تمرین ششم - عملی (واحد پردازشی و تست‌های قبلی)

واحد کنترلی (Control Unit): برای پیاده‌سازی control unit از منطق زیر و یک FSM با ۱۲ استیت استفاده کرده‌ایم:

instruction	operation	state	PCWriteCond	PCWrite	IorD	MemRead	MemWrite	MemToReg	IRWrite	PCPath	ALUOp	ALUSrcA	ALUSrcB	RegWrite	RegDst	MemRegSrc	ExtType	MultRes
all	i-fetch	0000	0	1	0	1	0	x	1	00	100	0	01	0	x	0	x	0
all	i-decode/RF: read	0001	0	0	x	0	0	x	0	xx	100	0	11	0	x	1	1	0
lb/sb	address calculation	0010	0	0	x	0	0	x	0	xx	100	1	10	0	x	0	1	0
lb	memory: read	0011	0	0	1	1	0	x	0	xx	xxx	x	xx	0	x	0	x	0
lb	RF: write	0100	0	0	x	0	0	1	0	xx	xxx	x	xx	1	0	0	x	0
sw	memory: write	0101	0	0	1	0	1	x	0	xx	xxx	x	xx	0	x	0	x	0
r-type	execution	0110	0	0	x	0	0	x	0	xx	111	1	00	0	x	0	x	0
r-type/immediate	RF: write	0111	0	0	x	0	0	0	0	xx	xxx	x	xx	1	y	0	x	0
beq/bnq	pc update	1000	1	0	x	0	0	x	0	01	yy1	1	00	0	x	0	x	0
j/jr	pc update	1001	0	1	x	0	0	x	0	1y	xxx	x	xx	0	x	0	x	0
immediate	execution	1010	0	0	x	0	0	x	0	xx	yyy	1	10	0	x	0	y	0
jal	pc update/RF: write	1011	0	1	x	0	0	1	0	10	xxx	x	xx	1	0	0	x	0
r-type: mult	mult execution	1100	0	0	x	0	0	x	0	xx	111	1	00	0	x	0	x	1

از ۱۶ سیگنال کنترلی تعریف‌شده، ۱۳ سیگنال ابتدایی در اسلایدها آمده‌اند و مطابق همان منطق مقادیر را تنظیم کرده‌ایم (سیگنال PCPath همان PCSourse داخل اسلایدهاست). اما برای پیاده‌سازی دستورات جدیدتر (نسبت به اسلایدها) ۳ سیگنال کنترلی جدید اضافه کردیم که در گزارش DataPath به توضیح عملکرد آن‌ها می‌پردازیم.

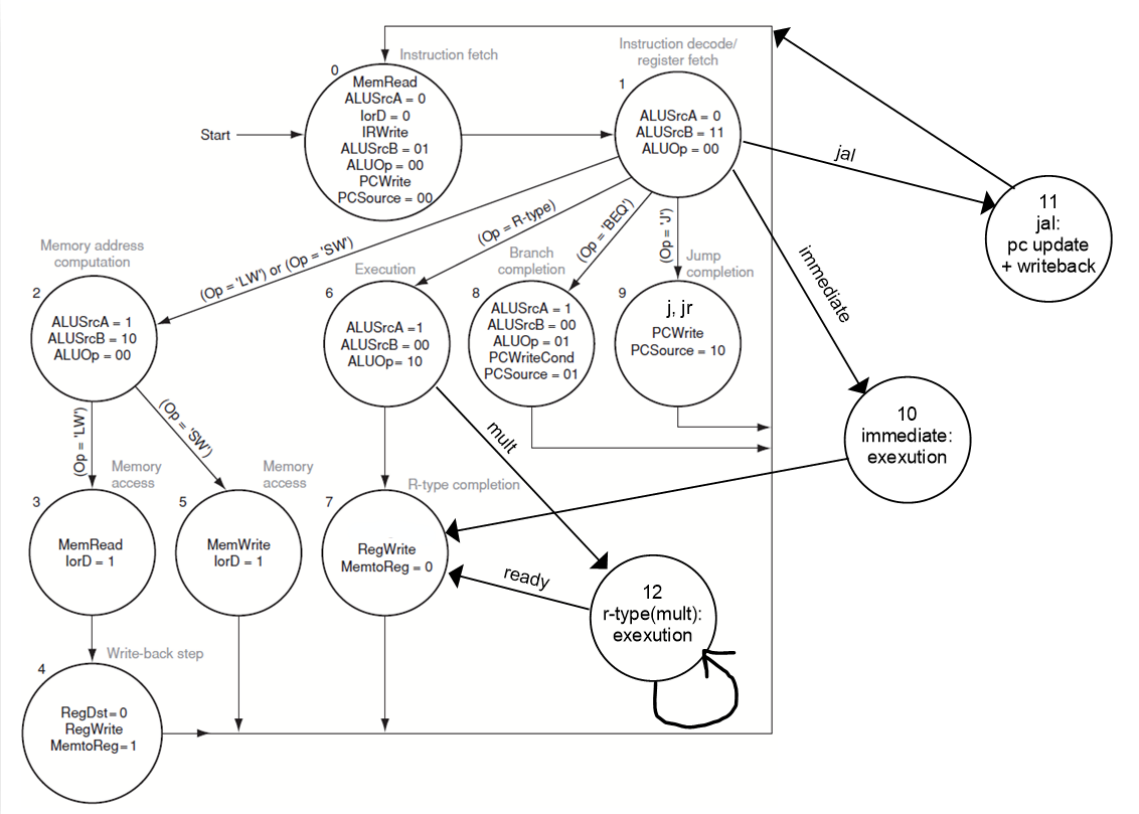
به طور مختصر MemRegSrc = 1 امکان R[7] ← PC در دستور jal را فراهم می‌کند؛ سیگنال ExtType امکان Zero-Extend کردن در دستورات ANDi و Ori را فراهم می‌کند و سیگنال MultRes امکان طول کشیدن بیش از یک cycle برای بخش execution دستور mult را فراهم می‌کند.

لازم به ذکر است که پردازنده‌ی multi-cycle به طور کامل و مطابق اسلایدها پیاده‌سازی شده‌است.

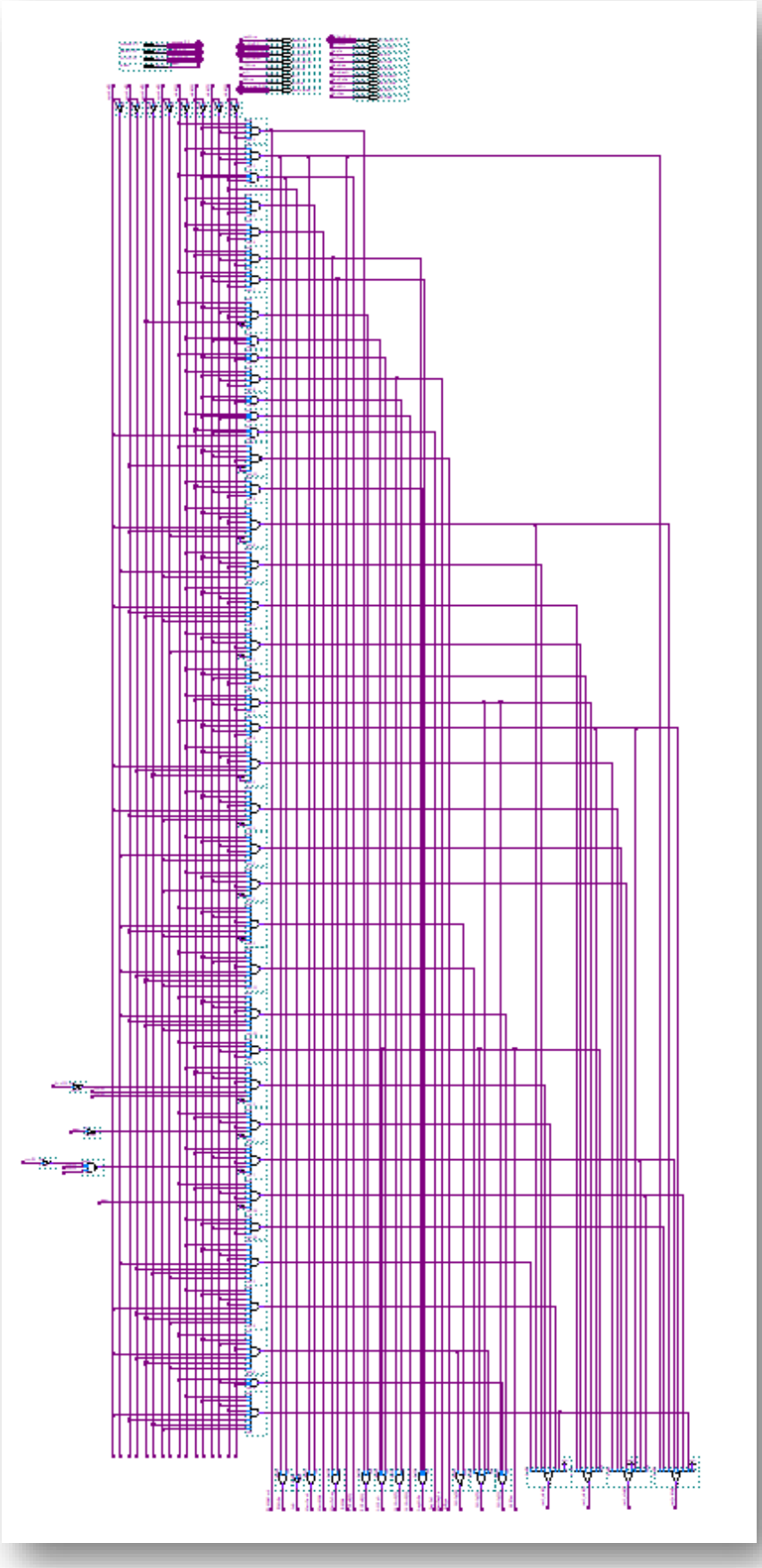
جدول بالا مقادیر سیگنال‌های کنترلی را نشان می‌دهد و در جدول روبرو می‌توانید منطق مربوط به جابجایی بین استیت‌ها در FSM را ببینید (در خطوط ۱۳ و ۱۴ جدول، ۳ بیت دوم نشان‌دهنده‌ی funct هستند):

current state	opcode	next state
0000	xxxx	0001
0001	100x	0010
0001	1111	0110
0001	0000/1010	1000
0001	11x0	1001
0001	0xx1	1010
0001	1101	1011
0010	xxx1	0011
0010	xxx0	0101
0011	xxxx	0100
0100	xxxx	0000
0101	xxxx	0000
0110	xxxx-011	1100
0110	xxxx-else	0111
0111	xxxx	0000
1000	xxxx	0000
1001	xxxx	0000
1010	xxxx	0111
1011	xxxx	0000
1100	ready = 0	1100
1100	ready = 1	0111

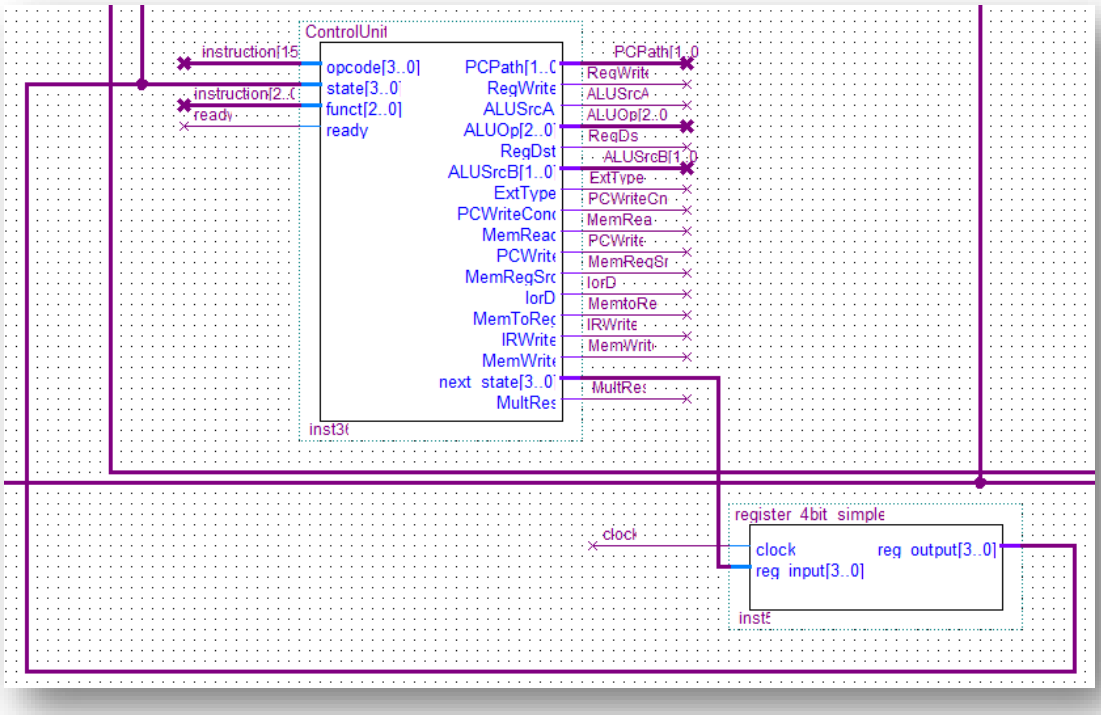
FSM مربوط به پردازنده مطابق شکل زیر است:



توضحیات استیت‌های اضافه‌شده در شکل آمده‌اند. برای پیاده‌سازی منطق control unit از PLA استفاده کردیم و به صورت دو لایه (لایه‌ای از and ها و or های قابل برنامه‌ریزی) درآمده‌اند. تصویر کلی آن را در زیر می‌بینید و در فایل ControlUnit که در پوژه آمده‌است، جزئیات آن واضح هستند:



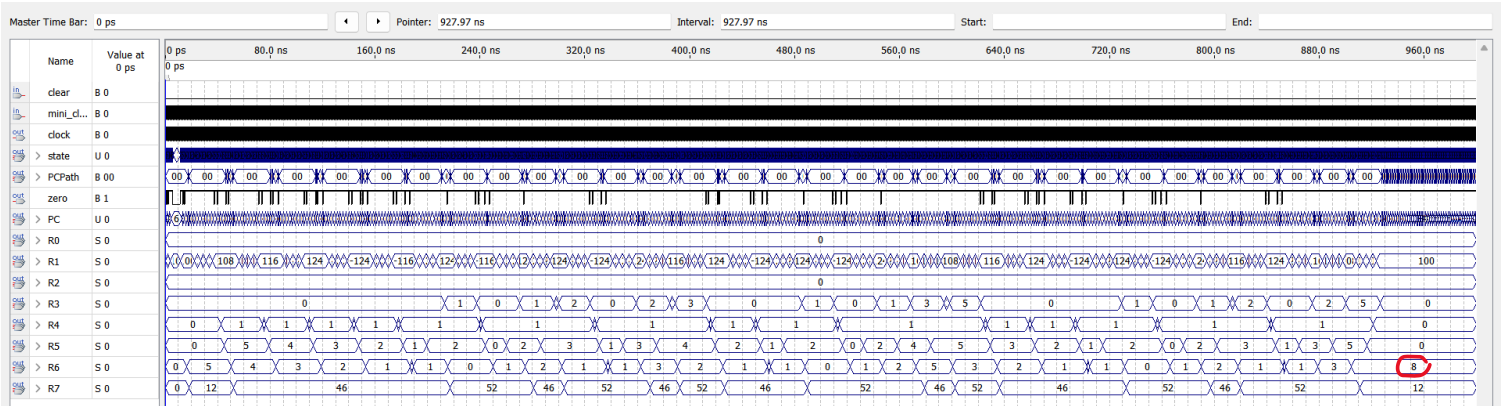
تصویر control unit در datapath در زیر آمده‌است:



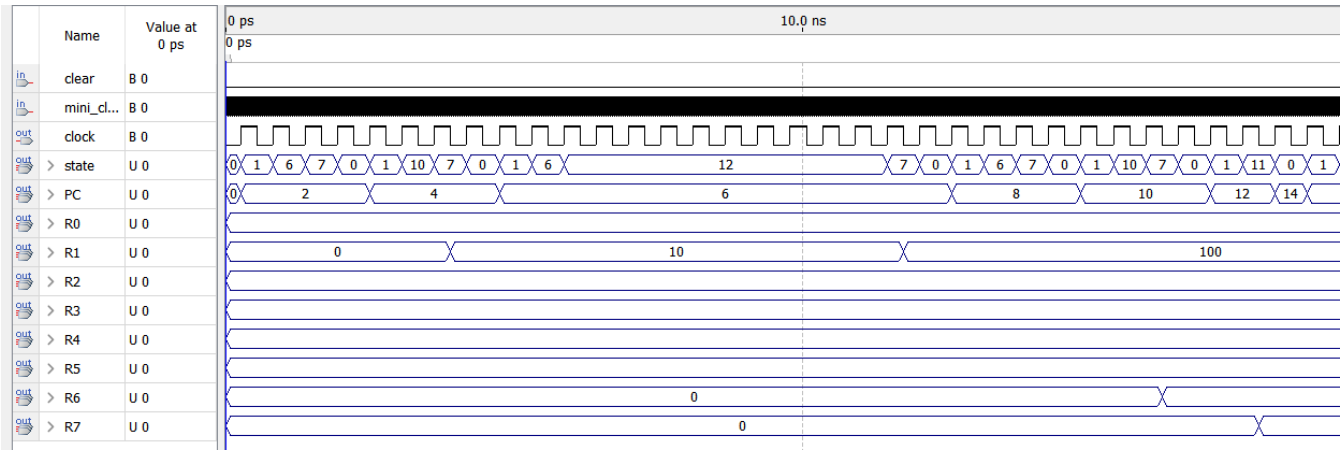
همان‌طور که می‌بینید این واحد ۳ ورودی دارد: ۱) opcode دستور؛ که همان بیت‌های ۱۲ تا ۱۵ دستور است. ۲) state فعلی؛ که برای مشخص کردن شماره‌ی استیت فعلی FSM می‌باشد و چون ۱۲ استیت داریم، به یک نشانگر ۴ بیتی احتیاج داریم.۳) funct دستورات R-type؛ برای تمایز بین دستور mult و دیگر دستورات R-type کاربرد دارد. ۴) سیگنال ready؛ در دستور mult از دسته‌ی دستورات R-type ممکن است مرحله‌ی execution چند cycle طول بکشد و پس از آماده‌شدن حاصل و اتمام execution، سیگنال ready فعال می‌شود و با ورودی دادن آن به control unit، FSM به استیت writeback می‌رود.

خروجی‌های این واحد نیز به این صورت هستند: ۱) ۱۶ سیگنال خروجی که در جدول ابتدایی گزارش آمده بودند. ۲) next_state؛ ۴ بیت برای مشخص کردن استیت بعدی FSM. برای این که کلاک دستورات به هم نریزد، این خروجی در یک رجیستر ۴ بیتی ذخیره می‌شود و در لبه بالارونده‌ی کلاک (هنگام شروع cycle بعدی)، مقدار آن را دوباره به واحد کنترلی ورودی می‌دهد.

تست فیبوناچی: همانطور که انتظار داشتیم، مقدار Fib(5) = 8 پس از اجرای بازگشتی برنامه در رجیستر R6 ذخیره شده‌است. کد اسمبلی آن نیز مشابه بخش قبلی است؛ با این تفاوت که چون پردازنده Unified Memory است، استک پوینتر (R1) مقدار اولیه‌ی برابر با 100 دارد و دستورات برنامه قبل از خانه‌ی ۱۰۰ حافظه نوشته شده‌اند تا تداخلی بین مقادیر مموری و دستورات رخ ندهد. خروجی waveform کلی را در زیر مشاهده می‌کنید:



برای بررسی MultiCycle بودن پردازنده، نگاهی جزئی‌تر به بخشی از دستورات می‌اندازیم:

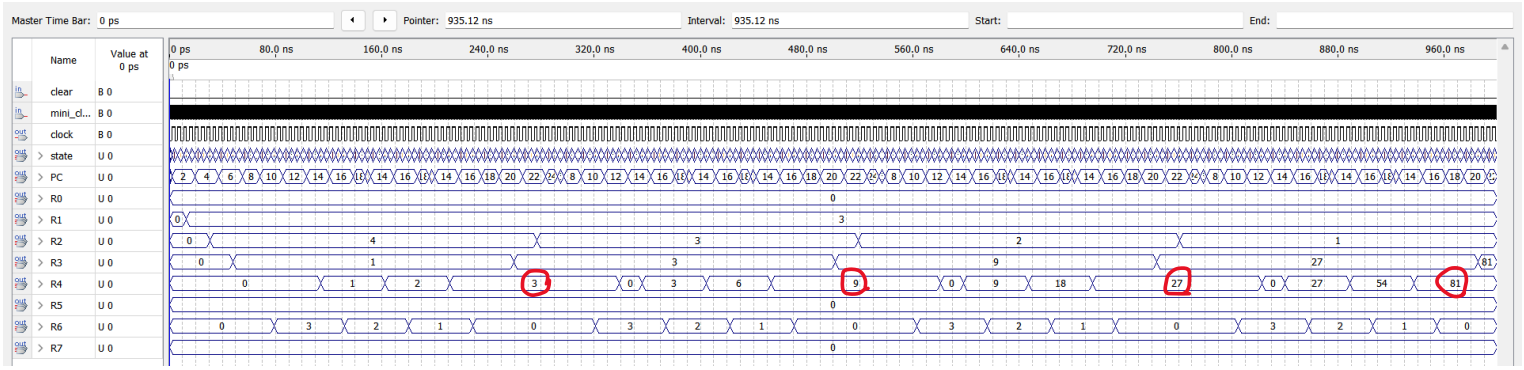


همان‌طور که می‌بینید، به طور مثال دستور اول که یک دستور R-type است، به ترتیب استیت‌های ۰ و ۱ و ۶ و ۷ را پیمایش می‌کند؛ یا دستور بعدی که immediate است، استیت‌های ۰ و ۱ و ۱۰ و ۷ را پیمایش می‌کند. دستور بعدی که mult است، استیت‌های ۰ و ۱ و ۶ و ۱۲ (چند بار) و ۷ را طی می‌کند. توجه شما را به نکات پیاده‌سازی جلب می‌کنیم:

مقادیر رجیسترها در دستوراتی که writeback in register file دارند، در استیت ۷ انجام می‌شود.

مقدار PC هر بار پس از استیت ۰ آپدیت شده و ۲ تا زیاد می‌شود؛ سپس در صورت نیاز به تغییر (در دستورایی مانند beq و bnq و j و jal و jr) پس از cycle سوم دستور مربوطه دوباره آپدیت می‌شوند. در غیر این صورت نیز تا انتهای دستور همان PC + 2 باقی می‌مانند.

تست توان: مشابه تمرین قبل سعی کردیم تست به توان ۴ رساندن عدد ۳ را انجام دهیم. از طرفی عملیات ضرب در این تست با جمع‌های متوالی پیاده‌سازی شده‌است (چون در تمرین قبل عملیات mult نداشتیم). می‌توانید فرآیند محاسبه‌شدن توان‌های عدد ۳ تا مقدار ۴ را در رجیستر R4 ببینید:

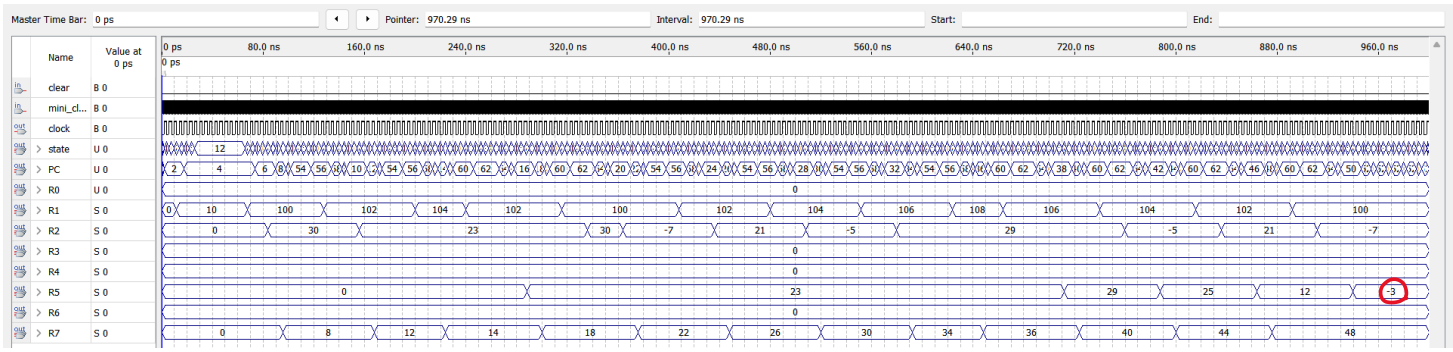


تنها تفاوت این تست با تمرین پیش در این است که اینجا دستورات در چند کلاک اجرا می‌شوند.

تست استک (پشته): برای تست استک نیز مقدار عبارت زیر را محاسبه می‌کنیم:

$$(۳۰ - ۲۳) \vee (۲۱ \oplus (-۵ \wedge ۲۹))$$

تصویر waveform را مشاهده می‌کنید:



همان‌طور که می‌بینید خروجی مورد انتظار 3- تولید شده‌است.

تفاوت این تست (و البته تست فیبوناچی) با تمرین پیش این است که چون این پردازنده multicycle است و unified memory دارد و این مموری کلمات ۱۶ بیتی دارد، لازم است در هر ۱۶ بیت یا یک دستور نوشته‌شود و یا در ۸ بیت سمت راست آن یک مقدار عددی نوشته‌شود. پس برای دسترسی به خانه‌های حافظه باید از آدرس‌های زوج استفاده کنیم و هنگام پیاده‌سازی استک، آدرس را ۲ تا ۲ تا کم یا زیاد کنیم.