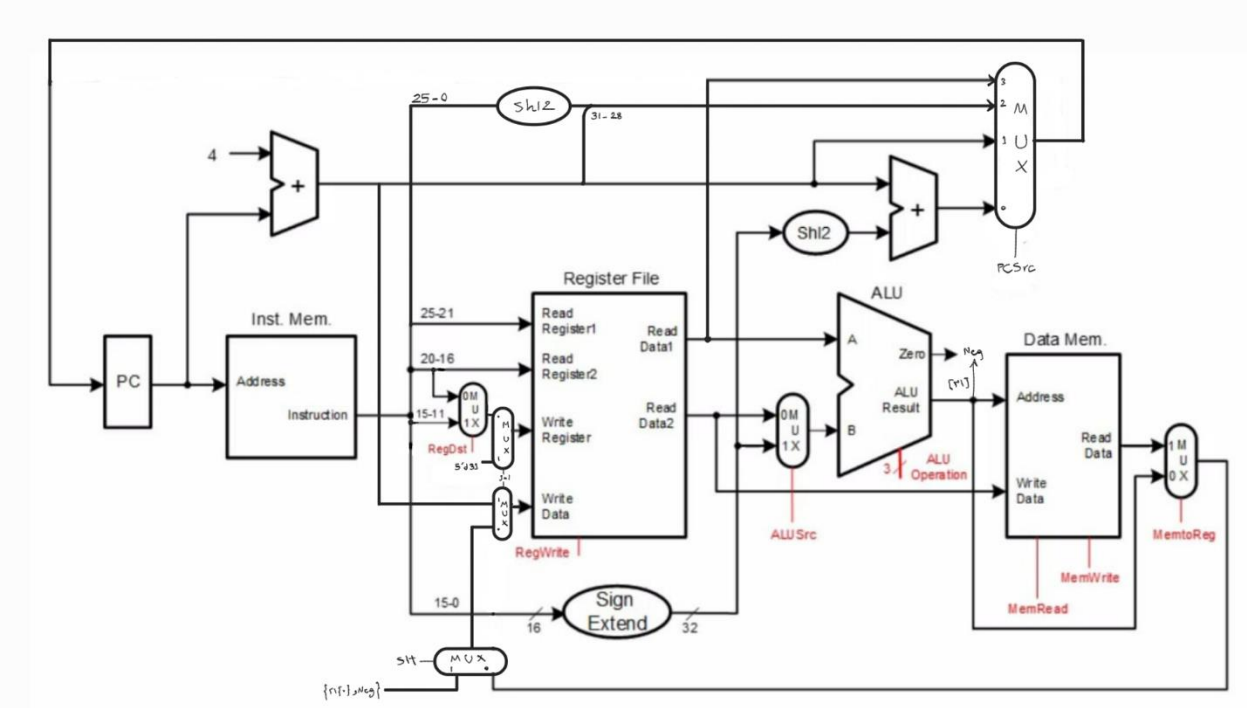
پویا صادقی – 810199447

مهدی چراغی – 810199399

گزارش تمرین کامپیوتری دوم معماری کامپیوتر



طراحی اولیه مسیر داده

مطالب قابل توجه در مسیر داده شده در بالا:

1. مولتی پلکسر های متصل به سیگنال Jal: این سیگنال فقط برای همین دستور فعال میشود و دو مولتی پلکسر موجود نیز برای پیش بردن این دستور قرار گرفته اند.
2. مولتی پلکسر متصل به سیگنال slt: در صورت دستور slt فعال میشود. همچنین مقدار تولیدی توسط این درستور که در یک رجیستر نوشته میشود، بصورت ترتیبی و در دیتاپث ساخته میشود – {31{1’b0},neg} که سیگنال neg از مقدار alu بصورت wiring(بیت اخر) صادر میشود و عملیات تفریق در alu تنها ابزار مقایسه ای ما میباشد.
3. مولتی پلکسر مربوط به ورودی PC: ورودی 1 آن برای پیشبرد عادی دستور میباشد. دستور هایی که منجر به پرش میشوند نیز توسط این مولتی پلکسر انتخاب میشوند.

Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R-format	op	rs	rt	rd	shamt	funct
I-format	op	rs	rt	address/immediate		
J-format	op	target address				

فرمت دستورات ورودی برحسب نوع دستور

	RegDst	Jal	RegWrite	slt	ALUsrc	ALUop	PCsrc	MemRead	MemWrite	MemToReg
add	1	0	1	0	0	add	1	0	0	0
sub	1	0	1	0	0	sub	1	0	0	0
and	1	0	1	0	0	and	1	0	0	0
or	1	0	1	0	0	or	1	0	0	0
slt	X	0	1	1	0	sub	1	0	0	0
jr	X	0	0	0	X	nothing	3	0	0	X
addi	0	0	1	0	1	push-add	1	0	0	0
slti	0	0	1	1	1	push-sub	1	0	0	0
lw	0	0	1	0	1	push-add	1	1	0	1
sw	X	0	0	0	1	push-add	1	0	1	0
j	X	0	0	0	X	nothing	2	0	0	X
jal	X	1	1	0	X	nothing	2	0	0	X
beq	X	0	0	0	0	push-sub	zero? 0:1	0	0	X
		R-type	I-type	J-type						

سیگنال های کنترلی مربوط به هر دستور(واحد کنترل)

دستورات پردازنده با جزئیات بیشتر

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three register operands
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three register operands
	add immediate	addi $s1,$s2,20	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	lw $s1,20($s2)	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	sw $s1,20($s2)	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	lh $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	lhu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	sh $s1,20($s2)	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	lb $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	lbu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	sb $s1,20($s2)	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	ll $s1,20($s2)	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	sc $s1,20($s2)	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	lui $s1,20	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,20	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,20	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq $s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne $s1,$s2,25	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set on less than unsigned	sltu $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	slti $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	sltiu $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 10000	For procedure call

نحوه عملکرد دستورات مختلف این پردازنده

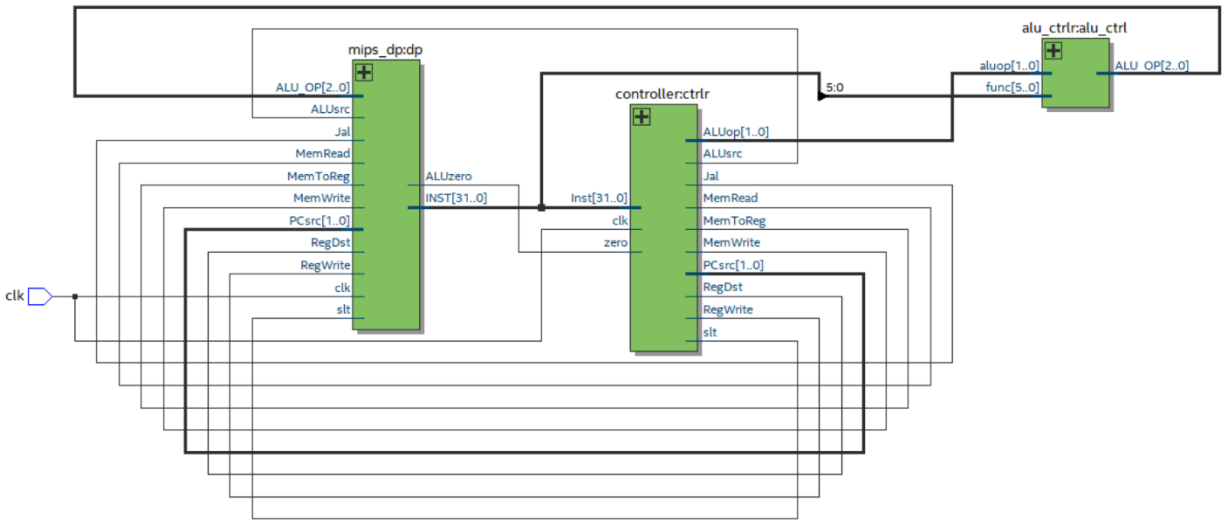
در باب کنترلر نیز توجه شود از آنجا که تمامی دستورات R-typr از opcode مشابه استفاده میکنند، پس از تشخیص این نوع دستور، باتوجه به func بطور دقیق عملیات را مشخص میکنیم. همچنین کنترلرهای اصلی و alu جداگانه طراحی شده اند که کنترلر اصلی بر دیگری تسلط دارد.

توجه شود که دستورات پردازنده الزاماً باید در فایل instructions.txt قرار داشته باشند. همچنین رجیستر فایل و دیتاموری به ترتیب میتوانند با فایل های registers.txt و data.txt مقدار دهی شوند که البته حضور آنها اجباری نمیباشد. فرمت txt نیز بدلیل آسانتر باز شدن فایل ها در کامپیوتر بصورت پیشفرض انتخاب شد.

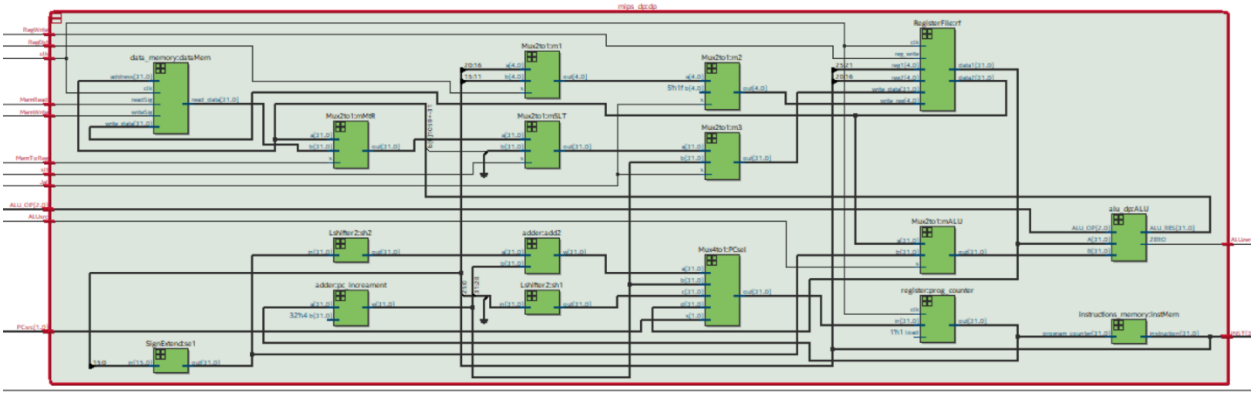
همچنین توجه شون که inst.memory , datamemory دستورات را بصورت مضربی از 4 میگیرند که چنین چیزی درباره ی reg.file صادق نمیباشد(البته دیتاموری اجباری از این جهت ندارد و از آدرس دهی شکسته پشتیبانی میکند).

یک فایل sample نیز برای دستورات ارائه میشود که برای تست کردن پاسخ دهی پردازنده استفاده شده است و در شرایط مرزی و پیچیده تر(در صورت وجود) آنرا آزموده است.

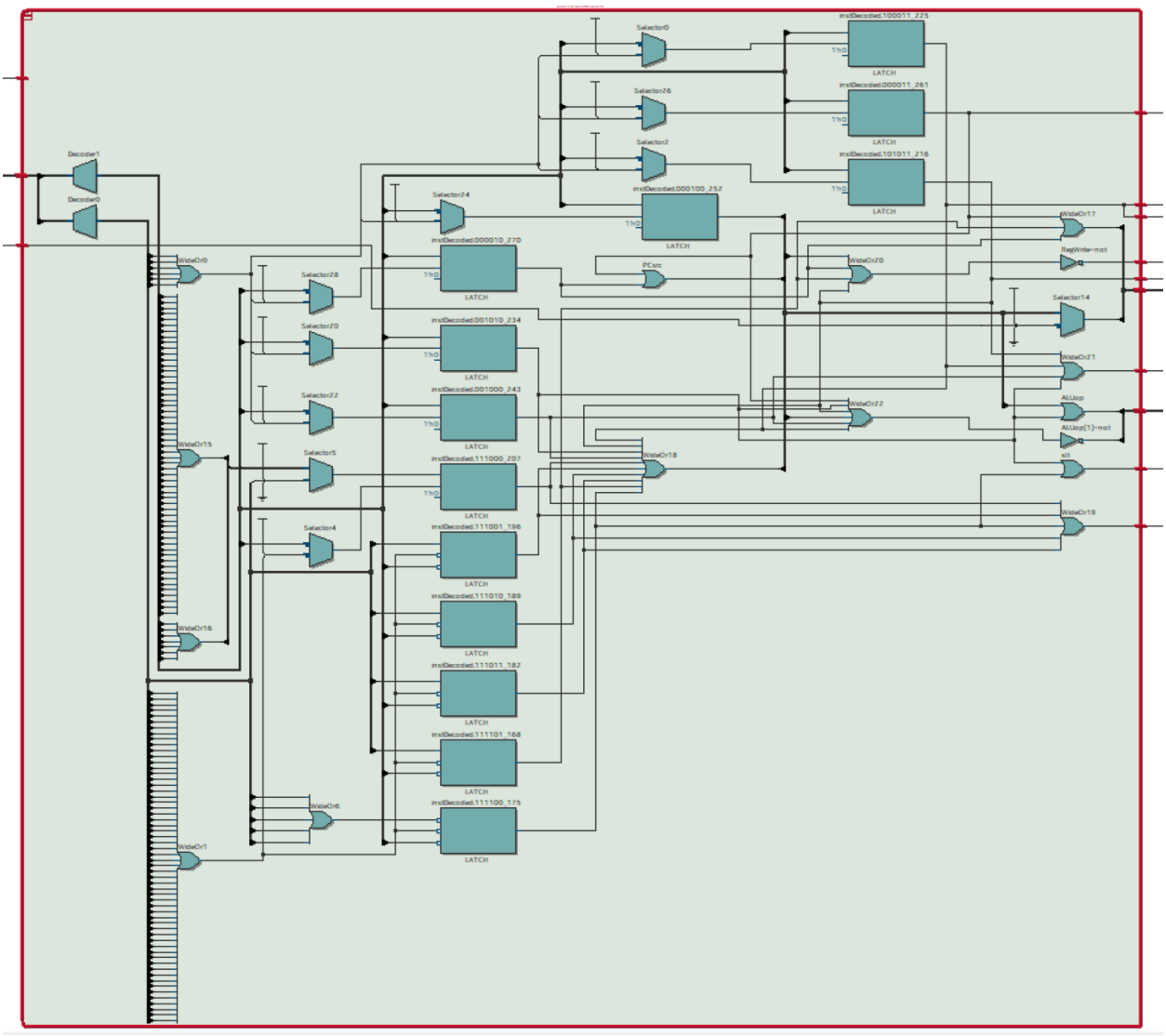
همچنین طراحی نهایی مسیر داده تفاوت ناچیزی با طراحی اولیه دارد(عملا تفاوتی حس نمیشود) اما این طراحی و طراحی های دقیق تر بخش های مختلف در ادامه آورده میشوند.



top-level



datapath



controller

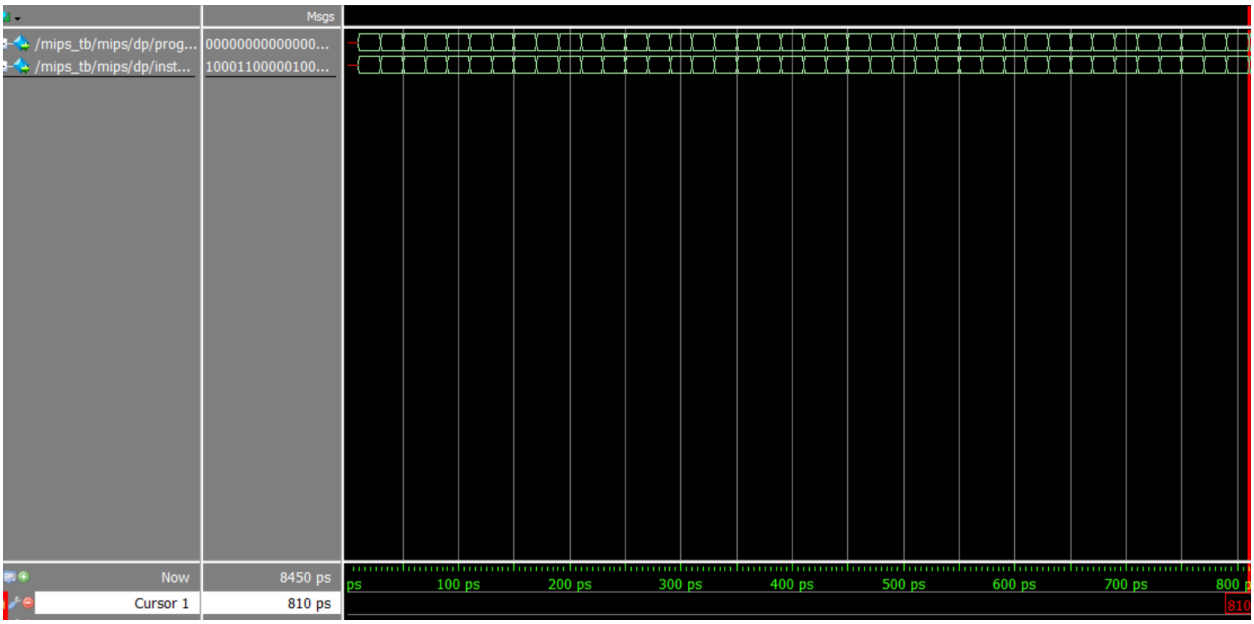
در نهایت نیز، نتایج تست بنچ صورت پروژه:

بعد از بارگذاری آرایه 20 تایی در حافظه مموری در آدرس های مربوطه، برنامه ای که پیشبرد دستورات را بر عهده دارد، کد زیر میباشد(که طبعاً بصورت اسمبلی نوشته شده):

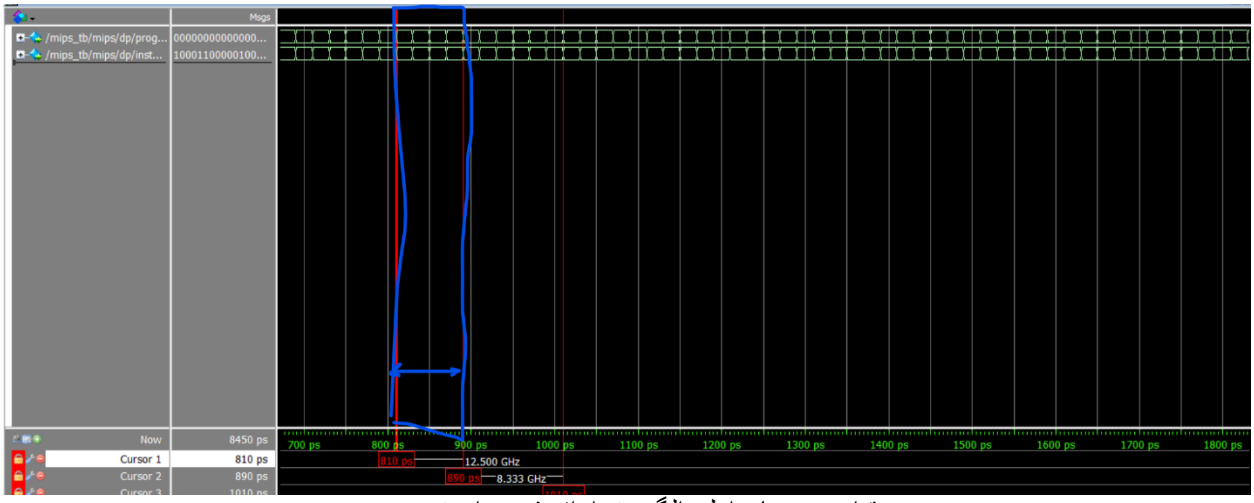
```
add R2,R0,R0
add R1,R0,R0
addi R8,R0,1000
add R3,R0,R0
Slti R7,R3,20          <-LOOP
beq R7,R0, END_LOOP
lw R4,0(R8)
slt R5,R4,R1
beq R5,R0,IF
J END_IF
add R1,R0,R4            <-IF
add R2,R0,R3
addi R8,R8,4            <-END_IF
addi R3,R3,+1
J LOOP
Sw R1,2000(R0)          <-END_LOOP
Sw R2,2004(R0)
```

کد فوق برای توضبح بهتر تابع استفاده شده در تست بنچ آورده شده و ارزش دیگری ندارد

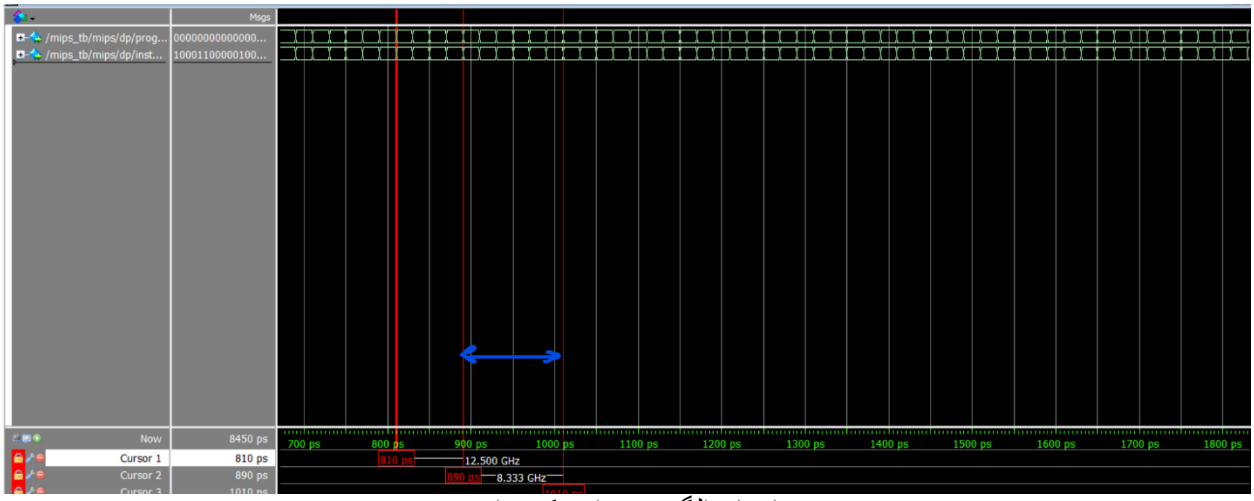
حال پس از 40 کلاک سایکل که مقادیر در مموری لود شدند:



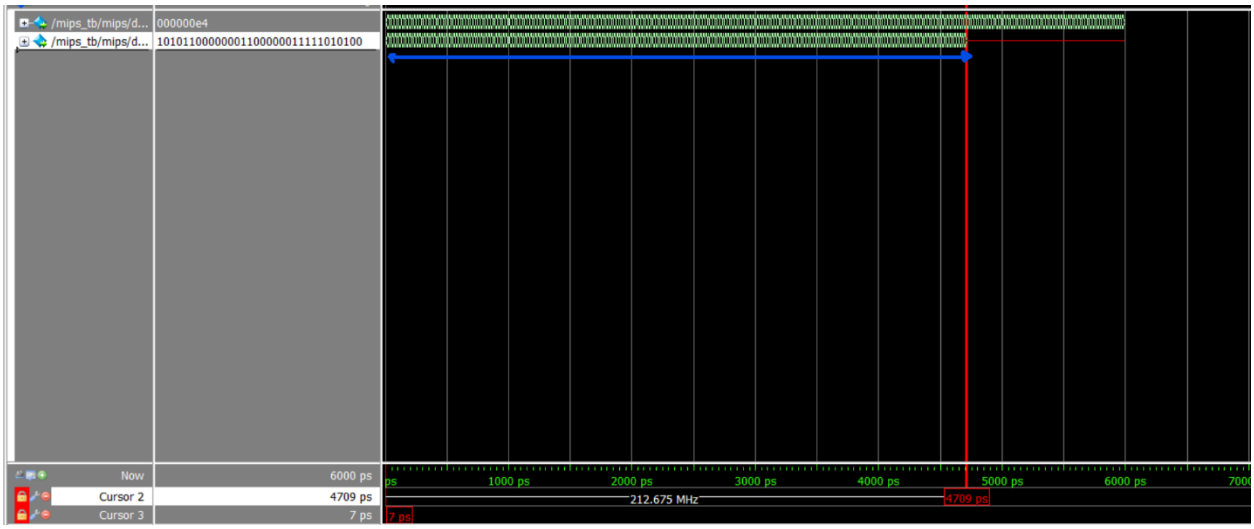
بازه لازم برای لود آرایه در موری با کلاک 20



مقدار دهی های اولیه الگوریتم ارائه شده برای تست بنچ

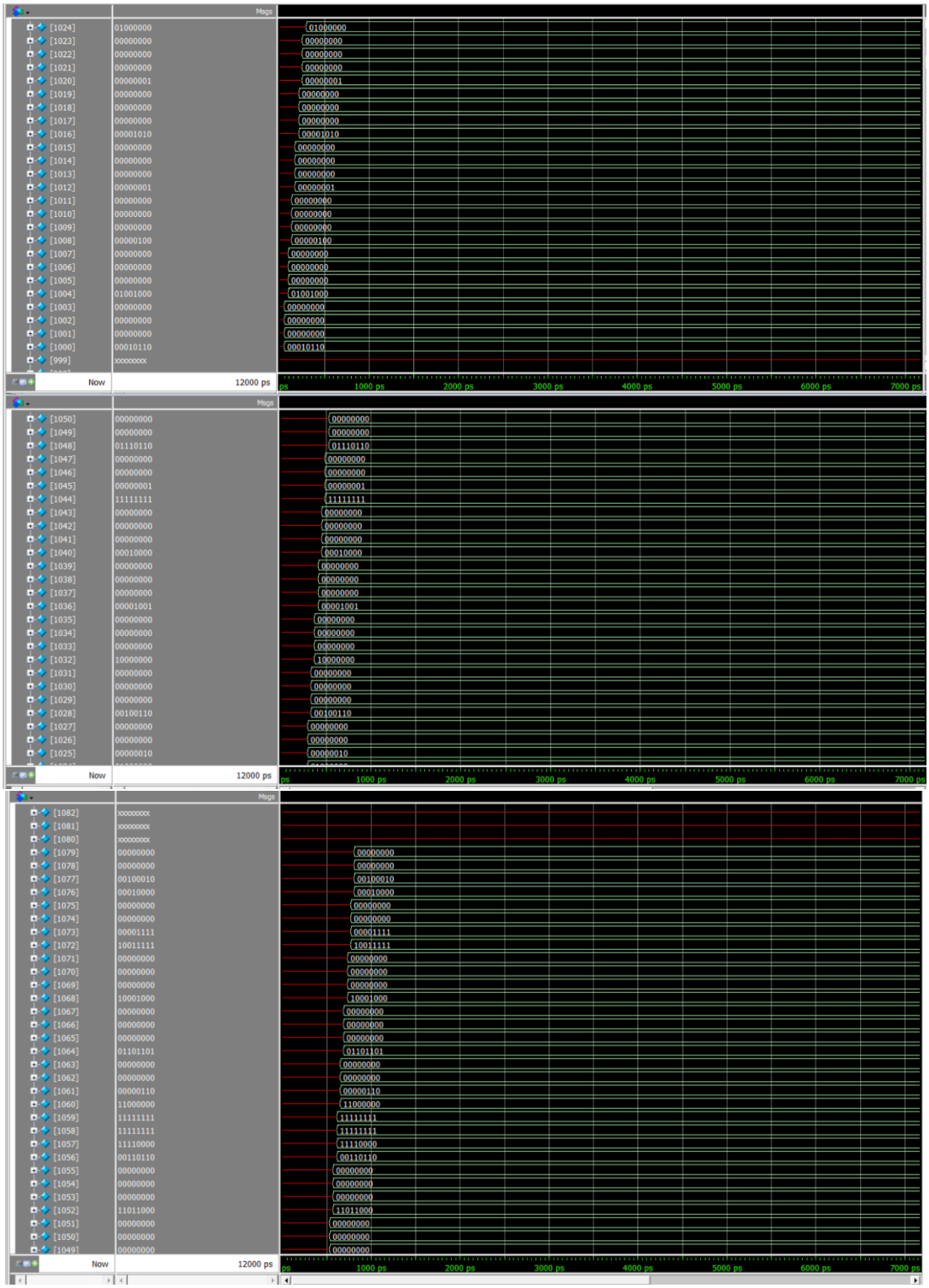


اجرای الگورینم برای یک درایه

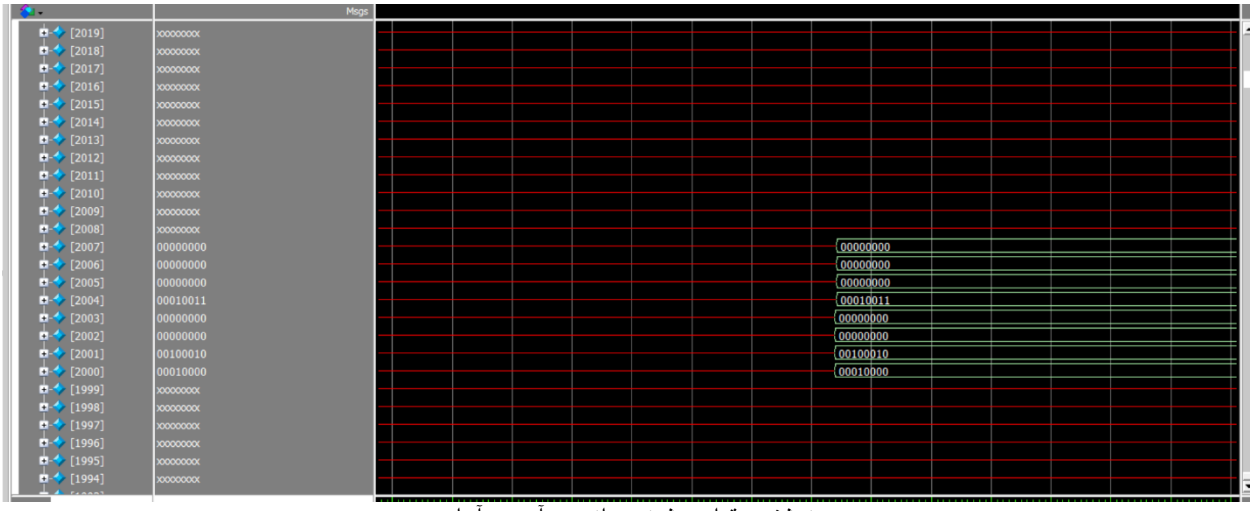


نمایی از اجرای برنامه و سپس اتمام فرامین موجود در حافظه دستورات

با مراجعه به فایل data.txt میتوان خانه های مموری را مشاهده و صحت الگوریتم بالا را باتوجه به مقدار موجود در آن تحقیق کنیم. شکل موج دیتاممووری:



پر شدن خانه های مموری با دستورsw (21 خانه پرشده و خانه 21 بیشینه کل این 21 مقدار است اما الگوریتم آنرا درنظر نمیگیرد چون فقط 20 درایه اول چک میشوند؛ هدف از این کاز نشان دادن صحت برنامه در شرایط مرزی بوده است)



نوشتن مقدار بیشینه و اندیس آن در آرایه

توجه شود که ما در دیتاموری از حافظه های 8 بیت استفاده کردیم اما خروجی آن که به پردازنده داده میشود، باید چهارتا رجیستر متوالی باهم کانکت شوند. البته این دیتاموری قابلیت آدرس دهی شکسته(دو بیت کم ارزش برابر 0و1و2و3) را پشتیبانی نمیکند؛ چراکه پردازنده ی میپس aligned میباشد.