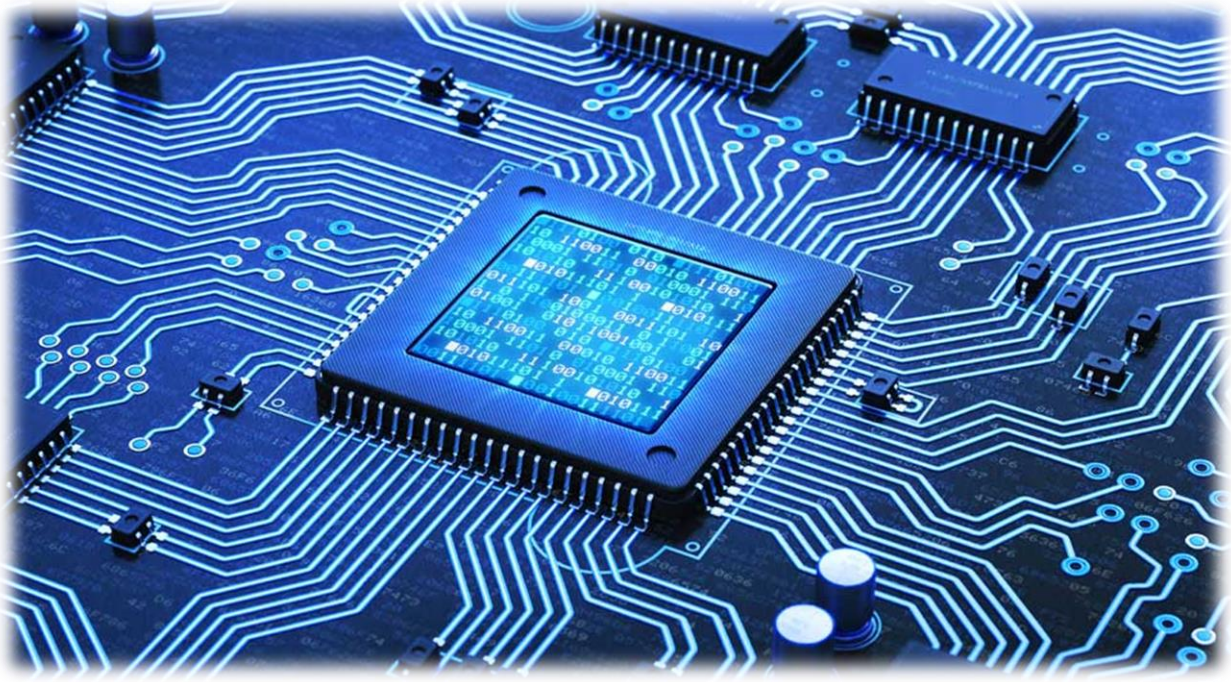
به نام خدا



پروژه اول معماری کامپوتر

دکتر سعید صفری

پریسا محمدی ۸۱۰۱۰۱۵۰۹ حامد گل کار ۸۱۰۱۰۱۴۹۹

بهار ۱۴۰۴

مقدمه

هدف از این پروژه، پیاده‌سازی سخت‌افزاری سیستمی است که بتوان به کمک آن مسیریابی یک موش در یک هزارتو را انجام داد و در نهایت موش را به نقطه مقصد هدایت کرد. البته باید توجه داشت که ممکن است مسیری برای رسیدن موش به مقصد وجود نداشته باشد؛ در چنین شرایطی سیستم موظف است تا سیگنال Fail را صادر کند. برای پیاده‌سازی این پروژه، از مدل هافمن استفاده می‌کنیم. این مدل از دو بخش اصلی تشکیل شده است: بخش اول که به‌صورت Combinational و مستقل از کلاک عمل می‌کند و وظیفه‌ی آن محاسبه‌ی سیگنال‌های مربوطه با استفاده از المان‌های محاسباتی وlatch ها است، و بخش دوم که به‌صورت Sequential عمل کرده و حساس به لبه‌ی بالارونده کلاک است. این بخش مسئول کنترل انتقال از یک وضعیت (State) به وضعیت دیگر با توجه به سیگنال‌های ورودی است.

در نهایت ماژولی تحت عنوان top module طراحی شده که وظیفه آن اتصال سیگنال‌های ورودی و خروجی بخش‌های Sequential و Combinational به یکدیگر است. ما به عنوان کاربر تنها از طریق این ماژول با سیستم تعامل داریم و سیگنال‌های کنترلی مانند شروع عملیات را به آن اعمال می‌کنیم. درون این ماژول با توجه به عملکرد بخش‌های محاسباتی و کنترلی، خروجی مرحله به مرحله محاسبه و به روز می‌شود تا در نهایت پس از گذشت زمان کافی، به مقدار مورد نظر دست پیدا کنیم.

ماژول حافظه هزارتو:

وظیفه این ماژول ذخیره‌سازی اطلاعات مربوط به جدول هزارتو است. ابعاد جدول هزارتوی ما ۱۶ در ۱۶ است و برای ایجاد این ساختار ذخیره‌سازی، یک آرایه دوبعدی از رجیسترها تعریف کرده‌ایم که شامل ۱۶ ردیف و ۱۶ ستون با اندیس‌های ۰ تا ۱۵ است. تعریف این آرایه در زبان وریلاگ به صورت زیر است:

```
reg [15:0] mem_16x16 [15:0];
```

در ادامه باید جدول حافظه را با اطلاعات مورد نظر خود پر کنیم. اطلاعات مربوط به هر خانه در یک فایل متنی (`.txt`) ذخیره شده‌اند که برای بارگذاری آنها در جدول حافظه، از دستور **$readmemb** استفاده می‌کنیم. حرف `b`در انتهای این دستور نشان‌دهنده این است که محتوای فایل به صورت باینری (۰ یا ۱) خوانده می‌شود. طبق صورت پروژه، عدد ۰ نشان‌دهنده خانه خالی و عدد ۱ نشان‌دهنده خانه پر یا دیوار است. ورودی‌های این دستور، مسیر دقیق فایل روی کامپیوتر و محل ذخیره‌سازی در حافظه‌ای است که در این ماژول تعریف کرده‌ایم.

علاوه بر بارگذاری اولیه، ماژول حافظه باید امکان خواندن و نوشتن اطلاعات را نیز فراهم کند. به همین منظور دو سیگنال کنترلی ورودی به نام‌های `write` و`read`تعریف کرده‌ایم. خواندن از حافظه به صورت ناهمگام با کلاک (asynchronous) انجام می‌شود، در حالی که نوشتن در حافظه هماهنگ با لبه بالارونده کلاک (posedge clk) است.

طبق صورت مسئله، مسیر حرکت در جدول هزارتو باید از گوشه پایین سمت چپ آغاز شده و به گوشه بالا سمت راست ختم شود. بنابراین زمانی که می‌خواهیم به مختصات خاصی از جدول دسترسی داشته باشیم، افزایش مقدار x ما را به سمت راست جدول هدایت می‌کند و افزایش مقدار y نیز ما را به سمت ردیف‌های بالاتر می‌برد. در نتیجه برای دسترسی درست به خانه موردنظر در حافظه، از مختصات زیر استفاده می‌کنیم:

```
mem_16x16[15 - Y][15 - X];
```

ماژول مسیر داده:

این ماژول شامل مسیر داده‌ی کلی سخت‌افزار است. به بیان دیگر، در این ماژول واحد محاسباتی برای پیدا کردن مسیر صحیح و همچنین ساختار داده‌ای (استک) جهت ذخیره کردن مسیر طی‌شده وجود دارد. علاوه بر این، چنانچه موفق به پیدا کردن مسیر شویم، بخش محاسباتی دیگری برای نمایش کل مسیر به‌صورت صف نیز در این ماژول تعریف شده است که در ادامه هر بخش را به تفصیل شرح می‌دهیم.

ورودی‌های این ماژول شامل سیگنال‌های مرتبط با زمان، یعنی کلاک (clk) و ریست (rst) هستند. همچنین سیگنال‌های کنترلی ورودی نیز وجود دارند که از سمت واحد کنترل دریافت می‌شوند. با تغییر مقدار این سیگنال‌های کنترلی، واحدهای محاسباتی شروع به فعالیت کرده و خروجی‌های مربوطه را تولید می‌کنند. خروجی‌های این ماژول نیز شامل سیگنال‌هایی هستند که بر اساس مقادیر آن‌ها و طبق مدل هافمن، واحد کنترل تصمیم می‌گیرد که در استیت فعلی باقی بماند یا به استیت بعدی منتقل شود. علاوه بر این خروجی‌ها، مقدار هشت‌بیتی به نام Move نیز تولید می‌شود که برابر با مقداری است که از dequeue کردن صف مسیر به دست می‌آید. در این هشت بیت، ۴ بیت کم‌ارزش نشان‌دهنده مختصات y خانه، و ۴ بیت پرارزش نمایانگر مختصات x آن خانه در مسیر هستند.

حال بخش‌های مختلف مسیر داده را از ابتدای الگوریتمِ پیدا کردن مسیر تا مرحله نمایش مسیر نهایی توضیح می‌دهیم.

برای پیدا کردن مسیر مناسب، وقتی خانه‌های جدول را طی می‌کنیم، نیاز داریم تا در هر خانه وضعیت خانه‌های اطراف آن را بررسی کنیم تا جهت حرکت بعدی را مشخص کنیم. همچنین اگر به بن‌بست برسیم، باید امکان بازگشت به خانه‌های قبلی وجود داشته باشد. بنابراین لازم است مختصات مسیر طی‌شده را ذخیره کنیم. طبق خواسته پروژه، برای این منظور از یک استک استفاده می‌کنیم. این استک شامل ۲۵۶ خانه ۸ بیتی است و تعریف آن به صورت زیر است:

```
reg [7:0] main_stack [255:0];
```

طبق صورت پروژه، مسیر حرکت را باید از مختصات (۰,۰) آغاز کنیم. به همین دلیل، در یک بلاک هماهنگ با کلاک بررسی می‌کنیم که اگر سیگنال ورودی start برابر با یک شود، مختصات (۰,۰) را به‌عنوان خانه شروع مسیر در نظر می‌گیریم و آن را در اولین خانه استک ذخیره می‌کنیم. همچنین نقطه مقصد که طبق صورت پروژه در مختصات (۱۵,۱۵) قرار دارد، در خانه آخر استک ذخیره می‌شود.

برای حرکت در مسیر و به‌روزرسانی استک لازم است موقعیت فعلی خود را در استک بدانیم. برای این منظور از یک متغیر به نام Count_256 استفاده کرده‌ایم. در ابتدا این متغیر را برابر با صفر قرار می‌دهیم و سپس هر بار که به خانه بعدی حرکت می‌کنیم، مقدار آن را یک واحد افزایش می‌دهیم. همچنین هر بار که با بن‌بست مواجه می‌شویم و نیاز است به خانه‌های قبلی بازگردیم، برای دسترسی به خانه قبلی مقدار آن را یک واحد کاهش می‌دهیم. این عملیات به‌صورت هماهنگ با لبه بالارونده‌ی کلاک انجام می‌شود.

در هر خانه‌ای که قرار می‌گیریم، باید بتوانیم تشخیص دهیم که حرکت بعدی ما به کدام‌یک از خانه‌های اطراف خواهد بود. طبق صورت مسئله، باید خانه‌های اطراف خانه‌ی فعلی را بررسی کرده و مشخص کنیم که خانه‌ی مدنظر، خالی است یا قبلاً از آن عبور کرده‌ایم و یا دیوار است و امکان ورود به آن وجود ندارد. اولویت بررسی خانه‌ها به ترتیب: بالا، راست، چپ و پایین است. برای

مدیریت ترتیب بررسی خانه‌ها از یک متغیر به نام Count_4 استفاده می‌کنیم. مقدار این متغیر در ابتدا صفر است و به کمک یک ساختار case مقدار صفر آن را به بررسی خانه‌ی بالا اختصاص می‌دهیم. در مراحل کنترلی، محتوای خانه‌ای که به آن اشاره می‌کند را می‌خوانیم. اگر خانه‌ی مورد نظر خالی باشد (برابر صفر)، بر اساس منطقِ واحد کنترلی که در ادامه توضیح می‌دهیم، دیگر نیازی به بررسی خانه‌های دیگر اطراف نخواهیم داشت؛ اما اگر مقدار آن خانه برابر یک باشد (خانه پر یا دیوار باشد)، باید به سراغ خانه‌ی بعدی در اولویت برویم. برای این کار، مقدار Count_4 را یک واحد افزایش می‌دهیم و مجدد در ساختار case، خانه‌ی بعدی را بررسی می‌کنیم. این روند ادامه پیدا می‌کند تا هر چهار خانه بررسی شوند.

در مسیر بررسی خانه‌های اطراف، یک مشکل دیگر ممکن است پیش آید و آن رسیدن به مرزهای جدول است؛ یعنی ممکن است در جهتی که قصد حرکت به آن را داریم خانه‌ای وجود نداشته باشد. برای تشخیص این مسئله باید بررسی کنیم که آیا حرکت بعدی موجب عبور از مرزهای جدول می‌شود یا نه. از آنجایی که مختصات خانه‌ها (x,y) از ۰ تا ۱۵ هستند و نمی‌توان با اعداد چهار بیتی، عددی کمتر از صفر یا بیشتر از ۱۵ داشت، از روشی جایگزین استفاده کرده‌ایم. بدین صورت که بررسی می‌کنیم آیا مختصات فعلی در مرز (۰ یا ۱۵) قرار دارد و حرکت بعدی (بر اساس مقدار Count_4) باعث عبور از مرز می‌شود یا خیر. اگر با هر یک از چهار حرکت ممکن، از مرز عبور کنیم، یک سیگنال خروجی را برابر با یک می‌کنیم و این سیگنال به واحد کنترل منتقل می‌شود.

علاوه بر این، هرگاه وارد خانه‌ای جدید می‌شویم، باید مقدار آن خانه را در حافظه جدول ۱۶ در ۱۶ به‌روز کنیم تا در صورت بازگشت به عقب و بررسی دوباره خانه‌ها، بدانیم که این خانه قبلاً بازدید شده و مسیر جدیدی نیست. برای این منظور از ماژول حافظه (memory) نمونه‌برداری (instance) می‌کنیم و سیگنال‌های کنترلی read و writeرا به آن اعمال می‌کنیم. این ماژول علاوه بر به‌روزرسانی حافظه، برای خواندن محتوای خانه‌های جدول نیز استفاده می‌شود.

سیگنال co_4 به این معناست که ما به بن‌بست رسیده‌ایم و هر چهار خانه‌ی اطراف خانه‌ی فعلی را بررسی کرده‌ایم اما هیچ‌کدام قابل ورود نبوده‌اند. بررسی این شرایط به کمک متغیر Count_4 انجام می‌شود؛ یعنی وقتی مقدار Count_4 به ۳ برسد، نشان‌دهنده‌ی این است که تمامی خانه‌های اطراف بررسی شده‌اند.

سیگنال دیگری به نام empty داریم که به ما اطلاع می‌دهد آیا پیدا کردن مسیری از نقطه شروع تا مقصد امکان‌پذیر است یا خیر. این سیگنال با بررسی مقدار متغیر Count_256 تعیین می‌شود. زمانی که به بن‌بست می‌خوریم، لازم است به خانه قبلی در استک بازگردیم و دوباره بررسی کنیم که آیا امکان حرکت از مسیر دیگری وجود دارد یا خیر. برای دسترسی به خانه قبلی، مقدار متغیر Count_256 را یک واحد کم می‌کنیم و سپس خانه‌های اطراف را دوباره بررسی می‌کنیم. حال اگر باز هم هیچ مسیری در اطراف این خانه قابل حرکت نبود، مجدداً مقدار Count_256 را کم کرده و یک خانه دیگر به عقب برمی‌گردیم. اگر همین روند ادامه یابد و ما همواره با بن‌بست مواجه شویم، در نهایت به خانه اول مسیر می‌رسیم و مقدار Count_256 برابر صفر خواهد شد. در این حالت سیگنال empty فعال شده و گزارش می‌دهد که از مسیر اولیه، رسیدن به مقصد امکان‌پذیر نیست.

برای بررسی اینکه آیا به مقصد رسیده‌ایم یا نه، کافی است مختصات فعلی خود را با مختصات خانه‌ی مقصد (که قبلاً در خانه‌ی آخر استک ذخیره کرده بودیم) مقایسه کنیم.

حال به توضیح بخش محاسباتی برای نمایش مسیرِ پیدا شده می‌پردازیم. طبق صورت مسئله، برای نشان دادن مسیرِ پیدا شده باید از یک ساختار داده‌ای به صورت صف استفاده کنیم. برای دسترسی به عناصر صف(dequeue کردن آن‌ها)، نیاز به یک اشاره‌گر

داریم که ما آن را متغیر Count_queue در نظر گرفته‌ایم. پس از اتمام فرایند پیدا کردن مسیر و پیش از شروع نمایش آن، این اشاره‌گر را برابر صفر قرار می‌دهیم. در واقع آرایه‌ای که به‌عنوان صف برای نمایش مسیر استفاده می‌کنیم همان آرایه استک (main_stack)است، زیرا این استک کل مسیر را در خود ذخیره کرده است و همچنین می‌دانیم که خواندن از صف باید از ابتدای آن شروع شود. مسیرِ ذخیره‌شده در استک ما به گونه‌ای است که خانه‌ی ابتدای مسیر در ابتدای استک ذخیره شده و می‌توانیم با خواندن به ترتیب از ابتدای آن، مختصات مسیر صحیح را به دست آوریم.

برای پیاده‌سازی عملیات dequeue از صف، مقدار موجود در خانه‌ای از استک که اشاره‌گر Count_queue به آن اشاره دارد را در متغیری به نام Move بارگذاری می‌کنیم و سپس یک واحد به اشاره‌گر Count_queue اضافه می‌کنیم تا در گام بعدی به عنصر بعدی مسیر دسترسی پیدا کنیم.

مقدار Move باید به عنوان خروجی این ماژول در دسترس باشد، چرا که با هر پالس کلاک، مقدار جدیدی از مختصات مسیر را ارائه می‌دهد. همچنین خروجی این ماژول به عنوان خروجیِ ماژول top ما نیز محسوب می‌شود. از آنجایی که وقتی از این ماژول در تست‌بنچ خود instance می‌گیریم خروجی آن باید از نوع wire باشد (نه از نوع رجیستر)، لازم است در نهایت آن را به یک سیگنال wire متصل کنیم.

در نهایت، متغیری به نام reached_the_end تعریف کرده‌ایم که وظیفه‌اش این است که به ما اطلاع دهد آیا تمام مسیر را خوانده‌ایم و به انتهای صفِ حاوی مختصات خانه‌ها رسیده‌ایم یا خیر. متغیر Count_256 مقدار تقریبی طول مسیر را در خود دارد، اما طبق استیت‌های بخش کنترلی، پس از اینکه متوجه شدیم در حرکت بعدی به مقصد می‌رسیم دیگر مقدار آن را افزایش نمی‌دهیم؛ به همین دلیل باید یک واحد به مقدار آن اضافه کنیم تا به آخرین خانه مسیر دسترسی پیدا کنیم و یک واحد دیگر نیز اضافه می‌کنیم تا زمانی که اشاره‌گر از آخرین خانه مسیر عبور کرد، متوجه شویم که نمایش مسیر به پایان رسیده است.

ماژول کنترلر:

این ماژول بر اساس مدل هافمن، به‌عنوان مکملی برای ماژول مسیر داده عمل می‌کند. هدف این ماژول کنترل زمان‌بندی مناسبِ عملیات مختلف است تا در هر وضعیت، سیگنال‌های کنترلی لازم را به ماژول مسیر داده بفرستد و محاسبات آن ماژول را در زمان صحیح فعال کند.

ورودی‌های این ماژول شامل سیگنال‌های خروجی ماژول مسیر داده هستند که با توجه به مقدار آن‌ها تصمیم گرفته می‌شود در چه استیتی قرار بگیریم. خروجی‌های این ماژول نیز سیگنال‌های کنترلی هستند که به ماژول مسیر داده ارسال شده و باعث شروع عملیات‌های محاسباتی مربوطه می‌شوند.

برای پیاده‌سازی کنترلر از یک ماشین حالت (FSM) استفاده کرده‌ایم که در مدل هافمن رایج است. این FSM شامل ۲۰ استیت است که در ادامه به شرح دقیق و ارتباط آن‌ها با یکدیگر می‌پردازیم:

۱. Idle:

این استیت وضعیت آغازین سیستم است که در آن منتظر دریافت سیگنال شروع هستیم تا فرآیند یافتن مسیر از مبدأ به مقصد آغاز شود. تا زمانی که سیگنال start از ورودی دریافت نشده باشد، در این استیت می‌مانیم. به محض دریافت

سیگنال start به استیت initializing می‌رویم. در این مرحله همچنین اشاره‌گر استک (Count_256) را مقداردهی اولیه می‌کنیم.

۲. Initializing:

در این مرحله سیگنال شروع دریافت شده و آماده شروع فرایند یافتن مسیر هستیم. برای بررسی خانه‌های اطراف هر خانه و انتخاب مسیر بعدی نیاز به اشاره‌گر جهت‌ها (Count_4) داریم که در این مرحله این اشاره‌گر را نیز به صفر مقداردهی اولیه می‌کنیم.

۳. checking_border:

برای هر یک از خانه‌های اطراف خانه کنونی باید بررسی کنیم که آیا حرکت در جهت مورد نظر موجب عبور از مرز جدول می‌شود یا خیر. اگر عبور از مرز رخ دهد ولی هنوز خانه‌های دیگری برای بررسی مانده باشند، باید به سراغ خانه بعدی برویم؛ اما اگر خانه‌ی فعلی آخرین خانه‌ی اطراف باشد و باز هم از مرز عبور کنیم، به این نتیجه می‌رسیم که در بن‌بست قرار داریم و باید به خانه قبلی بازگردیم (pop از استک انجام دهیم). در صورتی که حرکت مورد نظر موجب عبور از مرز نشود، به مرحله بعدی برای خواندن مقدار خانه از حافظه می‌رویم.

٤. start_reading:

در این مرحله سیگنال read را فعال می‌کنیم تا عملیات خواندن مقدار خانه بعدی از حافظه آغاز شود.

۵. completing_reading:

این مرحله صرفاً برای تأخیر لازم است، زیرا سیگنال read در مرحله قبلی مقداردهی شده و در لبه بعدی کلاک مقدار cell_val از حافظه گزارش خواهد شد.

٦. evaluating_cell_val:

در این مرحله مقدار خوانده شده (cell_val) را بررسی می‌کنیم و تصمیم می‌گیریم وارد چه استیتی شویم. اگر مقدار cell_val صفر باشد (خانه خالی)، می‌توانیم وارد آن خانه شده و آن را به استک اضافه کنیم. ولی اگر مقدار cell_valیک باشد (خانه پر یا دیوار باشد)، نمی‌توانیم به آن خانه برویم. در این صورت اگر هنوز خانه‌های دیگری برای بررسی وجود داشته باشد(co_4~) ، به مرحله بررسی خانه بعدی می‌رویم؛ ولی اگر این آخرین خانه کناری بوده باشد (co_4)، یعنی به بن‌بست رسیده‌ایم و باید به استیت pop کردن از استک برگردیم.

۷. going_to_next_cell:

در این استیت، آماده می‌شویم تا خانه‌ی بعدی را بررسی کنیم. به این منظور مقدار اشاره‌گر Count_4 را یک واحد افزایش می‌دهیم و برای بررسی مجدد وضعیت مرزِ خانه‌ی بعدی، به استیت checking_border بازمی‌گردیم.

۸. pushing_to_main_stack:

در این مرحله خانه‌ی بعدی که باید به آن وارد شویم مشخص شده است و می‌خواهیم آن را به استک مسیر اضافه کنیم. در اینجا سیگنال push_256 را فعال می‌کنیم. همچنین باید خانه‌ی فعلی را به‌عنوان خانه بازدید شده در حافظه مشخص کنیم، پس سیگنال write را نیز فعال می‌کنیم.

۹. completing_pushing:

این مرحله نیز برای تأخیرِ یک کلاک در نظر گرفته شده است، زیرا سیگنال write در مرحله قبل مقداردهی شده و در لبه بعدی کلاک مقدار خانه فعلی به ۱ تغییر می‌کند.

۱۰. comparing_curr_cell_to_goal:

حال بررسی می‌کنیم که آیا مختصات خانه انتخاب‌شده با مختصات مقصد مطابقت دارد یا خیر. اگر برابر باشند، به استیت اعلام رسیدن به مقصد می‌رویم؛ در غیر این صورت، به مرحله افزایش اشاره‌گر انتهای استک (Count_256) می‌رویم و برای بررسی خانه بعدی آماده می‌شویم.

۱۱. Increasing_stack_level:
در این استیت، اشاره‌گر انتهای استک (Count_256) را یک واحد افزایش می‌دهیم تا برای مراحل بعدی بتوانیم به خانه بعدی استک دسترسی داشته باشیم. سپس برای بررسی خانه‌های اطرافِ این خانه‌ی جدید و پیدا کردن عضو بعدی مسیر، مجدداً به استیت مقداردهی اولیه‌ی اشاره‌گر جهت‌ها (Count_4) باز می‌گردیم.

۱۲. poping_from_main_stack:
در این مرحله مشخص شده است که به بن‌بست رسیده‌ایم و باید خانه فعلی را از استک خارج کنیم. برای انجام این کار کافی است که اشاره‌گر استک (Count_256) را یک واحد کاهش دهیم. نکته قابل توجه این است که نیازی به صفر کردن مقدار قبلیِ ذخیره‌شده در این خانه از استک نیست، زیرا در صورت نیاز مجدد، به سادگی می‌توانیم روی آن خانه مقدار جدید را بنویسیم.

۱۳. completing_poping:
این استیت برای اطمینان از اعمال شدن کاهش مقدار اشاره‌گر استک (Count_256) در لبه کلاک بعدی در نظر گرفته شده است. به عبارت دیگر، در استیت قبلی فقط سیگنال کاهنده اشاره‌گر فعال شده و در این مرحله تازه تغییر آن اعمال شده است. پس از این کاهش، مقدار سیگنال empty را بررسی می‌کنیم که در ماژول مسیر داده محاسبه می‌شود. اگر با این کاهش مقدار، استک خالی شده و به خانه اول بازگشته باشیم، متوجه می‌شویم که هیچ مسیری برای رسیدن به مقصد وجود ندارد و باید به استیت failing برویم. در غیر این صورت، اگر استک همچنان خالی نشده باشد، مجدداً بررسی اطراف خانه فعلی را برای یافتن مسیر جدید آغاز می‌کنیم.

۱٤. Failing:
وارد شدن به این مرحله به این معنا است که استک کاملاً خالی شده و اشاره‌گر استک (Count_256) به صفر رسیده است؛ بنابراین در این استیت، سیگنال fail را برای اعلام اینکه مسیرِ ممکن برای رسیدن به مقصد وجود ندارد، فعال می‌کنیم و سپس به استیت Idle بازمی‌گردیم.

۱۵. reaching_the_goal:
در این مرحله می‌دانیم که در خانه بعدی، مختصات هدف قرار دارد و مسیر کامل شده است. بنابراین سیگنال Done را در خروجی فعال می‌کنیم تا رسیدن به مقصد را اعلام کنیم. پس از این مرحله، به استیتی می‌رویم که در آن برای نمایش مسیرِ صحیحِ پیداشده آماده می‌شویم.

۱٦. starting_to_show_the_path:
در این استیت، مسیر صحیح قبلاً پیدا شده و منتظر دریافت سیگنال Run از ورودی هستیم تا نمایش مسیر آغاز شود. تا زمانی که Run فعال نشود در این استیت باقی می‌مانیم. برای نمایش مسیر به‌ترتیبِ خانه‌ها از ساختار صف استفاده کرده‌ایم که نیاز به یک اشاره‌گر (Count_queue) برای خواندن از ابتدای صف دارد. در این مرحله، مقدار این اشاره‌گر را به صفر مقداردهی اولیه می‌کنیم.

۱۷. Loading:
در این مرحله، به‌ترتیب باید مختصات هر خانه از مسیر را در خروجی نمایش دهیم. متغیر خروجیِ ما برای این منظور Move است که برای لود کردن هر خانه از صف در این متغیر، سیگنال کنترلی load را فعال می‌کنیم. سپس در مرحله بعدی، برای خواندن خانه بعدیِ صف آماده می‌شویم.

۱۸. Increasing_queue_level:
در این مرحله سیگنال کنترلی مربوط به افزایش یک واحدی اشاره‌گر صف (Count_queue) را فعال می‌کنیم. سپس به استیت بعدی می‌رویم تا بررسی کنیم که به انتهای صف رسیده‌ایم یا خیر.

۱۹. checking_to_reach_the_end:
در این مرحله بررسی می‌کنیم که آیا اشاره‌گر صف به انتها رسیده است یا خیر. اگر به انتهای صف رسیده باشیم، به

مرحله نهایی اتمام نمایش صف می‌رویم؛ در غیر این صورت به استیت Loading بازمی‌گردیم تا خانه بعدیِ صف را در متغیر Moveبارگذاری کرده و در خروجی نمایش دهیم.

۲۰. finish_showing_the_path:

در این مرحله مشخص شده است که آخرین خانه صف را نیز نمایش داده‌ایم، بنابراین سیگنال اتمام کار (the_end) را فعال می‌کنیم و در کلاک بعدی به استیت اولیه (Idle) برمی‌گردیم.

همچنین توجه داریم که طبق منطق طراحیFSM ، در هر لبه بالارونده کلاک، استیت بعدی (nstate) در استیت کنونی (pstate) ذخیره می‌شود.

ماژول تاپ:

هدف این ماژول برقراری ارتباط بین سیگنال‌های ورودی و خروجی ماژول‌های مسیر داده و کنترلر است. در این ماژول از ماژول‌های مسیر داده و کنترلر نمونه‌برداری (instance) می‌کنیم. تعدادی از خروجی‌های ماژول کنترلر به تعدادی از ورودی‌های ماژول مسیر داده متصل هستند، و بالعکس، برخی خروجی‌های ماژول مسیر داده نیز به ورودی‌های ماژول کنترلر متصل می‌شوند. همچنین تعدادی سیگنال نیز از تاپ ماژول به عنوان ورودی به این دو ماژول داخلی متصل می‌شوند تا کل سیستم بتواند به درستی عمل کند.

ماژول تست بنچ:

هدف از این ماژول بررسی عملکرد صحیح تاپ ماژول طراحی‌شده و اطمینان از درستی سیگنال‌های خروجی آن است. برای این منظور، سیگنال‌های ورودیِ تاپ‌ماژول را به صورت رجیستر و سیگنال‌های خروجی آن را به صورت wire تعریف کرده‌ایم. سپس یک نمونه (instance) از تاپ‌ماژول ایجاد می‌کنیم. سیگنال کلاک (clk) را با دوره تناوب ۶ پیکو ثانیه و دیوتی‌سایکل ۵۰ درصد تعریف کرده‌ایم.

در ادامه، یک بلاک initial تعریف می‌کنیم تا سیگنال‌های ورودی را در آن مقداردهی کنیم. در ابتدا تمامی سیگنال‌های ورودی تاپ‌ماژول را صفر قرار می‌دهیم تا از حالت نامشخص (z) جلوگیری شود. سپس پس از گذشت ۳۰ پیکو ثانیه سیگنال rst را فعال (یک) می‌کنیم تا همه رجیسترهای داخل سیستم مقداردهی اولیه شوند. چند پیکو ثانیه بعد، سیگنال start را برابر با یک قرار می‌دهیم و به مدت ۱۰ پیکو ثانیه منتظر می‌مانیم تا اطمینان حاصل کنیم که این سیگنال حداقل برای یک سیکل کامل کلاک فعال بوده است. سپس سیگنال start را مجدداً صفر می‌کنیم.

اگر سیستم بتواند مسیری را به درستی پیدا کند، سیگنال خروجی Done فعال خواهد شد. برای بررسی این وضعیت، از دستور wait استفاده می‌کنیم تا زمانی که این سیگنال به مقدار یک برسد منتظر بمانیم و اجازه دهیم ماژول‌ها محاسبات خود را انجام دهند. پس از فعال شدن سیگنال Done و گذشت مدت کوتاهی، سیگنال Run را یک می‌کنیم تا سیستم شروع به نمایش مسیر صحیح از طریق متغیر خروجی Move کند.

البته لازم به ذکر است که در صورت پیدا نکردن مسیر درست، سیگنال خروجی fail بعد از گذشت مدتی به مقدار یک تغییر می‌کند، اما در این تست‌بنچ وضعیت پایان شبیه‌سازی برای این حالت پیش‌بینی نشده است و شبیه‌سازی ادامه خواهد داشت.

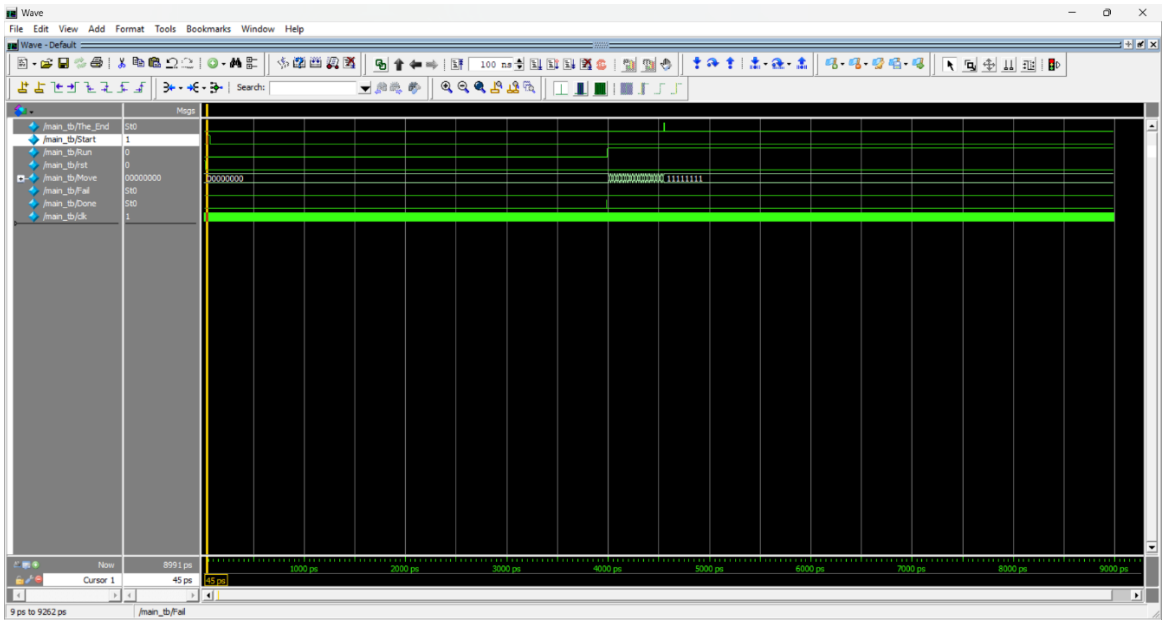
خروجی simulation و waveform و تحلیل آن:

تست ۱:

```
1111111111111100
0111111111111001
0111111000000011
0111110011100111
0111101111001111
0110001110011111
0110111100111111
0110111001111111
1110110011111111
1110100111111111
1110001111111111
0110011111111111
0100111111111111
0001111111111111
0011111111111111
0001111111111111
```

تصویر ۱ _ مپ تست اول

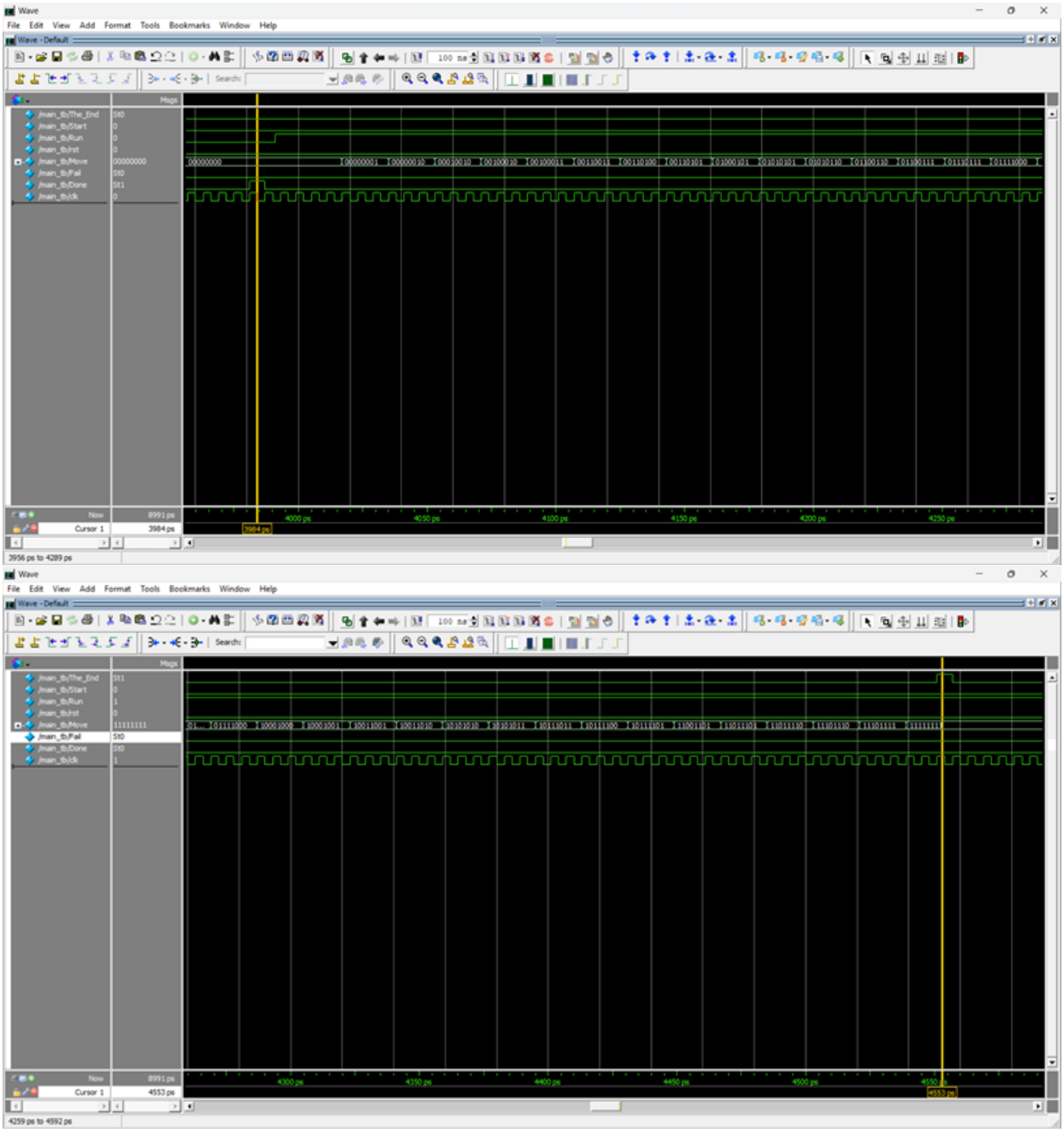
برای بررسی عملکرد سیستم از مپ مقابل به عنوان تست اول استفاده کردیم. همان‌طور که در این مپ مشخص است، مسیر صحیحی از نقطه مبدأ تا مقصد وجود دارد. همچنین برای بررسی بهتر شرایط مختلف مطرح‌شده در صورت سؤال، مسیرهایی را نیز به مپ اضافه کردیم. همان‌گونه که در waveform قابل مشاهده است، در ابتدای حرکت از نقطه شروع به جایی می‌رسیم که هم می‌توانیم مسیر بالایی را انتخاب کنیم و هم مسیر سمت راست را. با توجه به اولویت تعیین‌شده در سؤال (اولویت حرکت ابتدا به سمت بالا)، سیستم ابتدا مسیر بالایی را انتخاب می‌کند، ولی در نهایت این مسیر به بن‌بست ختم می‌شود. بنابراین سیستم اقدام به عقب‌گرد می‌کند و به خانه قبلی بازمی‌گردد. پس از رسیدن دوباره به این دوراهی، مسیر سمت راست را انتخاب کرده و با ادامه این مسیر به موفقیت به نقطه مقصد می‌رسد. این رفتار دقیقاً مطابق انتظار بوده و نشان‌دهنده صحت عملکرد الگوریتم و سیستم طراحی‌شده است.



تصویر ۲ _ شکل موج مقادیر سیگنال های ورودی و خروجی تاپ ماژول

در شکل موج بالا، سیگنال‌های ورودی و خروجی تاپ ماژول به وضوح قابل مشاهده هستند. ابتدا می‌بینیم که سیگنال ورودی Start برای مدتی فعال (یک) شده است. پس از آن، با گذشت ۳۹۸۱ پیکوثانیه از شروع شبیه‌سازی، سیگنال خروجی Done فعال شده و نشان‌دهنده آن است که سیستم توانسته مسیر درست را از نقطه مبدأ تا مقصد پیدا کند. چند پیکوثانیه پس از فعال

شدن Done، سیگنال ورودی Run نیز یک شده و بعد از آن می‌توانیم به ترتیب مختصات هر خانه از مسیر نهایی را در خروجی مشاهده کنیم.



تصاویر ۳ _ شکل موج مقادیر مختصات خانه های مسیر از مبدا به مقصد

در تصویر دوم، این بازه‌ی زمانی به طور دقیق‌تر مشخص شده است. در این بازه ابتدا فعال شدن سیگنال Done و سپس سیگنال Run را مشاهده می‌کنیم. حال، مسیر خروجی که از طریق سیگنال Move نشان داده می‌شود را تحلیل می‌کنیم. همان‌طور که در ماژول مسیر داده طراحی کردیم، هر مقدار خروجی Move شامل مختصات یک خانه از مسیر نهایی است؛ این مختصات به

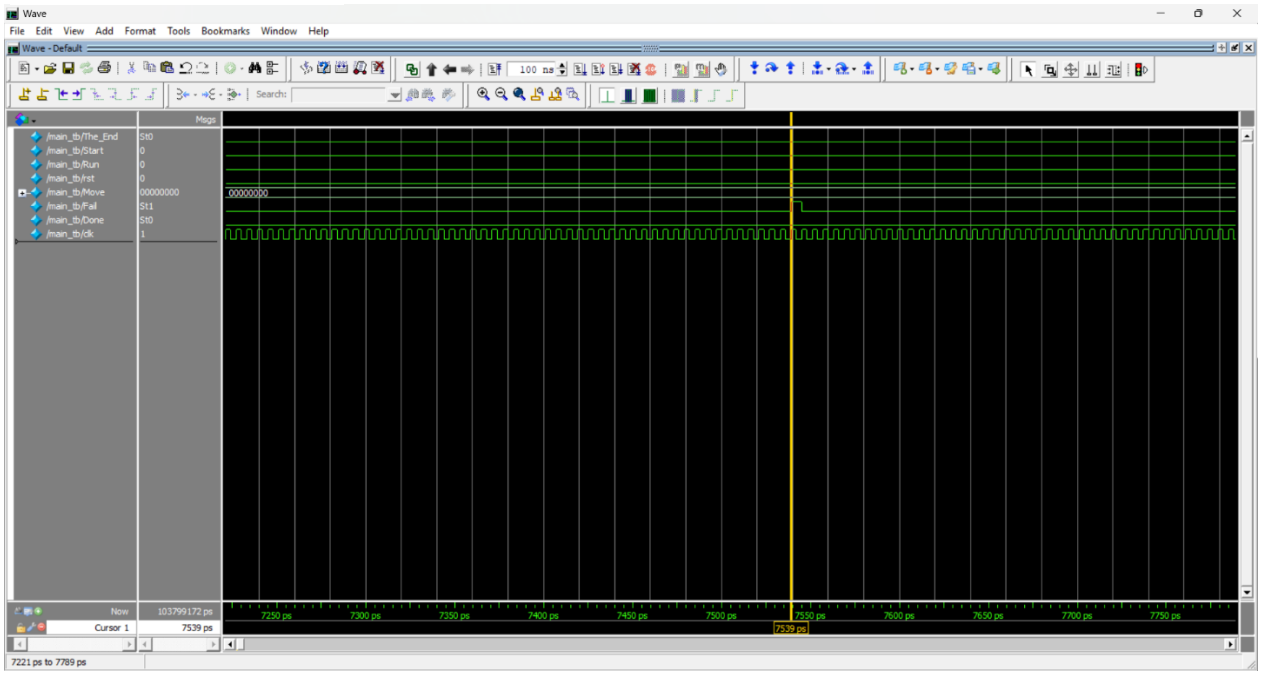
صورت ۸ بیت (۴ بیت برای مختصات x و ۴ بیت برای مختصاتy) در نظر گرفته شده که در آن ۴ بیتِ x در سمت پرارزش (MSB)و ۴ بیتِ y در سمت کم‌ارزش (LSB) قرار گرفته‌اند. در این waveform مشاهده می‌کنیم که طبق اولویت حرکت تعیین‌شده (بالا، راست، چپ، پایین)، ابتدا دو حرکت به سمت بالا انجام شده و در ادامه، سایر خانه‌ها نیز به‌ترتیب و مطابق با خط‌چین قرمز رنگ در مپ طی شده‌اند.

تست ۲:

برای تست دوم، از مپ دیگری استفاده کردیم که در آن امکان یافتن مسیر از مبدأ به مقصد وجود ندارد. در این مپ نیز مسیرهایی طراحی کردیم که بتوانیم شرایط مختلف صورت سؤال را به دقت بررسی کنیم. همان‌طور که مشاهده می‌کنید، در ابتدای مسیر حرکت از مبدأ، به نقطه‌ای می‌رسیم که دو مسیر برای حرکت وجود دارد (مسیر بالا و مسیر سمت راست). طبق اولویت حرکت که در صورت سؤال ذکر شده، سیستم ابتدا مسیر بالا را انتخاب کرده، اما در نهایت به بن‌بست می‌رسد. سپس به عقب بازگشته و در این بازگشت به همان دوراهی، مسیر سمت راست را انتخاب می‌کند؛ ولی این مسیر نیز نهایتاً در نزدیکی مقصد به بن‌بست ختم می‌شود.

```
1111111111111100
0111111111111001
0111111000000011
0111110011110111
0111101111001111
0110001110011111
0110111100111111
0110111001111111
1110110011111111
1110100111111111
1110001111111111
0110011111111111
0100111111111111
0001111111111111
0011111111111111
0001111111111111
```

تصویر۴ _ مپ تست دوک



تصویر5 _ زمان به پایان رسیدن عملیات یافتن مسیر و یک شدن سیگنال Fail

در waveform مربوط به این مپ، می‌بینیم که پس از گذشت ۷۵۳۹ پیکوثانیه سیگنال خروجی fail فعال می‌شود. علت طولانی شدن زمان شبیه‌سازی و تأخیر در فعال شدن سیگنالfail ، وجود دو بن‌بست مجزا در مسیر و بررسی دقیق تمامی مسیرهای ممکن و نیز مسیرهای بازگشتی بوده است.

پایان