سوال ۱
الف)

```
# First fraction
mov R1, b
add R1, R1, c # R1 = b + c
div R1, a, R1 # R1 = a / (b + c)
mov X, R1
# Second fraction
mov R1, d
add R2, R1, a  # R2 = a + d
div R1, R2, R1 # R1 = d / (a + d)
add R1, R1, X  # R1 += X
mov X, R1
# Third fraction
mov R1, a
add R1, R1, d # R1 = a + d
div R1, R1, e # R1 = (a + d) / e
sub R1, X, R1 # R1 = X - R1
mov X, R1
```

ب)

```
# First fraction
mov R1, b
add R1, c  # R1 = b + c
mov R2, a
div R2, R1 # R2 = a / (b + c)
mov X, R2
# Second fraction
mov R1, d
add R1, a  # R1 = a + d
mov R2, d
div R2, R1 # R2 = d / (a + d)
add R2, X  # R2 += X
mov X, R2
# Third fraction
mov R1, a
add R1, d  # R1 = a + d
div R1, e  # R1 = (a + d) / e
mov R2, X
sub R2, R1 # R2 = R2 - R1 = X - (a + d) / e
mov X, R2
```

ج)

```
# First fraction
load b
add c
store temp # temp = b + c
load a
div temp # AC = a / (b + c)
store X
# Second fraction
load a
add d
```

```
store temp # temp = a + d
load d
div temp # AC = d / (a + d)
add X
store X
# Third fraction
load a
add d
div e # AC = (a + d) / e
store temp
load X
sub temp # AC = X => AC = X - (a + d) / e
store X
```

د) در ابتدا عبارت را به صورت postfix می‌نویسیم. برای این کار عبارت را به صورت خطی و بدون خط کسری می‌نویسیم:

$$a/(b+c)+d/(a+d)-(a+d)/e$$

سپس می‌توانیم از دانش ساختمان داده، به کمک یک stack عبارت postfix را بنویسیم.

$$abc+/dad+/+ad+e/-$$

در نهایت عبارت را از چپ به راست می‌خوانیم. هر جا که متغیر دیدیم آنرا به در استک push می‌کنیم. وقتی که به عملگرد می‌رسیم آنرا اجرا می‌کنیم و در نهایت جواب نهایی را pop می‌کنیم.

```
push a
push b
push c
add
div
push d
push a
push d
add
div
add
push a
push d
add
push e
div
sub
```

سوال ۲

mult: از آنجا که این دستور مقادیر هر دو رجیستر $\$s0$ ، $\$s1$ را به عنوان ورودی می‌گیرد، از نوع Register Addressing است. دقت کنید که جواب *mult* در hi و lo می‌رود که آنها Implicit Addressing هستند.

mflo: در انجا $\$t4$ به صورت Register Addressing هستند ولی مبدأ جایی که عدد می‌آید از رجیستر lo است که Implicit Addressing است.

lw: مشخص است که $\$s2$ Register Addressing است. ولی از انجا که در قسمت دوم دستور یک offset را نیز علاوه بر یک رجیستر دریافت می‌کند که همان base آدرس ما است، این operand از نوع Base Addressing است.

addi: $\$s3$ و $\$t6$ از نوع Register Addressing هستند. از آنجا که یک عدد ثابت داریم در این قسمت از نوع Immediate Addressing است.

jr: از این دستور Register Addressing است چرا که آدرسی که می‌خواهیم به آن برویم در $\$ra$ است.
jal: دقت کنید که عملا این دستور psuedo memory direct است. چرا که ما نمی‌توانیم از هر جای برنامه به هر جا بپریم به خاطر محدودیت‌های MIPS.

سوال ۳

a) $R1 \times 100 = 991 \times 100 = 991000$

b) $R1 + R2 = 991 + 713 = 1704$

c) $R1 - M[FGH] = R1 - EDE = 991 - 171 = 820$

d) $R1 \times M[100] = 991 \times 200 = 198200$

e) $R1 - M[M[102]] = R1 - M[ABC] = R1 - CHA = 991 - 139 = 852$

f) $R1 + M[100 + 2] = R1 + ABC = 991 + 991 = 1982$

g) $R1 - M[101] = R1 - FFH = 991 - 333 = 658$

h) $R1 \times M[101 - 1] = R1 \times 200 = 991 \times 200 = 198200$

دقت کنید در قسمت g بعد از اتمام عملیات به $R6$ یکی زیاد می‌شود.

سوال ۴

قسمت الف

در ابتدا تعداد کل opcodeها را حساب می‌کنیم. $22 + 20 + 10 + 5 = 57$ حال حداقل تعداد بیتی را می‌خواهیم که بشود با آن ۵۷ را نشان داد. این عدد برابر $2^6 = 64$ است. پس برای قسمت opcode نیاز به ۶ بیت داریم. دقیقا با همین منطق حداقل تعداد بیت برای آدرس‌دهی رجیسترها را بدست می‌آوریم: $\log_2 = 128 = 7$ پس به ۷ بیت نیاز داریم (حداقل).
حال برای هر نوع از دستورها بیت‌های مورد نیاز را حساب می‌کنیم؛ در دستور نوع A حداقل به $6 + 7 + 7 + 7 = 27$ بیت نیاز داریم. از آنجا که تعداد بیت‌ها باید مضرب ۸ باشد باید ۳۲ بیت به طول دستورات اختصاص دهیم. دقت کنید که این تعداد بیت برای همه‌ی نوع‌های دستوراتمان یکسان است. برای دستور B نیز به همین ترتیب است و $6 + 7 + 7 = 20$ بیت مفید استفاده می‌شود. در نوع دستور C مجبوریم که $6 + 7 = 13$ بیت را به opcode و register اختصاص دهیم. پس $32 - 13 = 19$ بیت برای آدرس مموری باقی می‌ماند. در دستورات نوع D نیز مانند بالا فقط ۶ بیت باید به opcode اختصاص می‌دهیم. در این حال $32 - 6 = 26$ بیت برای آدرس حافظه باقی می‌ماند.

قسمت ب

دقت کنید که همچنان برای هر رجیستر به ۷ بیت نیاز داریم. حال دقت کنید که از آنجا که در کل ۴ نوع دستور و mode داریم نیاز به دو بیت هم برای ذخیره سازی mode در هر حالت داریم. حال دقیقا مثل قبل اولین عدد توان دو بزرگتر مساوی n را پیدا می‌کنیم که در آن n تعداد opcodeها است. پس برای دستورات نوع اول $2 + 5 + 7 + 7 + 7 = 28$ بیت نیاز داریم که این موضوع نشان می‌دهد که باز هم برای ذخیره سازی دستورات به ۳۲ بیت نیاز داریم. حال برای دستور B داریم: $2 + 5 + 7 + 7 = 21$ حال دقت کنید که برای دستور C فقط ۱۰ نوع opcode داریم. پس برای ذخیره سازی این opcode دستور به ۴ بیت نیاز داریم. پس $32 - (2 + 4 + 7) = 19$ بیت برای آدرس حافظه باقی می‌ماند. برای دستورات نوع D از آنجا که فقط ۵ نوع opcode داریم فقط به ۳ بیت نیاز داریم. پس $32 - (2 + 3) = 27$ بیت برای آدرس مموری باقی می‌ماند.

سوال ۵

الف)

PC: در واقع برای *PC* ۱۲ بیت کافی است. چرا که $2^{12} = \frac{32768}{8}$ است و می‌توان کل مموری را آدرس دهی کرد. ولی از آنجا که طول هر کلمه ۱۶ است، باید برای *PC* ۱۶ بیت در نظر بگیریم.
IR: از آنجا که طول دستورات ما ۱۶ بیت هست، برای این رجیستر به ۱۶ بیت نیاز داریم.
SP: دقیقا مانند *PC* می‌توان نتیجه گرفت که به ۱۶ بیت نیاز داریم.

ب)

به نظر من این ماشین حساب بیشتر RISC است. چرا که همه‌ی دستورالعمل‌های ما طول ثابت ۱۶ بیت دارند و اینکه دستورالعمل‌های ما ساده هستند. اما دقت کنید که تمامی دستورالعمل‌های ما با رجیستر سر و کار ندارند! به عنوان مثال دستورات `add` ، `sub` و غیره با استک سر و کار دارند به جای رجیستر. برای همین از این لحاظ شبیه CISC هست. ولی در کل به نظر من این ماشین حساب RISC است تا CISC.

ج)

در اول از همه دقت کنید که از آنجا که push در استک از آخر به اول هست و ماشین‌حساب نیز big endian است، پس در صورتی که دو عدد ۱ و ۲ را به ترتیب در استک پوش کنیم، مثل این است که عدد 0x0201 را در استک پوش کرده‌ایم. چرا که درصورتی که بخواهیم از استک pop کنیم ابتدا به ۲ می‌رسیم و سپس ۱ که نشان می‌دهد که ۲ در بیت‌های پر ارزش است و ۱ کم ارزش تر است.

حال کد را تحلیل می‌کنیم. در دو خط اول 0x000A را در استک پوش می‌کنیم. سپس آدرس دستور جلوی loop label را در رجیستر R۴ قرار می‌دهیم. دقت کنید که اگر آدرس بیشتر از ۸ بیت باشد عدد اشتباهی در رجیستر می‌رود و برنامه خراب می‌شود. سپس در loop می‌رویم و 0x0001 و 0x0002 را در استک پوش می‌کنیم. سپس این دو عدد را با هم جمع میزنیم و آنرا در R1 قرار می‌دهیم.

در ادامه چندین self modifying code اتفاق می‌افتد. بدین صورت که در ابتدا عدد دستور اولین pushi 0 بعد از loop را در R2 می‌ریزیم. سپس مقدار R1 که جمع دو عدد داده شده در این دستور بود را در عدد همان دستور ذخیره می‌کنیم. در ادامه مقدار اولیه‌ی دستور pushi 0 را در pushi 1 ذخیره می‌کنیم.

سپس عدد یک را در استک پوش می‌کنیم و عبارت $10 - 1$ را حساب می‌کنیم و درصورتی که برابر ۰ بود حلقه را تمام می‌کنیم. در غیر این صورت عدد حاصل را در استک پوش می‌کنیم و حلقه را دوباره اجرا می‌کنیم. منظور این پاراگراف این بود که حلقه را ۱۰ بار اجرا می‌کنیم.

در نهایت سعی می‌کنیم رابطه‌ای برای هر کدام از متغیرها در بیاوریم. فرض می‌کنیم که عدد دستور pushi 1 در مرحله‌ی iام برابر a_i است و عدد دستور pushi 0 بعدی در مرحله‌ی iام برابر b_i است. پس داریم:

$$a_0 = 1$$
$$b_0 = 0$$
$$a_{i+1} = b_i$$
$$b_{i+1} = (b_i + a_i \times 256 + 2)\%256 = (b_i + 2)\%256$$

د)

```
# Initialize the answer as 1
pushi 1
pushi 0
pop R2
# Loop to calculate the factorial
FACTORIAL_LOOP:
bz R1, END # Check end of loop
# We have to calculate R2 * R1
# So we copy R1 to R15 and R14 to loop over it and sum it
```

```
# We also copy R2 to R13
# To copy the values we use stack
push R0
pop R13 # R13 = 0
push R2
pop R14 # R14 = R2
push R1
pop R15 # R15 = R1; Loop counter
MULT_LOOP:
bz R15, MULT_LOOP_DONE
# We have to do R13 = R14 + R14 ... + R14, R15 times
push R13
push R14
add
pop R13 # R13 = R13 + R14
# Now decrease the loop counter
# These two commands pushes -1 in stack
pushi -1
pushi -1
push R15
add # Now we have R15 - 1
pop R15 # R15--
j MULT_LOOP
MULT_LOOP_DONE:
# Mult loop done; We have answer in R13
push R13
pop R2
# Now decrease factorial the loop counter
# These two commands pushes -1 in stack
pushi -1
pushi -1
push R1
add # Now we have R1 - 1
pop R1 # R1--
j FACTORIAL_LOOP
END:
```