Part 1

1.1 Comparing smaller instruction width with wider instruction width we find:

- wider instruction width == increased performance
- wider instruction width == increased power consumption, however lower voltages in silicon has meant VILW processes scale well compared to superscalar
- wider instruction width == additional silicon requirements as additional execution units must be provided
- wider instruction width == more complicated compiler back-ends as the compiler must organize the parallelism
- wider instruction width == potential generated code-bloat if application instructions or encoding cannot pack in instructions efficiently and a large number of nops are needed
  - can also cause increased memory bandwidth requirements if encoding not efficient

1.2 In comparing the 2-wide versus 4-wide width versions of the three provided programs, all showed a substantial improvement in the number of cycles required to executions.

- imgpipe - 20.9439% decrease
- madplay - 15.69616% decrease
- susan - 25.2738% decrease

Additionally, all three applications showed a decrease in the amount of generated output bytes required in the 4-wide versus 2-wide output.

- imgpipe - 4.37% decrease
- madplay - 3.86% decrease
- susan - 2.67% decrease

From these applications along, a 4-wide machine is a big benefit over a 2-wide machine. With an average of a 20% decrease in cycles required to run the application, an embedded implementation would likely see a reduction in power consumption given the fewer cycles required.

Part 2

2.1 - Comparing fixed-overhead encoding vs. template-based encoding schemes:

- fixed-encoding schemes are very simple to decode, whereas template-based encoding schemes are also simple to decode, but just slightly more complex
- template-based encoding requires additional complier complexity to support limited options for templates and picking the most appropriate code generation for those templates
- in a fixed-encoding scheme such as the masked-based encoding, the next instruction program counter is fetched based on the number of instructions the mask bits indicate (i.e. 0x10100111 would cause the processor to fetch the next instruction after 5 instruction words were processed)
- in a template-based encoding scheme, the next instruction program counter is fetched after all of the chained templates (if any) are processed; the width of each template is fixed and the processor moves forward and only is concerned if the template is chained to the next one or not

2.2 Assuming that the ;; instruction terminator does not count as a VEX syllable, then the uncompressed and compressed file size comparisons are as follows (all sizes in # of bits):

file	**uncompressed encoding**	**fixed-width encoding**	**% of uncompressed**	**template-based encoding**	**% of uncompressed**
bitcnts_8_wide	109056	28912	26.5%	27116	24.9%
bitcnts_4_wide	54528	27208	49.9%	26421	48.5%
basicmath_4_wide	69120	32048	46.4%	30944	44.8%
basicmath_8_wide	138240	34208	24.8%	31762	23.0%

As the table above shows, a significant object file reduction is possible with either encoding scheme. Of the two, the template-based encoding results in a smaller result, but not significantly so. The template-based encoding results in a smaller result because the absolute minimum number of bits is used to determine the encoding, so that many nop instructions wind up generating no overhead, whereas fixed-width encoding always has some overhead.

If the computational overhead of handling the templates were higher or more complex for the hardware architecture, I would lean towards a fixed-width encoding scheme.

Question 4

4. Given the following sequential list of instructions and instruction latencies:

> A. `sub $r2 = $r1, $r4`
> B. `store 4[$r3] = $r2`
> C. `load $r7 = 8[$r5]`
> D. `add $r9 = $r7, 20`

Cycle	0	1	2	3	4	5	6	7
	`sub $r2=$r1,$r4`							
			`store 4[$r3]=$r2`					
				`load $r7=8[$r5]`				
							`add $r9=$r7,20`	

I. CPI for these instructions is:

$$CPI = \frac{N_{cycles}}{N_{operations}} = \frac{8}{4} = 2$$

II. If we assume that memory access `4[$r3]` in operation B and `8[$r5]` in operation C do not collide, then we can move C and D freely to reduce overall cycle time. A schedule for this would be as follows, which reduces the total cycle count from 8 to 6.

The CPI then becomes:

$$CPI = \frac{N_{cycles}}{N_{operations}} = \frac{6}{4} = 1.5$$

Cycle	0	1	2	3	4	5
	`sub $r2=$r1,$r4`					
		`load $r7=8[$r5]`				
			`store 4[$r3]=$r2`			
					`add $r9=$r7,20`	