

# CAD (Computer Aided Design)

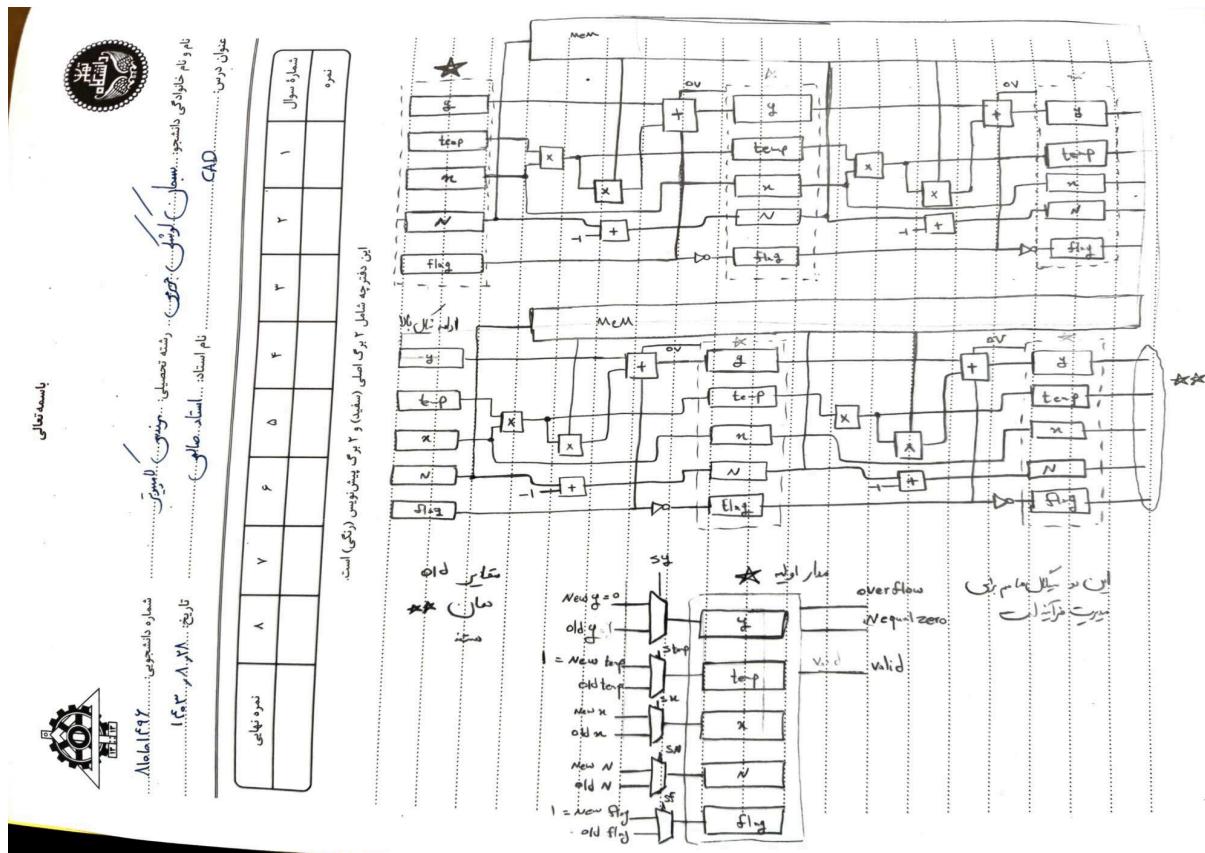
## Midterm

Sobhan Kooshki Jahromi 810101496

در ابتدا طرحی که در میانترم نوشته ام را به شما نشان میدهم.

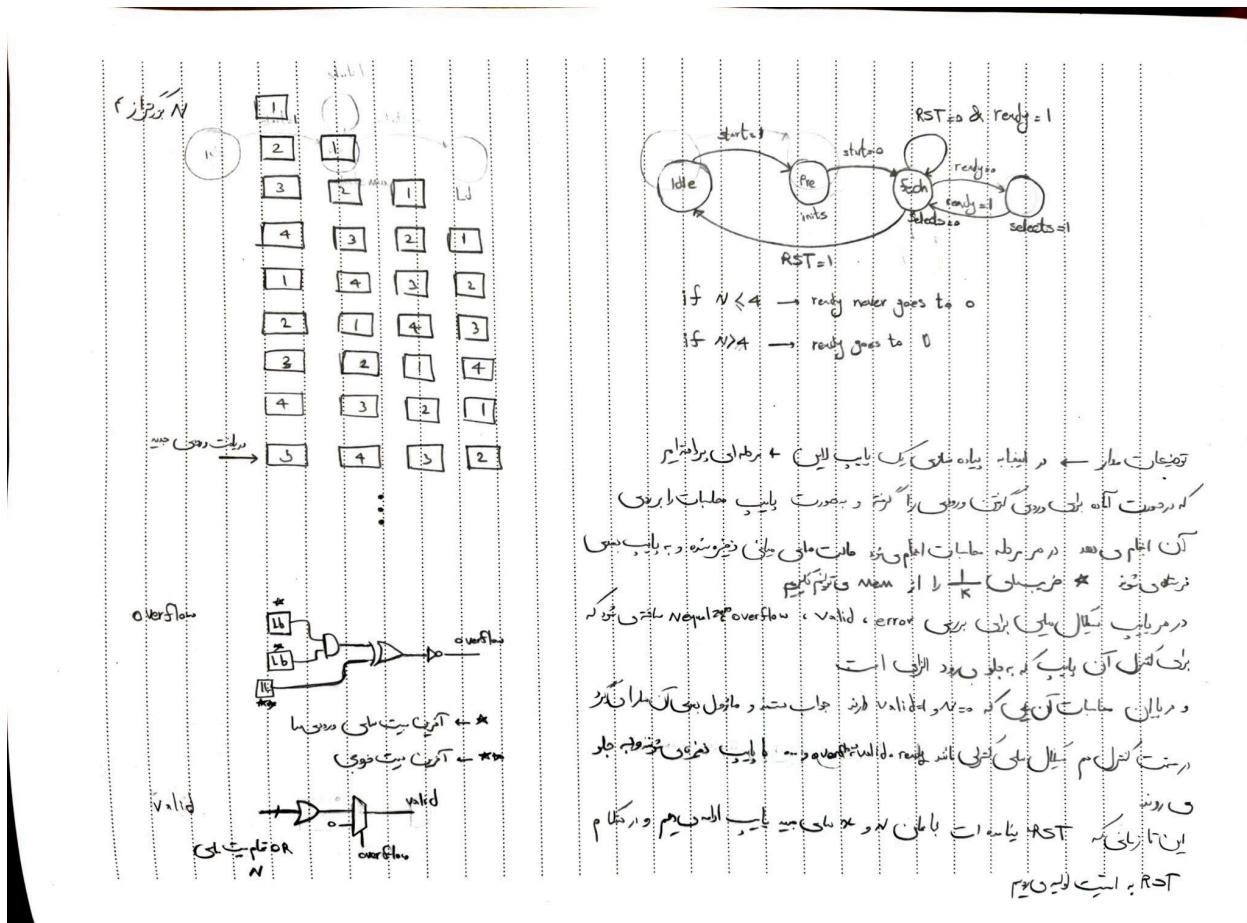
در این طرح اولیه من با استفاده از ایده کلی پایپ لاین در ابتدا 4 پایپ و 4 رجیستر لاین سعی بر ساختن ساختار اصلی پایپ هایم داشته باشم. سپس طرح درون هر پایپ را پیاده سازی کردم.

هر پایپ پس از اینکه تمام میشود به یک سری mux رفته اند و ان mux با استفاده از یک sel بین مقدار قبلی و مقدار جدید انتخاب میکند و در رجیستر های هر پایپ ذخیره میشوند. در این طرح از یک flag استفاده شده که مقدار منفی های ان ضریب های ثابت را مشخص میکند این کار برای محاسبه منفی ها به جمع کننده رفته و با  $flag = 0$  ما جمع را انجام میدهیم و بر عکس با  $flag = 1$  ما منفی را داریم.

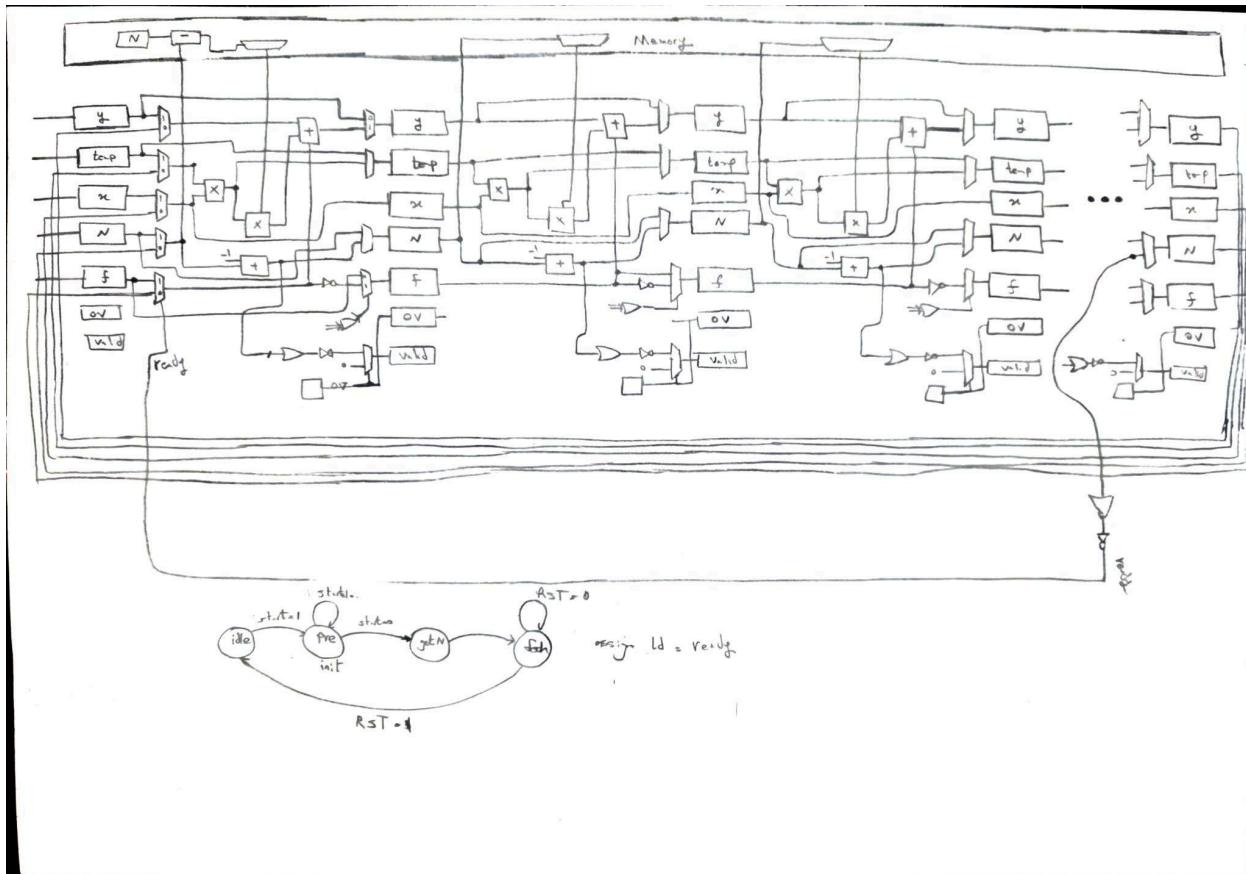


این عکس هم flow کلی و کنترلر این طرح را میبینیم که در این کنترلر من به اشتباه فکر کرده ام که  $n$  و  $x$  را ما با هم میگیریم در نتیجه استینی برای گرفتن  $n$  در نظر نگرفته ام. این controller که در امتحان میانterm نوشته ام به صورت کلی بوده و طرح های بعدی درست شده است.

سیگنال های overflow و valid را به شکل های کشیده شده ساخته بودیم ولی این طرح را با اشتباه در امتحان نوشته بودم و در ادامه صحیح آن طراحی کرده ام. توضیح هایی هم راجب این طرح و کنترلر که در میانterm نوشته ام را هم میبینید.



## طرح میانی :



در این طرح اولیه بعد از امتحان به جزییات بیشتری از datapath پرداخته ام و controller را تصحیح کرده ام.

## : Datapath

در اینجا تغییراتی که مشاهده میکنید را میگوییم.

★ جزییات مربوط به گرفتن ضریب ثابت از memory را میبینیم که در اینجا ما  $n$  اولیه را میگیریم و از  $n$  که ره پایپ وجود دارد کم کرده در این صورت میتوانیم  $index$  مموری را که میخواهیم بدست اوریم حالا کافی است که ضریب ان  $memory$  را از  $memory$  بخوانیم.

★ اضافه کردن رجیستر های **overflow** و **valid** به رجیستر میانی به جای سیگنال بودن ان ها در طرح امتحان میانترم ، این کار باعث میشود که ما در هر **mux** مقدار قبلی **valid** و **overflow** را داشته باشیم و برای **select pipe** ها استفاده کنیم.

★ درست کردن لاجیک سیگنال های **valid** و **overflow** ، در این قسمت به جای **or** بیت های **N** ما باید **nor** انجام بدھیم تا در صورت 0 بودن همه بیت ها ما یک را بگیریم نه صفر.

★ گرفتن سیگنال **valid** از پایپ اخر که ما این سیگنال را از  $n-1$  پایپ اخر میگیریم اگر تمام بیت های  $n-1$  را با هم **nor** کنیم به ما این لاجیک را میدهد اگر تمام بیت های  $n-1$  برابر 0 باشد یعنی این پایپ اخر این **X** است و ما میتوانیم ورودی جدید را وارد کنیم. اگر که این مقدار مخالف 0 بود یعنی که این **X** که الان در پایپ اخر است باید یک دور دیگه به پایپ یک وارد شود و محاسبات ان ادامه پیدا کند پس ما این سیگنال **ready** را که ساخته ایم به **mux** های اول کار داده ایم که بر اساس این سیگنال کار را انجام میدهد و اجازه ورود داده جدید بر اساس این **ready** تعیین میشود.

★ تعیین **mux** های میانی بعد از هر **pipe** با استفاده از **valid** و **overflow** رجیستر **pipe** قبلی که ذخیره شده ان اگر ما این دو سیگنال را با هم **xor** کنیم لاجیکی که به ما میدهد این است که اگر هرکدام از این سیگنال ها یک شده باشند نتیجه نهایی ما 0 میشود که این باعث میشود که ورودی 0 **mux** ها به روی خروجی روند و این یعنی که ما از روی هر **pipe** گذشته ایم. ( اصلا چرا اینکار را میخواهیم ؟ ما وقتی که نتیجه را میگیریم و یکی از دو حالت **valid** و **overflow** را داریم یعنی که کار **X** تمام شده است و دیگر نباید ان را وارد پایپ های بعدی کنیم که محاسبات را بر روی ان انجام دهیم پس باید وقتی که کارمان برای هر **X** تمام میشود از تمام **pipe** های روبروی ان پرسش کنیم.

## : Controller

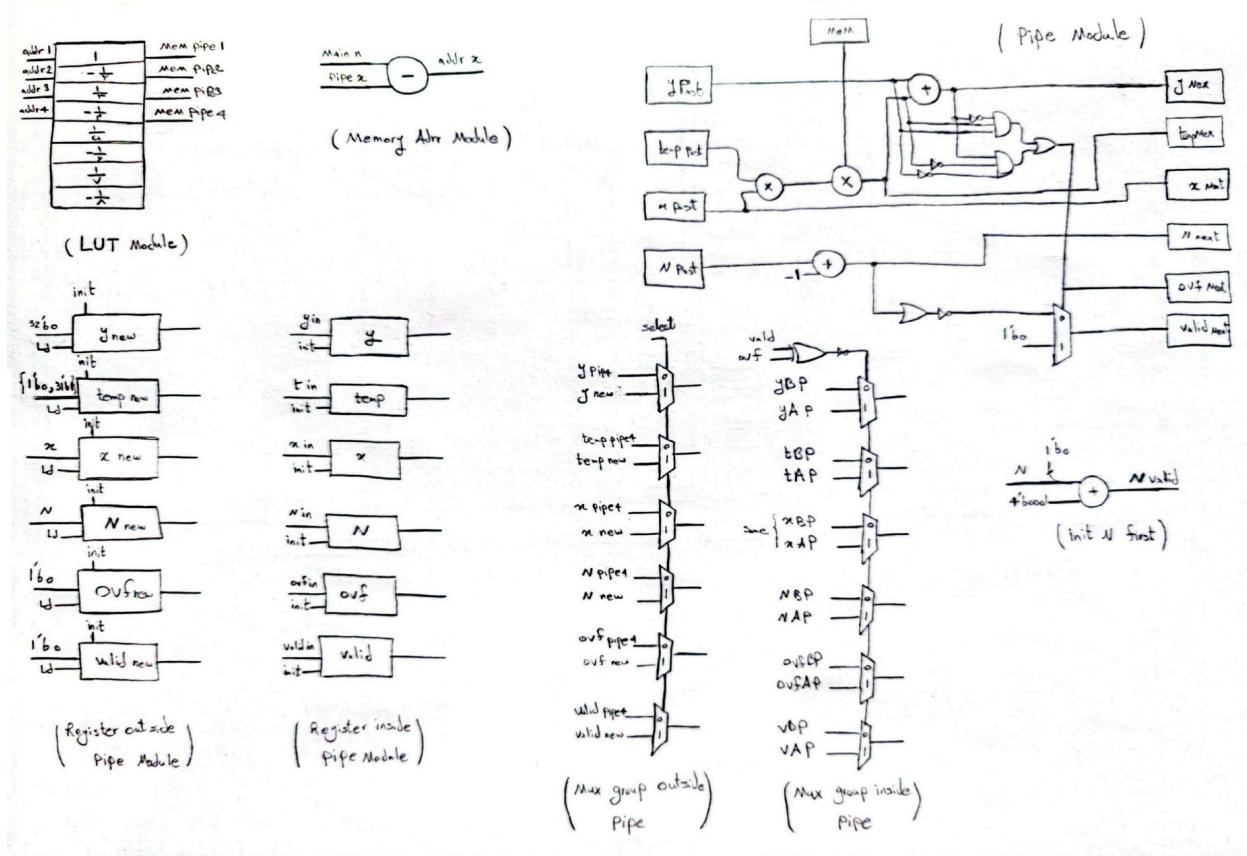
تغییرات کنترلر به صورت زیر است

★ به دلیل اینکه تقریبا تمام سیگنال های کنترلی را در خود **datapath** هندل کردیم پس الان دیگر کار سختی نداریم فقط **Id** های رجیستر های ورودی را با این **ready**

هندل میکنیم که اگر پایپ جای خالی داشت داده ها را وارد رجیستر های اولیه کنیم  
که به پایپ وارد شوند.

★ دیگر نیاز به ان استیت اخر که در امتحان نوشته بوده ام نداریم و به همان استیت  
کارمان را خاتمه میدهیم.

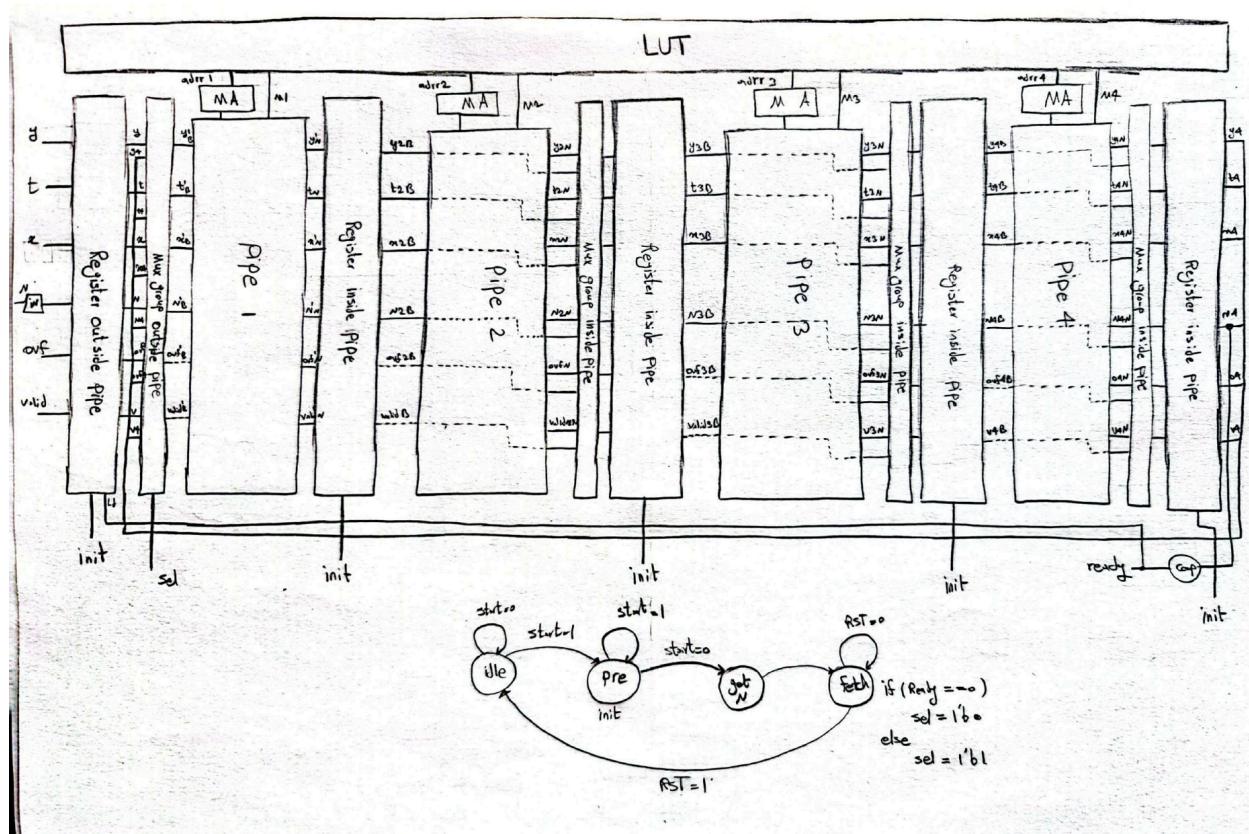
## طرح پایانی :



طرح خودم رو به چندین مازول بالا که میبینید پیاده سازی میکنم پس میتوانیم دیدمان را به  
یک پله کلی تر ببریم که در عکس بعد مشاهده میکنید.  
به ترتیب مازول ها را معرفی میکنم.

- LUT : این مازول همان حافظه ضریب های ثابت ما است که برای گرفتن هر  
ضریب pipe ها به ان یک **addr index** میدهند و یک خروجی را میگیرند.
- Memory addr : این مازول برای ساخت همان **index** است که در طرح میانی  
به ان اشاره کردم.

- در مازول پایپ ما محاسبات مربوط به  $y$ ,  $temp$ ,  $x$ ,  $n$ ,  $valid$  Pipe overflow را انجام میدهیم و مقادیر جدید این سیگنال ها را میسازیم.
- این رجیستر های اولیه است که مقادیر جدید یک  $x$  که میخواهد وارد پایپ شود را نگه میدارد و سپس به درون پایپ میفرستد.
- این رجیستر های در میان پایپ ها است که مقادیر میانی را در خود نگه میدارد.
- Mux group outside pipe : این mux ها برای انتخاب ورودی جدید یا خروجی پایپ چهارم برای یک دور بیشتر pipe ها استفاده میشود.
- Mux group inside pipe : این mux ها برای انتخاب رد کردن خروجی های جدید که از pipe ها می آید یا مقادیر سیگنال ها قبل از ورود به pipe است.
- Init N : این مازول برای ساخت یک  $n$  معتبر در ابتدا است چون که  $n$  ورودی برای این pipe سه بیت است ما در ابتدا این مقدار را  $1+1$  میکنیم که مقدار معتبری از  $n$  را داشته باشیم.



طرح کلی را به صورت کنار هم قرار دادن مازول های گفته شده در بالا میتوانیم به این صورت ببینیم.

تغییرات ان نسبت به طرح میانی به صورت زیر میباشد.

★ ما نیاز به **mux group inside pipe** بعد از 1 **pipe** نداریم و همیشه این **pipe** یک در حال کار کردن است پس دیگر نیاز به این **mux group** نداریم و از طرح کلی ان را برداشته ایم.

★ بعد از یکسری باگ ها که وجود داشت ما مجبور شدیم که سیگنال **ready** را از بعد از خروجی **n register** بعد از 4 **pipe** بگیریم و با استفاده از یک **comparator** یک سری مقادیر را که به دلیل **initial** کردن طرح در اول کار این مدار است را مقایسه کنیم و سپس مقدار سیگنال **ready** را بسازیم یک حالت دیگری هم وجود دارد که اگر ان ورودی **X** به پایپ چهارم رسیده و **overflow** شده است ان **X** را ببیرون ریخته و اجازه میدهیم که ورودی بگیریم.

★ چون ورودی **n** سه بیت است ان را ابتدا تبدیل به 4 بیت میکنیم سپس به اضافه یک میکنیم تا همان دقت های 1 تا 8 ساخته شود.

★ در کنترلر دیگر به جای اینکه **sel** گروه **mux outside pipe** را از همان کنترل کنیم به درون **controller** برده و در **fetch** با یک شرط ان **sel** را مقداردهی میکنیم. اینکار باعث میشود که فقط هنگامی که در **fetch** هستیم ان مقدار **sel** مقداردهی به یک شود.

## نکاتی راجع به کد:

برای اینکه ما در مدارمان ضرب های و جمع های اعداد مثبت در منفی را داریم پس باید ابتدا این را در نظر بگیریم که اعداد منفی را به صورت مکمل دو (two's comp.) به مدار بدهیم و یک نکته دیگر این است که ورودی های **multiplier** و **adder** را **signed** باید در نظر بگیریم.

```

module adder #(parameter size) (inp1, inp2, out);
    input signed[size-1:0] inp1, inp2;
    output signed[size-1:0] out;
    assign out = inp1 + inp2;
endmodule

```

```

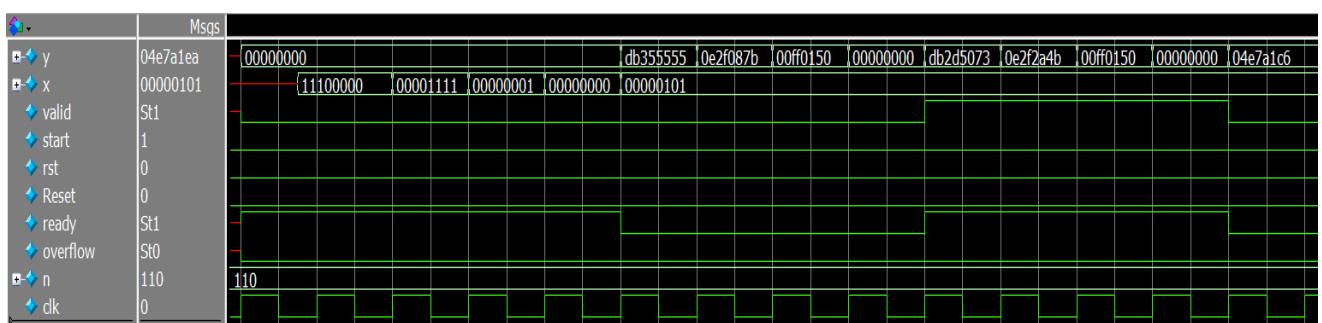
module mult #(parameter size)(inp1, inp2, mult_output);
    input signed[size-1:0] inp1, inp2;
    output signed [size-1:0] mult_output;
    wire signed [(2*size - 1):0] temp;

    assign temp = inp1 * inp2;
    assign mult_output = temp[((2*size - 1)-1):(size-1)];
endmodule

```

در قسمت ضرب کننده ما چون که ورودی های 32 بیت به ان میدهیم یک خروجی 64 بیت از ضرب میگیریم ولی باید 32 بیت ان را برداریم. در این قسمت باید یک تریک بزنید و مقدار 62:31 را برداریم ان بیت اخر به دلیل ضرب دو بیت sign در هم یک بیت extend میشود و برای ما مطلوب نیست پس ما بیت اخر را drop میکنیم و 32 بعد از ان را به خروجی میدهیم.

## بعضی از خروجی های تست:



در اینجا چندین ورودی و خروجی را میبینیم که به ازای دقت 7 یعنی  $2^7 = 6$  بدهست می‌اوریم.

یک نکته وجود دارد برای نتیجه اعداد منفی به باینری تبدیل میکنیم یک اخر را صفر میکنیم بعد یک dot بعد از بیت اخر گذاشته و نتیجه دسیمال را منهای 1 میکنیم و جواب ما بدهست میاید.

برای اعداد مثبت هم مشکلی دیگر نداریم و صرفا همین جواب را به دسیمال تبدیل میکنیم.