تمرین عملی پنجم معماری کامپیوتر، دکتر اسدی

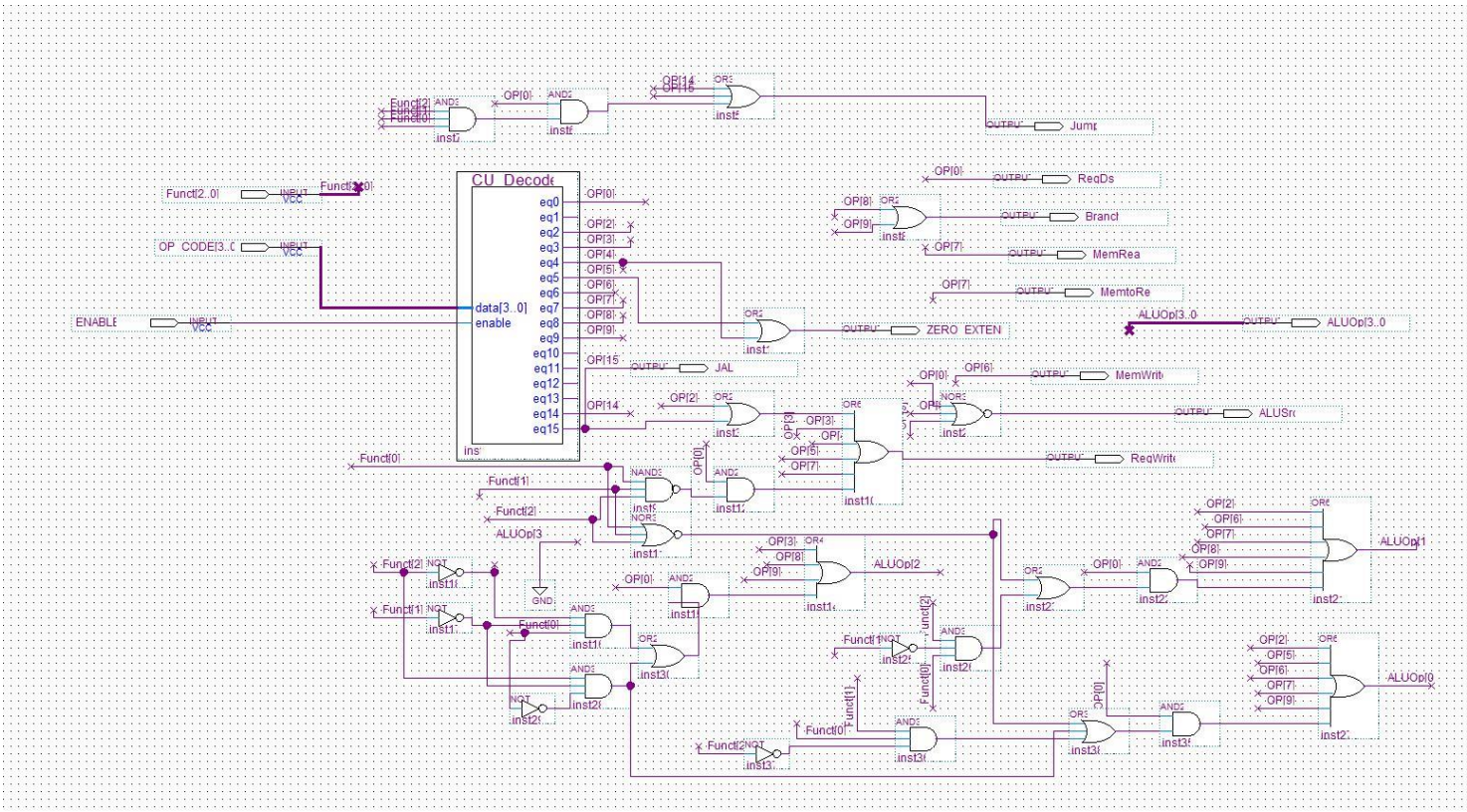
اعضای تیم: فرزام کوهی رونقی: 401106403 – **آریا همتی:** 401110523

گزارش کار:

در ابتدا برای بستن Mips_CPU خود، از قطعاتی که در دفعات قبل ساختیم استفاده می‌کنیم، شامل Instruction_Memory و Data_Memory، Register_File و ALU. مراحل بستن مدار و شکل کلی مدار مطابق اسلاید دکتر اسدی است، تنها تعدادی تفاوت دارد که در زیر به آنها اشاره می‌کنیم:

1 - بلوک Control: این بلوک با گرفتن مقدار Op_Code و Funct، مقادیر سیگنال‌های کنترلی شامل Jump, RegDst, Branch, MemRead, MemtoReg, AluOp, MemWrite, RegWrite, AluSrc را خروجی می‌دهد. جز این سیگنال‌های کنترلی که در Mips single cycle CPU عادی هم هستند، سینگال‌های کنترلی دیگری نیز خروجی می‌دهد که بخاطر بودن تعدادی دستور اضافه‌تر در داک تمرین عملی پنجم، احتیاج به ایجاد آن‌ها حس می‌شد. این سیگنال‌ها شامل JAL، ZERO_EXTEND هستند. JAL برای این است که دستور JAL از سایر دستورها تمایز داده شده و اگر دستور JAL باشد، مقداردیتای وارد شده در Register_File برابر PC+1 و همچنین این مقدار در رجیستر 7R ریخته می‌شود و این سیگنال کنترلی JAL به ما امکان پیاده سازی دستور JAL را می‌دهد. سیگنال ZERO_EXTEND نیز برای دستورهای دارای immediate استفاده می‌شود که اگر نیاز به ZERO_EXTEND باشد این سیگنال برابر یک می‌شود و اگر نیاز به Sign extend باشد این سیگنال صفر خواهد بود.

تصویر بلوک Control در زیر موجود است:



2 - بلوک CPU Clock Handler:

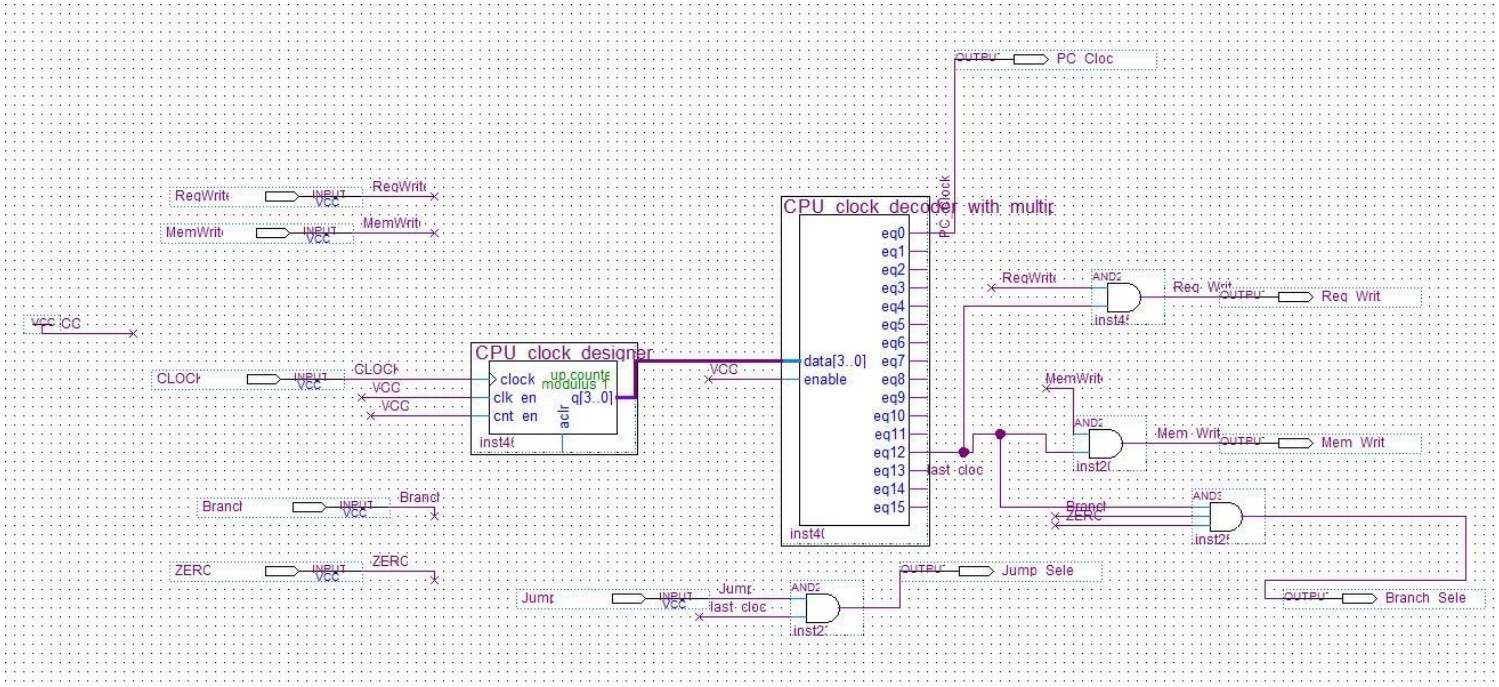
این بلاک ورودی های flag های حاصل از Control را گرفته و Clock کلی را کنترل میکند. فلسفه ی کلی این است که تمامی قطعات میتوانند هر operation ای را در هر زمانی انجام بدهند ولی مشکل تنها وقتی رخ میدهد که قرار باشد مقداری که در یک حافظه ای مانند register_file یا data_memory است تغییر بکند. برای همین flag هایی که write هستند در این بخش با توجه به clock روشن میشوند.

سایکل های کلی به طول ۱۳ هستند. برای رسیدن به این هدف از یک counter به 13 mod استفاده شده است که تمامی write ها با توجه به کلاک ها خورده میشوند (کلای write register و memory write هر دو در کلاک ۱۳ زده میشوند ولی PC در اولین کلاک زده میشود)

منطق کلاک ها هم بدین شکل است که عملیات های عادی در ۶ clock اوکی میشوند ولی تنها مشکل عملیات ضرب است که ۷ کلاک بیشتر از حالت عادی (دستور های تک کلاکه) طول میکشد. سر همین این تقسیمات با توجه به جدول زیر مشخص شده است:

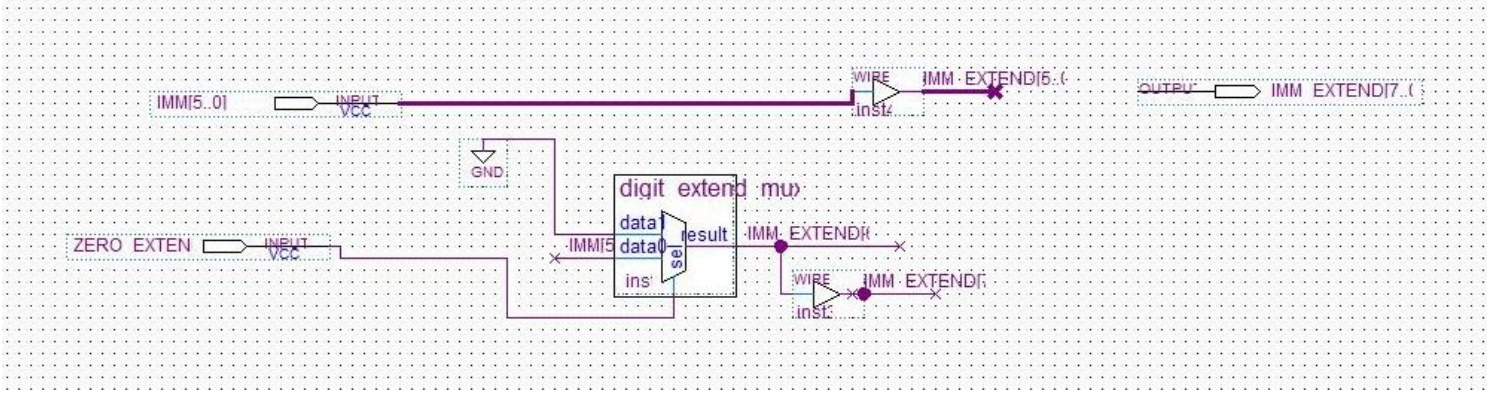
27					
28	PIECE	Clocks	6 cycle		
29	PC	1	6		
30	Instruction Mem	2	6		
31	Control	2	NAN		
32	Write register Mux	2	6		
33	Extender	2	6		
34	ALU SRC MUX	2	6		
35	Shift Left	2	6		
36	Register File (Read)	3	6		
37	ALU	4	6		
38	Memory (Load)	5	6		
39	Memory (Store)	5	6		
40	Register File (Write)	6	6		
41					

خروجی های این قطعه عبارتند از Mem_Write, Reg_Write, PC_Clock, Jump_Select, Branch_Select



تصویر بالا تصویر بلوک CPU_Clock_Handler است.

3 – بلوک Extender: کار این قطعه، extend کردن مقدار immediate با توجه به سیگنال کنترلی ZERO_EXTEND خارج شده از واحد کنترل است که تصمیم می‌گیرد به صورت sign یا zero، اسکتند را انجام دهد.



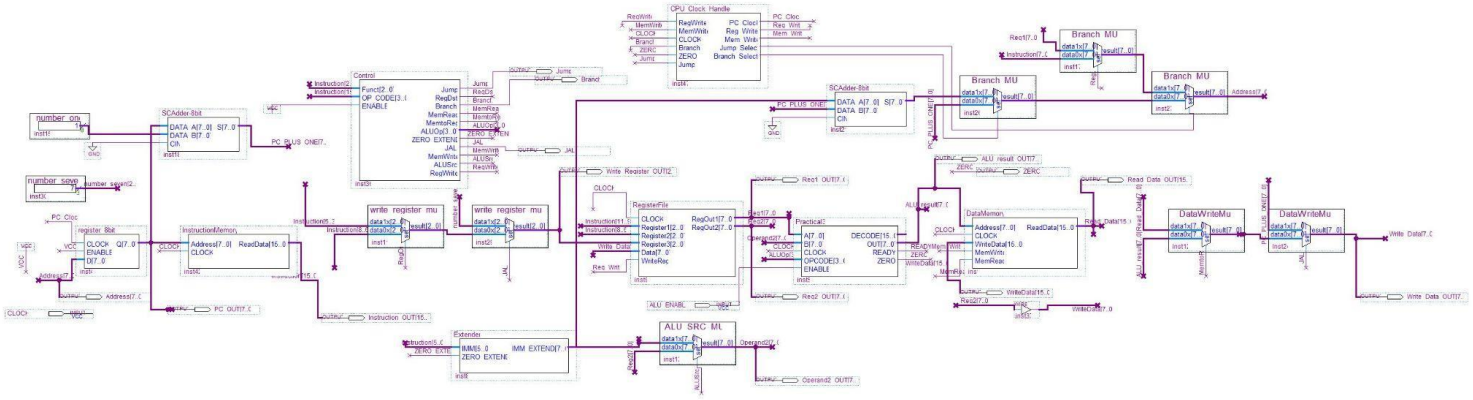
4 – سه عدد MUX اضافه بر اسلاید دکتر اسدی: سه عدد MUX اضافه تر استفاده شده که یکی از آنها در Write_Data مربوط به Register_File هستند که علت آن به داشتن دستور JAL برمی‌گردد که نیاز است PC -> R7 + 1. دیگری در ورودی مقدار 3Register در Register_File است که برای تعیین رجیستر مقصد برای write استفاده می‌شود و از آنجا که این مقدار در دستور JAL نیاز است 7R باشد، این MUX از سیگنال کنترلی JAL استفاده می‌کند. MUX آخر نیز برای نمایش مقصد Jump است، که برای 2 مقدار استفاده می‌شود: دستور JR که در آن رجیستر rs به عنوان مقصد جامپ استفاده می‌شودو دستور J ,JAL که در هر دو از مقدار address (همان immediate اما 8 بیتی) برای جامپ مستقیم استفاده می‌شود.

نکات پیاده سازی: تعدادی نکته حائز اهمیت در پیاده سازی ما موجود است که شامل موارد زیر است:

الف) اینکه در پیاده سازی ما برای دستورات Branch به گونه‌ای است که کلا احتیاجی به shift دادن نداریم، به این دلیل که instructin_memory ما word_addressable است و برای همین، در خطوط فرد نیز دستور موجود است.

ب) در دستورات Jump شامل JAL و J نیز address شیفت نمی‌یابد و در اصل مقدار آن 8 بیت است و 4 بیت Unused در این دستورات داریم. بنابراین نیازی به شیفت دادن نیست.

_شکل کلی مدار MIPS_CPU در زیر موجود است:



Testing:

برای تست کردن از دو فایل datas.mif و insts.mif استفاده کردیم که هر کدام به ترتیب مسئولیت مقداردهی اولیه حافظه‌های data و instruction را دارند. (دقت کنید چون 3 تا تست داریم 3 تا از این فایل‌ها موجود است، datas1, 2, 3 و همین منوال برای insts.) این دو فایل را به عنوان پارامتر به data_memory و instruction_memory می‌دهیم. سپس با کامپایل کردن مدار اصلی و اجرا کردن فایل ویوفرم، تغییرات مشاهده می‌شوند.

نحوه ساخت دستورات نیز مطابق داک تمرین است، در هر تست ابتدا دستورات اسمبلی مرتبط را نوشته و آنها را به کد ماشین تبدیل می‌کنم و بعد با یک مبدل آنها را به decimal تبدیل میکنم (امکان مقداردهی مستقیم به صورت باینری یا هگز نیز وجود دارد اما ما مقدار را به صورت decimal وارد کردیم.)

تست اول:

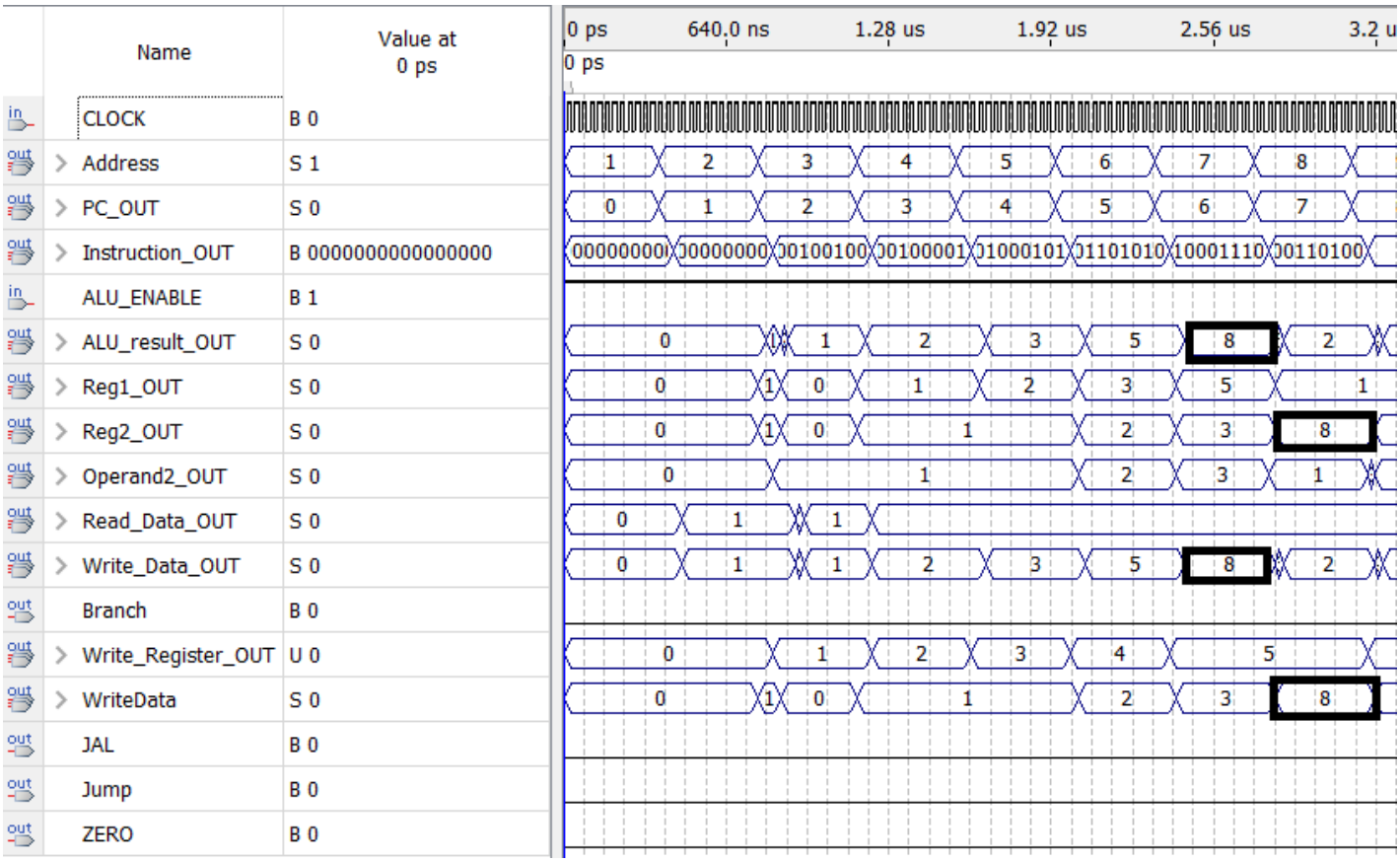
فیبوناتچی با ورودی 5: دقت شود که دنباله فیبوناتچی را با اندیس 0 و مقدار 1 شروع کردیم و اندیس 1 را نیز برابر 1 قرار دادیم، بنابراین فیبوناتچی با ورودی 5، خروجی 8 خواهد داد. مقدار اولیه حافظه نیز در خانه 0 و 1 برابر 1 است. خط اول در زیرف اندیس فیبوناتچی و خط دوم مقادیر آن اندیس ها را نشان می‌دهد.

0, 1, 2, 3, 4, 5
1, 1, 2, 3, 5, 8

Line	Assembly	Machine Code	Machine Code (DEC)
1	LB r0, 0(r0)	0111000000000000	28672
2	LB r1, 1(r1)	0111001001000001	29249
3	ADD r2, r1 r0	0000001000010000	528
4	ADD r3 r2, r1	0000010001011000	1112
5	ADD r4 r3 r2	0000011010100000	1696
6	ADD r5 r4 r3	0000100011101000	2280
7	SB r5, 1(r1)	0110001101000001	25409

نحوه کار کد بالا بدیهی است و صرفا مقدار فیبوناتچی با ورودی 5 را که برابر 8 است محاسبه می‌کند و مقدار را در r5 ریخته و در نهایت در خانه با آدرس 2 حافظه ذخیره می‌کند.

حال waveform به شرح زیر خواهد بود:



تست دوم: تست دلخواه: دستورات اسمبلی معادل این تست در زیر قرار گرفته است، در این تست هدف این بوده که از دستورات Rtype و immediate استفاده شده و همچنین مانور زیادی روی دستورات jump و branch داده شده و کارکرد صحیح آن مورد بررسی قرار گرفته است. دستورات اسمبلی این تست به شرح زیر است:

`Line1: LB r0, 15(r0)` - Load byte from memory address calculated by `15 + r0` into `r0`

`Line2: LB r1, 6(r0)` - Load byte from memory address calculated by `6 + r0` into `r1`.

`Line3: ADD r2, r1, r0` - Add: `r2 = r1 + r0`.

`Line4: SUB r5, r1, r0` - Subtract: `r5 = r1 - r0`.

`Line5: MULT r2, r1, r0` - Multiply: `r2 = r1 * r0`.

`Line6: OR r3, r1, r0` - Logical OR: `r3 = r1 OR r0`.

`Line7: XOR r4, r1, r0` - Logical XOR: `r4 = r1 XOR r0`.

`Line8: ADDi r2, r4, -1` - Add immediate: `r2 = r4 + (-1)`.

`Line9: SB r2, r6(7)` - Store byte from `r2` into memory address calculated by `7 + r6`.

`Line10: LB r1, r6(7)` - Load byte from memory address calculated by `7 + r6` into `r1`.

`Line11: J 14` - Jump to address `14`.

`Line12: JAL 16` - Jump and link (save return address) to address `16`.

`Line14: JR r4` - Jump to address contained in `r4`.

`Line16: BNE r0, r7, 5` - Branch to address `current address + 4 + (5*4)` if `r0` ≠ `r7`.

`Line22: BEQ r1, r2, -7` - Branch to address `current address + 4 + (-7*4)` if `r1` = `r2`.

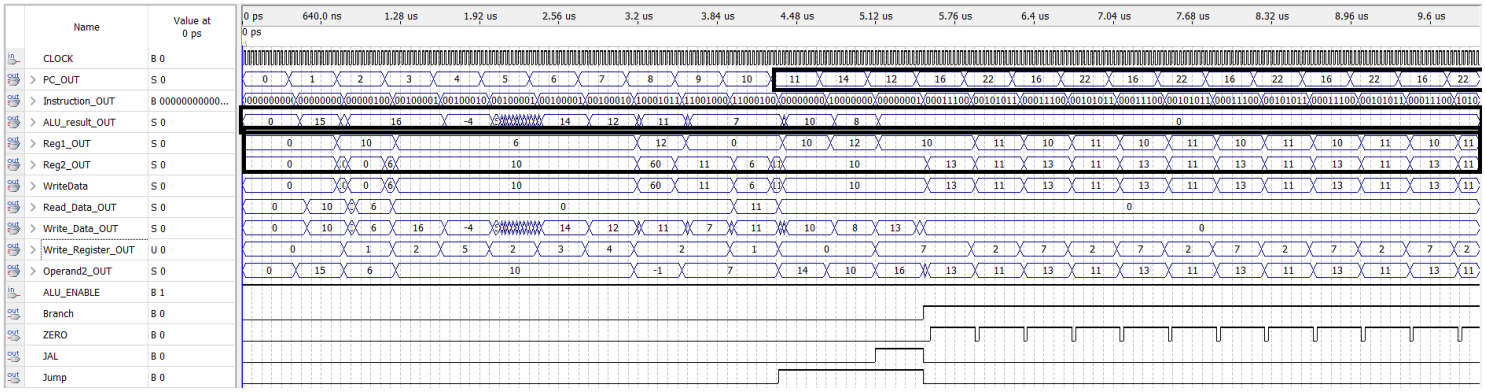
دستورات اسمبلی انجام شده به شرح زیر است، دقت کنید که در سمت چپ شماره خطوط (شماره خط دستور در حافظه) نوشته شده است تا بررسی jump ها و branch ها به دقت صورت گیرد.

#	Instruction	Binary Representation	Hexadecimal	Decimal
1	`LB r0, 15(r0)`	0111000000001111	700F	28687
2	`LB r1, 6(r0)`	0111000001000110	7046	28742
3	`ADD r2,r1,r0`	0000001000010000	0210	528
4	`SUB r5,r1,r0`	0000001000101001	0229	553
5	`MULT r2,r1,r0`	0000001000010100	0214	532
6	`OR r3,r1,r0`	0000001000011011	021B	539
7	`XOR r4,r1,r0`	0000001000100101	0225	549
8	`ADDi r2,r4,-1`	0010100010111111	28BF	10431
9	`SB r2, r6(7)`	0110110010000111	6C87	27783
10	`LB r1, r6(7)`	0111110001000111	7C47	31815
11	`J 14`	1110000000001110	E00E	57358
12	`JAL 16`	1111000000010000	F010	61456
14	`JR r4`	0000100000000111	0807	2055
16	`BNE r0,r7,5`	1001000111000101	91C5	37317
22	`BEQ r1,r2,-7`	1000001010111001	82B9	33465

توضیح مثال: ابتدا دستور 1 اجرا شده و مقدار r0 = 10 می‌شود (چون در مکان 15 حافظه مقدار 10 ریخته شده) سپس خط دوم اجرا شده و مقدار r1 = 6 می‌شود، (چون در مکان r0 + 6 حافظه یعنی مکان 16 حافظه مقدار 6 ریخته شده). سپس در خط سوم add = r2 = 16 و خط چهارم r5 = sub = -4 و در خط پنجم r2 = mult = 60 و در خط ششم or = r3 = 14 و در خط هفتم xor = r4 = 12 و در خط هشتم addi = r2 = r4 - 1 = 12 - 1 = 11 و در خط نهم مقدار خانه با آدرس 7 برابر مقدار r2 یعنی 11 می‌شود و در خط دهم، مقدار خانه با آدرس 7 در رجیستر r1 ریخته می‌شود یعنی r1 = 11.

حال به دستورات جامپ و برنچ میرسیم. در خط 11، ابتدا جامپ به 14 رخ می‌دهد و بعد آن JR r4 که یعنی جامپ به خط 12 رخ می‌دهد که در JAL داریم و به خط 16 می‌رویم و مقدار R7 برابر 13 می‌شود. در خط 16 نابرابر بودن r0, r7 چک می‌شود که r0 = 10 و r1 = 11 می‌شود و بنابراین نابرابر هستند و به 5 خط بعدی برنچ می‌شویم که PC + 1 + 5 می‌شود خط 22، در این خط نیز برابر بودن r1, r2 چک می‌شود که داریم r1 = r2 = 11. بنابراین برابری برقرار است و به 7 خط قبل می‌رویم یعنی PC + 1 - 7 که می‌شود خط 16، بنابراین یک لوپ بی‌نهایت بین خط 16 و 22 رخ می‌دهد چون شرط هر دو همواره صادق هستند.

Waveform Test2:



دقت شود که شماره خط دستورات توضیح داده شده دقیقا در PC_OUT قابل مشاهده است. مقادیر حاصل هر عملیات را نیز می‌توان در reg1_Out یا reg2_Out یا ALU_result_OUT بنابر دستور انجام شده مشاهده کرد. برای سادگی مصحح دور مقاصد احتیاج به مشاهده مستطیل رسم شده است.

تست سوم:

دستورات تست سوم به شکل زیر است:

Line	Asssembly	Machine Code	Decimal
1	ADDi r1, r0, 20	0010000001010100	8276
2	SUBi r2, r0, 10	0011000010001010	12426
3	ANDi r3, r1, 4	0100001011000100	17092
4	ORi r4, r1, 6	0101001100000110	21254
5	J 8	1110000000001000	57352
6	JR r1	0000001000000111	519
8	JAL 6	1111000000000110	61446
20	SB r7, r1(0)	0110001111000000	25536

دستورات خط به خط به شکل زیر اجرا می‌شوند:

خط اول: r1= r0 + 20 = 20

خط دوم: r2 = r0 - 10 = -10

خط سوم: r3 = r1 & 4 = 4

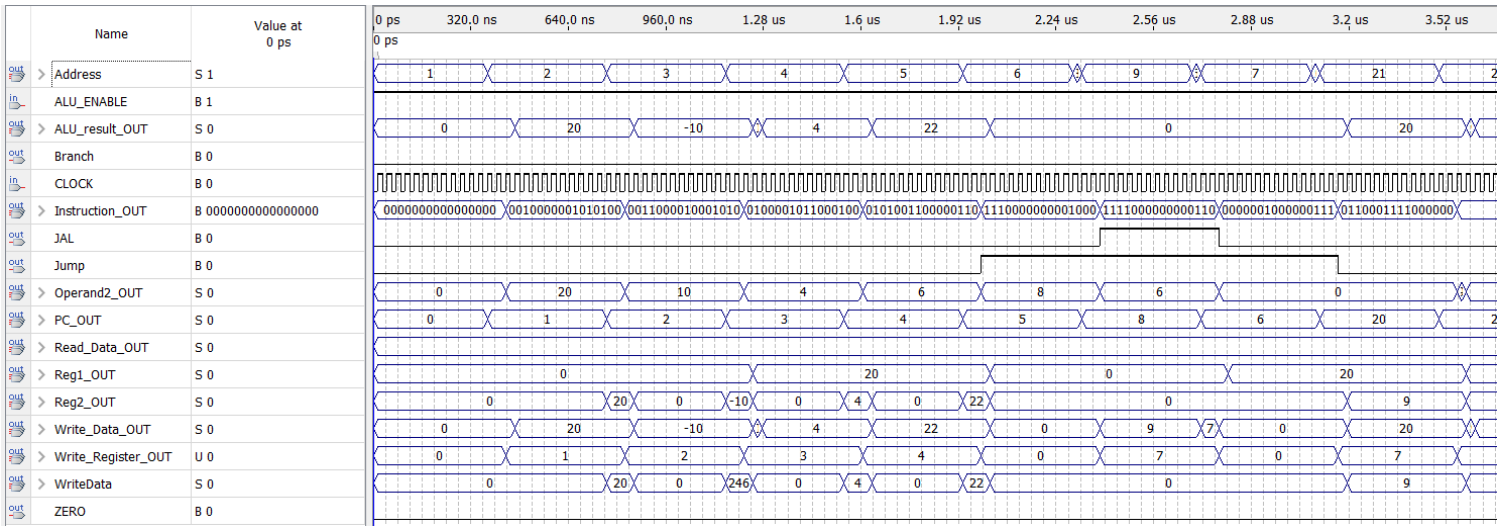
خط چهارم: r4 = r1 | 6 = 10100 | 00110 = 10110 = 22

خط پنجم: جامپ به خط هشتم

خط هشتم: جامپ اند لینک به خط 6 و r7 = 9.

خط ششم: جامپ به مقدار رجیستر r1 که یعنی به خط 20 ام.

خط بیستم: مقدار r7 = 9 را در خانه حافظه با آدرس 20 ذخیره میکند.



تصویر waveform بالا به وضوح مقادیر را نشان می‌دهد. (برای مشاهده مقادیر محاسباتی به ALU_result_OUT و برای مشاهده درست انجام شدن جامپ‌ها به مقادیر PC_OUT توجه شود.