به نام خدا

تمرین عملی ششم معماری کامپیوتر دکتر اسدی

اعضای تیم :

- فرزام کوهی رونقی 401106403
- آریا همتی 401110523
- ثنا بابایان 401105689

گزارش کار :

برای بستن پردازنده مولتی سایکل میپس خود از قطعاتی که در تمرینات قبلی انجام دادیم استفاده شد که آنها را نام می‌بریم :

- رجیستر 8 بیتی به عنوان PC
- یک unified memory برای نگه داشتن همزمان دستورات و مقادیر در حافظه
- رجیستر 16 بیتی به عنوان instruction memory
- رجیستر 8 بیتی به عنوان memory data register
- Register file
- رجیسترهای 8 بیتی A, B
- ALU که از تمرین 3 می باشد.
- رجیستر 8 بیتی ALUOut
- Control unit multi cycle
- یک بلوک به نام extender که کار zero extend و یا sign extend را قرار است انجام دهد
- Mux ورودی برای سیگنال IorD
- Mux سه ورودی برای مشخص کردن RegDst (یکی آدرس رجیستر هفت برای دستور JAL و بقیه طبق اسلایدها)
- Mux دو ورودی برای data write رجیستر فایل که یکی از ALUOut و دیگری از memory data register می آیند
- Mux دو ورودی برای قسمت بعد رجیستر A که ورودی های آن مشابه اسلایدهاست
- Mux سه ورودی برای قسمت بعد از رجیستر B که مانند اسلایدهاست ولی چون دیگری نیازی به 2 بیت شیف نداشتیم آن را از طراحی خود حذف نمودیم.
- Mux چهار ورودی برای PC که علاوه بر سه ورودی اسلاید دارای یک ورودی دیگر است که همان RegOut خارج شده از register file می باشد.

شرح بلوک کنترلی :

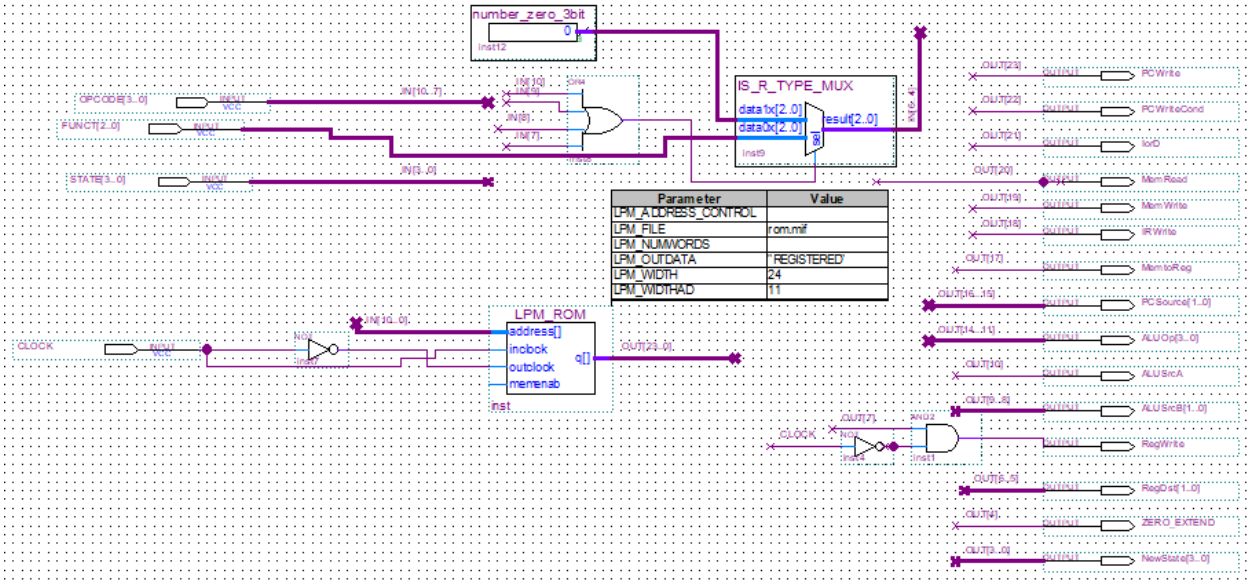
بلوک هایی استفاده شده در این تمرین همه در تمرین های قبلی توضیح داده شده‌اند البته به جز بلوک کنترلی که اکنون به آن میپردازیم :

ورودی های این بلوک برابر با 4 بیت upcode و 3 بیت funct و 4 بیت state و کلاک است.خروجی های این بلوک شامل سیگنال های کنترلی از جمله : (با ترتیب نوشته شده اند)

PCWrite, PCWriteCond, IorD, MemRead, MemWrite, IRWrite, MemtoReg, PCSource[1..0], ALUOp[3..0], ALUSrcA, ALUSrcB[1..0], RegWrite, RegDst[1..0], ZeroExtend

و همچنین 4 بیت State[3..0] هست.

شکل زیر نگاهی کلی به ساختار درون واحد کنترلی ارایه می دهد :

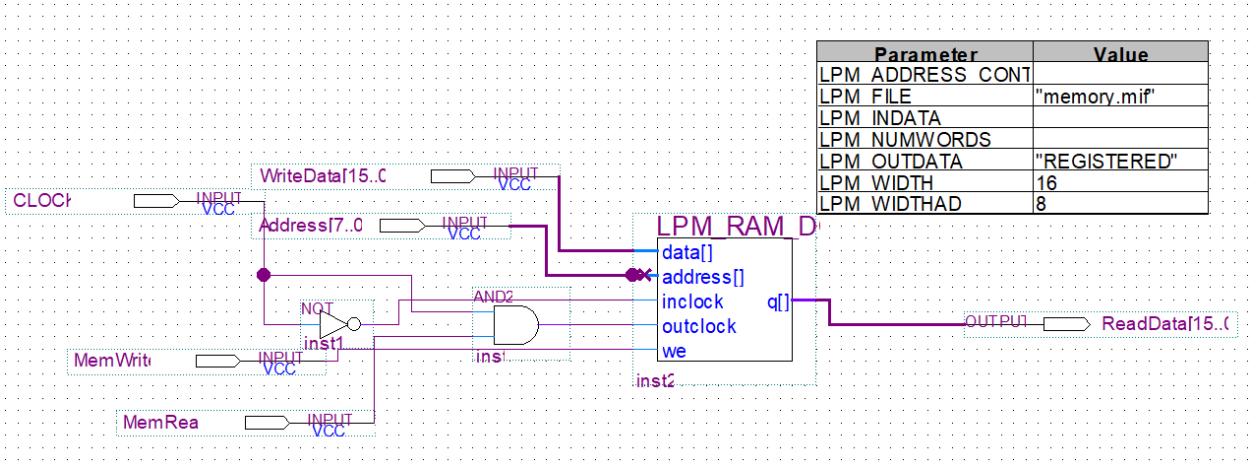


در طراحی این قسمت از یک LPM_ROM با LPM_WIDTHAD = 11 , LPM_WIDTH = 24 که به شکل REGISTERED است استفاده کردیم و فایلی که به این ROM داده می شود rom.mif است.

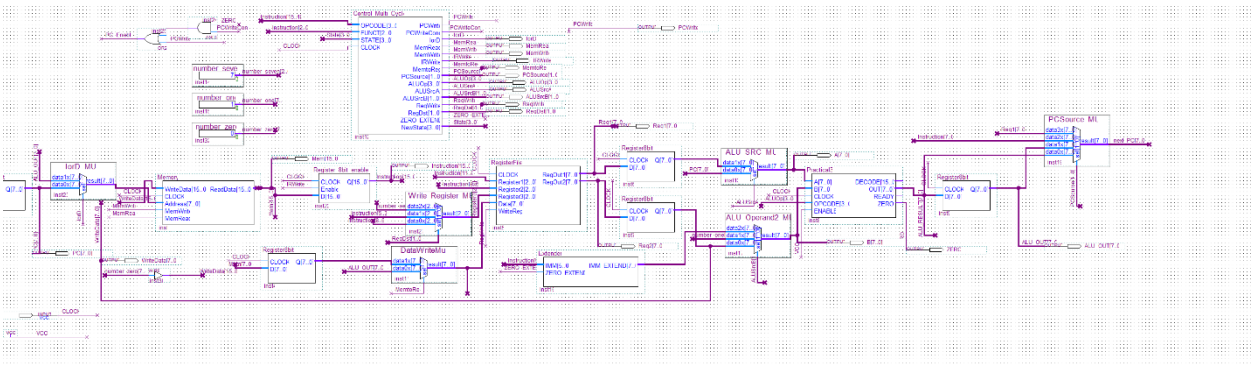
بنا به ضرورت بعضی از سیکنال ها با کلاک و یا با نات کلاک and شده اند تا تغییراتی که مورد نظر است از در لبه هاب درست کلاک انجام گیرند.

تغییر جزئی در مدار Memory به دلیل Unified شدن:

مدار Memory دچار تغییر جزئی ای شده که در تصویر زیر عکس مدار جدید آن قرار گرفته است:



شکل کلی مدار MIPS_CPU_MULTI_CYCLE:



تعداد کلاک های مورد نیاز برای انجام دستورات مختلف :

همانطور که واضح است در طراحی مولتی سایکل تعداد کلاک هایی که هر کدام از دستورات طول می کشند با همدیگر متفاوت است.مراحل را به اختصار دستور load توضیح می دهیم و بقیه هم به همان ترتیب خواهند بود : (برای پیاده سازی واحد کنترلی از تکنیک ROM و Micro-program استفاده کردیم به این شکل که هر دستور به تعدادی micro-ops تقسیم شده و سیگنال‌های کنترلی آنها مطابق زیر در فایل rom.mif قرار گرفته است.)

R-Type except MULT, JR :

```
-- ADD
00000000001 : 00010100000000000000010;
00000000010 : 00000000000000000000011;
00000000011 : 00000000000000000000100;
00000000100 : 00000000000111000000101;
00000000101 : 00000000000111000100110;
00000000110 : 00000000000111001101111;
00000000111 : 10000000000000000000001;

-- SUB
00000010001 : 00010100000000000000010;
00000010010 : 00000000000000000000011;
00000010011 : 00000000000000000000100;
00000010100 : 00000000001001000000101;
00000010101 : 00000000001001000100110;
00000010110 : 00000000000111001101111;
00000010111 : 10000000000000000000001;
```

```
-- AND
00000100001 : 00010100000000000000010;
00000100010 : 00000000000000000000011;
00000100011 : 00000000000000000000100;
00000100100 : 00000000000010000000101;
00000100101 : 00000000000010000100110;
00000100110 : 00000000001100110100111;
00000100111 : 10000000000000000000001;
-- OR
00000110001 : 00010100000000000000010;
00000110010 : 00000000000000000000011;
00000110011 : 00000000000000000000100;
00000110100 : 00000000000110000000101;
00000110101 : 00000000000110000100110;
00000110110 : 00000000001100110100111;
00000110111 : 10000000000000000000001;
-- XOR
00001010001 : 00010100000000000000010;
00001010010 : 00000000000000000000011;
00001010011 : 00000000000000000000100;
00001010100 : 00000000001010000000101;
00001010101 : 00000000001010000100110;
00001010110 : 00000000001100110100111;
00001010111 : 10000000000000000000001;
```

: JR

```
-- JR
00001110001 : 00010100000000000000010;
00001110010 : 00000000000000000000011;
00001110011 : 00000000000000000000100;
00001110100 : 00000000011010000000101;
00001110101 : 10000001100000000000001;
```

: MULT

```
-- MULT
00001000001 : 00010100000000000000010;
00001000010 : 00000000000000000000011;
00001000011 : 00000000000000000000100;
00001000100 : 00000000001011000000101;
00001000101 : 00000000001011000000110;
00001000110 : 00000000001011000000111;
00001000111 : 00000000001011000001000;
00001001000 : 00000000001011000001001;
00001001001 : 00000000001011000001010;
00001001010 : 00000000001011000001011;
00001001011 : 00000000001011000001100;
00001001100 : 00000000001011000001101;
00001001101 : 00000000001011000101110;
00001001110 : 00000000000111001001011111;
00001001111 : 10000000000000010100001;
```

: I-Type(ADDi, SUBi, ORi, ANDi)

```
-- ADDi
00100000001 : 00010100000000000000010;
00100000010 : 00000000000000000000011;
00100000011 : 00000000000000000000100;
00100000100 : 00000000001111000000101;
00100000101 : 00000000001111000000110;
00100000110 : 00000000001100110000111;
00100000111 : 10000000000000000000001;
-- SUBi
00110000001 : 00010100000000000000010;
00110000010 : 00000000000000000000011;
00110000011 : 00000000000000000000100;
00110000100 : 00000000001001100000101;
00110000101 : 00000000001001100000110;
00110000110 : 00000000001100110000111;
00110000111 : 10000000000000000000001;
-- ANDi
01000000001 : 00010100000000000000010;
01000000010 : 00000000000000000000011;
01000000011 : 00000000000000000000100;
01000000100 : 00000000000011000010101;
01000000101 : 00000000000011000010110;
01000000110 : 00000000001100110000111;
01000000111 : 10000000000000000000001;
-- ORi
01010000001 : 00010100000000000000010;
01010000010 : 00000000000000000000011;
01010000011 : 00000000000000000000100;
01010000100 : 00000000000111000010101;
01010000101 : 00000000000111000010110;
01010000110 : 00000000001100110000111;
01010000111 : 10000000000000000000001;
```

: I-Type(SB)

```
-- SB
01100000001 : 00010100000000000000010;
01100000010 : 00000000001111000000011;
01100000011 : 00000000001111000000100;
01100000100 : 00000000001111000000101;
01100000101 : 00100000001111000000110;
01100000110 : 00101000001100100000111;
01100000111 : 10000000000000000000001;
```

: I-Type(LB)

برای این دستور خط کد های زیر را داریم که به اختصار توضیح می دهیم اکنون :

```
-- LB:
01110000001 : 00010100000000000000010;
01110000010 : 00000000001111000000011;
01110000011 : 00000000000000000000100;
01110000100 : 00100000000000000000101;
01110000101 : 00010000000000000000110;
01110000110 : 00000010000000010000111;
01110000111 : 00000000001100100001000;
01110001000 : 10000000000000000000001;
```

خط اول : در خط اول سیگنال های IorD, MemWrite برابر با یک میشود و state آن برابر با 0001 خواهد بود.این سیگنال های برای خواندن دستور از رجیستر حاوی PC است.

خط دوم :state بعدی برابر با 0011 است.سیگنال های ALUOp باید 0011 باشد تا عملیات محاسبه آدرسی که باید مقدارش را از ممووری بخوانیم و هم چنین برای اینکه مقدار read data 1 از mux بعد رجیستر A و هم چنین ALUSrcB باید مقدار 1 را انتخاب کند.

خط سوم : این خط صرفا برای این کار است که یک کلاک تاخیر ایجاد کند و صرفا مقدار اسیت می شود 0100.

خط چهارم : مقدار IorD برابر با 1 می شود که این به این خاطر است تا مقدار محاسبه شده در ALUOut به عنوان آدرسی که باید مقدار آن را از ممووری بخواند و در رجیستر مقصد بریزد آماده شود.

خط پنجم : در این خط سیگنال MemRead برابر با 1 شده که یعنی ما مقدار خانه آدرس محاسبه شده در مرحله قبل را میخوانیم در این مرحله.

خط ششم : سیگنال MemtoReg برابر با 1 شده که یعنی میخواهیم مقداری از بلوک memory data register در قسمت write data قرار داشته باشد و هم چنین سیگنال RegWrite هم برابر با یک شده تا آن داده را در رجیستر فایل بنویسد.

و در دو خط آخر هم مقدار PC + 1 را محاسبه کرده و با یک کرددن سیگنال PCWrite مقدار PC را آپدیت می کنیم.

: I-Type(BEQ, BNE)

```
-- BEQ
10000000001 : 00010100000000000000010;
10000000010 : 00000000000000000000011;
10000000011 : 00000000000000000000100;
10000000100 : 00000000001101000000101;
10000000101 : 01000000001101000000110;
10000000110 : 01100000101101000000111;
10000000111 : 01110000100000000001000;
10000001000 : 00000000000000000000001;
-- BNE
10010000001 : 00010100000000000000010;
10010000010 : 00000000000000000000011;
10010000011 : 00000000000000000000100;
10010000100 : 00000000001101000000101;
10010000101 : 01000000001101000000110;
10010000110 : 01100000101111000000111;
10010000111 : 01110000100000000001000;
10010001000 : 00000000000000000000001;
```

: J

```
-- J
11100000001 : 00010100000000000000010;
11100000010 : 10000001000000000000001;
```

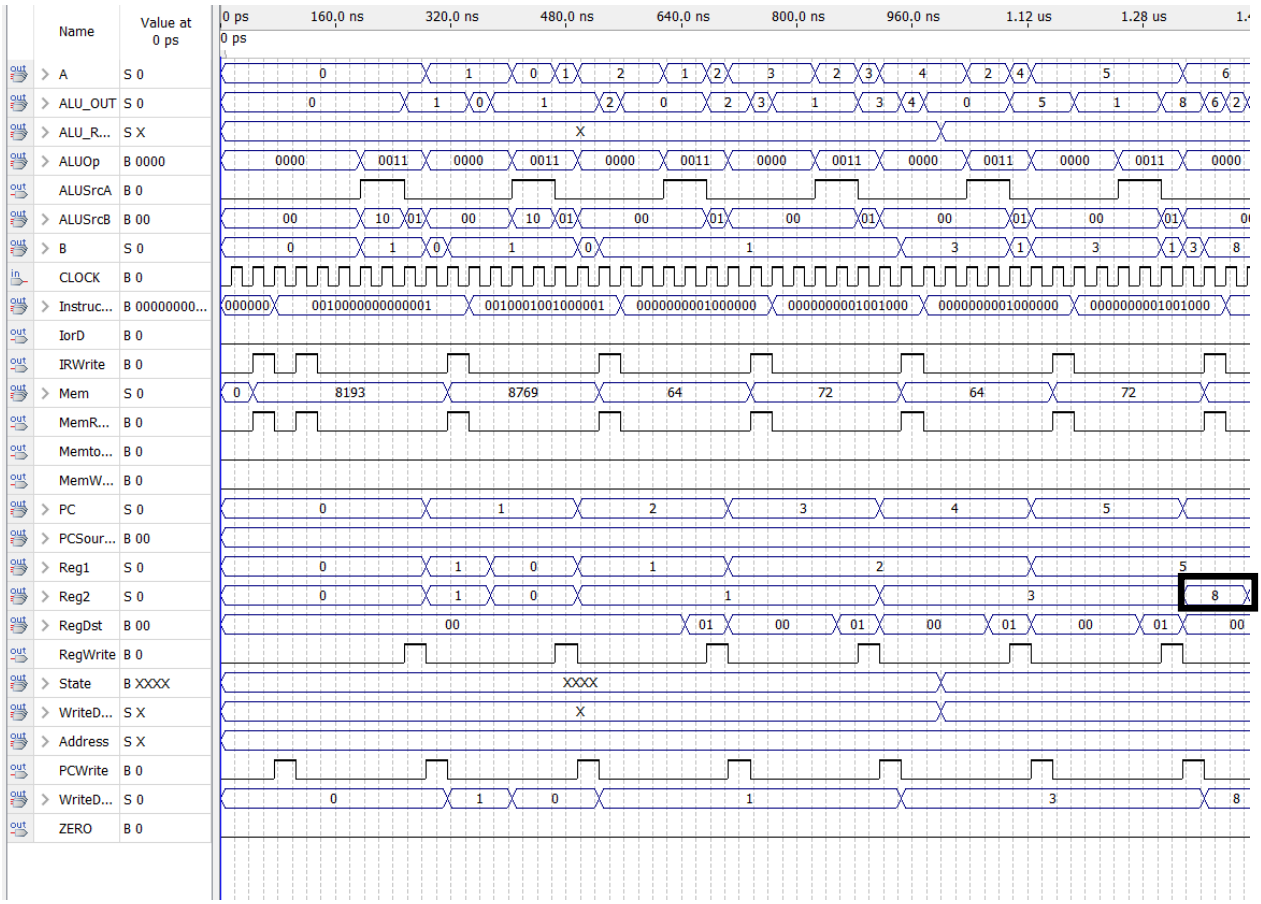
: JAL

```
-- JAL
11110000001 : 00010100000110010000010;
11110000010 : 00000000000110010000011;
11110000011 : 00000000000110010100100;
11110000100 : 00000000000110010100101;
11110000101 : 00000000000110010100110;
11110000110 : 00000000000110010100111;
11110000111 : 00000000000110010101000;
11110001000 : 00000000000110011001001;
11110001001 : 10000001000000000000001;
```

تست ها :

فیبوناچی با ورودی 5 :

برنامه مربوطه در فایل fibo.mif ذخیره شده است:



که در ویو فرم عدد 8 را مشاهده می کنیم.

تست دلخواه جدید:

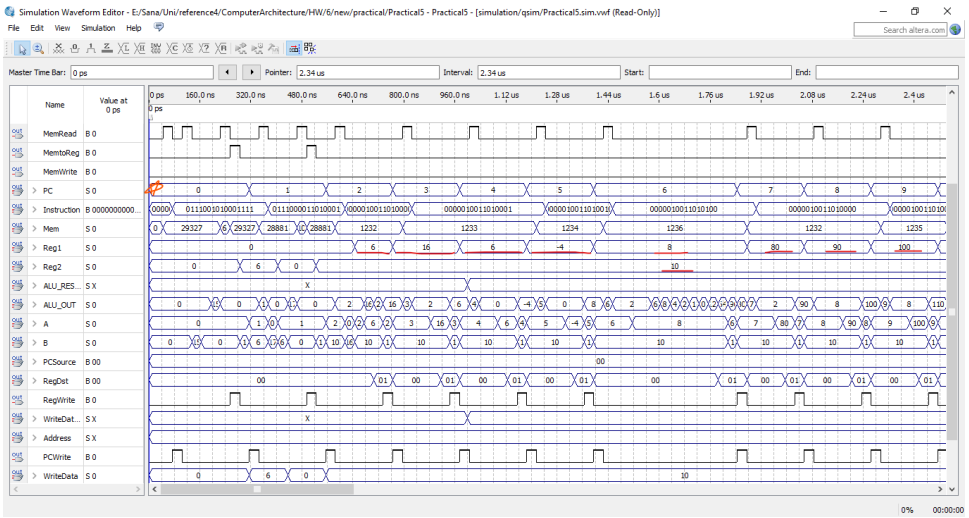
در زیر تست دلخواه جدیدی قرار می‌گیرد که کد اسمبلی متناظر با آن و خروجی مطلوب نوشته شده است:

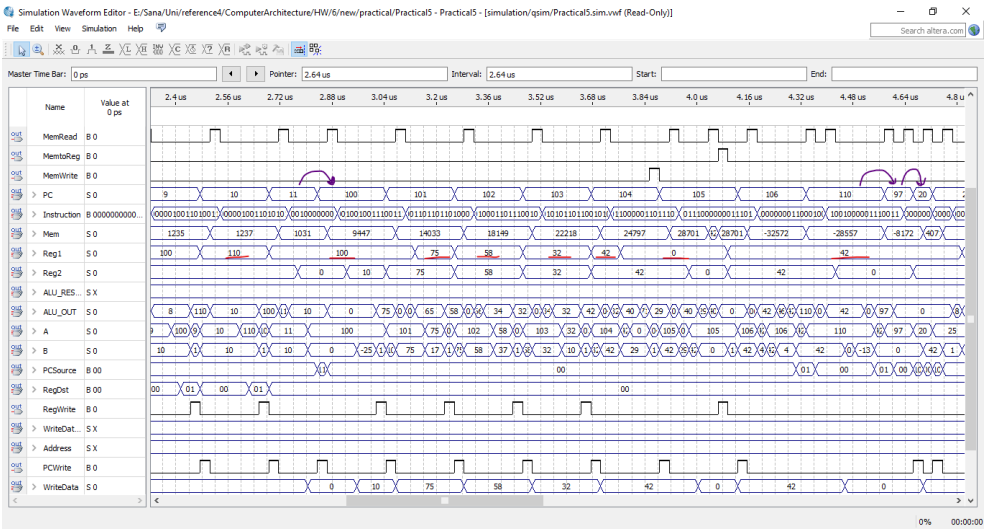
```
DEPTH = 256;
WIDTH = 16;

ADDRESS_RADIX = DEC;
DATA_RADIX = BIN;

CONTENT BEGIN
0 : 0111001010001111; -- LB r2, 15(r1) --> r2 = 6
1 : 0111000011010001; -- LB r3, 17(r0) --> r3 = 10
2 : 0000010011010000; -- ADD r2, r2, r3 --> r2 = 16
3 : 0000010011010001; -- SUB r2, r2, r3 --> r2 = 6
4 : 0000010011010001; -- SUB r2, r2, r3 --> r2 = -4
5 : 0000010011010010; -- AND r2, r2, r3 --> r2 = 8
6 : 0000010011010100; -- MULT r2, r2, r3 --> r2 = 80
7 : 0000010011010000; -- ADD r2, r2, r3 --> r2 = 90
8 : 0000010011010000; -- ADD r2, r2, r3 --> r2 = 100
9 : 0000010011010011; -- OR r2, r2, r3 --> r2 = 110
10 : 0000010011010101; -- XOR r2, r2, r3 --> r2 = 100
11 : 0000010000000111; -- JR r2 --> Go to 100
15 : 0000000000000110;
17 : 0000000000001010;
20 : 1111000000011001; -- JAL 25 --> Go to 25, r7 = 21
25 : 0000111000111000; -- ADD r7, r0, r7 --> r7 = 21
97 : 1110000000010100; -- J 20 --> Go to 20
100 : 0010010011100111; -- ADDi r3, r2, -25 --> r3 = 75
101 : 0011011011010001; -- SUBi r3, r3, 17 --> r3 = 58
102 : 0100011011100101; -- ANDi r3, r3, 37 --> r3 = 00111010 & 00100101 = 00100000 = 32
103 : 0101011011001010; -- ORi r3, r3, 10 --> r3 = 00100000 | 00001010 = 00101010 = 42
104 : 0110000011011101; -- SB r3, 29(r0) --> Mem[29] = 42
105 : 0111000000011101; -- LB r0, 29(r0) --> r0 = 42
106 : 1000000011000100; -- BEQ r0, r3, 4 --> Go to 110
110 : 1001000001110011; -- BNE r0, r1, -13 --> Go to 97
END;
```

که در ویو فرم می توان مقادیر مطلوب را مشاهده کرد :





محاسبه فاکتوریل عدد 5 :

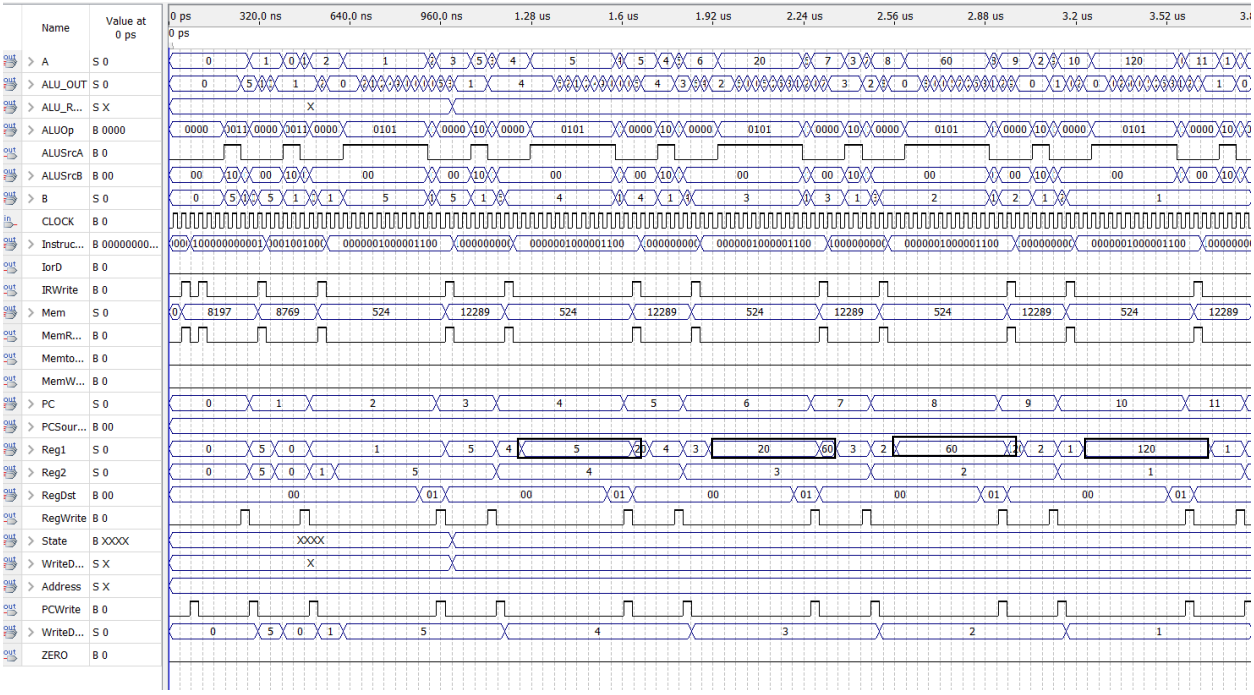
کد داده شده برای محاسبه فاکتوریل در فایل fact.mif موجود است.

```
DEPTH = 256;
WIDTH = 16;

ADDRESS_RADIX = DEC;
DATA_RADIX = BIN;

CONTENT BEGIN
0 : 0010000000000101; -- ADDi r0, r0, 5 --> r0 = 5
1 : 0010001001000001; -- ADDi r1, r1, 1 --> r1 = 1
2 : 0000001000001100; -- MULT r1, r1, r0 --> r1 = 5
3 : 0011000000000001; -- SUBi r0, r0, 1 --> r0 = 4
4 : 0000001000001100; -- MULT r1, r1, r0 --> r1 = 20
5 : 0011000000000001; -- SUBi r0, r0, 1 --> r0 = 3
6 : 0000001000001100; -- MULT r1, r1, r0 --> r1 = 60
7 : 0011000000000001; -- SUBi r0, r0, 1 --> r0 = 2
8 : 0000001000001100; -- MULT r1, r1, r0 --> r1 = 120
9 : 0011000000000001; -- SUBi r0, r0, 1 --> r0 = 1
10 : 0000001000001100; -- MULT r1, r1, r0 --> r1 = 120
11 : 0011000000000001; -- SUBi r0, r0, 1 --> r0 = 0
12 : 0110000001000011; -- SB r0, 3(r1) --> Mem[3] = 120
END;
```

و می توان عدد 120 را در ویو فرم مشاهده کرد :



دو تست دلخواه مربوط به تمرین قبلی نیز در فایل‌های insts2 و insts3 در فولدر پروژه موجود هستند.