① $s = 10$

speedup = 1.3

$$\text{speedup} = \frac{ET_{orig}}{ET_{new}} = \frac{1}{(1-f) + f/s}$$

$$1.3 = \frac{1}{(1-f) + f/10}$$

$$1 - f + f/10 = 1 - 9f/10$$

$$1.3 = \frac{1}{1 - 9f/10} \Rightarrow 1.3\left(1 - \frac{9f}{10}\right) = 1$$

$$1.3 - 1.17f = 1$$

$$f = 0.26 = \boxed{26\%}$$

② $$\text{speedup} = \frac{1}{(1 - 0.26) + 0.26/\infty} = \frac{1}{(1 - 0.26)} = \boxed{1.35}$$

($0.26/\infty \to 0$)

③ average cpi = $10(0.03) + 5(0.30) + 5(0.20) + 1(0.47)$

$= \boxed{3.27}$

④ data hazards: arise because of the nature of computation, that is when you need to use a value computed in one instruction in a later instruction (when does that value become available)

structural hazards: when there is contention for a shared hardware resource

control hazards: arise when a jump occurs, either conditional or unconditional, and you have to calculate whether to take the jump and where the target address is

⑤ DCA : 3000

0011 0000 0001 0000

© Superscalar processors generate conventional sequential machine code in the compiler, then at runtime, hardware figures out dependencies and issues instructions accordingly. The dependencies it works out of the code are generally anti and output dependencies. It does this by static single assignment (trying not to reuse register names). What's left are usually just data dependencies. This makes room for more instructions to run at the same time. Instructions are then fed into reservation stations according to instruction type. These reservation stations maintain information (including dependencies) and let instructions execute when their operands become available. This method respects dependencies and allows instructions to run parallel as long as these dependencies are respected.

⑦ No address or immediate : 1 dlx-word
Address : 2 dlx words
Immediate : 2 dx words

⑧ instruction 2 -> instruction 3 (RAW with R4)
instruction 1 -> instruction 3 (WAR with R2)
instruction 1 -> instruction 3 (RAW with R1)
instruction 3 -> instruction 4 (RAW with R2)
instruction 1 -> instruction 4 (RAW with R1)
instruction 3 -> instruction 5 (WAR with R4)
instruction 1 -> instruction 5 (RAW with R1)
instruction 3 -> instruction 5 (RAW with R2)
instruction 2 -> instruction 5 (WAW with R4)

(9) False

(10) 36 bit phys addr, cache BLOCK 2K bytes, cache is 1 MB, dm

2^{11} bytes, 2^{20} bytes

11 bit offset

$$2^{20}\ \frac{\text{bytes}}{\text{cache}} \Big/ \frac{2^{11}\ \text{bytes}}{\text{block}} = 2^9 = \frac{512\ \text{blocks}}{\text{cache}}$$

9-bit index

(11) $2K\ \frac{\text{bytes}}{\text{block}} = 2^{11}\ \text{bytes} =$ **11-bit offset**

(12) $36 - 9 - 11 =$ **16-bit tag**

(13) avg mem access time = hit time + miss rate x miss penalty

" instruction $1 + 0.01 \times 10 = 1.1$

" data $1 + 0.05 + 10 = 1.5$

$$\left(\frac{1}{1.30}\right) 1.1 + \left(\frac{0.30}{1.30}\right) 1.5 = \boxed{1.19}$$

(14) 2-bit branch predictor: with a 1-bit predictor the nested loops will continuously make two wrong predictions: ~~[illegible]~~ when the loop is on its last iteration it will predict to take the loop but will be wrong and when the loop has to run again it will predict not to take it but be wrong. With a 2-bit, the nested loops will only make one wrong prediction: when they are on ~~[illegible]~~ their current last iteration. But since it requires ~~to~~ two wrong predictions to change, when the loop has to run again, it will still choose to take the loop and be ~~wrong~~ right