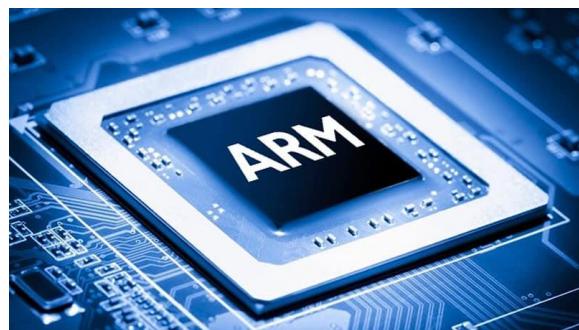


به نام خدا



دانشگاه تهران  
پردیس دانشکده‌های فنی  
دانشکده برق و کامپیوتر



آزمایشگاه معماری کامپیوتر

گزارش دستور کار شماره 4

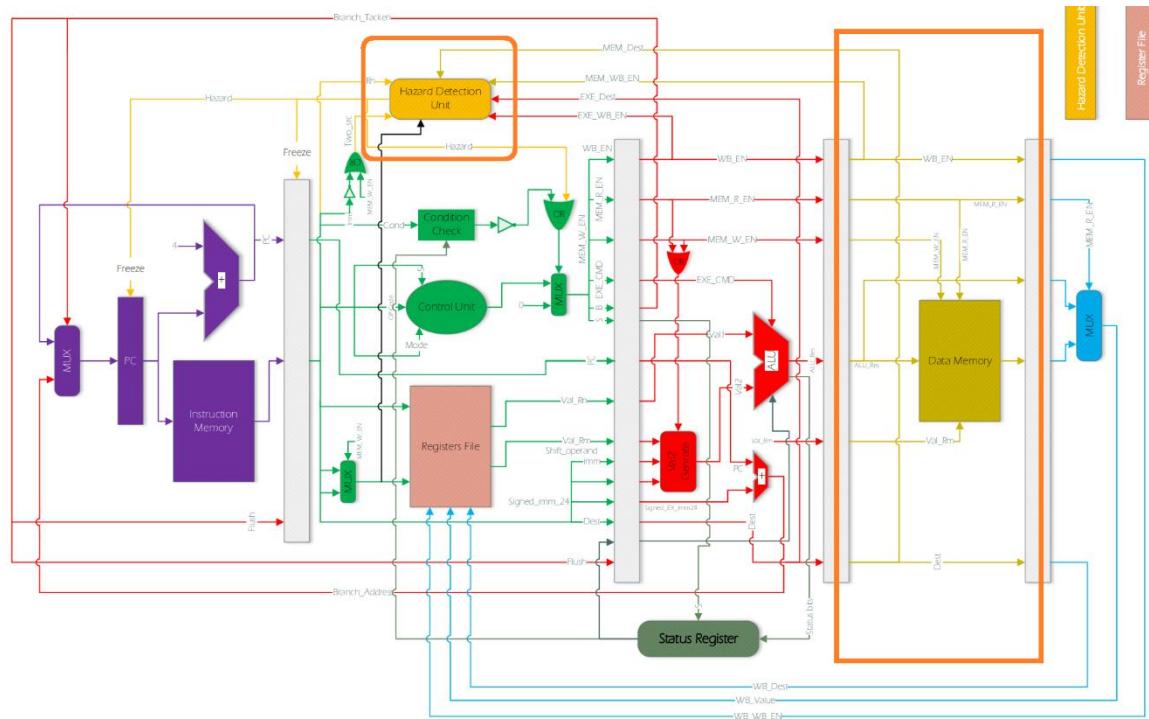
علی پادیاو  
810199388

محمد صالح عرفاتی  
810197543

فروردین 1402

## مقدمه

در آزمایشات قبلی مراحل IF و ID و EXE و WB از معماری ARM9 پیاده سازی شد. حال در این آزمایش به پیاده سازی مرحله MEM و مازول Hazard پرداختیم. بلوک دیاگرام این مرحله به صورت زیر می باشد:



شکل 1

پس از واکشی دستورات در مرحله قبل و به دست آوردن رجیستر های ورودی و فهمیدن نوع دستور خواسته شده، و اجرای عملیات ها محاسباتی در مرحله EXE نوبت آن رسیده است تا خواندن و نوشتن از مموری را انجام دهیم. پس از انجام محاسبات ALU، خواندن و نوشتن از مموری صورت میگیرد. اگر سیگنال MEM\_W\_EN یک باشد آنگاه مقدار رجیستر مورد نظر در آدرسی که ALU محاسبه کرد نوشته خواهد شد.

حال اگر سیگنال EM\_R\_EN یک باشد آنگاه از آدرسی که خروجی ALU فراهم کرده است از مموری خوانده شده و به مرحله WB فرستاده میشود.

## شرح ماذول ها:

## 1- Data Memory

کد:

```

DataMemory.v
1 module Data_Mem (
2     input clk, rst, MEM_W_EN, MEM_R_EN,
3     input[31:0] address, data,
4     output[31:0] out
5 );
6
7 reg[31:0] memory[0:63];
8
9 wire[7:0] i = (address - 1024) >> 2;
10
11 assign out = MEM_R_EN ? memory[i] : 32'b0;
12
13 always @(posedge clk)
14 begin
15     if (MEM_W_EN)
16         memory[i] <= data;
17 end
18
19 endmodule
20

```

این ماذول وظیفه مدیریت حافظه را در اختیار دارد. قابل ذکر است که آدرس های این ماذول برای خواندن و نوشتمن از آدرس 1024 شروع میشود(زیرا در معماری arm دو ماذول PC و Data Memory در یک ماذول حافظه قرار دارند و 1024 آدرس اول برای PC می باشد). به همین آدرس که به عنوان ورودی وارد این ماذول میشود، ابتدا 1024 از آن کم شده و سپس 2 واحد شیفت به راست را دارد تا همواره دیتایی که میخواهیم از مموری به خوانیم عددی با معنا باشد(مثلا از آدرس 1025 آدرسی با معنا نیست و به همین دلیل تقسیم بر 4 میشود تا همان آدرسی که 1024 خوانده میشود برای آن نیز خوانده شود).

اگر سیگنال MEM\_W\_EN یک باشد آنگاه مقدار رجیستر مورد نظر در آدرسی که ALU محاسبه کرد نوشته خواهد شد.

حال اگر سیگنال EM\_R\_EN یک باشد آنگاه از آدرسی که خروجی ALU فراهم کرده است از مموری خوانده شده و به مرحله WB فرستاده میشود. اگر این سیگنال یک نباشد خروجی 32 بیت صفر به مرحله WB می رود.

قابل ذکر است که این ماذول دارای سیگنال های CLK و RST نیز می باشد زیرا برای نوشتمن در مموری نیاز دارد که با سیگنال کلک این کار انجام گیرد.

## 2- MEM\_Stage:

همانطور که مشاهده میشود این مازول تنها شامل مازول Memory می باشد و وظیفه آن فراهم کردن دیتای خوانده شده از مموری و نوشتن دیتای مورد نظر می باشد و تنها در آن مازول DataMemroy تعریف شده است.

کد:

```
≡ MEM_Stage.v
 1  module MEM_Stage (
 2    input clk, rst, MEM_W_EN, MEM_R_EN,
 3    input[31:0] ALU_res, ST_val,
 4    output[31:0] mem_out
 5  );
 6
 7  Data_Mem data_mem(
 8    .clk(clk),
 9    .rst(rst),
10    .MEM_W_EN(MEM_W_EN),
11    .MEM_R_EN(MEM_R_EN),
12    .address(ALU_res),
13    .data(ST_val),
14    .out(mem_out)
15  );
16
17
18 endmodule
```

## 3- MEM\_Reg:

کد:

```
≡ MEM_Reg.v
 1  module MEM_Reg (
 2    input clk, rst, WB_EN_in, MEM_R_EN_in,
 3    input[31:0] ALU_res_in, mem_in,
 4    input[3:0] Dest_in,
 5
 6    output reg WB_EN_out, MEM_R_EN_out,
 7    output reg[31:0] ALU_res_out, mem_out,
 8    output reg[3:0] Dest_out
 9
10  );
11
12  always @(posedge clk, posedge rst) begin
13    if (rst) begin
14      Dest_out <= 0;
15      WB_EN_out <= 0;
16      MEM_R_EN_out <= 0;
17      ALU_res_out <= 0;
18      mem_out <= 0;
19    end else begin
20      WB_EN_out <= WB_EN_in;
21      MEM_R_EN_out <= MEM_R_EN_in;
22      Dest_out <= Dest_in;
23      ALU_res_out <= ALU_res_in;
24      mem_out <= mem_in;
25    end
26  end
27
28 endmodule
```

این مازول وظیفه ذخیره سازی سیگنالها و دیتای خوانده شده از مموری را برای سایر مراحل را دارد. سیگنال های زیر را حافظه خود ذخیره میکند تا در کلارک بعدی آن ها را برای سایر مازول ها فراهم آورد.

WB\_EN: این سیگنال ورودی به RegisterFile است و با سک شدن آن دیتای خروجی mux مرحله WB نوشته میشود.

: این سگنال در مرحله WB مشخص میکند که دیتای خوانده شده از مموری و یا RegisterFile باید در ALU نوشته شود.

Dest: این سیگنال آدرس رجیستر موردنظر برای نوشتمن را مشخص میکند.

ALU\_Res: در بعضی دستورات مانند ADD باید خروجی ALU به مرحله WB برسد تا در RegisterFile نوشته شود.

#### **4- HAZARD Detection Unit:**

کد:

```
 1  module HazardDetector (input [3:0] src1,
 2  |           input [3:0] src2,
 3  |           input [3:0] Exe_Dest,
 4  |           input Exe_WB_EN,
 5  |           input [3:0] Mem_Dest,
 6  |           input Mem_WB_EN,
 7  |           input Two_src,
 8  |           input use_src1,
 9  |           output hazard_Detected
10 | );
11 |
12 assign hazard_Detected = (Exe_WB_EN && (use_src1 && src1 == Exe_Dest)) ||
13 |                               (Exe_WB_EN && (Two_src && src2 == Exe_Dest)) ||
14 |                               (Mem_WB_EN && (use_src1 && src1 == Mem_Dest)) ||
15 |                               (Mem_WB_EN && (Two_src && src2 == Mem_Dest));
16 |
17 endmodule
18
```

این مازول وظیفه شناسایی و تولید وقفه برای اجرای درست دستورات را دارا می باشد و در چهار  
حالت زیر وقفه در روند اجرای دستورات در پایپلاین رخ میدهد:

هنگامی که سیگنال WB\_EN در مرحله EXE فعال باشد و میخواهد در رجیستری بنویسد و آدرس رجیستر RN که در مرحله ID میخواهد خوانده شود برابر با آدرس رجیستری باشد که مرحله EXE میخواهد در دو کلک بعد آن را آپدیت کند (به عبارت دستوری که در مرحله EXE است مقدار رجیستری را میخواهد تغییر دهد که دستور مرحله ID میخواهد از آن استفاده کند).

- هنگامی که سیگنال WB\_EN در مرحله EXE فعال باشد و میخواهد در رجیستری بنویسد و آدرس رجیستر Rm که در مرحله ID میخواهد خوانده شود برابر با آدرس رجیستری باشد که مرحله EXE میخواهد در دو کلک بعد آن را آپدیت کند (به عبارت دستوری که در مرحله EXE است مقدار رجیستری را میخواهد تغییر دهد که دستور مرحله ID میخواهد از آن استفاده کند).
- هنگامی که سیگنال WB\_EN در مرحله مموری یک باشد و میخواهد در رجیستری بنویسد و آدرس رجیستر Rn که در مرحله ID میخواهد خوانده شود برابر با آدرس رجیستری باشد که مرحله MEM میخواهد در کلک بعد آن را آپدیت کند (به عبارت دستوری که در مرحله MEM است مقدار رجیستری را میخواهد تغییر دهد که دستور مرحله ID میخواهد از آن استفاده کند).
- هنگامی که سیگنال WB\_EN در مرحله مموری یک باشد و میخواهد در رجیستری بنویسد و آدرس رجیستر Rm که در مرحله ID میخواهد خوانده شود برابر با آدرس رجیستری باشد که مرحله MEM میخواهد در کلک بعد آن را آپدیت کند (به عبارت دستوری که در مرحله MEM است مقدار رجیستری را میخواهد تغییر دهد که دستور مرحله ID میخواهد از آن استفاده کند).

در هر چهار حالت بالا وقفه رخ داده تا مطمئن شویم دیتای درستی از RegisterFile خوانده شود. قابل ذکر است که اگر دستور پرش نیز باشد باید سیگنال freeze یک شود تا بدون دلیل دستورات بعد از پرش اجرا نگردند.

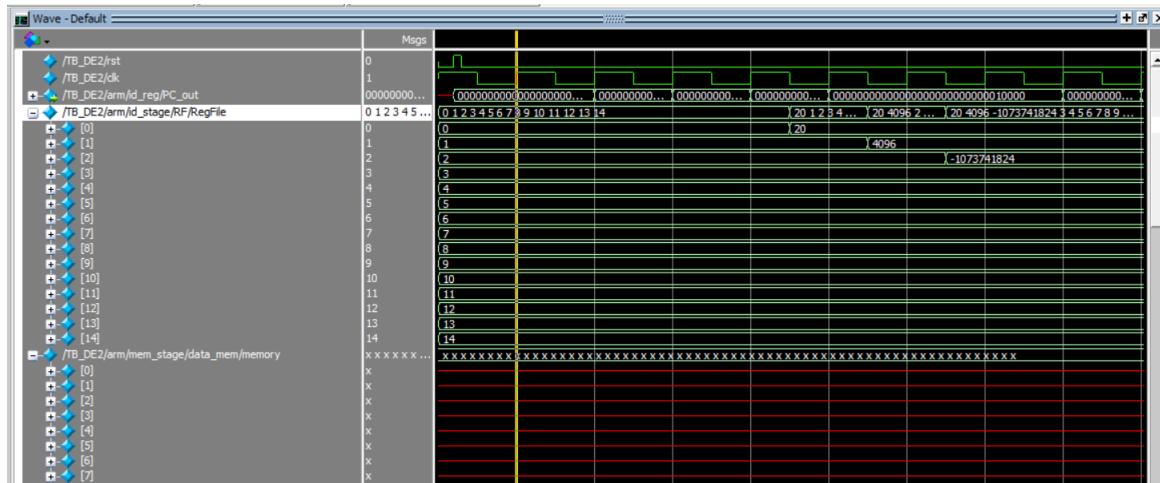
## دستورات برنامه:

طبق شرح آزمایش کل برنامه محک را داخل Ins\_Mem ذخیره کردیم تا دستورات یکی پس از دیگری اجرا گردند دقیق شود که وظیفه این برنامه مرتب سازی رجیسترها R1,R2,R3,R4 به صورت صعودی می باشد.

## SIMULATION

ابتدا پس از یک شدن سیگنال reset ماثول regFile در هر رجیستر آدرس آن نوشته می‌شود.

مطابق شکل زیر:



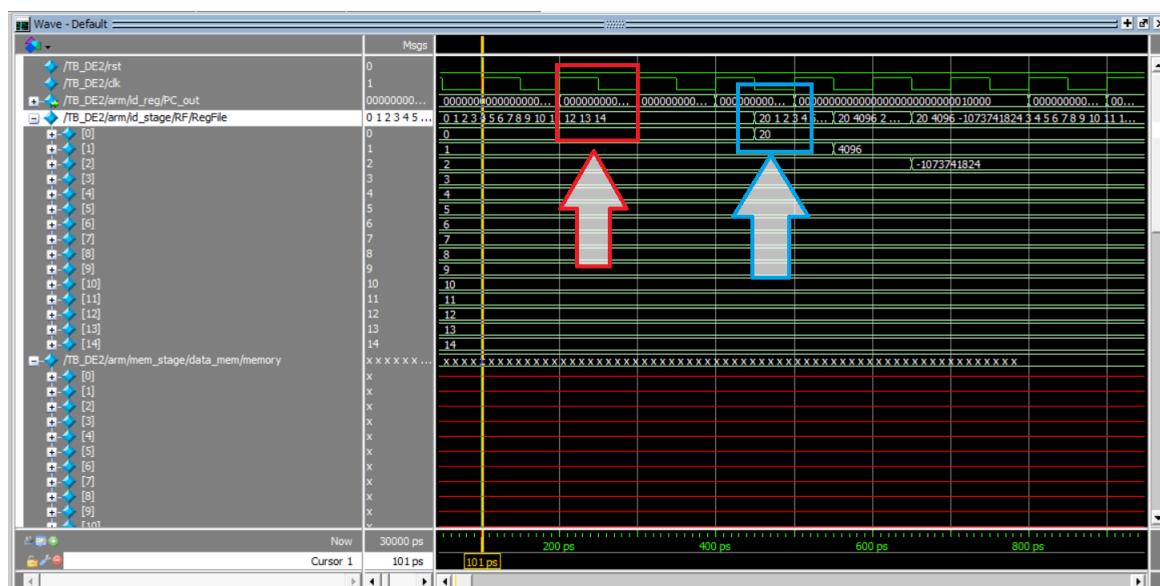
سپس دستور یک اجرا می‌گردد.

### دستور 1:

در دستور اول باید مقدار 20 در رجیستر 0 نوشته شود:

MOV R0, #20

32'b1110\_00\_1\_1101\_0\_0000\_0000\_000000010100

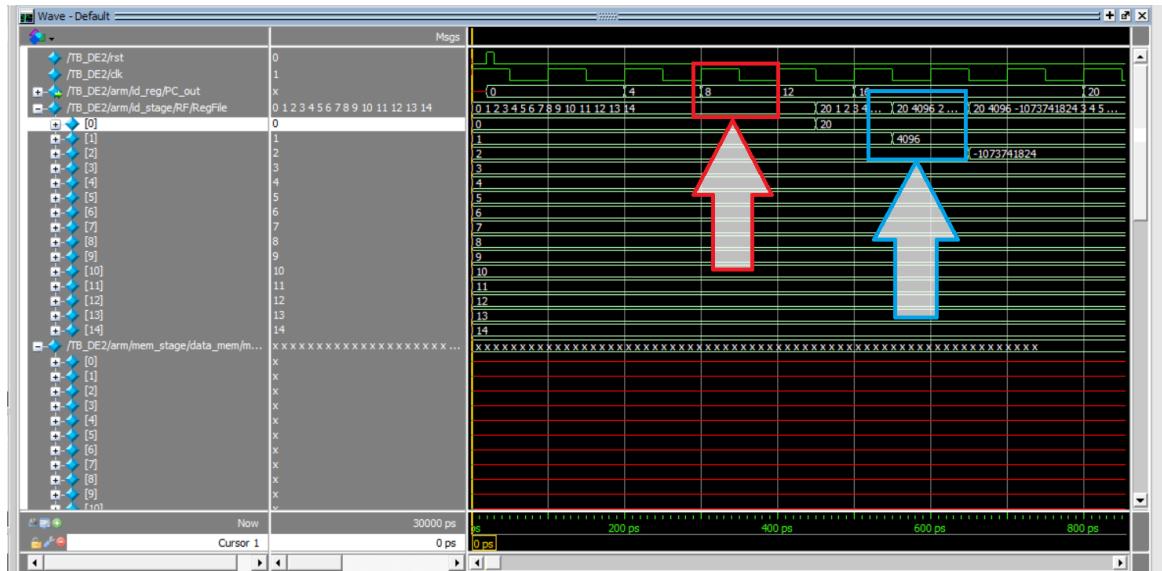


همانطور که فلش قرمز نشان میدهد دستور اول وارد ID\_Stage شده و سپس مقدار 20 در رجیستر شماره صفر نوشته می‌شود. همانطور که فلش آبی نشان میدهد.

## دستور 2:

MOV R1, #4096

32'b1110\_00\_1\_1101\_0\_0000\_0001\_101000000001

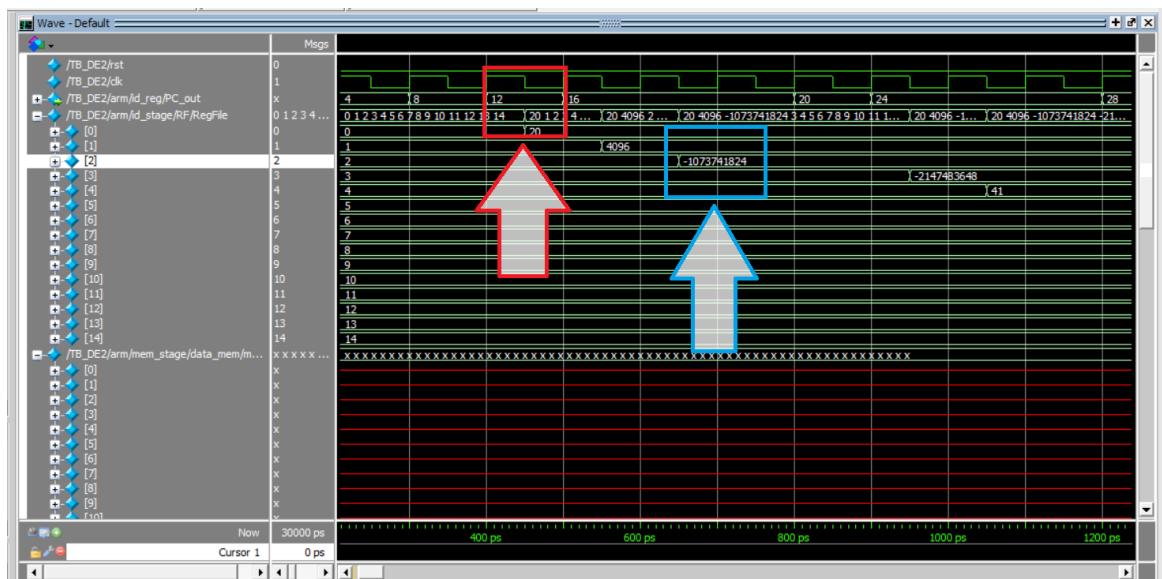


همانطور که در شکل بالا نیز مشاهده میکنید پس از اجرای دستور دوم مقدار 4096 در رجیستر شماره یک نوشته میشود.

## دستور 3:

MOV R2, # 0xC0000000 =&gt; R2 = -1073741824

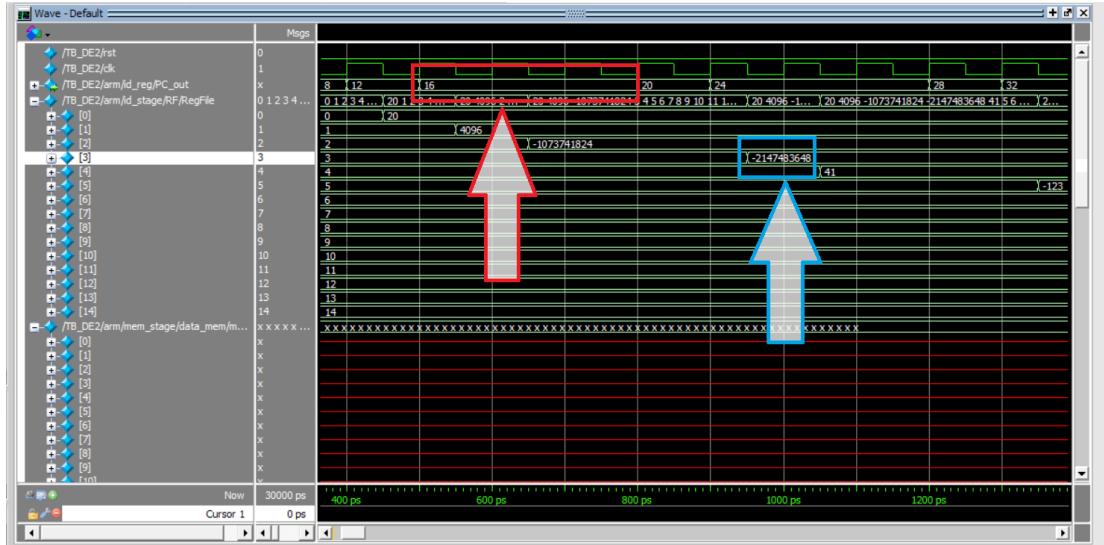
32'b1110\_00\_1\_1101\_0\_0000\_0010\_000100000011



همانطور که مشاهده میشود پس از اجرای دستور شماره 3، مقدار -1073741824 در رجیستر شماره 2 نوشته میشود.

## دستور 4:

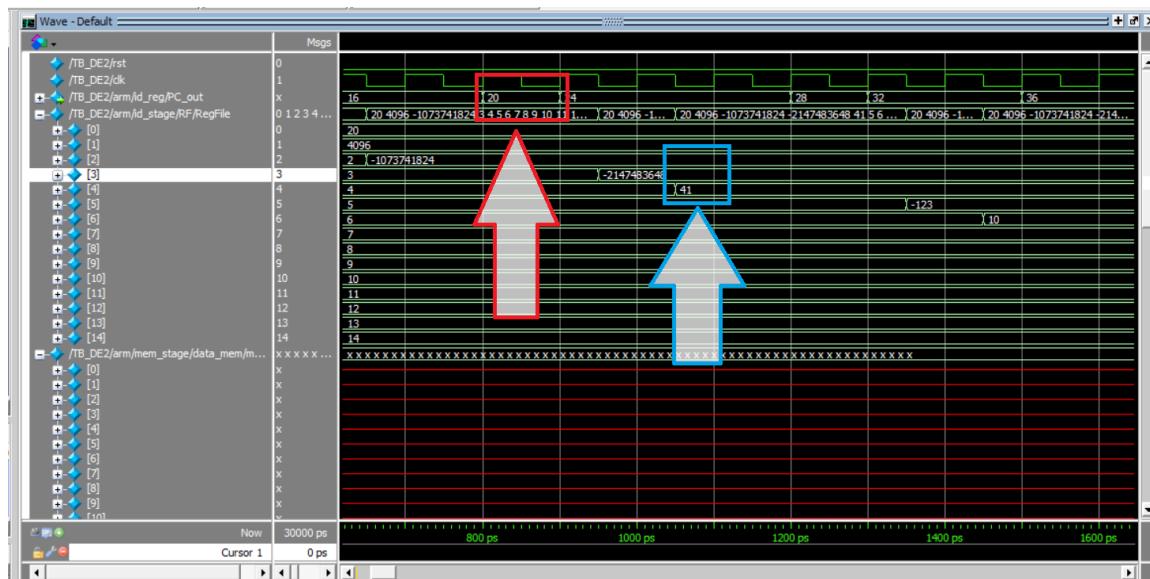
ADDS R3 ,R2,R2      =>      R3 = -2147483648  
 32'b1110\_00\_0\_0100\_1\_0010\_0011\_00000000000010



همانطور که مشاهده میشود مقدار -2147483648 در رجیستر شماره سه نوشته میشود. قابل ذکر است که چون اجرای این دستور وابسته به اجرای دستور قبل است(به دلیل استفاده از R2) پس در دو وقفه در سیستم مشاهده میکنیم.

## دستور 5:

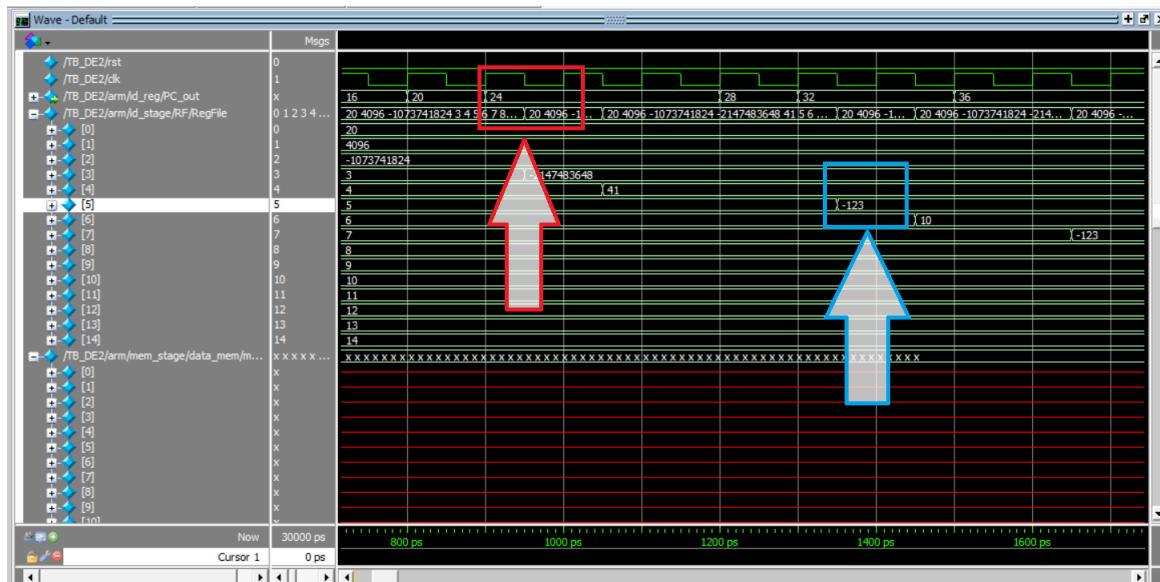
ADC R4 ,R0,R0      =>      R4 = 41  
 32'b1110\_00\_0\_0101\_0\_0000\_0100\_00000000000000



مقدار 41 به درستی در رجیستر شماره 4 می‌نشیند.

دستور 6:

SUB R5, R4, R4, LSL =&gt; R5 = -123

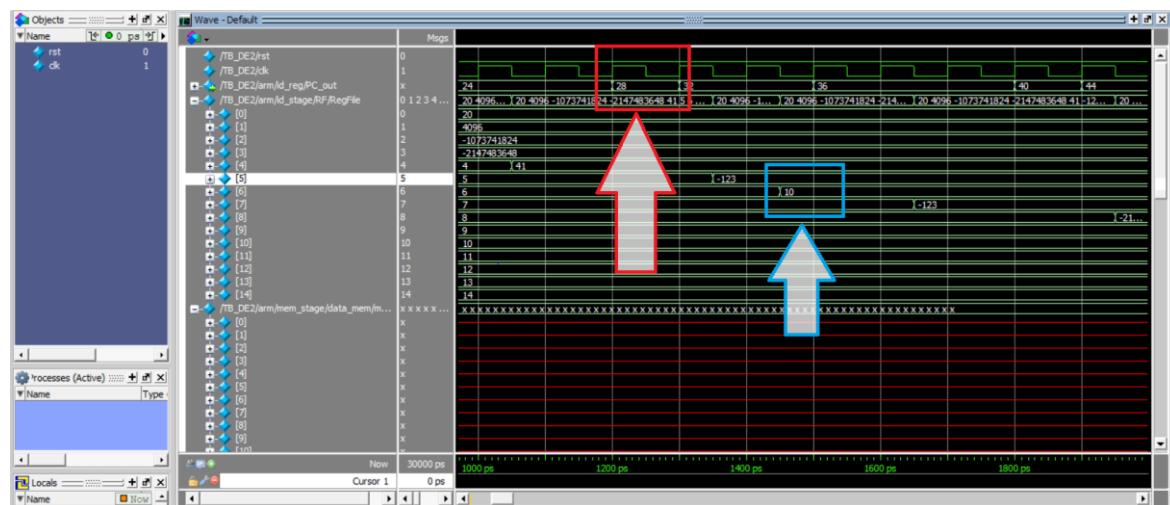


پس از اجرای دستور شماره 6 مقدار 123 در رجیستر شماره 5 نوشته میشود.

دستور 7:

SBC R6, R0, R0, LSR #1 =&gt; R6 = 10

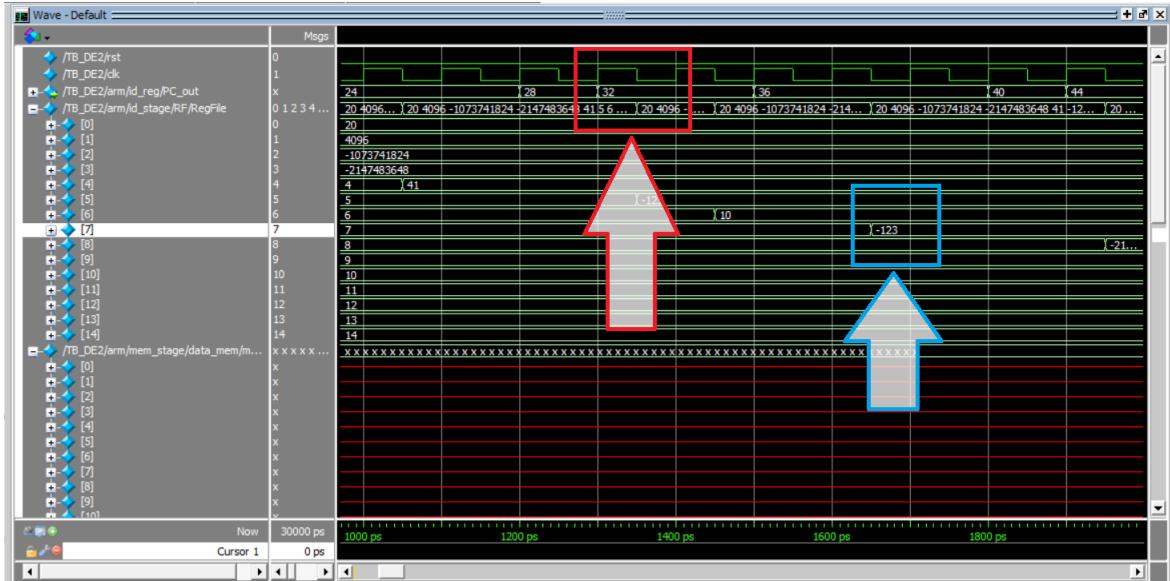
32'b1110\_00\_0\_0110\_0\_0000\_0110\_000010100000



همانطور که مشاهده میشود مقدار 10 در رجیستر شماره 6 نوشته میشود.

دستور: 8

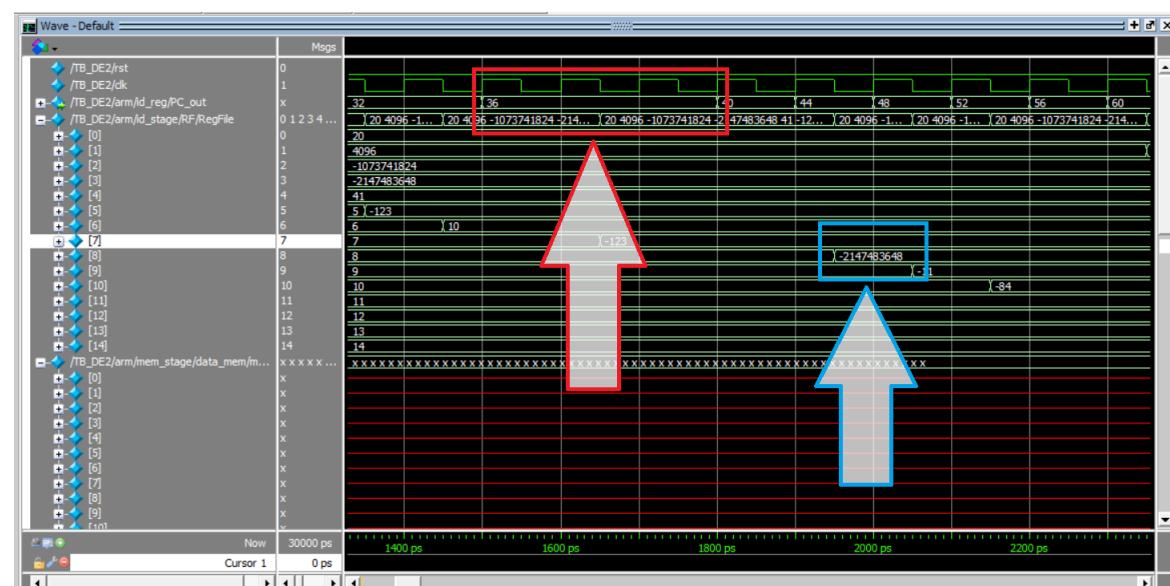
ORR R7, R5, R2, ASR #2 => R7 = -123  
 32'b1110\_00\_0\_1100\_0\_0101\_0111\_000101000010



در R7 مقدار -123 نوشته می شود.

دستور: 9

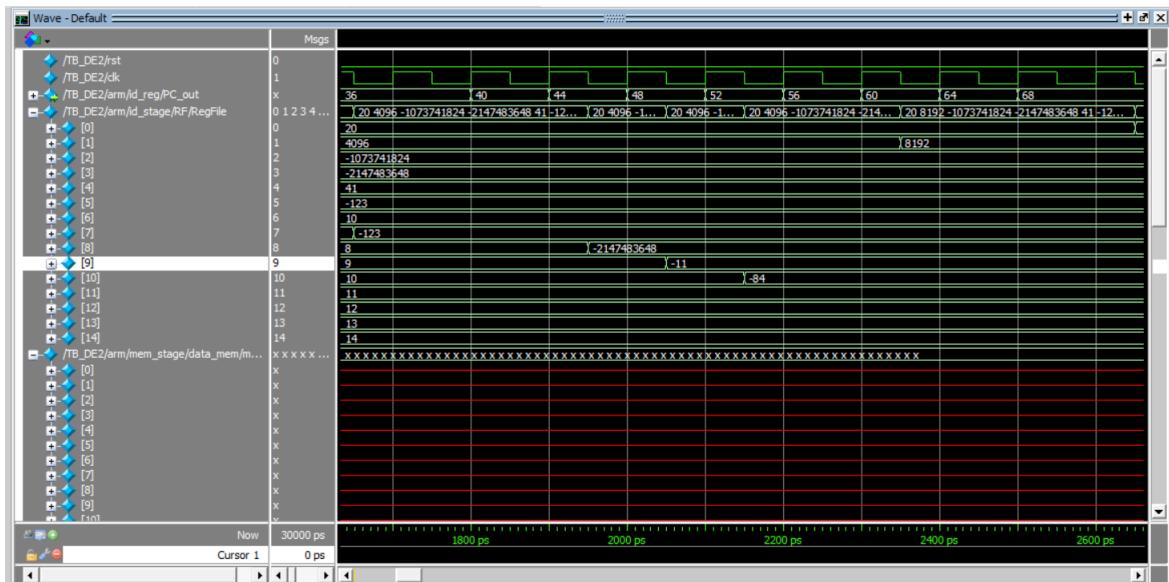
AND R8, R7, R3 => R8 = -2147483648  
 32'b1110\_00\_0\_0000\_0\_0111\_1000\_000000000011



مقدار -2147483648 در R8 نوشته میشود.

دستور 10:

MVN R9, R6 =&gt; R9 = -11

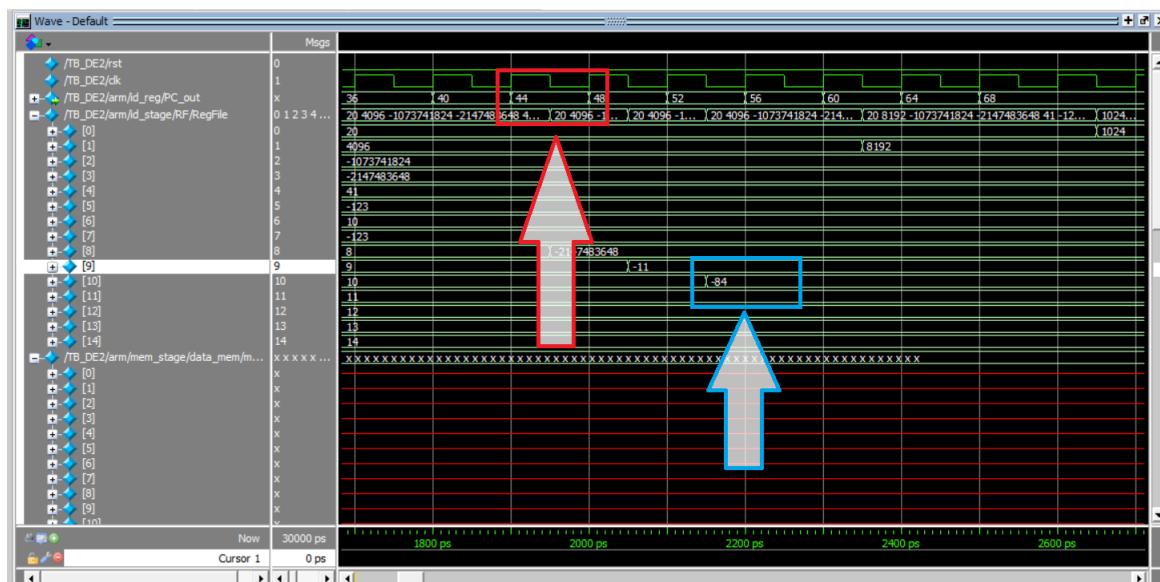


همانطور که مشاهده میشود فلش قرمز نشان میدهد که دستور 10 خوانده شده و سپس مقدار

11- در رجیستر شماره 9 نوشته می شود. همانطور که فلش آبی نشان میدهد.

دستور 11:

EOR R10, R4, R5 =&gt; R10 = -84



مقدار -84- در رجیستر R10 نوشته میشود.

دستور 12:

CMP R8, R6

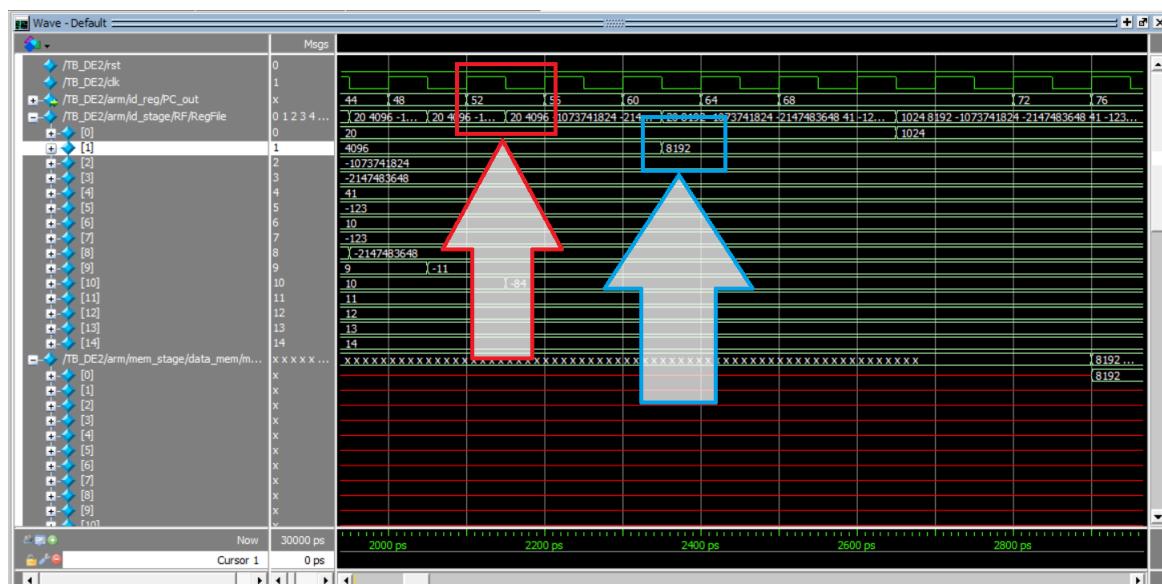
32'b1110\_00\_0\_1010\_1\_1000\_0000\_0000000000110

در دستور 12 مقدار رجیستر 8 و رجیستر 6 مقایسه میشود.

دستور 13:

ADDNE R1, R1, R1 =&gt; R1 = 8196

32'b0001\_00\_0\_0100\_0\_0001\_0001\_0000000000000001



پس همانطور که مشاهده می شود به درستی مقدار 8196 در رجیستر R1 نوشته میشود.

دستور 14:

TST R9, R8

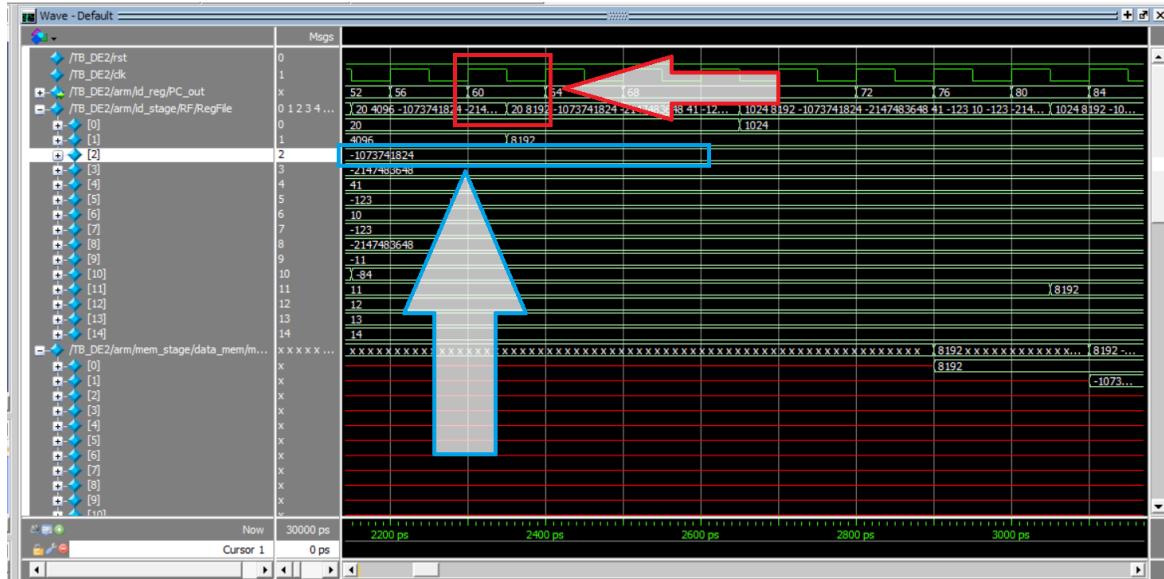
32'b1110\_00\_0\_1000\_1\_1001\_0000\_000000001000

این دستور عملیات AND را اجرا می کند و رجیستر وضعیت را بروزرسانی میکند.

## دستور 15:

ADDEQ R2, R2, R2 => R2 = -1073741824

32'b0000\_00\_0\_0100\_0\_0010\_0010\_000000000010

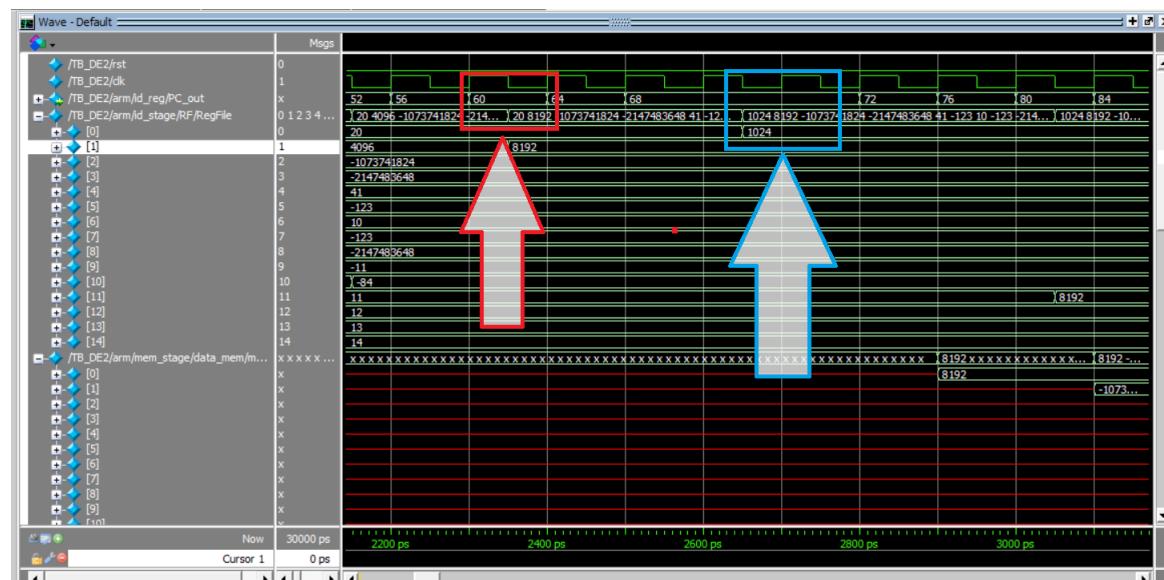


به درستی مقدار رجیستر شماره 2 همان مقدار -1073741824- حفظ می کند.

## دستور 16:

MOV R0, #1024 => R0 = 1024

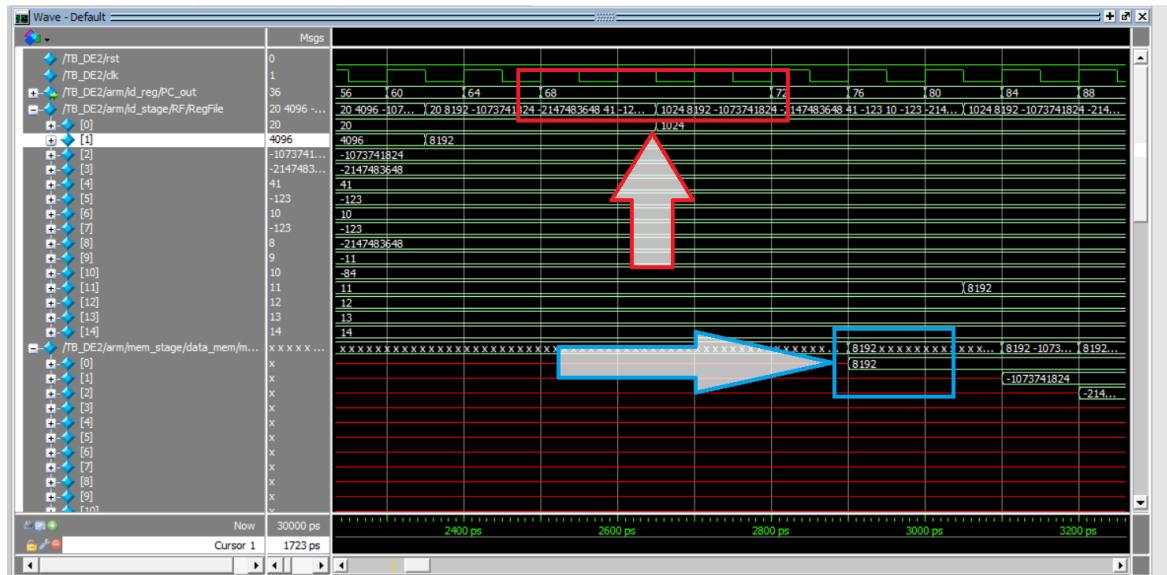
32'b1110\_00\_1\_1101\_0\_0000\_0000\_101100000001



همان طور که انتظار داشتیم مقدار 1024 در رجیستر R0 مینشیند.

## دستور 17:

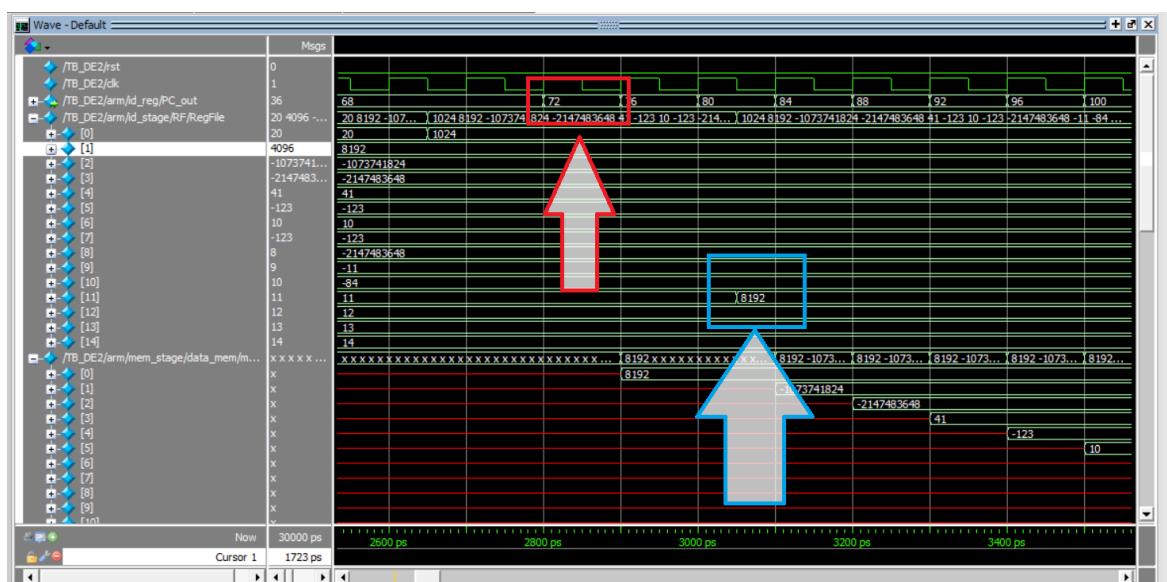
STR R1, [R0], #0 => MEM[1024] = 8192  
 32'b1110\_01\_0\_0100\_0\_0000\_0001\_000000000000



همانطور که مشاهده میشود مقدار 8192 در خانه صفر حافظه که آدرس 1024 را دارد، می‌نشیند.

## دستور 18:

LDR R11, [R0], #0 => R11 = 8192  
 32'b1110\_01\_0\_0100\_1\_0000\_1011\_000000000000;

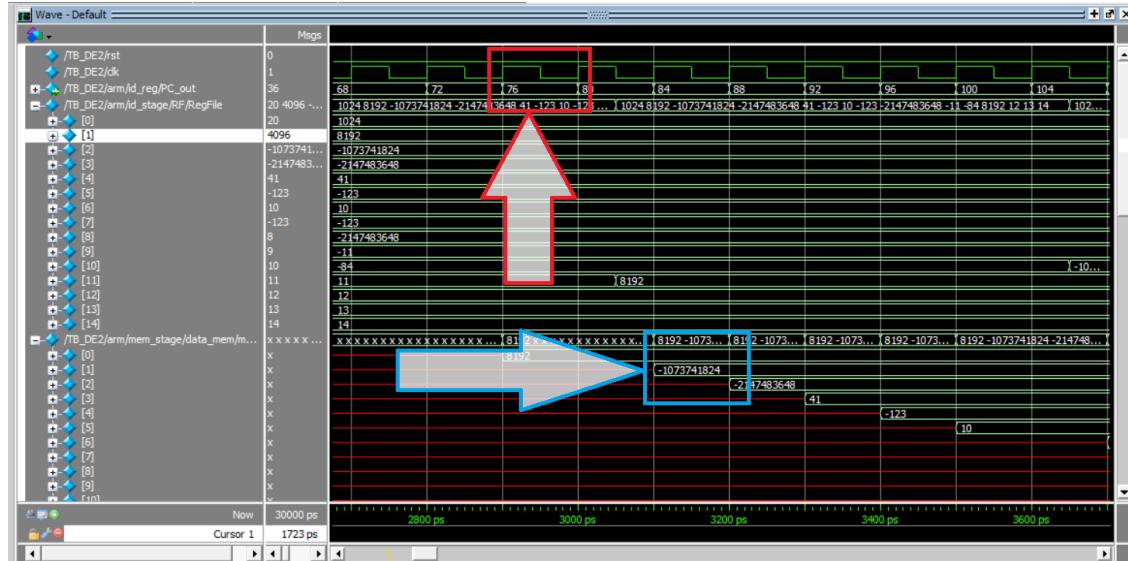


طبق تصویر بالا مقدار 8192 در رجیستر شماره 11 می‌نشیند.

## دستور 19:

STR R2, [R0], #4 => MEM[1028] = -1073741824

32'b1110\_01\_0\_0100\_0\_0000\_0010\_000000000100

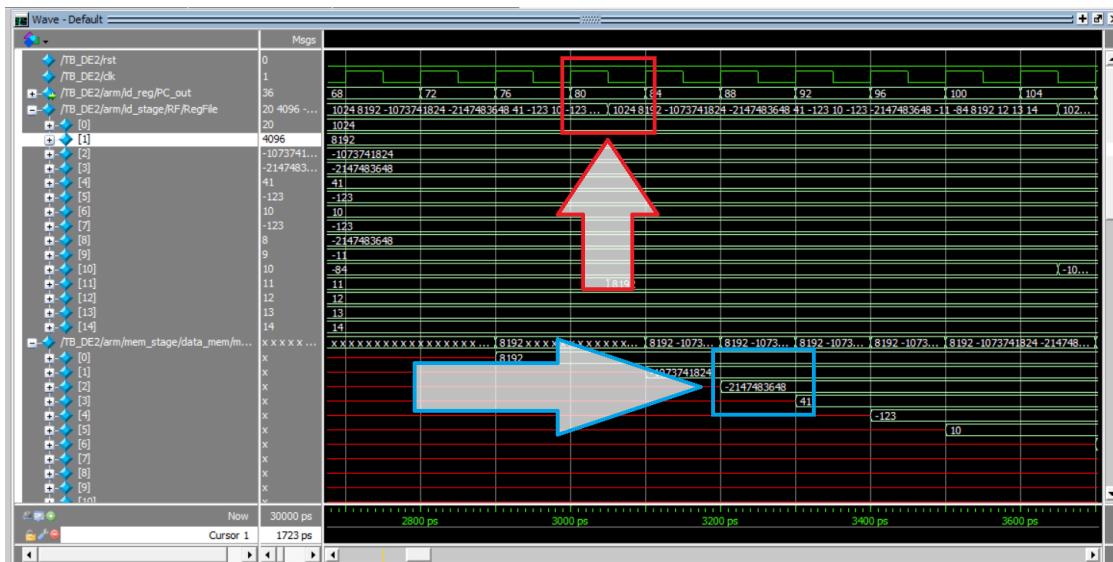


همانطور که مشاهده میشود در خانه شماره دو مموری با آدرس 1028 مقدار -1073741824 نشینید.

## دستور 20:

STR R3, [R0], #8 => MEM[1032] = -2147483648

32'b1110\_01\_0\_0100\_0\_0000\_0011\_0000000001000

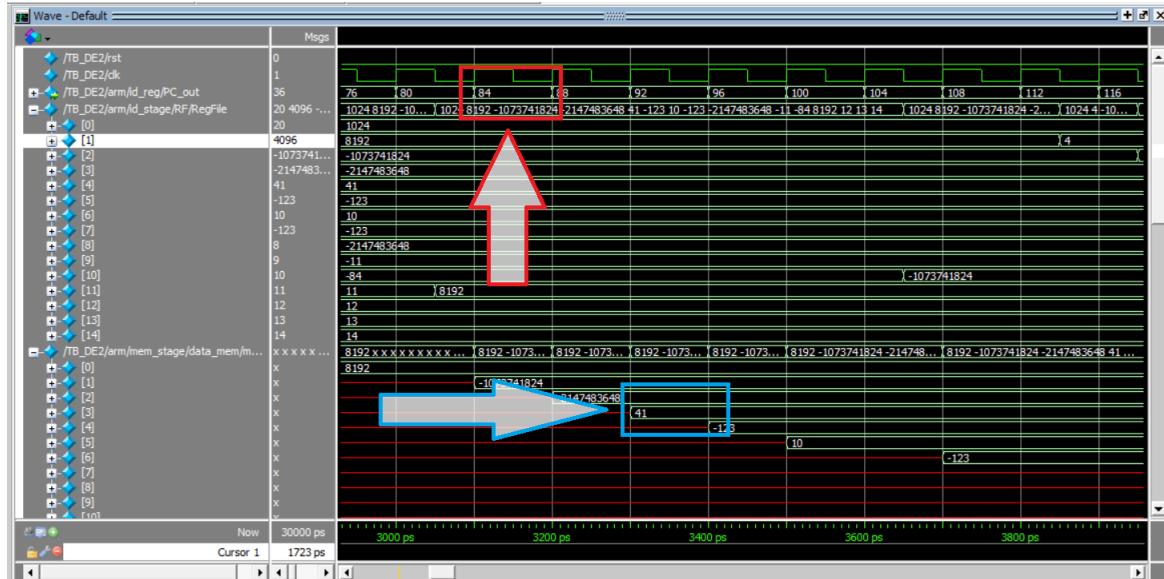


مقدار 2147483648- در آدرس 1032 نشینید.

دستور 21

STR R4, [R0], #13 =&gt; MEM[1036] = 41

32'b1110\_01\_0\_0100\_0\_0000\_0100\_000000001101

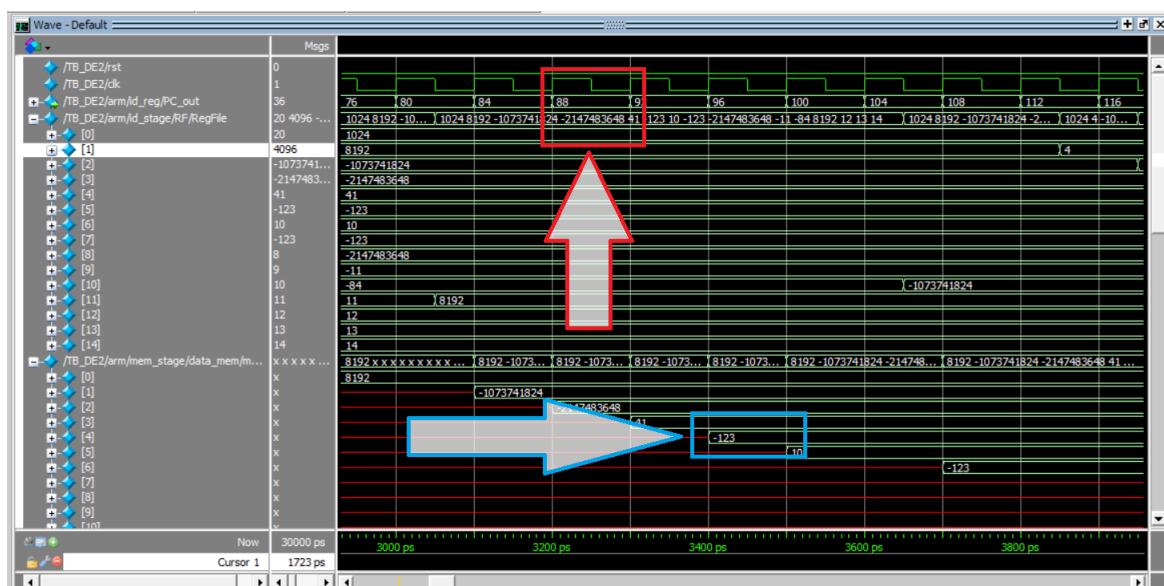


مقدار 41 در خانه سوم مموری با آدرس 1036 مینشیند.

دستور 22

STR R5, [R0], #16 =&gt; MEM[1040] = -123

32'b1110\_01\_0\_0100\_0\_0000\_0101\_000000010000

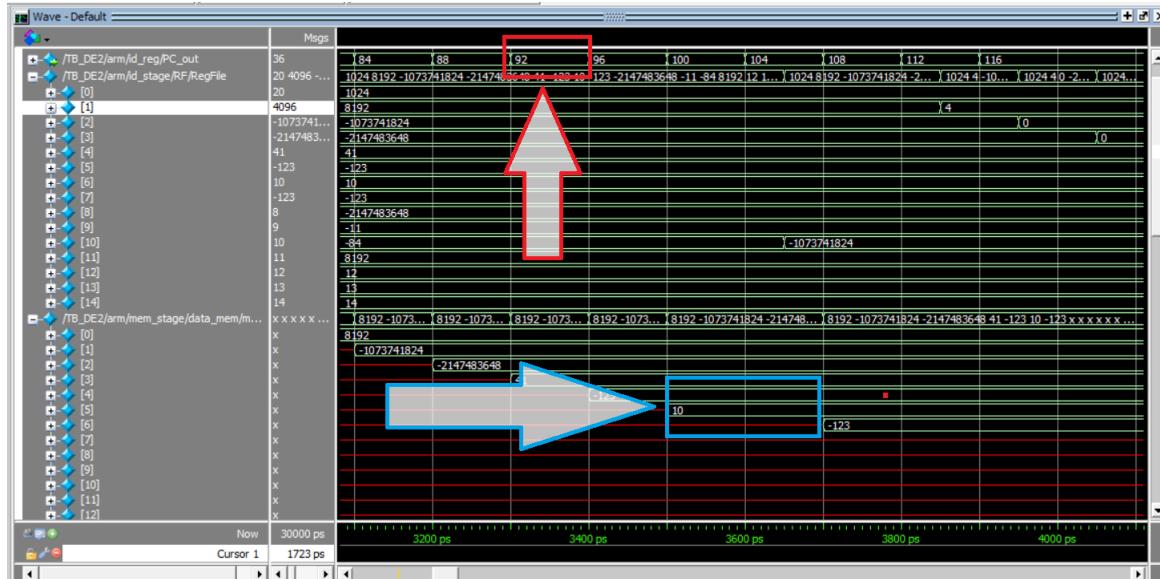


با اجرای دستور 22 ام از برنامه محک مقدار -123- در مموری با آدرس 1040 مینشیند.

## دستور 23:

STR R6, [R0], #20 => MEM[1044] = 10

32'b1110\_01\_0\_0100\_0\_0000\_0110\_000000010100

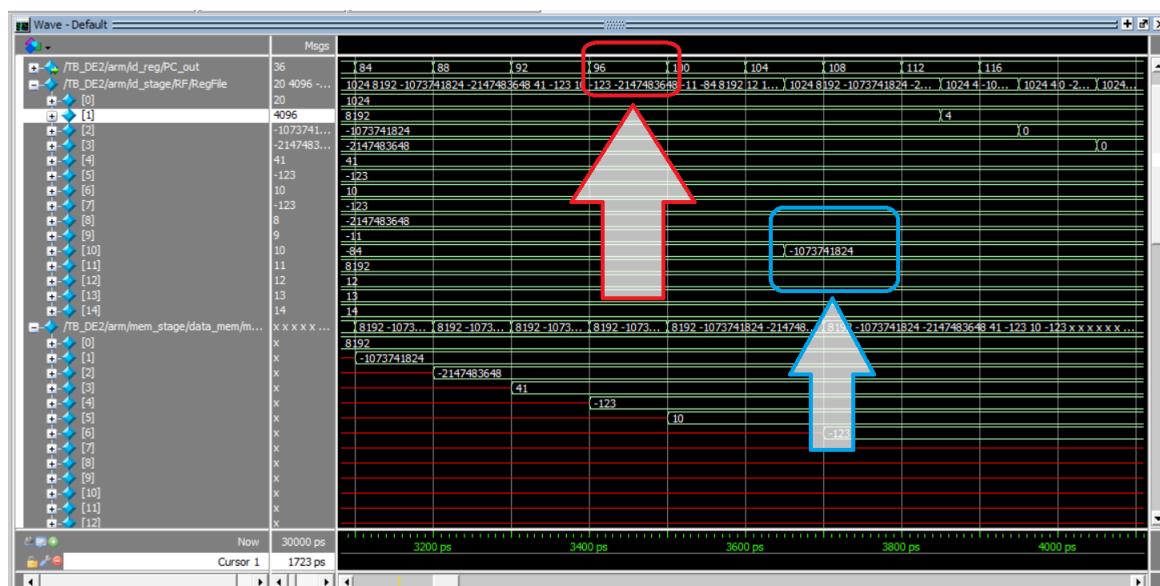


مقدار 10 در آدرس 1044 مموری و در خانه 5 آن مینشیند.

## دستور 24:

LDR R10, [R0], #4 => R10 = -1073741824

32'b1110\_01\_0\_0100\_1\_0000\_1010\_000000000100

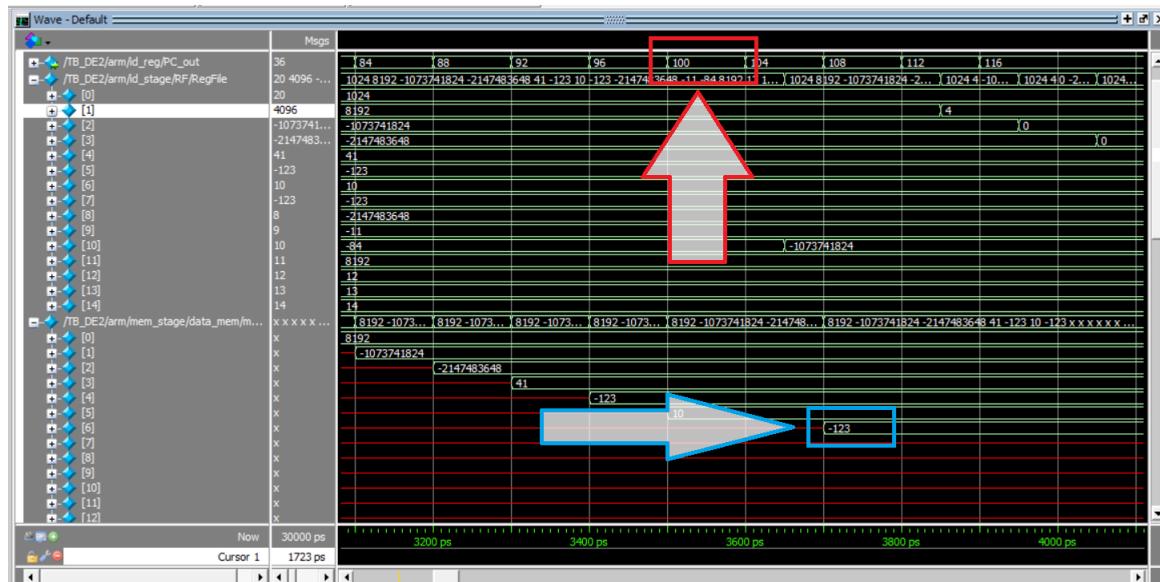


همان طور که در تصویر نیز قابل مشاهده است مقدار R10 به درستی تعیین میشود.

دستور 25:

STR R7, [R0], #24 =&gt; MEM[1048] = -123

32'b1110\_01\_0\_0100\_0\_0000\_0111\_000000011000



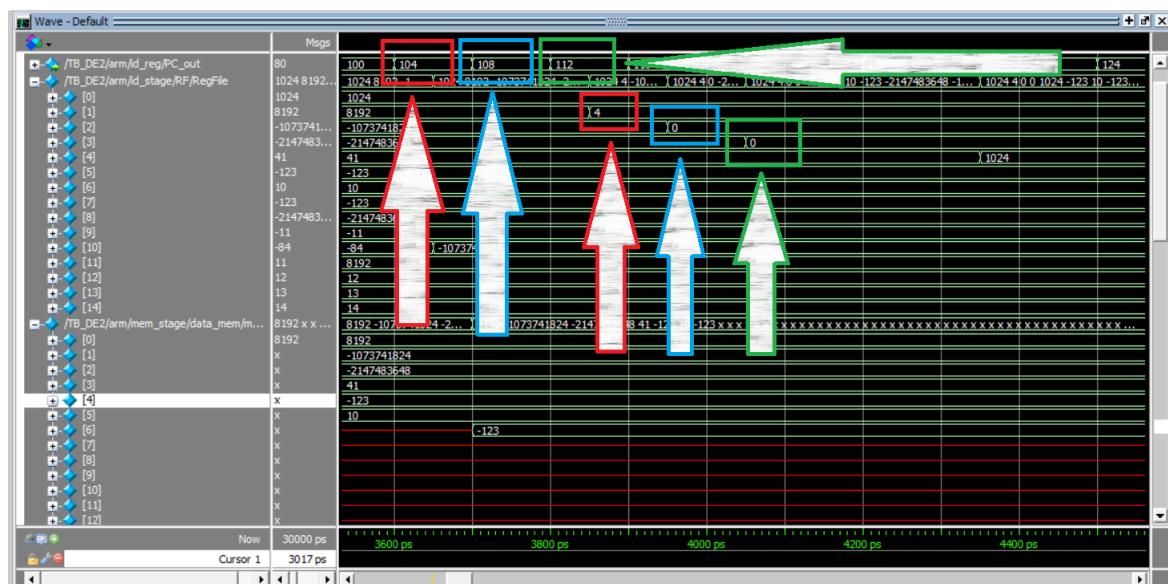
مقدار -123- به درستی در خانه 6 مموری به آدرس 1048 مینشیند.

در دستورات بعدی شروع به مرتب سازی رجیستر های R1, R2, R3, R4 پس از ذخیره سازی آنها در حافظه میپردازیم.

دقت شود که یک حلقه تو در تو در ادامه مشاهده میکنیم و ما فقط یک با این اجرای این حلقه را نشان میدهیم.

### دستور 26 و 27 و 28:

26. 32'b1110_00_1_1101_0_0000_0001_0000000000100; //MOV	R1 ,#4	//R1 = 4
27. 32'b1110_00_1_1101_0_0000_0010_0000000000000; //MOV	R2 ,#0	//R2 = 0
28. 32'b1110_00_1_1101_0_0000_0011_0000000000000; //MOV	R3 ,#0	//R3 = 0



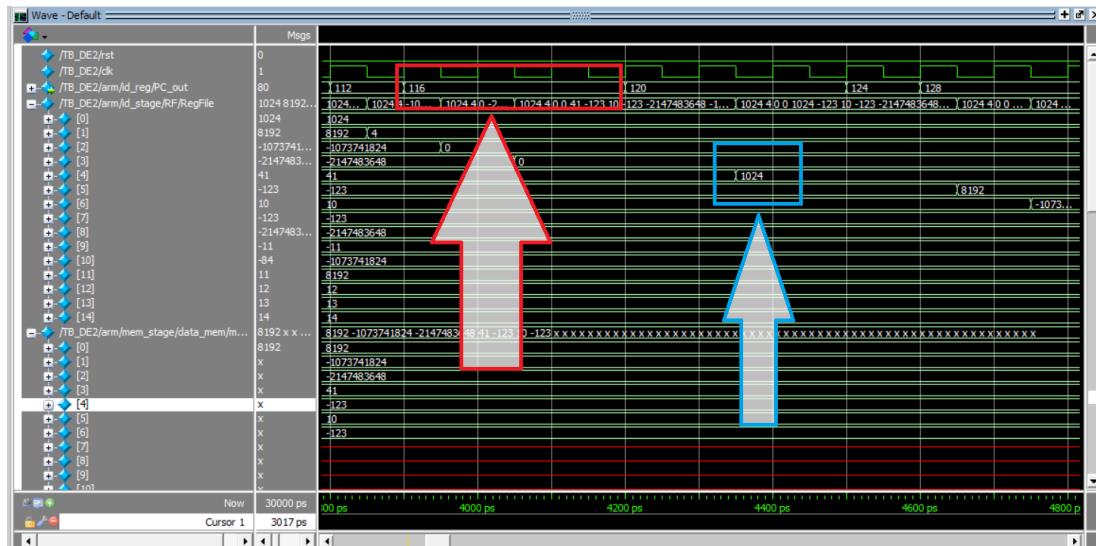
همانطور که مشاهده میشود بعد از اجرای این سه دستور مقدار 4 در رجیستر یک و مقدار صفر در رجیستر های دو و سه نوشته میشود.

## دستور 29:

ADD R4, R0, R3, LSL #2

32'b1110\_00\_0\_0100\_0\_0000\_0100\_000100000011

در این دستور مجموع R0 و R3 که دو واحد شیفت پیدا میکند در رجیستر شماره 4 نوشته میشود. پس باید در R4 مقدار 1024 نوشته شود.



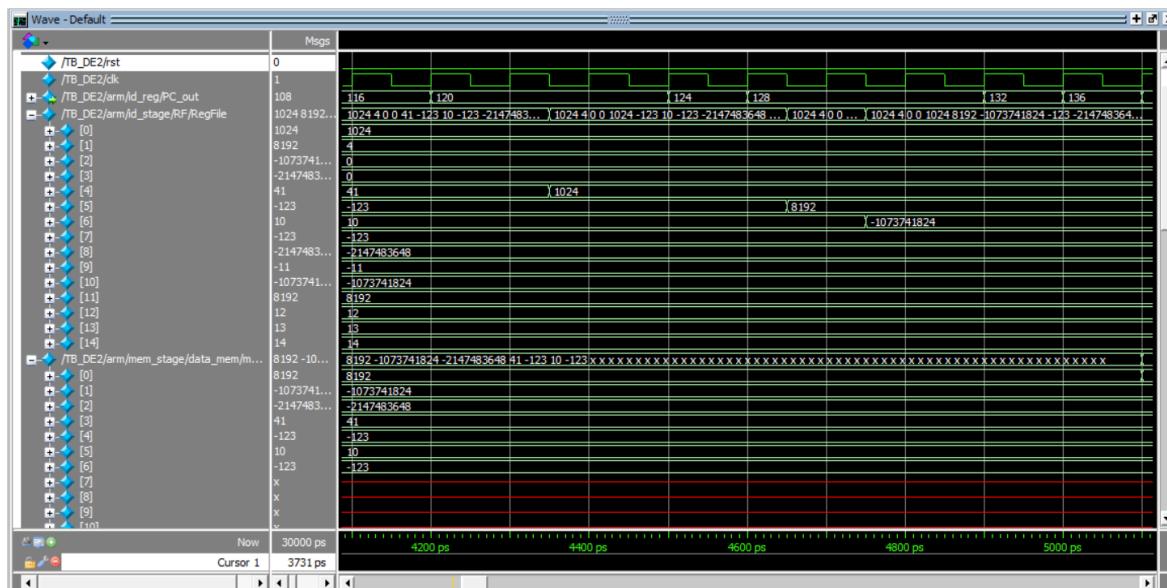
همانطور که مشاهده میشود مقدار 1024 در R4 نوشته میشود.

## دستورات 30 و 31:

30. 32'b1110\_01\_0\_0100\_1\_0100\_0101\_000000000000; //LDR R5,[R4],#0

31. 32'b1110\_01\_0\_0100\_1\_0100\_0110\_00000000100; //LDR R6,[R4],#4

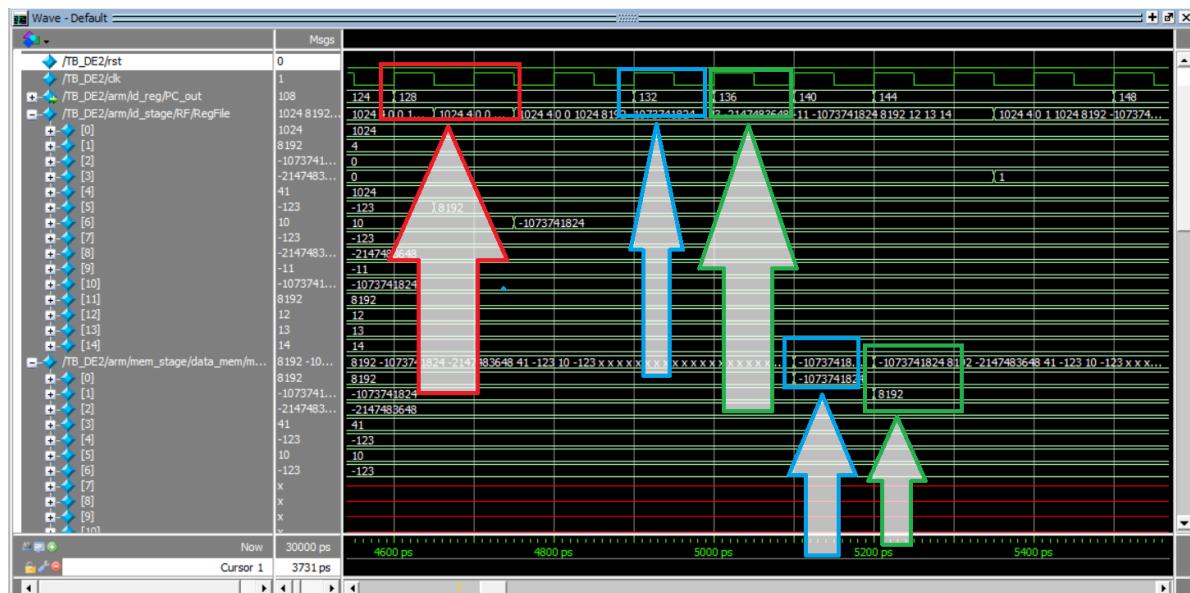
در این دو دستور باید خانه های صفر و یک مموري در رجیستر های شماره 5 و 6 نوشته شوند:



## دستورات 32 و 33 و 34:

32. 32'b1110_00_0_1010_1_0101_0000_000000000110; //CMP	R5 ,R6
33. 32'b1100_01_0_0100_0_0100_0110_000000000000; //STRGT	R6 ,[R4],#0
34. 32'b1100_01_0_0100_0_0100_0101_000000000100; //STRGT	R5 ,[R4],#4

پس از اجرای این سه دستور مقادیر R5 و R6 مقایسه شده و در صورت بزرگتر بودن رجیستر R5 دستورات STR اجرا می‌گردند (هدف از این سه خط و سه خط بالا مرتب سازی خانه های 0 و 1 مموری می‌باشد).

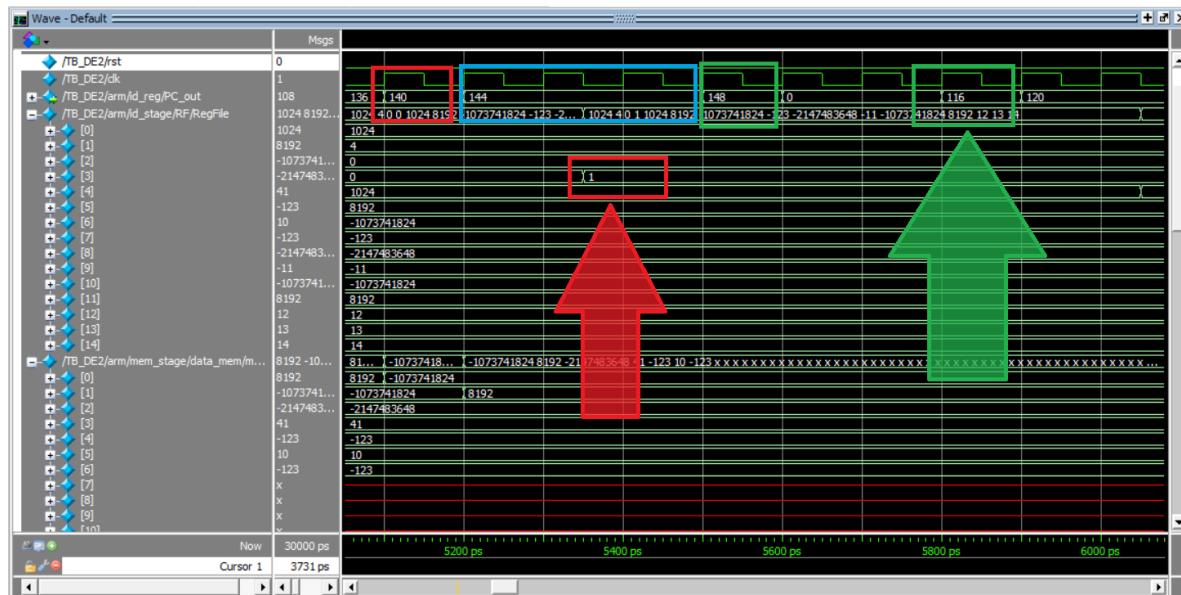


همان طور که مشاهده می‌شود به درستی این عمل صورت می‌پذیرد.

## دستورات 35 و 36 و 37:

35. 32'b1110_00_1_0100_0_0011_0011_0000000000000001; //ADD	R3 ,R3,#1
36. 32'b1110_00_1_1010_1_0011_0000_00000000000011; //CMP	R3 ,#3
37. 32'b1011_10_1_0_111111111111111111110111 ; //BLT	#-9

پس از اجرای این سه دستور مقدار R3 یکی زیاد شده و با عدد 3 مقایسه میشود و در صورت بزرگتر و یا مساوی بودن دستور برنج اجرا میگردد.

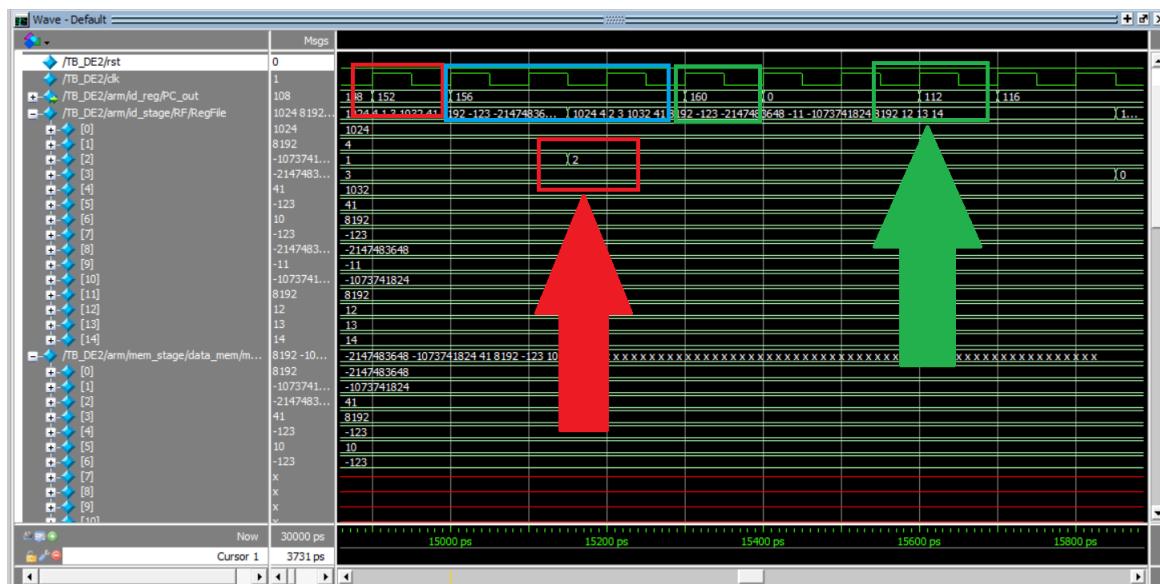


همانطور که مشاهده میشود مقدار رجیستر شماره سه به مقدار یک عدد اضافه شده و سپس با درست بودن دستور CMP مقدار PC ما به 9 دستور عقب تر (36 واحد کمتر) منتقل میشود.

## دستورات 38 و 39 و 40:

38. 32'b1110\_00\_1\_0100\_0\_0010\_0010\_000000000001; //ADD R2 ,R2,#1  
 39. 32'b1110\_00\_0\_1010\_1\_0010\_0000\_000000000001; //CMP R2 ,R1  
**40. 32'b1011\_10\_1\_0\_111111111111111111110011 ; //BLT #-13**

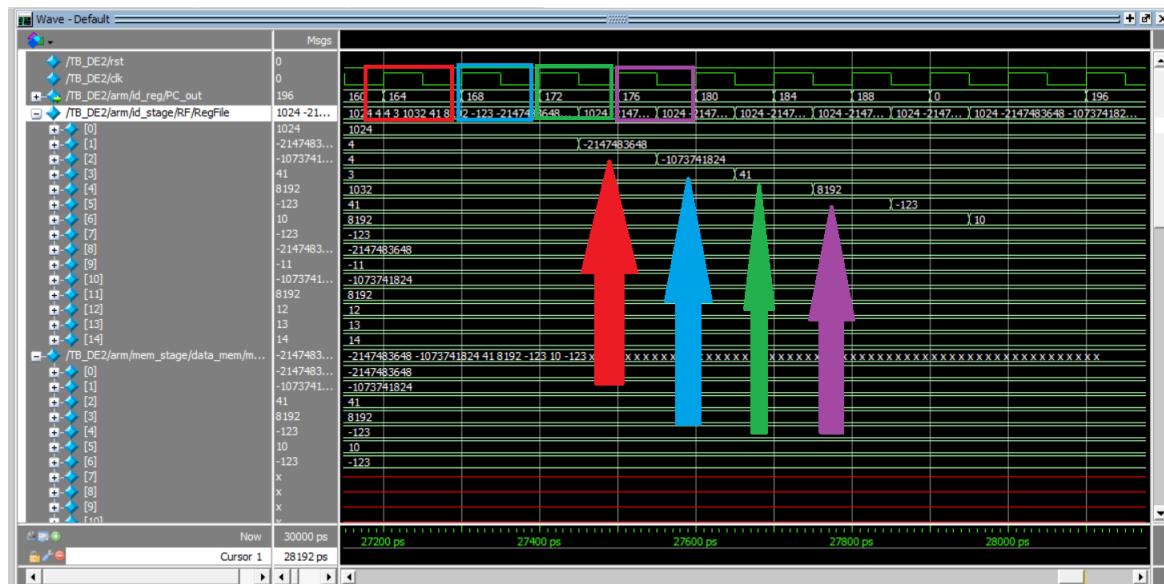
این سه دستور نیز برای اجرای شرط حلقه دوم می‌باشند:  
 ابتدا R2 یکی زیاد شده و با R1 مقایسه گشته و در صورت بزرگتر و یا مساوی بودن به 13 دستور عقب‌تر باز می‌گردیم.



دستورات 41 و 42 و 43 و 44:

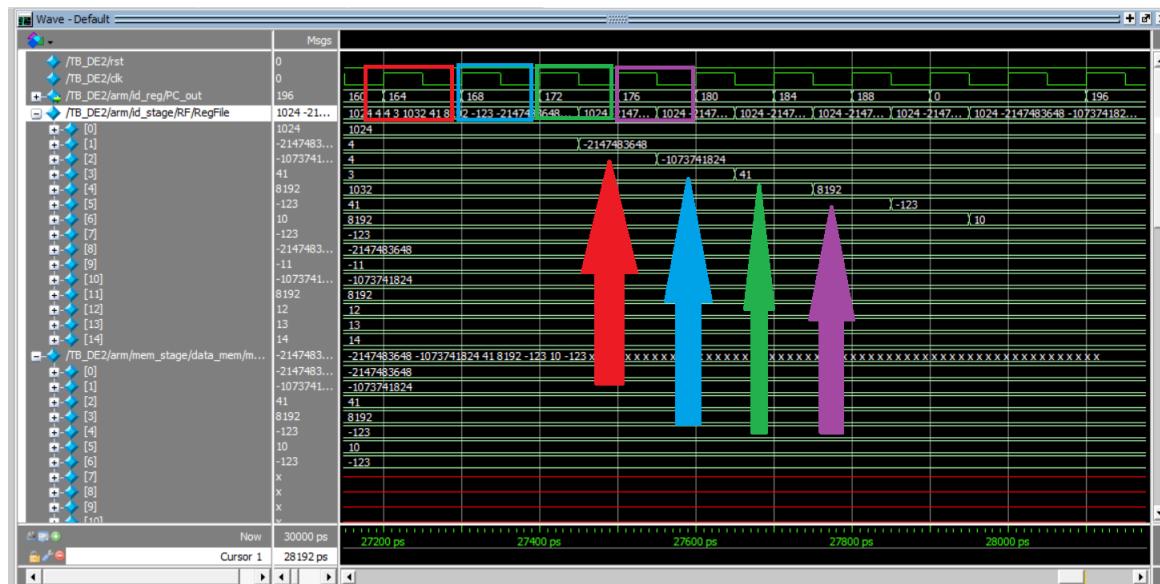
41. 32'b1110_01_0_0100_1_0000_0001_000000000000; //LDR	R1,[R0],#0	//R1 = -2147483648
42. 32'b1110_01_0_0100_1_0000_0010_000000000100; //LDR	R2,[R0],#4	//R2 = -1073741824
43. 32'b1110_01_0_0100_1_0000_0011_000000001000; //LDR	R3,[R0],#8	//R3 = 41
44. 32'b1110_01_0_0100_1_0000_0100_000000001100; //LDR	R4,[R0],#12	//R4 = 8192

و در انتهای پس از بیرون آمدن از دو حلقه مرتب‌سازی صورت گرفته و کافی است که مقادیر در رجیسترها R1 و R2 و R3 و R4 نوشته شود.



همانطور که مشاهده می‌شود مقادیر در 4 رجیستر مربوطه نوشته می‌شود.

## جواب نهایی :SIMULATION



همانطور که مشاهده میکنید 4 رجیستر بالا مرتب شده‌اند.

## جواب نهایی :SYNTHESIZER

پس از موفقیت آمیز بودن جواب‌ها در مرحله شبیه سازی پیاده سازی آن برروی FPGA پرداختیم و با استفاده از SIGNAL TAB مقادیر 4 رجیستر مورد نظر را مشاهده کردیم:

Type	Alias	Node	0 :egmer	0 :31Value:30	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240	256	272	288	304	320	336
		SW[0]	1																							
		:i...ile:RF RegFile[0]	00000000h		:000000014h)																					
		:i...ile:RF RegFile[1]	1	1	( X ) 4096 ( 8192 )																					
		:i...ile:RF RegFile[2]	2	2	( -1073741824 )	0	( 1 ) ( 2 ) ( 3 ) ( 0 ) ( 1 ) ( 2 ) ( 3 ) ( 0 ) ( 1 ) ( 2 ) ( 3 ) ( 0 ) ( 1 ) ( 2 ) ( 3 ) ( 0 ) ( 1 ) ( 2 ) ( 3 ) ( 4 ) ( -1073741824 )																			
		:i...ile:RF RegFile[3]	3	3	( -2147483648 )	0	( 1 ) ( 2 ) ( 3 ) ( 0 ) ( 1 ) ( 2 ) ( 3 ) ( 0 ) ( 1 ) ( 2 ) ( 3 ) ( 0 ) ( 1 ) ( 2 ) ( 3 ) ( 0 ) ( 1 ) ( 2 ) ( 3 ) ( 4 ) ( -2147483648 )																			
		:i...ile:RF RegFile[4]	4	4	( 41 )	1024	( 1028 )	1032	( 1024 )	1028	( 1032 )	1024	( 1028 )	1032	( 1024 )	1028	( 1032 )	1024	( 1028 )	1032	( 1024 )	1028	( 1032 )	8192	( )	

همانطور که در تصویر نیز قابل مشاهده است 4 رجیستر مربوطه مرتب سازی شده‌اند و به طور کامل و صحیح این فاز از پروژه به اتمام رسید.