معماری کامپیوتر – دکتر عطار زاده

تکلیف سوم

(1

روتین های جداگانه ای برای بررسی بخش پذیری اعداد به 3 و 5 بنویسید.

برای تعریف تابع بخش پذیری بر3، ابتدا عدد 3 را به وسیله دستور addi در متغیر t0 می‌ریزیم. سپس باقی مانده a0 بر t0 را با استفاده از دستور rem در متغیر t0 می‌ریزیم.

سپس خروجی تابع را مساوی با 0 قرار می‌دهیم. در خط پنجم چک می‌کنیم که اگر باقی مانده غیرصفر بود یا به عبارتی ورودی تابع بر 3 بخش پذیر نبود، به لیبل after_if_3 برویم و دستور خط 6 اجرا نشود و در این صورت خروجی تابع 0 می‌ماند. اگر شرط خط پنجم صحیح نباشد، دستور خط ششم اجرا می‌شود و خروجی تابع برابر با 1 می‌شود.

```
1 three_div:
2   addi t0, zero, 3
3   rem t0, a0, t0 # t0 = inp % 3
4   addi a0, zero, 0 # a0 = 0
5   bne t0, zero, after_if_3 # checks if rem is 0, if it is, it assigns 1 to it and if not, it will jump to after_if_3 label
6   addi a0, zero , 1
7 after_if_3:
8   jr ra
```

برای بخش پذیری بر 5، نیز همین منطق را پیاده ساری می‌کنیم. با این تفاوت که ابتدا در متغیر t0 عدد 5 را می‌ریزیم.

```
# (a0) -> a0
# a0 = (a0 % 5) == 0
five_div:
  addi t0, zero, 5
  rem t0, a0, t0 # t0 = inp % 5
    addi a0, zero, 0 # a0 = 0
    bne t0, zero, after_if_5 # checks if rem is 0, if it is, it assigns 1 to it and if not, it will jump to after_if_5 label
    addi a0, zero , 1
after_if_5:
  jr ra
```

برنامه ای بنویسید که با به کارگیری روتین های فوق مسئله را حل کند و خروجی را در ثابتی که تعیین می کنید ذخیره کند.

حال برای حل مسئله تابع main را تعریف می‌کنیم.

متغیر s0 را به عنوان شمارنده لوپ و متغیر s2 را جمع عناصر واجد شرایط در نظر می‌گیریم. همچنین متغیر s2 برای ایجاد شرط برای بررسی اعداد کوچکتر از 20 است که در ادامه به آن می‌پردازیم

```
main:
    addi s0, zero, 0 #loop_counter
    addi s1, zero, 20 #n
    addi s2, zero, 0 #sum = 0
```

سپس حلقه را شروع می‌کنیم. خط ششم شرط حلقه را نشان می‌دهد و اگر شمارنده حلقه از 20 کوچک تر باشد، حلقه اجرا می‌شود و درغیر اینصورت به لیبل done می‌پرد.

در خط هشتم، ورودی تابع که می‌دانیم a0 است را برابر با عدد شمارنده قرار می‌دهد. سپس تابعی را که برای چک کدپردن بخش پذیری بر 3 نوشتیم را صدا می‌زند. می‌دانیم که خروجی تابع در متغیر a0 ریخته می‌شود. پس خط دهم چک می‌کند که در صورت برابر نبودن خروجی با 0 (که به معنای بخش پذیربودن عدد برا 3 است)، خط های 12 تا 17 اجرا نشوند و شمارنده به جمع اعداد وارد شرایط اضافه شود. اگر شرط خط دهم برقرار نباشد، با همین منطق، بخش پذیری بر 5 را چک می‌کنیم و در صورت بخش پذیر بودن عدد آن را به جمع اعداد وارد شرایط اضافه می‌کنیم. اگر شمارنده بر هیچ کدام از اعداد 3 و 5 بخش پذیر نبود، خطوط شماره 18 تا 20 اجزا نمی‌شوند و مستقیم به لیبل continue_loop می‌رویم تا دوباره حلقه را اجرا کنیم.

```
5     for:
6         beq s0, s1, done #loop condition s0==s1==20
7
8         addi a0, s0, 0 #a0 = s0
9         jal three_div #check division by 3. a0 = (s0 % 3)==0
10        bne a0, zero, should_add #if(s0 % 3 == 0) sum+= loop_counter
11
12        addi a0, s0, 0 #a0 = s0
13        jal five_div #check division by 5. a = (s0 % 5) == 0
14        bne a0, zero, should_add #if(s % 5 == 0) sum+=loop_counter
15
16        j continue_loop
17
18        should_add:
19        add s2, s2, s0 #actually add loop_counter to sum (s2+=s)
20
21        continue_loop:
22        addi s0, s0, 1 #increament loop_counter
23        j for
24    done:
25  j done
```

تست کردن کد با شبیه ساز:

محتویات حافظه قبل از شروع به اجرای تابع main:

Registers Memory

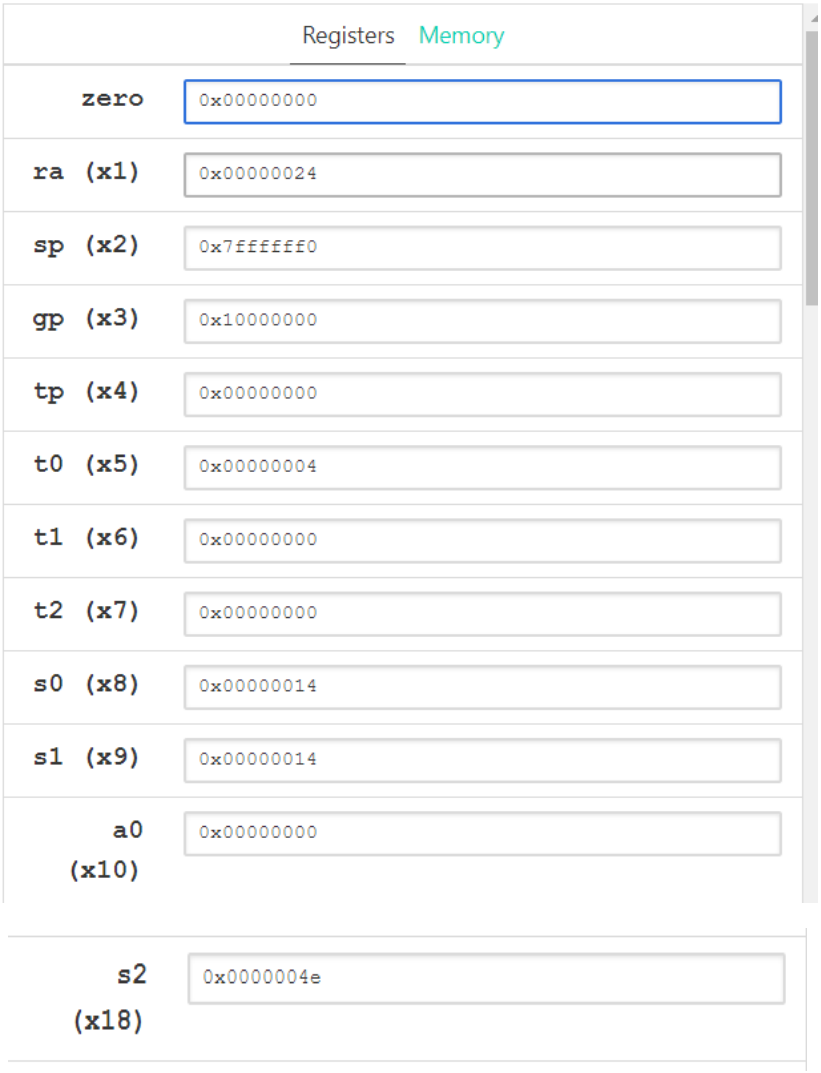
Address	+0	+1	+2	+3
0x00000018	63	1a	05	00
0x00000014	ef	00	80	02
0x00000010	13	05	04	00
0x0000000c	63	06	94	02
0x00000008	13	09	00	00
0x00000004	93	04	40	01
0x00000000	13	04	00	00

عکس زیر محتویات حافظه را بعد از چک کردن حلقه برای عدد0 تا 6 نشان می‌دهد. می‌دانیم که اعدادی که دارای شرایط خواسته شده سوال هستند و بین 1 تا 6 قرار دارند عبارتند از: 0 و 3 و 5 و6، که جمع این اعداد برابر با 14 می‌شود. خروجی کد ما به وسیله شبیه ساز به شکل زیر است:

Registers Memory

zero	0x00000000
ra (x1)	0x00000018
sp (x2)	0x7fffffff0
gp (x3)	0x10000000
tp (x4)	0x00000000
t0 (x5)	0x00000000
t1 (x6)	0x00000000
t2 (x7)	0x00000000
s0 (x8)	0x00000006
s1 (x9)	0x00000014
a0 (x10)	0x00000001
a1	0x00000000
a2 (x12)	0x00000000
a3 (x13)	0x00000000
a4 (x14)	0x00000000
a5 (x15)	0x00000000
a6 (x16)	0x00000000
a7 (x17)	0x00000000
s2 (x18)	0x0000000e
s3 (x19)	0x00000000
s4 (x20)	0x00000000

حال پس از اجرای کامل کد داریم:



همانطور که قابل انتظار بود، s1 و s0 بعد از اجرای حلقه باهم برابر و مساوی 20 شدند که نشان دهنده این است که حلقه به پایان رسیده است و مقدلر متغیر s2 برابر 78 است.

78 = 0+3+5+6+9+10+12+15+18

محتویات حافظه بعد از اجرای کامل کد:

Registers Memory

Address	+0	+1	+2	+3
0x00000018	63	1a	05	00
0x00000014	ef	00	80	02
0x00000010	13	05	04	00
0x0000000c	63	06	94	02
0x00000008	13	09	00	00
0x00000004	93	04	40	01
0x00000000	13	04	00	00
----------	--	--	--	--
----------	--	--	--	--

معماری کامپیوتر - دکتر عطارزاده

تمرین سری ۳

سوال ۲

الف

کد داده شده، دنباله فیبوناچی را تا عضو ۹ ام آن (یا به طور دقیق‌تر، t5 + 2) تولید می‌کند (با فرض این که صفر و ۱ نیز دو عضو اول آن هستند) و آن را در یک آرایه که آدرس خانه اول آن در t0 ذخیره شده است ذخیره می‌کند.

در هر مرحله از اجرای حلقه‌ی کد، t2 و t3 دو عضو آخر تولید شده را نگه‌داری می‌کنند. سپس عضو بعدی از جمع این دو مقدار به دست می‌آید و در t4 ذخیره می‌شود و این عدد ذخیره شده، در آرایه مورد نظر نیز ذخیره می‌شود. سپس سه رجیستر t2، t3، و t4 یک بار شیفت می‌خورند تا مقدارهای t2 و t3 بروز شوند. همین مراحل تکرار می‌شود تا همه t5 عضو دنباله فیبوناچی نظر ساخته شود.

ب

یک راه، پیاده سازی تابعی مثل fibonacci است که یک پارامتر n را گرفته و عضو nام این دنباله را برمی‌گرداند. در نهایت با استفاده از این تابع، در هر مرحله از اجرای حلقه، با کمک این تابع، عضو iام را حساب می‌کنیم و در آرایه ذخیره می‌کنیم.

این روش کمی کد را ساده‌تر می‌کند؛ اما باعث افزایش پیچیدگی زمانی تابع می‌شود؛ چون در هر مرحله از اجرای حلقه، برای محاسبه عضو iام دنباله، تابع از ابتدا شروع به محاسبه آن عضو می‌کند. در حالی که می‌توان از دو عضو قبلی محاسبه‌شده، بسیار سریعتر عضو بعدی را محاسبه کرد. (مخصوصا اگر تابع به صورت بازگشتی پیاده شود، پیچیدگی زمانی بسیار بالایی خواهد داشت).

سوال ۳

کد اسمبلی مربوط به این سوال، در فایل q03.s موجود است.

در ابتدا، با استفاده از تابع initialize، آرایه اولیه را مقدار دهی می‌کنیم. این تابع، ۶ عدد را در یک مکان مشخص از حافظه به صورت یک آرایه ذخیره می‌کند و آدرس شروع آرایه را برمی‌گرداند. سپس یک حلقه درست می‌کنیم که در آن t0 متغیر حلقه است. در هر مرحله از اجرای حلقه، آدرس شروع آرایه را با این متغیر جمع می‌کنیم تا آدرس عضو فعلی آرایه به دست بیاید. در نهایت، عضو فعلی آرایه را به تابع fibo داده و مقدار آن را در عضو فعلی آرایه ذخیره می‌کنیم. حلقه تا جایی ادامه پیدا می‌کند که ۶ عضو

آرایه را پیمایش کرده باشیم؛ یا به عبارتی دیگر، از آدرس شروع آرایه ۲۴ بایت جلوتر رفته باشیم (با توجه به این که هر عضو آرایه ۴ بایت است، ۶ عضو در مجموع ۲۴ بایت جا می‌گیرند).

همچنین در تابع `fibo`، ابتدا شرط `n <= 1` را چک می‌کنیم (`n` همان تنها پارامتر تابع است) و در صورتی که شرط برقرار بود، همان `n` را برمی‌گردانیم. در غیر این صورت به قسمت `else` می‌پریم، مقدار `n - 1` و `n - 2` را حساب می‌کنیم. سپس به نوبت تابع را با این دو مقدار صدا زده و حاصل جمع این دو مقدار را بر می‌گردانیم.

همچنین دقت داریم که برخی رجیسترها caller-saved و برخی callee-saved هستند. مثال در تابع `main` در هنگام صدا زدن تابع `fibo`، همه رجیسترهای caller-saved را در استک ذخیره می‌کنیم و پس از صدا زدن تابع، آن‌ها را از استک برمی‌گردانیم.

همچنین در مورد تابع بازگشتی `fibo`، به دلیل بازگشتی بودن تابع باید هر دو نوع رجیسترهای caller-saved و callee-saved را در استک ذخیره کنیم تا تابع به درستی کار کند.

اجرای کد

نتیجه اجرای تابع `initialize` در شبیه‌ساز venus در زیر آمده است:

Address	+0	+1	+2	+3
0x10008030	00	00	00	00
0x1000802c	00	00	00	00
0x10008028	00	00	00	00
0x10008024	00	00	00	00
0x10008020	00	00	00	00
0x1000801c	02	00	00	00
0x10008018	01	00	00	00
0x10008014	04	00	00	00
0x10008010	00	00	00	00
0x1000800c	03	00	00	00
0x10008008	05	00	00	00
0x10008004	00	00	00	00
0x10008000	00	00	00	00

Jump to -- choose -- Up Down

Display Settings Unsigned

شکل ۱: محتویات حافظه بعد از اجرای تابع initialize

همانطور که مشخص است، با شروع از خانه‌ی 0x10008008، خانه‌های آرایه به درستی پر شده‌اند.

نتیجه اجرای کد به طور کامل نیز در زیر آمده است:

Address	+0	+1	+2	+3
0x10008030	00	00	00	00
0x1000802c	00	00	00	00
0x10008028	00	00	00	00
0x10008024	00	00	00	00
0x10008020	00	00	00	00
0x1000801c	01	00	00	00
0x10008018	01	00	00	00
0x10008014	03	00	00	00
0x10008010	00	00	00	00
0x1000800c	02	00	00	00
0x10008008	05	00	00	00
0x10008004	00	00	00	00
0x10008000	00	00	00	00

Jump to -- choose -- Up Down

Display Settings Unsigned

شکل ۲: محتویات حافظه بعد از اجرای تابع main

در بالا، مقدار بازگشتی `fibo` در همان مکان آرایه در حافظه جایگزین شده است.