Question-1 (15 points)

Define the following terms (1 sentence per term):

a. **[3 points]** Instruction Commit:

Updating the register file and the memory with the changes buffered in the reorder buffer/LSQ.

b. **[3 points]** Reservation Station:

A hardware that holds instructions until the source registers are ready and dispatches ready instructions out-of-order.

c. **[3 points]** Reorder Buffer:

A hardware that holds the results of the instructions until they are committed in-order.

d. **[3 points]** Out-of-order execution:

Executing instructions out of program order.

Question-4 (35 points)

(b) **[4 Points]** Give examples of RAW, WAW and WAR dependences using MIPS assembly instructions. Which one of RAW, WAW and WAR is termed as "true dependence"? Explain why.

RAW:

```
add $1, $2, $3
add $4, $1, $5
```

WAR

```
add $1, $2, $3
add $2, $4, $5
```

WAW

```
add $1, $2, $3
add $1, $4, $5
```

RAW is a true dependence because the dependence cannot be eliminated. WAR and WAW can be eliminated with register renaming.

(c) **[8 Points]** Consider the following code sequence:

```
lw $t4, 0($t3)
add $t4, $t0, $t4
sub $t4, $t4, $t0
lw $t8, 0($t4)
add $t8, $t8, $t8
sub $t4, $t4, $t8
```

Assume that initially, $t0=0, $t1=1, $t2=2, $t3=3, $t4=4, $t8=8

Rename instructions using reservation station slot name RS0, RS1, RS2, … . Assume that all the RS slots are equivalent, and we have an infinite number of slots. Explain why register renaming is used in a computer architecture.

```
lw $RS0, 0(3)
add $RS1, 0, $RS0
sub $RS2, $RS1, 0
lw $RS3, 0($RS2)
add $RS4, $RS3, $RS3
sub $RS5, $RS2, $RS4
```

Register renaming eliminates WAR and WAW dependences, allowing more options of reordering.

(d) **[9 Points]** Consider the following code sequence, and explain, in detail, how "store queue" and "load queue" can be used to improve the performance of this code sequence.

```
sw $t7, 12($t1)
```

```
lw $t2, 8($t1)
lw $t3, 12($t1)
sub $t4, $t5, $t6
sw $t9, 4($t7)
lw $t5,12($t8)
lw $t6, 8($t9)
sub $t10, $t5, $t6
```

The following three instructions :

```
sw $t9, 4($t7)
lw $t5,12($t8)
lw $t6, 8($t9)
```

cannot be reordered with conservative reordering because there can be a RAW dependence between the store and the load. With LSQ, we can aggressively reorder these instructions and later check whether the reordering was correct.

(e) **[4 Points]** Explain why we need "age logic" in store queue.

When multiple stores write to the same memory location and are buffered in the store queue, the following load must read the latest value, which is generated by the youngest preceding store.

(f) **[5 Points]** Explain the role of “loop unrolling” in optimizing code for VLIW architectures. What are the drawbacks of loop unrolling?

Loop unrolling allows more instructions to be considered for scheduling by the compiler, improving the generated schedule. Drawbacks: larger code size,

3. VLIW

Consider following MIPS instruction sequence:

```
lw $t0,0($s1)

add $t0,$t0,$t3
```

```
sw $t0,0($s1)

addi $s9,$s9,4

lw $t1,0($s9)

add $t1,$t1,$t3

sw $t1,0($s9)

addi $s1,$s1, 4
```

You are required to fill the bundles in the table below for a VLIW machine to have the minimum number of execution cycles. The **LEFT** part of the bundle (second column in the table) can be only an ALU or branch instruction, whereas the **RIGHT** part (third column in the table) can be only a load or store instruction. You are allowed to *reorder* independent instructions and change the offset of addressing. You are not allowed to combine instructions. Explain each instruction-to-slot mapping decision you make in sufficient detail (i.e., why you have decided so, couldn't instruction be scheduled in an earlier slot (cycle)?).

Cycle	ALU/Branch	Load/Store
1	addi $s9, $s9, 4	lw $t0, 0($s1)
2	nop	lw $t1, 0($s9)
3	add $t0, $t0, $t3	nop
4	add $t1, $t1, $t3	sw $t0, 0($s1)
5	addi $s1, $s1, 4	sw $t1, 0($s9)
6		
7		
8		
9		

4 Superscalar Architectures

Consider two different 4-issue superscalar architectures (each can issue a bundle of 4 instructions in a cycle) with MIPS ISA. In the first architecture, each slot (in a bundle) can be an instruction of any type (ALU, load/store, branch), whereas in the second architecture, the first slot can only be a branch instruction, the second slot can only be a load/store instruction, and the third and fourth slots can only be ALU instructions.

(a) What are the advantages of the first architecture over the second architecture?

It has more flexibility in scheduling, so will end up with a higher IPC.

(b) What are the advantages of the second architecture over the first architecture?

Possibly cheaper and lower-power hardware (no need to support multiple access to the memory, need only 2 ALUs instead of 4, etc).

(c) What might be the reason that the designers of the second architecture chose the specific mix of slots above (branch,load/store,ALU,ALU)?

Because branch and load/store are less common than ALU operation.

5 Loop Unrolling

Consider the following for-loop:

```
for i = 1 to 100 step 1             // i iterates from 1 to 100 with a step size of 1
  { a[i] =  b[i]+b[i+1]+b[i+2];     // a, b and c are arrays
    c[i] = a[i+1]+a[i+2]+a[i+3]; }
```

(a) Show (as a high-level code like above, **not** in assembly) the unrolled loop if the unrolling factor is 2.

```
for i = 1 to 50 step 2
  { a[i] =  b[i]+b[i+1]+b[i+2];
    c[i] = a[i+1]+a[i+2]+a[i+3];
    a[i+1] =  b[i+1]+b[i+2]+b[i+3];
    c[i+1] = a[i+2]+a[i+3]+a[i+4];  }
```

(b) Show the unrolled version if the unrolling factor is 4.

```
for i = 1 to 25 step 4
  { a[i] =  b[i]+b[i+1]+b[i+2];
    c[i] = a[i+1]+a[i+2]+a[i+3];
    a[i+1] =  b[i+1]+b[i+2]+b[i+3];
    c[i+1] = a[i+2]+a[i+3]+a[i+4];
    a[i+2] =  b[i+2]+b[i+3]+b[i+4];
    c[i+2] = a[i+3]+a[i+4]+a[i+5];
    a[i+3] =  b[i+3]+b[i+4]+b[i+5];
    c[i+3] = a[i+4]+a[i+5]+a[i+6];
}
```