گزارش کار پروژه fpga

علی زینداری 9629183

فاز اول : دستورات add , addi

در این قسمت قرار است ابتدا دو مقدار در دو رجیستر load شوند و سپس این دو رجیستر با یکدیگر جمع شوند و جواب نهایی نیز در یک رجیستر دیگر ریخته شود.

برای ریختن عدد لازم است که رجیستر 0 را که مقدار آن 0 میباشد را با آن عدد جمع کرده و در رجیستر مقصد بریزیم برای خروجی های این قسمت به اسکرین شات های زیر دقت کنید:

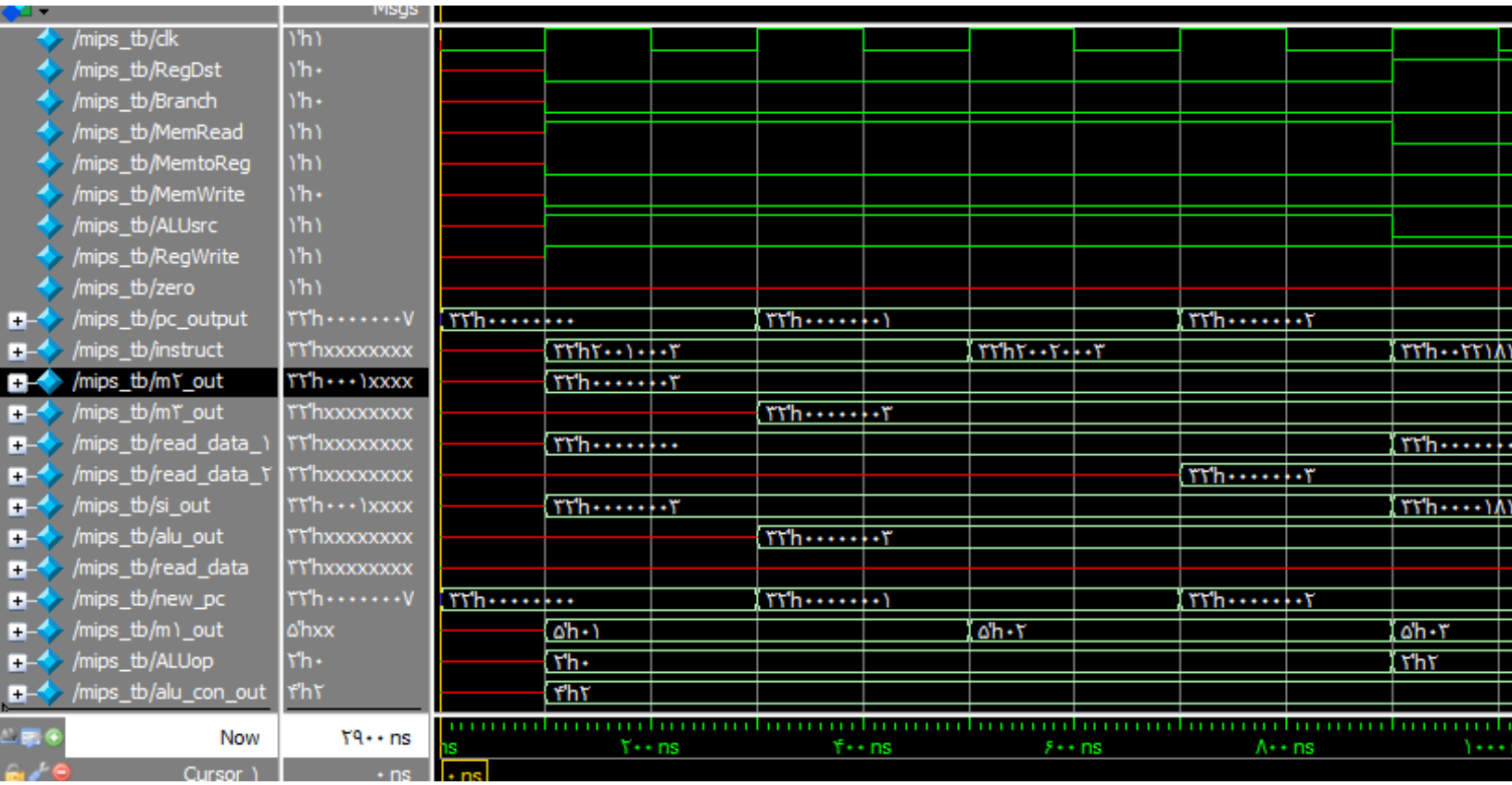
در اینجا دستور شماره یک از حافظه ی دستورات loadشده در حقیقت مقدار pc صفر است و سطر صفر دستورات بیرون امده که در شکل نیز مشخص است . بلافاصله پس از اولین کلاک خروجی ماکس 1 (m1_out) برابر ادرس رجیستر مقصد شده که همان 1است

خروجی ماکس 2 نیز (m2_out) همان عدد مد نظر ما یعنی 3 است که قرار است با صفر جمع شود.

نهایتا در کلاک بعدی خروجی alu ما یعنی 0+3 مشخص میشود که همان 3 است و در شکل نیز مشخص است.(alu_out)

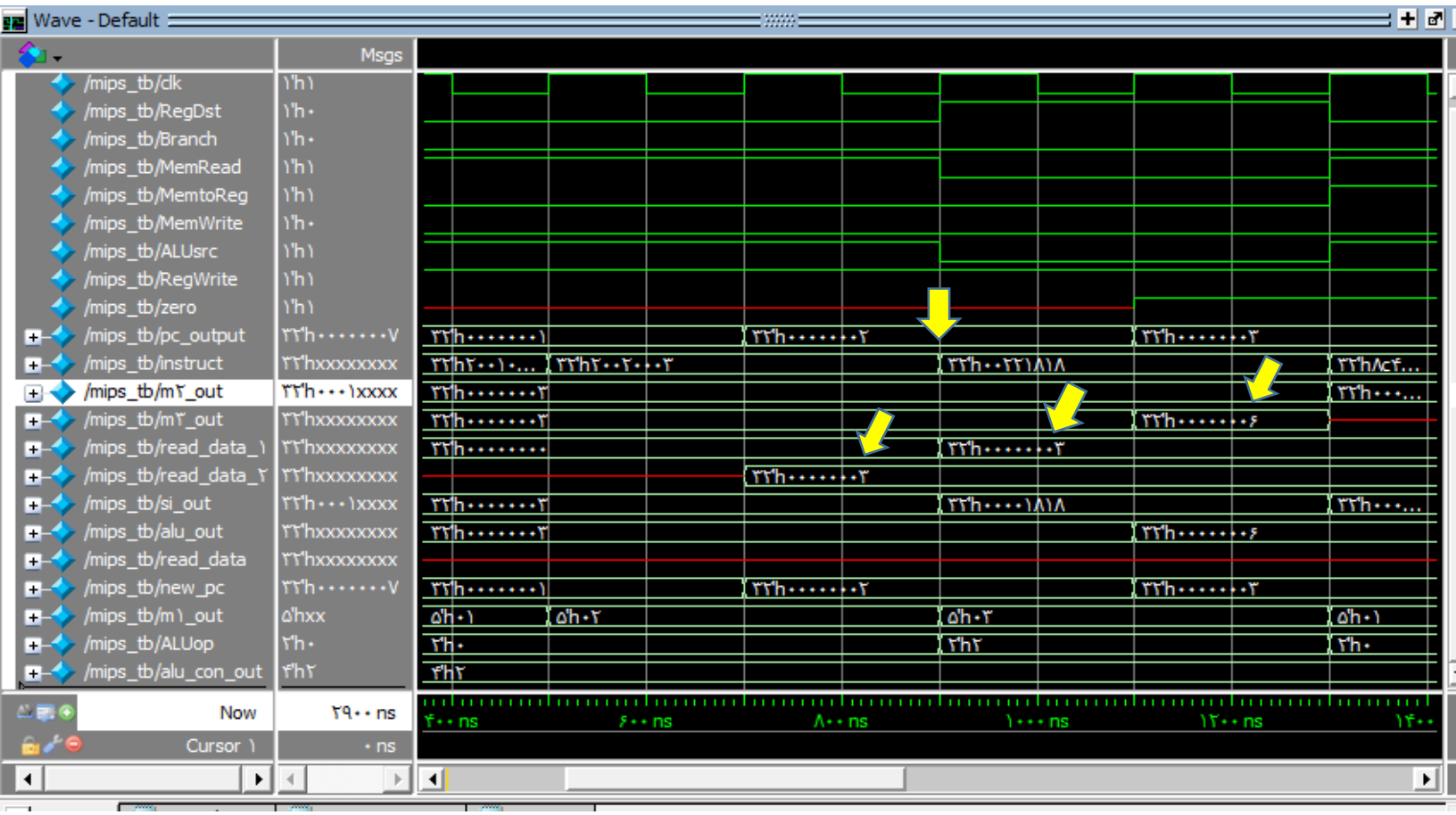
در نهایت باید خروجی ماکس 3 (m3_out) را دید که قرار است بین خروجی رم و خروجی alu یکی را انتخاب کند که در این مورد خاص خروجی مدنظر همان خروجی aluاست و در شکل هم مشخص است در کلاک بعد نیز این مقدار در رحیستر مقصد ریخته میشود.

همه ی این موارد برای دستور دوم نیز صادق است و همین کار ها دقیقا انجام میشود فقط ادرس رجیستر ها متفاوت است.


Msgs
/mips_tb/clk
/mips_tb/RegDst
/mips_tb/Branch
/mips_tb/MemRead
/mips_tb/MemtoReg
/mips_tb/MemWrite
/mips_tb/ALUsrc
/mips_tb/RegWrite
/mips_tb/zero
/mips_tb/pc_output
/mips_tb/instruct
/mips_tb/m۲_out
/mips_tb/m۳_out
/mips_tb/read_data_۱
/mips_tb/read_data_۲
/mips_tb/si_out
/mips_tb/alu_out
/mips_tb/read_data
/mips_tb/new_pc
/mips_tb/m۱_out
/mips_tb/ALUop
/mips_tb/alu_con_out
Now
Cursor ۱
۲۰۰ ns
۴۰۰ ns
۶۰۰ ns
۸۰۰ ns

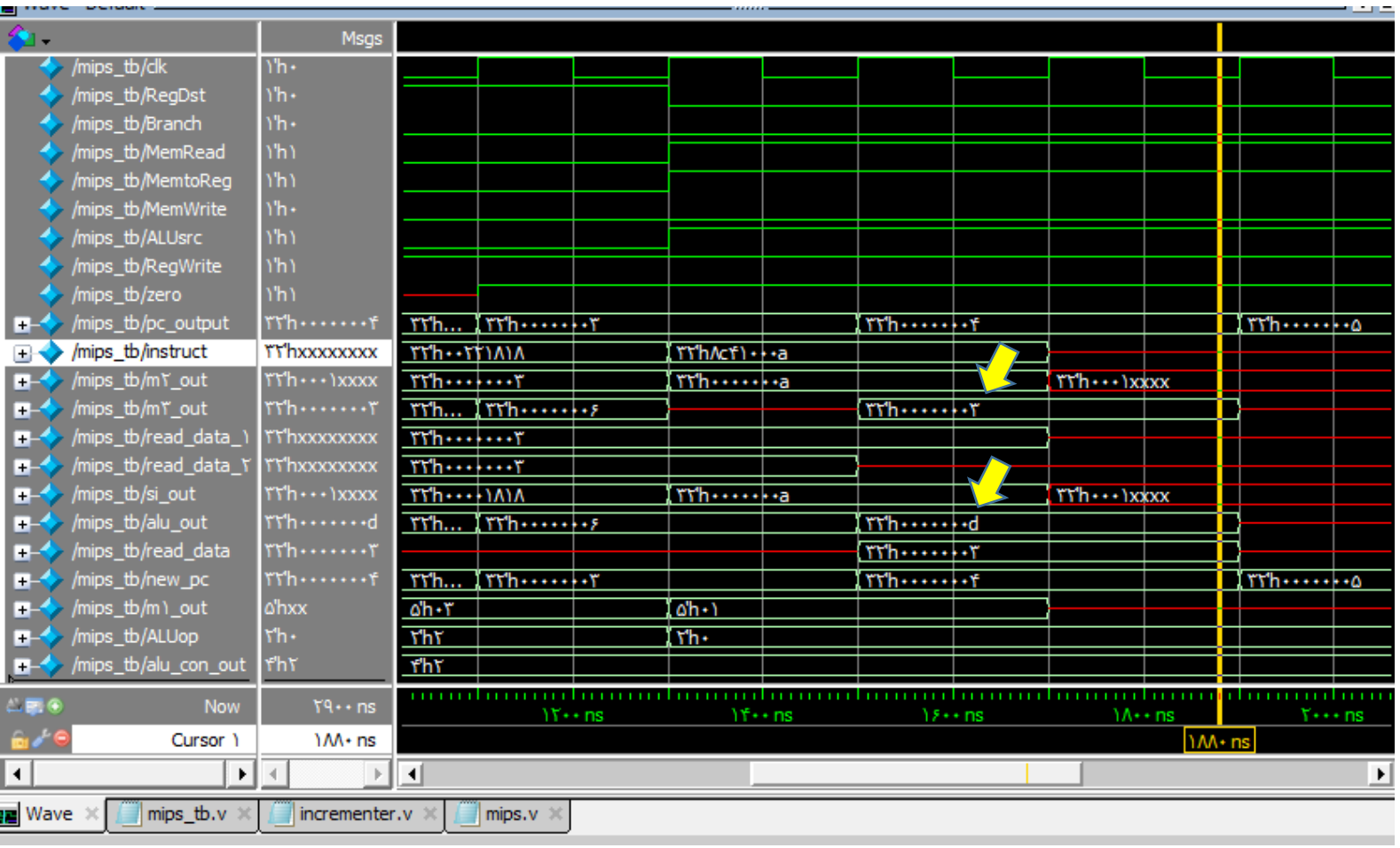
در مرحله ی بعد ما دو رجیستر که با مقدار 3 پر شده اند را اماده داریم و قرار است که این دو با هم جمع شوند و مقدار نهایی در یک رجیستر ریخته شود.

در نقطه ای که فلش مشخص کرده دستور سوم fetch میشود که همان دستور جمع دو رجیستر است و در همان لحظه میتوان خروجی های بانک رجیستر را مشاهده کرد که 2 عدد سه میباشد(read_data1,read_data_2) حال این دو عدد وارد alu میشوند و با کلاک بعد حاصل جمع انها مشخص میشود و داخل رجیستر مقصد نوشته میشود. این مطلب در شکل هم مشخص است و خروجی alu درکلاک بعدی برابر 3+3=6 شده است. ودر نهایت خروجی ماکس 3 مشخص میکند که داده ای که از alu امده است را انتخاب کند و این موضوع از شکل قابل مشاهده است.(m3_out =6) و در نهایت با کلاک بعدی این مقدار در رجیستر مقصد نوشته خواهد شد.



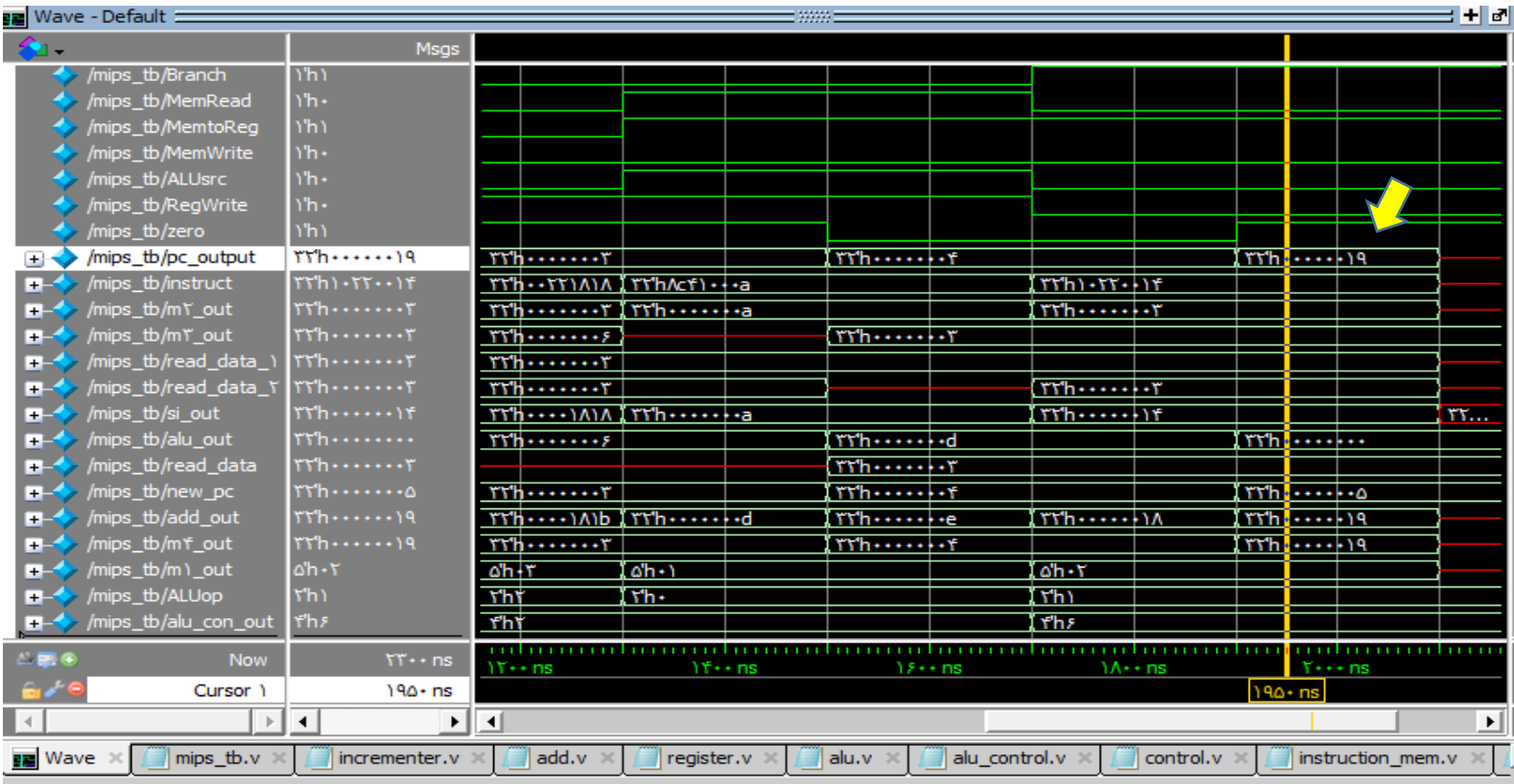
قسمت بعدی که مربوطه به شکل پایین برای دستور lw میباشد که قرار است یک خانه ی رم را که ادرس پایه ی ان در بانک رجیستر است را بخوانیم و با مقدار 10 جمع کنیم و ادرس حاصل بدست امده را از رم بخوانیم و در رجیستر مقصد ذخیره کنیم

این دستور در لحظه ای که نشان داده شده است fetch میشود و بلافاصله میتوان خروجی ماکس1(m1_out) را مشاهده کرد که برابر 1 شده که همان رجیستر R1 مقصد میباشد.در کلاک بعد میتوان دید که ادرس پایه با مقدار 10 در alu جمع میشود و همان موقع ادرس به رم فرستاده میشود و مقدار مد نظر ما که همان خانه ی 13 رم است که برابر با 3 است از ان خوانده میشود و در رجیستر مقصد نوشته میشود. خروجی alu همانطور که پیداست برابر 10+3=13 در مبنای هگز است و این مقدار به رم فرستاده میشود و مشاهده میشود که خروجی رم(read_data) برابر 3 شده که همان مقدار خانه ی 13 است سپس ماکس 3 باید بین خروجی رم و خروجی مستقیم alu یکی را انتخاب کند که در اینجا خروجی رم مد نظر ما است.(m3_out = 3) و سپس با کلاک بعدی این مقدار در رجیستر R1 نوشته میشود.



مرحله ی بعدی که یک دستور beq است به این شکل عمل میکند که دو رجیستر را مقایسه میکند و اگر محتوی انها یکسان بود یک پرش به خطی دیگر میکند و در حقیقت مقدار کنونی pc را با یک عدد مشخص جمع میکند.

فلش نشان داده شده همان جایی است که این دستور که در خانه ی 4 حافظه ی دستورات است fetch شده است و بلافاصله در کلاک بعد مشاهده میشود که سیگنال های beanch zero, هر دو 1 شده اند و این بدین معنی است که ماکس 4 باید مسیر دوم را برای اپدیت کردن مقدار جدید pc انتخاب کند که همین اتفاق نیز افتاده و میبینیم که خروجی m4_out و همینطور مقدار جدید pc برابر 25 شده است که 20+1+4 میباشد.



***نکات

در این ساختار ما نمیتوانیم در هر کلاک رجیستر pc را اپدیت کنیم و سراغ دستور بعدی برویم چونکه ما ساختار pipeline نداریم و باید صبر کنیم تا یک دستور تا اخر برود و وقتی کارش تمام شد دستور بعدی را وارد کنیم و pc را اپدیت کنیم برای اینکار نیاز است که یک تقسیم فرکانسی انجام دهیم و هر 3 کلاک یکبار pc را اپدیت کنیم که من هم این کار را با تعریف یک متغیر به عنوان شمارشگر کلاک انجام دادم.

همچنین باید دقت داشت عملیات های خواندن از رم هیچ کلاکی مصرف نمیکند و فقط نوشتن نیاز به کلاک دارد.

گزارش سنتز:

Primitive and Black Box Usage:

BELS : 747
GND : 1
INV : 2
LUT1 : 25
LUT2 : 37
LUT3 : 103
LUT4 : 143
LUT5 : 30
LUT6 : 106
MUXCY : 171
VCC : 1
XORCY : 128
FlipFlops/Latches : 159
FD : 42
FDE : 32
FDR : 1
LD : 80
LDCE : 2
LDPE : 2
RAMS : 3
RAMB8BWER : 3
Clock Buffers : 2
BUFG : 1
BUFGP : 1
IO Buffers : 403

OBUF : 403

Slice Logic Utilization:

Number of Slice Registers: 116 out of 18224 0%

Number of Slice LUTs: 446 out of 9112 4%

Number used as Logic: 446 out of 9112 4%

Slice Logic Distribution:

Number of LUT Flip Flop pairs used: 447

Number with an unused Flip Flop: 331 out of 447 74%

Number with an unused LUT: 1 out of 447 0%

Number of fully used LUT-FF pairs: 115 out of 447 25%

Number of unique control sets: 8

IO Utilization:

Number of IOs: 404

Number of bonded IOBs: 404 out of 232 174% (*)

IOB Flip Flops/Latches: 43

Specific Feature Utilization:

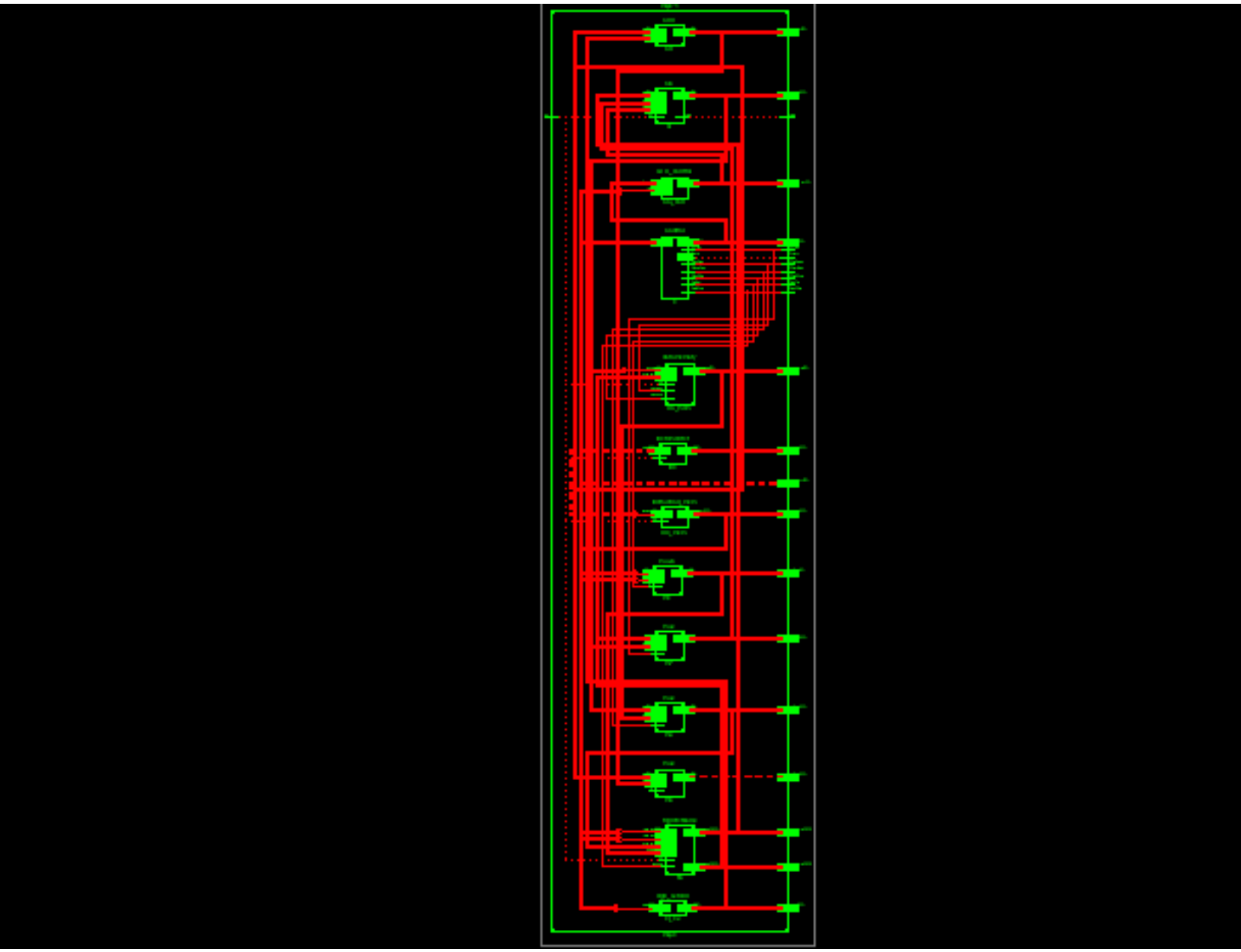
Number of Block RAM/FIFO: 2 out of 32 6%

Number using Block RAM only: 2

Number of BUFG/BUFGCTRLs: 2 out of 16 12%

مشاهده میشود که پورت هایی که مصرف کرده ایم از بردی که در اختیار داریم فراتر است و این به این دلیل است که برای مانیتور کردن خروجی تک تک ماژول ها باید همه ی انها را به عنوان خروجی تعریف میکردم.

Minimum period: 6.999ns (Maximum Frequency: 142.876MHz)



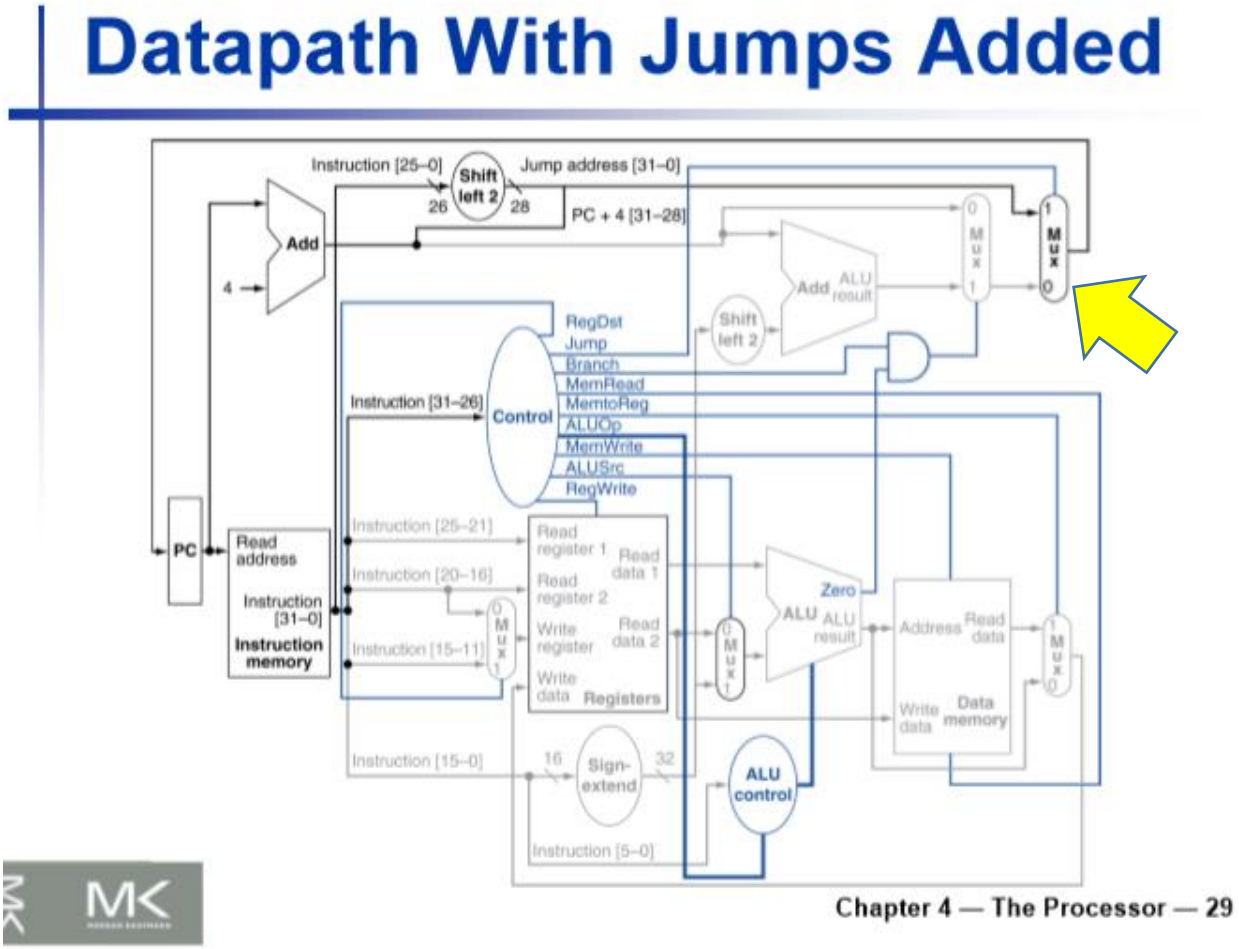
قسمت اختیاری پروژه :

در این قسمت به اضافه کردن 3 دستور jump,jr,jal به پردازنده میپردازیم.

1) jump

در این دستور هدف پرش از یک خط در حال اجرا به یک خط دیگر و اجرای بقیه ی دستورات از ان به بعد است در حقیقت میخواهیم مقدار pc را عوض کنیم.

طبق فصل 4 کتاب ما باید سخت افزار را به شکل زیر درست کنیم :



همانطور که در شکل مشخص است باید یک ماکس دیگر به مدار اضافه شود چون که باید تصمیم گرفته شود مقدار pc به چه شکل اپدیت شود به صورتی که قبلا داشتیم به با دستور جدید jump که الان اضافه شده؟ بنابراین باید دو ورودی و یک خروجی داشته باشد و همینطور برای پایه ی سلکت ان هم یک سیگنال کنترلی جدید باید به قسمت کنترل مدار اضافه شود.

همچنین من برای این دستور جدید عدد 110000 را در نظر گرفتم که این عدد همان 6بیت پر ارزش در دستور مد نظر است که با این 6 بیت است که ما متوجه میشویم به این دستور رسیدیم.

از طرفی طبق شکل یک ماژول شیفت به چپ میخواهیم که 26بیت باقی مانده ی دستور که داریم را شیفت بدهد و اینکار مثل این است که 2 بیت سمت چپ ان بگذارد و این عدد 26 بیتی را به یک عدد 28 بیتی تبدیل کند.

سپس در نهیت 4 بیت پرارزش pc را که در حالت عادی یکی به ان اضافه شده است را با 28 بیت قبلی concat میکنیم و این گونه الان یک ادرس 32 بیتی کامل داریم که میتواند جایگزین pc قبلی شود و دستور پرش اجرا شود.

حال یک مثال را در پردازنده ی جدید اجرا میکنیم :

فرض کنید دستور

32'b110000_00000_00000_0000000000000111

را در خانه ی صفر instruction memory لود میکنیم 6 بیت اول هماطور که گفته شد 110000 نشاندهنده ی دستور jump است و بقیه هم برای پرش استفاده میشوند

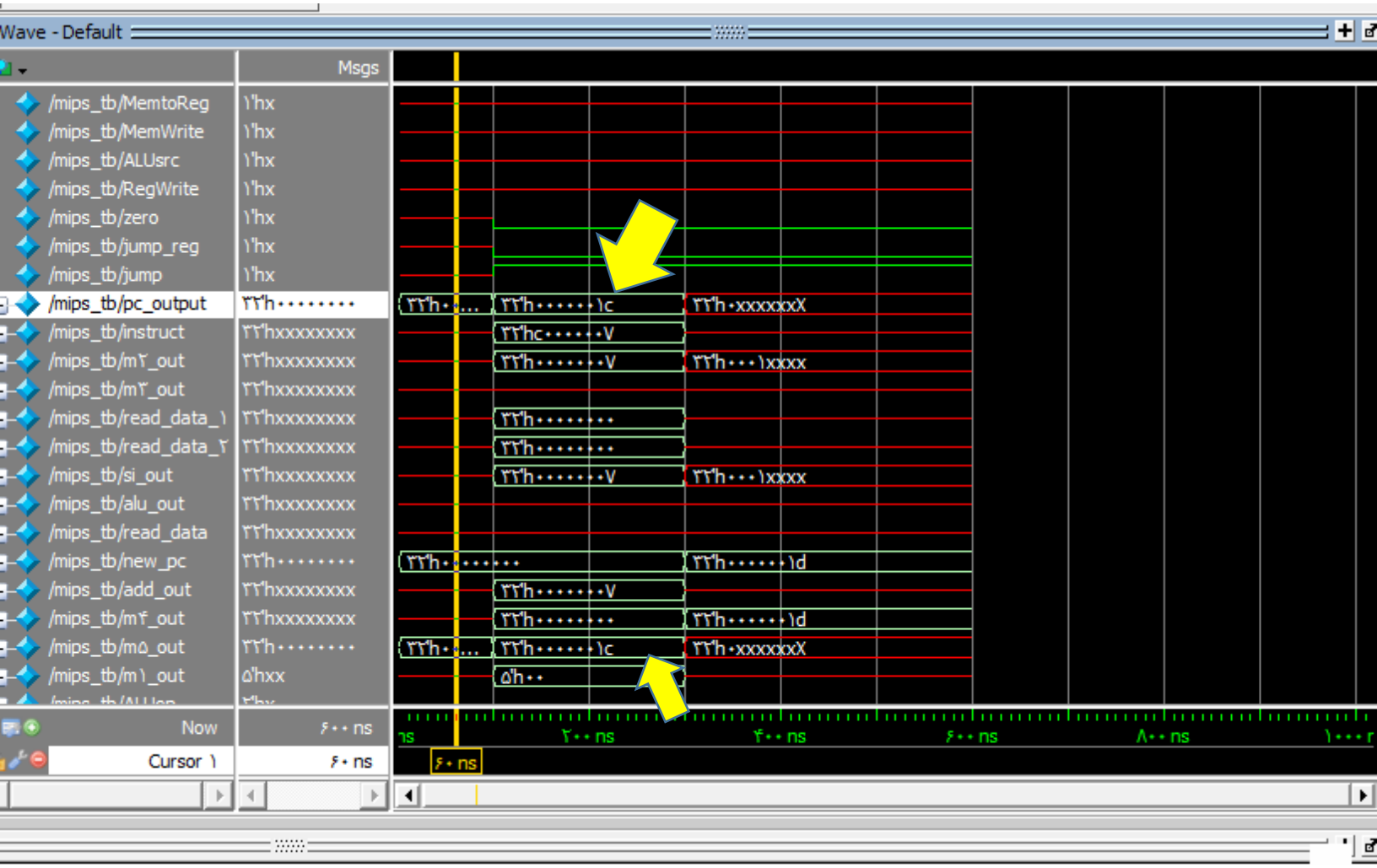
حال مراحل زیر طی میشود:

Pc = pc + 1 = 00000001(hex)

Shift(addr) = 0000001c

Concat(pc[31:27],addr) = 0000001c

پس نهایتا باید مقدار نهایی پی سی 0000001c میشود که تصویر زیر هم همین را نشان میدهد.



در شکل بالا خروجی ماکس 5 نیز که جدیدا اضافه شده مشخص است که عدد مد نظر شده چون دستور جامپ است و سیگنال ماکس مسیر جدید را انتخاب میکند.

مقدار جدید pc نیز مشخص است که همانطور که میبینید با مقدار جدید اپدیت شده.

دستور jr(jump register)

این دستور نیز یک نوع پرش است که به این ترتیب عمل میکند که ادرس یک رجیستر را میگیرد و به ادرسی که در داخل ان رجیستر نوشته شده است پرش میکند و در حقیقت pc را با ان مقدار جدید اپدیت میکند.

من برای این دستور کد 110000 را در نظر گرفتم که با این عدد شناسایی میشود.

برای پیاده سازی این دستور من ماکس جدیدی را که در قسمت قبل اضافه کردیم و 2به 1 بود را به یک ماکس 3 به 1 تبدیل کردم به این ترتیب به جز دو حالت قبلی که بحث شد یک مسیر جدید ایجاد شده که برای دستور jr است و pc ازین طریق نیز میتواند اپدیت شود .

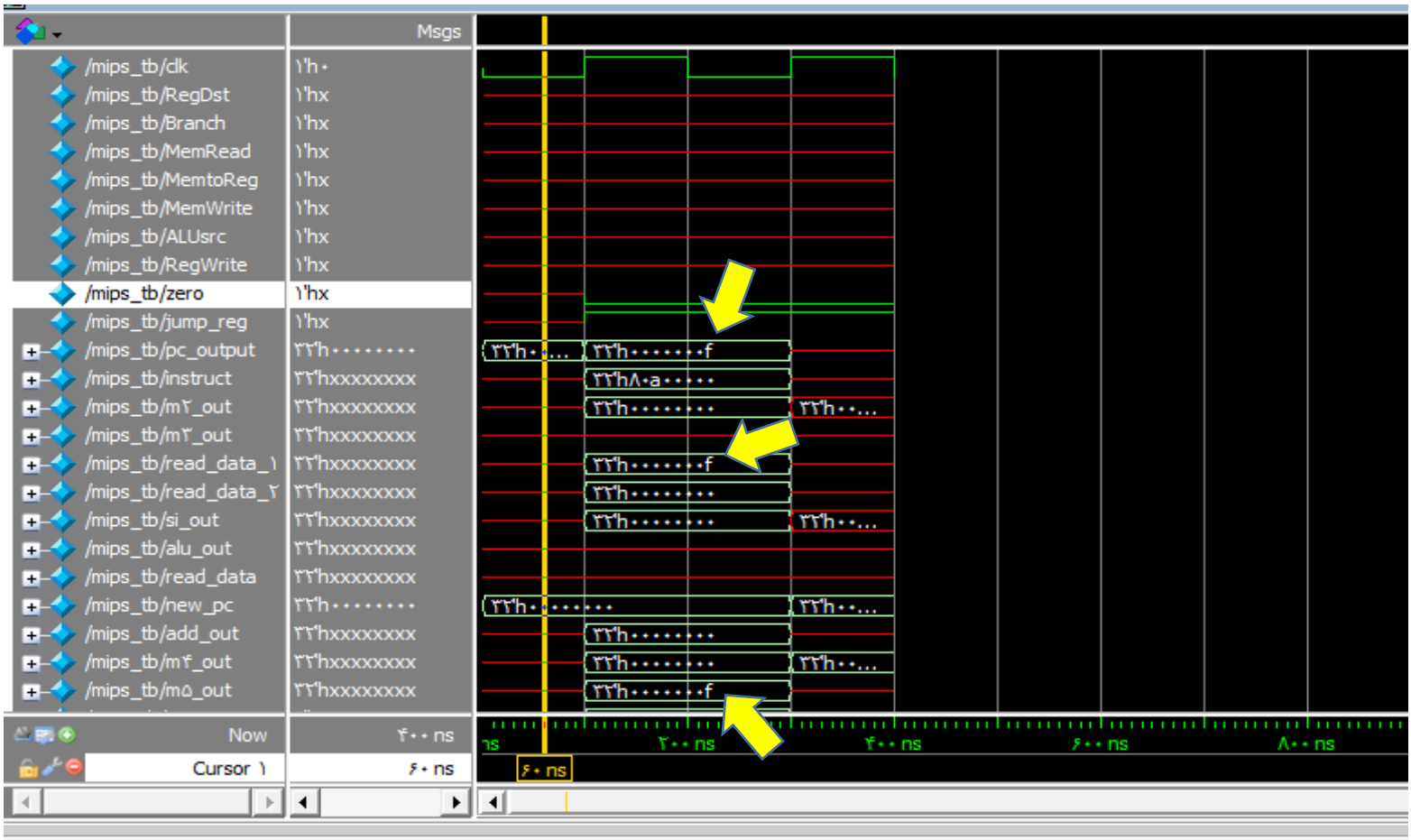
از طرفی چون این ماکس 3 ورودی دارد نیز به 2 پایه ی select دارد که این دو سیگنال عبارتند از : jump,jump_reg که با این دو سیگنال دو مسیر جدید مشخص میشود و اگر هر دو صفر باشند که مسیر همان مسیر عادی گذشته است. باید توجه داشت سیگتال جدیدی که برای این دستور اضافه شده نمیتواند همزمان با سیگنال jump یک باشد و ما باید به این موضوع دقت کنیم و هر وقت یکی 1 شد دیگری را 0 کنیم.

حال یک مثال میزنم :

فرض کنید عدد

32'b100000_00101_00000_0000000000000000

در instruction memory نوشته است این یعنی دستور مد نظر ما jump_reg است و ادرس رجیستری که در ان ادرس جدید امده است 0101 است که یعنی رجیستر 5 و من از قبل در رجیستر 5 مقدار 15 را نوشته ام پس توقع داریم ادرس جدید pc مقدار 15 را داشته باشد.



همانطور هم که مشخص است خروجی رجیستر بانک مقدار f است که همانطور است که انتظار داشتیم در نهایت خروجی ماکس 5 را میبینید که انهم مقدار 15 شده است و این یعنی مسیر جدیدی که اضافه کردیم انتخاب شده است و این مقدار داخل pc ریخته میشود و ما پرش را انجام میدهیم.

Jal (jump and link) :

این دستور هم نوع دیگری از پرش است که بسیار شبیه به دستور jump است فقط تفاوت ان این است که چون این دستور برای فراخوانی توابع به کار میرود ما بعد از پرش به تابع باید بعد اتمام ان به محل قبلی بازگردیم و به همین دلیل باید هنگام پرش ادرس محلی که داریم را یک جا ذخیره کنیم تا همنگام بازگشت بتوانیم به ان بازگردیم من این محل را رجیستر 31 در نظر گرفتم یعنی ادرس بازگشت به طور خودکار در اینجا ذخیره میشود.

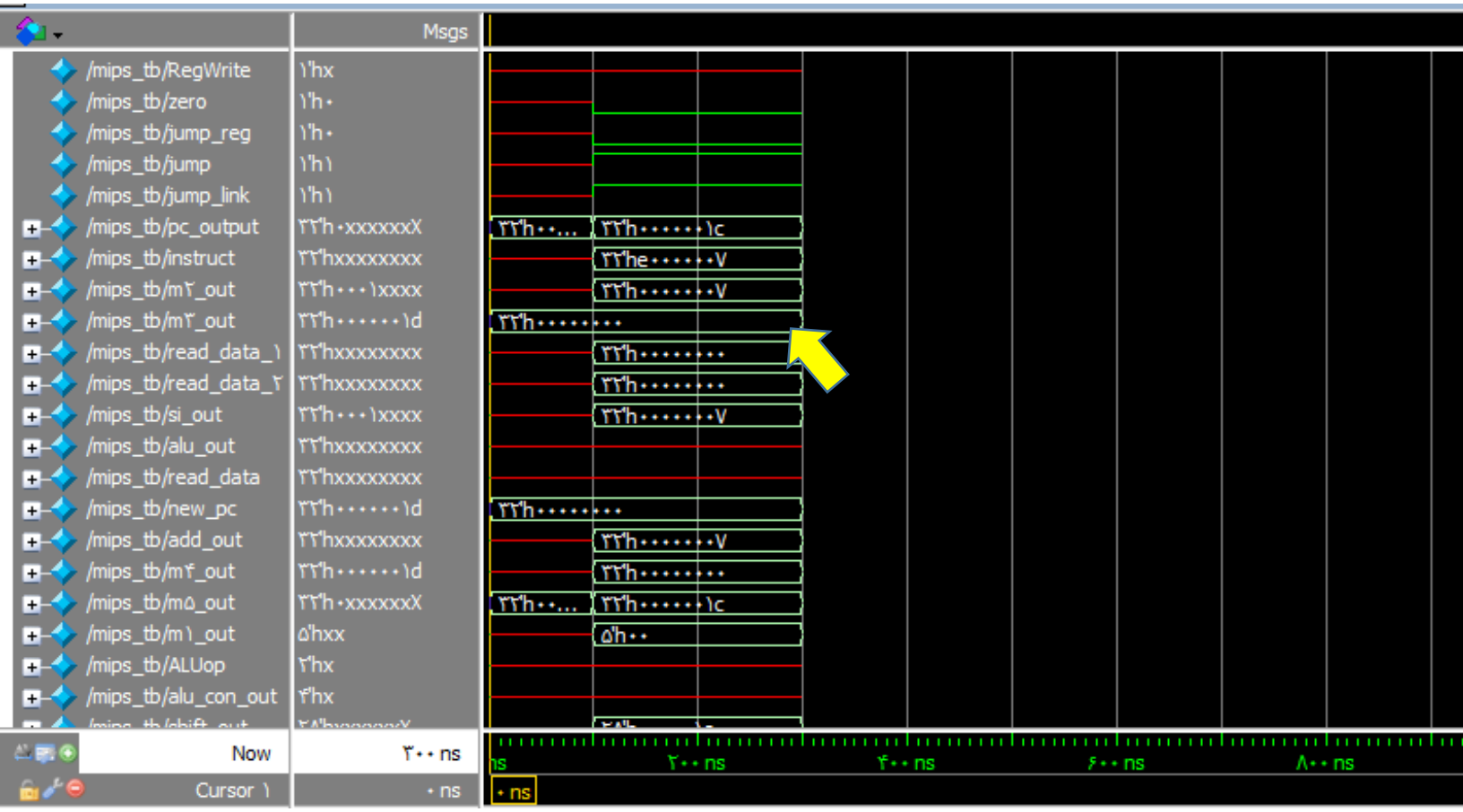
کد این دستو را هم 111000 در نظر گرفتم و همچنین برای پیاده سازی باید ماکس 3 که قبلا 2 ورودی بود را به یک ماکس 3 ورودی تبدیل کنیم تا در موقع رسیدن به این دستور از مسیر جدید بتوانیم ادرس را مستقیما در رجیستر بانک بنویسیم این ماکس به دو ورودی سلکت نیزنیاز دارد که انها memtoreg,jump_link هستند که یکی ازین ها تازه به واحد کنترل اضافه شده.

حال برای مثال فرض کنید که دستور زیر را وارد پردازنده کردیم :

32'b111000_00000_00000_0000000000000111

6 بیت اول که نشانه ی این دستور است و بقیه ی ان نشاندهده ی ادرس مقصد است.

این کد تماما مثل دستوری است که برای jump مثال زدم و خروجی ها هم مثل همان است فقط قرار است ادرس فعلی pc نیز ذخیره شود که 0 است(چون این دستور در خاانه ی 0 نوشته شده است) پس باید خروجی ماکس 3 را ببینیم.



همانطور که مشخص است ماکس 3 خروجی تمام 0 داده است که نشاندهنده ی ادرس فعلی pc است که قرار است با کلاک بعدی در رجیستر بانک ذخیره شود.

سایر چیز ها هم مثل قبل میباشد.

گزارش جدید سنتز :

Primitive and Black Box Usage:

# BELS		: 770
#	GND	: 1
#	INV	: 2
#	LUT1	: 27
#	LUT2	: 38
#	LUT3	: 109
#	LUT4	: 148
#	LUT5	: 37
#	LUT6	: 110

```
#      MUXCY                 : 175
#      VCC                   : 1
#      XORCY                 : 128
# FlipFlops/Latches          : 159
#      FD                    : 42
#      FDE                   : 32
#      FDR                   : 1
#      LD                    : 83
#      LDCE                  6
#      LDPE                  : 2
# RAMS                       : 3
#      RAMB8BWER             : 3
# Clock Buffers              : 2
#      BUFG                  : 1
#      BUFGP                 : 1
# IO Buffers                 : 403
#      OBUF                  : 403
```

Slice Logic Utilization:

Number of Slice Registers: 126 out of 18224 0%

Number of Slice LUTs: 466 out of 9112 4%

Number used as Logic: 476 out of 9112 4%

Slice Logic Distribution:

Number of LUT Flip Flop pairs used: 447

Number with an unused Flip Flop: 343 out of 447 76%

Number with an unused LUT: 1 out of 447 0%

Number of fully used LUT-FF pairs: 122 out of 447 28%

Number of unique control sets: 9

IO Utilization:

Number of IOs: 412

Number of bonded IOBs: 404 out of 232 174% (*)

IOB Flip Flops/Latches: 43

Specific Feature Utilization:

Number of Block RAM/FIFO: 2 out of 40 12%

Number using Block RAM only: 2

Number of BUFG/BUFGCTRLs: 2 out of 23 21%

Maximum Frequency: 113.32MHz