant additional hardware like 64-bit adder, and a new forwarding unit, is not really providing much improvement in performance. Existing pipeline and forwarding, already allows two add instructions to be executed in pretty much the same amount of time. So this new instruction and new hardware added to support it does not offer any significant improvement in performance.

3. **Add to Memory**

Solution:

(A) **Suggest if any of the existing instruction formats is a good choice to encode the new instruction. If not, then propose a new instruction format.**

I-format can be used for the given instruction:

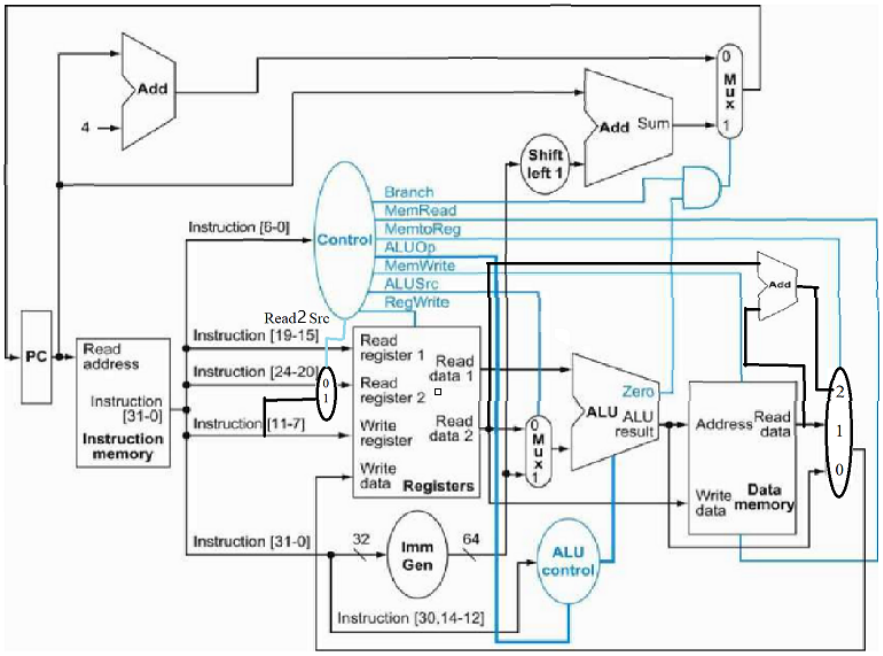
immediate	rs1	func3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

(B) **Modify the datapath and control signals of the single-cycle RISC-V processor (Figure 4.17 of the book) to execute the new instruction using the instruction format suggested in part (A). Use the minimal amount of additional hardware and clock cycles/control states. Remember when adding new instructions, don't break the operation of the standard ones.**

The basic working of the processor stays unchanged. An additional MUX is added at the input of ReadRegister2. This is done because there is no explicit reference to source register 2 here, rather the destination register serves as both an rd and rs2. The MUX controls whether to pass rd (in this case) or rs2 in other cases. An additional control line called Read2Src is added which controls this MUX. An additional ALU which functions as an adder is added which takes the value from rs2 and the data loaded from the memory and adds them. This output is then sent to the MemtoReg MUX. This MUX is also modified from a 2 input MUX to a 3 input MUX where the output from the adder is given as the 2nd input. The bits of the signal MemtoReg are also changed to 2 now. The values of the control lines would be as follows:

- Branch = 0
- MemRead = 1
- MemtoReg = 10
- ALUOP = 00
- MemWrite = 0
- ALUSrc = 1
- RegWrite = 1
- Read2Src = 1

The processor would like something like this after this instruction is added:

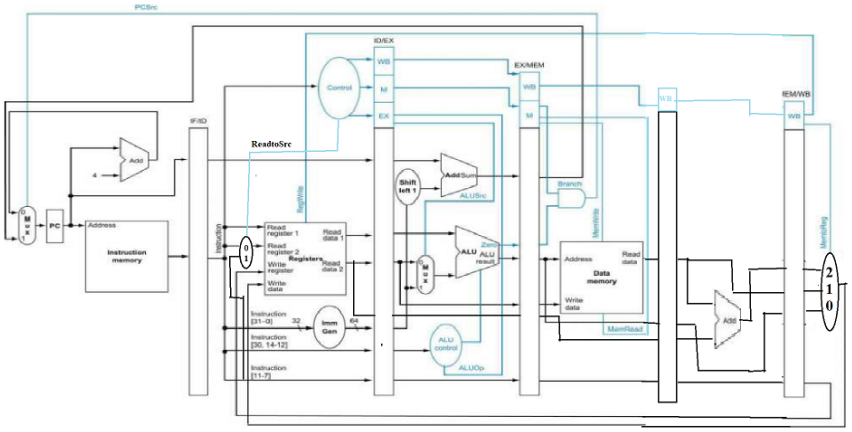


(C) **Discuss the effect of the modification in part (B) on the latency of single-cycle non- pipelined CPU**

Because of the addition of an extra Adder, the time needed for an instruction to completely execute would now increase.

(D) **Discuss if the suggested modification in part (B) should be handled by increasing/decreasing the number of pipelining stages that were discussed in class. Draw a pipelined version of the new processor similar to Figure 4.49 of the book.**

An extra stage can be added for the second ALU. The new MUX is added in the IF stage.



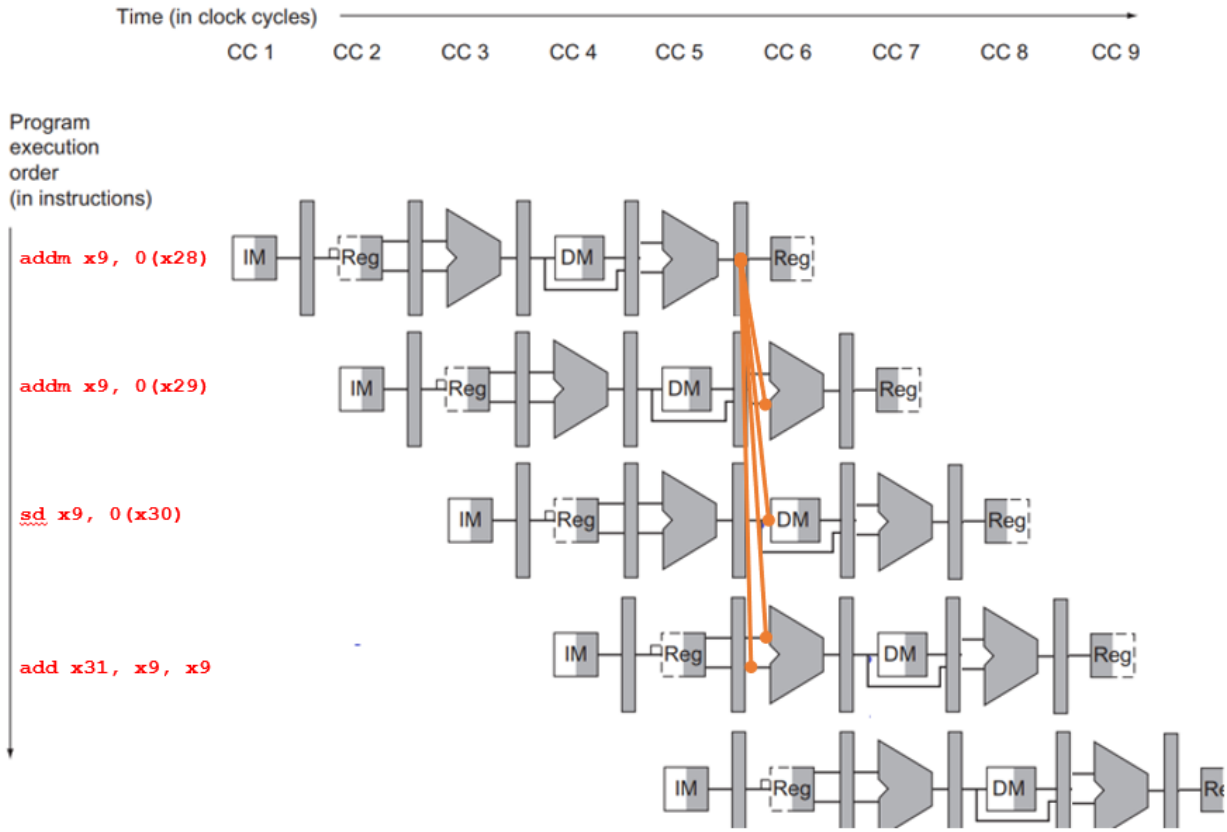
(E) **Discuss if any new types of data hazards are introduced due to the new instruction? If yes, can they be mitigated through forwarding? Use a multi-cycle pipeline diagram like Figure 4.51 of the book to illustrate the new forwarding paths.**

Four new kinds of data hazards are created due to the introduction of a new pipeline register, and new adder which consumes register value. Four forwarding paths will be needed from the new pipeline register to the four points in our pipeline which consume register data.

The following code sequence illustrates the data hazards.

```
addm x9, 0(x28)
addm x9, 0(x29)
sd x9, 0(x30)
add x31, x9, x9
```

The following figure illustrates the forwarding paths needed.



(F) **Discuss, based on the above analysis, why the new instruction was not made part of the RISC-V architecture.**

This instruction is a combination of two already present instructions. Implementing this instruction needed additional hardware and an additional pipeline stage which increases the overall latency of the processor. Using the already present instructions (addi and store) would be efficient.

4. **Branch Equal to Memory**

Solution:

(A) **Suggest if any of the existing instruction formats is a good choice to encode the new instruction. If not, then propose a new instruction format.**

There is no existing instruction format for this instruction. A possible new instruction format can be:

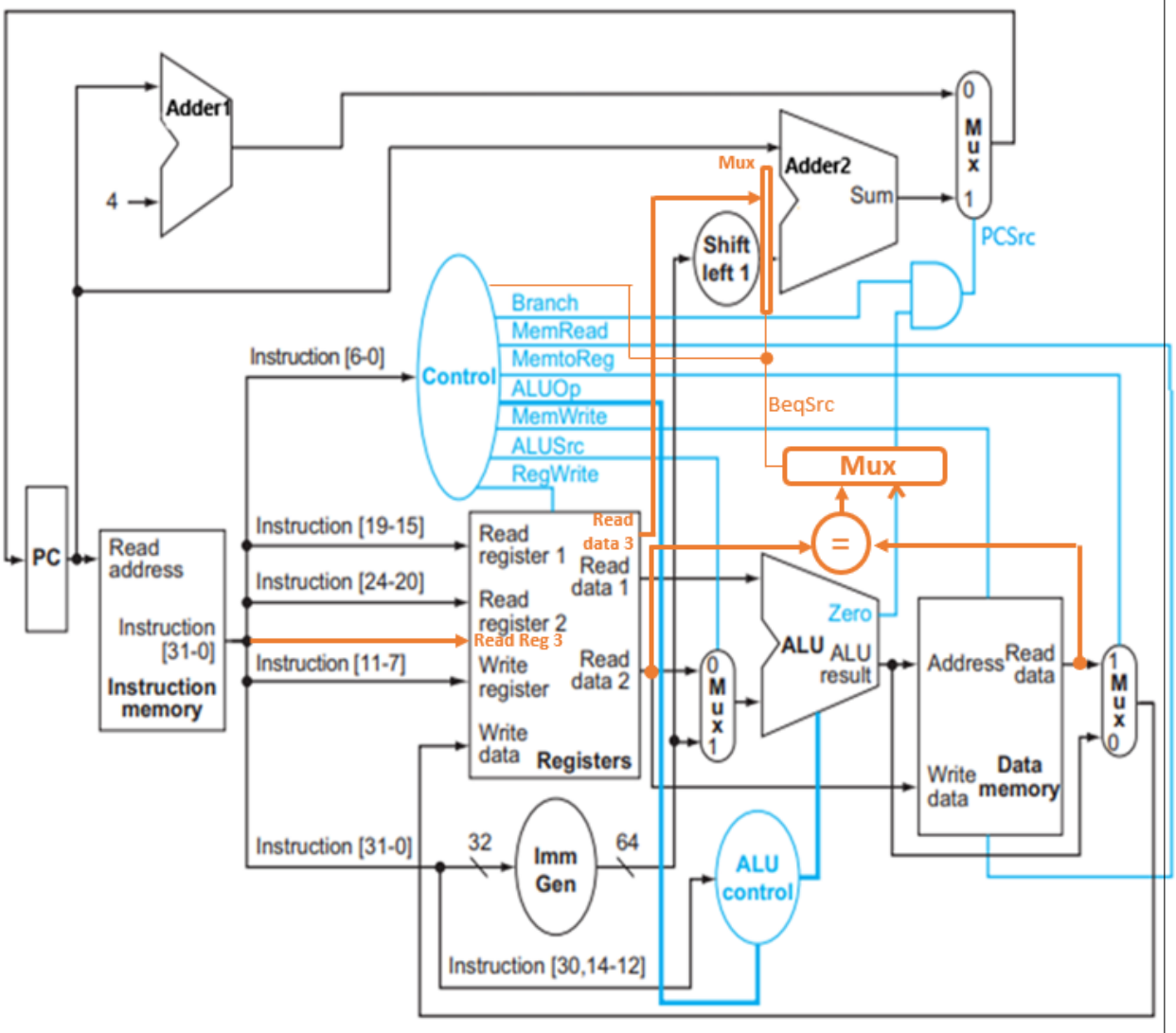
imm[9:3]	rs1	rs2	imm[2:0]	rs3	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Note that we have swapped the standard places of rs1 and rs2 in this format. This is because the given instruction uses rs1 for purposes that standard instructions use rs2 for. Same goes for rs2. So swapping places in the instruction format will avoid the need to deal with this irregularity using hardware modifications.

(B) **Modify the datapath and control signals of the single-cycle RISC-V processor (Figure 4.17 of the book) to execute the new instruction using the instruction format suggested in part (A). Use the minimal amount of additional hardware and clock cycles/control states. Remember when adding new instructions, don't break the operation of the standard ones.**

Equality checker is added and we increase the register reading port from 2 to 3 in the register file. We have also added two additional muxes. One mux is to chose between normal branching and branch equal to memory instruction. The second mux is to chose between output of ReadData3 and that of ShiftLeft1 to be sent to the adder. An additional control signal which I have called 'BeqSrc' is added and it is sent as a selection line to the added multiplexors. Also, branch control will now be asserted in both beq and beqm instructions.

- Branch = 1
- MemRead = 1
- MemtoReg = X
- ALUOP = 00
- MemWrite = 0
- ALUSrc = 1
- RegWrite = 0
- BeqSrc = 1

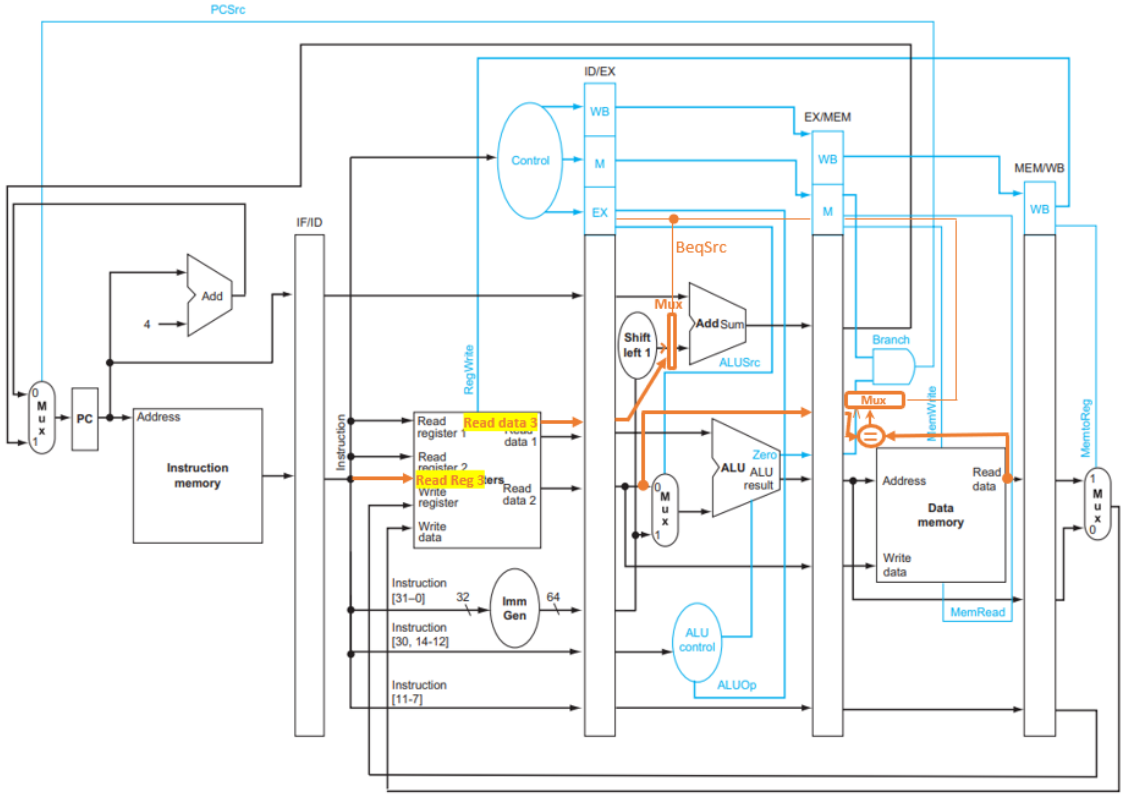


(C) **Discuss the effect of the modification in part (B) on the latency of single-cycle non- pipelined CPU**

The comparator circuit is dependent on the output of the memory so this comparison is not happening in parallel with memory access. Hence we have a new worst-case latency instruction (beqm) in the single-cycle processor. Cycle time for all instructions will have to be slightly increased to deal with this.

(D) **Discuss if the suggested modification in part (B) should be handled by increasing/decreasing the number of pipelining stages that were discussed in class. Draw a pipelined version of the new processor similar to Figure 4.49 of the book.**

Although the comparator check is done sequentially after memory access, it is a fast operation, so no need to make another pipeline stage out of it.

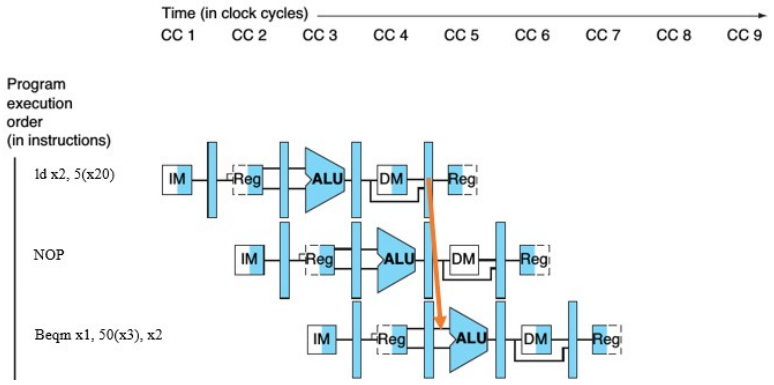


(E) **Discuss if any new types of data hazards are introduced due to the new instruction? If yes, can they be mitigated through forwarding? Use a multi-cycle pipeline diagram like Figure 4.51 of the book to illustrate the new forwarding paths.**

There can be two novel data hazards. Let us take an arbitrary set of instructions.

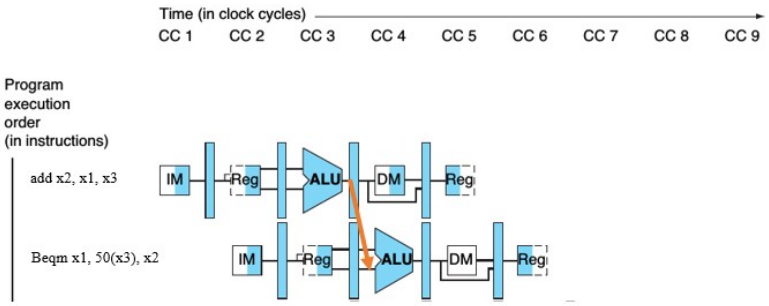
ld x2, 5(x20)
beqm x1, 50(x3), x2

The third operand i.e x2 of beqm is dependent on the destination register of the preceding lood instruction. This hazard cannot be mitigated using forwarding alone. We to have insert a NOP instead as show in the diagram below. This hazard can be called MEM/WEB to adder.



Another sample instruction set can be as follows.
add x2, x1, x3
beqm x1, 50(x3), x2

This hazard can simply be mitigated by forwarding. In the above case, the third operand x2 of beqm is dependent on the destination register of the preceding add instruction. The data hazard can be called EX/MEM to adder and a forwarding path between EX/MEM and adder can solve the hazard as demonstrated below:



(F) **Discuss, based on the above analysis, why the new instruction was not made part of the RISC-V architecture.**

The additional comparator will increase the latency of memory access stage which is already the worst stage hence the clock-cycle time for the pipeline goes up, thus degrading performance. Since control decision can't be moved to the ID stage, the cost of control hazards is going up. Finally, the forwarding from MEM/WB to ALU can already implement the beq followed by a load pretty fast in the pipelined case. So the sacrifice in clock-cycle time by combining load and beq is not really offering much benefit.

5. **Store Word and Increment**

Solution:

(A) **Suggest if any of the existing instruction formats is a good choice to encode the new instruction. If not, then propose a new instruction format.**

S format can be used for the given instruction:

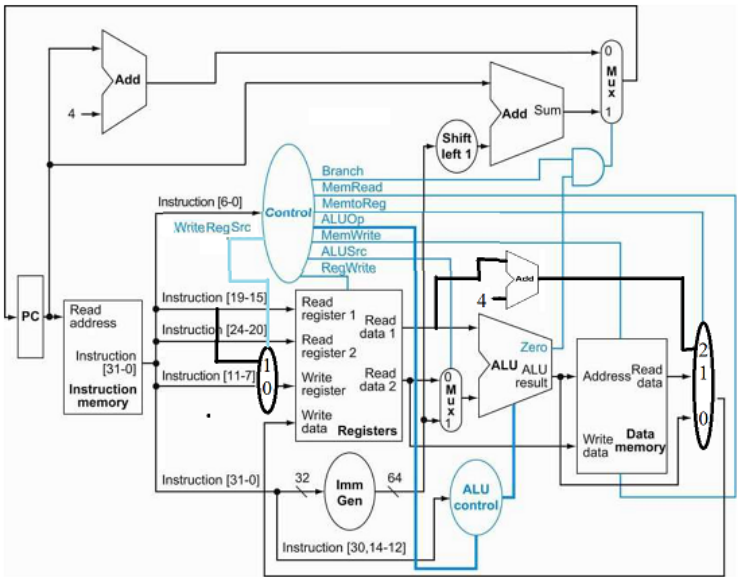
immediate[11:5]	rs2	rs1	func3	immediate[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

(B) **Modify the datapath and control signals of the single-cycle RISC-V processor (Figure 4.17 of the book) to execute the new instruction using the instruction format suggested in part (A). Use the minimal amount of additional hardware and clock cycles/control states. Remember when adding new instructions, don't break the operation of the standard ones.**

This instruction is a combination of a store and an addi instruction. The basic working of the processor for a store instruction is used to cater the first part of this instruction ie Mem[Reg[rs1] + offset]= Reg[rs2]. For the second part ie Reg[rs1] = Reg[rs1] + 4, an additional ALU is added. This ALU takes ReadData1 given by rs1 and the value 4 and passes the computed value on to the MUX which decides which data is to be written back in the Register Block. The MUX is modified from a 2 input MUX to a 3 input MUX where input 0 is the data loaded from Data Memory, input 1 is the output of the already present ALU and input 2 is the output of the new ALU. The control signal ie MemtoReg is set to 2 so that the new computed value of rs1 can be sent back to the Register Block for writing. A new MUX is added to decide which register to write to. The values of the control lines would be as follows:

- Branch = 0
- MemRead = 0
- MemtoReg = 10
- ALUOP = 00
- MemWrite = 1
- ALUSrc = 1
- RegWrite = 1
- WriteRegSrc = 1

The processor would now look like this:

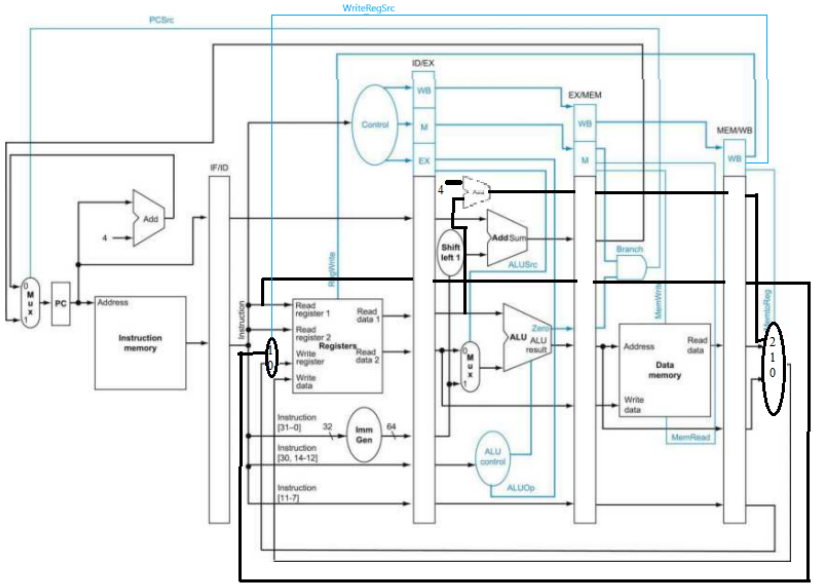


(C) **Discuss the effect of the modification in part (B) on the latency of single-cycle non- pipelined CPU**

The addition of MUX would increase the latency. Since the ALU is working in parallel, it won't effect the latency.

(D) **Discuss if the suggested modification in part (B) should be handled by increasing/decreasing the number of pipelining stages that were discussed in class. Draw a pipelined version of the new processor similar to Figure 4.49 of the book.**

The new ALU that has been added can be adjusted in the already existing EX stage. This would not lead to any conflicts because the first part of the instruction requires the previous value and the new value is not written until the writeback stage Thus, both the ALUs are independent and can work simultaneously under the execution stage and hence no new stage is needed. The cycle time would not change either because both the ALUs take the same amount of time t compute the results. in short, no major changes are to be made for a pipelined version of the modified processor.
The modified pipelined processor is as follows:



(E) **Discuss if any new types of data hazards are introduced due to the new instruction? If yes, can they be mitigated through forwarding? Use a multi-cycle pipeline diagram like Figure 4.51 of the book to illustrate the new forwarding paths.**

No new type of data hazard is created. The register value consumed by the new adder can be supplied by the output of the existing mux that is used for forwarding to ALU 1st arg.

(F) **Discuss, based on the above analysis, why the new instruction was not made part of the RISC-V architecture.**

Addition of mux is increasing the latency of register read, especially because of the the control signal needed for choosing the mux channel, because that signal will not be available before the "Control" unit produces its output. If the two operations (store and increment) are performed in sequence instead of combining them into one instruction, the operation will execute very fast in a pipelined case due to forwarding. So adding new hardware (adder) and latency is a sacrifice that is not resulting in significant performance improvements.

6. **Structural Hazard Analysis [5 marks]**
Consider the fragment of RISC-V assembly below:
ld x31, 32(x10)
sd x11, 8(x10)
sub x12, x14, x13
add x15, x12, x13
beq x24, x0, label
add x31, x31, x14
Suppose we modify the pipeline so that it has only one memory (that handles both instructions and data). In this case, there will be a structural hazard every time a program needs to fetch an instruction during the same cycle in which another instruction accesses data.

(i) Draw a pipeline diagram to show where the code above will stall.

Solution:
... indicates a stall/bubble.
I have numbered the lines of the code and will use this numbering in the pipeline execution diagram.
1 ld x31, 32(x10)
2 sd x11, 8(x10)
3 sub x12, x14, x13
4 add x15, x12, x13
5 beq x24, x0, label
6 add x31, x31, x14

1	IF	ID	EX	ME	WB						
2		IF	ID	EX	ME	WB					
3			IF	ID	EX	ME!	WB				
4				...	...	IF	ID	EX	ME!	WB	
5							IF	ID	EX	ME!	WB!
6								IF	ID	EX	ME!	WB

(ii) In general, is it possible to reduce the number of stalls/NOPs resulting from this structural hazard by reordering code?

Solution:
In any case, there will be a structural hazard every time a program needs to fetch an instruction during the same cycle in which another instruction accesses data. Reordering the code will simply change the instructions in conflict. Number of stalls won't be reduced since every instruction has to be fetched. This means that data access for ever instruction results in a stall.

(iii) Must this structural hazard be handled in hardware? We have seen that data hazards can be eliminated by adding NOPs to the code. Can you do the same with this structural hazard? If so, explain how. If not, explain why not.

Solution:
Since NOP instructions also have to be fetched from instruction memory, they will also cause structural hazards. Therefore, we cannot solve a structural hazzard using NOPs at the code level.
So it must be handled by the hardware. We will need a hazard detection unit needs to insert stalls.

(iv) Approximately how many stalls would you expect this structural hazard to generate in a typical program? Use the instruction mix from Question 2.

Solution:
ld and sd instructions access memory. Hence,

$$20 + 25 = 45\%$$

stalls are expected.

7. 12 points A 64-bit RISC-V pipeline is shown in Fig. a. Only execute stage and relevant inter-stage registers are shown. The clock rate of this RISC-V is decided by the execute stage, which is dominated by ALU's propagation delay. We are using this pipeline to implement 3 operand R-format instructions by cascading two ALUs as shown in Fig. b. Assume we have extended the instruction encoding and we are only focusing on pipeline design here. We can achieve cascading of ALUs in two ways as shown in Fig. b and Fig. c. In Fig. c, the execute stage is split into two stages, named EXE1 and EXE2.

1: If a single ALU unit has a propagation delay of 150 psec, compute pipeline cycle time for Fig. a, Fig. b and Fig. c.

Solution: Fig. a → 150 psec, Fig. b → 300 psec and Fig. c → 150 psec

2: Assuming "Register 1", "Register 2", and "Register 3" represent values of the registers, whereas PC and immediate

are 64-bit. Calculate the width of the inter-stage pipeline register for Fig. b (ID/EXE and EXE/MEM) and Fig. c (ID/EXE1, EXE1/EXE2, and EXE2/MEM). You do not need to consider control signals for this part.

Solution: Fig. b; ID/EXE → 64*5 + 5 (rd), EXE/MEM → 64 + 64 + 1 + 5
Fig. c; ID/EXE1 → 64*5 + 5 (rd), EXE1/EXE2 → 64*3 + 5 + 1, EXE2/MEM → 64 + 64 + 1 + 5

3: Assume both ALUs in Fig. b and Fig. c always perform the same operation. Using the assumption, identify and write names of the control signals for Fig. b (execute stage) and Fig. c (execute 1 and execute 2 stages).

Solution: Fig. b; EXE stage → ALUop (x2 bits, same) + ResSelect (1-bit) + ALUSrc (1-bit)
Fig. c; EXE1 stage → ALUop (x2 bits) + ALUSrc (1-bit)
EXE2 stage → ALUop (x2 bits) + ResSelect (1-bit)

4: For Fig. b write the hazard detection and forwarding conditions

Solution: Fig. b; Detection:

- 1a: EX/MEM.ResgiterRd = ID/EX.RegisterRs1
- 1b: EX/MEM.ResgiterRd = ID/EX.RegisterRs2
- 1c: EX/MEM.ResgiterRd = ID/EX.RegisterRs3
- 2a: MEM/WB.ResgiterRd = ID/EX.RegisterRs1
- 2b: MEM/WB.ResgiterRd = ID/EX.RegisterRs2
- 2c: MEM/WB.ResgiterRd = ID/EX.RegisterRs3

Forwarding: Figure 4.53 from book with ForwardC addition for operand 3, same as ForwardA and ForwardB

5. Compare the two implementations in terms of register size and pipeline clock-speed.

Solution: Register size (chip-area) vs clock-rate trade-off. Also, more stages mean deeper pipeline and costly stalls.

6. Which implementation do you think is better and why? (Max 5 sentences.)

Solution: The implementation with both ALUs in the same stage is better due to its simpler structure, potential for higher clock speeds, and avoidance of costly stalls associated with a deeper pipeline.

8. **Forwarding Logic Design Trade-offs [5 marks]**
This exercise is intended to help you understand the cost/complexity/performance trade-offs of forwarding in a pipelined processor. Problems in this exercise refer to pipelined datapaths from Figure 4.53 of the book. These problems assume that, of all the instructions executed in a processor, the following fraction of these instructions has a particular type of read-after-write (RAW) data dependence.

The type of RAW data dependence is identified by the stage that produces the result (EX or MEM) and the next instruction that consumes the result (1st instruction that follows the one that produces the result, 2nd instruction that follows, or both). We assume that the register write is done in the first half of the clock cycle and that register reads are done in the second half of the cycle, so "EX to 3rd" and "MEM to 3rd" dependences are not counted because they cannot result in data hazards. We also assume that branches are resolved in the EX stage (as opposed to the ID stage), and that the CPI of the processor is 1 if there are no data hazards.
Assume the following latencies for individual pipeline stages. For the EX stage, latencies are given separately for a processor without forwarding and for a processor with different kinds of forwarding.

(i) For each RAW dependency listed above, give a sequence of at least three assembly statements that exhibits that dependency.

Solution:

EX to 1st only:

```
add x5, x6, x7
add x9, x5, x8
add x10, x11, x12
```

The first instruction writes some new value in x5. This register is also needed by the second instruction. Because of pipelining, a data hazard would occur in reading register x5.

MEM to 1st only:

```
ld x5, 40(x6)
add x9, x5, x8
add x10, x11, x12
```

The first instruction loads some data from the memory to x5. x5 is also needed by the second instruction to perform addition. This creates a data hazard.

EX to 2nd only:

```
add x5, x6, x7
add x9, x8, x10
add x11, x5, x12
```

The first instruction writes some new value in x5. This register is also needed by the third instruction. Because of pipelining, a data hazard would occur while reading the data from x5.

MEM to 2nd only:

```
ld x5, 40(x6)
add x9, x8, x7
add x10, x5, x12
```

The first instruction loads some data from the memory to x5. x5 is also needed by the third instruction to perform addition. This creates a data hazard.

EX to 1st and EX to 2nd:

```
add x5, x6, x7
add x9, x5, x10
add x11, x5, x12
```

The value in register x5 is written by the first instruction. The new value is also needed by the second and the third instruction, thus creating a conflict ie data hazard.

(ii) For each RAW dependency above, how many NOPs would need to be inserted to allow your code from (i) to run correctly on a pipeline with no forwarding or hazard detection? Show where the NOPs could be inserted.

Solution:

EX to 1st only: (2 NOPs)

```
add x5, x6, x7
NOP
NOP
add x9, x5, x8
add x10, x11, x12
```

MEM to 1st only: (2 NOPs)

```
ld x5, 40(x6)
NOP
NOP
add x9, x5, x8
```

add x10, x11, x12

EX to 2nd only: (1 NOP)
add x5, x6, x7
add x9, x8, x10
NOP
add x11, x5, x12

MEM to 2nd only: (1 NOP)
ld x5, 40(x6)
add x9, x8, x7
NOP
add x10, x5, x12

EX to 1st and EX to 2nd: (2 NOPs)
add x5, x6, x7
NOP
NOP
add x9, x5, x10
add x11, x5, x12

(iii) Analyzing each instruction independently will over-count the number of NOPs needed to run a program on a pipeline with no forwarding or hazard detection. Write a sequence of three assembly instructions so that, when you consider each instruction in the sequence independently, the sum of the stalls is larger than the number of stalls the sequence actually needs to avoid data hazards.

Solution:

ld x5, 0(x2)
add x9, x8, x10
add x11, x5, x9

In the above line of code, the lines 1 and 3 have a MEM to 2nd RAW dependency which needs one stall (1 NOP) and the lines 2 and 3 have an EX to 1st RAW dependency which needs two stalls (2 NOPs). So, a total of three stalls (3 NOPs) is needed if each instruction is analyzed independently. However, when looking at them althogether, only two stalls (2 NOPs) are needed.

ld x5, 0(x2)
add x9, x8, x10
NOP
NOP
add x11, x5, x9

(iv) Assuming no other hazards, what is the CPI for the program described by the table above when run on a pipeline with no forwarding? What percent of cycles are stalls? (For simplicity, assume that all necessary cases are listed above and can be treated independently.)

Solution:
Number of stalls per instruction as per part (b) = (0.10 * 2 + 0.25 * 2 + 0.10 * 1 + 0.15 * 1 + 0.15 * 2) = 1.25
CPI = 1 + 1.25 = 2.25
Percentage of the number of cycles that are stalls = 1.25/2.25 = 55.55%.

(v) What is the CPI if we use full forwarding (forward all results that can be forwarded)? What percent of cycles are stalls?

Solution:
All of the above dependencies/conflicts can be resolved using forwarding except MEM to 1st only which con-

stitute 25%.
CPI = 1.25
Percentage of the number of cycles that are stalls = 20%.

(vi) Let us assume that we cannot afford to have three-input multiplexors that are needed for full forwarding. We have to decide if it is better to forward only from the EX/MEM pipeline register (next-cycle forwarding) or only from the MEM/WB pipeline register (two-cycle forwarding). What is the CPI for each option?

Solution:
We have two options, either to forward from EX/MEM register or MEM/WB register.

For EX/MEM register, the number of stalls per RAW would be:
EX to 1st: 0
MEM to 1st: 2
EX to 2nd: 1
MEM to 2nd: 1
EX to 1st and 2nd: 1
This would give a stall of 0.9 per instruction and a CPI of 1.9.

For MEM/WB register, the number of stalls per RAW would be:
EX to 1st: 1
MEM to 1st: 1
EX to 2nd: 0
MEM to 2nd: 0
EX to 1st and 2nd: 1
This would give a stall of 0.5 per instrcution and a CPI of 1.5.

Considering the CPI of both the possibilities, forwarding from MEM/WB register would be a wise choice.

(vii) For the given hazard probabilities and pipeline stage latencies, what is the speedup achieved by each type of forwarding (EX/MEM, MEM/WB, for full) as compared to a pipeline that has no forwarding?

Solution:

Type of Forwarding	No Forwarding	EX/MEM Forwarding	MEM/WB Forwarding	Full Forwarding
CPI	2.25	1.9	1.5	1.25
Period	680ps	690ps	690ps	720ps
Time	1530n	1311n	1035n	900n
Speed Up	-	1.16	1.47	1.7

(viii) What would be the additional speedup (relative to the fastest processor from vii) be if we added "time-travel" forwarding that eliminates all data hazards? Assume that the yet-to-be-invented time-travel circuitry adds 100ps to the latency of the full-forwarding EX stage.

Solution:
If we add the time travel circuitry, the latency would increase by 100 so it would be 820ps but the CPI would drop down to 1 because of no NOP. This would give a speed up of (720*1.25)/(1*820) = 1.09.
Hence, the speedup decreased with the addition of the time travel component.