دانشگاه صنعتی شریف
دانشکده مهندسی کامپیوتر

عنوان:

طراحی و پیاده‌سازی پروتکل ارتباطی سریال شبهUART برای پردازنده‌ی MIPS در محیط Logisim

Design and Implementation of a UART-Inspired Serial Communication Protocol for a MIPS Processor in Logisim Environment

نگارش

نرگس کاری، ثنا نیرومند، یاسمین رجبی ریسه و حسنا شاه حیدری

استاد درس

دکتر حسین اسدی

دستیار آموزشی مسئول پروژه

مهدی بهرامیان

تابستان ۱۴۰۴

چکیده:در این پروژه، یک پروتکل ارتباطی سریال ساده با الهام از پروتکل UART طراحی و پیاده‌سازی شده است. این پیاده‌سازی در بستر یک پردازنده‌ی MIPS تک‌چرخه‌ای و با استفاده از محیط شبیه‌سازی Logisim انجام گرفته است. هدف اصلی، ایجاد پلی میان دنیای سخت‌افزار و نرم‌افزار بوده است، به‌گونه‌ای که پردازنده بتواند با دستگاه‌های ورودی/خروجی نظیر صفحه‌کلید، نمایشگر متنی TTY و هاLED ارتباط برقرار کند. برای تحقق این هدف، مجموعه‌ای از دستورهای سفارشی (Instructions Custom) به فرمت I-type طراحی و به معماری پردازنده افزوده شده‌اند تا امکان ارسال و دریافت داده به/از دستگاه‌های جانبی فراهم شود. پروتکل طراحی‌شده شامل بسته‌های داده‌ای با ساختار مشخص (شامل بیت شروع، آدرس مقصد و داده) است که از طریق رجیسترهای شیفت‌دهنده به صورت سریال و از طریق bus تبادل می‌شوند. در طراحی این پروتکل از مفهوم بیت‌های آغاز و پایان که در UART وجود دارد الهام گرفته شده تا انتقال داده به صورت قابل اطمینان انجام شود. کارایی سیستم طراحی‌شده با چهار برنامه‌ی تستی شامل کنترل LED، چاپ کاراکتر روی TTY، خواندن رشته از کیبورد و نمایش دنباله فیبوناچی ارزیابی شده و عملکرد صحیح آن در تبادل داده‌ها تأیید شده است.

واژه‌های کلیدی: پردازنده MIPS، ارتباط سریال، UART، رابط ورودی/خروجی، Logisim، رجیستر شیفت‌دهنده، سیستم‌های نهفته، معماری تک‌چرخه، دستور سفارشی، صفحه‌نمایش TTY، ورودی کیبورد

۱ مقدمه

۱-۱ تعریف مساله

در پردازنده‌ها، برای برقراری ارتباط با دنیای بیرون نیاز به استفاده از رابط‌های ورودی/خروجی (I/O) وجود دارد. این رابط‌ها امکان تبادل داده بین پردازنده و دستگاه‌های جانبی مانند صفحه‌کلید، نمایشگر، حافظه خارجی و غیره را فراهم می‌کنند. یکی از رایج‌ترین روش‌ها برای انتقال داده، استفاده از پروتکل‌های سریال مانند UART است. در این پروژه، با الهام از پروتکل UART، یک پروتکل ساده و سفارشی طراحی شده است که بر بستر معماری MIPS پیاده‌سازی شده و هدف آن ایجاد یک بستر ارتباطی بین رجیسترهای داخلی پردازنده و دستگاه‌های جانبی (مانند LED و صفحه‌نمایش TTY) است.

۱-۲ اهمیت موضوع

درک عمیق از نحوه طراحی و پیاده‌سازی پروتکل‌های ارتباطی، یکی از مباحث کلیدی در معماری کامپیوتر و طراحی سیستم‌های نهفته است. این پروژه با فراهم آوردن بستری برای تعامل میان سخت‌افزار و نرم‌افزار، زمینه مناسبی برای درک بهتر مفاهیمی مانند طراحی پردازنده، نحوه انتقال داده، طراحی رجیسترهای کنترلی و کاربرد پروتکل‌های سریال فراهم می‌سازد. پیاده‌سازی این سیستم در محیط‌هایی مانند Logisim همچنین دانش عملی دانشجو را در زمینه طراحی دیجیتال افزایش می‌دهد.

۱-۳ ادبیات موضوع

پروتکل UART یکی از پرکاربردترین پروتکل‌های ارتباط سریال است که در بسیاری از سیستم‌های نهفته، دستگاه‌های الکترونیکی و پردازنده‌ها برای ارسال و دریافت داده استفاده می‌شود. در UART، داده‌ها به صورت سریال و با استفاده از رجیسترهای انتقال و دریافت (TX و RX) بین دو دستگاه تبادل می‌شوند. پروژه حاضر با ساده‌سازی و شبیه‌سازی این پروتکل در محیط آموزشی و با استفاده از یک پردازنده ساده تک‌چرخه‌ای (MIPS single-cycle) گامی در جهت آموزش عملی مفاهیم پایه‌ای معماری و ارتباطات برداشته است.

۱-۴ اهداف پروژه

اهداف اصلی این پروژه عبارت‌اند از:

طراحی و پیاده‌سازی یک ماژول ارتباطی سریال ساده بر پایه پروتکل UART

شبیه‌سازی پروتکل پیاده‌سازی‌شده در محیط Logisim

برقراری ارتباط بین رجیسترهای داخلی پردازنده و دستگاه‌های جانبی مانند LED و TTY

افزایش توانمندی دانشجو در درک مفاهیم پایه‌ای معماری پردازنده و ارتباط با دنیای خارج

۲ مفاهیم اولیه

۲-۱ پردازنده MIPS

پردازنده مورد استفاده در این پروژه از نوع single-cycle است و قابلیت اجرای دستورات پایه را دارد که در تمرینات عملی درس پیاده‌سازی شده و تست شده بود. برای ارتباط با دستگاه‌های جانبی، دستورات I/O اختصاصی به مانند فرمت I-type اضافه شده‌اند. اکنون به اختصار به شرح دستوراتی که در پردازنده پشتیبانی میشوند میپردازیم:

Instruction	Type	Opcode	Funct	Notes
`add`	R-Type	`000000`	`100000`	`rd = rs + rt`
`addi`	I-Type	`001000`	—	`rt = rs + imm` (sign-extended)
`sub`	R-Type	`000000`	`100010`	`rd = rs - rt`
`or`	R-Type	`000000`	`100101`	`rd = rs | rt` (bitwise OR)
`and`	R-Type	`000000`	`100100`	`rd = rs & rt` (bitwise AND)
`xor`	R-Type	`000000`	`100110`	`rd = rs ^ rt` (bitwise XOR)
`sll`	R-Type	`000000`	`000100`	`rd = rs << rt` (logical shift left)
`srl`	R-Type	`000000`	`000110`	`rd = rs >> rt` (logical shift right)
`sra`	R-Type	`000000`	`000111`	`rd = rs >>> rt` (arithmetic shift right, sign-extended)

Instruction	Type	Opcode	Funct	Notes
`div`	R-Type	`000000`	`011010`	`hi = rs % rt; lo = rs / rt`
`sll`	R-Type	`000000`	`000000`	`rd = rt << shamt` (logical shift left)
`mfhi`	R-Type	`000000`	`010000`	`rd = hi`
`mflo`	R-Type	`000000`	`010010`	`rd = lo`
`sw`	I-Type	`101011`	—	`*(int*)(offset+rs)=rt` (sign-extended)
`lw`	I-Type	`100011`	—	`rt=*(int*)(offset+rs)` (sign-extended)
`bne`	I-Type	`000101`	—	`if(rs!=rt) pc+=offset` (sign-extended)
`slti`	I-Type	`001010`	—	`rt=rs<imm` (sign-extended)
`jmp`	I-Type	`000010`	—	`pc=target`

Instruction	Type	Opcode	Funct	Notes
`mul`	R-Type	`011100`	`000010`	`rd = rs * rt (signed-multiply)`
`jr`	R-Type	`000000`	`001000`	`pc = rs`
`jal`	J-Type	`000011`	—	`r31(ra) = pc+1 & pc=addr`
`beq`	I-Type	`000100`	—	`if(rs==rt) pc+=offset` (sign-extended)

۲-۱-۱ دستورات ویژه I/O

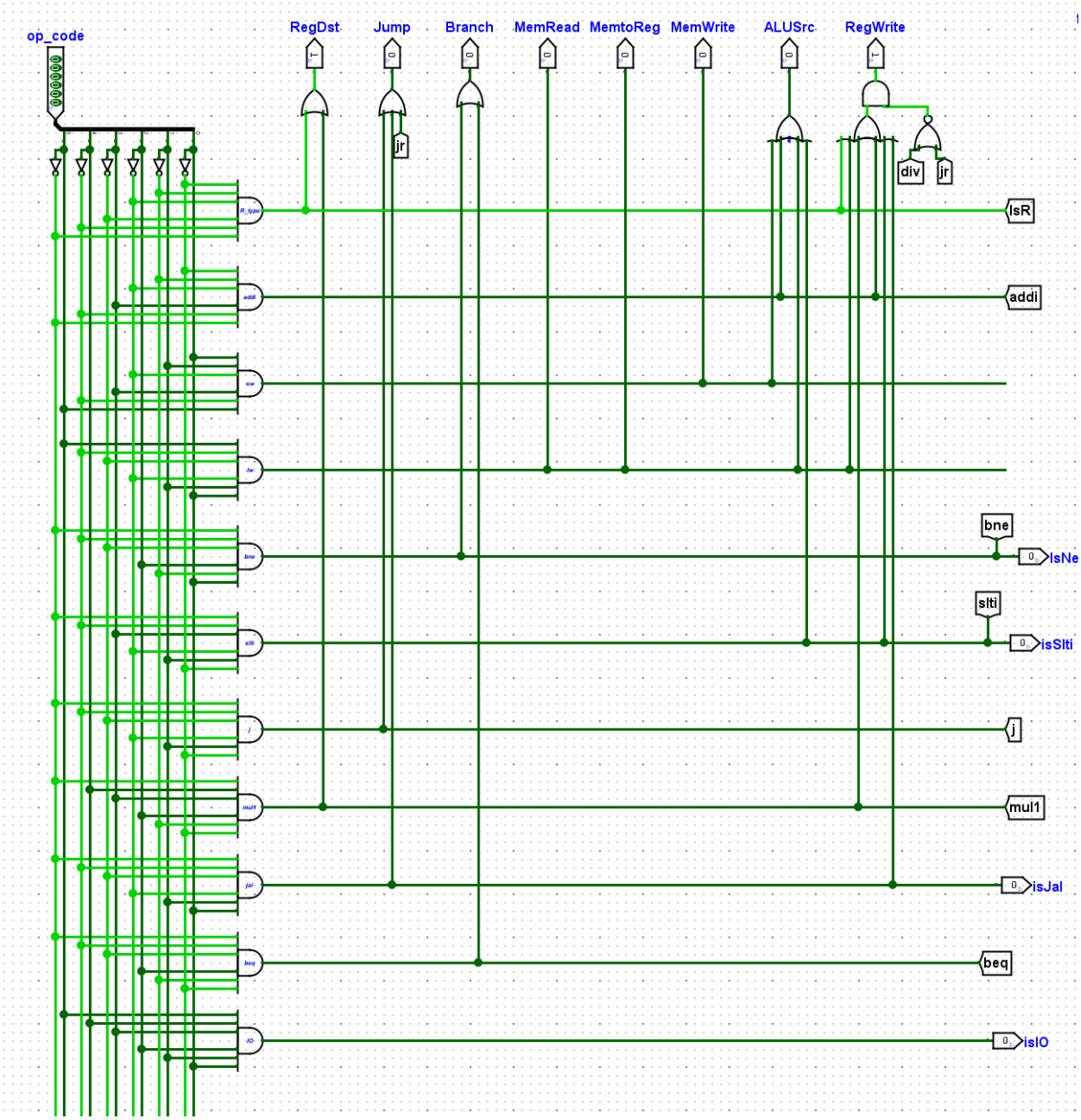
- led $rt
- tty $rt
- kb $rt

۲-۱-۲ فرمت دستورات I/O

- **Opcode**: مقدار 111111 برای فعال‌سازی سیگنال isIO
- **آدرس دستگاه**: ۵ بیت بعدی که مقصد دستور را مشخص می‌کند:
  - 0 تا 6: آدرس هاLED
  - 7: آدرس TTY (صفحه نمایش)
  - 8: آدرس کیبورد
  - 11: علاوه بر این دستورات، با آدرس ۱۱۱۱ می‌توان عدد موجود در رجیستر مشخص شده را به صورت کامل با همه‌ی LED ها نمایش داد.
- **رجیستر داده**: ۵ بیت بعدی مشخص‌کننده رجیستری است که داده در آن ذخیره یا از آن خوانده می‌شود.در لامپ‌های LED علاوه بر آدرس لامپ نیاز به داده‌ی صفر یا یک در جایگاه i ام عدد داریم که بگوید لامپ i ام روشن یا خاموش شود. در صفحه‌ی نمایشگر TTY نیاز به کد ASCII هفت بیتی یک کاراکتر داریم تا روی صفحه‌ی نمایشگر چاپ شود. برای این منظور یک رجیستر باید مشخص کنیم در دستور ورودی تا این داده در آن ذخیره شود. در نتیجه شماره‌ی این رجیستر را در پنج بیت بعدی ذخیره می‌کنیم.

۲-۱-۳ واحد کنترلی

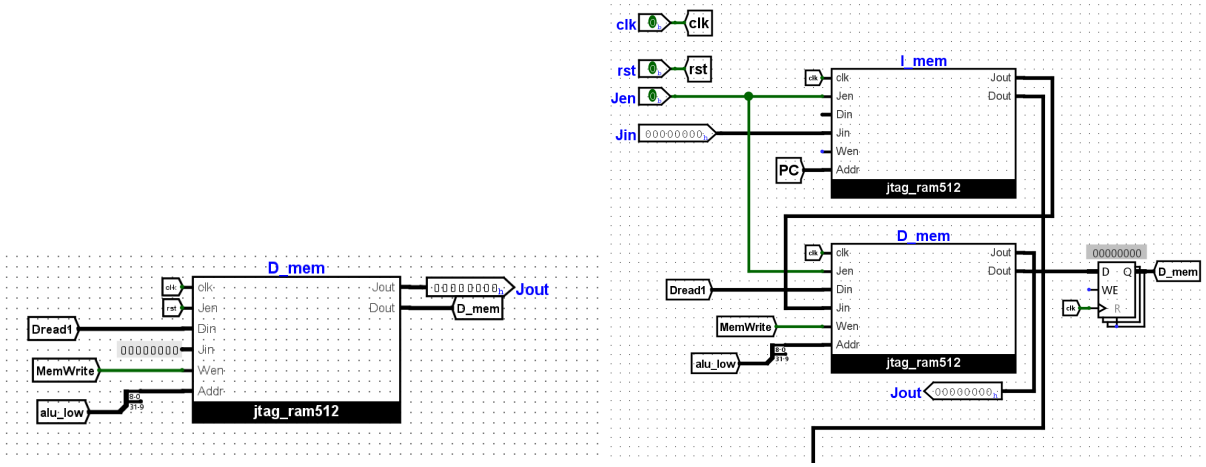
در اینجا دستور I/O تشخیص داده شده و سیگنال آن در واحد کنترلی تولید می‌شود.



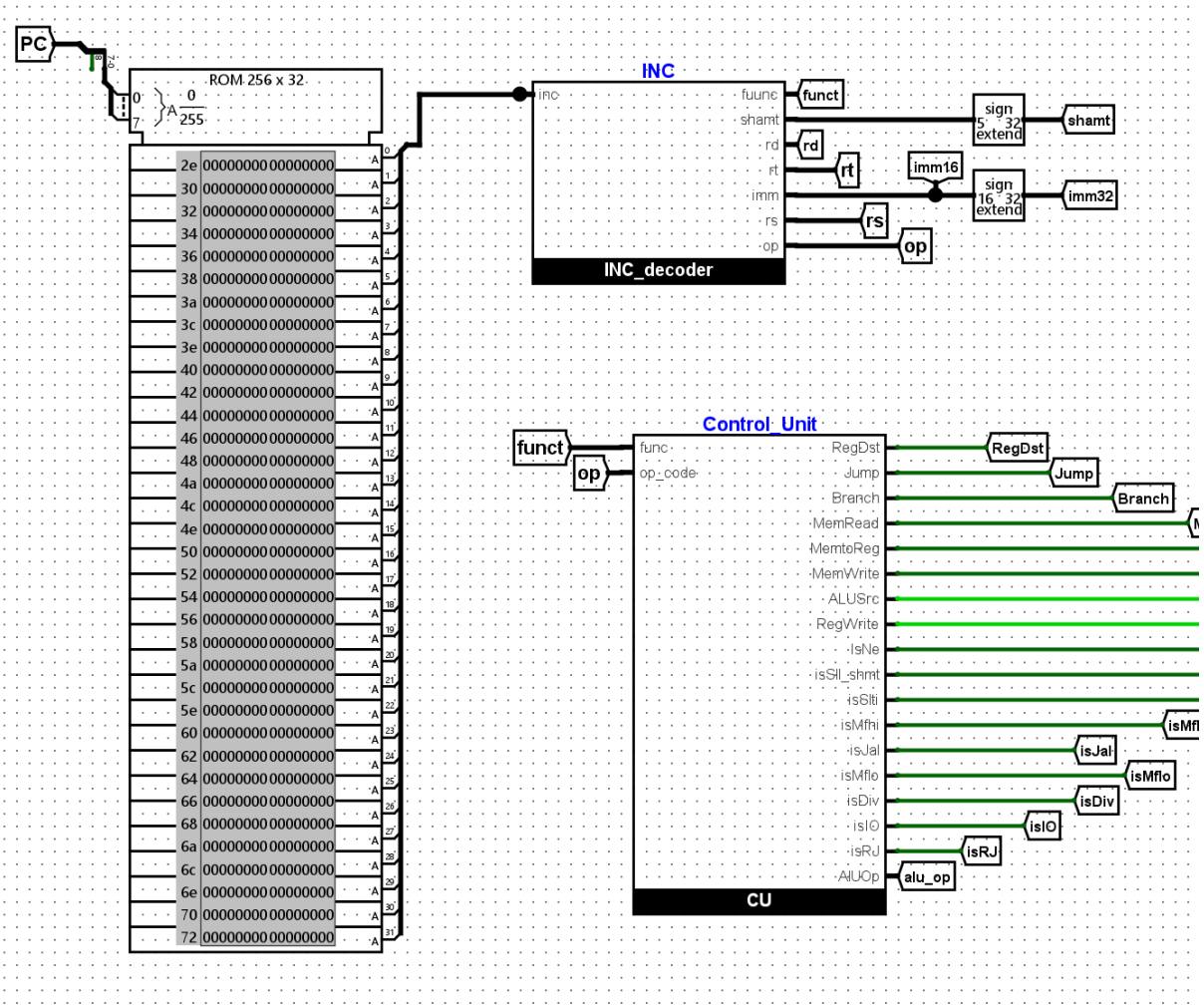
۲-۱-۴ واحد حافظه

لازم است که تغییرات اندکی به فرمت حافظه در پروژه‌ی خود بدهیم تا بتوانیم دستورات خودمان را در حافظه ذخیره کنیم و به عنوان ورودی به برنامه بدهیم. در تمرین‌های عملی از فرمت JTAG استفاده شده بود تا حافظه‌ی دستور و داده توسط برنامه‌ی judge پر شوند. از آنجایی که در این پروژه دستورات را دستی وارد می‌کنیم نیازی به این ارتباطات نیست و در حافظه‌ی D-mem کافی‌است

فقط این اتصالات را مطابق تصویر حذف کنیم.

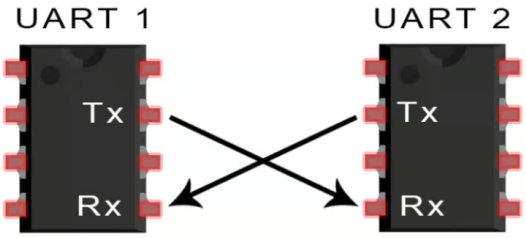


به جای I-mem هم برای اینکه خودمان در برنامه دستورات را دستی وارد کنیم یک ROM می‌گذاریم که از بلوک‌های ۳۲ بیتی تشکیل شده و هنگام سیموله کردن برنامه درون آن را با برنامه‌ی مدنظر پر می‌کنیم.

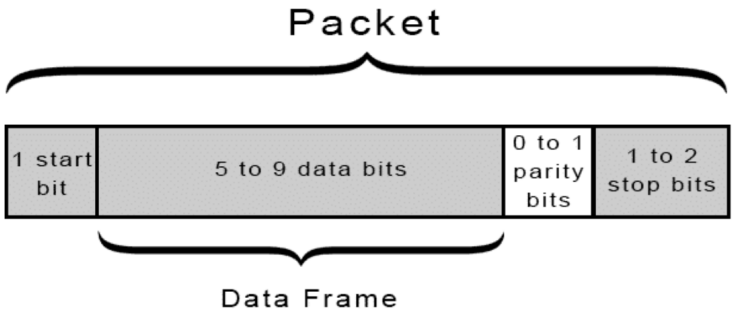


۲-۲ پروتکل UART

بعد از این تغییرات نوبت به رد و بدل کردن اطلاعات با I/O و انجام عملیات مدنظر روی آن‌ها می‌باشد. به این صورت که طبق خواسته‌ی صورت پروژه نمی‌توانیم مستقیم داده‌ی خوانده شده از رجیستر یا حتی آدرس دستگاه جانبی مورد هدف را با یک سیم به سمت I/O بفرستیم و بایستی یک پروتکل سریال برای انتقال اطلاعات در نظر بگیریم. برای این منظور، ما یک پروتکل شبه UART پیاده سازی کرده‌ایم. پروتکل UART مطابق تصویر شامل دو سیم برای تبادل داده می‌باشد که با استفاده از هر دو سمت می‌توان اطلاعات گرفت و دریافت کرد.



نکته‌ی مهم در مورد پروتکل UART این است که آسنکرون عمل می‌کند و با استفاده از کلاک داده را ارسال نمی‌کند. در نتیجه برای انجام این پیاده سازی از دو مکانیزم استفاده می‌کند: یکی تنظیم rate baud برای ارسال داده با سرعت مشخص و از پیش تعیین شده و دیگری استفاده از بیت start و stop در ابتدا و انتهای داده و ارسال کل این بسته‌بندی به سمت گیرنده. نکته‌ی مهم برای ما در مورد این پروتکل مورد دوم یا همان نحوه‌ی ساختن packet و بسته‌بندی کردن داده با استفاده از بیت شروع و پایان است.



۱-۲-۲ packet

ما از این ایده استفاده کرده‌ایم و برای مشخص شدن شروع ارسال اول هر داده بیت برابر با یک ارسال می‌کنیم.
با توجه به توضیحات داده شده مربوط به UART پروژه‌ی ما به همان‌گونه پیاده‌سازی شده است.

با این تفاوت که اندازه‌ی هر بسته‌بندی داده یا همان packet مشخص است. هر packet ما شامل ۴ بیت آدرس I/O مقصد و ۷ بیت داده‌ایست که باید به آن داده شود و آن را از رجیستر موجود در دستور ورودی خوانده‌ایم. ابتدای packet هم یک بیت برابر با عدد یک برای نشان دادن پایان فعالیت می‌گذاریم.

۲-۲-۲ ماژول فرستنده و گیرنده داده

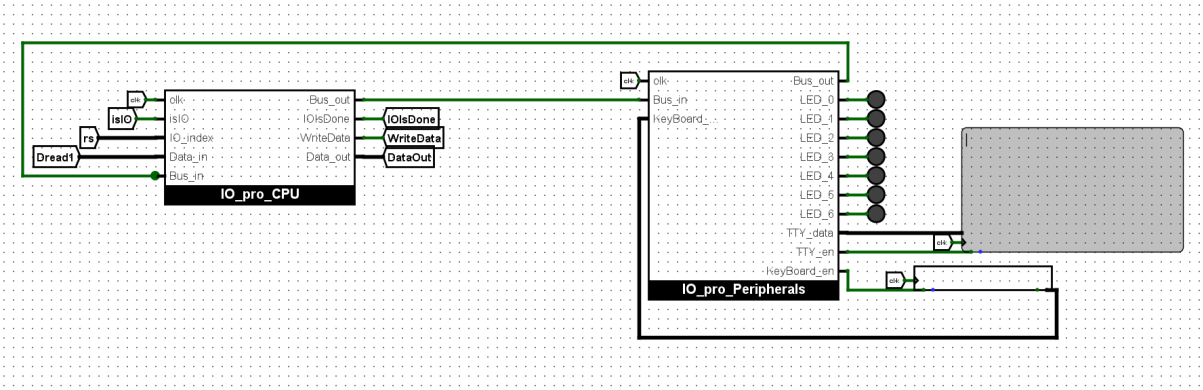
مورد بعدی که در مورد این پروتکل در پروژه‌ی ما تاثیر گذاشته load موازی داده‌ها در هر ماژول فرستنده‌ی پروتکل است که بعد داده را بیت به بیت جدا می‌کند و روی bus به گیرنده ارسال می‌کند و بعد از دریافت، دوباره در ماژول گیرنده داده به صورت موازی به خارج پروتکل فرستاده می‌شود. هر ماژول فرستنده یا گیرنده از تعداد برابر بیت‌های هر packet یا همان ۱۲ رجیستر تشکیل شده که با هم یک شیفت رجیستر را تشکیل می‌دهند. این شیفت رجیستر از مکانیزم input parallel output parallel (PIPO) برای ورود و خروج داده از پروتکل استفاده می‌کند. ابتدا داده به صورت موازی داخل همه‌ی رجیسترها ریخته می‌شود. سپس آنقدر بیت به بیت این داده شیفت پیدا می‌کنند و از طریق bus به گیرنده می‌رسند تا کل داده به گیرنده رسیده باشد. نکته‌ی مهم این است که از کجا متوجه می‌شویم که انتقال کامل انجام شده؟ جوابش دقیقا با همان ایده‌ی بیت start و stop در پروتکل UART می‌باشد. اما در اینجا چون تعداد بیت packet یک اندازه‌ی ثابت و مشخص است کافی‌است از فقط یک بیت نشانگر پایان فعالیت استفاده کنیم و پروتکل هم با روشن کردن سیگنال load با استفاده از سیگنال isIO تولید شده از واحد کنترلی روشن می‌شود. در کنار ۱۱ بیت داده برای انتقال، یک بیت در ابتدای آن را با شروع فعالیت پروتکل یک می‌کنیم و آنقدر شیفت می‌دهیم تا این بیت به اولین رجیستر در سمت گیرنده برسد که خود نشان‌دهنده‌ی پایان انتقال است.

۲-۳ چالش ها

یک چالشی که در پیاده‌سازی پروتکل‌ها از اهمیت بسیاری برخوردار است این است که نباید به اشتباه هیچ سیگنالی در دو طرف به جز از طریق bus پروتکل با هم در ارتباط باشند. به عنوان مثال در اینجا که از پروتکل برای ارتباط پردازنده و دستگاه‌های I/O استفاده کرده‌ایم نباید هیچ یک از سیگنال‌های پردازنده مثل isIO در سمت I/O به کار گرفته شوند مگر این که از طریق bus ارسال شوند. اشتباهی که در ابتدای پیاده‌سازی کردیم این بود که بیت شروع نداشتیم و با استفاده از counter و شمردن تعداد شیفت‌های انجام شده انتهای انتقال را بررسی می‌کردیم و برای شروع شمارنده هم از سیگنال

isIO استفاده کردیم بدون گذر از bus چرا که شمارنده در سمت I/O کار می‌کرد و پایان فعالیت را در ماژول گیرنده بررسی می‌کرد اما این سیگنال در سمت پردازنده تولید شده بود. به این منظور شد که از بیت نشانگر پایان فعالیت موجود در UART ایده گرفتیم و مکانیزم پروتکل خود را بر مبنای آن پیاده‌سازی کردیم.

۲-۳-۱ شمای کل و ماژول بندی

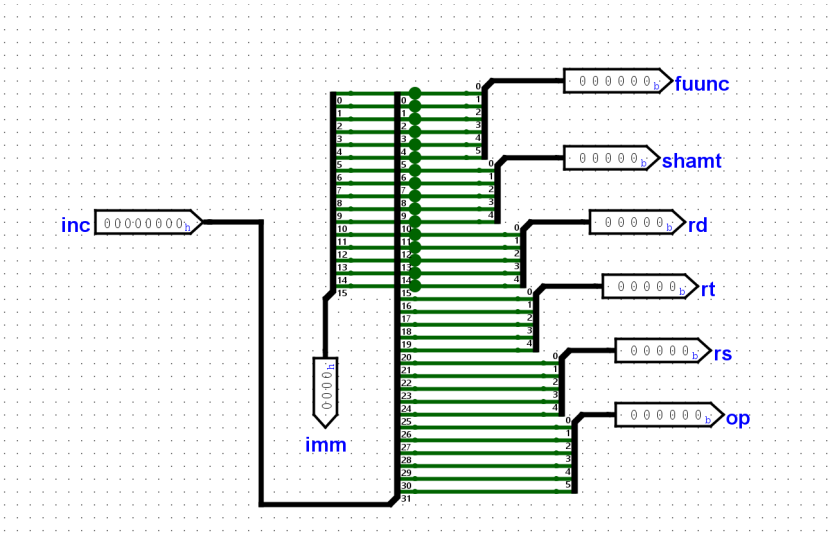


این شکل شمای کلی و ماژول‌بندی شده‌ی پروتکل را نمایش می‌دهد. به این صورت که به ماژول فرستنده یا همان سمت CPU سیگنال فعال شدن isIO داده می‌شود و با آن کار خود را آغاز می‌کند. علاوه بر آن آدرس مقصد داده شده که طبق فرمت دستورات I-type در پردازنده‌ی MIPS و با اطلاع به اینکه در ۵ بیت بعد از opcode قرار می‌گیرد می‌توان آن را از تونلی به نام rs گرفت که در پیاده‌سازی خود انجام داده داده‌ایم و متصل به پنج بیت مدنظر در دستور است.

MIPS I-Type Instruction Format

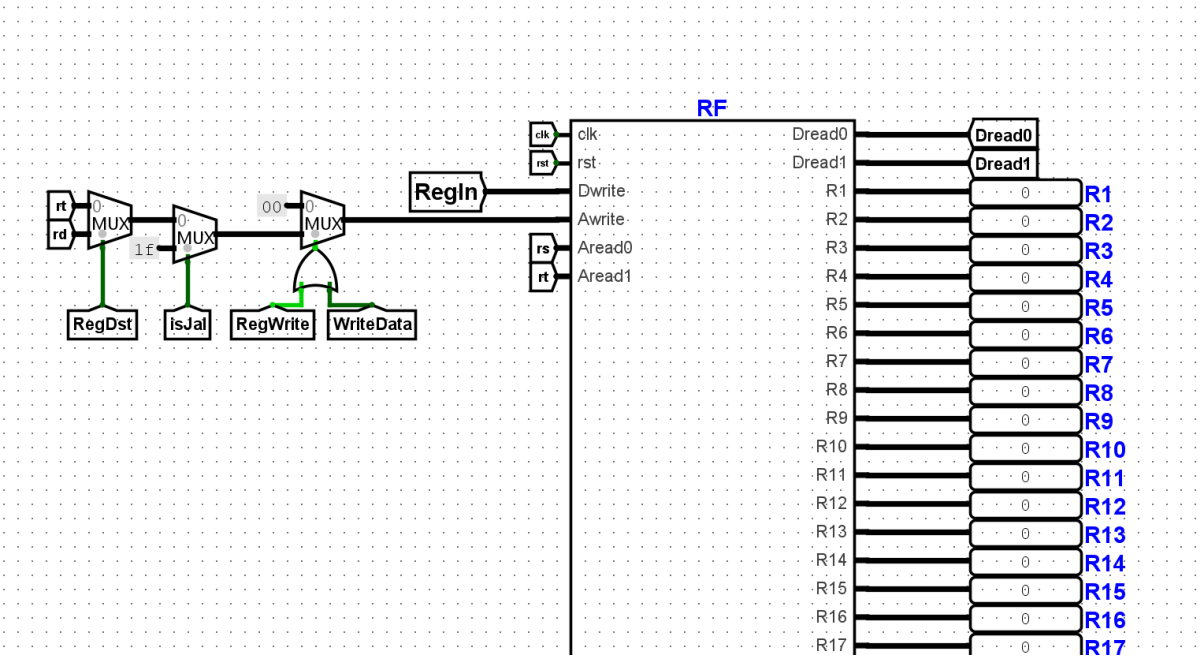
Bit Range	Field Name	Width (bits)	Description
31 – 26	`opcode`	6	Operation code (determines instruction type).
25 – 21	`rs`	5	Source register operand.
20 – 16	`rt`	5	Destination/target register operand.
15 – 0	`immediate`	16	Immediate value or offset.

```
 31  26|25  21|20  16|15              0
+------+------+------+------------------+
|opcode|  rs  |  rt  |     immediate    |
+------+------+------+------------------+
```



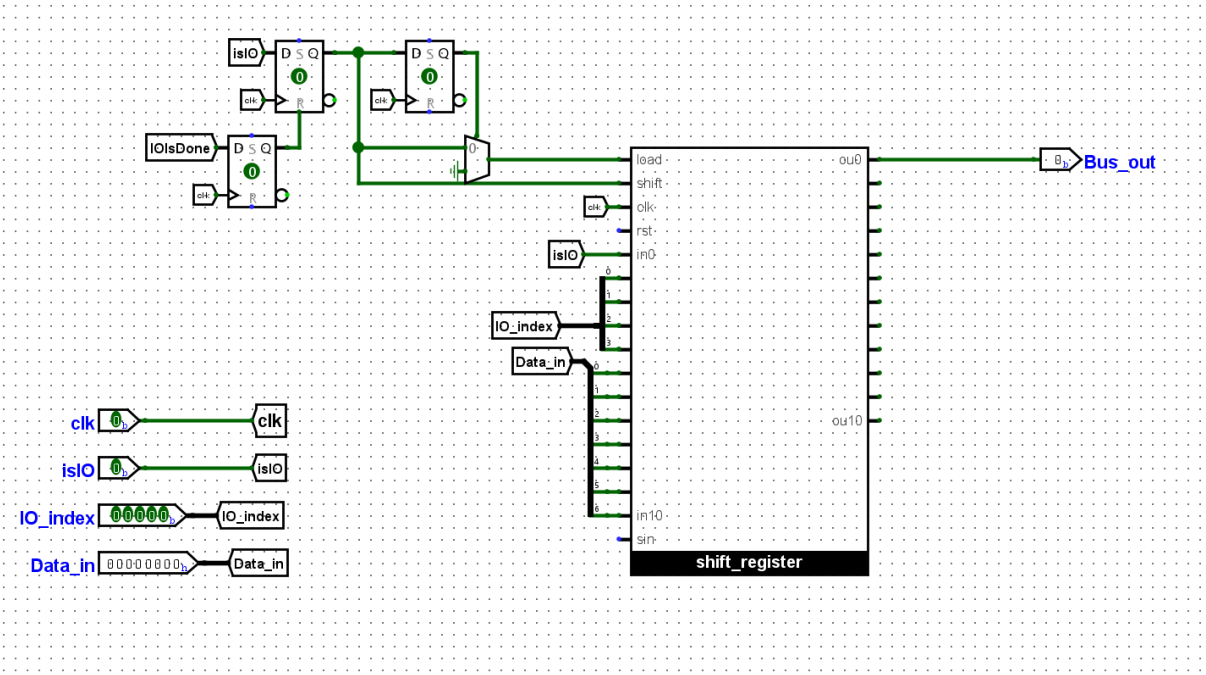
۲-۳-۲ انتقال داده به I/O با استفاده از ماژول فرستنده

داده‌ی مورد نیاز در سمت I/O را هم از لیبل Dread۱ در شکل زیر می‌گیریم چرا که همچنان طبق گفته‌های قبلمان آدرس رجیستری که داده‌مان در آن قرار دارد از پنج بیت بعد از rs می‌آید که در پیاده‌سازی ما یکسان با لیبل rt است که رجیستر فایل بعد از خواندن داده‌ی موجود در رجیستر rt آن را در Dread۱ قرار می‌دهد.



در خروجی هم bus داریم که محل اتصال دو ماژول فرستنده و گیرنده است. دقت کنید که فعلا در حال توضیح بخش مربوط به ارسال داده به I/O هستیم. می‌دانیم که در هنگام کار با کیبورد نیاز داریم از I/O داده دریافت کنیم که بخش‌های ورودی و خروجی دیگر روی ماژول که فعلا توضیح داده نمی‌شوند مربوط به این عملیات هستند. در حال حاضر روی ارسال داده به I/O تمرکز می‌کنیم.

در سمت ماژول گیرنده هم که خروجی‌های فعال کردن یا نکردن I/O های مقصد و داده‌های ارسالی به آن‌ها مشخص هستند.

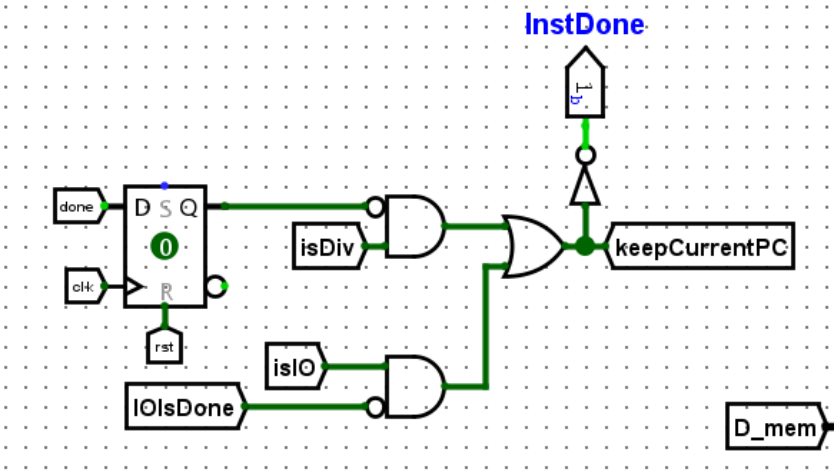


شکل زیر پیاده‌سازی‌های سمت ماژول فرستنده را نمایش می‌دهد. در ورودی load مکانیزمی برای اینکه فقط یک کلاک روشن باشد و ورودی موازی بگیرد و بعد خاموش شود صورت گرفته است. به این شکل که از رجیستر برای ذخیره یک مقدار در هر کلاک استفاده می‌کنیم به این صورت که اگر در کلاک قبل isIO روشن بوده ورودی صفر خواهد بود (در کلاک قبلی داده گرفته شده و load خاموش می‌شود) در غیر این حالت ورودی از isIO می‌آید. برای اینکه در هنگام اجرای دو دستور I/O پشت هم مشکل ایجاد نشود یک رجیستر دیگر هم این وسط اضافه می‌کنیم تا با یک کلاک اضافی در آن وسط مشکلات تغییر isIO حل شود. برای اتمام دستور isIO هم یک سیگنال در نظر گرفته شده که با انجام آن PC افزایش پیدا می‌کند وگرنه در قبل از آن چون سینگل سایکل داریم با این همه کلاکی که خورده شده PC بارها افزایش و به دستور بعد رفته بود.

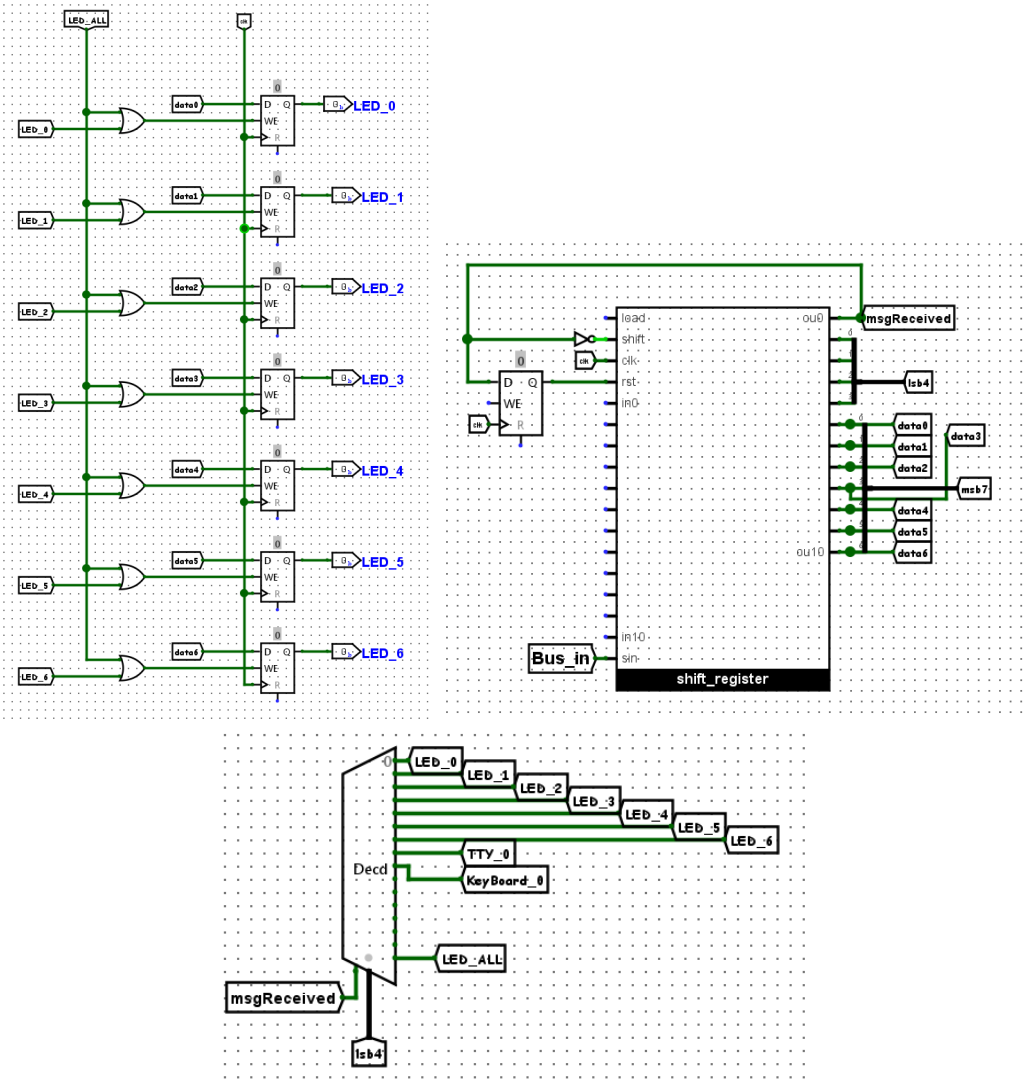
در هنگام استفاده از IOIsDone برای reset کردن هم کلاک اضافی قرار می‌دهیم تا در شیفت رجیستر زیر با خودش race ایجاد نشود.

۲-۳-۳ گرفتن و پردازش داده در سمت I/O

در مرحله‌ی آخر داده‌ی انتقال یافته را پردازش می‌کنیم تا عملیات‌های مورد نظر در سمت I/O انجام شود. به این صورت که داده‌ها در سمت ماژول گیرنده دریافت می‌شوند و در نتیجه با آدرس مقصد



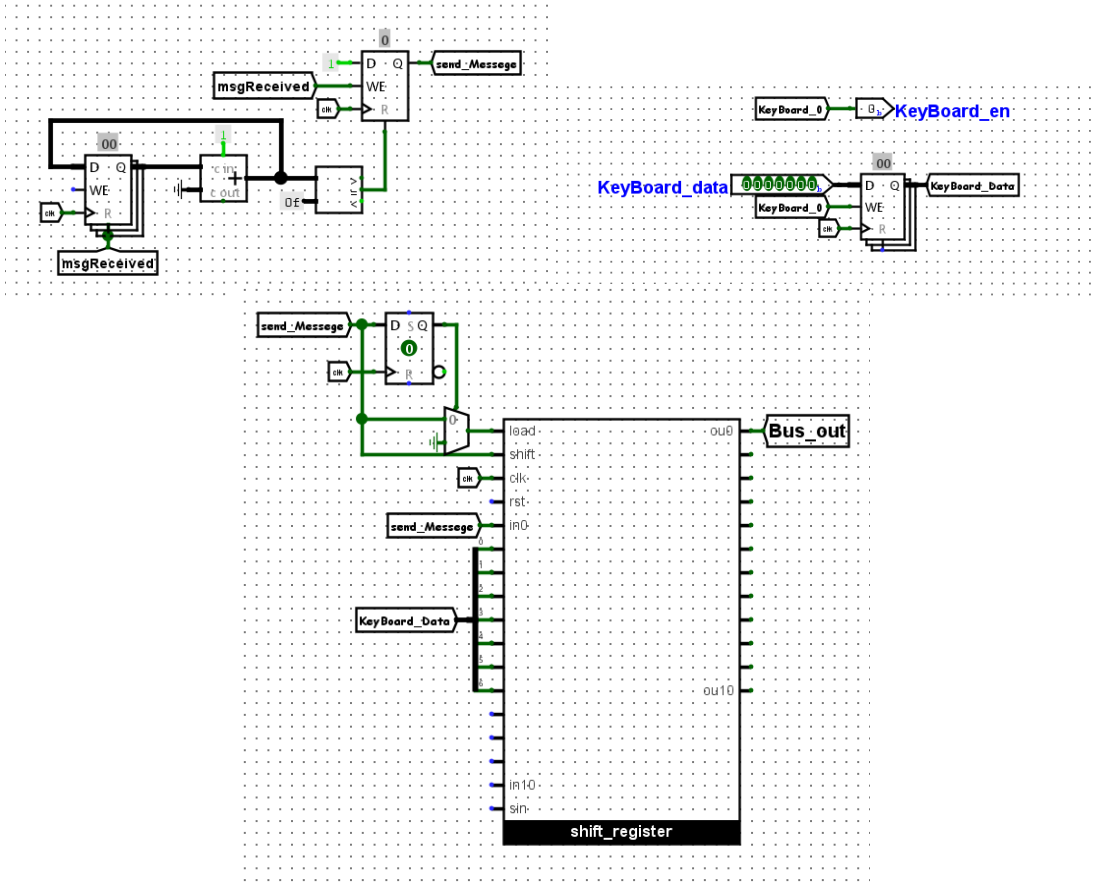
و یک دیکودر I/O مقصد را فعال می‌کنیم و داده‌ی لازم را که در ۷msb قرار دارد به مقصد می‌دهیم. حواسمان هست که با دریافت اطلاعات و رسیدن بیت نشانگر پایان شیفت را متوقف می‌کنیم و در کلاک بعدی هم برای اجرای دستور بعدی کلا شیفت رجیستر را reset می‌کنیم.



با استفاده از بیت i ام داده‌ی دریافت شده و بر حسب یک (روشن) و صفر (خاموش) بودن آن

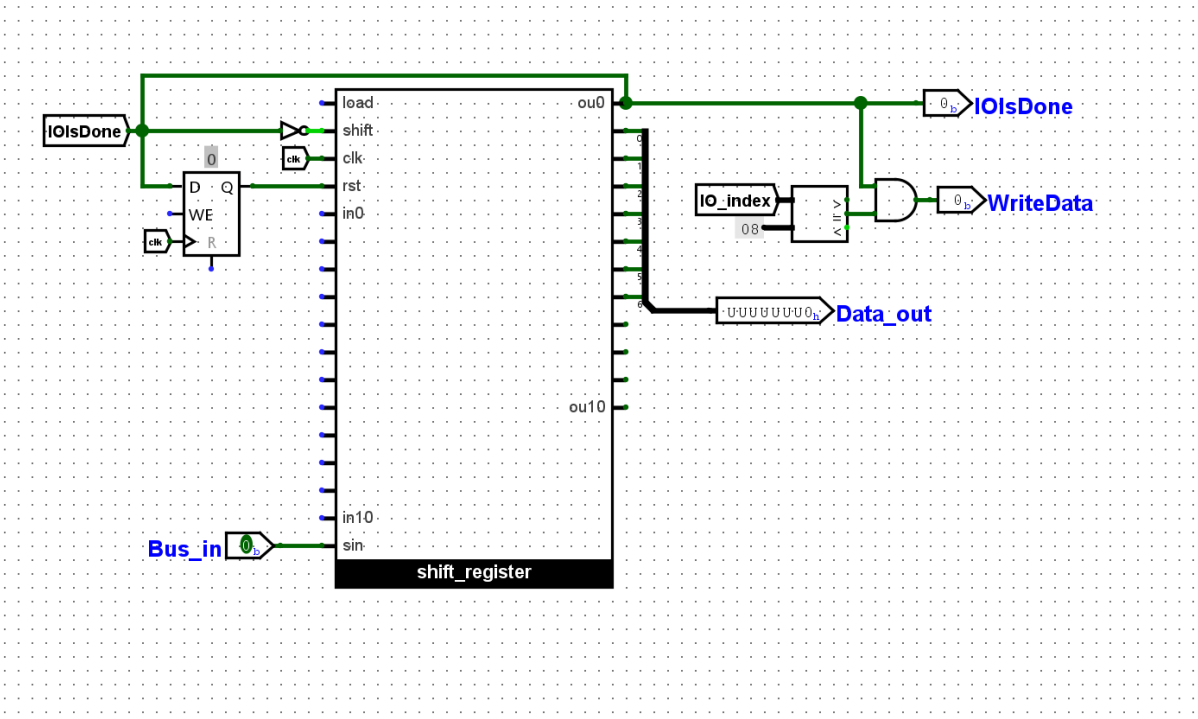
وضعیت LED i ام مشخص می‌شود. دقت کنید که برای پایدار بودن وضعیت لامپ تا زمانی که تغییر بعدی روی آن رخ دهد یک رجیستر در نظر می‌گیریم که تنها در زمانی که آدرس I/O مقصد لامپ موردنظر بود WE آن را فعال کند و تغییرش دهد. در غیر این حالت وضعیت قبلی لامپ را نگه می‌دارد. با استفاده از آدرس ۱۱۱۱ می‌توان همه‌ی لامپ‌ها را با توجه به مقدار باینری موجود در رجیستر مدنظر روشن کرد. برای TTY هم هفت بیت کاراکتر ورودی داده می‌شود و روی صفحه چاپ می‌شود.

نحوه ارسال داده KeyBoard و پردازش و گرفتن آن از سمت پردازنده و اما می‌رسیم به بحث کیبورد. در کیبورد علاوه بر دریافت اطلاعات از پردازنده مبنی بر اینکه می‌خواهیم با کیبورد کار کنیم و همچنین اطلاعات آن در چه رجیستری دخیره شود، باید کارکتر موجود در کیبورد را دریافت کرده و در رجیستر مقصد درون پردازنده ذخیره کنیم. این یعنی یک برگشت دیگر از سمت I/O به پردازنده. برای این منظور مشابه قسمت قبل عمل می‌کنیم و دو تا شیفت رجیستر دیگر یعنی به عبارت دیگر دو ماژول فرستنده و گیرنده‌ی دیگر قرار می‌دهیم، با این تفاوت که این دفعه فرستنده در سمت I/O قرار دارد و گیرنده در سمت پردازنده.

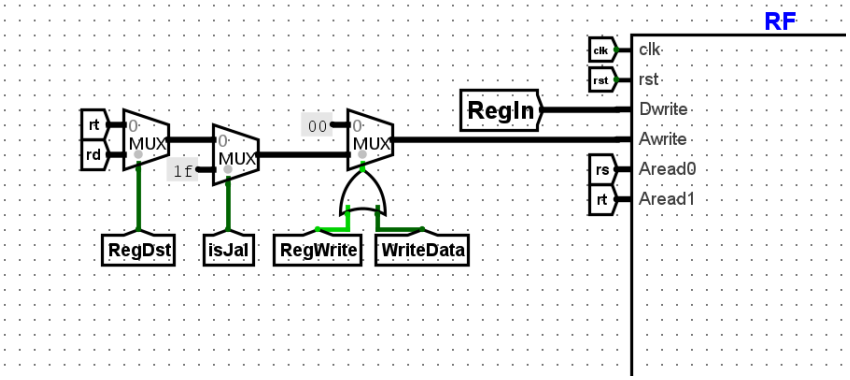


عملیات کیبورد بدین صورت عمل می‌کند که با فعال کردن کیبورد و دریافت داده‌ی آن، به صورت موازی آن را به شیفت رجیستر می‌دهیم تا آن به سمت پردازنده ارسال کند. برای این منظور

یک سیگنال آغاز ارسال پیام به پردازنده هم مطابق شکل مشابه isIO برای این ماژول می‌گذاریم و با شیفت داده را به سمت پردازنده ارسال می‌کنیم. این سیگنال sendMessage که علاوه بر برطرف کردن نیاز کیبورد، می‌توان از آن برای نشان دادن پایان کار پردازنده در دستور I/O استفاده کرد و با توجه به سیگنال IOIsDone که معرفی کردیم PC را افزایش داد و به دستور بعدی رفت.



برای گیرنده در قسمت پردازنده هم درست شبیه I/O عمل می‌کنیم با این تفاوت که اگر دستور کیبورد باشد یا به عبارتی آدرس مقصد برابر با ۸ باشد، داده‌ی خروجی باید در رجیستر مقصد نوشته شود که مطابق دستورات I-type در rt قرار دارد. فقط باید حواسمان باشد که سیگنال WriteData را فعال کنیم تا با استفاده از آن مطابق مدار زیر نوشتن در رجیستر فایل مجاز شود و بتوانیم $Data_{out}$.



۳ آزمون و تست کردن طراحی

برای بررسی عملکرد مدار و دستگاه طراحی شده از چهار برنامه‌ی زیر استفاده می‌کنیم.// این دستورات در ROM برنامه ذخیره میشوند.

۳-۰-۱ آزمون اول: روشن و خاموش کردن LED

Instruction	Machine Code
addi $r2, $r2, 4	00100000010000100000000000000100
led $r2	11111100010000100000000000000000
addi $r2, $r0, 0	00100000000000100000000000000000
led $r2	11111100010000100000000000000000

۳-۰-۲ آزمون دوم: نمایش کاراکتر موجود در رجیستر مورد نظر روی TTY

کاراکتر a با کد اسکی ۹۷ روی صفحه نمایش داده می‌شود

Instruction	Machine Code
addi $r2, $r2, 97	00100000010000100000000001100001
tty $r2	11111100111000100000000000000000

۳-۰-۳ آزمون سوم: نمایش کل کلمه‌ی نوشته شده روی کیبورد روی صفحه‌ی TTY

Instruction	Machine Code
kb $r2	11111101000000100000000000000000
tty $r2	11111100111000100000000000000000
bne $r2, $r0, kb	00010100010000001111111111111101

حلقه تا وقتی که به کاراکتر null با کد اسکی صفر نرسیده‌ایم، تکرار میشود.

۳-۰-۴ آزمون چهارم: تولید توالی فیبوناچی روی LED ها

این سری را تا جایی که ۷ بیت لامپ می‌تواند نمایش دهد (یعنی عدد ۵۵) تولید و با نور آن‌ها به شکل عدد باینری درآورده و به نمایش می‌گذاریم.

Instruction	Machine Code
xor $r1, $r1, $r1	00000000010000100001000000100110
xor $r2, $r2, $r2	00000000010000100001000000100110
addi $r2, $r2, 1	00100000010000100000000000000001
xor $r4, $r4, $r4	00000000100001000010000000100110
addi $r4, $r4, 89	00100000100001000000000001011001
xor $r5, $r5, $r5	00000000101001010010100000100110
led $r2	11111111111000100000000000000000
add $r3, $r2, $r1	00000000010001000011000000100000
add $r1, $r2, $r5	00000000101000100000100000100000
add $r2, $r3, $r5	00000000101000110001000000100000
bne $r2, $r4, fibo	00010100010001001111111111111011

۳-۰-۵ آزمون پنجم: بررسی عملکرد Shell

در این آزمون، یک شل ساده طراحی شده که دستورات ورودی کاربر از طریق صفحه‌کلید را تحلیل کرده و بسته به نوع دستور، عملیات خاصی را اجرا می‌کند. عملکرد این شل به‌صورت زیر است:

- اگر کاربر کلمه fib را وارد کند، برنامه توالی فیبوناچی را تا عدد ۵۵ محاسبه کرده و هر عدد را به‌صورت دودویی روی هاLED نمایش می‌دهد.
- اگر کاربر دستور LED on x وارد کند (که x عددی بین ۰ تا ۶ است)، LED متناظر با عدد روشن می‌شود.
- اگر کاربر دستور LED off x را وارد کند، LED مشخص‌شده خاموش می‌شود.
- اگر کاربر دستور type <something> را وارد کند، متن واردشده بعد از type در TTY نمایش داده می‌شود.
- در صورت وارد کردن دستور اشتباه یا ناقص، شل هیچ کاری انجام نمی‌دهد یا وارد حالت نامتعارف می‌شود.

این شل از ساختارهای state machine بهره می‌برد و از نقطه‌ی شروع تمام ورودی‌ها را بررسی

کرده و به هر استیت (مانند FIB یا ON یا OFF یا TXT) در صورت انطباق پرش می‌کند. در انتهای اجرای هر استیت، برنامه به نقطه‌ی شروع بازمی‌گردد و منتظر دستور بعدی می‌ماند.

در کد فوق که در قسمت shell and test در گیتهاب قرار داده شده است:

- شناسایی دستورات با مقایسه‌ی پی‌درپی کاراکترهای ورودی با مقادیر ASCII آن‌ها انجام می‌شود.
- اگر ورودی با رشته‌ای خاص (مثلاً fib) مطابقت داشته باشد، به برچسبی مانند FIB پرش می‌شود.
- دستوراتی که با LED در ارتباط هستند، مستقیماً به روشن یا خاموش کردن LED مشخص‌شده می‌پردازند.
- در حالت TXT، تمام کاراکترهای پس از type در TTY نمایش داده می‌شوند تا زمانی که کاراکتر خط جدید (0x0A) وارد شود.

این شبیه‌سازی ساده از یک شل، پایه‌گذار درک بهتری از نحوه عملکرد واسط‌های متنی و تحلیل دستورات در سیستم‌عامل‌های سطح پایین است.