Sys3 Open Assessment

Part A

A(i)

Hazard Type	1st Instruction	2nd Instruction
RAW	(1, 6, WB.3)	(2, 5, RR.3)
WAW	(2, 8, WB.5)	(3, 7, WB.5)
WAW	(4, 9, WB.2)	(5, 8, WB.2)
Structural	(2, 8, WB.5)	(5, 8, WB.2)
Memory	(2, 4, OF)	(4, 4, IF)

A(ii)

Hazard Type	1st Instruction	2nd Instruction	3rd Instruction
Memory	(1, 3, OF)	(3, 3, IF)	
Memory	(2, 4, OF)	(4, 4, IF)	
Structural	(1, 7, IALU)	(3, 7, IALU)	(4, 7, IALU)
Structural	(3, 8, WB.5)	(4, 8, WB.2)	
WAR	(1, 5, RR.3)	(3, 5, WB.3)	

- If a register reads and writes to the same register in the same cycle, is it a RAW / WAR?
  - (1, 5, RR.3) (3, 5, WB.3)
- Which hazard???
- **Simulate on practical.**
- WAR Hazard ? -- RR.5(i4,c14)) WB.5,(i1,c14)
- ??? Hazard ? -- RR.3(i1, c5) WB.3(i3, c5)

A(iii)

Hazard Type	1st Instruction	2nd Instruction	3rd Instruction
Structural	(2, 4, RR.3)	(3, 4, RR.5)	(4, 4, RR.7)
Structural	(1, 6, IALU)	(2, 6, IALU)	(4, 6, IALU)
Structural	(3, 6, FALU)	(6, 6, FALU)	
Structural	(2, 8, WB.4)	(6, 8, WB.12)	
Memory	(1, 7, DS)	(2, 7, DF)	(5, 7, OF)

- Is final memory hazard valid???
- Is it memory or DATA
- 2 Memory read ports?

Part B

B(i)

Instr.												
1	IF	ID	OF	RR.2	RR.3	IALU	IALU	DS				
2		IF	ID	OF	---	WB.3						
3			---	---	IF	ID	RR.4	IALU	IALU	WB.5		
4				---	---	IF	ID	RR.2	IALU	---	WB.2	
	1	2	3	4	5	6	7	8	9	10	11	← cycle

B(ii)

Instr.												
1	IF	ID	OF	RR.2	RR.3	IALU	IALU	DS				
2		NOP										
3			NOP									
4				IF	ID	OF	WB.3					
5					IF	ID	RR.4	IALU	IALU	WB.5		
6						NOP						
7							IF	ID	RR.2	IALU	WB.2	
	1	2	3	4	5	6	7	8	9	10	11	← cycle

B(iii)

Instr.												
1	IF	ID	RR.1	RR.2	IALU	WB.3						
2		IF	ID	OF	WB.13	IALU	IALU	IALU	WB.5			
3			IF	ID	RR.4	IALU	IALU	WB.15				
4				IF	ID	RR.13	IALU	WB.2				
5					IF	ID	OF	RR.9	RR.15	IALU	WB.2	
6						IF	ID	WB.19				
	1	2	3	4	5	6	7	8	9	10	11	← cycle

- **Still has memory hazards:** (2,4,OF) (4,4,IF)

Part C

C(i)

Test Case	Serial cycles	Pipelined cycles (hazards)	Pipelined cycles (hazards resolved)	Speedup (hazards)	Speedup (hazards resolved)
B(i)	23	8	11	2.9	2.1
B(ii)	23	8	11	2.9	2.1
B(iii)	35	11	11	3.2	3.2

- Example gives the number of cycles without hazards $<$ number of cycles with hazards
  - Resolving hazards should always take more cycles?

- Inserting stalls/NOP should increase number of cycles or stay the same.

C(ii)

(a) RAW: An instruction I_1 intends to write to a register R. However, before the W.B cycle is executed, a later pipelined instruction, I_2 reads R. This causes I_2 to have an outdated version of R, not the value that would be written from I_1.

Solution: Insert stalls before the R.R cycle in I_2 until it is executed the cycle after the W.B.

(b) WAR: An instruction I_1 intends to read a register R. However, before the R.R cycle is executed, a later pipelined instruction I_2 writes a new value to R. This causes I_1 to read the new incorrect value of R, rather than the expected that I_1 would read the value in R before either instruction was executed.

Solution: Insert non-dependant instructions between the read and write instructions, such that the write instruction occurs at least a cycle after the read instruction. If no non-dependant instructions exist in this particular context, insert NOP instructions.

(c) WAW: Instructions I_1, I_2 intend to write to register R, such that instruction I_1 is executed before I_2. A WAW hazard occurs when I_2 executes its W.B cycle before I_1, causing the value from I_2 to be overwritten by the outdated value from I_1, when I_1 executes its W.B cycle.

Solution: Operand forwarding with Tomasulo algorithm, which uses register renaming.

Rename the second write instruction register with a new register, and rename any occurrences of that register in instructions after the second write to the new register.

Part D

D(i)

> Explain the concept of dependency DAG, and how this dependency DAG graph representation relates to **dependencies** and **register hazards**.

A DAG represents the dependencies within a “basic block”, a sequential code sequence with no branches except entry and exit. Each instruction is represented as a node, and the edges between them “serialization dependencies”.

The DAG referenced in the paper has three types of dependencies: *definition vs. definition* (a register being overwritten, a potential WAW hazard), *definition vs. uses* (a value stored to a register, and loaded in a subsequent instruction a potential RAW hazard) and *uses vs. definition* (a register used in an instruction, then being overwritten in a subsequent instruction, a potential WAR hazard).

Any edge $\vec{ab}$ asserts that the two instructions a and b have a dependency and that the compiler must execute a before b. Using the DAG, the compiler can rearrange the execution order, while ensuring dependant instructions are executed sequentially.

Only hazards are register & memory based:

1. Loading a register from memory followed by using *that* register as a source
2. Storing to any memory location followed by loading from any location

3. Loading from memory followed by using *any* register as the target of an ALU or load/store with address modification

NO3 Seems pretty common??

D(ii)

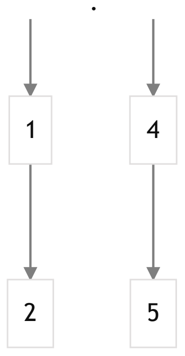
CODE:

```
// Potential WAW (1, 2)
1. IMUL R1, R1, R2
2. IADD R3, R3, R2

// Potential WAR (3, 4)
3. LDR R9, R5
4. IADD R9, R9, R6
```

Instructions 1 & 2 give an example of a potential WAW hazard, where instruction 1 attempts to store a value to $R2$, followed by instruction 2 attempting to store a value in $R2$. This is referenced in the DAG as a *definition vs definition* dependency.

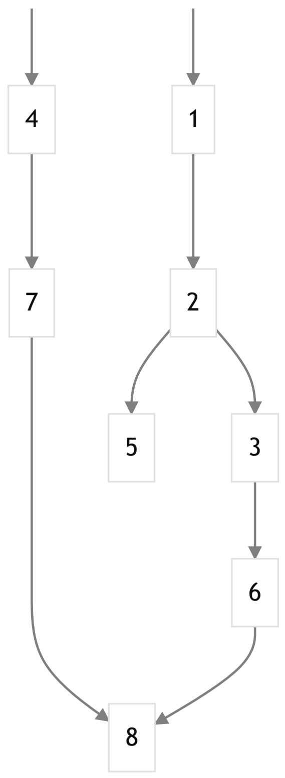
Instructions 3 & 4 give an example of a potential WAR hazard, where instruction 3 attempts to store a value in $R9$ followed by instruction 4 which attempt to use $R9$ as a source. This is referenced in the DAG as a *definition vs use* dependency.



D(iii)

TEST CASE:

```
1. LDR R1, R5
2. LDR R5, R6
3. IMUL R3, R6, R8
4. IMUL R1, R4, R4
5. IADD R2, R6, R5
6. IMUL R3, R6, R8
7. IMUL R1, R4, R4
8. IADD R4, R8, R7
```



Appendix Notes

$O(n^4) \to O(n^2)$

Only hazards are register & memory based:

1. Loading a register from memory followed by using *that* register as a source
2. Storing to any memory location followed by loading from any location
3. Loading from memory followed by using *any* register as the target of an ALU or load/store with address modification

Approach

1. Divide each procedure into basic blocks
2. Construct dependency DAG expressing scheduling constraints within each basic block
3. Schedule instructions in block, guided by applicable heuristics using no lookahead in the graph

DAG
Node: instructions
Edge: Serialization dependencies between instructions

Edge leading from instruction a to instruction b indicates a must be executed before b

DAG serializes

- Definitions v definitions
- Definitions v uses
- Uses v definitions

Scan backwards across basic block, noting each definition or use of resource and then later the definitions or uses which must precede it

```
1    add      #1,r1,r2
2    add      #12,sp,sp
3    store    r0,A
4    load     -4(sp),r3
5    load     -8(sp),r4
6    add      #8,sp,sp
7    store    r2,0(sp)
8    load     A,r5
9    add      #1,r0,r4
```

Figure 1. Sample code sequence.

$$\begin{tikzcd}
1 \arrow[dddrr, bend right] & & 2 \arrow[dl] \arrow[dr] & & & 3 \arrow[d] \\
& 4 \arrow[dr] & & 5 \arrow[dl] \arrow[dr] & & 8 \\
& & 6 \arrow[d] & & 9 & \\
& & 7 & & &
\end{tikzcd}$$

Figure 2. Dependency dag for code in Fig. 1.

Result: (3, 2, 4, 5, 8, 1, 6, 7, 9)

Heuristics to use instead of lookahead:

1. Whether an instruction interlocks with any of its immediate successors in the DAG
2. The number of immediate successors of the instruction
3. The length of the longest path from the instruction to the leaves of the DAG

These properties bias towards selecting properties which:

1. May cause interlocks (need to be scheduled as early as possible, when there is most likely to be a wide choice of instructions to follow them)
2. Uncover most potential successors
3. Balance the progress along the various paths towards the leaves of the DAG (leave the largest number of choices available at all stages of the process)

Scheduling algorithm

1. Make a prepass backward over the basic block to construct scheduling DAG comparing each instruction to the nodes of scheduling DAG constructed so far
2. Put roots of DAG into candidate set
3. Select first instruction to be scheduled from the candidate set