CA #4 report

حمیدرضا خدادادی 810197499
محمدعلی زارع 810197626

آپکد ها استاندارد هستند .

مسیرداده :

ساختار مسیر داده به طور کلی همانند درس است .
مسیر داده برای هر استیج (به جز استیج WB که یک ماکس در مسیرداده است .) و همچنین به ازای هر رجیستر میان آن استیج ها یک ماژول دارد . هر استیج سیگنال های مورد نیاز استیج های بعد و اطلاعات خروجی خود را به بیرون می فرستد و در مسیر داده این اطلاعات و سیگنال ها به رجیستر میان آن استیج و استیج بعد میرود .
برخی خروجی ها هم مستقیم به استیج های قبل می روند . (مانند آدرس های PC که از ID به IF میروند .) هازارد یونیت و فورواردینگ یونیت نیز بیرون استیج ها در مسیرداده قرار داده شده اند .
بعد از خروجی های "رجیستر فایل" یک مقایسه کننده قرار داده شده که برای تصمیم های برنچ (beq , bne) استفاده می شود .

کنترلر :

کنترلر در این مدار تفاوتی با کنترلر سینگل سایکل ندارد .
همان سیگنال ها برای هر دستور تولید میشوند . کنترلر در استیج ID قرار داده شده و سیگنال های مورد نیاز استیج های بعدی وارد رجیستر ID / EX می شوند . سیگنال pc_src هم مستقیم به استیج IF داده میشود .
کنترلر با توجه به آپکود (6 بیت اول دستور) سیگنال های مورد نیاز را ارسال می کند . کنترلر یک زیرماژول هم برای سیگنال مورد نیاز ALU دارد که در زیرماژول ها توضیح داده شده است .
6 بیت آخر دستور هم به عنوان func وارد کنترلر می شود تا در صورتی که دستور از نوع R-Type بود در زیرماژول Alu_controller استفاده شود.

CA #4 report

زیرماژول ها :

اکثر ماژول هایی که در پروژه های قبل وجود داشتند بدون تغییر در اینجا استفاده شده اند . تنها تفاوت در ماژول **regFile** است که عملیات نوشتن در این مدار با negedge کلاک ، انجام می شود . دلیل آن هم شبیه سازی نوشته شدن ، در نیمه دوم کلاک است .

استیج ها :

هر استیج با یک ماژول شبیه سازی شده است و ساختار آن ها مانند درس است .

استیج IF دارای ماژول های مرتبط به PC و اینستراکشن مموری است .

استیج ID شامل کنترلر و رجیسترفایل و ماژول های مربوط به آن ها و یک adder برای محاسبه آدرس و یک مقایسه کننده برای خروجی های رجیسترفایل (سیگنال فلاش با استفاده از نتیجه این مقایسه صادر می شود .) است .

استیج EX شامل ALU و ماکس های مربوط به آن است .

استیج MEM نیز Data Memory را در خود جای داده است .

رجیسترها : رجیستر های میان استیج ها ساختار مشابهی دارند که با لبه مثبت کلاک اطلاعات ورودی را روی خروجی می گذارند . هر کدام از رجیستر ها با توجه به موقعیت خود ، ورودی و خروجی های متفاوتی دارند . (سیگنال ها و اطلاعات) تنها رجیستری که کمی با بقیه ی رجیستر ها متفاوت است IF/ID است که سیگنال write و flush دارد که دلیل آن هم برای stall و حباب وارد کردن است . این دو سیگنال write به وسیله ی هازارد یونیت و flush به وسیله ی رجیستر ID صادر می شود .

فروواردینگ یونیت :

برای فوروارد کردن اطلاعات هنگام وابستگی داده ای استفاده میشود .

شرط های آن مانند شرط های گفته شده در درس است . که این یونیت سیگنال های انتخاب ماکس های پشت ورودی ALU را تعیین می کند .

CA #4 report

هازارد یونیت :

برای تشخیص هازارد کنترلی است و سیگنال های pc_write و IF_ID_write و یک سیگنال برای صفر کردن سیگنال های کنترلی را صادر میکند . که در هازارد ماکس استفاده شده است .
هازارد هایی که در صورت پروژه گفته شده نیز در این ماژول حل شده اند .

هازارد ماکس :

یک ماژول که کار ماکس سیگنال های کنترلی را انجام می دهد . از هازارد یونیت یک سیگنال دریافت می کند و اگر هازاردی اتفاق افتاده بود همه سیگنال های کنترلی را صفر میکند و به استیج بعد می دهد . این کار باعث می شود دستوری که جلو می رود بی اثر شود .

DateMemory :

از reg دو بعدی برای ذخیره دیتا ها استفاده شده (512 تا reg 32 بیتی) . دیتاها از فایل memroy.data خوانده میشود و در این ساختمان داده ریخته میشوند . ورودی ماژول آدرس 32 بیتی است که با تقسیم به 4 کردن آن ، ایندکس در آرایه به درست می آید و آن را در خروجی قرار می دهد . با کلاک خوردن و در صورت فعال بودن سیگنال mem_write ، ورودی write_data در آدرس ریخته می شود .
در صورت وجود سیگنال mem_read نیز داده ی موجود در آدرس بر روی خروجی read_data قرار میگیرد .

InstructionMemory :

از reg دو بعدی برای ذخیره دیتا ها استفاده شده (512 تا reg 32 بیتی) . دستورات از فایل instruction.data خوانده می شود و در این ساختمان داده ریخته می شوند . ورودی ماژول آدرس 32 بیتی است که با تقسیم به 4 کردن آن ، ایندکس در آرایه به درست می آید و دستور متناظر با آن ایندکس را در خروجی قرار می دهد .

RegFile :

از reg دو بعدی (32 تا 32 بیتی) برای نگه داری استفاده شده است . indexing در این ماژول همان عدد ورودی است . با خوردن کلاک در صورت وجود سیگنال reg_write ، دیتای write_data در رجیستر شماره write_reg_address ریخته میشود.

CA #4 report

خروجی های read_data نیز همیشه محتویات آدرس های read_reg 1 / 2 را نشان می دهند.
مقدار R0 نیز همیشه صفر باقی می ماند.

PC :

خروجی آن نشان دهنده آدرس دستور بعدی است. در صورت وجود سیگنال rst مقدار آن صفر میشود و با هر بار اجرای دستور و عملی نیز مقدار next_pc را از mux قبل خود دریافت و در خروجی خود قرار میدهد .

ALU :

عمل مورد نظر که با alu_op مشخص شده و از ALU_controller دریافت میکند را روی دو ورودی 32 بیتی خود اعمال میکند و در خروجی قرار می دهد .

ALU_Cntroller :

با دریافت 6 بیت func و دو بیت alu_case عمل موردنیاز ALU را مشخص می کند .

Shift2 :

ورودی خود را دو بیت به سمت چپ شیفت می دهد.

Sign_Extend :

ورودی 16 بیت می گیرد و با رعایت علامت آن را به 32 بیت تبدیل می کند .

MUX : چهار نوع mux استفاده شده است .

1. 2 ورودی 5 بیت برای انتخاب بین rt , rd خروجی از ID/EX
2. 2 ورودی 32 بیت برای ورودی دوم ALU و ماکس استیج WB
3. 3 ورودی 5 بیتی برای ورودی آدرس write_reg
4. 3 ورودی 32 بیتی برای ورودی pc و ورودی اول ALU و ورودی اول ماکس قبل از ALU

CA #4 report

برای تست برنامه :

می توانید نام فایل مموری را که در ماژول DataMem که در حال حاضر memory.data است و به صورت دیفالت محتوای برنامه ی اول را اجرا میکند به memory_Q2.data یا تست خودتان تغییر دهید و نتیجه ی برنامه را چک کنید .

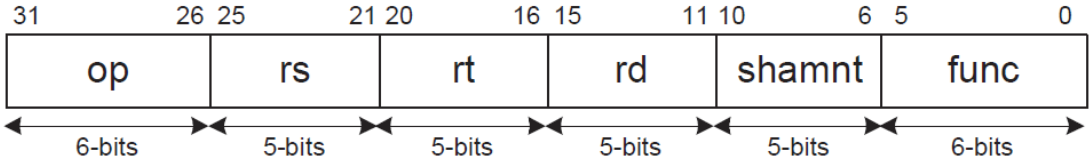
یا میتوانید نام فایل را در ماژول DataMem ثابت نگه دارید و نام اصلی برنامه ی دوم یا تست خودتان را به memory.data تغییر بدهید و برنامه را تست کرده و نتیجه را مشاهده فرمایید .

Arithmetic / Logical Instructions : add, sub, and, or, slt, addi, andi

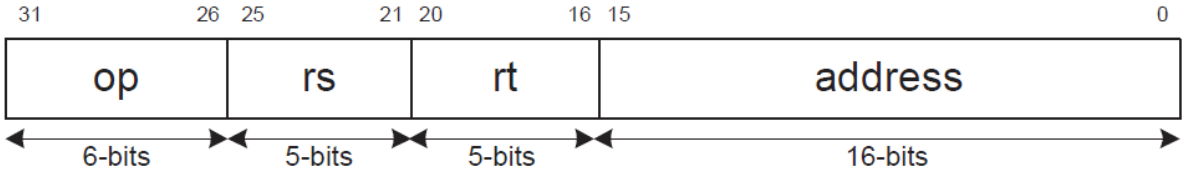
Memory Reference Instruction : lw, sw

Control Flow Instructions : j, beq, bne

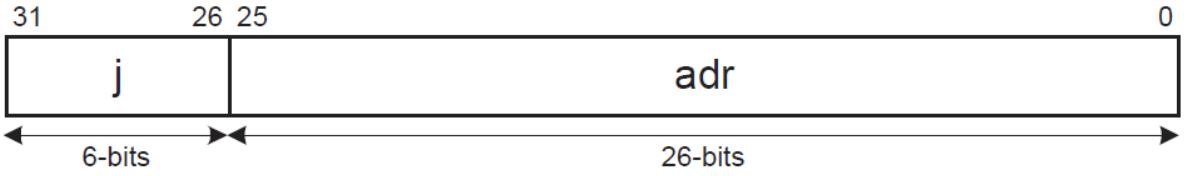
R type format : add-sub-and-or-slt



I type format : lw-sw-beq-bne-addi-andi



J type format : j



CA #4 report

برنامه ی اول : دستورات در اینستراکشن مموری و آرایه ی اعداد در مموری

110 + 5 + 0 + (-10) + 1000 + (2-) + 53 + 41 + (-1000) + 32 = 229

instructions.data

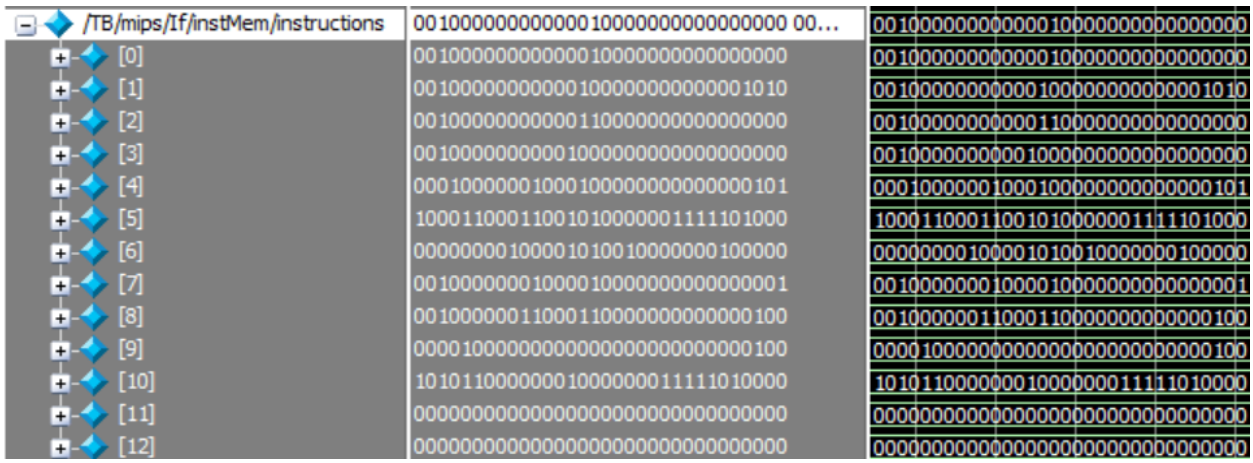
```
001000_00000_00001_0000000000000000     // addi R1, R0, 0 #loop_counter
001000_00000_00010_0000000000001010     // addi R2, R0, 10 #loop_end
001000_00000_00011_0000000000000000     // addi R3, R0, 0 #address_counter
001000_00000_00100_0000000000000000     // addi R4, R0, 0 #sum
000100_00001_00010_0000000000000101     // LOOP : beq R1, R2, END_LOOP
100011_00011_00101_0000001111101000     //        lw R5, 1000(R3)
000000_00100_00101_00100_00000_100000   //        add R4, R4, R5
001000_00001_00001_0000000000000001     //        addi R1, R1, 1
001000_00011_00011_0000000000000100     //        addi R3, R3, 4
000010_00000000000000000000000100       //        j LOOP
101011_00000_00100_0000011111010000     // END_LOOP : sw R4, 2000(R0)
00000000000000000000000000000000
```

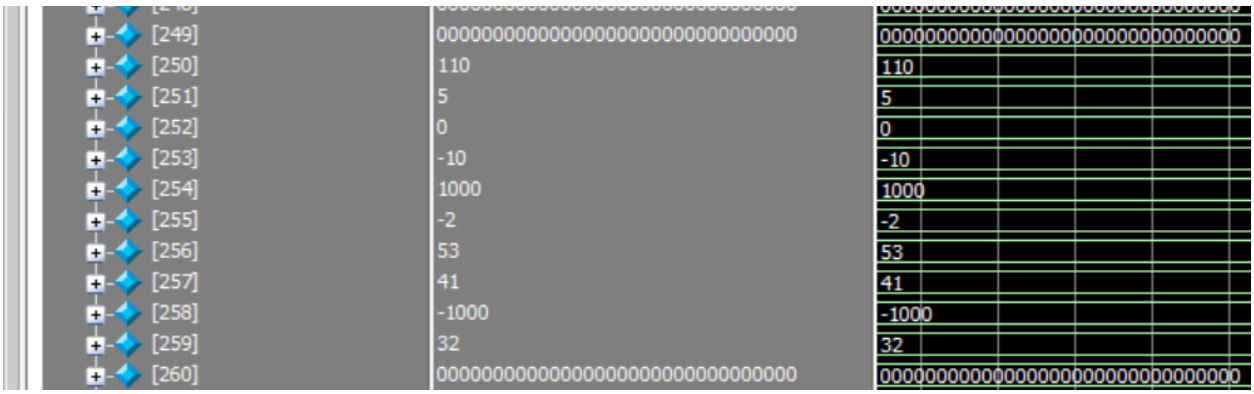
memory.data

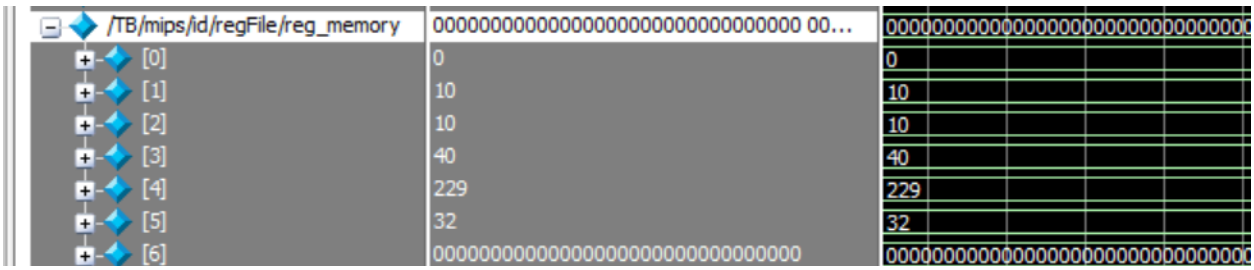
```
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000001101110    //110
00000000000000000000000000000101    //5
00000000000000000000000000000000    //0
11111111111111111111111111110110    //-10
00000000000000000000001111101000    //1000
11111111111111111111111111111110    //-2
00000000000000000000000000110101    //53
00000000000000000000000000101001    //41
11111111111111111111110000011000    //-1000
00000000000000000000000000100000    //32
00000000000000000000000000000000
```

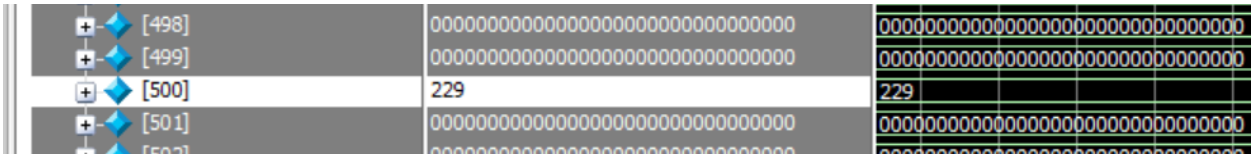
CA #4 report

در عکس اول محتوای اینستراکشن مموری برای دستورات ، و در عکس دوم محتوای مموری برای آرایه 10 تایی اعداد با شروع از خانه ی 1000 حافظه و در عکس سوم محتوای رجیستر های استفاده شده در دستورات در پایان اجرای برنامه و عکس چهارم محتوای خانه ی 2000 حافظه بعد از اجرای برنامه و خروجی نهایی برنامه را مشاهده می فرمایید :









CA #4 report

برنامه ی دوم : دستورات در اینستراکشن مموری و آرایه ی اعداد در مموری

MAX = 110 and index of MAX = 17

instructions_Q2.data × | memory_Q2.data

ca-project-4-2 > instructions_Q2.data

```
001000_00000_00001_0000000000000000     // addi R1, R0, 0 #loop_counter
001000_00000_00010_0000000000010100     // addi R2, R0, 20 #loop_end
001000_00000_00011_0000000000000000     // addi R3, R0, 0 #address_counter
001000_00000_00100_0000000000000000     // addi R4, R0, 0 #max
001000_00000_00101_0000000000000000     // addi R5, R0, 0 #max_index
000100_00001_00010_0000000000001001     // Loop : beq R1, R2, END_LOOP
100011_00011_00110_0000001111101000     //        lw R6, 1000(R3) #data
000000_00100_00110_00111_00000_101010   //        slt R7, R4, R6 #max < current
000100_00111_00000_0000000000000010     //        bqe R7, R0, END_DO_UPDATE
000000_00000_00110_00100_00000_100000   // DO_UPDATE : add R4, R0, R6
000000_00000_00001_00101_00000_100000   //             add R5, R0, R1
00000000000000000000000000000000        // END_DO_UPDATE
001000_00001_00001_0000000000000001     //        addi R1, R1, 1
001000_00011_00011_0000000000000100     //        addi R3, R3, 4
000010_00000000000000000000000101       //        j LOOP
101011_00000_00100_0000011111010000     // END_LOOP : sw R4, 2000(R0)
101011_00000_00101_0000011111010100     //            sw R5, 2004(R0)
00000000000000000000000000000000
```

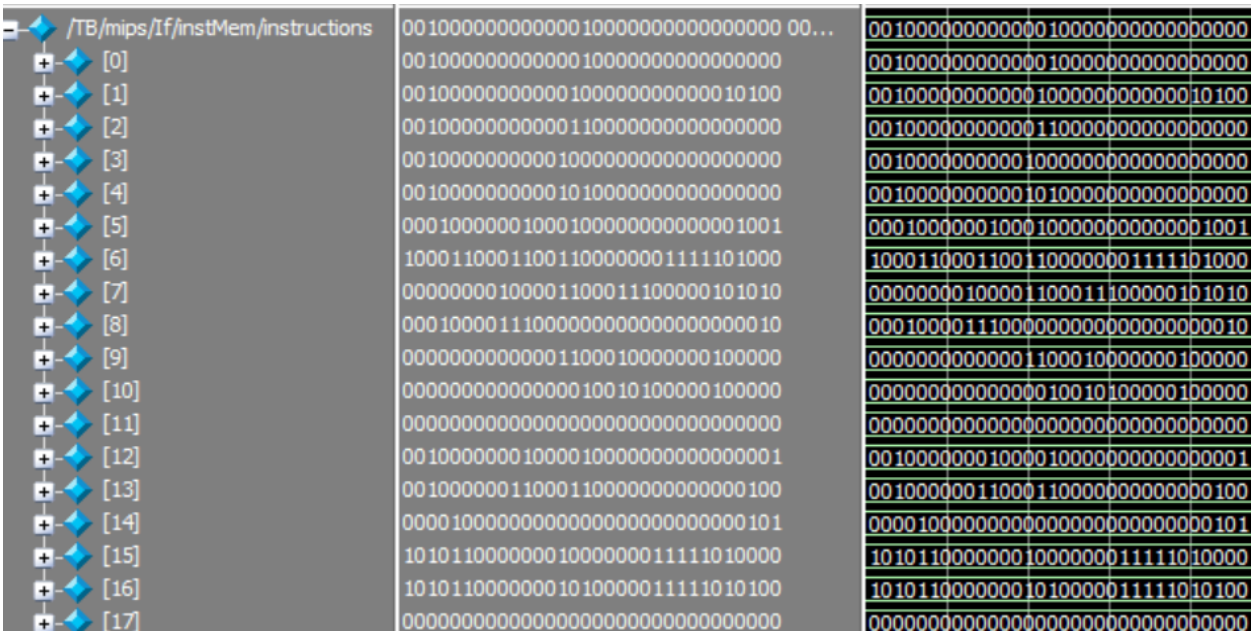
instructions_Q2.data | memory_Q2.data ×

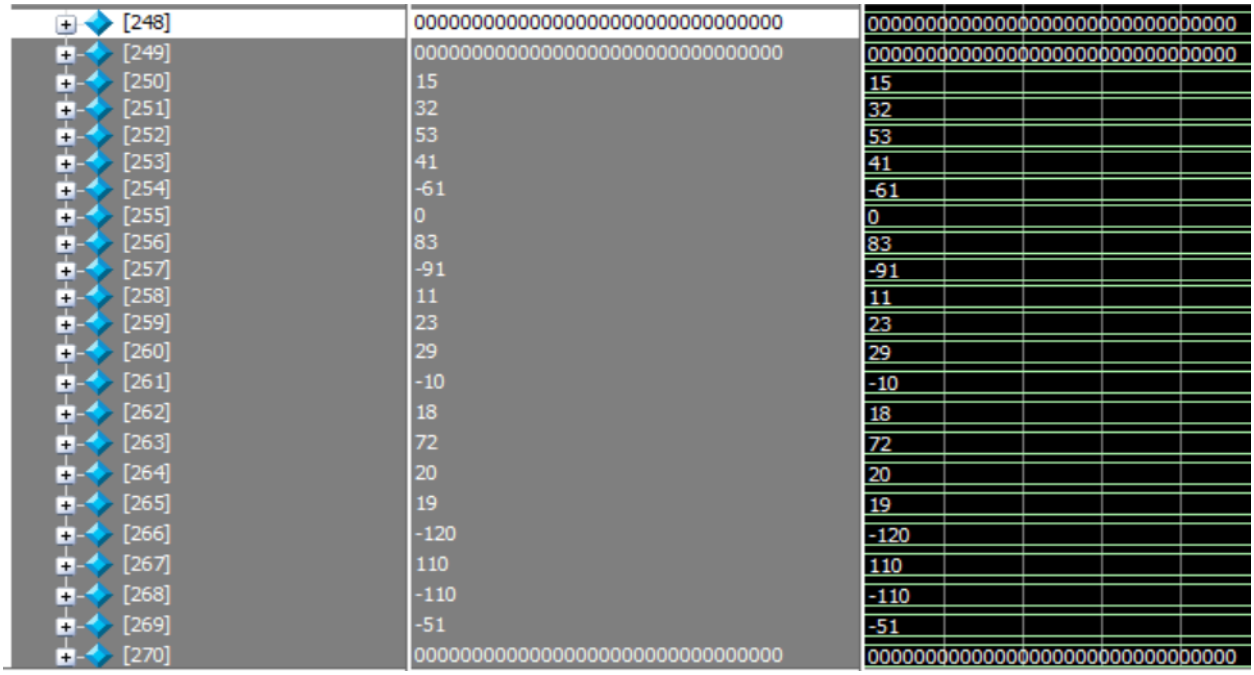
ca-project-4-2 > memory_Q2.data

```
250    00000000000000000000000000000000
251    00000000000000000000000000001111    //15
252    00000000000000000000000000100000    //32
253    00000000000000000000000000110101    //53
254    00000000000000000000000000101001    //41
255    11111111111111111111111111000011    //-61
256    00000000000000000000000000000000    //0
257    00000000000000000000000001010011    //83
258    11111111111111111111111110100101    //-90
259    00000000000000000000000000001011    //11
260    00000000000000000000000000010111    //23
261    00000000000000000000000000011101    //29
262    11111111111111111111111111110110    //-10
263    00000000000000000000000000010010    //18
264    00000000000000000000000001001000    //72
265    00000000000000000000000000010100    //20
266    00000000000000000000000000010011    //19
267    11111111111111111111111110001000    //-120
268    00000000000000000000000001101110    //110
269    11111111111111111111111110010010    //-11
270    11111111111111111111111111001101    //-51
271    00000000000000000000000000000000
```

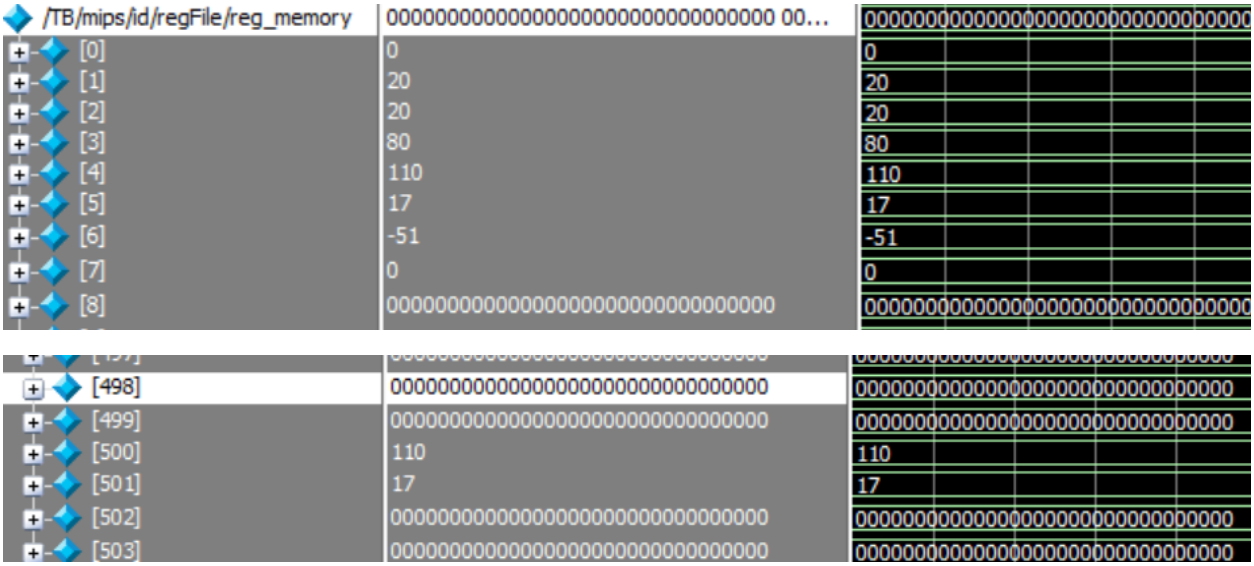
CA #4 report

در عکس اول محتوای اینستراکشن مموری برای دستورات ، و در عکس دوم محتوای مموری برای آرایه 20 تایی اعداد با شروع از خانه ی 1000 حافظه و در عکس سوم محتوای رجیستر های استفاده شده در دستورات در پایان اجرای برنامه و عکس چهارم محتوای خانه ی 2000 و 2004 حافظه بعد از اجرای برنامه و خروجی نهایی برنامه را مشاهده می فرمایید :





CA #4 report


/TB/mips/id/regFile/reg_memory
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[498]
[499]
[500]
[501]
[502]
[503]