1 a) Different pipeline hazards possible in a pipelined processor –

i) Structural Hazards – As the name suggests this type of hazard is related to the actual hardware present in the processor. In structural hazards the hardware resources cannot support the combination of certain instructions which we want to execute in the same cycle. As a result the pipeline needs to be stalled before the next instruction proceeds.

ii) Data Hazards – Data hazards are usually created when the result of the earlier instruction is used by the later instruction. There is dependence of one instruction on an earlier one that is still in pipeline. Certain forwarding logic could be used to avoid pipeline stalls when such a scenario is generated.

iii) Control Hazards – Such hazards are related to branch/jump instructions present in the processor. The control hazards are raised as there is certain need to make a decision based on the result of one instruction while other instructions are executing.

b) Assembly language code –

I1: ADD R4, R0, R8

I2: MUL R2, R0, R4

I3: ADD R4, R4, R8

I4: MUL R8, R4, R2

The following hazards are possible in the above instruction stream -

The following assumes a 5-stage pipeline implementation

Data Hazards – I1 – I3, I2 – I4

As the source register of I3/I4 is dependent on the result of I1/I2 instructions respectively.

c) Tomasulo's algorithm –

i) Given state of the structures at Cycle 0:

Rename	Tag	Val	Tag	Val
Adder				

Rename	Tag	Val	Tag	Val
Mult				

Register Name	Busy	Tag	Data
0			6
2			5
4			10
8			7.8

ii) After ADD commits and starts execution

Rename	Tag	Val	Tag	Val
A	-	6	-	7.8
Adder				

Rename	Tag	Val	Tag	Val
Mult				

Register Name	Busy	Tag	Data
0			6
2			5
4	Yes	A	-
8			7.8

iii) ADD is executing (2nd stage) MULT is in RS waiting for ADD to complete

Rename	Tag	Val	Tag	Val
A	-	6	-	7.8
Adder				

Rename	Tag	Val	Tag	Val
X	-	6	A	-
Mult				

Register Name	Busy	Tag	Data
0			6
2			5
4	Yes	A	13.8
8			7.8

iv) ADD completes, MULT proceeds to execution and next ADD also proceeds to execution

Rename	Tag	Val	Tag	Val
A	-	13.8	-	7.8
Adder				

Rename	Tag	Val	Tag	Val
X	-	6	-	13.8
Mult				

Register Name	Busy	Tag	Data
0			6
2	Yes	X	-
4	Yes	A	-
8			7.8

v) ADD completes, MULT proceeds to second stage and next MULT is in RS

Rename	Tag	Val	Tag	Val
	-		-	
Adder				

Rename	Tag	Val	Tag	Val
X	-	6	-	13.8
Y	-	21.6	X	-
Mult				

Register Name	Busy	Tag	Data
0			6
2	Yes	X	-
4		-	21.6
8	Yes	Y	-

vi) MULT completes and next MULT proceeds to execution

Rename	Tag	Val	Tag	Val
	-		-	
Adder				

Rename	Tag	Val	Tag	Val
Y	-	21.6	-	82.8
Mult				

Register Name	Busy	Tag	Data
0			6
2			82.8
4		-	21.6
8	Yes	Y	-

Finally, the following 2 cycles MULT proceeds and completes the result in R8 – 1788.5

2) We have an in-order pipelined system with branch delay slot –
Floating point multiply – 4 cycles
Floating point divider – 8 cycles
Floating point adder – 2 cycles
Integer operations – 1 cycle
Memory load-store – 3 cycles
No delay between integer instruction and dependent branch instruction
The given filter code -

```
FILTER: LDF     F3,     0(R1)
        MULTF   F10,    F3,     F0
        LDF     F4,     4(R1)
        ADDF    F11,    F4,     F10
```

```
        DIVF    F12,    F11,    F1
        STF     0(R2),  F12
        ADDI    R1,     R1,     #8
        ADDI    R2,     R2,     #8
        SUBI    R3,     R3,     #1
        BNE     R3,     FILTER
        NOP
```

a) Number of cycles the code would take –

```
FILTER: LDF     F3,     0(R1)
        2 –cycle stall (load dependency on F3)
        MULTF   F10,    F3,     F0
        LDF     F4,     4(R1)
        2 –cycle stall (MULTF dependency on F10)
        ADDF    F11,    F4,     F10
        1 –cycle stall ADDF dependency on F11)
        DIVF    F12,    F11,    F1
        7 –cycle stall (DIVF dependency on F12)
        STF     0(R2),  F12
        ADDI    R1,     R1,     #8
        ADDI    R2,     R2,     #8
        SUBI    R3,     R3,     #1
        BNE     R3,     FILTER
        NOP
```

Total cycles = 11 instructions + 12 stall cycles = 23 cycles

b) One of the possible solutions for the above code –

```
FILTER: LDF     F3,     0(R1)
        LDF     F4,     4(R1)
        1 –cycle stall (LDF dependency on F3)
        MULTF   F10,    F3,     F0
        ADDI    R1,     R1,     #8
        2 –cycle stall (MULTF dependency on F10)
        ADDF    F11,    F4,     F10
        1 –cycle stall ADDF dependency on F11)
        DIVF    F12,    F11,    F1
        SUBI    R3,     R3,     #1
        6 –cycle stall (DIVF dependency on F12)
        STF     0(R2),  F12
        ADDI    R2,     R2,     #8
        BNE     R3,     FILTER
        NOP
```

Total cycles = 11 instructions + 10 stall cycles = 21 cycles

3) We have an in-order pipelined system with branch delay slot –
Floating point multiply – 4 cycles
Floating point adder – 2 cycles
Integer operations – 1 cycle
Memory load-store – 3 cycles
Branches – 2 cycles
No delay between integer instruction and dependent branch instruction
The given code –

```
Loop:  LD      F5,   0(R1)
       LD      F6,   0(R2)
       MULTD   F7,   F5,   F6
       ADDD    F4,   F4,   F7
       ADDI    R1,   R1,   #8
       ADDI    R2,   R2,   #8
       SUBI    R3,   R3,   #1
       BNEZ    R3,   Loop
```

a) The number of cycles the original code would take -

```
Loop:  LD      F5,   0(R1)
       LD      F6,   0(R2)
       2 –cycle stall (LD dependency on F6)
       MULTD   F7,   F5,   F6
       3 –cycle stall (MULTD dependency on F7)
       ADDD    F4,   F4,   F7
       ADDI    R1,   R1,   #8
       ADDI    R2,   R2,   #8
       SUBI    R3,   R3,   #1
       BNEZ    R3,   Loop
       1 –cycle stall (branch delay slot)
```

Total number of cycles = 8 instructions + 6 stall cycles = 14 cycles

b) One of the possible solutions for the above code after re-arranging -

```
Loop:  LD      F5,   0(R1)
       LD      F6,   0(R2)
       ADDI    R1,   R1,   #8
       ADDI    R2,   R2,   #8
       MULTD   F7,   F5,   F6
       SUBI    R3,   R3,   #1
       BNEZ    R3,   Loop
```

1 –cycle stall on F7 (MULTD)

```
ADDD    F4,    F4,    F7
```

Total number of cycles = 8 instructions + 1 stall cycles = 9 cycles

c) Unrolling the given code once and avoiding any stalls by scheduling -
Here is the updated code -

```
Loop:  LD      F5,    0(R1)
       LD      F6,    0(R2)
       LD      F8,    8(R1)
       LD      F9,    8(R2)
       MULTD   F7,    F5,    F6
       ADDI    R1,    R1,    #16
       MULTD   F10,   F8,    F9
       ADDI    R2,    R2,    #16
       ADDD    F4,    F4,    F7
       SUBI    R3,    R3,    #2
       BNEZ    R3,    Loop
       ADDD    F4,    F4,    F10
```

For the second iteration, F5 is used as F8, F6 is used as F9 and F7 is used as F10. Notice the adjustments done to the pointers in the ADDI and SUBI instructions.
All the stalling slots have been utilized and there are no stalls.
Total number of cycles = 12 instructions = 12 cycles
But effectively one iteration of this code produces two results, therefore the total number of cycle for a single result would be 6 cycles.

4)
a) Consistency – The consistency aspect determines when a written value will be returned by a read. The coherency guarantees that each processor would have a coherent view of the shared data memory but it doesn't say when that would be available. This is what is determined by consistency. The issue of exactly when a written value must be seen by the reader is defined by consistency.
b) Coherency – In most of the multi-processor systems caching of shared data introduces the concept of coherency. In simple terms it refers that the shared data across various processor in the multi-processor system should be consistent i.e. every processor should be able to observe the updates done to the shared data. As a result of this in a multi-core system there needs to be some level of cache memory shared which is typically L3 in the modern processors.

Coherence and consistency are complementary: Coherence defines the behaviour of reads and writes to the same memory location, while consistency defines the behaviour of reads and writes with respect to accesses to other memory locations.

b) Sequentially consistent – The simplest memory model which offers consistency is referred as the sequential consistency. This memory model of sequential consistency requires that the result of any execution be same as if the memory accesses executed by the processor and are kept in program

order. This helps in removing the possibility of some non-obvious execution as the assignments must be complete before the next execution can proceed.
This is desirable for a multi-processor system as it resolves the above mentioned non-obvious execution problem between processors.
Conditions required to implement this type of memory model -
i) Processor needs to delay the completion of any memory access until all the invalidations related to that memory access are completed.
ii) The two accesses which needs to be ordered must be to different memory locations.

c) Synchronization is required in a multi-processor system as it ensures that all accesses to shared data are ordered by synchronization operations. The data reference is ordered in a synchronization operation if a write to a variable by one processor and an access of that variable by another processor is separated by few synchronization operations (which could be some instructions). The synchronization is very important for the processor as it guarantees data-race-free, i.e. we could be sure that the write is observable by every other processor and there aren't any data-races created. The key ability we need to implement such a synchronisation in a multiprocessor system is a set of certain hardware primitives. These hardware primitives should have the ability to atomically read and modify a memory location.

d) The following is code run on the two processors -

Processor 1	Processor 2
X = X+2	Y = X+2
Y=X+Y	

Here are the following which are the possible scenarios without using synchronization for consistency –

Sequence 1 -
X = X+ 2 => X=2
Y = X+2 => Y = 4
Y = X + Y => Y = 6

Sequence 2 -
Y = X + 2 => Y = 2
X = X + 2 => X = 2
Y = X + Y => Y = 4
As we can see without consistency there is no guarantee that the values 'Y' would get would be same.
On assuring sequential consistency we would always get sequence 1 executed (as there would be synchronization between the two processors and we could be assured that the final value of Y would be 6 and X would be 2).

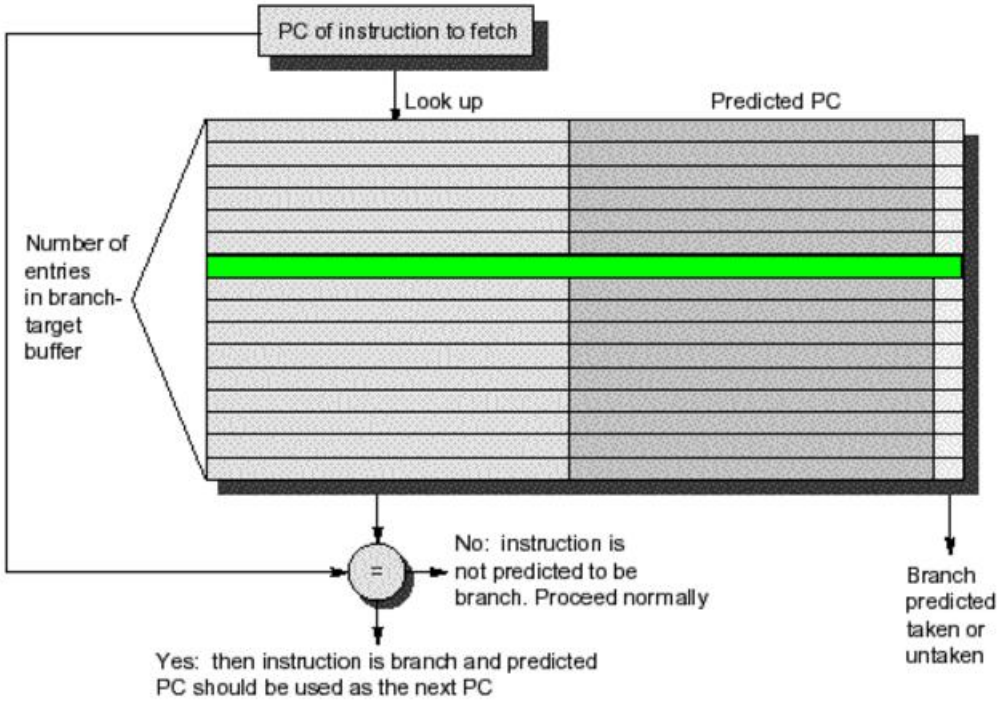
5)
a) BTB stands for Branch Target Buffer. It is used by the branch predictor to indicate the target address of the predicted branch. It is data structure in hardware which holds certain bits of the program counter (those bits depend on the implementation of the predictor) which is used to index the branch target buffer and it also contains the target program counter for the predictor branch. If

the branch predictor, predicts certain branch to be taken, then the target address is supplied by the branch target buffer to update the next program counter information.

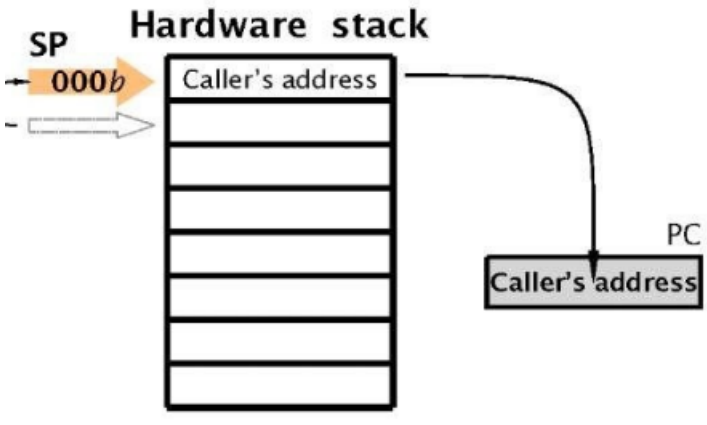
A BTB is quite different from Return Address Stack. As for the later, the name only suggests that it is a stack which holds the return address. The stack is accessed on a special function return instructions and the return address is popped from the stack. There is less prediction involved when compared to the BTB. Both BTB and RAS are accessed for different operations and are used for providing different branch predictions.

Here are the structure of BTB and RAS -

Branch target buffer (BTB) –

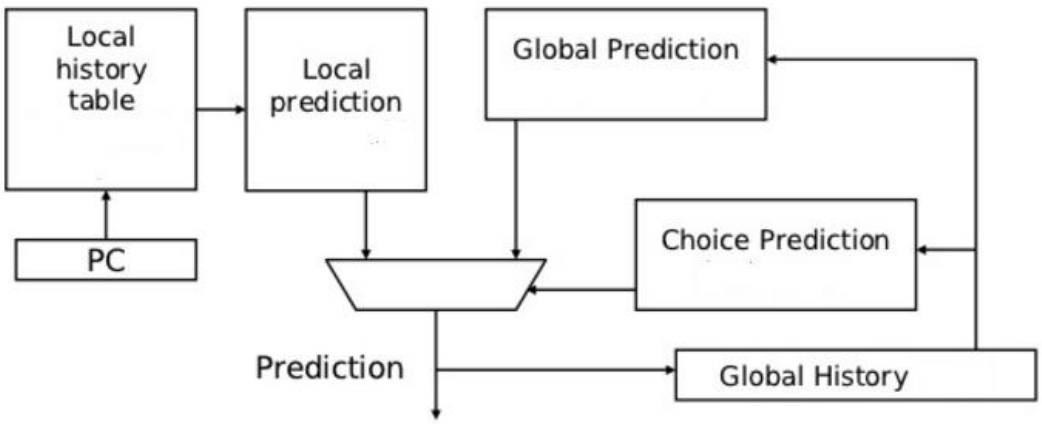


Return Address Stack (RAS) –



b) ROB stands for re-order buffer whereas RS stands for reservation station. These both hardware components are used in an out-of-order core and help the processor to maintain the program order during execution. The re-order buffer as the name suggests helps in re-ordering instructions such that they maintain the program order when they are visible to the programmer. This is necessary as otherwise the instructions would get committed in any order and this would mean a lot of trouble to the programmer once he has to debug the program. The reservation station on the other hand is a place where the instructions rest if they cannot be issued right away due to the some of their required operands aren't available. This ensures that other independent instructions could proceed ahead of execution even when there is an earlier instruction which dependent and blocked (rests in the reservation station). The ROB contains the valid bit and as well as the instructions destination register on the other hand the reservation station contains all the operand registers of the instructions along with the valid bit for the entry and all the operand register. The instruction leaves the reservation station once all its operand entries have become valid.

c) Here is the architecture of the Tournament predictor –



The basic components of a tournament predictor are –
i) Local History Table – Contains the prediction information of the branch correlated with the taken-ness of that particular branch and is indexed using the PC.
ii) Global History Register – Contains the correlation information of a particular branch with all the other branches. It is used to index into the Global predictor.
iii) Global Predictor – A table indexed by the global history register which gives the prediction information of the particular branch.
iv) Mux – The mux selects between the local prediction and the global prediction and is dependent on the confidence of prediction (choice pattern) of the global prediction.

d) The following are the differences between statically scheduled superscalar processor and VLIW -
i) Order of execution – For a statically scheduled superscalar processor there might one or more instructions in flight but the instruction will always be in-order as of the program. For VLIW, it is possible to issue a fixed set of number as either one large instruction or as a fixed parallel package.
ii) Issue width – Since the statically scheduled processor may schedule a varying number of instructions dependent on the statically scheduled compiler code and as well as on the issue wifth of these processors as different widths provide different functionalities. On the other hand the VLIW

can group as such instructions together and issue them altogether giving more performance with the multiple independent functional units.
iii) Parallelism – To gain more out of an VLIW processor we need almost all the independent functional units to remain busy throughout as otherwise we would not be utilizing all the hardware present in the processor. As a result of this it becomes an additional overhead on the designer to write the code with enough parallelism in it. There are no such needs for a statically scheduled processor as it is independent of such parallelism.

e) Multithreading is the capability of the processor to execute multiple processes or threads simultaneously. In case of the multithreaded system, some of the resources are shared by the the two threads running concurrently on the processor. The computing units like caches, translation lookaside buffer, etc are shared across the threads supported by the single processor. Such processors try to maximize the use of thread-level along with instruction level parallelism in them. The main difficulties which multithreading CPU introduces is the now we need to have a system to communicate across threads as well. One way to avoid this could by the use of extra hardware. It is possible that the multi-threaded CPU have register files per thread which again poses more hardware requirements. Not only register files but certain different computing units like the ALU, etc might also be needed to be replicated in order to have the threads function properly. Also, such multithreaded CPUs would work well when the OS is aware about the functionalities supported by the underlying processor. As a result the processor performance also depends on the ability of the OS to perform multiple tasks by spawning appropriate threads.
The basic multithread design options -
i) Fine grained multithreading – In this approach the hardware switch threads on each clock causing the execution of instructions from the multiple threads to be interleaved. Such interleaving may be done in round-robin fashion which skips inactive threads. Example processor – Sun Niagara processor
ii) Coarse grained multithreading – In this approach the hardware switch to other thread only on hard stalls such as a TLB miss followed by a cache miss. In this the hardware switches is not very often and is much less likely to slow down the execution speed.
iii) Simultaneous Multithreading – This approach is the variation on fine-grained multithreading which comes naturally when it is implemented on a multiple issue dynamically scheduled processor. It uses thread-level parallelism to hide long-latency in the processor thus increasing the usage of all the functional units present in the processor.

6)
a) Branch sequence – T, T, T, N, N, T, T, T, N, N, T, T, T, N, N
The 2-bit predictor is used and it starts at the following state – Predict Not Taken

Branch Sequence	Prediction	Predictor Next State
T	Not taken	Predict Not taken
T	Not taken	Predict Taken
T	Taken	Predict Taken
N	Taken	Predict Taken
N	Taken	Predict Not taken
T	Not taken	Predict Taken
T	Taken	Predict taken

T	Taken	Predict Taken
N	Taken	Predict Taken
N	Taken	Predict Not taken
T	Not taken	Predict Taken
T	Taken	Predict taken
T	Taken	Predict Taken
N	Taken	Predict Taken
N	Taken	Predict Not taken

Number of correct predictions = 5
Prediction accuracy = 5/15 = 33.33%

b) Code –

```
if (x is even) then              ;(branch b1)
        incr a                   ;(b1 taken)
if (x is multiple of 10) then    ;(branch b2)
        incr b                   ;(b2 taken)
```

Same 2-bit predictor is used. Initial state => NT (00)

X =	8	9	10	11	12	20	29	30	31
B1 predicted	NT	NT	NT	T	NT	NT	T	NT	NT
B1 actual	T	NT	T	NT	T	T	NT	T	NT
B2 predicted	NT	NT	NT	NT	NT	NT	NT	T	NT
B2 actual	NT	NT	T	NT	NT	T	NT	T	NT

Accuracy for B1: 2/9 = 22.22%
Accuracy for B2: 6/9 = 66.67%

Overall prediction accuracy = 8/18 = 44.44%

7)

a) i) P0: read 120
Updates -
P0.B0 : (S, 120, 00 20)
ii) P0: write 120 ← 80
P0.B0 : (M, 120, 00 80)
P3.B0 : (I, 120, 00 20)
iii) P3: write 120 ← 80
P0.B0 : (I, 120, 00 80)
P3.B0 : (M, 120, 00 20)
iv) P1: read 110

P1.B2: (S, 110, 00 30)
P0.B2: (S, 110, 00 30)
v) P0: write 108 <- 48
P0.B1: (M, 108, 00 48)
P3.B1: (I, 108, 00 08)
vi) P0: write 130 <- 78
P0.B2 (M, 130, 00 78)
Memory update – 110 <- 00 30
vii) P3: write 130 – 78
P3.B2 (M, 130, 00 78)

b) Let LL be load linked instruction and SC be store condition instruction -

```
MOV   R3, #0
LL    R2, 0(R1)
SC    R3, 0(R1)
```

Typically, the LL/SC code is used in a loop that loops until 1 is returned in R3.