[6] 1. The following data has been loaded in memory locations `0x100` and following of the Y86 CPU:

```
0x100:  FF  2A  40  EE  30  F1  02  00
0x108:  00  00  50  71  00  01  00  00
0x110:  00  FF  47  00
```

(recall that our version of the Y86 CPU is a little-endian machine). The program counter is then set to `0x104`, and the program starts executing. What value will register `%edi` contain when the machine halts? You must justify your answer fully.

Solution : If we disassemble the code starting from address `0x104`, we get

```
irmovl $2, %ecx
mrmovl 0x100(%ecx), %edi
halt
```

The value stored in `%edi` will thus contain the data from addresses `0x102` to `0x105`, that is, it will be `0xF130EE40`.

[3] 2. Saving registers on the stack at the beginning of a function can require multiple `pushl` instructions, and it is error prone since we must use `popl` instructions in the reverse order. It would be nice if our Y86 CPU had `save` and `restore` instructions that save and restore all registers except `%esp` and `%eax` (the latter will contain the return value, so we don't want it restored to its initial value).

Explain briefly why this instruction can not be implemented using our design.

Solution : It would require multiple Memory stages (one for each register) because we can only read or write from one memory location at a time. However our CPU only has one Memory stage.

[8] 3. A student wants to implement an instruction `immovl V, (rB)` that stores a constant in the memory location whose address is contained in register `rB`. That, is, the semantics of this instruction is:

$$\texttt{M}_4\texttt{[R[rB]]} \leftarrow V$$

Use the notation described in class (and the book) to document each stage's role in implementing the instruction. That is, use the signal names (i.e., `PC`, `valC`, etc), and the notation `R[x]` for access to register number `x` and $\texttt{M}_\texttt{b}\texttt{[a]}$ for access to `b` bytes of memory starting at address `a`. List the behaviour of each stage separately:

Solution :

Fetch:

```
icode:ifun ← M1[PC]
rA:rB ← M1[PC+1]
valC ← M4[PC+2]
valP ← PC + 6
```

Decode:

```
valB ← R[rB]
```

Execute:

```
valE ← valB + 0
```

Memory:

```
M[valE] ← valC
```

Write-back:

```
# Do nothing
```

PC-update:

```
PC ← valP
```

[8] 4. Complete the following implementation of the `linear_search` function in Y86 assembly language. Hint: my solution uses 7 instructions, three of which are `addl`. You do not need to aim for this minimum, but if you use more than 12 instructions then you are probably making your solution more complicated than necessary. Part marks **will** be given for partially correct answers.

```
#
# Linear search:
# %eax: number of elements of the array.
# %ebx: address of the first element of the array
# %ecx: element we are searching for.
#
# returns %eax: the position of the element, or -1 if not found.
#
linear_search:
    irmovl $1,   %edx  # Decrement the position to search
    subl   %edx, %eax
    jl not_found
```

Your Answer goes here

```
    ret

not_found:
    irmovl $-1, %eax
    ret
```

Solution :

```
  rrmovl    %eax, %edx # Compute offset of element in bytes
  addl      %edx, %edx
  addl      %edx, %edx
  addl      %ebx, %edx # Now %edx is the address of the element.
  mrmovl 0(%edx), %edx # This is the element A[i]
  subl      %ecx, %edx # Compare to the one we want
  jne linear_search    # Not here, keep searching.
```