درآمدی بر پردازنده ها

محمدرضا حقیری

۲۷ فروردین ۱۳۹۵

فهرست مطالب

۱ مقدمه

در روزگار امروز، در هر خانه ای دست کم یک دستگاه رایانه می‌توان یافت. دستگاهی که گاه با ارزان ترین و گاه با گرانقیمت ترین تکنولوژی ها ساخته شده است.این دستگاه، پس از پیشرفت های علم الکترونیک، به شکل مدرن و امروزی در آمد و به قدری ارزان شد که بتوانیم همه ما، بسته به نیاز و بودجه‌مان، دست کم یک عدد در خانه داشته باشیم.

اما آیا تا به حال به این اندیشیده اید که رایانه ها چطور کار میکنند؟ و چه چیزی است که رایانه ها را از سایر دستگاه‌های الکتریکی یا الکترونیکی متمایز نموده است؟ برای مثال رایانه از کجا می‌داند که با یک داده باید چه کند؟ چیزی که در این کتابچه بررسی خواهیم کرد، «از کجا دانستن» و «از کجا آمدن» داده ها و کامپیوتر است.

۱.۱ این کتابچه برای چه کسانی است؟

سعی بر آن بوده تا همه کاربران رایانه بتوانند این کتابچه را درک کنند. اما در حقیقت این کتابچه، برای کسانی نوشته شده است که قصد دارند با تیری چند نشان زده، و اطلاعات خود در باب مدارات منطقی، برنامه نویسی سیستمی [۱] و معماری کامپیوتر را بالاتر ببرند. همچنین، دانشجویان رشته های کامپیوتر و آی‌تی نیز میتوانند برای انجام تمرینات و پروژه هایی که در این دروس به آن ها داده شده است بهره گرفته و استناد کنند.

محمدرضا حقیری - بهار ۹۵

[۱] Assembly

۲ همه چیز از اینجا شروع می شود

پایه علوم کامپیوتر، منطق است. در واقع منطقی که در کامپیوتر استفاده می‌شود، نوع بسیار کهن منطق است. ما نیازی نداریم تا خیلی هم پیشرفته و پیچیده فکر کنیم تا بتوانیم عملکرد کامپیوتر را درک کنیم. منطق نخستین بار توسط فیلسوف یونانی، ارسطو، مطرح شد. ساده ترین مثال منطق این است :

یک رابطه دوستی ساده را در نظر بگیرید. علی با محمد دوست است. محمد با حسن دوست است. پس علی نیز با حسن دوست است.

گرچه به هیچ عنوان نمی‌توان به مثال «رابطه دوستی» استناد محکمی کرد. پس نیاز داریم از چیزی نتیجه گیری مطلق کنیم که قابل نقض نباشد. چرا که ممکن است علی، حسن را هرگز ندیده باشد و نشناخته باشد، چیزی که هر روز در زندگی ما ممکن است اتفاق بیفتد. اکنون این مثال را در نظر بگیرید :

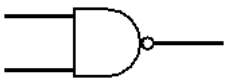
علی از محمد قد بلند تر است، محمد از حسن قد بلند تر است. پس علی از حسن قد بلند تر است.

اکنون، مثال ما نه تنها به واقعیت نزدیک تر شد، بلکه کاملا درست و غیرقابل نقض کردن است. اتفاقاتی که در کامپیوتر میفتند و باعث می‌شوند که درصد خطای کامپیوتر در بسیاری از کارها، پایین تر از چیزی باشد که ما فکر میکنیم، تبعیت از منطق است. اکنون دیدی از منطق داریم. پس برویم سراغ یک مثال، و بعد ببینیم که در کامپیوتر چه رخ میدهد.

۱.۲ روز آفتابی و لامپ روشن

اکنون فرض کنیم وارد اتاقی شده ایم که نورگیر خوبی دارد. یک لامپ هم در اتاق روشن است. چه اتفاقی میفتد؟ قطعا به سرمان میزند که لامپ را خاموش کنیم. پس بدنبال کلید میگردیم، می بینیم که دو کلید وجود دارد و هردو در موقعیت خاموش قرار دارند [۲] نخستین فرضیه ما، این خواهد بود که شخصی که کلید را نصب کرده، سیم کشی را برعکس انجام داده است. فعلا کاری با وضعیت خاموش نداریم، کلیدی که به در اتاق نزدیک تر است را فشار میدهیم و می بینیم که تغییری در وضعیت لامپ حاصل نشد. حالا کلید را به وضعیت اولیه برگردانده و بعد کلید بعدی را فشار میدهیم. باز هم تغییری در وضعیت حاصل نمی شود. ممکن است عجیب و غیرعقلانی به نظر برسد، ولی ناگهان با قرار دادن هر دو کلید در وضعیت روشن، می‌بینیم که چراغ خاموش شد.

بعد از باز کردن و شکافتن کلید، به این میرسیم که سیم کشی ها کاملا درست است. اما چه چیزی باعث شده که در صورت خاموش بودن دو ورودی، یک خروجی خاموش شود؟ جواب این است که یک «دروازه» [۳] در راه جریان الکتریکی قرار داده ایم، که عملکردش به این شکل است. این، ساده ترین دروازه و پایه ساخت همه کامپیوترهای جهان، اعم از ساعت های دیجیتال، تلویزیون های هوشمند، گوشی موبایل و ... است. نام این دروازه را NAND گذاشته اند. و برای نمایش آن از نماد زیر استفاده می‌شود.



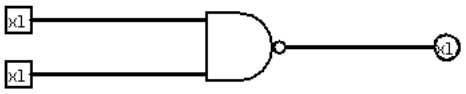
شکل ۱: NAND

[۲] فرض را بر این بگیرید که با استفاده از یک نوشته، یا برچسب یا نماد خاصی میتوانیم وضعیت روشن و خاموش بودن کلیدها را تشخیص دهیم

[۳] Gate

۲.۲ مدیریت خروجی ها

اکنون نیاز داریم تا خروجی های NAND را بررسی کنیم. در کامپیوتر، برای راحتی کار، همیشه جریان «وصل» را «یک» و جریان «قطع» را «صفر» در نظر میگیریم. از این جهت، نمایش خروجی ها برای ما بسیار بسیار ساده می‌شود.برای نمایش خروجی ها، از جدولی موسوم به «جدول درستی» [۴] بهره می‌گیریم. کافیست اکنون روی ورودی ها و خروجی هایمان نام بگذاریم. برای مثال، یک ورودی را A و دیگری را B و خروجی را C مینمامیم. و مداری به این شکل میسازیم : حالا کافیست جدول درستی را به این شکل رسم کنیم :



شکل ۲: منطق NAND

A	B	C
۰	۰	۱
۰	۱	۱
۱	۰	۱
۱	۱	۰

اگر بر فرض، A و B کلید باشند و C لامپ، اکنون میتوان گفت که در سه حالت اول، یعنی خاموش بودن هر دو کلید، خاموش بودن A و روشن بودن B ، روشن بودن A و خاموش بودن ،B لامپ روشن می‌شود. چنانچه هر دو کلید روشن باشند نیز لامپ خاموش می‌شود.

[۴] Truth table

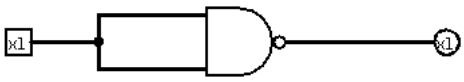
۳ منطق های بیشتر

بیاید منطق های بیشتری را با استفاده از NAND بسازیم. همانگونه که گفته شد، همه چیز در کامپیوتر با استفاده از NAND ساخته می‌شود. پس طبیعتا هر منطق یا دستور و عملگر منطقی ای، قرار است که با استفاده از NAND ساخته شود. در این بخش، دو منطق پرکاربرد ولی ساده را با استفاده از NAND می‌سازیم.

۱.۳ کامپیوتر ها «نه» می‌گویند

یکی از توانایی هایی که هر انسانی برای آن تربیت می‌شود، «نه گفتن» است. در واقع گاهی اوقات شرایط طوری است که نمیتوان چیزی را قبول کرد. کامپیوتر ها هم ساخته می‌شوند که تا حدودی مانند انسان رفتار کنند. در این قسمت، میخواهیم قسمت «نه گو»ی کامپیوتر را با هم بسازیم.

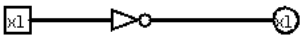
همان دروازه NAND را فرض کنید، اکنون به جای A و B ، فقط و فقط A را با دو سیم به دو ورودی NAND متصل میکنیم : اکنون بیاید جدول درستی را رسم کنیم :



شکل ۳:

A	A	C
۰	۰	۱
۱	۱	۰

جدول درستیمان ساده تر شد، چرا که به جای دو ورودی، تنها یک ورودی داشتیم. این گیت و این منطق را به صورت ساده تر نیز میتوان نشان داد :



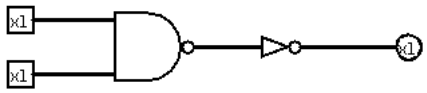
شکل ۴:

اکنون، ما دروازه ای جدید ساخته ایم، که نامش NOT یا «نقض کننده» است. خروجی مدار NOT را «نقیض» میگوییم. در واقع، با استفاده از NOT میتوانیم یک ها را به صفر، و صفر ها را به یک تبدیل کنیم. در واقع به ورودی ها، «نه» میگوییم و نقیضشان را در خروجی قرار می‌دهیم. جدول درستی دروازه NOT هم به شکل زیر است :

A	C
۱	۰
۰	۱

۲.۳ مدار همیشه راستگو

اکنون که با منطق NOT و NAND آشناییم، بیاییم مداری بسازیم که همیشه راست می گوید! در واقع ما اکنون به مداری نیاز داریم تا «زمانی خروجی درست دهد، که هر دو ورودی درست باشند». ساده ترین کاری که میتوانیم انجام دهیم این است که یک بار خروجی NAND را NOT کنیم. به این شکل :



شکل ۵:

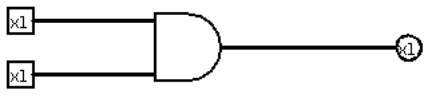
اکنون سیمی که از NAND خارج شده را C در نظر بگیرید، و خروجی را D . جدول درستی برای C به این شکل است :

A	B	C
۰	۰	۱
۰	۱	۱
۱	۰	۱
۱	۱	۰

حالا بیاید برای D جدول درستی رسم کنیم :

C	D
۱	۰
۱	۰
۱	۰
۰	۱

همانطور که دیدید، سه بار مقدار ۱ توسط NAND تولید شده، که هر بار توسط NOT به صفر تغییر کرده. و سپس، یک بار خروجی صفر توسط NAND تولید شده که با استفاده از NOT به مقدار «یک» رسیده ایم. این منطق، AND نام داشته و مدارش به این شکل نشان داده می‌شود :



شکل ۶: مدار AND

اکنون، اگر ورودی ها را A و B و خروجی را C در نظر بگیریم، جدول درستیمان به این شکل است :

A	B	C
۰	۰	۰
۰	۱	۰
۱	۰	۰
۱	۱	۱

تا اینجا، سه دروازه ،NOT NAND و AND را شناخته ایم. در بخش های بعدی، با استفاده از این دروازه ها، قطعاتی را میسازیم که میتوانند در ساخت یک کامپیوتر ساده، با ما کمک کنند.

۴ رمزگشایی در کامپیوتر

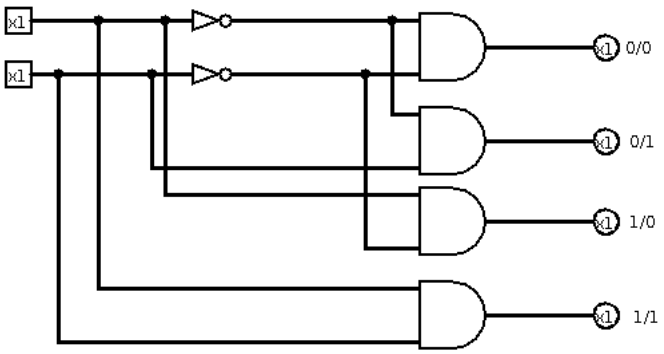
در زمان‌های حساس، مانند جنگ، یا یک عملیات محرمانه و اوضاع مشابه، هرچه لازم است به زبان «رمز» نوشته می‌شود. در واقع، زبان رمز یعنی چیزی که بتواند در عین انتقال پیام، آن را از دید نامحرم نیز حفظ کند. ساده ترین نوع رمزنگاری، این است که گفته ها یا نوشته های خود را به غریبه ترین زبان ممکن که طرف مقابل از آن آگاهی دارد ترجمه کنیم. یا از نماد ها و شکل های دیگری به جای حروف الفبا استفاده کنیم.اما، مهم این است که اتفاقی که در کامپیوتر میفتد این است که هیچ رمزی گشوده نمی شود! بلکه فقط توسعه داده می‌شود. برای درک بهتر بیاید این مساله را در نظر بگیریم :

ما دو ورودی داریم، و چهار لامپ. چگونه میتوانیم با دو کلید چهار لامپ را روشن و خاموش کنیم؟

مساله ای که حل کردن آن واقعا کاری ندارد. تنها کافیست از ترکیب چند منطق استفاده هوشمندانه ببریم. نخستین چیزی که باید به آن توجه کنیم، این است که «چهار» و «دو» با یکدیگر رابطه دارند.در واقع میتوانیم «چهار» را به صورت «دو به توان دو» بنویسیم. پس با این حساب، کارمان بسیار بسیار ساده تر می‌شود. فقط کافیست از چهار دروازه AND استفاده کنیم و با دو ورودی، چهار خروجی به دست آوریم. البته باید در نظر بگیریم که چهار حالت باید داشته باشیم که به این شکل هستند :

A	B
۰	۰
۱	۰
۰	۱
۱	۱

نظر به این که AND تنها زمانی به ما خروجی درست میدهد که هر دو ورودی درست باشند، نیاز است که صفر ها قبل از ورود به دروازه، به یک تبدیل شوند. پس از دروازه NOT هم کمک میگیریم.
مدار طراحی شده نهایی ما به شکل زیر است:



شکل ۷: مدار رمزگشا

با توجه به این مدار، ترتیب روشن شدن خروجی ها به شرح زیر است :

A = 0, B = 0 -> 0/0
A = 0, B = 1 -> 0/1
A = 1, B = 0 -> 1/0
A = 1, B = 1 -> 1/1

در واقع، ما برای ساخت دیکدر چنین فرمولی داریم :

$$nx2^n$$

یعنی برای دیکدری با تعداد ۲ ورودی (چیزی که بالاتر ساختیم)، داریم :

2x4

و به همین ترتیب با ۳ ورودی، میتوانیم ۸ خروجی، با ۴ ورودی میتوانیم ۱۶ خروجی بسازیم و همینطور بالاتر برویم.

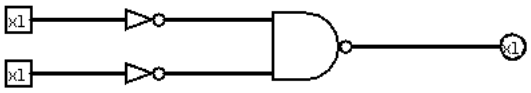
در این بخش هم با ساخت یک رمزگشا آشنا شدیم. در بخش های بعدی، منطق های جدیدتر را فرا خواهیم گرفت و سپس، با استفاده از آنها، به ساخت قطعات جدیدی میپردازیم.

۵ دو منطق جدید

تا اینجا، با NAND آشنا شدیم، با استفاده از NAND ، منطق های NOT و AND را ساختیم، با استفاده از NOT و AND نیز دیکدر طراحی کردیم و ساختیم. اما اکنون به منطق های جدیدتر نیاز داریم.

۱.۵ یا این، یا آن، یا هر دو!

ما مدارها و دروازه هایی را طراحی کرده ایم که حالات مختلفی را برای ما میتوانند بسازند. اما اینگونه نیست که «در صورتی که حداقل یکی از ورودی ها درست باشند» به ما نتیجه درست بدهد. اکنون میخواهیم این مشکل را برطرف نماییم. برای طراحی چنین مداری، نقیض ورودی ها را به NAND میدهیم. در واقع چنین مداری طراحی میکنیم :



شکل ۸:

بیایید ابتدا دو جدول درستی برای سیم هایی که از NOT ها خارج می شوند را رسم کنیم (چون مدار NOT است، فقط جدول درستی یکی از آنها اینجا آورده شده) :

A	C
۱	۰
۰	۱

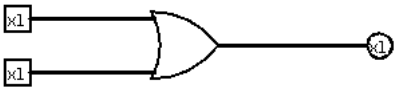
حالا بیایید جدول درستی NAND را بار دیگر بررسی کنیم :

A	B	C
۰	۰	۱
۰	۱	۱
۱	۰	۱
۱	۱	۰

بسیار خوب، اکنون جدول درستی کل مدار فوق را بررسی میکنیم :

A	B	C
۰	۰	۰
۰	۱	۱
۱	۰	۱
۱	۱	۱

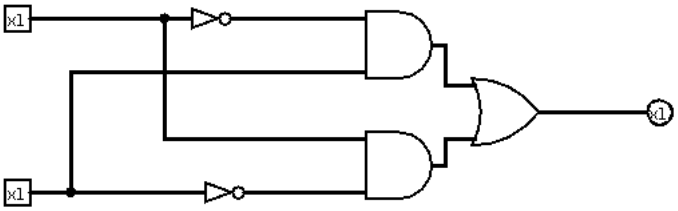
منطق جدید ما «یا» یا OR نام دارد. در واقع با استفاده از OR مطمئن هستیم که تنها زمانی نتیجه درست می‌گیریم که حداقل یک ورودی درست داشته باشیم. نمایش مدار OR به شکل زیر است :



شکل ۹: مدار OR

۲.۵ یاء انحصاری

اکنون منطق «یا» را فرا گرفتیم. نیاز به منطق دیگری نیز هست؟ بله نیاز به یک منطق دیگر داریم تا تمامی منطق های موجود پوشش داده شود، تا بتوانیم مدارات عملیاتی طراحی کرده و بسازیم. در واقع میخواهیم نتایج OR را محدود تر کنیم. یعنی بگوییم «خروجی درست است اگر و فقط اگر یک ورودی درست باشد». اینجاست که برای اعمال محدودیت های لازم، از AND نیز استفاده می کنیم :



شکل ۱۰:

در مدار فوق، هر گاه یکی از ورودی ها روشن باشد، خروجی روشن است. این در صورتیست که اگر هر دو ورودی را همزمان روشن کنیم، یا همزمان خاموش، خروجی خاموش خواهد شد. پس یک جدول درستی کلی به این شکل میتوانیم برای این مدار ارائه کنیم :

A	B	C
۰	۰	۰
۰	۱	۱
۱	۰	۱
۱	۱	۰

این منطق را XOR یا «یاء انحصاری» [۵] نیز می‌گویند. همچنین دروازه ای به این شکل برای این منظور طراحی شده است :

شکل ۱۱:

[۵] Exclusive OR

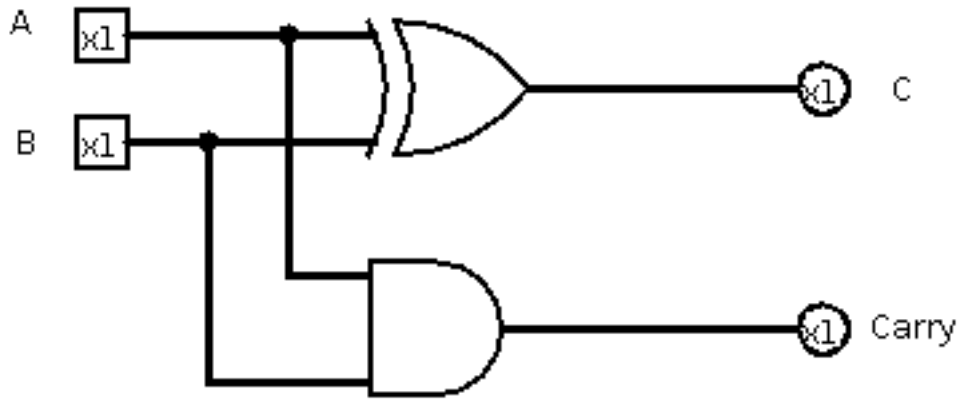
با استفاده از این دروازه، میتوان بسیاری از عملیات های محاسباتی و منطقی را تسریع کرد و انجام داد، که در بخش های بعدی با آنها آشنا خواهیم شد.

۶ محاسبه و مقایسه

اکنون نوبت آن رسیده که منطق ها را با یکدیگر جمع کنیم، و به یک سری مدار برسیم که به ما در حل مسائل محاسباتی و منطقی کمک کنند. در واقع یکی از اهداف اختراع کامپیوتر، تسریع محاسبات بوده است. پس طبیعی است که مدارهای محاسباتی قبل از هرچیزی اختراع شده باشند. ما هم قرار است اینجا، یک بار دیگر چرخ را اختراع کنیم، پس بیاید مدارات محاسباتی را بررسی کنیم.

۱.۶ عمل جمع

ساده ترین و نخستین عملی که انسان می‌آموزد، عمل جمع است. پس با تفکر این که تکامل کامپیوتر هم مانند تکامل انسان است، باید این عمل را به کامپیوتر نیز یاد بدهیم. پس باید به شکلی مداری را بسازیم. ساده ترین مداری که میتوان ساخت، مداریست که «نیم جمع کننده» [۶] نام دارد. این مدار به این شکل ساخته می‌شود :



شکل ۱۲: نیم جمع کننده

در این مدار، با استفاده از XOR دو مقدار ورودی را با یکدیگر جمع کرده، و «رقم نقلی» آن را توسط AND به دست می آوریم. بیاید جدولی برای محاسبه نتیجه محاسبات نیم جمع کننده رسم کنیم :

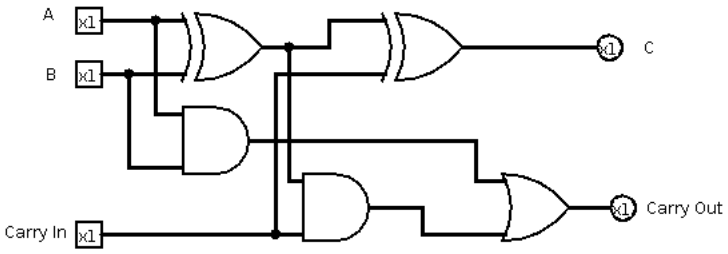
A	B	C	carry
۰	۰	۰	۰
۰	۱	۱	۰
۱	۰	۱	۰
۱	۱	۰	۱

اکنون حاصل جمع ها را به چشم دیدیم. اما وقتی دو ورودی داریم، باید چهار خروجی داشته باشیم. برای حل این مشکل، باید مدار دیگری طراحی شود. مدار جدید ما «تمام جمع کننده» [۷] نام دارد که در واقع ترکیب دو نیم جمع کننده است.

[۶] Half Adder
[۷] Full Adder

۱.۱.۶ طراحی تمام جمع کننده

برای طراحی این مدار، باید دو «نیم جمع کننده» را با هم ترکیب کنیم، و مداری به این شکل بدست آوریم :



شکل ۱۳: تمام جمع کننده

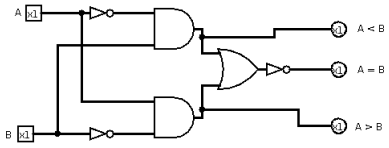
در این مدار، یک «رقم نقلی ورودی» نیز داریم. برای راحت تر شدن کار، «رقم نقلی ورودی» را cin و رقم نقلی خروجی را cout نام گذاری میکنیم. بیاید جدول نتایج این مدار را به‌دست آوریم :

A	B	cin	C	cout
۰	۰	۰	۰	۰
۰	۱	۰	۱	۰
۱	۰	۰	۱	۰
۱	۱	۰	۰	۱
۱	۱	۱	۱	۱

اکنون، از این بابت که مدارمان کامل است، و میتوانیم مقادیر ۰۰، ۰۱، ۱۰ و ۱۱ یعنی از صفر تا سه را با استفاده از تمام جمع کننده به دست آوریم، خیالمان راحت است. اکنون نوبت آن است که مقایسه مقادیر را بررسی کنیم.

۲.۶ مقایسه مقادیر

برای مقایسه مقادیر نیز، ما باید مداری طراحی کنیم. ابتدا باید حالات مختلف مقایسه را بررسی نماییم. در واقع، ساده ترین نوع مقایسه، بررسی تساوی دو مقدار است. نوع دیگر، بزرگتر بودن و کوچکتر بودن مقادیر نسبت به یک دیگر است. گرچه با ترکیب XOR و NOT میتوان تساوی را بررسی کرد، اما برای بررسی کوچکتر یا بزرگتر بودن مجبوریم مدار بزرگتری طراحی کنیم. مدار مقایسه گر به طور کلی چنین است :



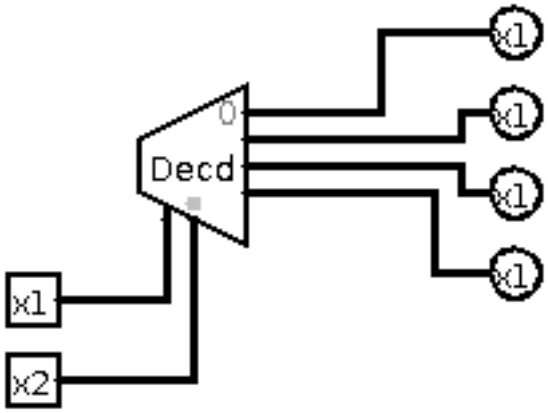
شکل ۱۴: مدار مقایسه گر

این مدار وظیفه مقایسه دو مقدار A و B را عهده دار است. در واقع وقتی هر دو وردی در وضعیت خاموش باشند، یا هر دو در وضعیت روشن، خروجی میانی روشن می شود. اگر یکی از آنها روشن باشد، خروجی بالایی (کوچکتر بودن (A یا خروجی پایینی (بزرگتر بودن A) روشن است.

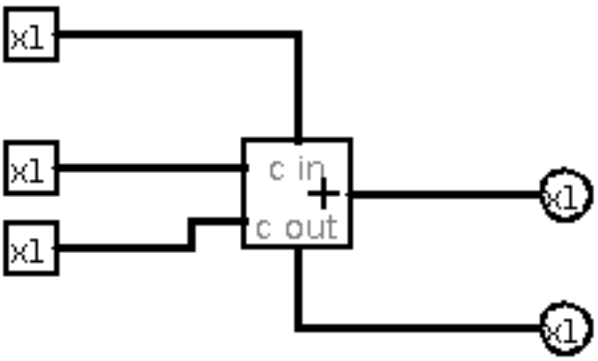
تا اینجا، با چند مدار آشنا شدیم. در بخش های بعدی، با مدارهای بیشتر و نحوه ترکیب آنها با یکدیگر آشنا خواهیم شد.

۷ نمایش مدارات به صورت خلاصه

نوشتن به صورت خلاصه، و استفاده از نمادها و ... ، همیشه برای تفهیم بهتر و مصرف فضای کمتر، مطرح بوده است. تمامی مدارهایی که تا الان طراحی کردیم را میتوانیم به صورت خلاصه نشان دهیم. به هر کدام از این شکل ها یک «دیاگرام» [۸] میگوییم. در اینجا مدارهایی که طراحی کرده ایم را با هم می‌بینیم :

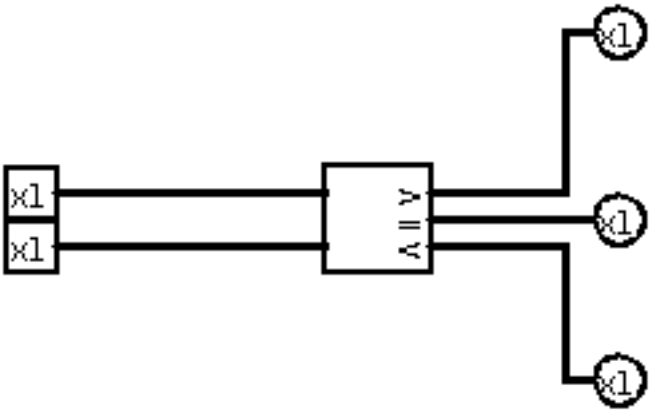


شکل ۱۵: دیاگرام مدار رمزگشا یا دیکدر ۲ در ۴



شکل ۱۶: دیاگرام تمام جمع کننده

[۸] Diagram



شکل ۱۷: دیاگرام مقایسه گر

با استفاده از این دیاگرام ها و چیدن آنها کنار هم، میتوان رسم نقشه مدار های پیچیده تر را ساده و ساده تر نمود، چرا که نقشه کلی ما ساده تر خواهد شد. گرچه دانستن ترکیب و نقشه مدارها الزامی است.

۸ واحد محاسبه و منطق

از آنجایی که قرار است در این کتابچه، پردازنده ها [۹] را بررسی نماییم، باید با بخش های مختلف آن آشنا شویم. مهم ترین بخش یک پردازنده، «واحد محاسبه و منطق» نام دارد.این واحد را در دیاگرام ها، ALU یا Arithmetic and Logical Unit می گوییم. تقریبا تمامی مداراتی که در بخش های گذشته طراحی کرده ایم را نیاز داریم کنار هم بگذاریم، تا به یک «واحد محاسبه و منطق» برسیم. اما قبل از آن، بیایید چندین قدم عقب تر برویم.

۱.۸ چالش های پیش رو برای طراحی یک ریزپردازنده

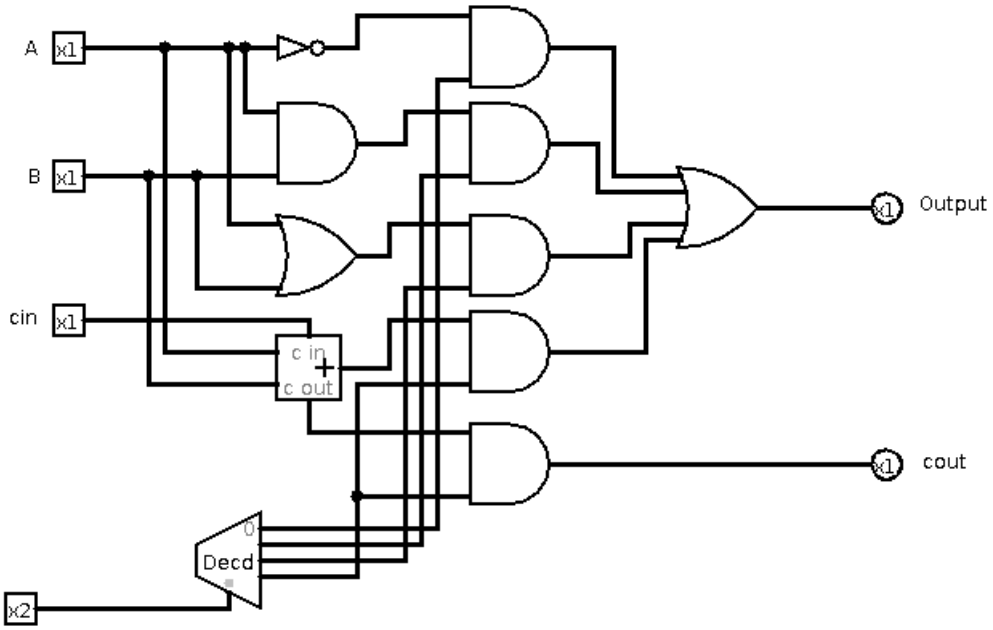
نخستین چالش، این است که هدف ما از طراحی ریزپردازنده چیست. برای مثال، قصد و هدف ما از طراحی در این کتابچه، این است که با ترکیب کردن و چیدن قطعات کنار یک دیگر آشنا شویم. همچنین، قصدمان این است که زبان اسمبلی را بسیار بهتر و از نزدیک درک کنیم. پس بهتر است که تا حد امکان، پردازنده خود را ساده نگه داریم. در اینجا، سایز ورودی هایمان را روی یک بیت قرار میدهیم تا صرفا چند عملیات ساده را انجام دهیم. گرچه میتوان بعد ها این عملیات را پیچیده تر کرد و حتی سایز ورودی ها را نیز بالاتر برد. بسیار خوب، مشخصات لازم برای پردازنده کوچک خود را داریم. بیاید ببینیم چه دستوراتی را میتوانیم کنار هم قرار دهیم؟ برای مثال میخواهیم ریزپردازنده ما قابلیت NOT و AND و OR و همچنین جمع کردن ورودی ها را داشته باشد. همانطور که می بینید، چهار ورودی داریم. پس نیاز داریم در صورت لزوم هم از دستوری به دستور دیگر، سوییچ کنیم.

۱.۱.۸ فعال کردن دستورات

ما نیاز داریم که خروجی ها را بسته به نیازمان، تغییر دهیم. مثلا وقتی قرار است NOT صورت گیرد چه اتفاقی بیفتد یا همینطور برای AND و OR . بنابراین نیاز به قطعه ای داریم که برای ما، بتواند دستورات را رمزگزاری و رمزگشایی کند. قاعدتا اینجاست که به دیکدر نیاز پیدا میکنیم. اکنون کافیست خروجی دیکدر را با خروجی دستور مورد نظرمان، AND کنیم. ممکن است بپرسید چرا AND ؟ و چرا از دستور دیگری استفاده نکنیم؟ پاسخ شما این است که «ما همیشه نیاز داریم خروجی کنترل شده و صحیحی داشته باشیم. قاعدتا زمانی که بخواهیم خروجی درست یک دستور را داشته باشیم، نیاز داریم که صرفا زمان «فعال» بودن کد و دستور، خروجی را بگیریم و باقی حالات را خارج کنیم.

[۹] Processor

۲.۸ دیاگرام واحد مقایسه و منطق



شکل ۱۸: دیاگرا واحد محاسبه و منطق

اگر به دیاگرام بالا دقت کنید، خواهید دید که از چهار حالت دیکدر، برای مقاصد خاصی استفاده کردیم. مثلا وقتی هر دو ورودی دیکدرمان صفر است، ورودی NOT می شود، وقتی روی وضعیت صفر-یک قرار دارد، عملیات AND رخ میدهد و الی آخر. میتوانیم خروجی های دیکدر، یا به عبارتی دیگر، مقدار دو رقمی ای که دیکدر به عنوان ورودی دریافت میکند را «رمز عملیات» یا Operation Code بنامیم. همچنین، اگر دقت کنید، خواهید دید که تمامی خروجی ها غیر از رقم نقلی، با یکدیگر، OR شده اند و این به این منظور انجام شده است که خروجی ما نیز «یک بیت» باشد.

اکنون، می‌توانیم ساختار زبان اسمبلی را نیز با استفاده از منطقی که در ALU خود داریم، بررسی کنیم.

۹ بررسی زبان ماشین

هر چه ما به عنوان ورودی، دستور و خروجی در یک کامپیوتر داریم، به صورت رشته ای از صفرها و یک هاست. در این مورد خاص، صرفا یا صفر است و یا یک (چرا که سایز ورودی ما یک بیت است). اما همچنان، نیاز داریم بررسی کنیم که این صفر ها و یک ها چگونه توسط کامپیوتر خوانده می‌شوند؟
پاسخ بسیار ساده است، شما کافیست واحد محاسبه و منطق را یک بار دیگر بررسی کنید. سپس ورودی های آن را در چهار حالت ممکن، با خروجی ای که میگیریم یا در واقع دستوری که فعال میکنیم، کنار هم قرار دهیم. اکنون، یک جدول ساده میکشیم که دستورات و ورودی های دیکدر را بررسی میکند :

A	B	دستور
۰	۰	NOT
۰	۱	AND
۱	۰	OR
۱	۱	ADD

همانگونه که گفتیم، این کد های دو رقمی را Operation Code می‌گوییم. هر دستوری که قرار باشد کاری انجام دهد، از دو بخش کد عملیات و عملوند یا Operand تشکیل شده است. در واقع، اینجا عملوند های یک بیتی (صفر و یک) داریم. اما مهم این است که ما، برای نوشتن برنامه برای ماشین، از این کدها استفاده نمی کنیم. بلکه از زبانی به نام «اسمبلی» استفاده می کنیم که بسیار به زبان انسان نزدیک تر است.

۱.۹ زبان اسمبلی برای پردازنده یک بیتی

ما برای پردازنده یک بیتی خود، چهار دستور ساختیم. دستورات ما در واقع این ها بودند :

```
NOT Operand1
AND Operand1, Operand2
OR  Operand1, Operand2
ADD Operand1, Operand2
```

حالا با استناد به جدول بالا، و این دستورات، یک قطعه برنامه کوچک مینویسیم :

```
AND 1, 1
OR  0, 1
ADD 0, 1
```

الان برنامه ای که نوشته ایم، در پردازنده مان به چه شکلی دیده می شود؟ دقت کنید که ویرگول ها و دستوراتی که نوشته ایم، صرفا برای نزدیک شدن به زبان انسان است. این درحالیست که ماشین ما به این شکل به این کد نگاه میکند :

```
01 1 1
10 0 1
11 0 1
```

و کل این کد های دو دویی [۱۰] را «زبان ماشین» یا Machine Code می گوییم. شما با هر پردازنده و سیستم عاملی که کار کنید، با اسمبلی متفاوتی رو به رو خواهید بود. اما این بخش و این کتابچه صرفا برای این است که شما عملکرد زبان اسمبلی را درک نمایید.

[۱۰] Binary

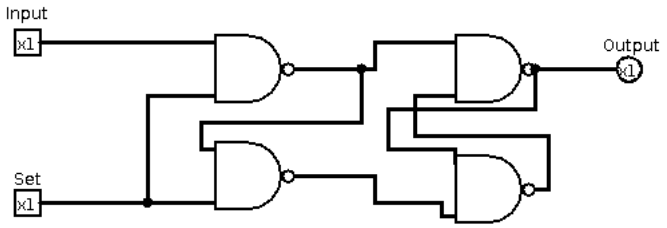
۱۰ حافظه

پس از انجام یک عملیات، نیاز داریم تا نتیجه را در جایی ذخیره کنیم. مثلا در زمانی که کامپیوتر در دسترسمان نیست، خروجی یک عمل جمع را روی کاغذ مینویسیم تا بعدا بتوانیم راحت تر به آن دسترسی پیدا کنیم. این نشان میدهد که ما نیاز به جایی برای ذخیره سازی داریم. در این بخش، قرار است در مورد حافظه صحبت کنیم. همانند طراحی ALU ، چالش هایی نیز برای طراحی حافظه داریم. مثلا، حافظه ما باید به قدری باشد که بتواند بزرگترین خروجی ممکن را در خود جای دهد. ما در واحد محاسبه و منطق، بزرگترین خروجیمان بدون احتساب cout یک بیت داریم. پس یک حافظه یک بیتی برایمان کافیست. گرچه این حافظه موقتی است، اما تا زمانی که عملیات جدیدی انجام نشود، میتوانیم خروجی را در آن نگه داری کنیم.

۱.۱۰ طراحی یک واحد حافظه

ما نیاز داریم تا «واحد حافظه» بسازیم. در واقع، در کامپیوتر کوچک ترین واحد حافظه ای که میتوان ساخت، «ثبات» یا Register نام دارد. در هر ثبات می توان حداکثر سایز ورودی را ذخیره نمود، مثلا اگر پردازنده ۸ بیتی بسازیم، می‌توانیم ۸ بیت داده را در آن ذخیره کنیم. حالا بیایید بررسی کنیم که یک ثبات چگونه ساخته می‌شود؟

کمی عقب تر برویم، نخستین منطقی که آموختیم، منطق NAND بود. این منطق،پایه ساخت یک کامپیوتر برشمرده شد، پس می توانیم با کنار هم گذاشتن تعدادی NAND یک حافظه کوچک بسازیم، در واقع به این شکل یک رجیستر یک بیتی طراحی کرده و می سازیم :

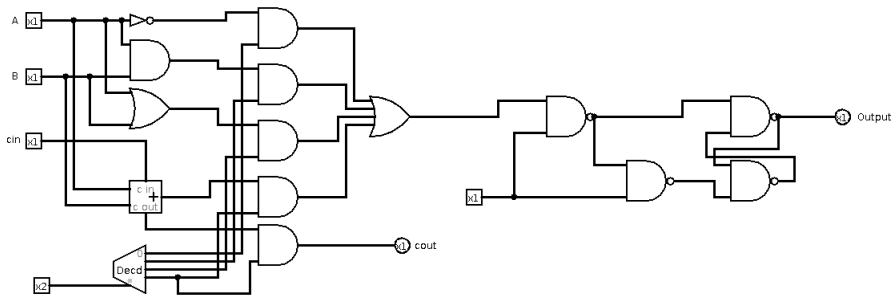


شکل ۱۹: دیاگرام رجیستر یک بیتی

تا موقعی که ورودی صفر باشد، Set هم صفر باشد، خروجی صفر است. وقتی ورودی یک باشد و Set صفر، باز هم خروجی صفر است. کافیست Set را فشار دهیم تا خروجی یک شود. با خاموش کردن Set در واقع هیچ تغییری در خروجی نمی توان ایجاد کرد. به همین دلیل است که در مدار یک رجیستر، میتوانیم به راحتی از ذخیره سازی بیت ورودی اطمینان حاصل کنیم.

حالا سوال دیگری پیش می آید این است که چگونه از یک رجیستر برای نگهداری خروجی خود استفاده کنیم؟ کافیست بزرگترین (و مهم ترین) خروجی ALU ای که طراحی کرده ایم را بعنوان ورودی به رجیستر یک بیتی خود بدهیم.

۲.۱۰ دیاگرام واحد محاسبه و منطق با رجیستر



شکل ۲۰:

همانگونه که ملاحظه میکنید، رجیستری که ساخته ایم را به بزرگترین و مهم ترین خروجی متصل کردیم. اکنون میتوانیم با خیال راحت عملیات های مورد نظر را انجام دهیم، سپس کلید Set را بزنیم و نتیجه را ذخیره کنیم تا عملیات بعدی. به این شکل خیالمان راحت است که نتیجه درست را میتوانیم تا مدت زمان معینی در جایی ذخیره کنیم.

در این بخش ها، با تمامی اجزای مهم یک پردازنده آشنا شدیم. شما میتوانید پردازنده های بیشتری را مورد بررسی و تحقیق قرار دهید، چرا که تقریبا همه پردازنده های موجود در بازار، مستندات بسیار خوبی دارند که از طریق وبسایت سازندگان آنها، در دسترس است. همچنین میتوانید پردازنده ساده ای که در اینجا ساخته شد را توسعه داده، ورودی های بزرگتر (مثلا ۸ بیتی) و دستورات بیشتر برای آن طراحی کنید. به این شکل، شما میتوانید کارکرد خود ماشین و زبان ماشین را بیشتر و بهتر درک کنید.

۱۱ منابع و مآخذ

English References:

- Digital Design - Morris Mano
- But How do it know? - Scott J. Clark

منابع فارسی:

- طراحی دیجیتال (مدار منطقی) - موریس مانو - ترجمه قدرت سپیدنام
- برنامه نویسی اسمبلی ۸۰۸۶ - مزیدی - ترجمه قدرت سپیدنام