

Partial Solution For Exercises

Chapter 1

1. See p.2.
2. Virtual machines corresponds to layers of the real machine
3. To ease the thinking and to hid the complexities behind the lower layers.
4. Upper layers are more conceptual than lower ones, 1 instruction of upper layer can be translated into several instructions on the lower layer. Lower layers are faster and more powerful than the upper ones. Lower layers are usually more complex to program than upper ones.
5. You can argue something like atomic layers or mechanical layers.
6. Most computers are in ISA, only a few in microarchitecture level.
7. The code is faster, but it is bigger and more complex to program.
8. To hide the complexities from the lower layers and to manage resources of the system.
9. See p.11
10. To encapsulate some necessary commands to perform system-level tasks (see p. 11 for details).
11. See p.17-18 and notes
12. Examine figure 1-5 and 1-6

From Exercise

5. Each level we lose the speed of the factor of n . So, for level 1 the speed would be k , level 2, 3, 4 would be kn , kn^2 , kn^3 respectively.
6. Similar to number 5, but we lose of a factor n/m . So, the answer would be kn/m , kn^2/m^2 , kn^3/m^3 .
8. See page 8
10. Cycle time is not a sole factor in determining the overall performance ratio. Still there are other factors, such as maximum data channels, bytes fetched per cycle, etc. See table 1-7.

Chapter 2

1. You can argue that at least there are at least 2 needed: PC (Program Counter) and IR (Instruction Register). You can also argue about other necessary registers as well, such as accumulator, memory address register, memory buffer register, and flag. See page 39-40.
2. Usually no. We can use registers as temporary variable and use a couple register-memory instructions.
3. You cannot have instruction that has data in the memory.
4. The CPU cannot fetch the correct data.

5. It implies that all instructions takes only 1 word in size.
6. You have to modify the program so that we determine the type of instruction first then increment PC accordingly.
7. Yes, it is the same. However, in this case the interpretation is done by hardware, i.e. CPU.
8. Because more complex instructions often led to faster program execution (see p 43-44)
9. See p 46
10. Most cases the winner would be CISC because in one cycle CISC would do more instructions than RISC does.
11. I would choose a kind of flash memory (it would be weird though). The reason: see p.154.
12. The last two principles: See p 48-p 49
13. See p 50 – p 51
14. Because in nature, pipeline processor processes the instruction sequentially, using its sub-blocks optimally.
15. See p 55 – p 56
16. See p 53 – 54
17. If the error position $> m + r$
18. It's bit 8.
19. See p 65 – p 66
20. You can argue something like increasing cache size or invent a more effective new caching algorithm.
21. See descriptions p 69
22. Hard drive. Because hard drive's RPM is a lot faster and there is a delay for floppy drive to get up to speed.
23. No. Many of them are used for error checking and correcting. If we omit this part, then we have a lot more capacity but the data inside is more error prone.
24. Since 16,777,216 can be represented by 24 bits, so we need 3 bytes each pixel. Therefore the whole screen need $640 \times 480 \times 3$. Because we need to redraw the whole screen 30 times the bandwidth needed is: $640 \times 480 \times 3 \times 30$ per second. Then convert this number to MBps.
25. For a true color mapping, each entry needs 3 bytes. To map 65,536 colors at the same time will need $3 \times 64 \text{KB}$ of memory. This would be a mere waste of memory. A wiser approach would be mapping to 256 colors (which need only 768 bytes) or no mapping at all, i.e. show the entire colors.

Chapter 3

1. $A \rightarrow B = \neg A \vee B = \neg(A \wedge \neg B)$. It should be simple enough to implement then.
2. It is merely NOT(A XOR B). It should be simple enough to implement.
3. It is the same as constructing 4 bit adder with 1 bit-adders. But we first negate the second operand and add the result by one, i.e. making 2's complement number out of the negation of the second operand.
4. See p 140

5. See p 152
6. See p 640
7. Yes, it is possible in excess 128, since 0 represents -128, i.e. $-128 + 128 = 0$. Not in signed magnitude since the range is only -127 to +127.

Chapter 7

1. See p 485-488.
2. Only instruction 6, 7, 9, and 12 is invalid. Instruction 1 is merely forward reference. Doing arithmetic with constants in instruction 2, 3 and 5 is perfectly fine because the result is then pre-calculated by the assembler. Instruction 4 is perfectly fine too. You cannot do instruction 6 because the result of EAX cannot be predetermined by the assembler, therefore $EAX+D$ will result in a non-constant number. So it is invalid. It should be broken down into several instructions. Instruction 7 cannot be done: you cannot change the value of constants. Instruction 8 is OK, assigning memory to a constant. Instruction 9 and 12 is not allowed because we don't have memory-memory instructions. 10 and 11 is OK even the type is different. EAX would be 325, After instruction 11, $I = 0$, $J = 1$, $K = 69$.
3. No, it is not legal, therefore no result in EAX.
4. Only instruction 2 is illegal. Everything else is legal even for number 6. In number 6, it is assumed that we have a pre-compiled instruction with the code 200. Doing arithmetic with memory reference (in instruction 7) will result in moving pointers, i.e. $EDX = 5$.
5. The consequence is that we cannot pass both memory references as the parameter as we do not have memory-memory instruction (after the translation of `MOV X,Y`)
6. After instruction 1: $I = 4$, $J = 3$, $A = 7$, $K = 5$, $L = 2$, $B = 8$
 After instruction 2: $I = 4$, $J = 2$, $A = 8$, $K = 5$, $L = 3$, $B = 7$
 After instruction 3: $I = 4$, $J = 2$, $A = 3$, $K = 705H$, $L = 8$, $B = 8$
 After instruction 4: $EAX = 8$, $EBX = 3$, $ECX = 8$, $EDX = 3$
 After instruction 5: $EAX = 4$, $EBX = 4$, $ECX = 4$, $EDX = 4$
 After instruction 6: $EAX = 20$, $EBX = 20$, $ECX = 4$, $EDX = 4$
7. No, but it is merely an instruction to assembler. It is to accomodate conditional assembling.
8. $EAX = 15$. The labels that cause forward reference are: A, L2, RESULT.
 After first pass:

Symbol	Type	Value	Address
A	EQU	2*B	--
B	EQU	5	--
START	LABEL	--	1000
L1	LABEL	--	1008
L2	LABEL	--	1028
RESULT	DD	0	1036

In the second pass, the value A is not determined yet, since now the B value is known, we can solve forward reference problem for A in the next pass. Then the instruction `JE L2`. It was not be able to be converted at the first pass because at that time the address of L2 is not known. After the first pass finished, we can resolve this issue. `MOV RESULT, EAX` caused the same problem. Therefore, it is similarly solved.

If we forgot to define B, `MOV ECX, B` would be viewed as a forward reference problem too. By the end of pass 2, there is still an unresolved forward reference problem. And all of them relates to variable B. Therefore, the assembler will say an error that B is not defined.

9. The similarity is that both contains machine language. However, in object modules the codes are not linked yet. So the address is not resolved yet and they are likely to have external reference problem. Thus, object modules are not ready to run. After object modules are linked, they are formed into one executable binary program. So, executable binary program is just the ready-form of object module after linking.
10. All five modules are combined to one and then all external reference problems are resolved. M1 would start at 0, M2 would start at 400, M3 would start at 700, M4 would start at 1200, M4 would start at 1900. The reference problem would be as follows:

Variable Name	After-Linking Address
V1	100
V2	300
I	450
J	600
DATA	850
TEMP	1050
ARRAY	1800
STATUS	1950

Even the module is loaded at address 400 it should be no problem at all since the program is addressed by **relative address** of the starting address of the program.

Chapter 4

1. See p.207
2. Because they convey very important information. If the information was altered it will cause the system to chaos.
3. MAR inherently denotes the fetched address. It is not meant to be modified.
4. MBR inherently conveys the result of the operation to be written back. So, it is naturally write only.
5. See p.209