به نام خدا

پروژه پایان‌ترم درس مدارهای منطقی

کیمیا منتظری و سینا شریعتی

۹۹۳۱۰۷۸ و ۹۹۳۱۰۷۲

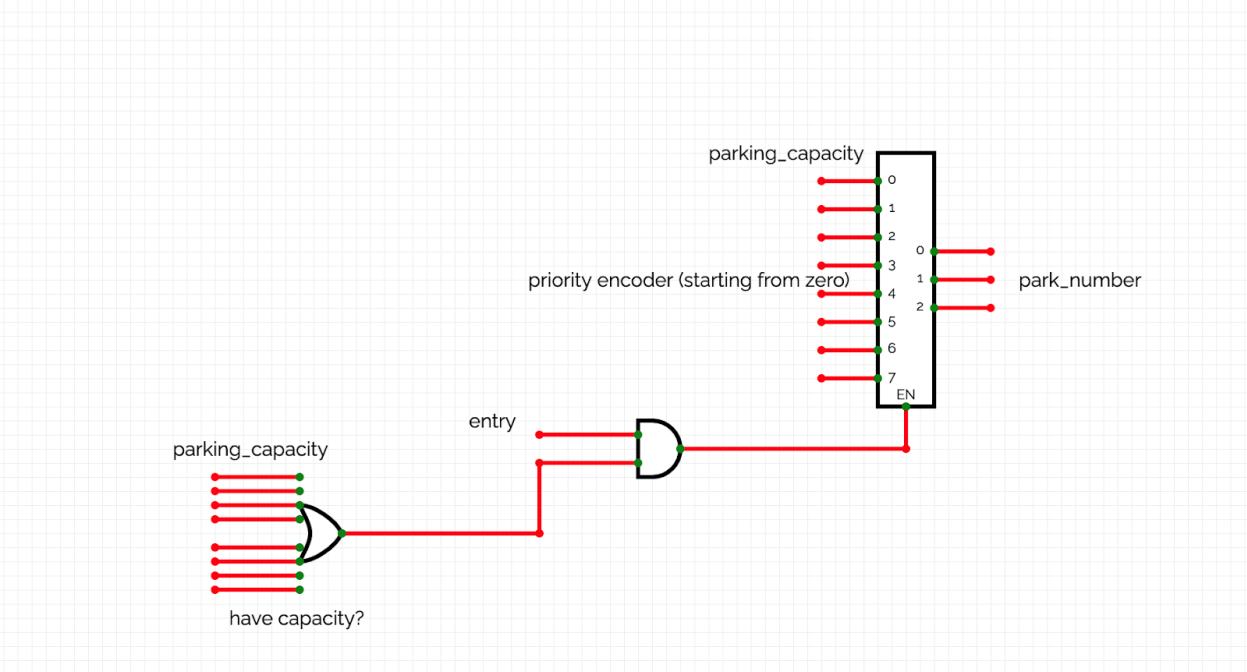
استاد درس: دکتر صاحب الزمانی

استاد آزمایشگاه: زهرا معین نجف آبادی

ماژول ۱

در این ماژول باید برای خودروی وارد شده یک جای پارک پیدا کنیم.

طراحی ساختاری این ماژول به صورت زیر است.



که به این صورت کار میکند که:

ابتدا منتظر می شود که یک ماشین وارد شود و اگر وارد شود entry به صورت active در می اید.

سپس میدانیم که دستور کار پروژه امده است که parking capacity یک vector است که 8 بیت دارد و هر بیت اگر 1 باشد نشان دهنده ی خالی بودن آن جای پارک است.

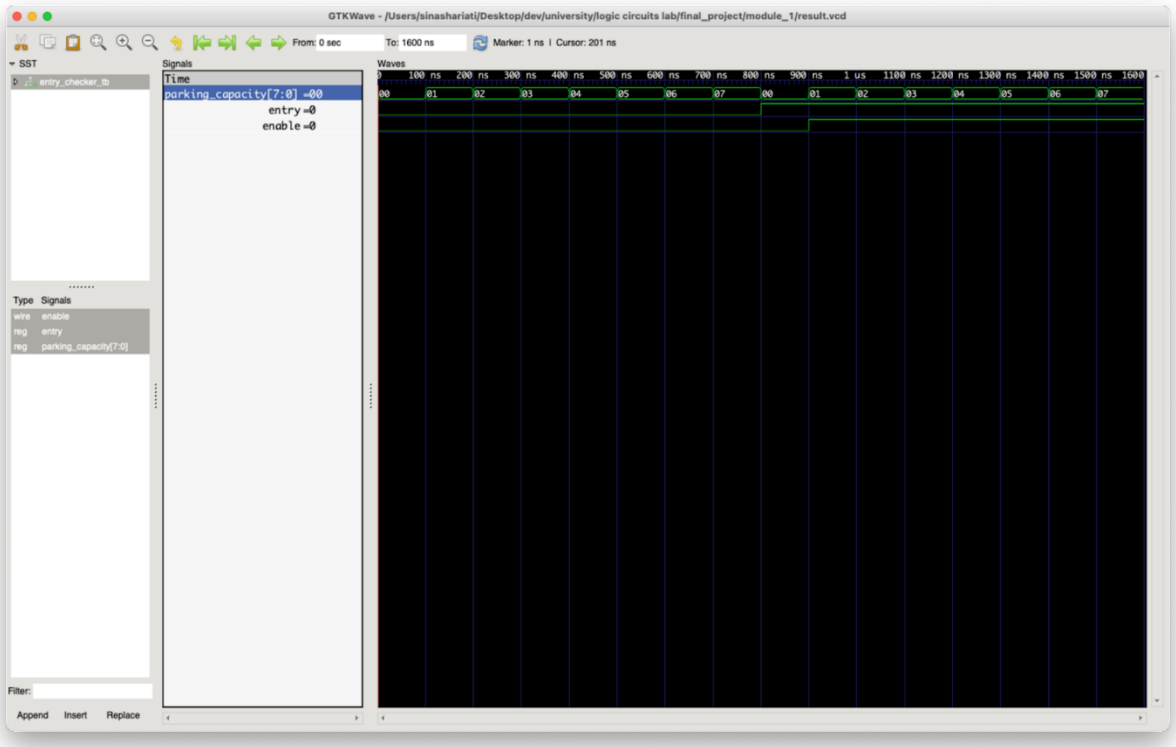
پس اگر یکی از بیت ها 1 باشد یعنی پارکینگ هنوز جا برای پارک ماشین جدید دارد که این را با استفاده از or مشخص شده در طراحی می توان متوجه شد.

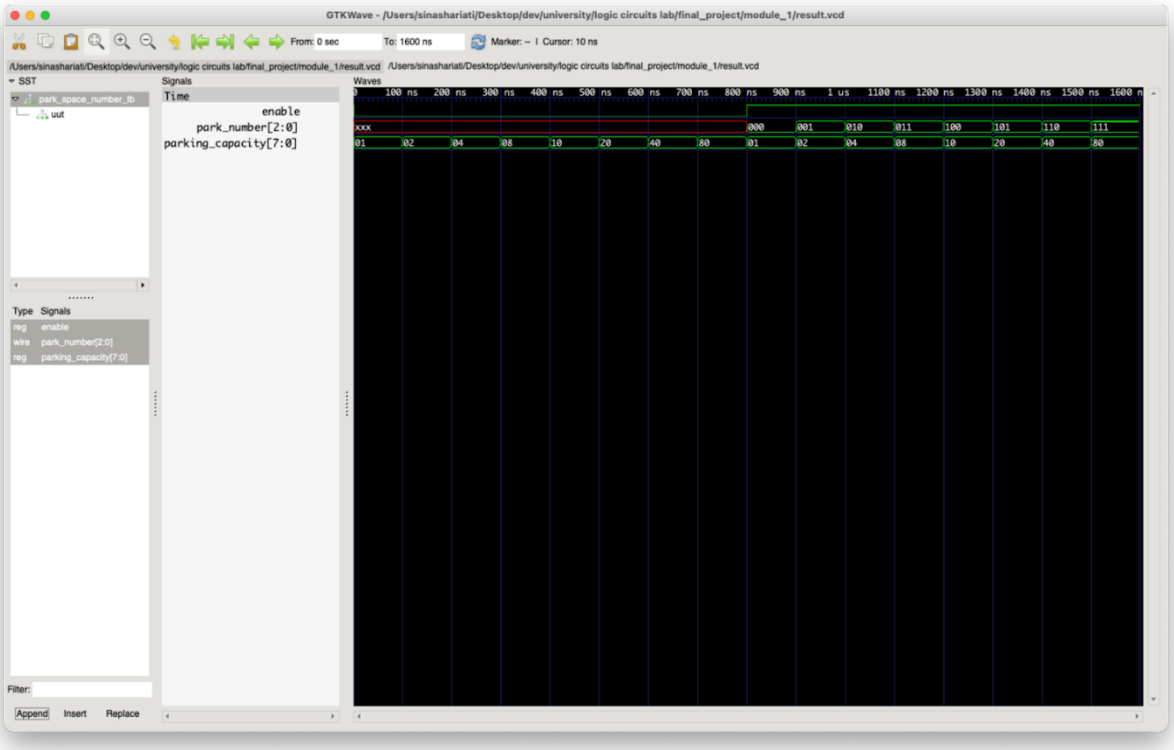
حال که ماشین وارد شده و ما نیز برای ان جا داریم باید آن را به نزدیکترین جای ممکن ببریم که برای این کار از یک priority encoder باید استفاده کنیم. حال چون هیچ testbench یی به ما داده نشد و نه در دستورکار نه توسط مسئول آزمایشگاه یک حالت خاصی مطرح نشد ما فرض میکنیم که هر چه ارزش بیت کمتر باشد ارزش بیشتری دارد.

پس priority encoder ما در واقع low bit priority encoder است و ماشین ها را از 0 تا 7 به ترتیب پخش می کند.

برای اینکه entry park درست کار کند باید به عنوان top level module عمل کند برای entry checker و park space number عمل می کند.

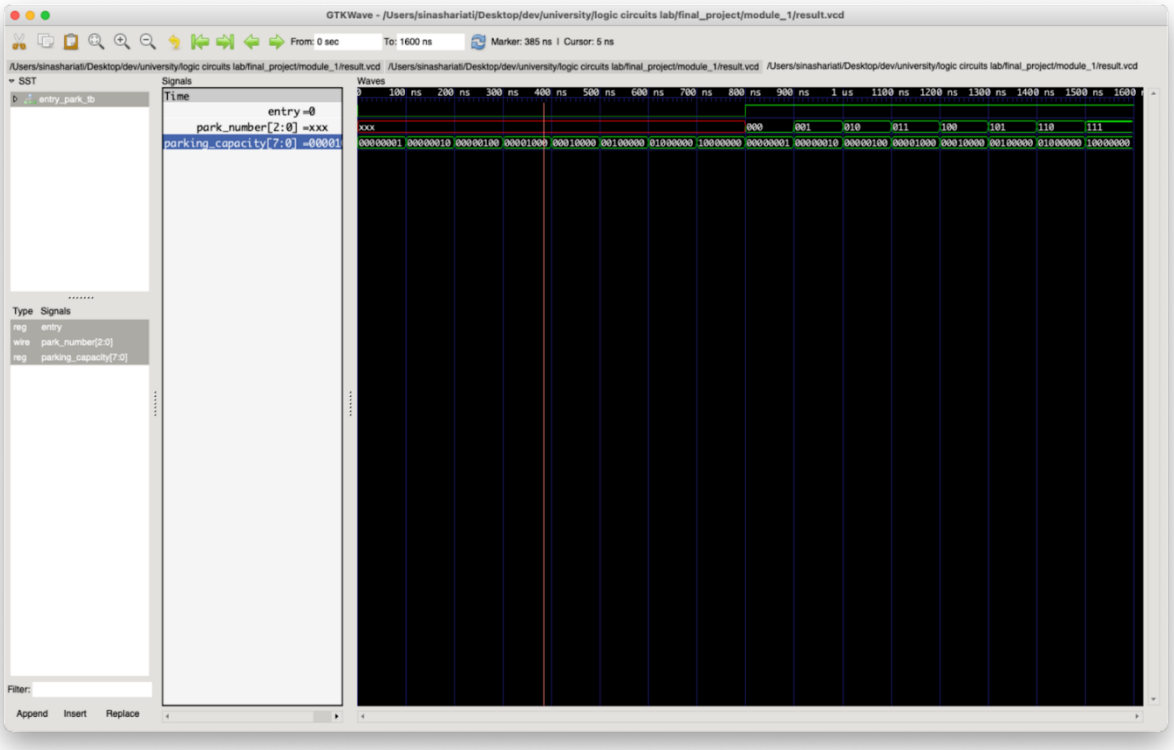
حال اگر testbench های مربوط به entry checker و park space number را به ترتیب run کنیم خروجی های زیر را خواهیم داشت :





توجه داشته باشید که در ماژول park space number زمانی که enable یک نشده باشد نباید هیچ خروجی مشخصی بدهد پس خروجی ان به صورت xxx در نظر گرفته شده است.

حال برای entry park داریم که:

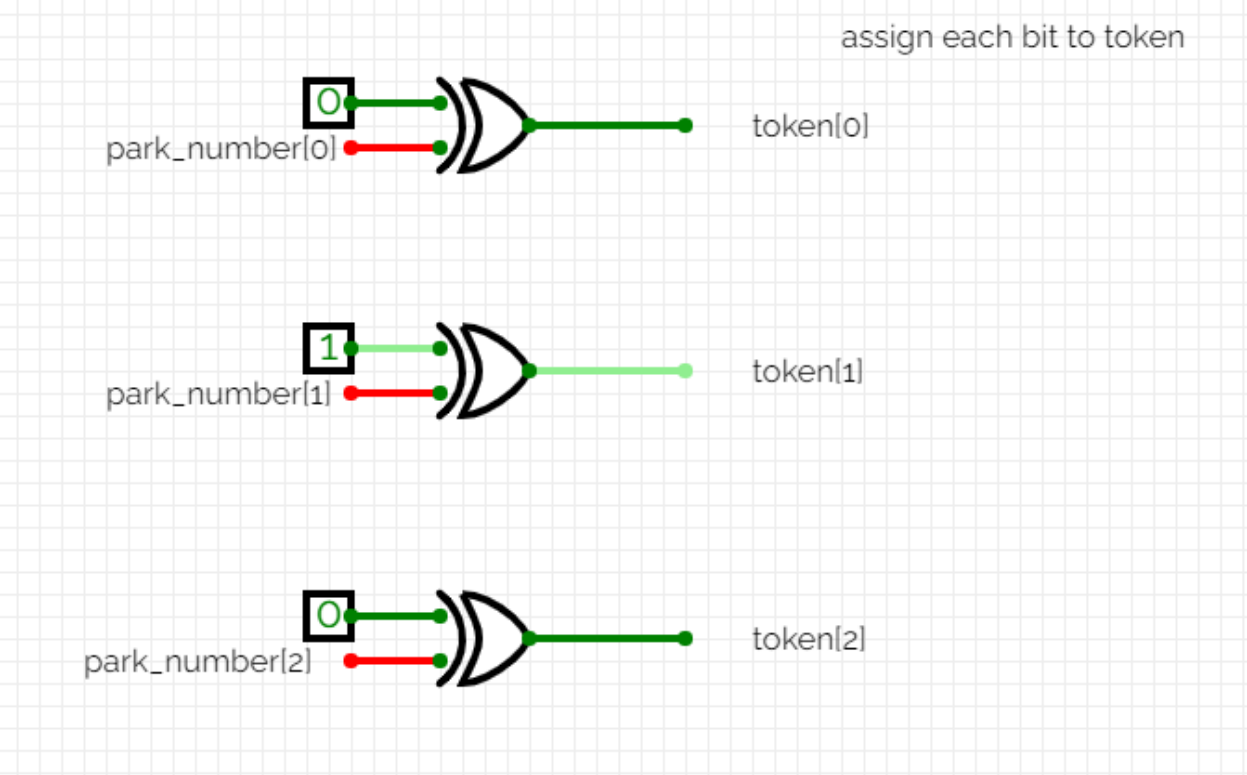


و دلیل وجود xxx در ان به این دلیل این است که اصلا ورودی ماشین نداشته ایم پس نباید هیچ park number یی را تولید کنیم.

ماژول ۲

در این ماژول، به کمک یک پترن ۳ بیتی دلخواه و مشخص – که در طراحی ما ۰۱۰ در نظر گرفته شده است – توانستیم park_number را رمزنگاری کنیم.

این رمزنگاری به کمک گیت xor قابل انجام است؛ به طوری که هر بیت park_number را با بیت متناظرش در پترن، xor کرده‌ایم.



دلیل استفاده از xor

در این روش، پیدا کردن داده رمزنگاری شده بدون داشتن کلید رمزنگاری غیر ممکن است.

ماژول ۳

این ماژول موظف است تا عملیات مربوط به خروج خودرو را انجام دهد، که شامل ۲ بخش است:

- مدار خروج ماشین یا exit_park، که شامل:

  - بدست آوردن park_number از طریق توکن ورودی: کافیست تا توکن را با پترن (۰۱۰) xor کنیم
  - بدست آوردن park_location از طریق park_number به کمک دیکدر

- مدار به روز رسانی ظرفیت قرارگیری خودروها که با داشتن park_location خودروی خارج شده، میتواند ظرفیت کل پارکینگ را به روز رسانی کند:

  - به کمک یک حلقه for بیت‌های park_location را بررسی می‌کنیم که ۱ بودن آن به این معنی است که خودروی خارج شده در این بیت قرار داشته و در نتیجه، باید parking_capacity متناظر را not کنیم.

برای تمام بخش‌های فوق، تست بنچ طراحی شده است تا از درستی عملکرد آنها مطمئن شویم.
لازم به ذکر است که این ماژول به صورت دیتافلو طراحی شده است، زیرا این نوع طراحی سریع‌تر بوده و احتمال خطا را کاهش می‌دهد.

ماژول ۴

این ماژول که برای محاسبه مدت زمان حضور خودرو در پارکینگ طراحی شده است، شامل ۳ بخش می‌باشد:

- مدار جمع/تفریق کننده ۱ بیتی، که به صورت ساختاری پیاده سازی شده است.
- مدار جمع/تفریق کننده ۸ بیتی، که در آن از ۸ مدار جمع/تفریق کننده ۱ بیتی استفاده و به اصطلاح cascade شده است.
- مدار اصلی محاسبه زمان، که در آن اختلاف زمان‌های ورود و خروج خودرو به کمک یک جمع/تفریق کننده ۸ بیتی حساب می‌شود.

در ادامه، برای هر یک از بخش‌های فوق تست بنچ طراحی شد تا از درستی عملکرد هر یک مطمئن شویم.

ماژول ۵

در این ماژول باید به صورت hierarchical سیستمی طراحی کنیم که در نهایت تعداد صفر ها و یک های موجود در new capacity را بشمارد و به ما خروجی بدهد.

به صورت کلی می دانیم اگر تعداد 1 ها در new capacity بشماریم و مطمئن شویم که empty به درستی به دست آمده است می توانیم برای parked عدد 8 را از empty کم کنیم و به جواب برسیم.

پس ابتدا به دنبال پیدا کردن راهی hierarchical می رویم که تعداد 1 های موجود در یک عدد 8 بیتی را بشماریم:

اگر یک عدد 8 بیتی داشته باشیم می توانیم به سه دسته ی زیر بیت های ان را تقسیم کنیم:

1. بیت های شماره ی 0,1,2
2. بیت های شماره ی 3,4,5
3. بیت های شماره ی 6,7

حال اگر برای گروه های شماره ی 1,2 یک full adder داشته باشیم و برای گروه 3 از یک half adder استفاده کنیم هر گروه به ترتیب خروجی های زیر را تولید می کند:

1. result_1 , carry_1
2. result_2 , carry_2
3. result_3, carry_3

حال منطقی که نرم افزار ما دارد به این صورت است که تمام result های درست شده را دوباره به یک full adder داده و از ان خروجی های زیر را می گیرد:

- results_sum_result : حاصل جمع بیت های مربوط به result ها.
- results_sum_carry: خروجی مربوط به carry در جمع کردن همه result ها.

حال همین کار را برای carry های تولید شده در مرحله ی قبل تولید میکند:

- carries_sum_result: حاصل جمع بیت های مربوط به carry ها.
- carries_sum_carry: خروجی carry مربوط به جمع کردن carry ها.

حال اگر بیت های تولید شده را به صورت زیر باهم جمع کنیم دقیقا تعداد 1 ها را به ما میدهد:

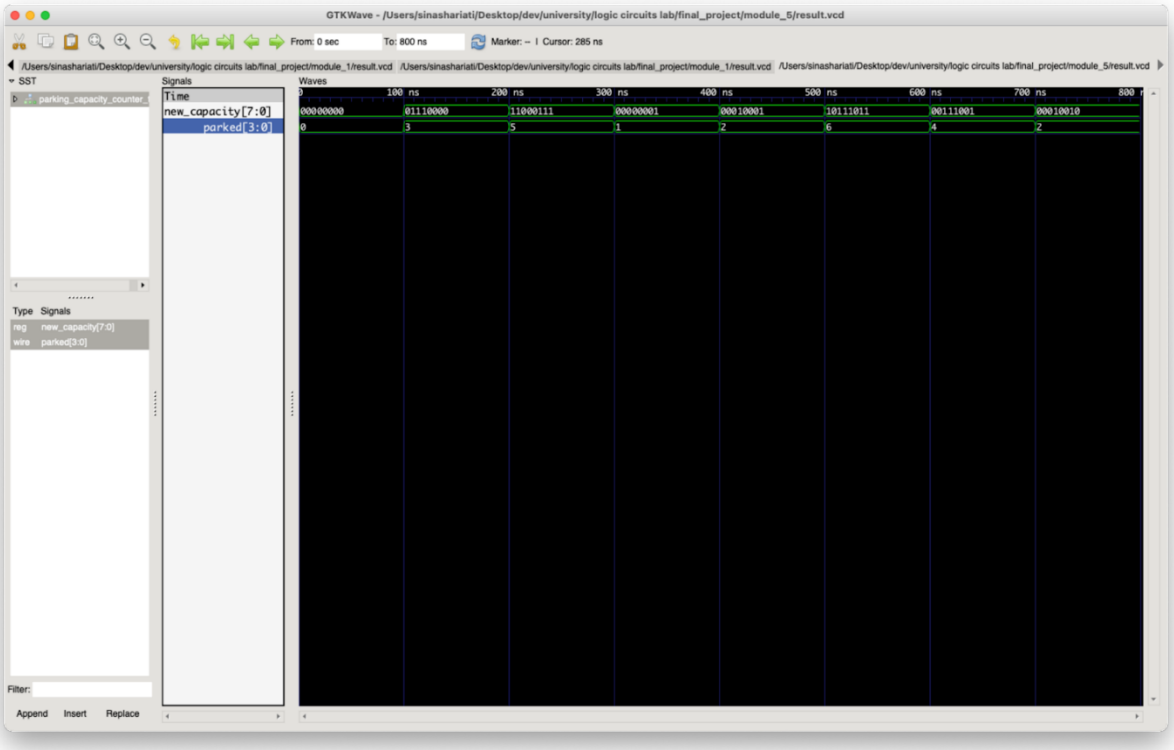
Parked[0] = 0 + results_sum_result

Parked[1] = carries_sum_result + results_sum_carry

Parked[2] = bit_1_carry + carries_sum_carry

Parked[3] = 1 + bit_2_carry

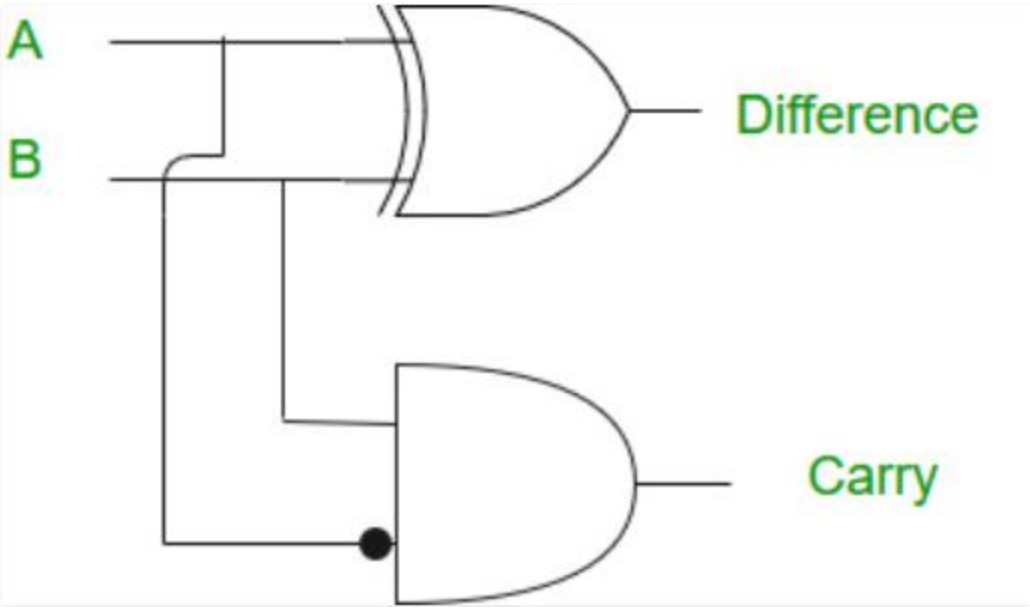
نکته: نمی توان مطمئن بود که این روش برای هر پارکینگی با هر ظرفیتی کار میکند؛ اما همانطور که در testbench پیوست شده امده است برای تمام حالت های یک پارکینگ 8 جای خالی درست کار میکند:

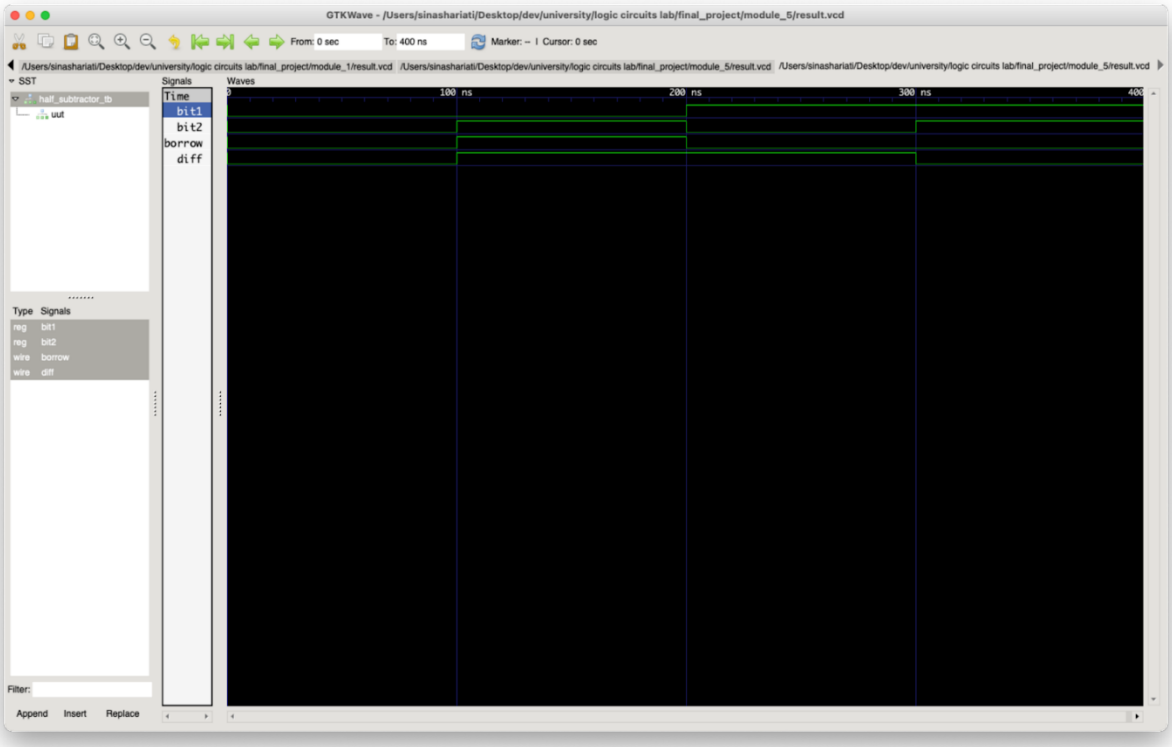


حال که تعداد 1 ها را داریم صرفا تنها مرحله ای که باقی مانده است این است که عدد 8 را از عدد بدست آمده برای empty کم کرد و خروجی parked را تولید کرد.

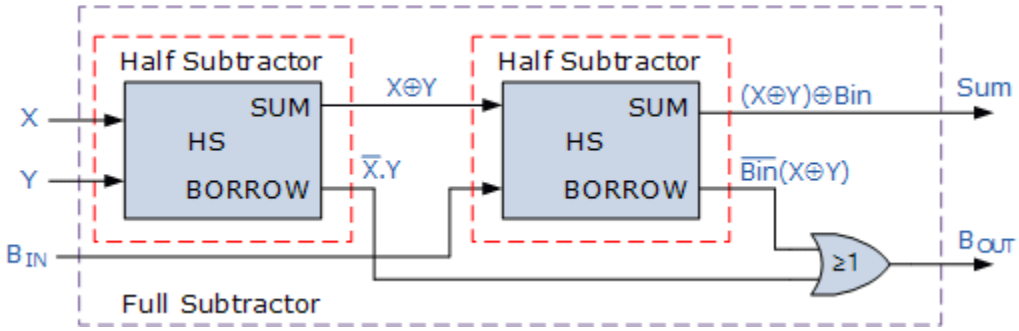
نکته ای که در اینجا مهم است این است که این کار را باید به صورت hierarchical انجام دهیم و از طرفی به این دلیل که ما از این ماژول هیچ وقت به عنوان جمع کننده استفاده نمیکنیم ؛ منطقی نبود که یک adder subtractor بسازیم پس یک bit subtractor4 ساختیم.

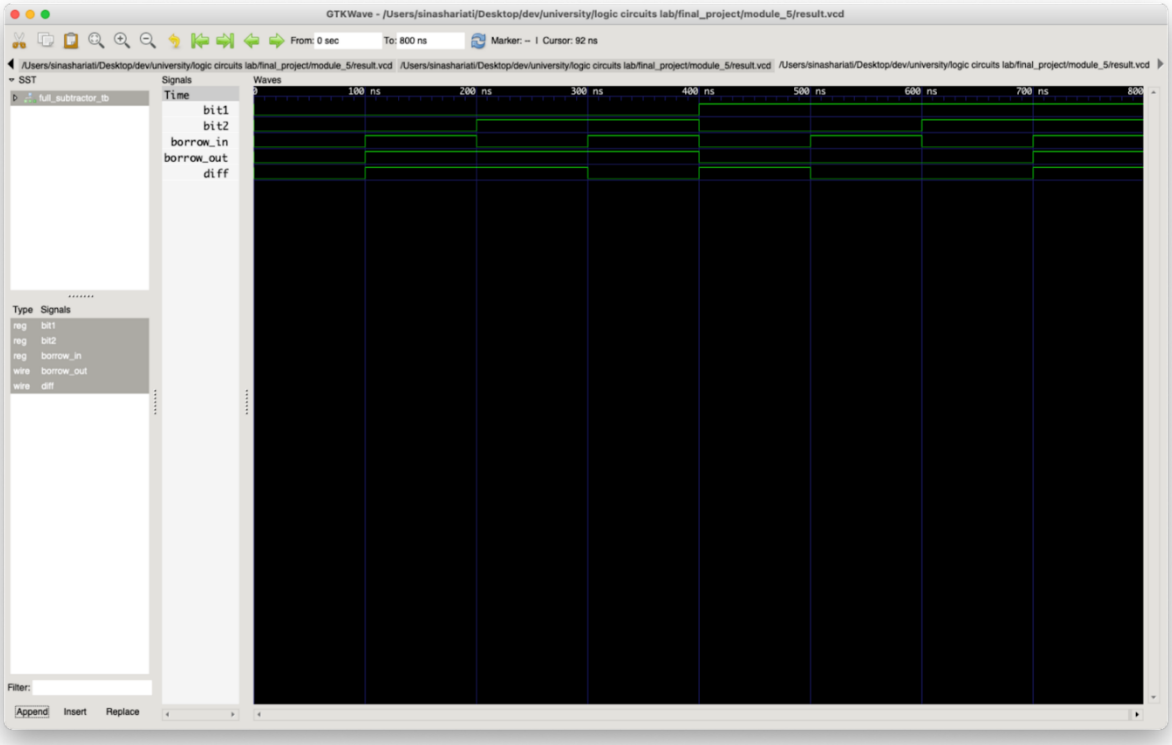
در کلاس درس subtractor تدریس نشد اما می توان با استفاده از ساختاری که برای adder توضیح داده شد به راحتی subtractor هم تولید کرد. به این صورت که ابتدا یک half subtractor میسازیم که وظیفه ی آن از هم کم کردن ۲ ورودی است که ساختار و خروجی test bench ان به صورت زیر است:



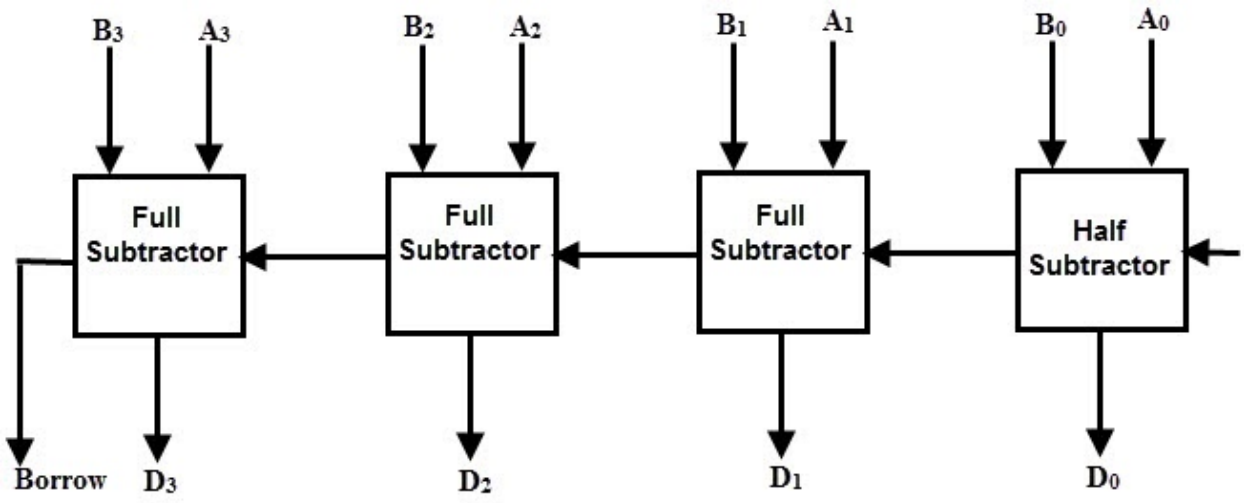


حال با استفاده از این half subtractor یک full subtractor می سازیم که ساختار و خروجی ان به صورت زیر است:

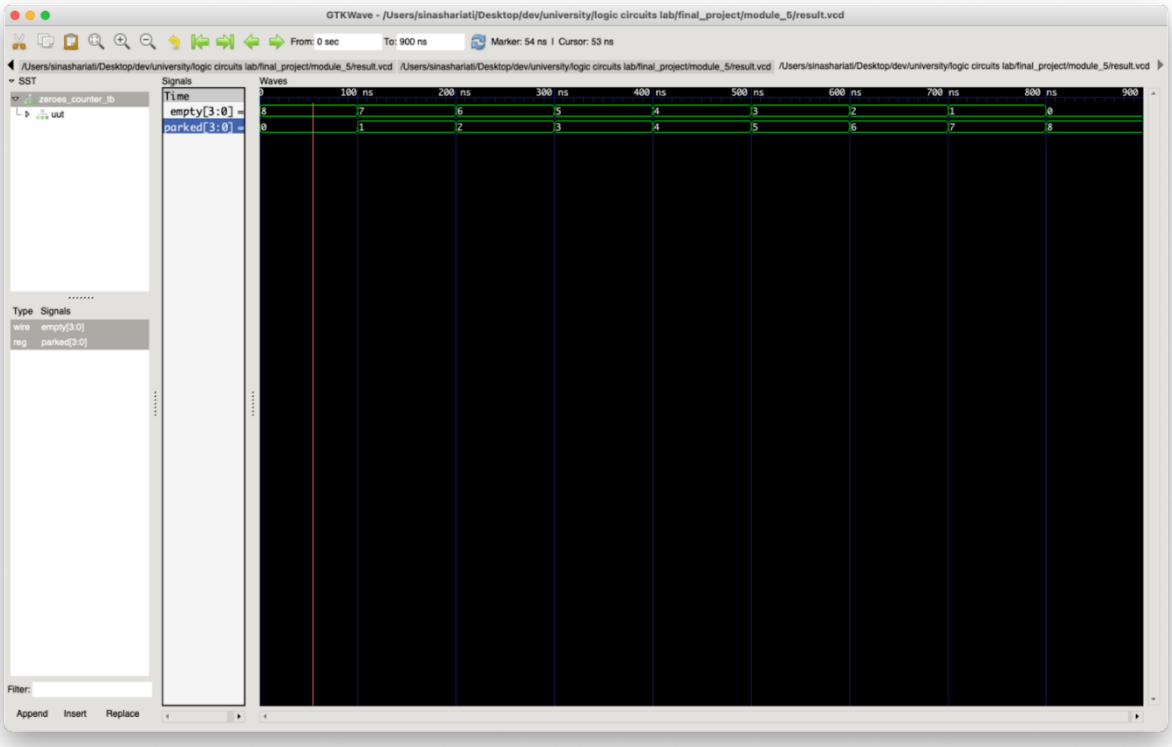




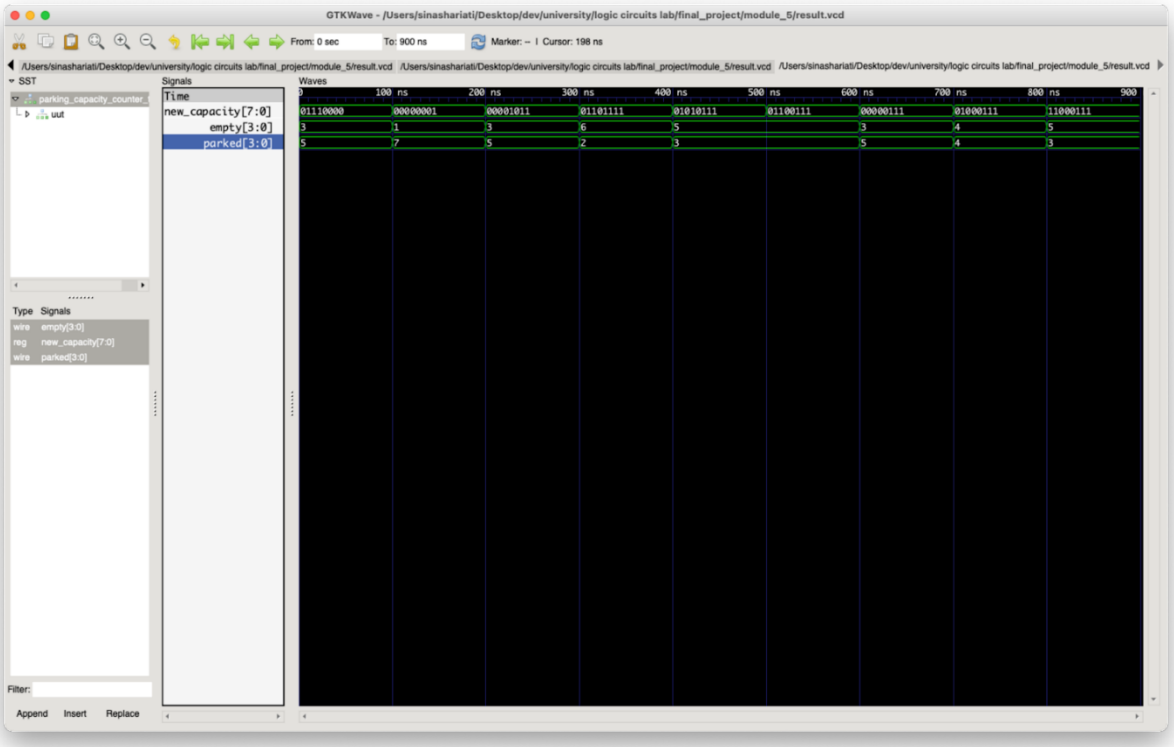
حال با استفاده از full subtractor زیر یک subtractor می سازیم که توانایی subtract کردن ۴ بیت را از هم دارد.



حال می توانیم که $(1000)_2 = (8)_{10}$ را از empty کم کنیم و یک عدد ۴ بیتی تولید کنیم که تعداد parked ها را نشان میدهد و خروجی ان به صورت زیر خواهد بود :



حال که هم تعداد صفر ها و تعداد یک ها درست می شماریم می توانیم top level module را نیز که همان parking capacity counter است نیز تست کنیم که مطمین شویم تمام اجزا به درستی کنار هم قرار میگیرند :



که همانطور که در عکس بالا آمده است کاملا درست کار میکنند.

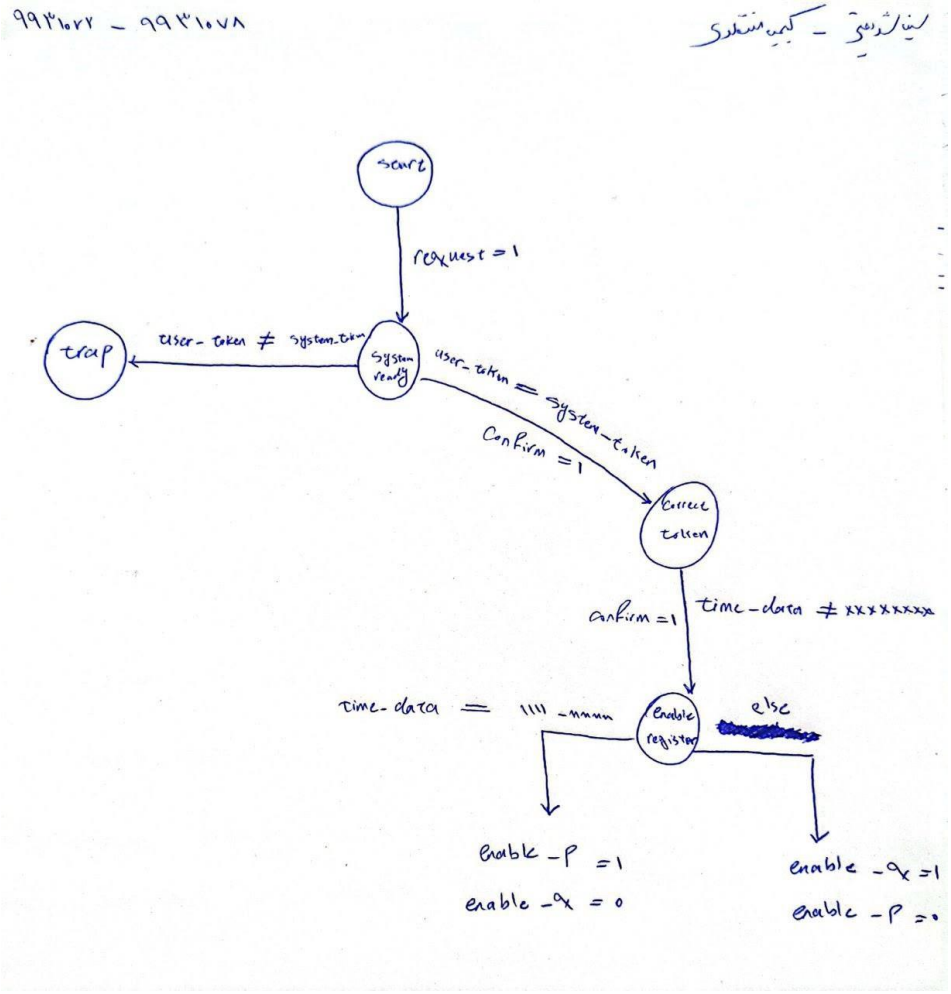
ماژول ۶

این ماژول وظیفه وصل کردن اجزای ماژول‌های ۱ تا ۵ را دارد.
کافی است تا بر اساس نمودار داده شده در صورت پروژه، top-level module هر یک را instantiate کنیم و ورودی و خروجی مناسب را به پورت‌های آن بدهیم.
لازم به ذکر است که ماژول‌های update_capacity و calculate_new_capacity موظف هستند تا ظرفیت پارکینگ را در حالت‌های مختلف، به روز رسانی کنند.
در آخر به کمک تست بنچ داده شده، درستی عملکرد هر بخش را بررسی کردیم.

ماژول ۷

در این ماژول ما باید یک fsm را درست می کردیم که کار آن مدیریت قسمت پرداخت ذخیره سازی سیستم بود.

در این fsm یک دیاگرام حالت به شکل زیر پیاده شده است:



که طبق صورت پروژه طراحی شده است.

در هر کدام از state های مطرح شده اگر request = 0 شود دوباره به حالت start برمیگردیم که کمی جدول را ناخوانا می کرد به همین دلیل کشیده نشده است.