Organization & Architecture Assignment 4 (Final)- John - Leo - Nguyen

Team

- John Akujobi
- Trung Nguyen Nguyen
- Leo Ostigaard

Question 1 - 4.3

Consider the following instruction mix:

Question 4.3 all parts (8 pts)

4.3 Consider the following instruction mix:

R-type	I-Type	LDUR	STUR	CBZ	B
24%	28%	25%	10%	11%	2%

4.3.1 [5] <§4.4> What fraction of all instructions use data memory?

4.3.2 [5] <§4.4> What fraction of all instructions use instruction memory?

4.3.3 [5] <§4.4> What fraction of all instructions use the sign extend?

4.3.4 [5] <§4.4> What is the sign extend doing during cycles in which its output is not needed?

R-type	I-type	LDUR	STUR	CBZ	B
24%	28%	25%	10%	11%	2%

4.3.1: What fraction of all instructions use data memory?

> $35\% == 0.35 == 7/20$

The types of instructions from the set that uses data memory:

- LDUR (Load): 25%
- STUR (Store): 10%

Adding these together, we get $25\% + 10\% = 35\%$

So, 35% of all instructions in the set use data memory.

4.3.2: What fraction of all instructions use instruction memory?

> 48% == 0.48 = 12/25

```
R-type: 24% - Doesn't use instruction memory
I-type: 28% - Doesn't use instruction memory
LDUR: 25%
STUR: 10%
CBZ: 11%
B: 2%

So, the ones that use the instruction memory are
25 + 10 + 11 + 2 = 48
```

4.3.3: What fraction of all instructions use the sign extend?

```
- I-type: 28%
- CBZ: 11%
    (Conditional Branch if Zero, which uses an offset that might require s
- B: 2%
    (Unconditional Branch, which also uses an offset that might require si

28 + 11 + 2 =41
```

Chapter 4: The Processor, specifically in **Section 4.3: Building a Datapath**.

4.3.4: What is the sign extend doing during cycles in which its output is not needed?

When the output of the sign extension unit is not required, the sign extendoperates in a "do-nothing" or "idle" state.

Question 2

Question 4.5 all pars (12 pts)

a. Hint: I would convert 0xf8014062 to LEGv8 assembly first

4.5 In this exercise, we examine in detail how an instruction is executed in a single-cycle datapath. Problems in this exercise refer to a clock cycle in which the processor fetches the following instruction word: 0xf8014062.

0xf8014062 → 1111 1000 000*0 0001 0100* 00*00 011*0 0010 → STUR instruction (p.122)

D format- opcode: **1111 1000 000** address: **0000 1010 0** op2: **00** Rn: **0001 1** Rt: **0001 0** (p.124)

4.5.1- What are the outputs of the sign-extend and the “shift left 2” unit (near the top of Figure 4.23) for this instruction word?

Answer: Need to sign extend 9 bit offset (0000 1010 0) then shift left 2 to this sign extended offset field:

```
Sign extended: 0000 0000 0000 0000 0000 0000 0000 0000 0000 1010 0
Shifted left 2:   0000 0000 0000 0000 0000 0000 0000 0000 0010 1000 0
Result: 1111 1000 0000 0000 0000 0000 0000 0000 0000 0000 0101 0000 0
```

The above offset field is underlined

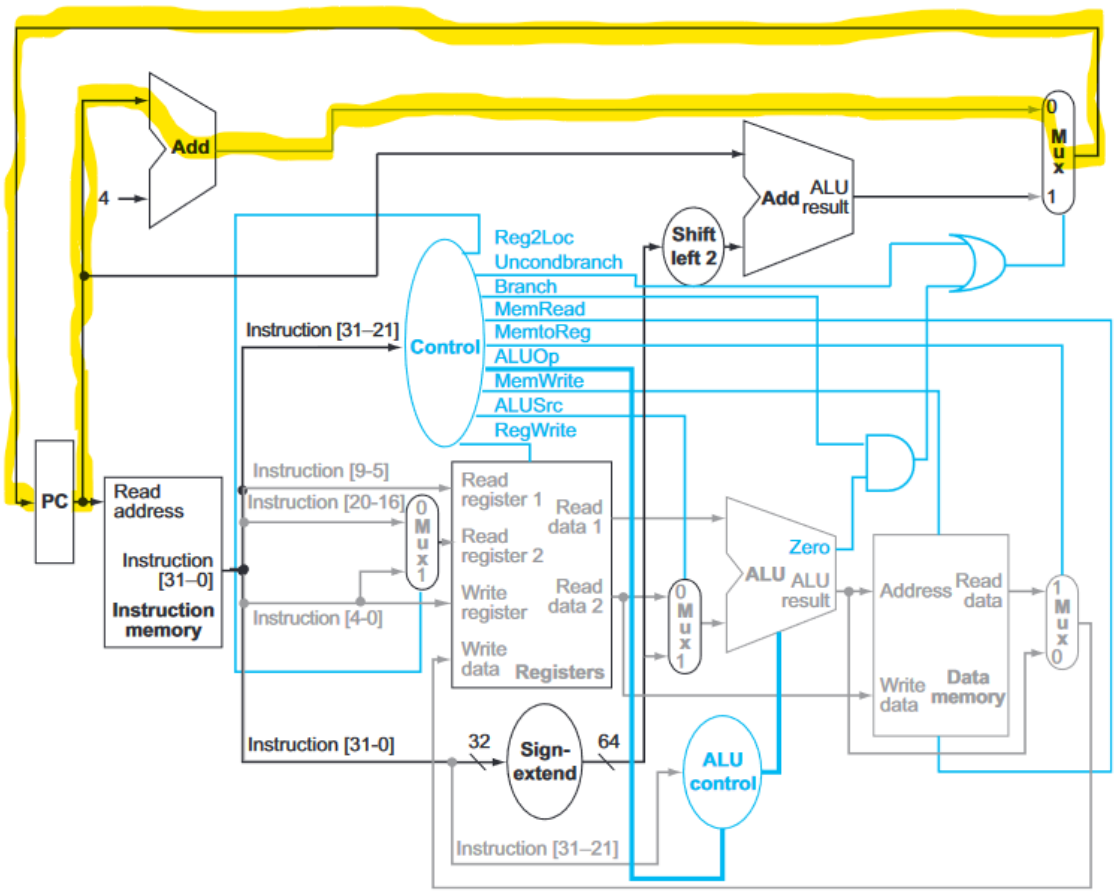
Sign-extend extends the sign bit of the 9-bit address part of the instruction to convert to 64-bit result and preserve the rest, and “shift left 2” would simply add 00(base 2) to the low order end of the offset (p. 267-268)

4.5.2- What are the values of the ALU control unit’s inputs for this instruction?

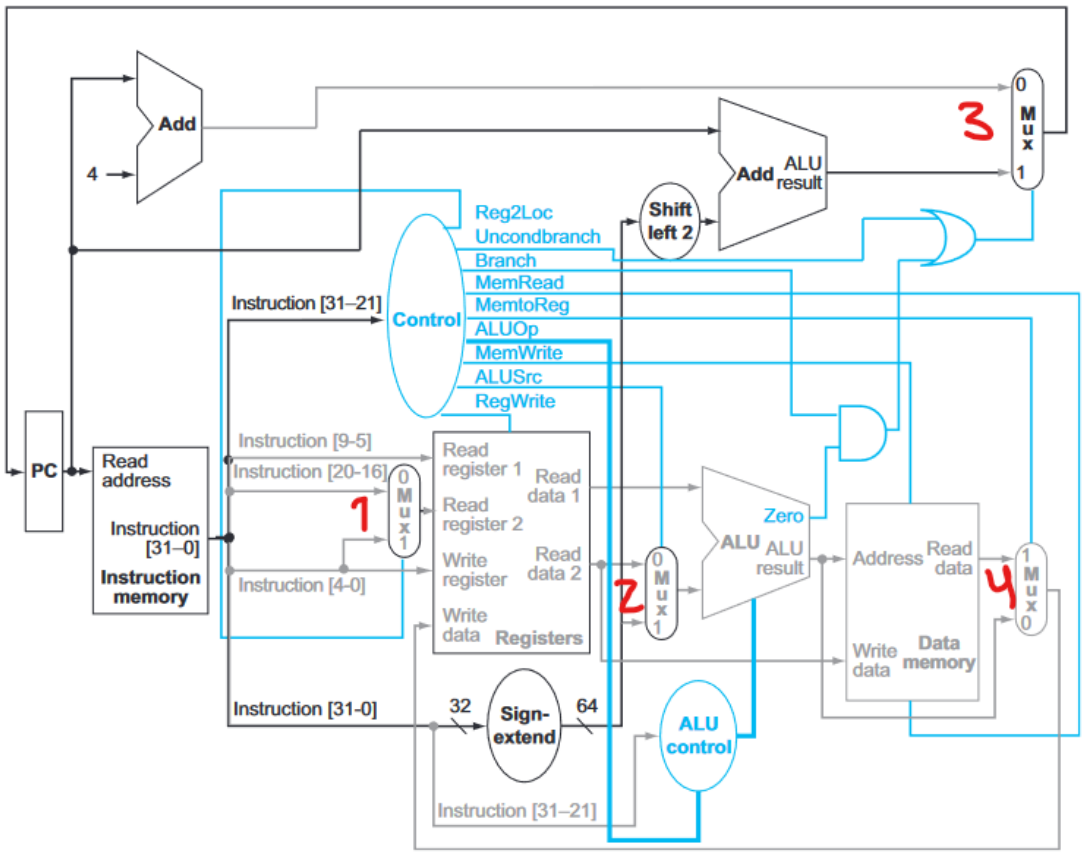
Answer: The values of the ALUop is **00** since it corresponds to the STUR instruction (p.272) control unit inputs are 00, since they correspond to instruction[25-24] The values for the ALU control inputs are **0010**, independent of the opcode. (as described on p.273)

4.5.3- What is the new PC address after this instruction is executed?
Highlight the path through which this value is determined.

Answer: The new address is equivalent to 4 + the old PC address



4.5.4- For each mux, show the values of its inputs and outputs during the execution of this instruction. List values that are register outputs at Reg [Xn].



Answer: Refer to image above for numbering the 4 Mux

Mux 1: Determines the the read register value. Since this is STUR instruction, we will use instruction bits [4-0] for the 2nd read register (Rt). Thus the Mux is selected “on” and propagates **Rt** (0 0010)

Mux 2: Determines the 2nd ALU input to be from the registers or from the offset field of the instruction. Since it is a STUR instruction, the Mux is selected “on” to use the **offset field** as the 2nd ALU input (0 0000 0000 0000 0000 0000 0000 0000 0000 0101 0000)

Mux 3: Determines what value replaces the program counter. In this case, it is selected “off” and thus the value is **PC + 4** (p.259)

Mux 4: Determines the output of data memory. It is selected “on” in this case, and propagates the value to be carried to “write data” so it can be stored into data, **Rt** (0 0010) (p.259)

The register Rn is a register output from the register file (p. 88)

4.5.5- What are the input values for the ALU and the two add units?

Answer: The first input value for the ALU is the register Rn. The second input value for the ALU is the sign-extended offset field.

The two add units, if referring to the results from the 2 adders in the datapath are: PC + 4 and PC + sign-extended and shifted 64-bit value

4.5.6- What are the values of all inputs for the registers unit?

Answer: For the registers unit, the values are as follows:

Read reg1= address of Rn (**0 0011**)

Read reg2= address of Rt (**0 0010**)

Write reg= address of Rt (**0 0010**)

Write data= value stored in Rt (**0 0010**)

Question 3

Question 4.7. all part (8pts)

Question: Problems in this exercise assume that the logic blocks used to implement a processor's datapath have the following latencies

I-Mem / D-Mem	Register File	Mux	ALU	Adder	Single gate	Register Read	Register Setup	Sign extend	Control
250 ps	150 ps	25 ps	200 ps	150 ps	5 ps	30 ps	20 ps	50 ps	50 ps

"Register read" is the time needed after the rising clock edge for the new register value to appear on the output. This value applies to the PC only. "Register setup" is the amount of time a registers data input must be stable before the rising edge of the clock. This value applies to both the PC and Register File.

4.7.1. Although the control unit as a whole requires 50ps, it so happens that we can extract the correct value of the Reg2Loc control wire directly from the instruction.

Thus, the value of this control wire is available at the same time as the instruction. Explain how we can extract this value directly from the instruction.

Hints: carefully examine the opcodes shown in figure 2.20.

Also, remember that LSR and LSL do not use the Rm field.

Finally, ignore STXR.

Figure 2.20 → PDF p. 161

Figure 4.16 → PDF p. 360 - The effect of each of the seven control signals

Figure 4.17 → PDF p. 361 - The simple datapath with the control unit

Signal name	Effect when deasserted	Effect when asserted
Reg2Loc	The register number for Read register 2 comes from the Rm field (bits 20:16)	The register number for Read register 2 comes from the Rt field (bits 4:0)

4.7.2. What is the latency of an R-type instruction (i.e., how long must the clock period be to ensure that this instruction works correctly)?

From figure 4.19 - the datapath in operation for an R-type instruction - PDF p. 362

We have this:

Time = MUX + Single gate + Register read + register setup (PC) + I-mem + Adder + control + MUX + Register setup + register file + MUX + ALU + MUX + ALU control (?)

= 25 + 5 + 30 + 20 + 250 + 150 + 50 + 25 + 20 + 150 + 25 + 200 + 25 = 975 ps [note: without ALU control]

4.7.3. What is the latency of LDUR? (Check your answer carefully. Many students place extra muxes on the critical path.)

Figure 4.20. The datapath in operation for a load instruction - PDF p. 363

Time = Register read + Register setup (PC) + Adder + I-mem + Control + register file + register setup + sign extend + MUX + ALU + D-mem + Mux + ALU control (?)

= 30 + 20 + 150 + 250 + 50 + 150 + 20 + 50 + 25 + 200 + 250 + 25 = 1220 ps [Note: without ALU control]

4.7.4. What is the latency of STUR?

Figure 4.20. The datapath in operation for a load instruction. In the note.

Time = MUX (?) + Register read + Register setup (PC) + Adder + I-mem + Control + Register file + Register setup + sign extend + MUX + ALU + D-mem + ALU control (?)

= 25 + 30 + 20 + 150 + 250 + 50 + 150 + 20 + 50 + 25 + 200 + 250 = 1220 ps [Note: without ALU control]

4.7.5. What is the latency of CBZ?

Figure 4.21. The datapath in operation of a compare-and-branch-on-zero instruction.

Time = Add + MUX + Single gate + Register read + Register setup (PC) + I-mem + MUX + Sign extend + Adder + Register file + Register setup + Control + MUX + ALU + ALU control (?)

= 150 + 25 + 5 + 30 + 20 + 250 + 25 + 50 + 150 + 150 + 20 + 50 + 25 + 200 = 1150 ps [note: without ALU control]

4.7.6. What is the latency of B?

Figure 4.23. The simple control and datapath are extended to handle the unconditional branch instruction - PDF p. 366

Time = MUX + Register read+ register setup (PC) + I-mem + Adder + control + sign-extend + adder + single gate (AND) + single gate (OR) + ALU Control (?)

= 25 + 30 + 20 + 250 + 150 + 50 + 50 + 150 + 5 + 5 = 735 ps [Note: without ALU control]

4.7.7. What is the latency of an I-type instruction?

Recall I-format (PDF p. 163)

Opcode	Immediate	Rn	Rd
10 bits	12 bits	5 bits	5 bits

I-type instruction doesn't use adder to perform branching, not use Read Register 2 and not access to Data memory.

Time = Mux + Register read + Register setup (PC) + I-mem + Adder + Control + Sign-extend + Register file + Register setup + MUX + ALU + Mux + single gate + ALU control

= 25 + 30 + 20 + 250 + 150 + 50 + 50 + 150 + 20 + 25 + 200 + 25 + 5 = 1000 [Note: without ALU control]

4.7.8. What is the minimum clock period for this CPU?

At this point, latency will be from all part of the circuit, the maximum latency will be used to calculate the minimum clock period - figure 4.17: the simple datapath with the control unit (PDF p. 361)

Time: MUX + Register read + Register setup (PC) + I-mem + Adder + Control + MUX + Sign-extend + Register file + Register Setup + MUX + ALU + Single Gate + D-Mem + MUX + Adder + ALU Control (?)

= 25 + 30 + 20 + 250 + 150 + 50 + 25 + 50 + 150 + 20 + 25 + 200 + 5 + 250 + 25 + 150 = 1425 ps

Question 4

Question 4.16 all parts (10 pts)

Question: 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
250 ps	350 ps	150 ps	300 ps	200 ps

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

ALU/Logic	Jump/Branch	LDUR	STUR
45%	20%	20%	15%

Note: Figure 4.27. graphical representation of the instruction pipeline, similar in spirit to the laundry pipeline (PDF p. 374)

IF = instruction fetch stage

ID = instruction decode/register file read stage

EX = execution stage

MEM = memory access stage

WB = Write back stage

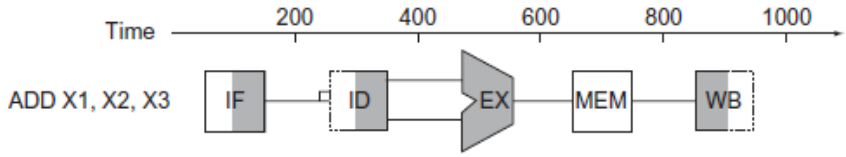


FIGURE 4.27 Graphical representation of the instruction pipeline, similar in spirit to the laundry pipeline in Figure 4.24. Here we use symbols representing the physical resources with the abbreviations for pipeline stages used throughout the chapter. The symbols for the five stages: *IF* for the instruction fetch stage, with the box representing instruction memory; *ID* for the instruction decode/register file read stage, with the drawing showing the register file being read; *EX* for the execution stage, with the drawing representing the ALU; *MEM* for the memory access stage, with the box representing data memory; and *WB* for the write-back stage, with the drawing showing the register file being written. The shading indicates the element is used by the instruction. Hence, MEM has a white background because ADD does not access the data memory. Shading on the right half of the register file or memory means the element is read in that stage, and shading of the left half means it is written in that stage. Hence the right half of ID is shaded in the second stage because the register file is read, and the left half of WB is shaded in the fifth stage because the register file is written.

4.16.1. What is the clock cycle time in a pipelined and non-pipelined processor?

Non-pipelined time:

Time = 250 + 350 + 150 + 300 + 200 = 1250 ps

Pipelined time:

Time = the longest latency time among stages = 350 ps

Sentence from page 286 - PDF 370

> In the nonpipelined design is 3 × 800 ps or 2400 ps.
>
> All the pipeline stages take a single clock cycle, so the clock cycle must be long enough to accommodate the slowest operation. Just as the single-cycle design must take the worst-case clock cycle of 800 ps, even though some instructions can be as fast as 500 ps, the pipelined execution clock cycle must have the worst-case clock cycle of 200 ps, even though some stages take only 100 ps. Pipelining still offers a fourfold performance improvement: the time between the first and fourth instructions is 3 × 200 ps or 600 ps.

4.16.2. What is the total latency of an LDUR instruction in a pipelined and non-pipelined processor?

The processor executes all instructions → The latency of the LDUR will be the total time multiply with the percentage in instruction set.

Non-pipelined time for LDUR instruction:

Time = total non-pipelined time * instruction percentage = 1250 ∗ 20% = 250 ps

Pipelined time for LDUR instruction:

Time = pipelined time * instruction percentage = * 20% = 70 ps

4.16.3. 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?

I will split the ID stage → since it is the longest stage

IF	ID/2	ID/2	EX	MEM	WB
250	175	175	150	300	200

New time = max of all stage time = 300 ps

4.16.4. Assuming there are no stalls or hazards, what is the utilization of the data memory?

Without stalls or hazard → which means loading and storing to data memory is not having to wait each other to complete. Both LDUR and STUR will be executed together.

Data Hazards

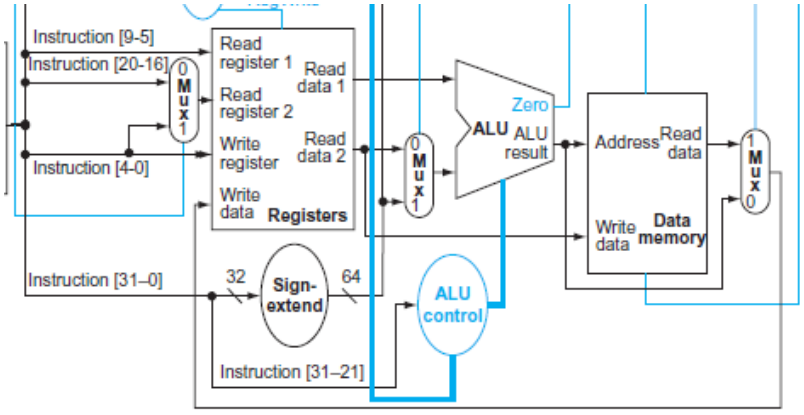
Data hazards occur when the pipeline must be stalled because one step must wait for another to complete. Suppose you found a sock at the folding station for which no match existed. One possible strategy is to run down to your room and search through your clothes bureau to see if you can find the match. Obviously, while you are doing the search, loads that have completed drying are ready to fold and those that have finished washing are ready to dry.

Hence, the utilization will be the LDUR and STUR time combine = 20% + 15% = 35%

PDF p. 372 - 373. Definition about hazards and data hazards

4.16.5. Assuming there are no stalls or hazards, what is the utilization of the write-register port of the “registers” unit?

From the figure 4.17. The simple datapath with the control unit. We can see that the write-register port of the register unit will write data to register. However, the write data must come from the ALU or Data Memory (PDF p. 361)



Hence, the utilization will be the combination of ALU/Control (ALU) and Jump/branch (MUX) = 45% + 20% = 65%

Question 5 (Leo)

Question 4.18- Assume that X1 is initialized to 11 and X2 is initialized to 22.
Suppose you executed the code below on a version of the pipeline from Section 4.5 that does not handle data hazards (i.e., the programmer is responsible for addressing data hazards by inserting NOP instructions where necessary).
- What would the final values of registers X3 and X4 be?

```
ADDI X1, X2, #5
ADD X3, X1, X2
ADDI X4, X1, #15
```

```
//New code

ADDI X1, X2, #5 //X1 = X2 + 5
NOP
NOP //STALL TWO OPERATIONS
ADD X3, X1, X2 //X3 = (X2+5) + X2
ADDI X4, X1, #15 //X4 = (X2+5) + 15
```

This code above avoids data hazards by waiting until the ADDI instruction finished to access the value at X1. The final values of the registers is below:

X3 = 2*(X2) + 5

X4 = X2 + 15

Question 6 - 4.22 all parts (8pts)

Consider the fragment of LEGv8 assembly below

```
STUR X16, [X6, #12]
LDUR X16, [X6, #8]
SUB  X7, X5, X4
CBZ  X7, Label
ADD  X5, X1, X4
SUB  X5, X15, X4
```

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.

4.22.1: Draw a pipeline diagram to show where the code above will stall.

1. Lets break down the instructions

```
STUR X16, [X6, #12]   ; Store operation: accesses data memory
LDUR X16, [X6, #8]    ; Load operation: accesses data memory
SUB  X7, X5, X4       ; Arithmetic operation: no memory access
CBZ  X7, Label        ; Branch operation: needs to fetch next instruction
ADD  X5, X1, X4       ; Arithmetic operation: no memory access
SUB  X5, X15, X4      ; Arithmetic operation: no memory access
```

1. When `STUR` is in the Memory Access stage (accessing data memory), the next instruction needs to be stalled.

   `LDUR` cannot be fetched from memory during the same cycle, because it will lead to a structural hazard.

2. Also, when the `LDUR` instruction is accessing data memory, the nextinstruction (`SUB` in this case) cannot be fetched from memory during the same cycle, creating another structural hazard.

Pipeline stages:

- Instruction Fetch (IF)
- Instruction Decode (ID)
- Execute (EX)
- Memory Access (MEM)
- Write Back (WB)
- Stall *

```
Cycle:  1   2   3   4   5   6    7    8    9    10   11  12  13  14  15  1
STUR:   IF  ID  EX  MEM WB
LDUR:       IF  ID  EX  *   MEM  WB
SUB :           IF  ID  EX  MEM  WB
CBZ :               IF  ID  *    EX   MEM  WB
ADD :                   IF  ID   EX   MEM  WB
SUB :                       IF   ID   EX   MEM  WB
```

Or

```
Cycle: 1   2   3   4    5   6    7    8    9    10  11  12  13  14  15
STUR: IF  ID   EX  MEM  WB
LDUR:     IF   ID  EX   *   MEM  WB
SUB :          IF  ID   EX  MEM  WB
CBZ :              IF   ID  EX   MEM  WB
ADD :                  IF  ID   EX   MEM  WB
SUB :                      IF   ID   EX   MEM  WB
```

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

Yes, we can do so by reordering the code

```
STUR X16, [X6, #12]  ; Store instruction, uses MEM
SUB  X7, X5, X4      ; Independent arithmetic instruction
ADD  X5, X1, X4      ; Another independent arithmetic instruction
LDUR X16, [X6, #8]   ; Load instruction, uses MEM, moved down
```

```
CBZ  X7, Label       ; Branch instruction, depends on SUB result
SUB  X5, X15, X4     ; Depends on the result of the previous ADD
```

1. **STUR X16, [X6, #12]**:
    - This store instruction accesses the memory unit.
    - We keep it at the beginning because it's necessary to complete this memory operation before any potential loads that might depend on this stored value.
2. **SUB X7, X5, X4**:
    - Do this immediately after the `STUR` instruction.
    - It doesn't need memory access, so we can do it without stalling while `STUR` is in the MEM stage.
    - This way, we use the cycle that would be idle because of `STUR` instruction's memory access.
3. **ADD X5, X1, X4**:
    - Doesn't depend on the previous `LDUR` or `STUR` instructions.
    - By placing it here, we'll also be using what would have been stall cycles if we had followed up `STUR` with `LDUR` immediately.
4. **LDUR X16, [X6, #8]**:
    - Execute the after two independent arithmetic operations.
    - Gives a buffer of cycles for the `STUR` operation to complete its access to the shared memory unit. This reduces the chance of a stall when `LDUR` needs to access memory.
5. **CBZ X7, Label**:
    - This branch instruction depends on the result of `SUB X7, X5, X4`.
    - By the time we reach this instruction, `SUB` has completed, and its result is available for the branch decision.
    - By placing it here, we also reduce the chance of an instruction fetch conflicting with `LDUR`'s memory access.
6. **SUB X5, X15, X4**:
    - Depends on the result of `ADD X5, X1, X4`.
    - Since `ADD` is placed earlier and has had time to execute, the data dependency is maintained, and we can execute `SUB` without waiting.

4.22.3: 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.

Handling with Hardware

1. **Out-of-Order Execution:**

   More advanced processors can execute instructions out of their original program order when data dependencies allow.

   This can help us work around structural hazards by keeping the pipeline filled with other work while waiting for a contested resource to become available.

2. **Separate Instruction/Data Caches:**

   Having separate caches for instructions (I-cache) and data (D-cache) allows us to have concurrent accesses without conflict.

3. **Dual-Ported Memory:**

   By having dual-ported memory, we can have simultaneous access to two memory addresses. This can be used to separate instruction fetch and data access.

Handling with Software

1. **Instruction Reordering:**
   - We can reorder the instructions to avoid immediate consecutive access to the memory unit for both fetching and data access can reduce stalls.
   - However, the effectiveness of this is limited by how available independent instructions that can be moved between memory access instructions without breaking or changing the program.
2. **Inserting NOPs or Independent Instructions:**
   - Inserting NOPs (No Operation Instructions) between a memory access instruction and the next instruction fetch can create a buffer, allowing the memory access to complete before the next fetch.
   - Because NOPs do nothing useful, it is a brute-force approach and can lead to inefficient use of the CPU.

So, while using software techniques can help us reduce the impact of the structural hazard, they come with limitations and tradeoffs in efficiency.

We think the most effective solutions are hardware-based. Here, we address the root of the problem by allowing concurrent access to memory for instruction fetches and data access.

4.22.1: Approximately how many stalls would you expect this structural hazard to generate in a typical program?

(Use the instruction mix from Exercise 4.8.)

R-type/I-type	LDUR	STUR	CBZ	B
52%	25%	10%	11%	2%

1. **LDUR (Load)** and **STUR (Store):**

   They need the memory unit for data access.

   Their combined percentage is 25% + 10% = 35%.

2. **CBZ (Conditional Branch if Zero)** and **B (Branch):**

   They might also require the memory unit if the branch is taken and a new instruction needs to be fetched from a non-sequential address.

   Their combined percentage is 11% + 2% = 13%.

```
From LDUR/STUR: 35% chance of a stall per instruction.
From CBZ/B: 13% chance of a stall per instruction.

35% + 13% = 48%
```

I expect there to be about 1 stall every 2 instructions

Since 48% ~= 50% or half of the time