Name: TOOBA KHAN
Entry Number: 2021JCS2245
Subject: SIL-618

Question 1

Answer

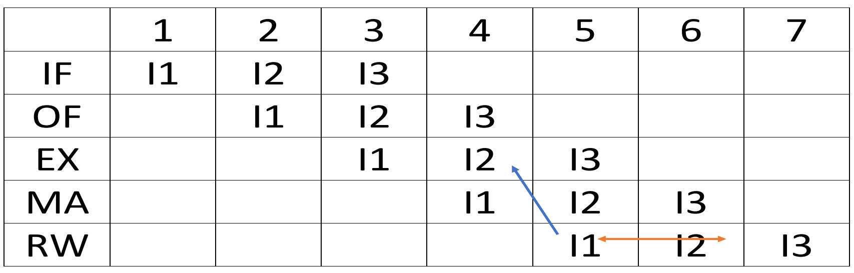
Marks: 10

1. Register renaming maps architectural registers to a set of physical registers.
2. WAW (Write after write) and WAR (Write after read) are not real dependencies. They do not occur in Simple Risc pipeline as well. They occur majorly in out of order pipelines because the set of architectural registers available to us for use are not sufficient.
3. When there are lesser registers than required, they will have to be reused. But, if somehow, we can get a greater number of registers, we can totally eliminate the problem.
4. Physical registers are a set of registers visible only to the processor and not the user i.e only with CPL=0.
5. In renaming, every time assignment is done to a new physical register, and it is used in all the subsequent read operations until it's architectural register is assigned a new value again.
6. This preserves the correctness of the code. And since, every time assignment is done to a new physical register, write after write dependency (which occurs when we try to assign values to same register in two instructions which are in the pipeline at the same time) cannot occur.
7. It will also remove all the WAR dependencies because every time a new physical register is used as a destination, so it will never happen that the physical register which has already been read will be assigned values. Thus, removing WAR dependencies.

If we have a large number of architectural registers, renaming will still be required. Register renaming is used to remove WAR and WAW dependencies in out-of-order processors but in such processors, any independent instruction can be executing. If we have a fixed (even if the fixed number is very large) number of architectural registers, renaming will still be required. Also, if a given processor has infinite number of registers, then register renaming will not be required. But such a scenario cannot exist, infinite registers mean no fixed length of address space which is not possible.

Example: For the instruction given below:
Original:
add r1, r2, r3
sub r4, r1, r2
add r1, r5, r3

Pipeline Diagram shows that RAW and WAW dependency exists.

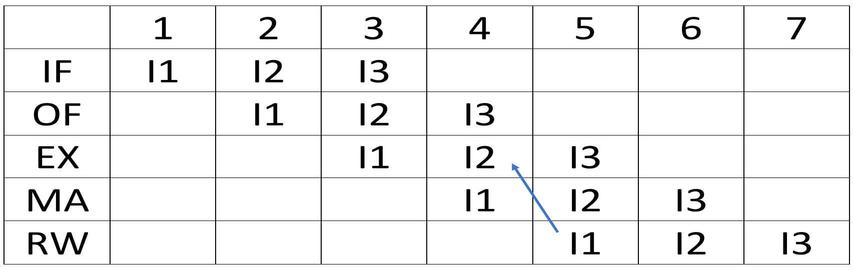


	1	2	3	4	5	6	7
IF	I1	I2	I3				
OF		I1	I2	I3			
EX			I1	I2	I3		
MA				I1	I2	I3	
RW					I1	I2	I3

RAW dependencies cannot be resolved using register renaming, because register renaming maps every new assignment to a different physical register which is read in subsequent instructions. It preserves all kinds of real dependencies and does not change the meaning of the program.

After register renaming (only new assignments will have new physical registers. Read operation will be performed on existing physical registers).
add q1, q2, q3
sub q4, q1, q2
add q5, q6, q3

Pipeline Diagram shows that RAW and WAW dependency exists.



	1	2	3	4	5	6	7
IF	I1	I2	I3				
OF		I1	I2	I3			
EX			I1	I2	I3		
MA				I1	I2	I3	
RW					I1	I2	I3

Question 2

Answer Marks: 9

Register window is the set of registers which are visible and/or accessible by any instruction or function at any point of time. In Simple RISC, register windows segregate the registers into two parts: one available to users and other reserved only for the processor and privileged instructions.

1. User programs: r0 to r16
2. Privileged instructions: OldPC, OldSP, OldFlags, flags, r0, sp .

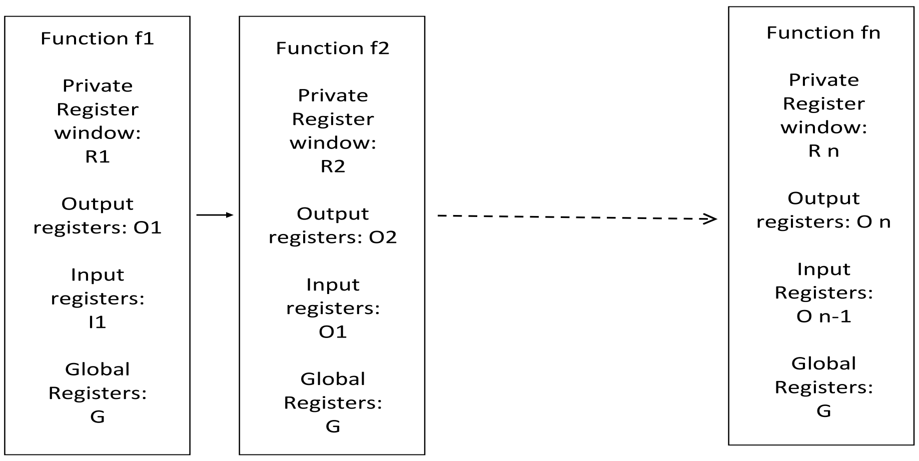
The current register window is decided by the CPL or current privilege level bit. CPL is 0 for privileged instructions and 1 for user programs.
The available registers can be divided into three groups: Input registers, output registers, global registers, and private registers.

1. Every function can be given its private window of registers. These can be used by the function for local manipulations.
2. The global set of registers can be accessed by any function.
3. Input and output registers can be used by functions to pass values. Every function (caller) which calls another function (callee), places it's arguments in it's output registers. These output registers overlap with the input register of the callee.

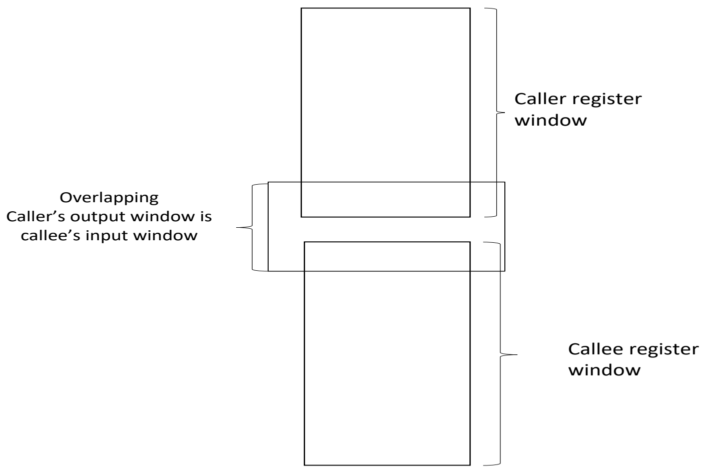
Function calls can benefit from register windows by saving costly register spills as follows:

1. Every function has it's own register window which consists of its output registers, input registers and private registers.
2. The input registers of callee overlap with the output register of the caller. This saves major time and cost which was caused by register spilling in the caller saved and callee saved scheme.
3. The private registers will be used by each function for internal computations, hence no spills will be required when invoking another function.
4. The input registers of callee overlap with the output registers of caller.
5. When a function wants to give arguments as input to callee, it places them in its output registers and when callee wants to return values, callee places them in its input registers.

Below is the representation of function calls using register windows.



Below is the representation of how register windows of functions overlap.



Disadvantages of this scheme:

1. The number of nested function calls will be restricted by the number of private, input, output registers available by the processor.

2. The number of arguments that can be passed will be restricted by the output register capacity of the caller.

3. The number of return values that can be passed will be restricted by the input register capacity of the callee.

Question 3

Answer Part A: Marks: 5

We require latches in a pipeline to store the intermediate results of stages.

1. Pipeline stages have slightly different circuit delays. Which means that it is possible that one stage finishes its task and computes results for next stage even before next stage is ready.

2. In such a case, the outputs of each pipeline stage is latched in a pipeline register for the use at next stages.

3. Also, in a pipeline there can be at max 5 instructions at a time (for a 5 stage pipeline). When Instruction I is in stage MA, instruction j in state EX can execute and change the control signals required by instruction I. Now, instruction I will not be able to complete its execution properly because the control signals it required have been corrupted by another instructions.

4. This may happen with operand values, ALU result computed, memory addresses computed, the PC of the instruction and other values required by instruction as well.

5. In such a scenario, it becomes essential to store the values computed by an instruction in any stage for future use until it is removed from the pipeline. Thus, pipeline registers are essential.

Part B:

If any stage can violate the hold time for a latch, we can make use of enabling and disabling read ports. This can be done as follows:

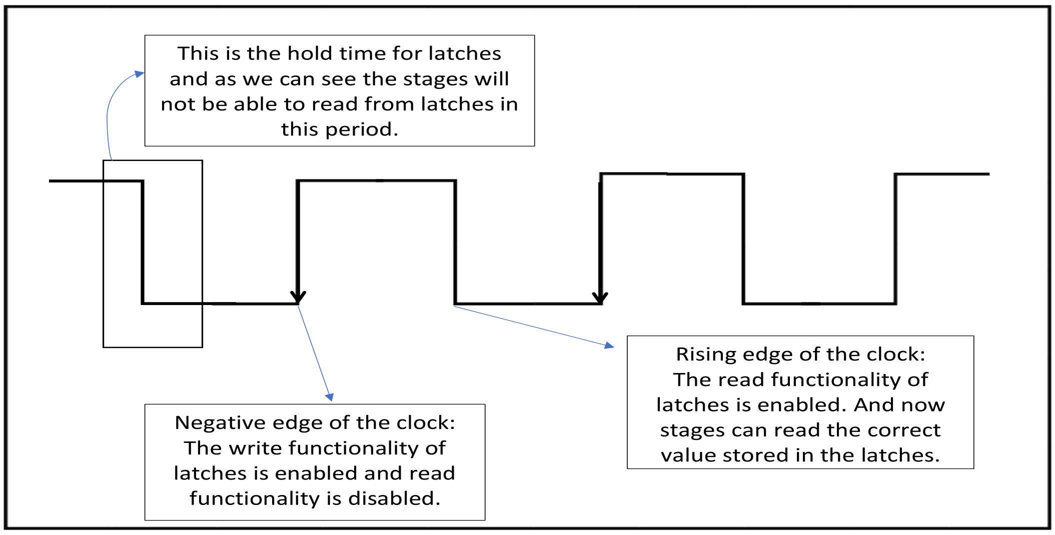
1. Every latch will have it's read and write port.

2. Every latch can be disabled using a signal.

3. We disable the read functionality of the latches on falling or negative edge of the clock.

4. Now, when the negative edge of clock comes, the values of instruction packet will be latched or stored in L1 and L2. Since the read functionality has been disabled, the stage after L2 won't be able to read the values stored in L2 even if they are wrong values, wrong results will not be computed in this stage.

5. In the next cycle, when the positive edge is encountered, every stage reads it's corresponding instruction packet from the its register/ pipeline latch. These values will be correct because the latches have not been read for time¿ hold time.


This is the hold time for latches and as we can see the stages will not be able to read from latches in this period.
Negative edge of the clock: The write functionality of latches is enabled and read functionality is disabled.
Rising edge of the clock: The read functionality of latches is enabled. And now stages can read the correct value stored in the latches.

The diagram above, shows how enabling and disabling read/write functionality of latches can remove violations of hold time.

Question 4

Answer Marks: 8

We can have the following forwarding paths in an ISA where the memory can be accessed in OF stage:

1. RW –>MA
2. RW –>EX
3. RW –>OF
4. MA –>EX
5. MA –>OF
6. MA –>EX

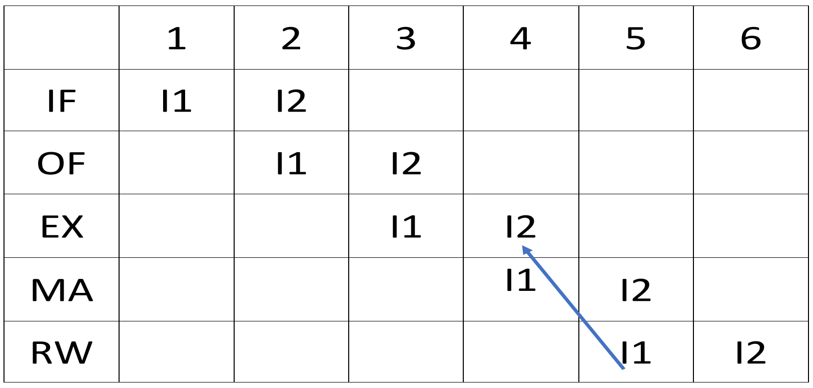
Dependencies that can arise because of the above forwarding paths and the fact that OF accesses memory are:

1. Data hazards: Read after write Hazard will still occur. However, WAR and WAW can not occur. Example:

   add r1, r2, r3

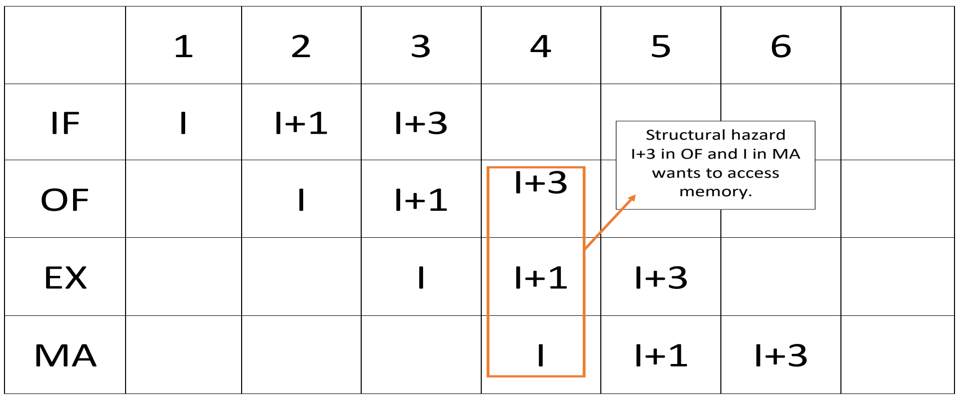
   sub r4, r1, r2

   In the example above, instruction 2 tries to read the value written by instruction 1, so it will lead to a RAW hazard.



	1	2	3	4	5	6
IF	I1	I2				
OF		I1	I2			
EX			I1	I2		
MA				I1	I2	
RW					I1	I2

2. Structural Hazards: Structural hazards will occur in such a pipeline because two instructions will try to access the memory unit at the same time. Example: If instruction i tries to access the memory unit in OF stage and instruction i+2 tries to access memory in MA stage, it will lead to a conflict of resources and thus a structural hazard.


Structural hazard I+3 in OF and I in MA wants to access memory.

	1	2	3	4	5	6	
IF	I	I+1	I+3				
OF		I	I+1	I+3			
EX			I	I+1	I+3		
MA				I	I+1	I+3	

In the diagram above, in clock cycle 4 the instruction i is in OF stage and wants to access the memory and instruction i+2 is in MA stage and wants to access the memory. This leads to a resource conflict and thus a structural hazard.

3. Control Hazards: Control Hazards will be present in the pipeline because the branch condition (if branch is taken or not) will be known in the 3rd which is the EX stage and this will require 2 stalls.

No, forwarding can only eliminate the RAW hazards. Forwarding will not be able to eliminate structural hazards as well as load use hazards. Load use hazards are data hazards when an instruction which is loading a value in a register is followed immediately by an instruction which wants to read the value of the same register. Such hazards will not be eliminated using forwarding.

Question 5

Answer Marks: 7

In such a scenario, the following can be done:

1. Using the concept of instruct commit.
2. Instructions will commit whatever changes they have made, only if they complete execution in all the stages.
3. If an instruction j is in RW stage, an instruction j+1 will be in the MA stage.
4. If now, an interrupt or exception occurs, then by the rule of precise exception the instruction in MA will be marked.
5. But now, we introduce a bit 'r' which tells whether the exception has occurred in RW stage or not. If the exception is in RW stage, r=1. Else it is 0.
6. If r=1, then the instruction j+1 will not be marked. In this case instruction j will be marked.
7. Now, a confusion may arise that the instruction in j+1 would have already made changes in memory in MA stage. But, given the fact that we are using commit concept, we can easily ignore the changes made by instruction j+1 and mark instruction j.
8. Now, after servicing the Interrupt service routine, we will come back to instruction j and execute it again.

	1	2	3	4	5	6	Commit stage
IF	I1	I2					
OF		I1	I2				
EX			I1	I2			
MA				I1	I2		✗
RW					I1	I2	

Exception occurred here.
Mark I1 and save it's address in OldPC
Do not commit I2

Question 6

Answer Marks: 9

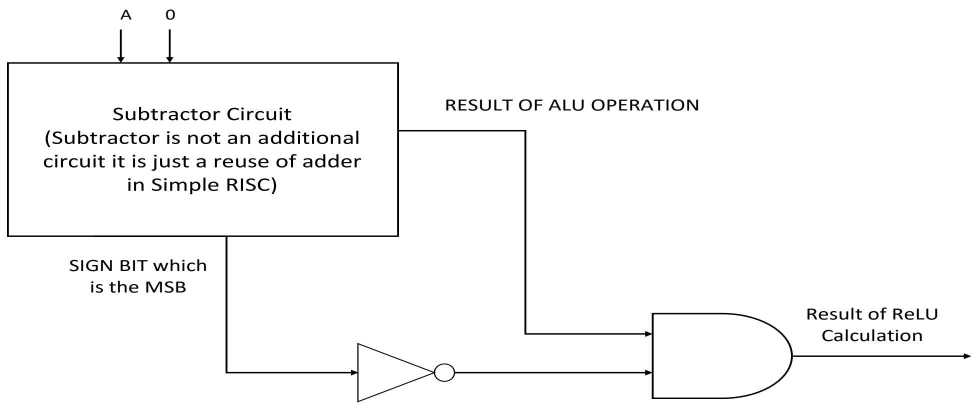
Dot Product:

1. Assuming that the two arrays or vectors whose dot product is to be calculated, are stored at locations l1 and l2.

2. Now, at a third location l3, I store the multiplications of the corresponding index of the vectors l1 and l2. i.e at l3[x] = l3+ (l1[x] * l2[x])
3. Now, since this dot product calculation will take 0(n) time for calculating dot products of two different vectors, we need not make changes in the architecture.
4. Instead, we can create a function for calculating dot product and provide it as an instruction in the extended architecture.
5. The instruction can be a 3- address instruction which can utilize the existing instruction format like: dot r1, r2, r3 Here, r1 will store the dot product. R2 stores the address of vector1 and r3 stores the address of vector 3.
6. Making hardware changes for dot product calculation will bring unnecessary complexities into pipeline. We cannot even divide EX stage into multiple stages for each multiplication calculation because the dimensionality of vector is unknown. If we consider a vector of size 100 and divide EX into 100 equal stages, then this method won't work for a vector of length say 150.
7. The dot instruction whenever found in the assembly language can then be unrolled as follows: Assuming that addresses of vectors are stored at r1 and r2. .dot: mov r0, 0 ld r3, [r1] ld r4, [r2] mul r4, r4, r3 add r0, r0, r4 add r1, r1, 4 add r2, r2, 4 cmp r5, r1 bgt .dot ret This method takes O(n) time which is n multiplication operations and n addition operations. By the end of this procedure, dot product will be saved in r0.
8. However, there can be a way to speed up the calculation of dot product. This can be done by using Single Instruction Multiple data parallelism but for this we require parallel programming which is not supported by the Simple RISC.

ReLU:

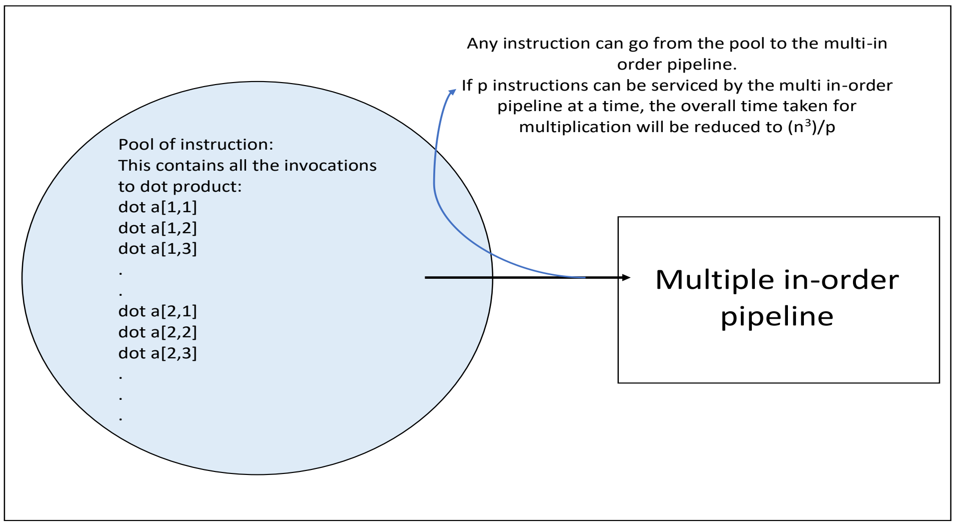
For computation of ReLU operation, an additional unit can be added in the ALU known as ReLU unit. Which will look like the diagram below:


A
0
Subtractor Circuit
(Subtractor is not an additional circuit it is just a reuse of adder in Simple RISC)
RESULT OF ALU OPERATION
SIGN BIT which is the MSB
Result of ReLU Calculation

Matrix multiplication:

1. Similar to the dot product procedure, a procedure can be written to calculate the matrix multiplication.
2. Matrix multiplication requires O(n^2) dot product calculations. Since every dot product is calculated in O(n) time. Matrix multiplication is done in O(n^3) time.
3. The following algorithm can be used for calculation of matrix multiplication: multiply: for every row r in resultant: for every column c in resultant: find dot product of vectors at r and c.

4. Here we can make use of Multiple in-order pipeline concept. If more than one in-order pipelined processor is available, we can speed up the calculation. Since, in matrix multiplication, computation of any two cells of resultant matrix are independent, all the calculations will be independent and all these instructions can be executed in parallel.


Any instruction can go from the pool to the multi-in order pipeline.
If p instructions can be serviced by the multi in-order pipeline at a time, the overall time taken for multiplication will be reduced to (n³)/p
Pool of instruction:
This contains all the invocations to dot product:
dot a[1,1]
dot a[1,2]
dot a[1,3]
dot a[2,1]
dot a[2,2]
dot a[2,3]
Multiple in-order pipeline

Now, since ReLU unit requires 2 cycles for EX stage, if we divide EX into EX1 and EX2 and use two ALUs, we can not only complete ReLU faster but we can also achieve speed in matrix multiplication and dot product calculation too.

If we divide dot product calculation into two ALUs where ALU1 calculates dot product of lower half and ALU2 calculates dot product of upper half and then add up these two sums, we can double up the speed.