Q1.

a)

Opcode	rs	rt	rd	sa	funct
	7	7	7		

Here 21 bits are taken by the three registers. And its also given that we need to expand opcode and funct fields to encode them.
Therefore, total length of the instruction would be greater than 32 bits. This new format ~~[illegible]~~ is enough for the increased number of instruction as the length of opcode is increased and also of registers so ~~[illegible]~~ increased variety of instructions can be performed.

b) 1) 8 registers

length of register field = $\log_2(8) = 3$

Opcode	rs	rt	rd	sa.	funct
6	3	3	3	5	6

Total length = 26 bits

2) 10 bit immediate constants.

opcode	rs	rt	imme.	sa	funct
6	5	5	10	5	6

Total length = 37 bits.

3) 128 registers

Length of register = $\log_2(128) = 7$

opcode	rs	rt	rd	sa	funct
6	7	7	7	5	6

Total length = 37 bits.

~~Q8~~ Ignoring 4 inst. before the loop, we see that outer loop has 3 inst before the inner loop & 2 after. The cycle needed to execute these are $4+5+4=13$ & $4+5=9$, for a total of 22 cycles per iteration i.e $22N$ cycles. The inner loop require $1+5+3+1+1+3=14$ cycles per iteration & it repeats N^2 times for total of $14N^2$ cycles.

∴ Total no of cycles = $22N + 14N^2$

Overall execution time = $\frac{N(22+14N)}{2 \times 10^9}$ ms

③ ori t1, t0, 25

④ The problem is that we are using PC relative addressing, so if that add. is too far away, we won't be able to use 16 bits to describe where it is relative to PC.

⑤

```
gcd: addi $sp, $sp, -4    # create 4word long stk frame
     move $t0, $sp        # simply track sp
     sw   $ra, 0($sp)     # save the Return Add
     beq  $a1, $zero exit_gcd   # if $a1=0 goto exitgcd
     div  $a0, $a1        # Lo = $a0/$a1; Hi = $a0 % $a1
     mfhi $t1             # $t1 = Hi;
     move $a0, $a1        # $a0 = $a1
     move $a1, $t1        # $a1 = $t1
     jal gcd              # go to gcd

exit_gcd: move $v0, $a0   # $v0 = $a0
     lw   $ra, 0($sp)     # restore RA
     addi $sp, $p, 4      # adjust stk ptr
     jr   $ra
```

⑥ One way to implement beqr $s0, $s1, $s2

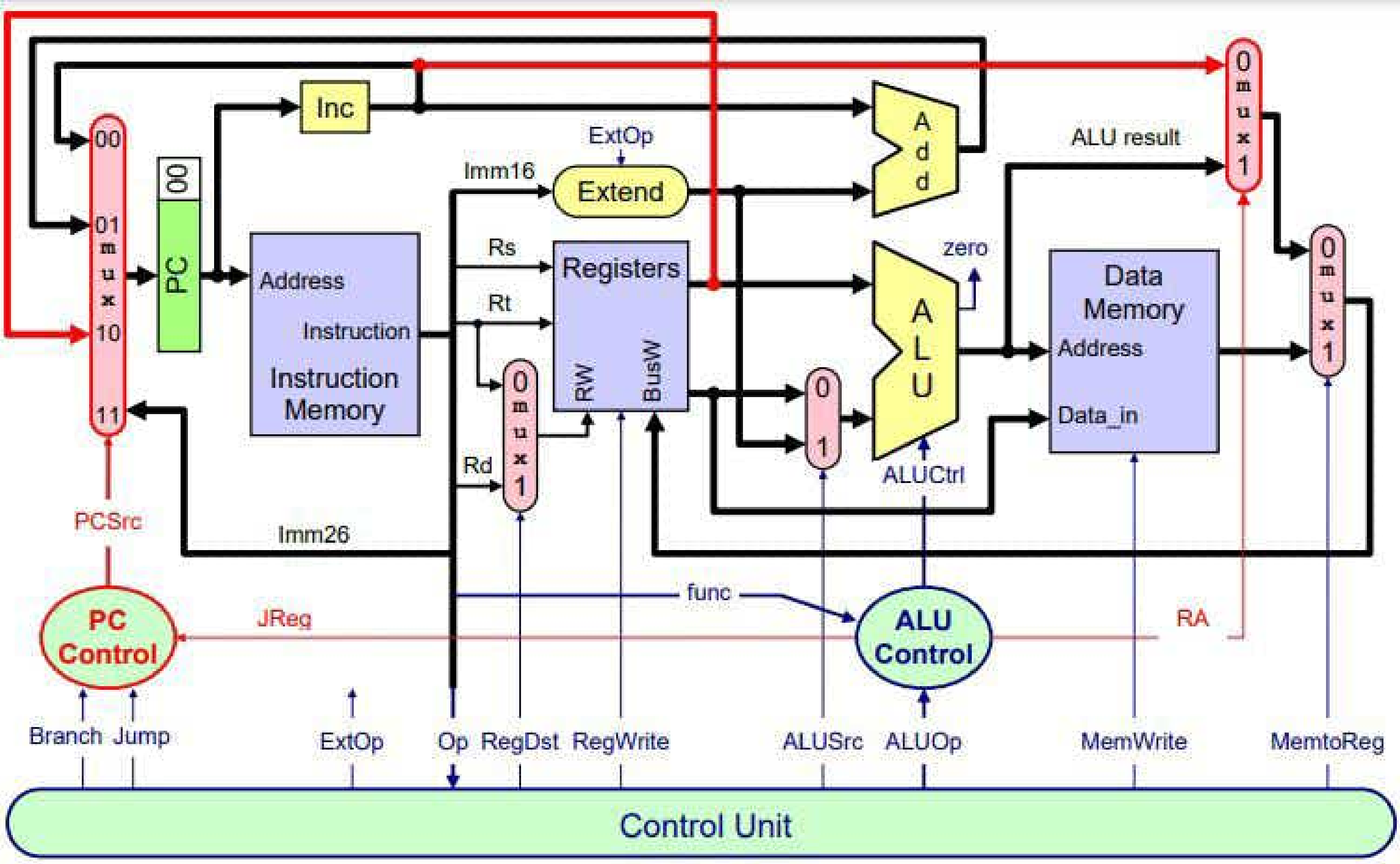
```
     bne  $s0, $s1, skip      // if (s0 != s1) => skip
     nop                      // nop
     jr   $s2                 // branch to $s2
     nop                      # nop

skip:
```

bene is same, replace bne with beq.

The inst. can be encoded as R type cor it has 3 register parameters
We can use opcode 0, & any funct value that has not yet been assigned.

For It reduce code size & impleement in HW can save a branch.
Against Though, It inc processor complexity, might slow down decode login/ other parts of pipeline & can be implement with existing instruction

7.


Inc
Add
ExtOp
Imm16
Extend
ALU result
0 mux 1
00 01 mux 10 11
PC
00
Address
Instruction
Instruction Memory
Rs
Rt
Rd
Registers
RW
BusW
0 mux 1
zero
ALU
0 1
ALUCtrl
Data Memory
Address
Data_in
0 mux 1
PCSrc
Imm26
func
PC Control
JReg
ALU Control
RA
Branch
Jump
ExtOp
Op
RegDst
RegWrite
ALUSrc
ALUOp
MemWrite
MemtoReg
Control Unit

The necessary changes to the datapath and control are shown in red.

For the datapath, we need a bigger 4-input multiplexer at the input of the PC. The first input is used to increment the PC. The second input is used for taken branches, where the branch target is PC-relative. The third input is used to jump register, where the input to the PC comes from a general-purpose register, and the fourth input is used for jump instructions.

Our focus here is on the implementation of the JALR instruction. Part of this instruction is to jump to register 'Rs', so we must ensure that we add a path from the output of register Rs (first ALU input) back to the PC multiplexer input. This path is shown in red for clarity.

We need a 'JReg' (Jump Register) control signal to jump according to the value of register 'Rs'. This signal is best generated by the ALU control logic, since it depends on the function field. This control signal is shown in red and is used as an input to the 'PC Control' logic. The 'PC Control' logic generates the 'PCSrc' control signal, which is used to control the 4-input multiplexer at the input of the PC. When JReg is equal to '1', PCSrc will be '10' to select the value of register Rs as input to PC.

We also need to store PC+4 in register Rd. To accomplish this, we need another multiplexer (shown in red) to select between the incremented PC and the ALU result to be placed on BusW. Again here, multiple solutions exist.

We must add a path from the output of the incremented PC to the input of this new multiplexer. This path is shown in red in the above diagram. Another control signal called 'RA' (Return Address) selects between the incremented PC and the ALU result. The MemtoReg multiplexer selects between the output of the 'RA' multiplexer and the Data Memory output to place on BusW.

The main control signals for the JALR instruction are the same for other R-type instructions, such as ADD and SUB. These control signals are shown in the table below:

Instr.	RegDst	RegWrite	ALUSrc	ALUOp	MemWrite	MemtoReg	Branch	Jump
JALR	Rd = 1	1	Rt = 0	R-type	0	0	0	0

The ALU Control signals for the JALR instruction are shown below. JReg = 1 and RA = 1. ALUCtrl is a don't care.

ALUOp	func	JReg	RA	ALUCtrl
R-type	JALR	1	1	X

8i. What is the total delay for each instruction class and the clock cycle for the single-cycle CPU design?

Instruction Class	Instruction Memory	Register Read	ALU Operation	Data Memory	Register Write	Total
ALU	190	150	190		150	680 ps
Load	190	150	190	190	150	870 ps
Store	190	150	190	190		720 ps
Branch	190	150	190			530 ps
Jump	190					190 ps
Mul/div	190	150	550		150	1040 ps

Clock cycle = 1040 ps determined by the longest delay.

ii. Assume we fix the clock cycle to 200 ps for a multi-cycle CPU, what is the CPI for each instruction class and the speedup over a fixed-length clock cycle? Note that this implies that multiply and divide operations will be performed in multiple cycles.

Instruction Class	CPI
ALU	4
Load	5
Store	4
Branch	3
Jump	2
Mul/div	6

Average CPI= 4*0.3 + 5*0.15 + 4*0.15 + 3*0.15 + 2*0.1 + 6*0.15=4.1
Note that we assumed that load and store instructions have equal percentage.
Speedup = 1040 ps / (4.1*200 ps) = 1.268.