گزارش پروژه درس VHDL

میکروکنترلر ربات دانشگاه استفورد SuCRA

استاد درس: دکتر غزنوی

استاد راهنما: خانم دکتر غم خواری

تهیه و تنظیم: احمد خیراندیش

مردادماه 1396

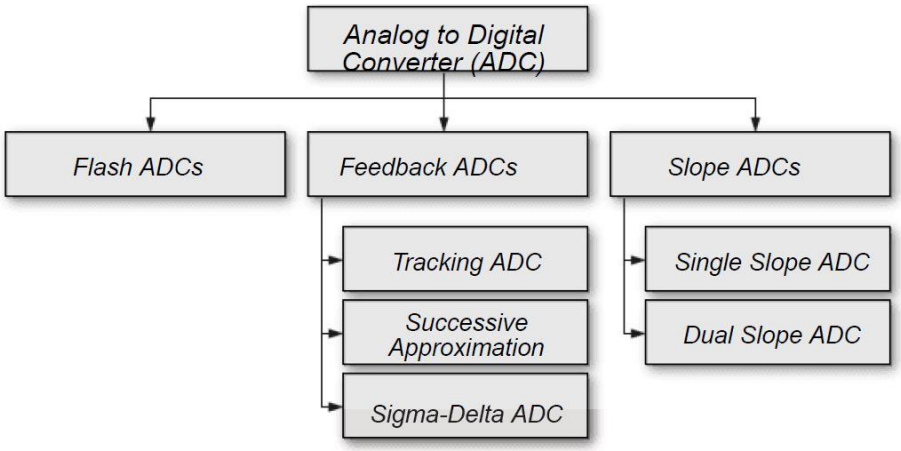
میکروکنترلر SuCRA یک میکروکنترلر 8 بیتی می باشد که به منظور کنترل یک ربات توسعه یافته است .

طراحی میکروکنترلر رباتی دانشگاه استنفورد به نام SuCRA دارای بخش های زیر می باشد :

- Three 4-bit resolution analog-to-digital converters
- One 4-bit general purpose digital I/O port
- Two 8-bit precision DC motor driver ports
- One 8-bit precision servo motor driver port

ADC :

انواع مبدل های آنالوگ به دیجیتال عبارتند از :



همانطور که می دانیم مراحل تبدیل سیگنال آنالوگ به دیجیتال به صورت زیر هستش :

1 - نمونه‌برداری

با استفاده از تبدیل فوریه می‌توان نشان داد که اگر از یک سیگنال آنالوگ با بسامد بیش از ۲ برابر حداکثر بسامد موجود در آن نمونه‌برداری کنیم، می‌توان با استفاده از مقادیر به دست آمده، سیگنال اصلی را دقیقاً بازسازی کرد. به بسامد دو برابر مزبور بسامد نایکویست گفته می‌شود و در سیستم‌های عملی جهت ملاحظات خاصی ۲٫۲ در نظر گرفته می‌شود. حاصل نمونه‌برداری از سیگنال آنالوگ را سیگنال گسسته گویند.

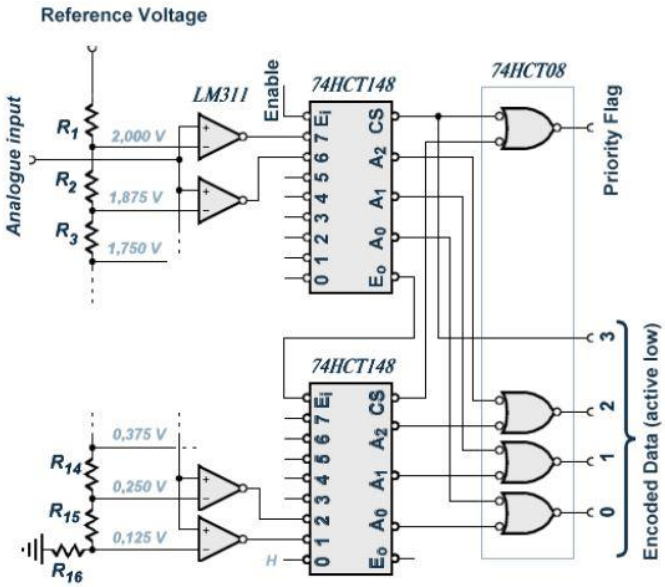
2 -کوانتیزه‌سازی

سیگنال گسسته را جهت دیجیتال‌سازی باید به مقادیر خاصی محدود کرد، به این عملیات، کوانتیزه‌سازی گویند. یک دلیل کوانتیزه سازی آن است که دستگاه‌های کنونی قدرت تشخیص صد در صد یک سیگنال و ذخیره سازی آن را ندارند.

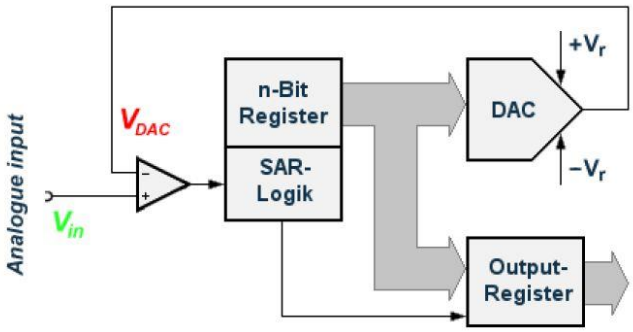
3 - دیجیتال سازی

سیگنال کوانتیده را به صورتهای مختلف می‌توان دیجیتال (یعنی به رشته صفر و یک) تبدیل کرد، که این خود اساس پیدایش دانش کدینگ است. هر سطح سیگنال کوانتیده را به صورت‌های مختلف می‌توان دیجیتال کرد.

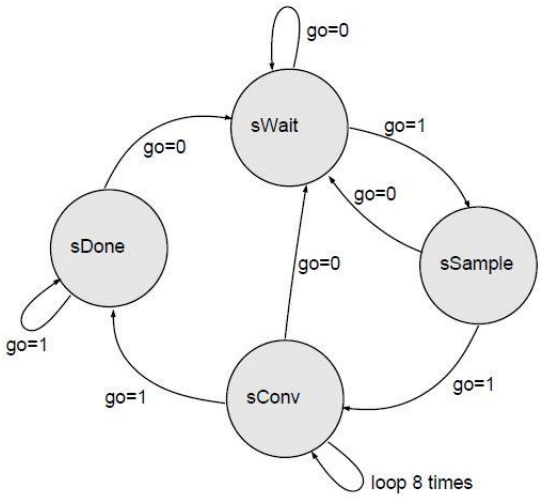
به عنوان مثال یکی از این روش ها به Flash ADC و یا Parallel A/D Converter معروف است که تصاویر یکی از مدارهای آن در زیر آورده شده است :

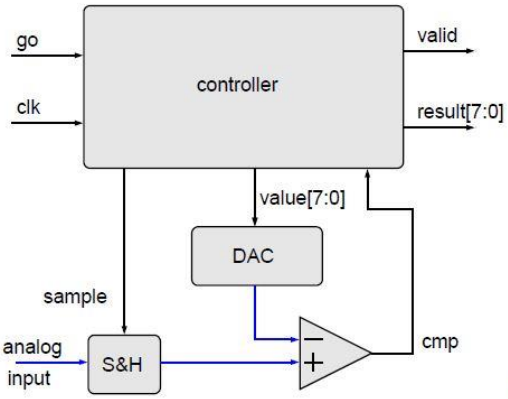


یکی دیگر از انواع مبدل های ADC عبارت است از Successive Approximation (SAR- ADC) که در زیر شمای کلی این نوع مبدل و نمونه ای از پیاده سازی آن و توضیحات مرتبط در حالت 8 بیتی را مشاهده می کنید :



برای بخش مبدل آنالوگ به دیجیتال می توان از مدار Sample & Hold استفاده کرد .مثلا برای مبدل SAR_ADC داریم :





(analog signals are shown in blue)

میکروکنترلر SuCRA علاوه بر مبدل آنالوگ به دیجیتال دارای پورت های ورودی و خروجی نیز می باشد که به سه بخش تقسیم می شوند :

1 – یک پورت 4 بیتی برای استفاده های عمومی

2- دو پورت 8 بیتی برای اتصال به درایور موتور DC روبات

3 – یک پورت 8 بیتی برای اتصال به درایور سرو موتور روبات

هدف از این طراحی میکروکنترلر در نهایت اتصال آن به کیت رباتیک Lego Mindstorms هستش که دارای سنسورهای مختلفی میباشد که می توان به پورت های ورودی و خروجی میکرو متصل نمود البته برخی از این سنسورها و ماژول ها نیاز به منبع تغذیه جداگانه نیز دارند .

پردازنده یک معماری ساده Load و Store بر مبنای معماری RISC با 16 دستور در فرمت دو رجیستری را پیاده سازی می کند . کلیت کار این بخش همانند دستورات RISC معمول می باشد با این تفاوت که سه دستور خاص برای کنترل task های ورودی-خروجی های میکروکنترلر به سیستم اضافه شده است .

Task ها شامل خواندن و نوشتن رجیسترها برای فعال کردن پورت های موتور و سرو ، مبدل های آنالوگ به دیجیتال ، پورت های I/O و رجیسترهای کنترل وضعیت آنهاست .

برای کاهش پیچیدگی طراحی ، از هرگونه ورود داده حافظه جلوگیری شده است و پردازنده به یک 512 ROM بایتی خارجی محدود شده است ، لذا این بدین معنی است که داده ها فقط در 16 رجیستر عمومی ذخیره شوند و عمل Load و Store از/به حافظه دیگر لازم نیست .

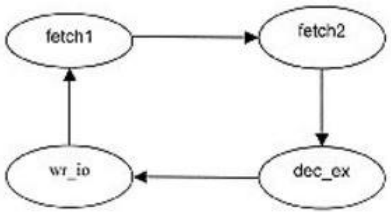
مجموعه کدهای دستوری بکار برده شده عبارتند از :

Instruction Set				
Operation	Opcode [15:12]	Destination [11:8]	Constant [7:0]	
			Source [7:4]	Not used
ADD	0000	RD	RS	n/a
ADDI	0001	RD	Immediate	
AND	0010	RD	RS	n/a
ANDI	0011	RD	Immediate	
NOT	0100	RD	RS	n/a
BRNZ	0101	RD	Target	
XOR	0110	RD	RS	n/a
JMP	0111	n/a	Target	
SUB	1000	RD	RS	n/a
BRZ	1001	RD	Target	
OR	1010	RD	RS	n/a
ORI	1011	RD	Immediate	
IN	1100	RD	IO	n/a
LDI	1101	RD	Immediate	
OUT	1110	IO	RS	n/a
OUTI	1111	IO	Immediate	

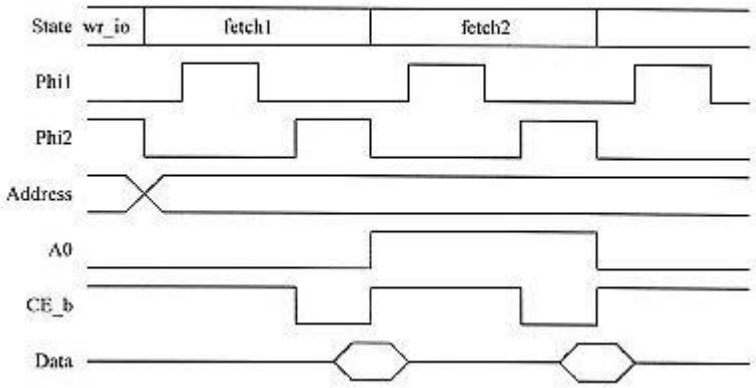
FSM کنترلر شامل 4 وضعیت :

Fetch1 , Fetch2 , Decode/Execute , Write/IO

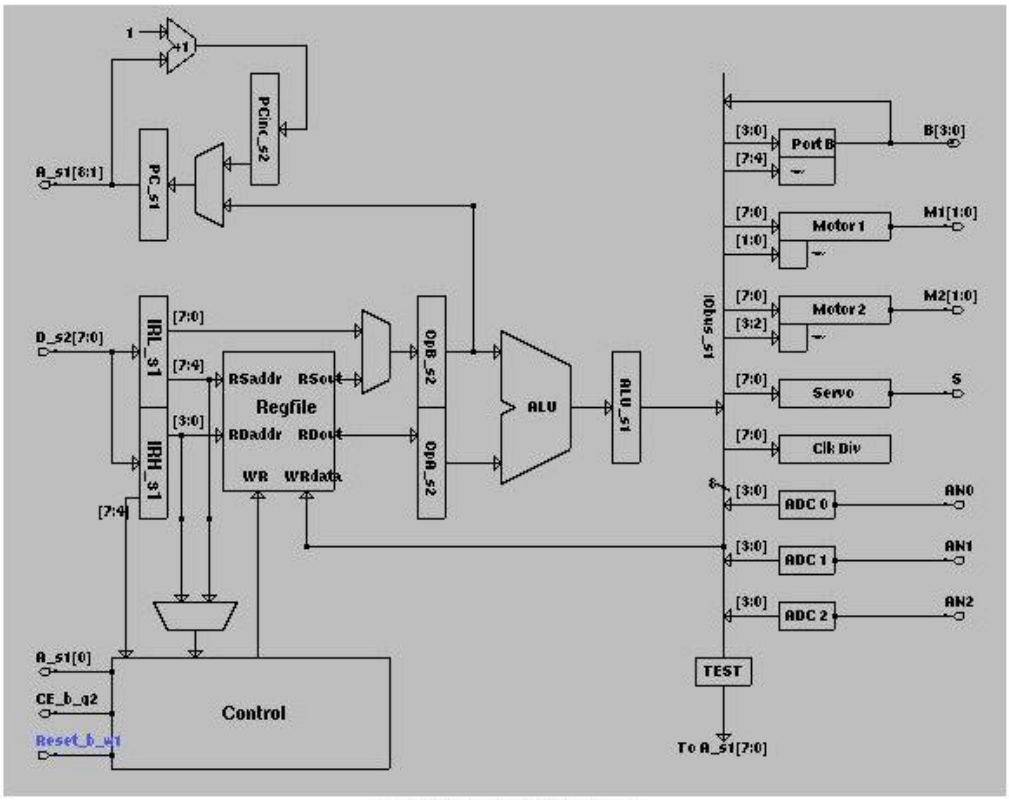
می باشد . وضعیت ارتباطی FSM به صورت زیر است :



وضعیت زمانی حافظه خارجی پردازنده به صورت زیر هستش :

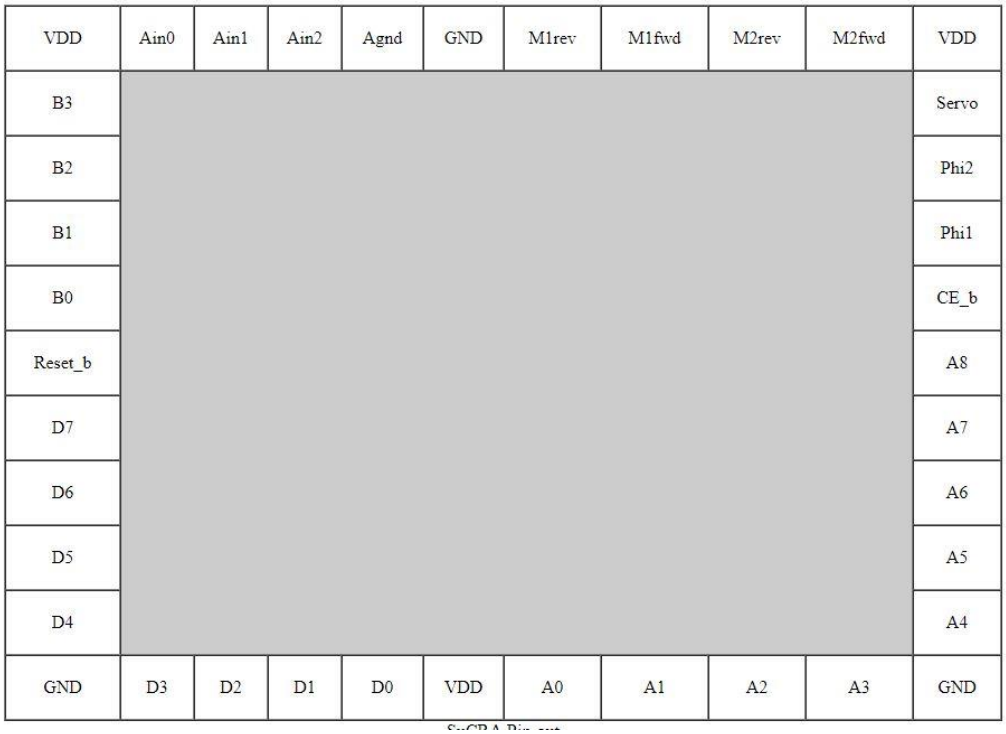


بلوک دیاگرام میکروکنترلر 8 بیتی مان به صورت زیر هستش :



SuCRA Block Diagram

شمای پین های ورودی/خروجی سیستم به صورت زیر می باشد :



پورت های موتور :

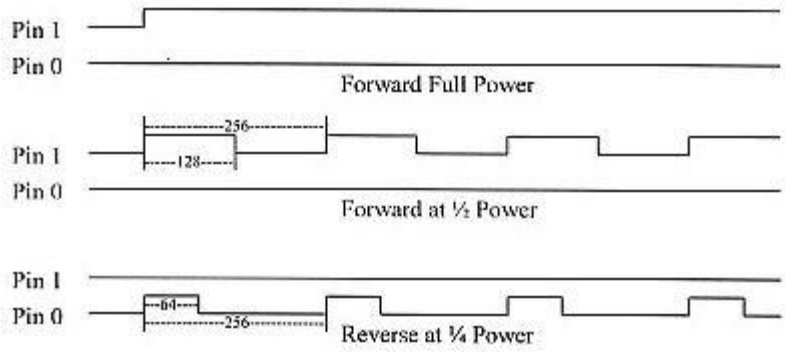
هر پورت موتور شامل یک رجیستر 8 بیتی برای کنترل سرعت موتور و یک رجیستر 2 بیتی برای کنترل جهت چرخش موتور و خاموش-روشن کردن آن می باشد .از این مقادیر جهت تولید سیگنال PWM برای تغذیه موتور DC استفاده می شود .نوعا ، کاربر خروجی دو پورت موتور را به مدار خارجی H-Bridge متصل می کند و سپس خروجی های H-Bridge را به ترمینال های موتور وصل می کند .

پورت موتور از یک شمارنده 8 بیتی برای ایجاد پهنای پالسی سیگنال خروجی استفاده می کند .کاربر سرعت مورد نظر خود را در یک رجیستر 8 بیتی سرعت ذخیره می کند و پورت موتور این دو مقدار را باهم مقایسه می کند و اگر شمارنده کمتر از سرعت باشد ، یکی از پین های خروجی یک شده و دیگری صفر می گردد . بنابراین یکی از پین ها یک پالسی که دوره سیکل آن برابر با مقدار سرعت می باشد به خروجی می فرستد .

پورت موتور از بیت لچ جهت برای مشخص کردن اینکه کدام پین پالس را دریافت می کند و کدام یک در حالت Low باقی می ماند . در واقع این پروسه جهت چرخش موتور را مشخص می کند . اگر به پین 1 پالس داده شود در حالیکه پین 0 درحالت Low باقی بماند ، موتور به سمت مستقیم حرکت خواهد کرد و اگر به پین 0 پالس داده شود درحالی که پین 1 درحالت Low قرار دارد ، موتور به صورت معکوس حرکت خواهد کرد .

بیت لچ On/Off برای روشن و خاموش کردن موتور مورد استفاده قرار میگیرد . اگر این بیت مقدار High داشته باشد ، پین های خروجی طبق توضیحات بالا عمل می کنند .اگر مقدار Low داشته باشد ، هر دو پین ، بدون توجه به مقادیر شمارنده ، رجیستر وضعیت یا بیت جهت، در حالت Low قرار می گیرند.

فرکانس سیگنال خروجی باید تحت فرمان (وضعیت) 1 تا 20 کیلوهرتز باشد . برای نیل به این فرکانس خروجی ، پورت موتور ممکنه نیاز به استفاده ازکلاک آهسته تر نسبت به ریست چیپ داشته باشد . بنابراین ، یکی از خروجی های ماژول مقسم کلاک توسط مالتی پلکسر انتخاب شده و به عنوان ورودی به پورت تغذیه می شود .



پورت سرو شامل رجیسترهایی برای نگهداشتن یک مقدار 8بیتی وضعیت و یک 8 بیتی سرو با حداقل مقدار پالس شمارشی است ، از این مقادیر برای تولید یک سیگنال با پهنای پالسی مدوله شده (PWM) که می تواند به سرو مدل RC استاندارد به صورت مستقیم متصل شود ، استفاده می شود .

سروها نیاز به تایمینگ خیلی خاصی دارند ، نرخی که پالس ها به سرو ارسال می شوند به طور یکسان اهمیتی ندارند (40-400 به ازای هر ثانیه قابل قبول هست) ، اما پهنای پالس اهمیت دارد . پهنای پالس وضعیت 1.5 میلی ثانیه ای سرو در مرکزیت آن ، با 1 و 2 میلی ثانیه محدوده مورد قبول دو طرف است . برای پیاده سازی این تایمینگ ، پورت سرو از یک شمارنده 10 بیتی استفاده می کند که نیاز دارد تا در نرخی نزدیک 256 کیلوهرتز کلاک بخورد .(به دلایلی که در اینجا به آن اشاره نمی شود ، بهتر است که شمارنده در نرخی کمتر از 256 کیلوهرتز کلاک بخورد نه بیشتر از آن) . همچنین درباره پورت موتور هم به همین منوال ، یکی از خروجی های ماژول مقسم کلاک توسط مالتی پلکسر انتخاب شده و بعنوان ورودی به پورت سرو تغذیه می گردد .

پورت سرو خروجی های خود را ، با مقایسه مقدار شمارنده 10 بیتی با مجموع مقدار 8 بیت وضعیت و مقدار شمارش پالسی حداقلی ، تولید می کند . اگر شمارنده کمتر از مجموع باشد ، پین خروجی درحالت High قرار می گیرد ، اگر شمارنده از مجموع بیشتر شود ، پین در وضعیت Low قرار می گیرد . بنابراین ، پین برای دوره ای که برابر
(minimum pulse count + Position) * (1/fcounter) می باشد در وضعیت High می باشد ، که fcounter فرکانسی می باشد که شمارنده در آن کلاک می خورد . درحالیکه fcounter می تواند متغیر باشد ، و اینکه هرگز نقطه مرکزی هر دو سرو با هم یکی نیست ، ماژول پورت سرو ، نیاز به این دارد تا کاربر شمارش پالسی

حداقلی را در شروع برنامه اش تعیین کند .عموما ، کاربر یک مقدار ابتدایی را انتخاب می کند و لذا fcounter * minimum pulse count برابر 1 میلی ثانیه می شود و سپس این مقدار را تنظیم می کند و لذا سرو در مرکز آن قرار میگیرد زمانی که وضعیت آن بر روی 128 قرار دارد .

ازآنجاییکه سرو ها نیاز به پالس هایی دارند که بین 40 و 400 به ازای هر ثانیه ارسال شوند ، پورت سرو، شمارنده 10 بیتی به ازای هر (1/fcounter)*(minimum pulse count +512) ثانیه ، را ریست می کند .در واقع این پورت مسئولیت این را برعهده دارد که این اطمینان را پیدا کند که کاربر یک قرکانس شمارنده و شمارش پالسی حداقلی مثل (minimum pulse count +512) را انتخاب می کند بین محدوده 40 تا 400 هرتز قرار دارد .

ماژول مقسم کلاک :

این ماژول Phi1 و Phi2 را از ورودی دریافت کرده و پنج کلاک دو فاز را تولید می کند و کلاک ها در فرکانس هایی با نسبت ½ , ¼ , 1/8 , 1/16 و 1/32 فرکانس Phi1 و Phi2 کار می کنند .10 تا لچ در پنج گام قرار گرفته اند که هر کدام سیگنال a_q1 و a_q2 را در یکی از فرکانس های بالایی تولید می کنند .

اولویت به انکدر باینری :

مبدل A/D کدینگ اولویت ولتاژ را به پین آنالوگ می فرستد . این بلوک شامل منطق برای تبدیل این مقدار به یک مقدار باینری 4 بیتی می باشد .

کدهای سورس و تست Verilog بهمراه بلوک دیاگرام بخش های مختلف میکروکنترلر :

شبیه سازی ها در نرم افزار **Xilinix Vivado** انجام شده و بلوک دیاگرام ها در نرم افزار **Visio** کشیده شده است .

ALU

Source Code	Test Bench
`define OP_ADD 3'b000 `define OP_SUB 3'b100 `define OP_NOT 3'b010 `define OP_AND 3'b001 `define OP_OR 3'b101 `define OP_XOR3'b011 module alu(OpA_s2, OpB_s2, ALU_op_s2, ALU_out_s2, Zero_s2); input [7:0] OpA_s2; input [7:0] OpB_s2; input [2:0] ALU_op_s2; output [7:0] ALU_out_s2; output Zero_s2; wire [7:0] OpA_s2, OpB_s2; reg [7:0] ALU_out_s2; wire Zero_s2; wire [2:0] ALU_op_s2; always @(OpA_s2 or OpB_s2 or ALU_op_s2) case (ALU_op_s2) `OP_ADD: ALU_out_s2 = OpA_s2 + OpB_s2; `OP_AND: ALU_out_s2 = OpA_s2 & OpB_s2; `OP_OR: ALU_out_s2 = OpA_s2 | OpB_s2; `OP_SUB: ALU_out_s2 = OpA_s2 - OpB_s2; `OP_XOR: ALU_out_s2 = OpA_s2 ^ OpB_s2; `OP_NOT: ALU_out_s2 = ~OpB_s2; default: ALU_out_s2 = OpB_s2; endcase // case(ALU_op_s2) assign Zero_s2 = (OpA_s2 == 0) ? 1 : 0; endmodule // alu	module alu_test; reg [7:0] OpA_s2, OpB_s2; wire [7:0] ALU_out_s2; reg [2:0] ALU_op_s2; wire Zero_s2; alu u0(.OpA_s2(OpA_s2) , .OpB_s2(OpB_s2) , .ALU_out_s2(ALU_out_s2) , .ALU_op_s2(ALU_op_s2) , .Zero_s2(Zero_s2)); initial begin 0# OpA_s2 = 8'b10101010; OpB_s2 = 8'b01010101; ALU_op_s2 = 3'b000; 10# ALU_op_s2 = 3'b001; 10# ALU_op_s2 = 3'b010; 10# ALU_op_s2 = 3'b011; 10# ALU_op_s2 = 3'b100; 10# ALU_op_s2 = 3'b101; 10# ALU_op_s2 = 3'b110; 10# ALU_op_s2 = 3'b111; 10# OpA_s2 = 0; 10# OpB_s2 = 0; $ stop; end always @(ALU_out_s2) $ display("%h %h %h %h %d\n", ALU_op_s2, OpA_s2, OpB_s2, ALU_out_s, Zero_s2); alu alu(OpA_s2, OpB_s2, ALU_op_s2, ALU_out_s2, Zero_s2); endmodule

Clock Divider

Source Code	Test Bench
module clk_divider (Phi1, Phi2, Reset_b_s1, Reset_b_s2, IObus_s1, Clock_enable_q1, MotorClk_q1, MotorClk_q2, ServoClk_q1, ServoClk_q2, ADClk_q1, ADClk_q2); // CONSTANT DECLARATIONS `define AD 7:6 `define SVO 5:3 `define MOT 2:0 `define MOT_BY1 3'b000 `define MOT_BY2 3'b001 `define MOT_BY4 3'b010	module Clk_div_test; reg [7:0] IObus_s1; reg Clock_enable_q1; reg Phi1, Phi2; reg Reset_b_s1, Reset_b_s2; wire MotorClk_q1, MotorClk_q2; wire ServoClk_q1, ServoClk_q2; wire ADClk_q1, ADClk_q2; clk_divider u0 (.IObus_s1(IObus_s1) , .Clock_enable_q1(Clock_enable_q1) , .Phi1(Phi1) , .Phi2(Phi2) , .Reset_b_s1(Reset_b_s1) , .Reset_b_s2(Reset_b_s2) , .MotorClk_q1(MotorClk_q1), .MotorClk_q2(MotorClk_q2),

```
`define MOT_BY8  3'b011
`define MOT_BY16 3'b100
`define MOT_BY32 3'b101
`define AD_BY4   2'b00
`define AD_BY8   2'b01
`define AD_BY16  2'b10
`define AD_BY32  2'b11
// PORT DECLARATIONS
  input  [7:0] IObus_s1;
  input  Clock_enable_q1;
  input       Phi1, Phi2;
  input  Reset_b_s1, Reset_b_s2;
  output          MotorClk_q1, MotorClk_q2;
  output          ServoClk_q1, ServoClk_q2;
  output          ADClk_q1, ADClk_q2;
// VARIABLE DECLARATIONS
  reg     [7:0] Clock_select_s1, Clock_select_s2;
  wire [7:0]  Clock_select_in_s1, Clock_select_in_s2;
  reg       DivBy2_s1,  DivBy2_s2;
  reg       DivBy4_s1,  DivBy4_s2;
  reg       DivBy8_s1,  DivBy8_s2;
  reg       DivBy16_s1, DivBy16_s2;
  reg       DivBy32_s1, DivBy32_s2;
  wire      DivBy2_q1,  DivBy2_q2;
  wire      DivBy4_q1,  DivBy4_q2;
  wire      DivBy8_q1,  DivBy8_q2;
  wire      DivBy16_q1, DivBy16_q2;
  wire      DivBy32_q1, DivBy32_q2;
  wire      DivBy2_in_s1,  DivBy2_in_s2;
  wire      DivBy4_in_s1,  DivBy4_in_s2;
  wire      DivBy8_in_s1,  DivBy8_in_s2;
  wire      DivBy16_in_s1, DivBy16_in_s2;
  wire      DivBy32_in_s1, DivBy32_in_s2;
// CLOCK DIVISION
  // DIVIDE BY 2
  always @(Phi1 or DivBy2_in_s1)
    if (Phi1)
      DivBy2_s2 = DivBy2_in_s1;
  always @(Phi2 or DivBy2_in_s2)
    if (Phi2)
      DivBy2_s1 = DivBy2_in_s2;
  assign DivBy2_in_s1 = DivBy2_s1;
  assign DivBy2_in_s2 = (Reset_b_s2) ? ~DivBy2_s2 : 0;
  // DIVIDE BY 4
  always @(DivBy2_q1 or DivBy4_in_s1)
    if (DivBy2_q1)
      DivBy4_s2 = DivBy4_in_s1;
  always @(DivBy2_q2 or DivBy4_in_s2)
    if (DivBy2_q2)
      DivBy4_s1 = DivBy4_in_s2;
  assign DivBy4_in_s1 = DivBy4_s1;
  assign DivBy4_in_s2 = Reset_b_s2 ?  ~DivBy4_s2 : 0;
  // DIVIDE BY 8
  always @(DivBy4_q1 or DivBy8_in_s1)
    if (DivBy4_q1)
      DivBy8_s2 = DivBy8_in_s1;
```

```
  .ServoCLK_q1.ServoCLK_q1) , .ServoCLK_q2(ServoCLK_q2) ,
  .ADClk_q1(ADClk_q1) , .ADClk_q1(ADClk_q1) );
initial begin
 #0 [7:0] IObus_s1=8'b00001111;
      Clock_enable_q1=1'b0;
       Phi1=1'b0;
        Phi2=1'b1;
       Reset_b_s1=1'b1;
       Reset_b_s2=1'b0;
    #50  [7:0] IObus_s1=8'b00110000;
           Clock_enable_q1=1'b1;
            Phi1=1'b1;
             Phi2=1'b0;
            Reset_b_s1=1'b0;
            Reset_b_s2=1'b1;
       #20 [7:0] IObus_s1=8'b00001111;
              Clock_enable_q1=1'b1;
               Phi1=1'b0;
                Phi2=1'b1;
               Reset_b_s1=1'b0;
               Reset_b_s2=1'b0;
end
endmodule
```

ادامه کد سورس :

```
assign DivBy2_q1  = (DivBy2_s1  || !Reset_b_s1) && Phi1;
  assign DivBy2_q2  = (DivBy2_s2  || !Reset_b_s2) && Phi2;
  assign DivBy4_q1  = (DivBy4_s1  || !Reset_b_s1) &&
DivBy2_q1;
  assign DivBy4_q2  = (DivBy4_s2  || !Reset_b_s2) &&
DivBy2_q2;
  assign DivBy8_q1  = (DivBy8_s1  || !Reset_b_s1) &&
DivBy4_q1;
  assign DivBy8_q2  = (DivBy8_s2  || !Reset_b_s2) &&
DivBy4_q2;
  assign DivBy16_q1 = (DivBy16_s1 || !Reset_b_s1) &&
DivBy8_q1;
  assign DivBy16_q2 = (DivBy16_s2 || !Reset_b_s2) &&
DivBy8_q2;
  assign DivBy32_q1 = (DivBy32_s1 || !Reset_b_s1) &&
DivBy16_q1;
  assign DivBy32_q2 = (DivBy32_s2 || !Reset_b_s2) &&
DivBy16_q2;
// CLOCK SELECTION
  // Load the Clock Select register
  always @(Clock_select_in_s1 or Clock_enable_q1)
    if (Clock_enable_q1)
      Clock_select_s2 = Clock_select_in_s1;
  always @(Phi2 or Clock_select_in_s2)
    if (Phi2)
      Clock_select_s1 = Clock_select_in_s2;
```

```
  always @(DivBy4_q2 or DivBy8_in_s2)
    if (DivBy4_q2)
          DivBy8_s1 = DivBy8_in_s2;
  assign DivBy8_in_s1 = DivBy8_s1;
  assign DivBy8_in_s2 = Reset_b_s2 ?  ~DivBy8_s2 : 0;
  // DIVIDE BY 16
  always @(DivBy8_q1 or DivBy16_in_s1)
   if (DivBy8_q1)
    DivBy16_s2 = DivBy16_in_s1;
  always @(DivBy8_q2 or DivBy16_in_s2)
   if (DivBy8_q2)
    DivBy16_s1 = DivBy16_in_s2;
  assign DivBy16_in_s1 = DivBy16_s1;
  assign DivBy16_in_s2 = Reset_b_s2 ?  ~DivBy16_s2 : 0;
  // DIVIDE BY 32
  always @(DivBy16_q1 or DivBy32_in_s1)
    if (DivBy16_q1)
          DivBy32_s2 = DivBy32_in_s1;
  always @(DivBy16_q2 or DivBy32_in_s2)
    if (DivBy16_q2)
      DivBy32_s1 = DivBy32_in_s2;
  assign DivBy32_in_s1 = DivBy32_s1;
  assign DivBy32_in_s2 = Reset_b_s2 ?  ~DivBy32_s2 : 0;


  // Gate the s1 signals to create q1 signals

                (Clock_select_s1[`SVO] == `MOT_BY16) ?
DivBy16_q1:
                (Clock_select_s1[`SVO] == `MOT_BY8)  ?
DivBy8_q1:
                (Clock_select_s1[`SVO] == `MOT_BY4)  ?
DivBy4_q1:
                (Clock_select_s1[`SVO] == `MOT_BY2)  ?
DivBy2_q1:
                Phi1;
  assign ServoClk_q2 = (Clock_select_s2[`SVO] ==
`MOT_BY32) ? DivBy32_q2:
                (Clock_select_s2[`SVO] == `MOT_BY16) ?
DivBy16_q2:
                (Clock_select_s2[`SVO] == `MOT_BY8)  ?
DivBy8_q2:
                (Clock_select_s2[`SVO] == `MOT_BY4)  ?
DivBy4_q2:
                (Clock_select_s2[`SVO] == `MOT_BY2)  ?
DivBy2_q2:
                Phi2;
  // Select the clock for the AD converters
  assign ADClk_q1 = (Clock_select_s1[`AD] == `AD_BY32) ?
DivBy32_q1:
              (Clock_select_s1[`AD] == `AD_BY16) ?
DivBy16_q1:
              (Clock_select_s1[`AD] == `AD_BY8)  ? DivBy8_q1:
              DivBy4_q1;
  assign ADClk_q2 = (Clock_select_s2[`AD] == `AD_BY32) ?
DivBy32_q2:
  assign Clock_select_in_s1 = Reset_b_s1 ? IObus_s1 : 8'b0;
  assign Clock_select_in_s2 = Clock_select_s2;

  // Select the clock for the motors
  assign MotorClk_q1 = (Clock_select_s1[`MOT] ==
`MOT_BY32) ? DivBy32_q1:
                (Clock_select_s1[`MOT] == `MOT_BY16) ?
DivBy16_q1:
                (Clock_select_s1[`MOT] == `MOT_BY8)  ?
DivBy8_q1:
                (Clock_select_s1[`MOT] == `MOT_BY4)  ?
DivBy4_q1:
                (Clock_select_s1[`MOT] == `MOT_BY2)  ?
DivBy2_q1:
                Phi1;
  assign MotorClk_q2 = (Clock_select_s2[`MOT] ==
`MOT_BY32) ? DivBy32_q2:
                (Clock_select_s2[`MOT] == `MOT_BY16) ?
DivBy16_q2:
                (Clock_select_s2[`MOT] == `MOT_BY8)  ?
DivBy8_q2:
                (Clock_select_s2[`MOT] == `MOT_BY4)  ?
DivBy4_q2:
                (Clock_select_s2[`MOT] == `MOT_BY2)  ?
DivBy2_q2:
                Phi2;
  // Select the clock for the servos
  assign ServoClk_q1 = (Clock_select_s1[`SVO] ==
`MOT_BY32) ? DivBy32_q1:
```

```
        (Clock_select_s2[`AD] == `AD_BY16) ?
DivBy16_q2:
        (Clock_select_s2[`AD] == `AD_BY8)  ? DivBy8_q2:
        DivBy4_q2;
endmodule // clk_divider
```

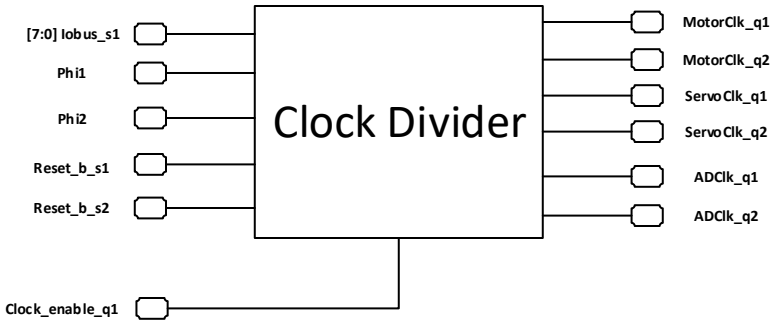
Clock Generator

Source Code (With Initialization)

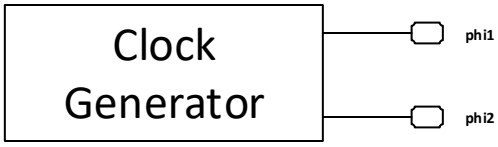
```
module clkgen(phi1, phi2);
  output      phi1,        // Two-phase non-overlapping clocks
              phi2;
  reg         phi1,
              phi2;
initial         // Start with both clocks low
    begin
      phi1 = 0;
      phi2 = 0;
    end
always   // Generate two-phase non-overlapping clock waveforms
    begin
      #100 phi1 = 0;
      #25  phi2 = 1;
      #100 phi2 = 0;
      #25  phi1 = 1;
    end
endmodule
```

ALU Module Diagram

Clock Divider Module Diagram

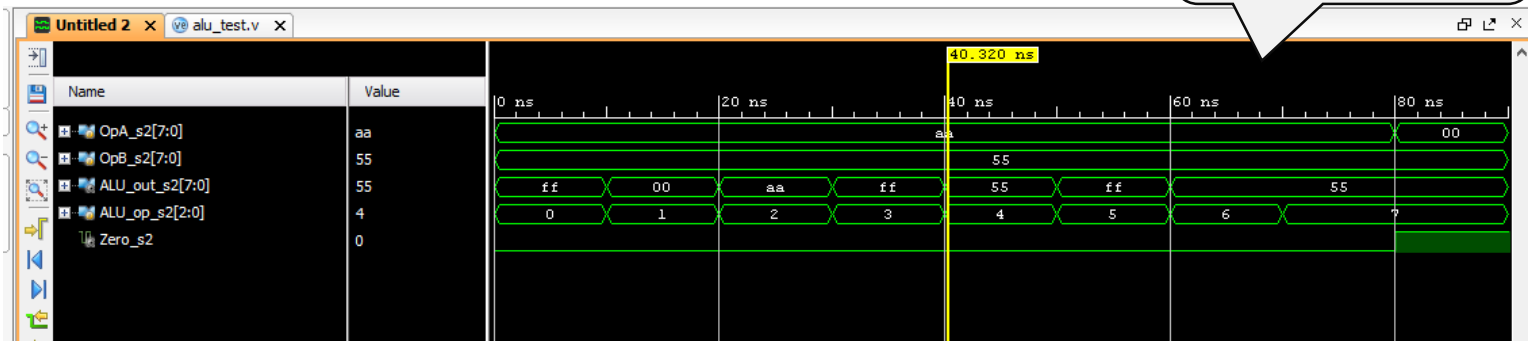


Clock Generator Module Diagram

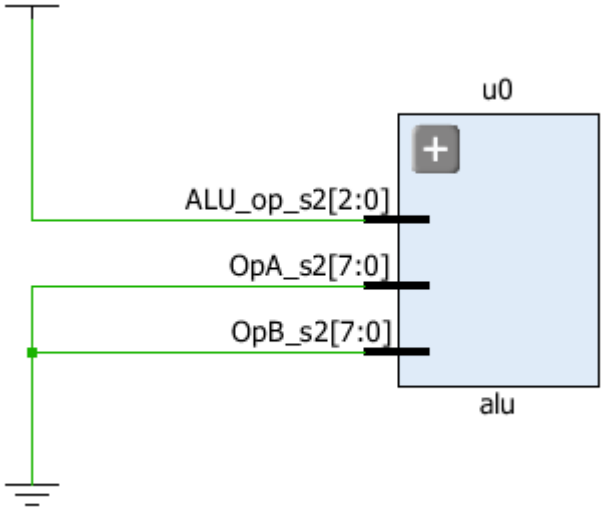


Simulation Waveform

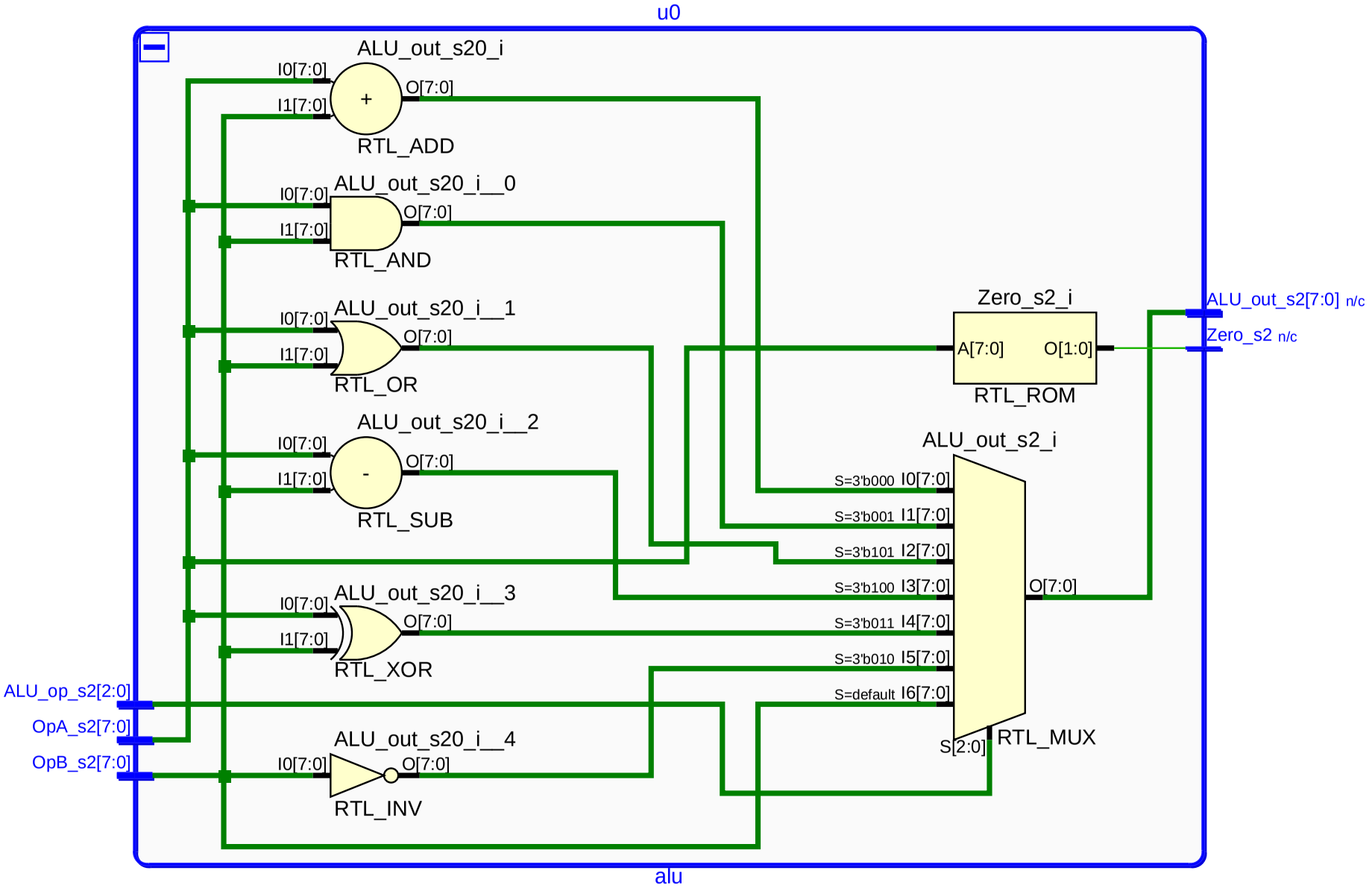
ALU



RTL Analyze Schematic

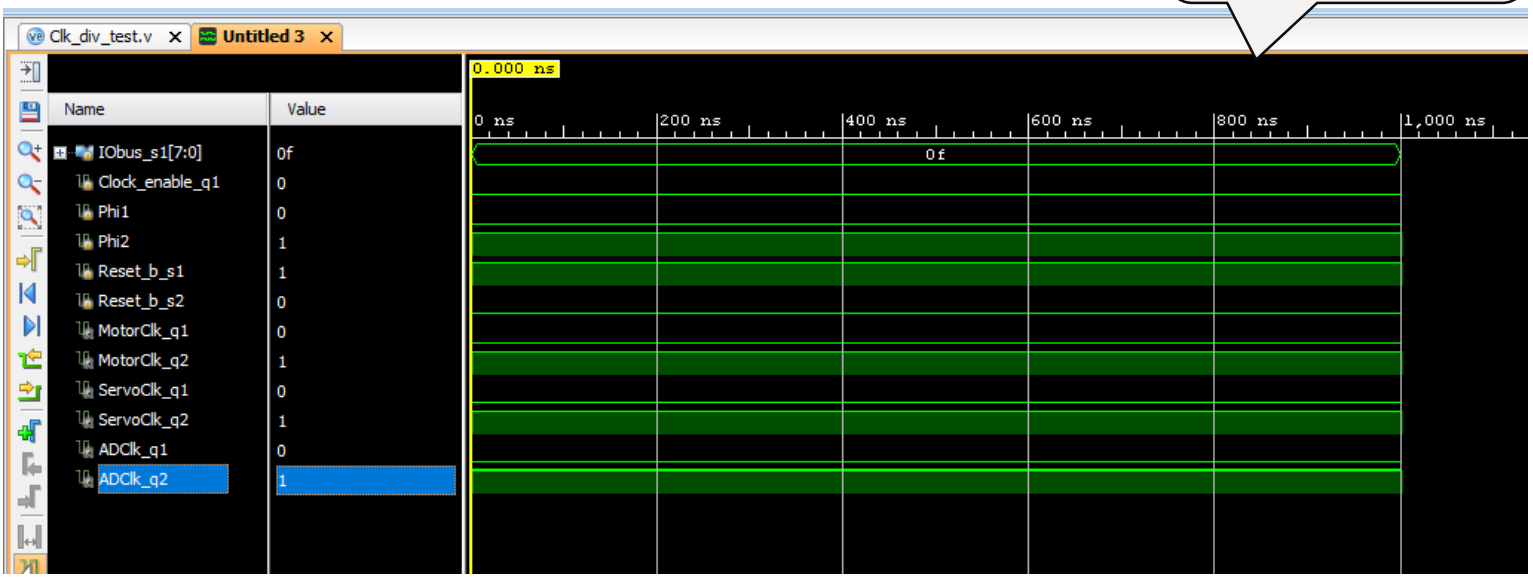


For view details, you can zoom in

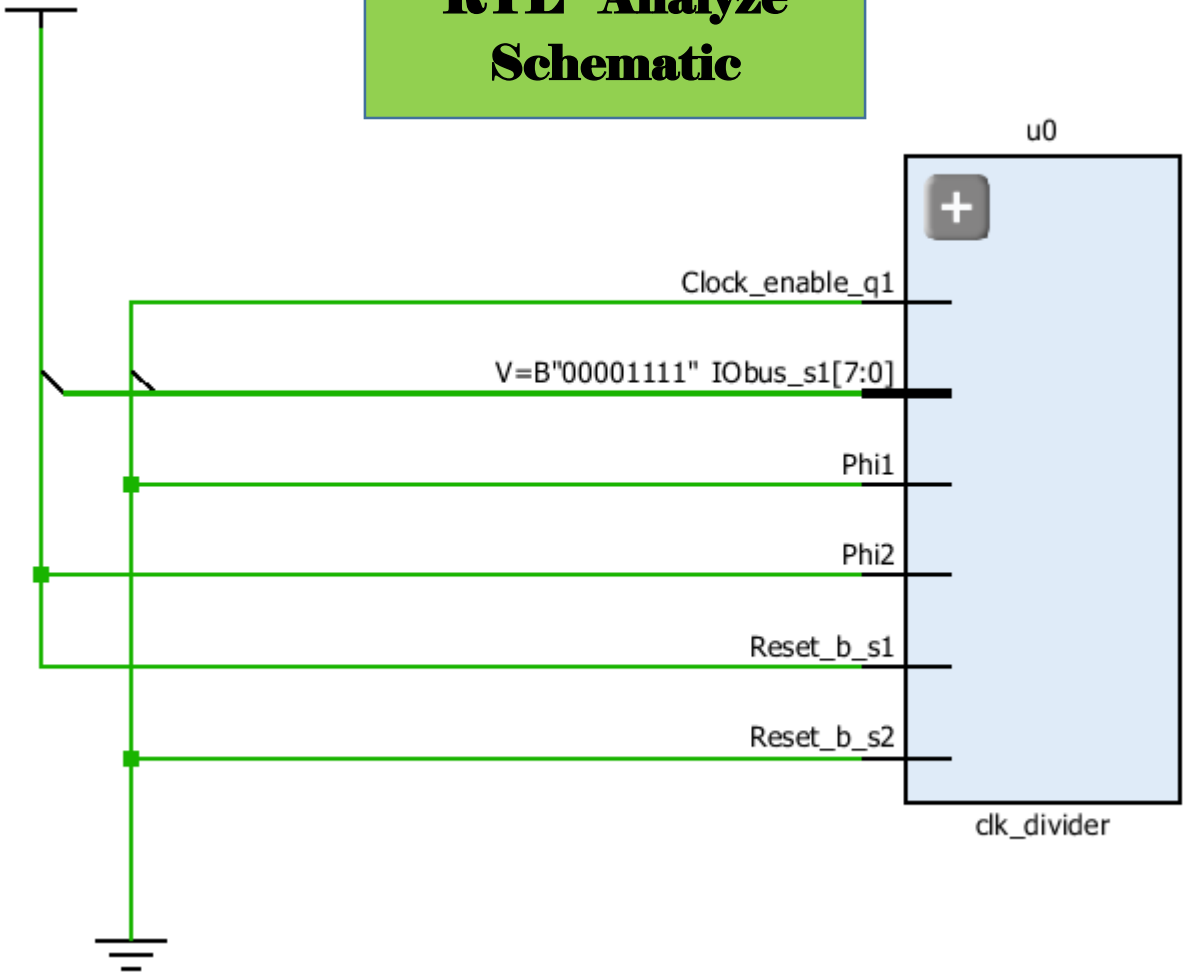


Simulation Waveform

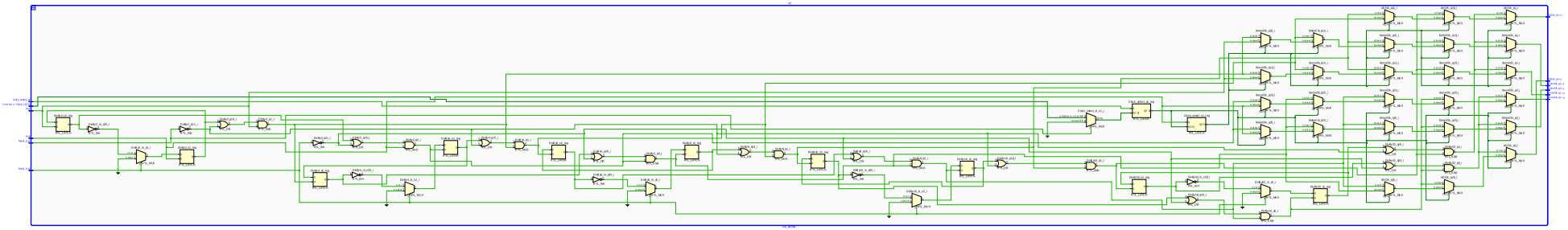
Clock Divider



RTL Analyze Schematic

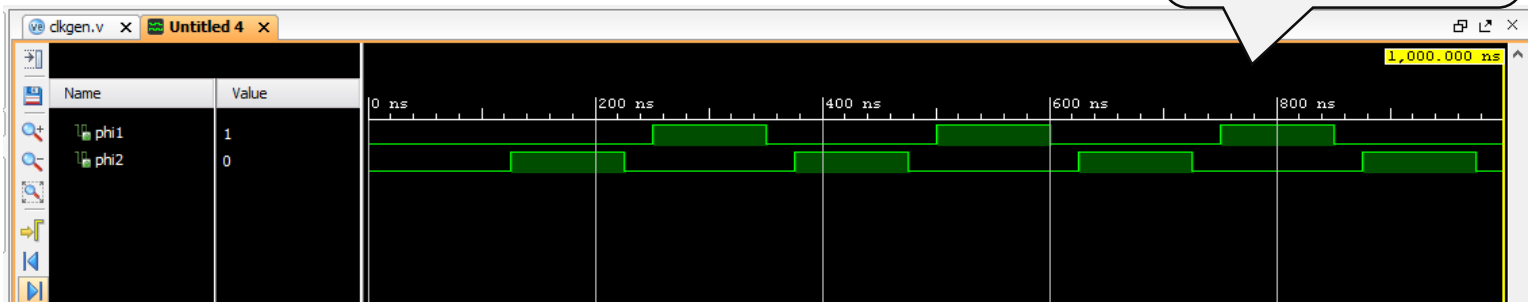


For view details, you can zoom in

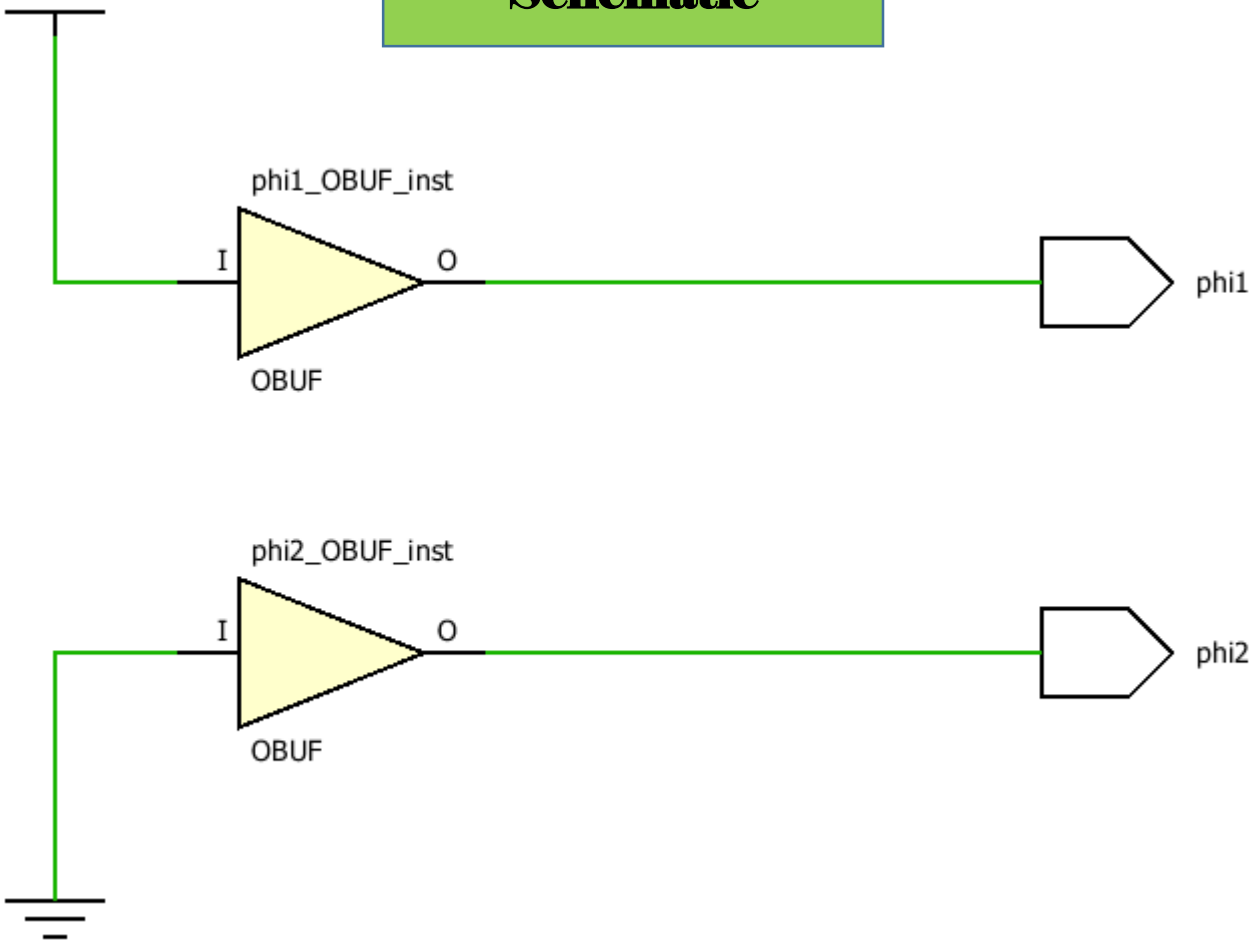


Simulation Waveform

Clock Generator



RTL Analyze Schematic



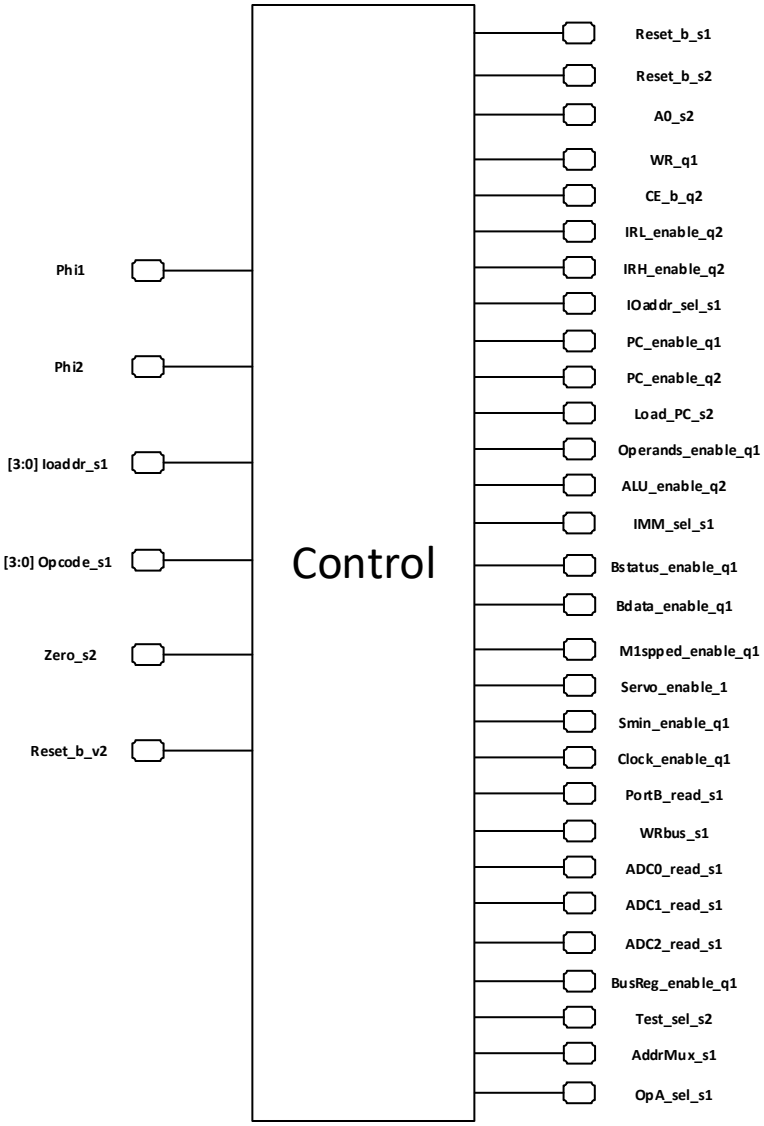
Control

Source Code	Test Bench

بدلیل حجم زیاد خطوط کد، در این قسمت ذکر نشده است .	module Control_test;
	reg Phi1, Phi2; // 2-Phase Clock
	reg [3:0] IOaddr_s1; // I/O register number
	reg [3:0] Opcode_s1; // Opcode from IR
	reg Zero_s2; // RD == 0 flag
	reg Reset_b_v2;
	wire Reset_b_s1; // Reset
	wire Reset_b_s2;
	wire A0_s2; // LSB of memory address
	wire WR_q1; // Write enable for regfile
	wire CE_b_q2; // Flash memory Chip enable (Read)
	wire IRL_enable_q2; // Latch control for low bits of IR
	wire IRH_enable_q2;// Latch control for high bits of IR
	wire IOaddr_sel_s1;
	wire PC_enable_q1, // Latch controls for PC
	PC_enable_q2;
	wire Load_PC_s2; // PC <- Branch or Jump target
	wire Operands_enable_q1; // Latches regfile outputs
	wire ALU_enable_q2; // Latches ALU result
	wire IMM_sel_s1; // Selects input to ALU operand B
	wire Bstatus_enable_q1; // Latch control for Port B status
	wire Bdata_enable_q1; // Latch control for Port B data
	wire M1speed_enable_q1; // Latch control for Motor1 speed
	wire M1status_enable_q1; // Latch control for Motor1 status
	wire M2speed_enable_q1; // Latch control for Motor2 speed
	wire M2status_enable_q1; // Latch control for Motor2 status
	wire Servo_enable_q1; // Latch control for Servo speed
	wire Smin_enable_q1; // Latch control for Servo min
	wire Clock_enable_q1; // Latch control for clock scaler
	wire PortB_read_s1; // Put Port B onto IO bus
	wire WRbus_s1; // Put ALU results on bus
	wire ADC0_read_s1; // Put ADC0 onto IO bus
	wire ADC1_read_s1; // Put ADC1 onto IO bus
	wire ADC2_read_s1; // Put ADC2 onto IO bus
	wire BusReg_enable_q1; // Latch Control for Bus Capture
	wire Test_sel_s2; // Facilitates testing by putting bus result on
	wire AddrMux_s1;
	wire OpA_sel_s1;
	control u0 (.Phi1(Phi) , .Phi2(Phi2) , IOaddr_s1(IOaddr_s1) ,
	.Opcode_s1(Opcode_s1) ,
	.Zero_s2(Zero_s2) , .Reset_b_v2(Reset_b_v2) , .Reset_b_s1(Reset_b_s1) ,
	.Reset_b_s2(Reset_b_s2) , .A0_s2(AO_s2) , .WR_q1(WR_q1),
	.CE_b_q2(CE_b_q2), .IRL_enable_q2(IRL_enable_q2) ,
	.IRH_enable_q2(IRH_enable_q2) , .IOaddr_sel_s1(IOaddr_sel_s1) ,
	.PC_enable_q1(PC_enable_q1) , .PC_enable_q2(PC_enable_q2) ,
	.Load_PC_s2(Load_PC_s2), .Operands_enable_q1(Operands_enable_q1),
	.ALU_enable_q2(ALU_enable_q2) , .IMM_sel_s1(IMM_sel_s1) ,
	.Bstatus_enable_q1(Bstatus_enable_q1) ,
	.Bdata_enable_q1(Bdata_enable_q1),

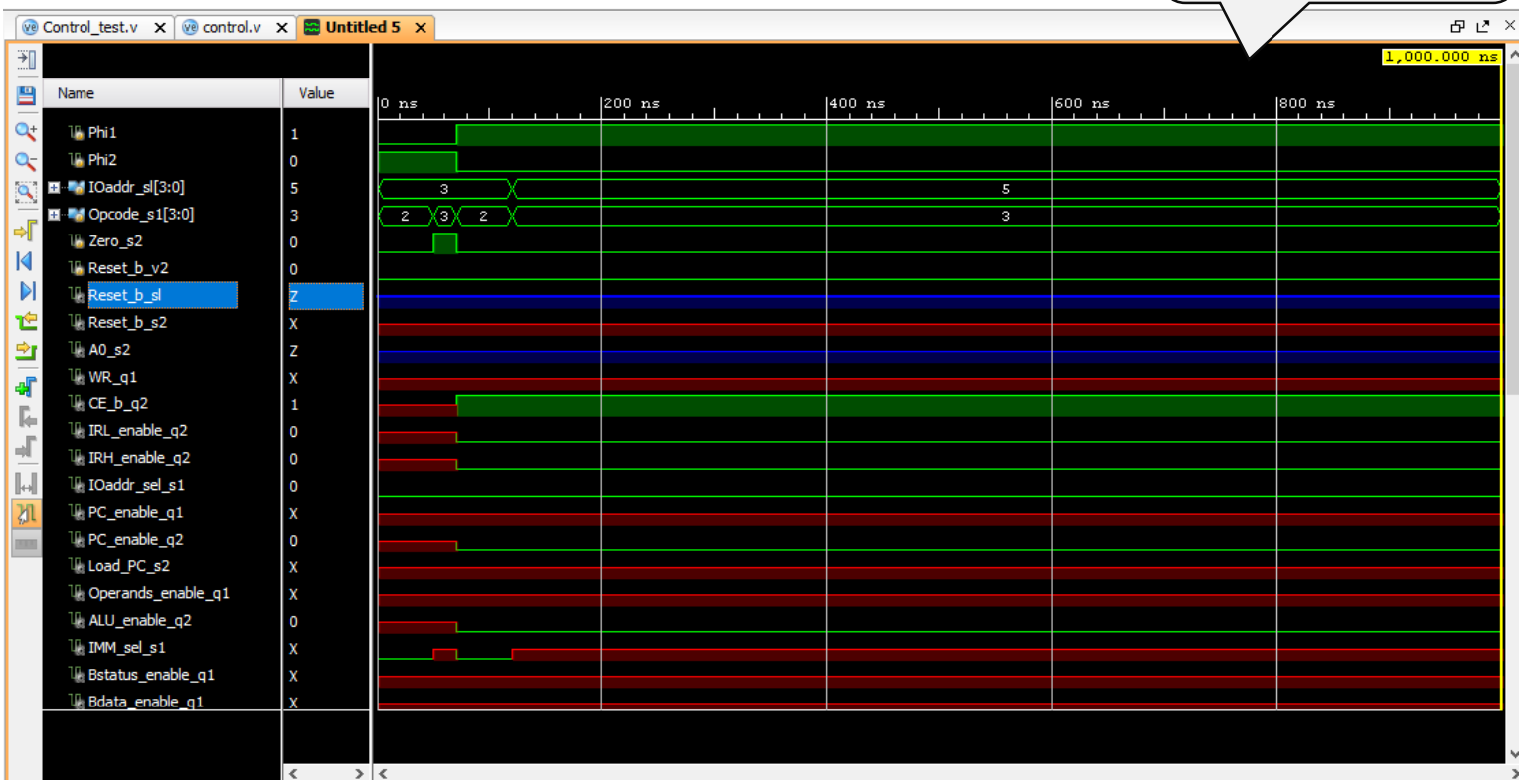
```
.M1speed_enable_q1(M1speed_enable_q1) ,
.M1status_enable_q1(M1status_enable_q1),
.M2speed_enable_q1(M2speed_enable_q1),
.M2status_enable_q1(M2status_enable_q1) ,
.Servo_enable_q1(Servo_enable_q1) ,
.Smin_enable_q1(Smin_enable_q1) , .Clock_enable_q1(Clock_enable_q1)
, .PortB_read_s1(PortB_read_s1),
.WRbus_s1(WRbus_s1) , .ADC0_read_s1(ADCO_read_s1) ,
.ADC1_read_s1(ADC1_read_s1) , .ADC2_read_s1(ADC2_read_s1),
.BusReg_enable_q1(BusReg_enable_q1) , .Test_sel_s2(Test_sel_s2) ,
.AddrMux_s1(AddrMux_s1) , .OpA_sel_s1(OpA_sel_s1)      );
initial begin
#0        Phi1=1'b0;
     Phi2=1'b1;
          IOaddr_s1=4'b0011;      // I/O register number
          Opcode_s1=4'b0010;     // Opcode from IR
      Zero_s2=1'b0;         // RD == 0 flag
     Reset_b_v2=1'b0;
     #50
          Phi1=1'b0;
          Phi2=1'b1;
           IOaddr_s1=4'b0011;    // I/O register number
           Opcode_s1=4'b0011;    // Opcode from IR
           Zero_s2=1'b1;    // RD == 0 flag
          Reset_b_v2=1'b0;
          #20
           Phi1=1'b1;
                Phi2=1'b0;
                 IOaddr_s1=4'b0011;    // I/O register number
                 Opcode_s1=4'b0010;    // Opcode from IR
                 Zero_s2=1'b0;    // RD == 0 flag
                Reset_b_v2=1'b0;
               #50
                         Phi1=1'b1;
                            Phi2=1'b0;
                             IOaddr_s1=4'b0101;    // I/O register number
                             Opcode_s1=4'b0011;    // Opcode from IR
                             Zero_s2=1'b0;    // RD == 0 flag
                            Reset_b_v2=1'b0;
end
endmodule
```

Control Module Diagram

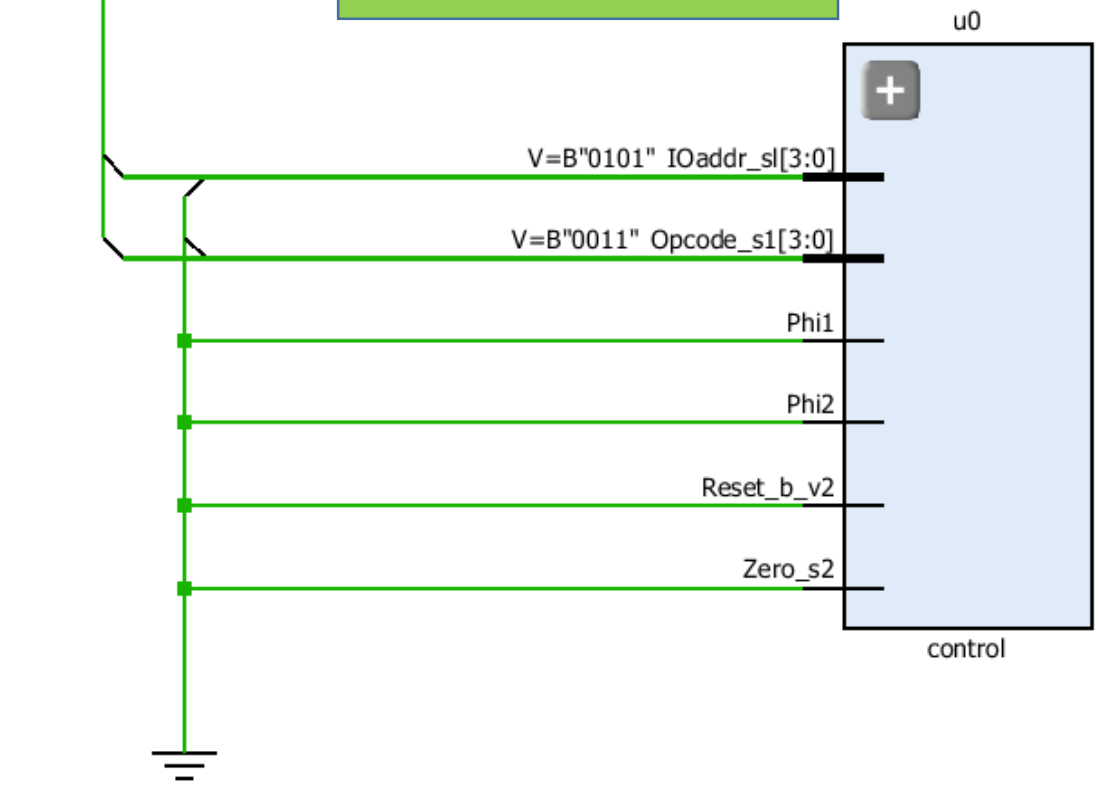


Simulation Waveform

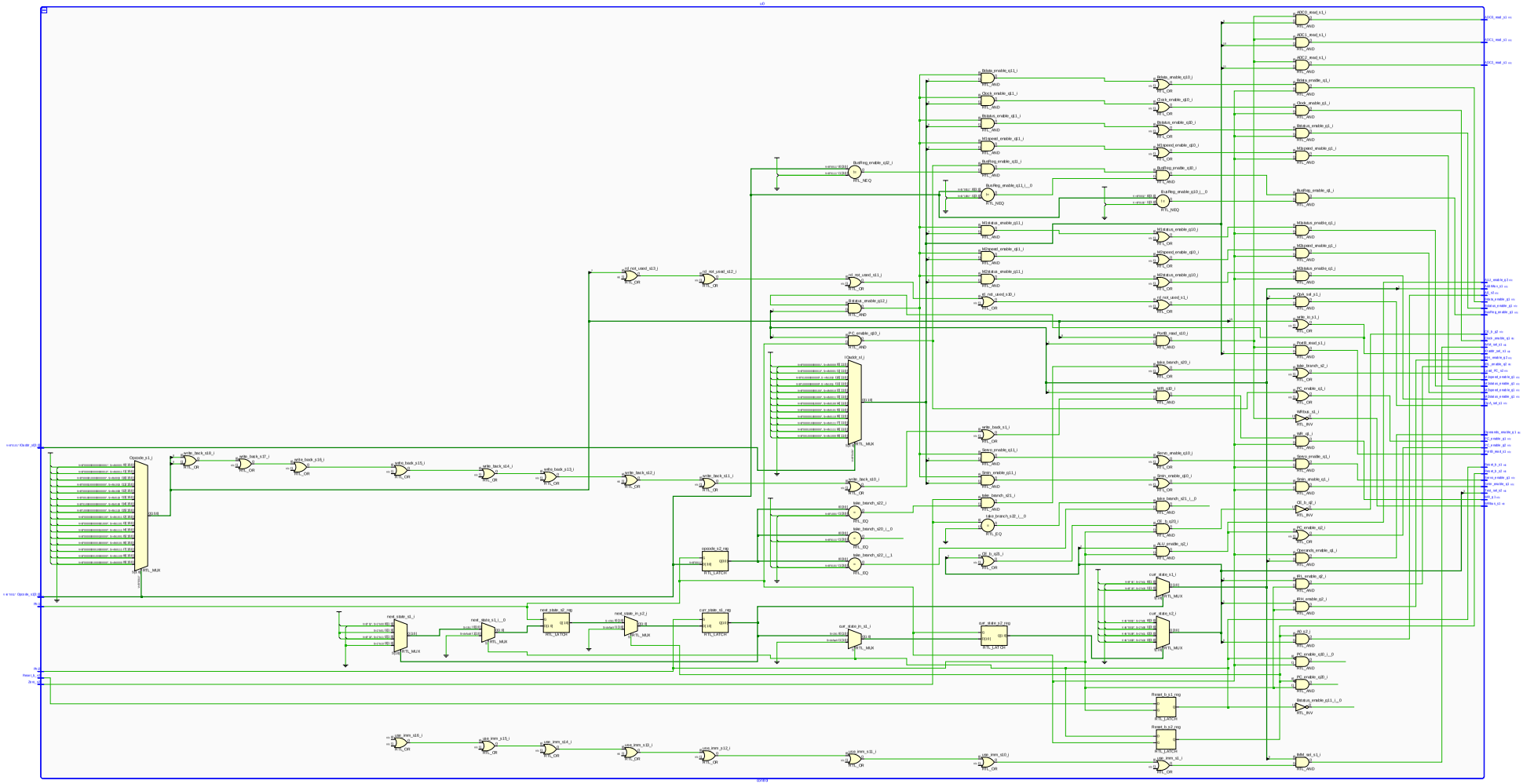
Control



RTL Analyze Schematic



For view details, you can zoom in



Datapath

Source Code

```
`define  OP       7:4
`define  RD       3:0
`define RS        7:4
`define ALU_OP 7:5
module datapath(Reset_b_s1, Reset_b_s2, Data_s2,
Address_s2,
                IObus_out_s1, Opcode_s1, IOaddr_s1,
Zero_s2, IRL_enable_q2,
                IRH_enable_q2, PC_enable_q1,
PC_enable_q2, Load_PC_s2, IOaddr_sel_s1,
Operands_enable_q1, ALU_enable_q2, IMM_sel_s1,
RSaddr_s1, RDaddr_s1, RSin_v1, RDin_v1,OpA_sel_s1);
input             Reset_b_s1,     // Global Reset
                  Reset_b_s2;
input    [7:0]    Data_s2;        // Data pins from external
memory
output  [7:0]     Address_s2;     // Address pins to external
memory
output  [7:0]     IObus_out_s1;   // Output to I/O data bus
output  [3:0]     Opcode_s1;      // Opcode from IR
output  [3:0]     IOaddr_s1;      // IO address from IR
output            Zero_s2;        // Zero flag
input             IRL_enable_q2; // Latch control for low bits
of IR
input             IRH_enable_q2;// Latch control for high bits
of IR
input             PC_enable_q1,  // Latch controls for PC
                  PC_enable_q2;
input             Load_PC_s2;     // PC <- Branch or Jump
target
input             IOaddr_sel_s1;
input    Operands_enable_q1;   // Latches regfile outputs
input             ALU_enable_q2;          // Latches ALU result
input             IMM_sel_s1;     // Selects input to ALU
operand B
output [3:0]      RSaddr_s1,
                  RDaddr_s1;
input  [7:0]    RSin_v1,
                  RDin_v1;
input        OpA_sel_s1;
wire     [7:0]    Address_s2;
wire     [3:0]    Opcode_s1,
                  IOaddr_s1;
wire     [7:0]    IObus_out_s1;
wire              Zero_s2;
wire     [3:0]    RSaddr_s1,
                  RDaddr_s1;
// Local signals for FETCH stage
reg      [7:0]    IRL_s1;         // Low 8-bits of IR
reg      [7:0]    IRH_s1;         // High 8-bits of IR
```

Test Bench

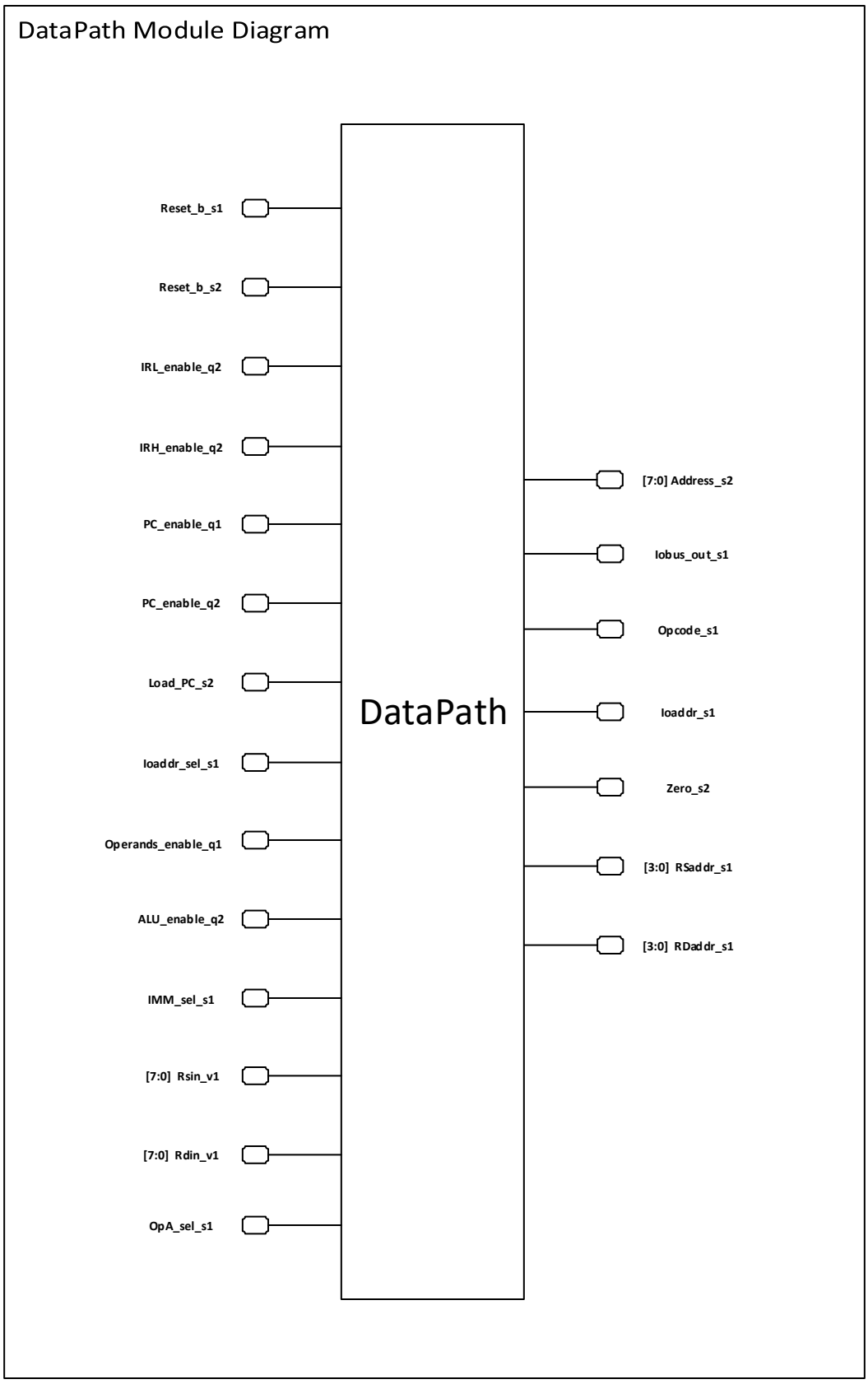
```
module Datapath_test;
reg               Reset_b_s1,     // Global Reset
                  Reset_b_s2;
reg      [7:0]    Data_s2;        // Data pins from external
memory
wire     [7:0]    Address_s2;     // Address pins to external
memory
wire     [7:0]    IObus_out_s1;   // Output to I/O data bus
wire     [3:0]    Opcode_s1;      // Opcode from IR
wire     [3:0]    IOaddr_s1;      // IO address from IR
wire              Zero_s2;        // Zero flag
reg               IRL_enable_q2; // Latch control for low bits
of IR
reg               IRH_enable_q2;// Latch control for high bits
of IR
reg               PC_enable_q1,  // Latch controls for PC
                  PC_enable_q2;
reg               Load_PC_s2;     // PC <- Branch or Jump
target
reg               IOaddr_sel_s1;
reg      Operands_enable_q1;   // Latches regfile outputs
reg               ALU_enable_q2;          // Latches ALU result
reg               IMM_sel_s1;     // Selects input to ALU
operand B
wire  [3:0]       RSaddr_s1,
                  RDaddr_s1;
reg  [7:0]    RSin_v1,
                  RDin_v1;
reg       OpA_sel_s1;

datapath u0( .Reset_b_s1(Reset_b_s1) ,
.Reset_b_s2(Reset_b_s2) ,
.Data_s2(Data_s2) , .Address_s2(Address_s2) ,
.IObus_out_s1(IObus_out_s1) ,
.Opcode_s1(Opcode_s1) , .IOaddr_s1(IOaddr_s1) ,
.Zero_s2(Zero_s2) , .IRL_enable_q2(IRL_enable_q2) ,
.IRH_enable_q2(IRH_enable_q2) ,
.PC_enable_q1(PC_enable_q1) ,
.PC_enable_q2(PC_enable_q2) , .Load_PC_s2(Load_PC_s2) ,
.IOaddr_sel_s1(IOaddr_sel_s1) ,
.Operands_enable_q1(Operands_enable_q1) ,
.ALU_enable_q2(ALU_enable_q2) ,
.IMM_sel_s1(IMM_sel_s1) ,
.RSaddr_s1(RSaddr_s1) , .RDaddr_s1(RDaddr_s1) ,
.RSin_v1(RSin_v1) ,  .RDin_v1(RDin_v1) ,
.OpA_sel_s1(OpA_sel_s1)    );
initial begin
#0
          Reset_b_s1=1'b0;
                  Reset_b_s2=1'b0;
```

```
reg     [7:0]   PC_s2;          // Program Counter
reg     [7:0]   PC_s1;
wire    [7:0]   PCinc_s2;       // PC + 1
wire    [7:0]   PCmux_s2;       // Multiplexed input to PC
// local signals for DECODE stage
reg     [7:0]   OpA_s2;                 // RD operand latch
reg     [7:0]   OpB_s2;                 // RS operand latch
reg     [2:0]   ALU_op_s2;      // ALU operation latch
wire    [7:0]   OpBmux_v1;      // Multiplexed input to ALU
operand B
wire   [7:0]    OpAmux_v1;
// local signals for EXECUTE stage
reg     [7:0]   ALU_s1;                 // ALU result latch
wire    [7:0]   ALU_out_s2;     // ALU result
// FETCH stage latches
always @(Data_s2 or IRL_enable_q2)
 if (IRL_enable_q2) IRL_s1 = Data_s2;    // Latch low byte of
IR
always @(Data_s2 or IRH_enable_q2)
 if (IRH_enable_q2) IRH_s1 = Data_s2;   // Latch high byte of
IR
always @(PC_s1 or PC_enable_q1 or Reset_b_s1)
 if (PC_enable_q1)
  if (Reset_b_s1)
   PC_s2 = PC_s1;        // Latch new PC value
  else
   PC_s2 = 8'b0;
always @(PCmux_s2 or PC_enable_q2 or Reset_b_s2)
 if (PC_enable_q2)
  if (Reset_b_s2)
   PC_s1 = PCmux_s2;// Latch PC + 1 value
  else
   PC_s1 = 8'b0;
// FETCH stage combinational logic
//assignAddress_s1 = (MUX_ALU_s1) ? ALU_s1: PC_s1; //
Address <- PC
assign   Address_s2 = PC_s2;
assign   PCinc_s2 = PC_s2 + 1;          // Increment PC by 1
// Multiplex inputs to PC
assign   PCmux_s2 = (Load_PC_s2) ? OpB_s2 : PCinc_s2;
// DECODE stage latches
always @(Operands_enable_q1 or OpBmux_v1)
 if (Operands_enable_q1) OpB_s2 = OpBmux_v1;         //
Latch Operand B
always @(Operands_enable_q1 or OpAmux_v1)
 if (Operands_enable_q1) OpA_s2 = OpAmux_v1;         //
Latch Operand A
always @(Operands_enable_q1 or IRH_s1)
 if (Operands_enable_q1) ALU_op_s2 = IRH_s1[`ALU_OP];
          // make ALU op s2
 // DECODE stage combinational logic
assign   RSaddr_s1 = IRL_s1[`RS];
assign  RDaddr_s1 = IRH_s1[`RD];
assign   Opcode_s1 = IRH_s1[`OP];
assign   IOaddr_s1 = (IOaddr_sel_s1) ? IRH_s1[`RD] :
IRL_s1[`RS];
```

```
        Data_s2=8'b00001111;
        IRL_enable_q2=1'b1;
  IRH_enable_q2=1'b0;
  PC_enable_q1=1'b1;
                PC_enable_q2=1'b1;
                Load_PC_s2=1'b1;
        IOaddr_sel_s1=1'b1;
        Operands_enable_q1=1'b1;
        ALU_enable_q2=1'b1;
        IMM_sel_s1=1'b0;;
          RSin_v1=8'b00001111;
                RDin_v1=8'b00111111;
           OpA_sel_s1=1'b1;
#50
       Reset_b_s1=1'b0;
         Reset_b_s2=1'b0;
       Data_s2=8'b00001111;
       IRL_enable_q2=1'b1;
       IRH_enable_q2=1'b0;
       PC_enable_q1=1'b0;
         PC_enable_q2=1'b1;
          Load_PC_s2=1'b0;
       IOaddr_sel_s1=1'b1;
        Operands_enable_q1=1'b0;
       ALU_enable_q2=1'b1;
       IMM_sel_s1=1'b0;;
         RSin_v1=8'b00000001;
          RDin_v1=8'b00111111;
          OpA_sel_s1=1'b1;
#50
            Reset_b_s1=1'b1;
              Reset_b_s2=1'b1;
            Data_s2=8'b00001111;
            IRL_enable_q2=1'b1;
            IRH_enable_q2=1'b0;
            PC_enable_q1=1'b0;
               PC_enable_q2=1'b1;
                Load_PC_s2=1'b1;
            IOaddr_sel_s1=1'b0;
             Operands_enable_q1=1'b1;
            ALU_enable_q2=1'b1;
            IMM_sel_s1=1'b0;;
              RSin_v1=8'b00000111;
               RDin_v1=8'b00001111;
               OpA_sel_s1=1'b0;
end
endmodule
```

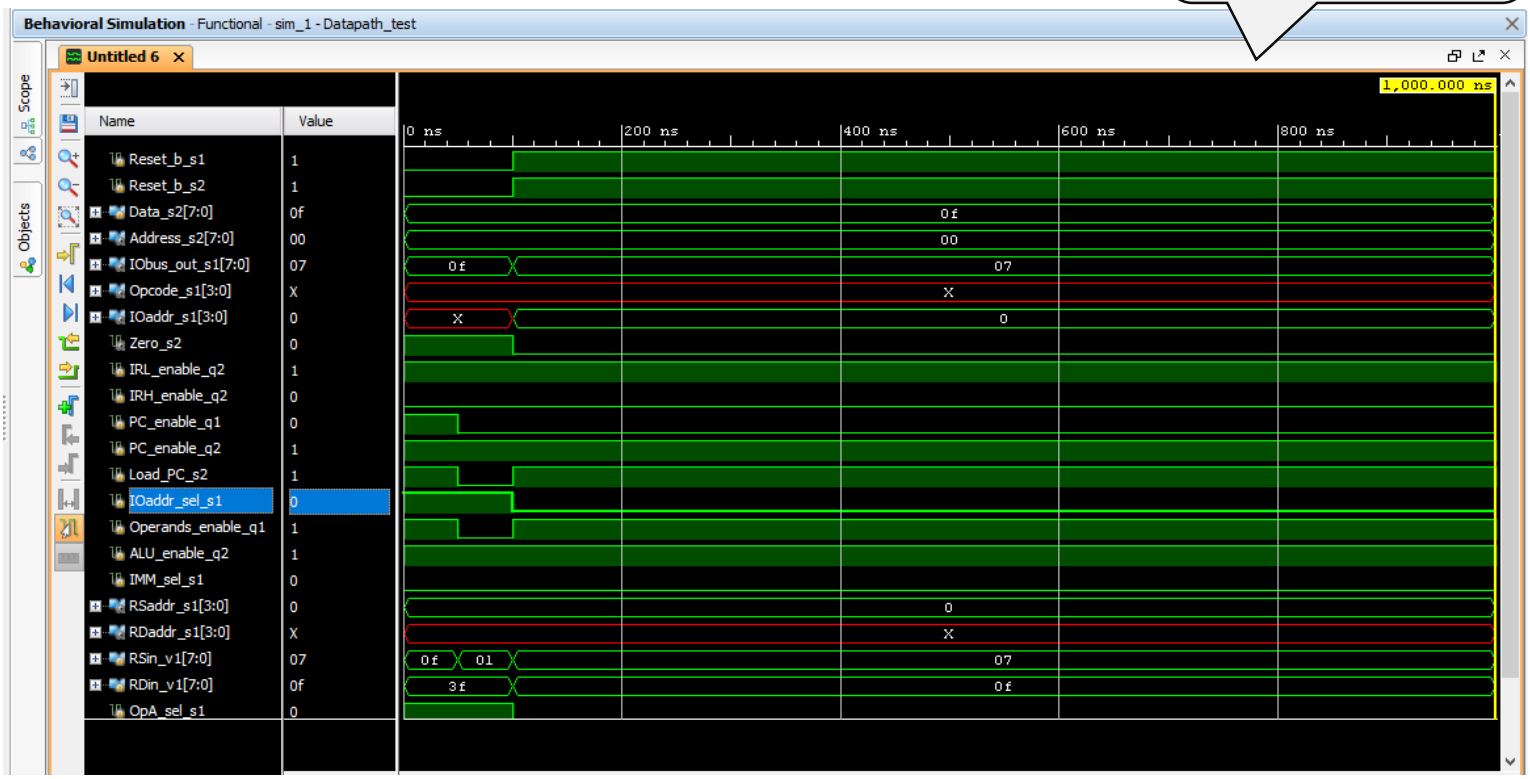
```
assign   OpBmux_v1 = (IMM_sel_s1) ? IRL_s1 : RSin_v1;
assign  OpAmux_v1 = (OpA_sel_s1) ? 8'b0 : RDin_v1;
// EXECUTE stage latch
always @(ALU_enable_q2 or ALU_out_s2)
  if (ALU_enable_q2) ALU_s1 = ALU_out_s2;        // Latch ALU
result
// EXECUTE stage combinational logic and ALU
alu alu(OpA_s2, OpB_s2, ALU_op_s2, ALU_out_s2, Zero_s2);
   assign IObus_out_s1 = ALU_s1;
endmodule // datapath
```

DataPath Module Diagram

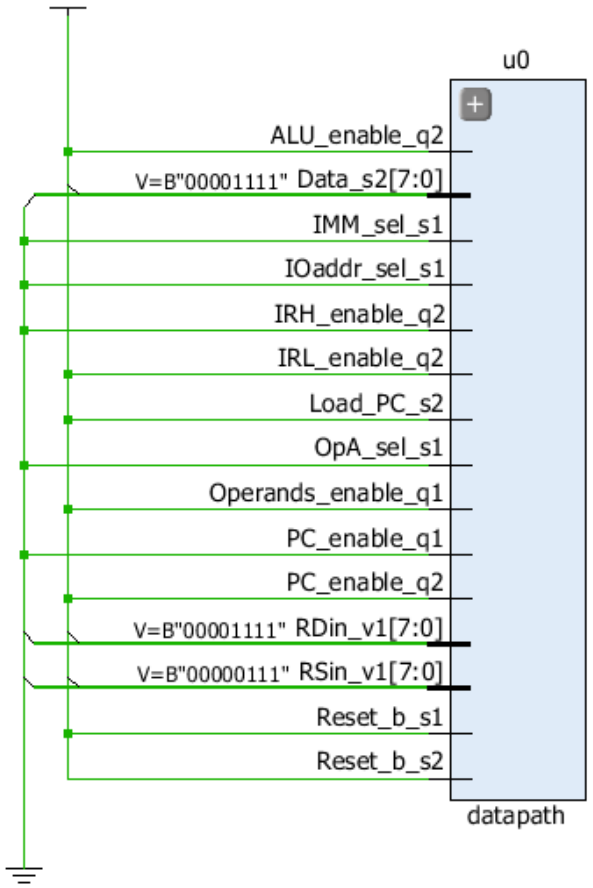


Simulation Waveform

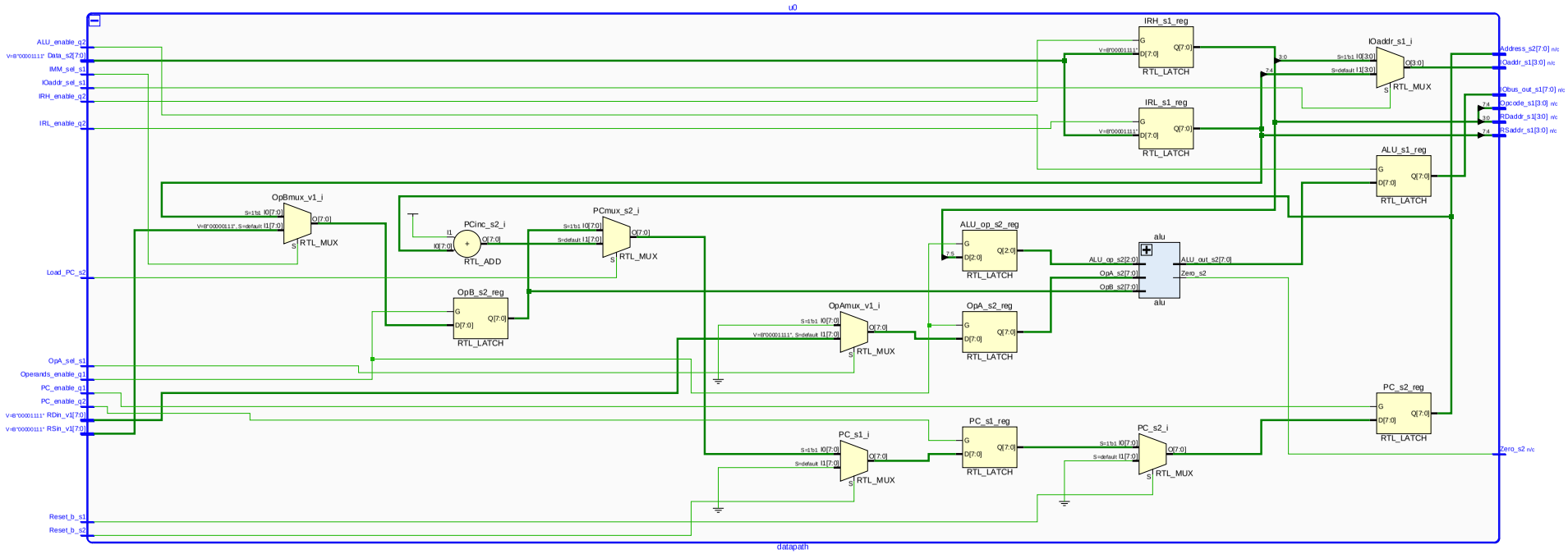
DataPath



RTL Analyze Schematic



For view details, you can zoom in



Flash Memory

Source Code	Test Bench

module flash_mem(Data, Address, A0, CE_b);
output [7:0] Data; // Data bus
input [7:0] Address; // Address bus
input A0; // High/Low select
input CE_b; // Latches Address
reg [7:0] Data;
reg [15:0] MEM[0:255];
wire [15:0] word;
// Simulate External Memory
`include "program.v"
assign word = MEM[Address];
always @(negedge CE_b)
if (A0 == 1'b0)
#10 Data = word[15:8];
else
#10 Data = word[7:0];
endmodule //
	module Flash_Mem_test;
	wire [7:0] Data;
	reg [7:0] Address;
	reg A0;
	reg CE_b;
	flash_mem u0 (.Data(Data) , .Address(Address) , .AO(AO) ,
	.CE_b(CE_b));
	initial begin
	Address=8'b00000001;
	A0=1'b0;
	CE_b=1'b1;
	#50
	Address=8'b00000001;
	AO=1'b1;
	CE_b=1'b0;
	end
	endmodule

Motor Port

Source Code	Test Bench
module motor_port (IObus_s1, Speed_en_q1, Status_en_q1, Speed_clk_q1, Speed_clk_q2, Reset_b_s1, Reset_b_s2, Fwd_pulse_s1, Rev_pulse_s1); // CONSTANT DECLARATIONS `define PWR 0 `define DIR 1 `define FWD 1'b1 `define REV 1'b0 `define ON 1'b1 `define OFF 1'b0 // PORT DECLARATIONS input [7:0] IObus_s1; input Speed_en_q1; input Status_en_q1; input Speed_clk_q1; input Speed_clk_q2; input Reset_b_s1, Reset_b_s2; output Fwd_pulse_s1; output Rev_pulse_s1; // VARIABLE DECLARATIONS reg [7:0] speed_reg_s2; reg [7:0] status_reg_s2; wire [7:0] speed_reg_in_s1, status_reg_in_s1; reg [7:0] speed_ctr_reg_s1; reg [7:0] speed_ctr_reg_s2; reg fwd_pulse_reg_s1;	module Motors_Ports_Test; reg [7:0] IObus_s1; reg Speed_en_q1; reg Status_en_q1; reg Speed_clk_q1; reg Speed_clk_q2; reg Reset_b_s1, Reset_b_s2; wire Fwd_pulse_s1; wire Rev_pulse_s1; motor_port u0(.IObus_s1(IObus_s1) , .Speed_en_q1(Speed_en_q1) , .Status_en_q1(Status_en_q1) , .Speed_clk_q1(Speed_clk_q1) , .Speed_clk_q2(Speed_clk_q2) , .Reset_b_s1(Reset_b_s1) , .Reset_b_s2(Reset_b_s2) , .Fwd_pulse_s1(Fwd_pulse_s1) , .Rev_pulse_s1(Rev_pulse_s1)); initial begin #0 IObus_s1=8'b00001111; Speed_en_q1=1'b1; Status_en_q1=1'b1; Speed_clk_q1=1'b1; Speed_clk_q2=1'b1; Reset_b_s1=1'b0; Reset_b_s2=1'b0; #50 IObus_s1=8'b00000111;

```
  reg          rev_pulse_reg_s1;
  wire [7:0]  speed_ctr_s1;
  wire [7:0]  speed_ctr_s2;
  wire       fwd_pulse_s2;
  wire       rev_pulse_s2;
// SPEED, STATUS, AND PULSE LATCHES
  always @(Speed_en_q1 or speed_reg_in_s1)
    if (Speed_en_q1)
      speed_reg_s2 = speed_reg_in_s1;
  assign speed_reg_in_s1 = (Reset_b_s1) ? IObus_s1 : 8'b0;
  always @(Status_en_q1 or status_reg_in_s1)
    if (Status_en_q1)
      status_reg_s2 = status_reg_in_s1;
  assign status_reg_in_s1 = (Reset_b_s1) ? IObus_s1 : 8'b0;
  always @(Speed_clk_q2 or fwd_pulse_s2)
    if (Speed_clk_q2)
      fwd_pulse_reg_s1 = fwd_pulse_s2;
  always @(Speed_clk_q2 or rev_pulse_s2)
    if (Speed_clk_q2)
      rev_pulse_reg_s1 = rev_pulse_s2;
// SPEED COUNTER
  always @(Speed_clk_q1 or speed_ctr_s1)
      if (Speed_clk_q1)
            speed_ctr_reg_s2 = speed_ctr_s1;
  always @(Speed_clk_q2 or speed_ctr_s2 or Reset_b_s2)
    if (Speed_clk_q2)
            speed_ctr_reg_s1 = speed_ctr_s2;
  assign speed_ctr_s1 = speed_ctr_reg_s1;
  assign speed_ctr_s2 = (Reset_b_s2) ? speed_ctr_reg_s2 + 1
: 8'b0;
// GENERATION OF PWM SIGNAL
  assign fwd_pulse_s2 = ((status_reg_s2[`PWR] == `ON) &&
                  (status_reg_s2[`DIR] == `FWD) &&
                  (speed_reg_s2 > speed_ctr_reg_s2));
  assign rev_pulse_s2 = ((status_reg_s2[`PWR] == `ON) &&
                  (status_reg_s2[`DIR] == `REV) &&
                              (speed_reg_s2 >
speed_ctr_reg_s2));
  assign Fwd_pulse_s1 = fwd_pulse_reg_s1;
  assign Rev_pulse_s1 = rev_pulse_reg_s1;
endmodule // motor_port
```

```
          Speed_en_q1=1'b0;
          Status_en_q1=1'b1;
          Speed_clk_q1=1'b0;
          Speed_clk_q2=1'b1;
          Reset_b_s1=1'b0;
          Reset_b_s2=1'b1;
          #50
               IObus_s1=8'b00111100;
               Speed_en_q1=1'b1;
               Status_en_q1=1'b0;
               Speed_clk_q1=1'b0;
               Speed_clk_q2=1'b0;
               Reset_b_s1=1'b0;
               Reset_b_s2=1'b0;
end
endmodule
```

PortB

Source Code

```
module portb(IObus_s1, PortB_enable_q1,
Status_enable_q1,
            PortB_data_s2, PortB_dir_s2, Reset_b_s1);
  input  [3:0]    IObus_s1;              // Data from IO bus
  input           PortB_enable_q1,       // Latches Data
                  Status_enable_q1;      // Latches Status
  output          [3:0]    PortB_data_s2; // Output to pins
```

Test Bench

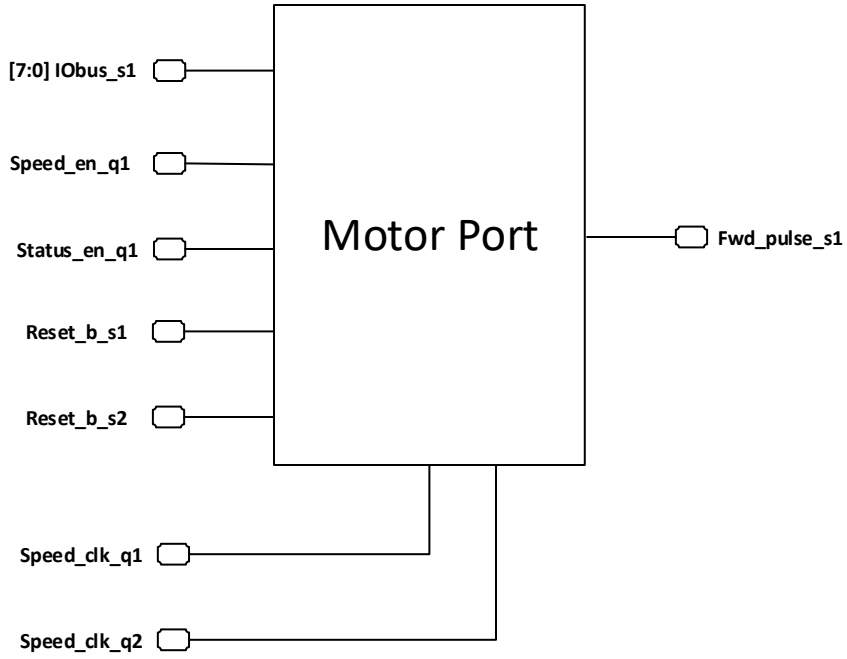
```
module PortB_test;
  reg    [3:0]    IObus_s1;              // Data from IO bus
  reg             PortB_enable_q1,       // Latches Data
                  Status_enable_q1;      // Latches Status
  wire   [3:0]    PortB_data_s2;         // Output to pins
  wire   [3:0]    PortB_dir_s2;          // In/Out pin
selector
  reg             Reset_b_s1;
```

```
  output        [3:0]    PortB_dir_s2;            // In/Out pin
selector
  input         Reset_b_s1;
  reg [3:0]               PortB_data_s2;
  reg [3:0]               PortB_dir_s2;
  wire [3:0]              data_in_s1, dir_in_s1;
  assign                  data_in_s1 = (Reset_b_s1) ?
IObus_s1 : 4'b0;
  assign                  dir_in_s1 = (Reset_b_s1) ? IObus_s1 :
4'b0;
always @(PortB_enable_q1 or data_in_s1)
  if (PortB_enable_q1) PortB_data_s2 = data_in_s1;
always @(Status_enable_q1 or dir_in_s1)
  if (Status_enable_q1) PortB_dir_s2 = dir_in_s1;
endmodule // portb
```

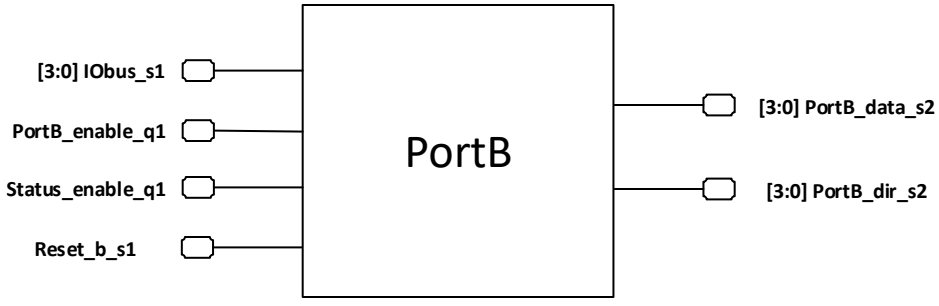
```
portb u0 (.IObus_s1(IObus_s1) ,
.PortB_enable_q1(PortB_enable_q1) ,
.Status_enable_q1(Status_enable_q1) ,
.PortB_data_s2(PortB_data_s2) ,
.PortB_dir_s2(PortB_dir_s2) , .Reset_b_s1(Reset_b_s1) );
initial begin
#0
      IObus_s1=4'b0001;
    PortB_enable_q1=1'b0;
          Status_enable_q1=1'b1;
    Reset_b_s1=1'b0;

    #50
         IObus_s1=4'b0011;
        PortB_enable_q1=1'b1;
        Status_enable_q1=1'b1;
        Reset_b_s1=1'b0;

        #50
             IObus_s1=4'b0111;
            PortB_enable_q1=1'b1;
            Status_enable_q1=1'b0;
            Reset_b_s1=1'b0;
end
endmodule
```

Flash Memory Module Diagram

Motor Port Module Diagram

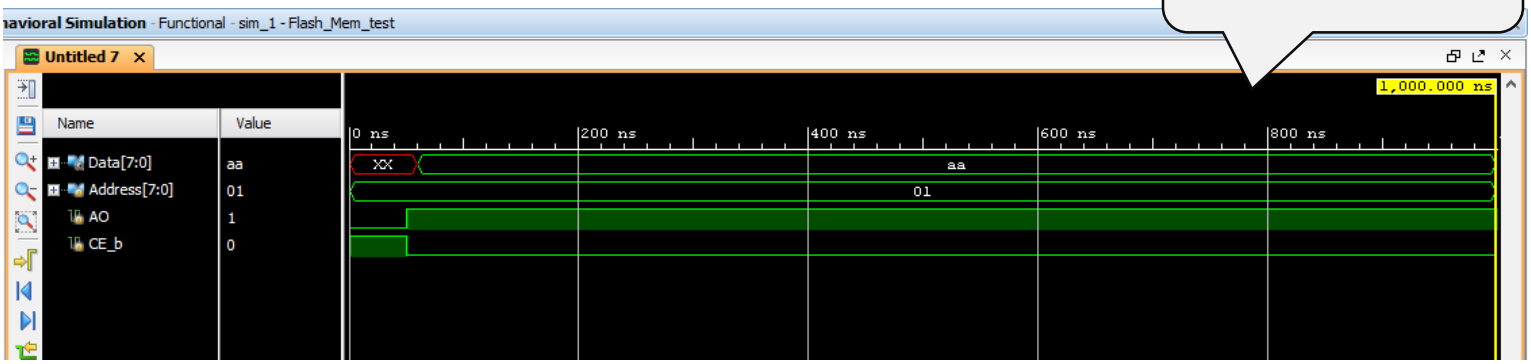


PortB Module Diagram

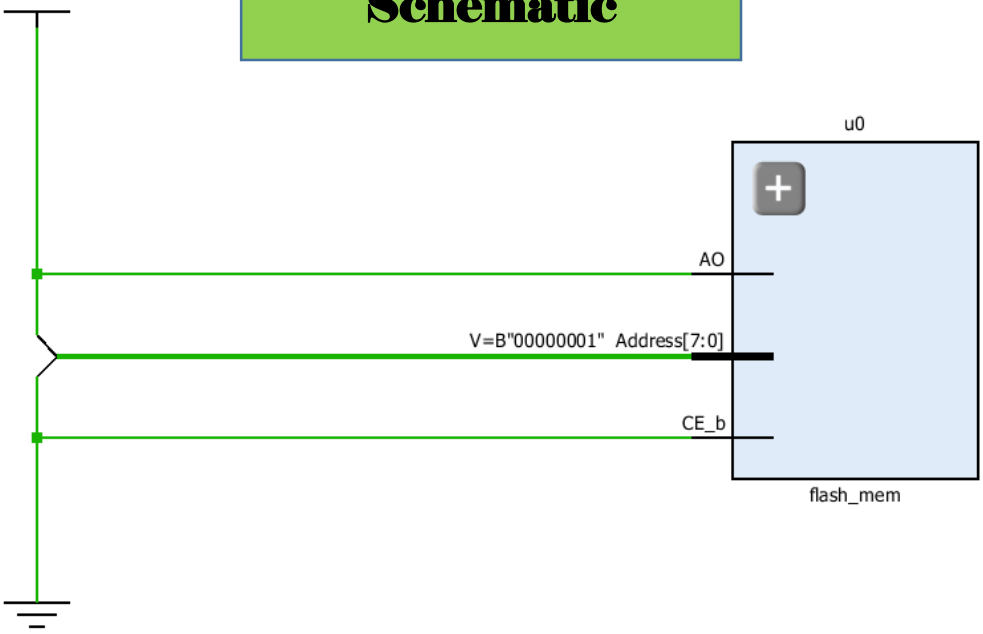


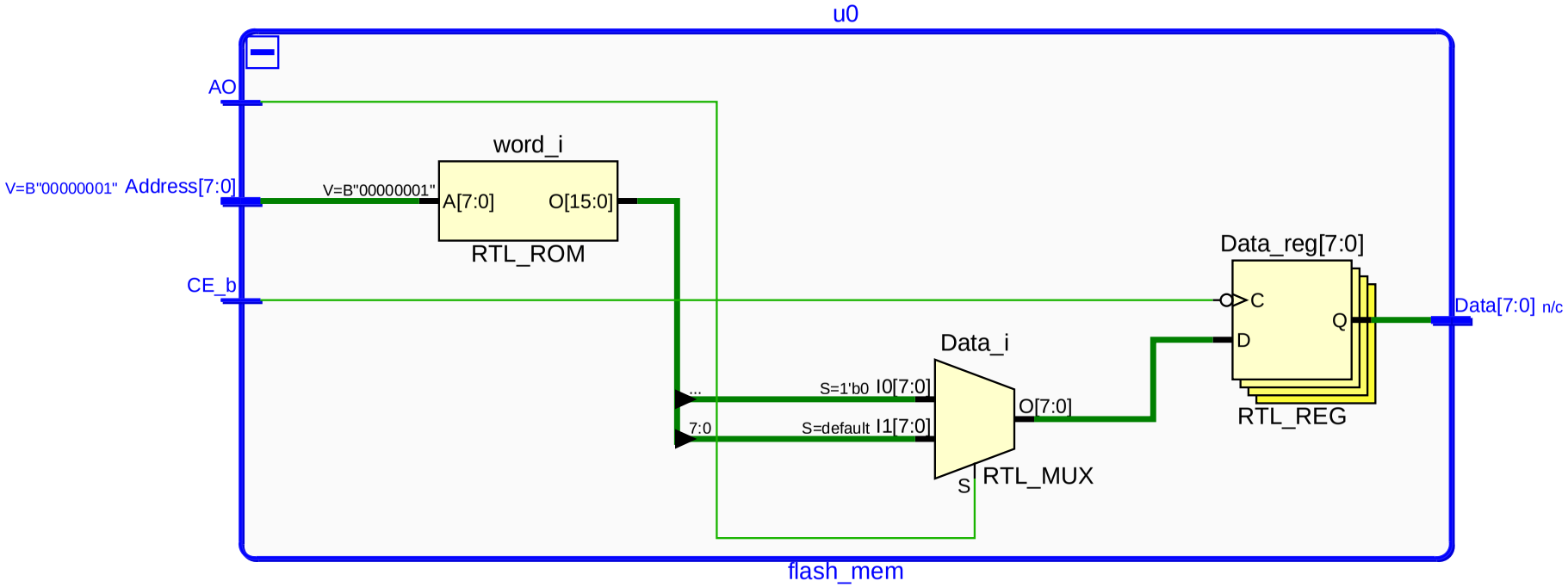
Simulation Waveform

Flash Memory



RTL Analyze Schematic

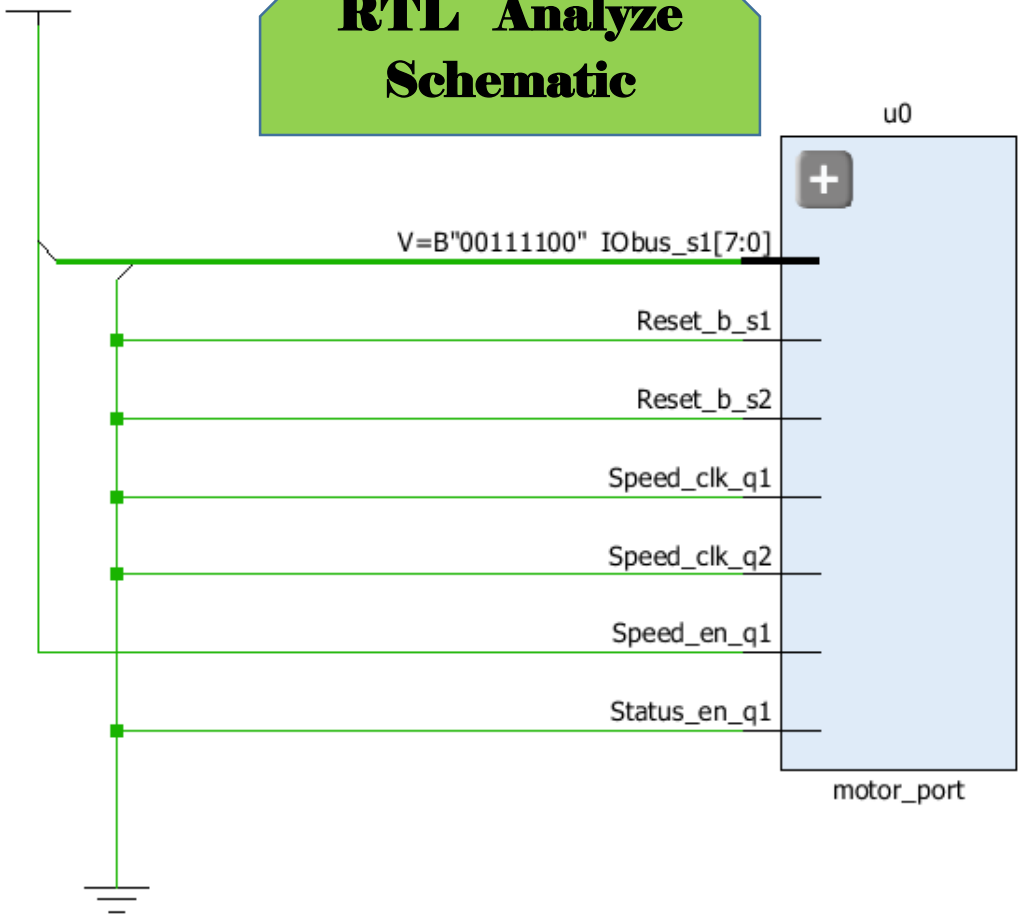



u0
AO
V=B"00000001"
Address[7:0]
CE_b
word_i
V=B"00000001"
A[7:0]
O[15:0]
RTL_ROM
Data_reg[7:0]
C
D
Q
RTL_REG
Data[7:0]
n/c
Data_i
...
S=1'b0
I0[7:0]
7:0
S=default
I1[7:0]
O[7:0]
S
RTL_MUX
flash_mem

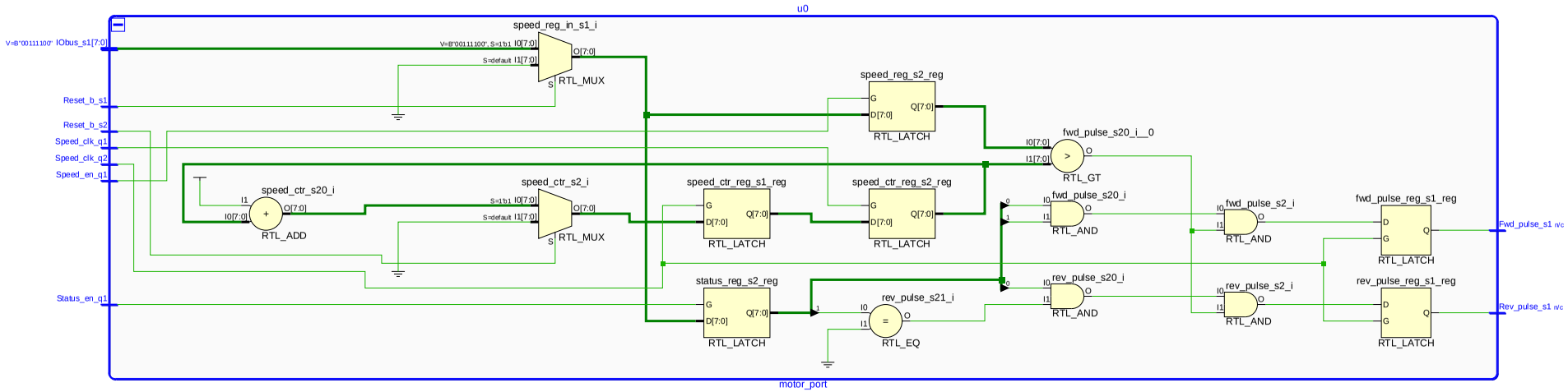
Simulation Waveform

Motor Port

RTL Analyze Schematic

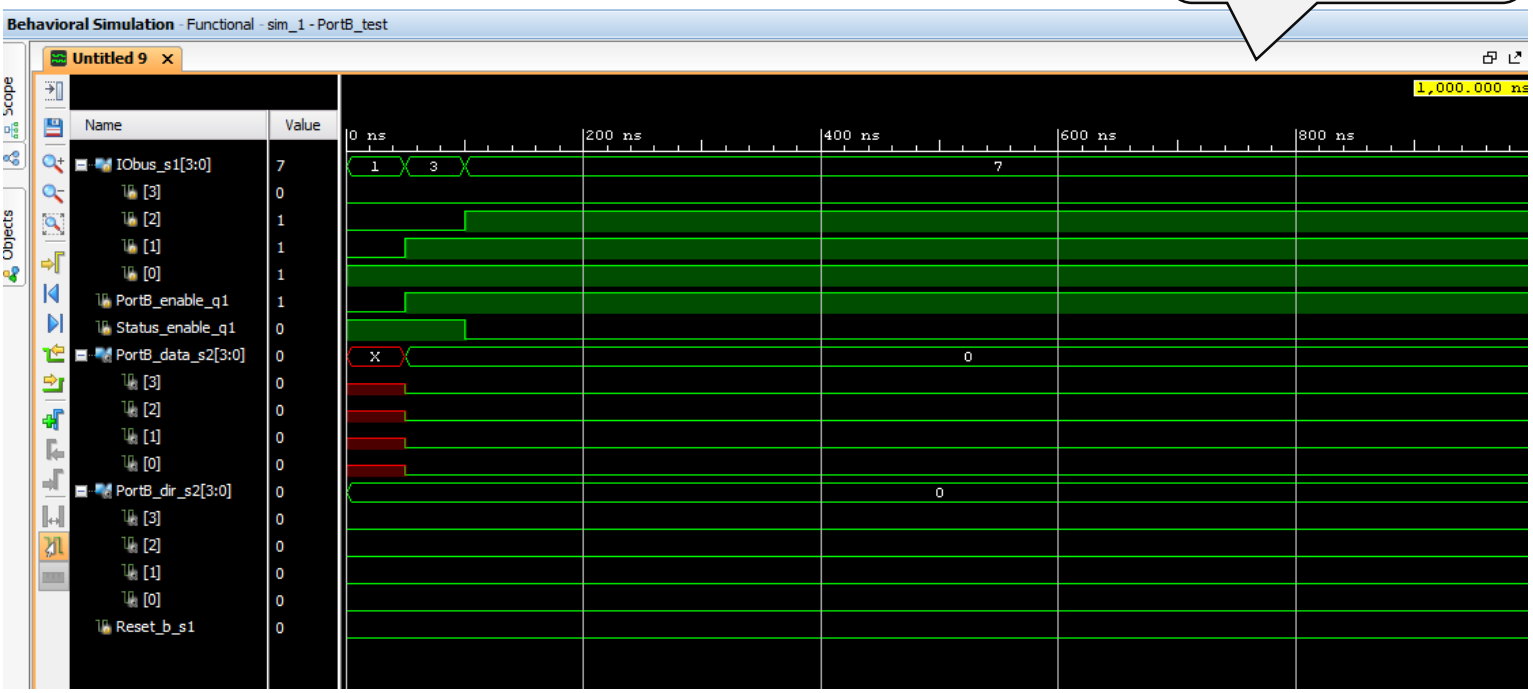


For view details, you can zoom in

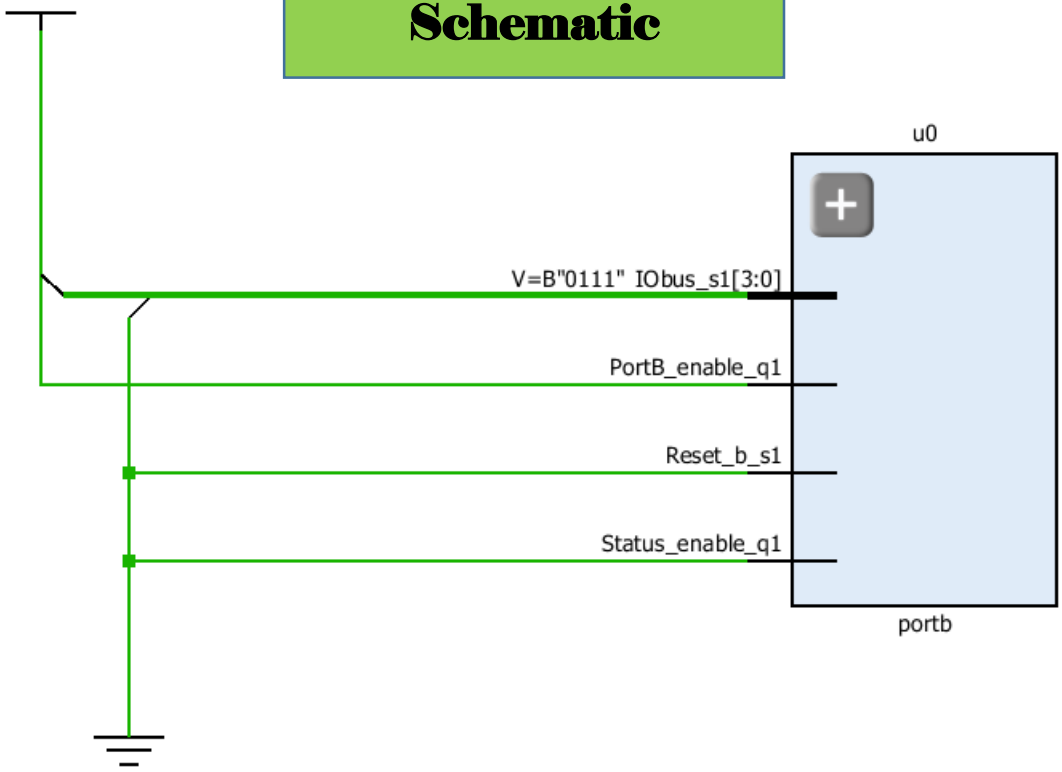


Simulation Waveform

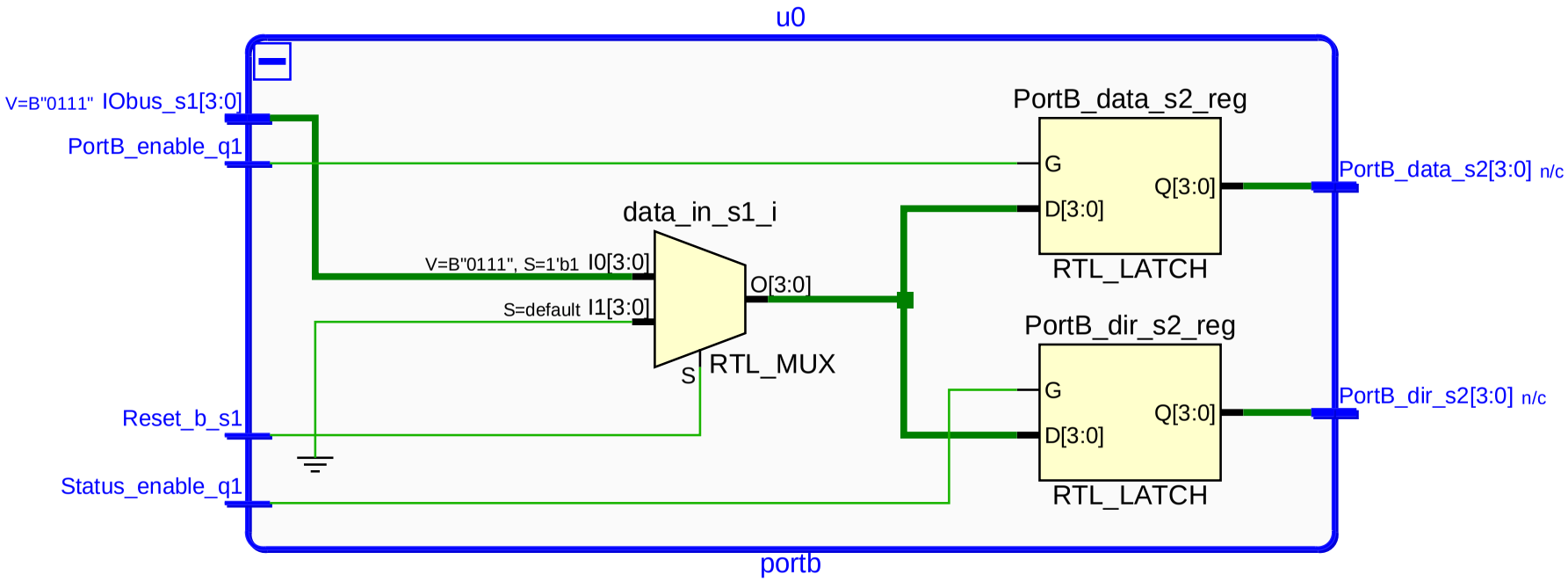
PortB



RTL Analyze Schematic



For view details, you can zoom in



Priority to Binary

Source Code

```
module prio2bin (Prio_s1, Prio_b_s1, Binary_s1);
// CONSTANT DECLARATIONS
// PORT DECLARATIONS
  input [14:0] Prio_s1;
  input [13:0] Prio_b_s1;
  output [3:0] Binary_s1;
// VARIABLE DECLARATIONS
  wire [14:0]     bin_in_s1;
// DECODER
  assign          bin_in_s1[0]  = Prio_s1[0]  && Prio_b_s1[0]  &&
Prio_b_s1[1];
  assign          bin_in_s1[1]  = Prio_s1[1]  && Prio_b_s1[1]  &&
Prio_b_s1[2];
  assign          bin_in_s1[2]  = Prio_s1[2]  && Prio_b_s1[2]  &&
Prio_b_s1[3];
  assign          bin_in_s1[3]  = Prio_s1[3]  && Prio_b_s1[3]  &&
Prio_b_s1[4];
  assign          bin_in_s1[4]  = Prio_s1[4]  && Prio_b_s1[4]  &&
Prio_b_s1[5];
  assign          bin_in_s1[5]  = Prio_s1[5]  && Prio_b_s1[5]  &&
Prio_b_s1[6];
  assign          bin_in_s1[6]  = Prio_s1[6]  && Prio_b_s1[6]  &&
Prio_b_s1[7];
  assign          bin_in_s1[7]  = Prio_s1[7]  && Prio_b_s1[7]  &&
Prio_b_s1[8];
  assign          bin_in_s1[8]  = Prio_s1[8]  && Prio_b_s1[8]  &&
Prio_b_s1[9];
  assign          bin_in_s1[9]  = Prio_s1[9]  && Prio_b_s1[9] &&
Prio_b_s1[10];
  assign          bin_in_s1[10] = Prio_s1[10] && Prio_b_s1[10] &&
Prio_b_s1[11];
  assign          bin_in_s1[11] = Prio_s1[11] && Prio_b_s1[11] &&
Prio_b_s1[12];
  assign          bin_in_s1[12] = Prio_s1[12] && Prio_b_s1[12] &&
Prio_b_s1[13];
  assign          bin_in_s1[13] = Prio_s1[13] && Prio_b_s1[13];
  assign          bin_in_s1[14] = Prio_s1[14];
  assign          Binary_s1[0] = bin_in_s1[0]  || bin_in_s1[2]  ||
bin_in_s1[4]  ||
                               bin_in_s1[6]  || bin_in_s1[8]  ||
bin_in_s1[10] ||
                               bin_in_s1[12] || bin_in_s1[14];
  assign          Binary_s1[1] = bin_in_s1[1]  || bin_in_s1[2]  ||
bin_in_s1[5]  ||
                               bin_in_s1[6]  || bin_in_s1[9]  ||
bin_in_s1[10] ||
                               bin_in_s1[13] || bin_in_s1[14];
  assign          Binary_s1[2] = bin_in_s1[3]  || bin_in_s1[4]  ||
bin_in_s1[5]  ||
                    bin_in_s1[6]  || bin_in_s1[11] || bin_in_s1[12] ||
```

Test Bench

```
module Prio2Binary_test ;
reg [14:0] Prio_s1;
reg [13:0] Prio_b_s1;
wire [3:0] Binary_s1;
prio2bin u0(.Prio_s1(Prio_s1) , .Prio_b_s1(Prio_b_s1) ,
.Binary_s1(Binary_s1) );
initial begin
#0
 Prio_s1=15'b000001111100000;
 Prio_b_s1=14'b00001111001111;
 #50
  Prio_s1=15'b000000011100000;
  Prio_b_s1=14'b00001111001111;
  #50
   Prio_s1=15'b000111111100011;
   Prio_b_s1=14'b00000011001111;
end
endmodule
```

```
                        bin_in_s1[13] || bin_in_s1[14];
  assign        Binary_s1[3] = bin_in_s1[7]  || bin_in_s1[8]  ||
bin_in_s1[9] ||
                        bin_in_s1[10] || bin_in_s1[11] ||
bin_in_s1[12] ||
                        bin_in_s1[13] || bin_in_s1[14];
endmodule // ad_enc.v
```

Register File

Source Code

```
module regfile(RSaddr_s1, RDaddr_s1, RSout_v1, RDout_v1,
WriteData_s1,
               WR_q1, AddrMux_s1, Phi1, Phi2);
input   [3:0]   RSaddr_s1;
input   [3:0]   RDaddr_s1;
output  [7:0]   RSout_v1;
output  [7:0]   RDout_v1;
input   [7:0]   WriteData_s1;
input           WR_q1;
input   AddrMux_s1;
reg     [7:0]   MEM_v1[0:15];
wire    [7:0]   RSout_v1, RDout_v1;
wire    [3:0]   WRaddr_s1;
wire   [3:0]  rs_addr_q1, rd_addr_q1;
  assign rs_addr_q1 = (Phi1) ? RSaddr_s1 : 4'bX;
  assign rd_addr_q1 = (Phi1) ? RDaddr_s1 : 4'bX;
  assign        WRaddr_s1 = (AddrMux_s1) ? RDaddr_s1 :
8'bZ;
  assign        RSout_v1 = (Phi2) ? MEM_v1[rs_addr_q1] :
8'b11111111;
  assign RDout_v1 = (Phi2) ? MEM_v1[rd_addr_q1] : 8'b0;
always @(WR_q1 or RDaddr_s1 or WriteData_s1)
   if (WR_q1) MEM_v1[WRaddr_s1] = WriteData_s1;
endmodule // regfile
```

Test Bench

```
module RegFile_test;
reg     [3:0]   RSaddr_s1;
reg     [3:0]   RDaddr_s1;
wire    [7:0]   RSout_v1;
wire    [7:0]   RDout_v1;
reg     [7:0]   WriteData_s1;
reg             WR_q1;
reg     AddrMux_s1;
regfile u0 (.       RSaddr_s1(RSaddr_s1)
,.RDaddr_s1(RDaddr_s1) ,
.RSout_v1(RSout_v1) , .RDout_v1(RDout_v1) ,
.WriteData_s1(WriteData_s1), .WR_q1(WR_q1) ,
.AddrMux_s1(AddrMux_s1) );
initial begin
#0
        RSaddr_s1=4'b0001;
        RDaddr_s1=4'b0011;
        WriteData_s1=8'b00001111;
        WR_q1=1'b1;
        AddrMux_s1=1'b1;
        #50
    RSaddr_s1=4'b0111;
    RDaddr_s1=4'b0001;
    WriteData_s1=8'b0011111;
    WR_q1=1'b1;
    AddrMux_s1=1'b0;
    #50
      RSaddr_s1=4'b0111;
      RDaddr_s1=4'b1001;
      WriteData_s1=8'b00000011;
      WR_q1=1'b0;
      AddrMux_s1=1'b1;
end
endmodule
```

Servo Port

Source Code

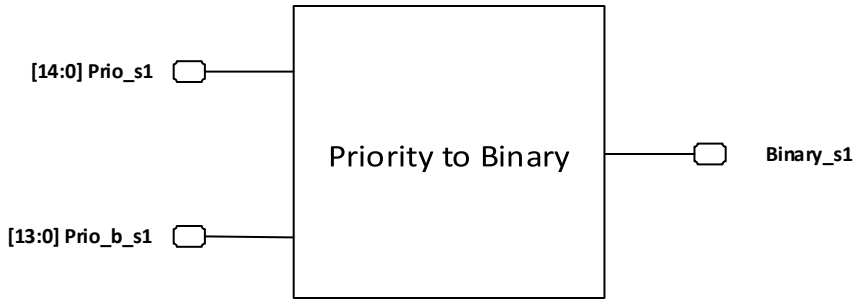
```
module servo_port (IObus_s1, Position_en_q1,
Min_pos_count_en_q1,Reset_b_s1, Reset_b_s2, Pos_clk_q1,
Pos_clk_q2,  Pulse_s1);
// CONSTANT DECLARATIONS
`define DEF_MIN_COUNT 8'd128
`define DEF_POSITION  8'd128
// PORT DECLARATIONS
  input      Pos_clk_q1, Pos_clk_q2;
  input [7:0] IObus_s1;
  input      Position_en_q1;
  input      Min_pos_count_en_q1;
  input      Reset_b_s1, Reset_b_s2;
  output     Pulse_s1;
// VARIABLE DECLARATIONS
  reg [7:0]   pos_reg_s2;
  reg [7:0]   pos_min_reg_s2;
  reg [9:0]   pos_ctr_reg_s1;
  reg [9:0]   pos_ctr_reg_s2;
  reg        pulse_reg_s1;
  wire [9:0]  pos_ctr_s1;
  wire [9:0]  pos_ctr_s2;
  wire       reset_s2;
  wire       pulse_s2;
  wire [9:0]  pulse_width_s2;
  wire [7:0]  position_in_s1,
                min_pos_in_s1;
// POSITION AND PULSE LATCHES
  always @(Position_en_q1 or position_in_s1)
    if (Position_en_q1)
      pos_reg_s2 = position_in_s1;
  assign     position_in_s1 = (Reset_b_s1) ? IObus_s1 :
`DEF_POSITION;
  always @(Min_pos_count_en_q1 or min_pos_in_s1)
    if (Min_pos_count_en_q1)
      pos_min_reg_s2 = min_pos_in_s1;
  assign     min_pos_in_s1 =  (Reset_b_s1) ? IObus_s1 :
`DEF_MIN_COUNT;
  always @(Pos_clk_q2 or pulse_s2)
    if (Pos_clk_q2)
      pulse_reg_s1 = pulse_s2;
// POSITION COUNTER
  always @(Pos_clk_q1 or pos_ctr_s1)
          if (Pos_clk_q1)
            pos_ctr_reg_s2 = pos_ctr_s1;
  always @(Pos_clk_q2 or pos_ctr_s2)
      if (Pos_clk_q2)
           pos_ctr_reg_s1 = pos_ctr_s2;
  assign pos_ctr_s1 = pos_ctr_reg_s1;
  assign pos_ctr_s2 = (Reset_b_s2 && !reset_s2) ?
pos_ctr_reg_s2 + 1 : 10'b0;
```

Test Bench

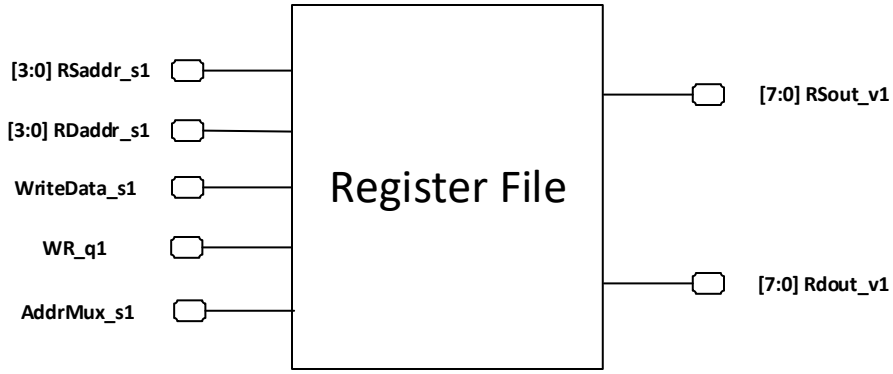
```
module Servo_Port_test;
  reg      Pos_clk_q1, Pos_clk_q2;
  reg      [7:0] IObus_s1;
  reg      Position_en_q1;
  reg      Min_pos_count_en_q1;
  reg      Reset_b_s1, Reset_b_s2;
  wire     Pulse_s1;
servo_port u0(.Pos_clk_q1(Pos_clk) ,
.Pos_clk_q2(Pos_clk_q2) ,
.IObus_s1(IObus_s1) , .Position_en_q1(Position_en_q1) ,
.Min_pos_count_en_q1(Min_pos_count_en_q1) ,
.Reset_b_s1(Reset_b_s1),
.Reset_b_s2(Reset_b_s2) , .Pulse_s1(Pulse_s1) );
initial begin
  #0
  Pos_clk_q1=1'b1;
  Pos_clk_q2=1'b0;
    IObus_s1=8'b00001111;
    Position_en_q1=1'b1;
    Min_pos_count_en_q1=1'b1;
    Reset_b_s1=1'b1;
     Reset_b_s2=1'b0;
      #50
     Pos_clk_q1=1'b0;
     Pos_clk_q2=1'b1;
       IObus_s1=8'b00000011;
       Position_en_q1=1'b1;
       Min_pos_count_en_q1=1'b0;
       Reset_b_s1=1'b0;
        Reset_b_s2=1'b1;
         #50
        Pos_clk_q1=1'b1;
        Pos_clk_q2=1'b1;
          IObus_s1=8'b11000011;
          Position_en_q1=1'b1;
          Min_pos_count_en_q1=1'b0;
          Reset_b_s1=1'b0;
           Reset_b_s2=1'b0;
end
endmodule
```

```
// GENERATION OF PWM SIGNAL
  assign pulse_width_s2 = pos_min_reg_s2 +  pos_reg_s2;
  assign pulse_s2  = pulse_width_s2 >  pos_ctr_reg_s2;
  assign reset_s2 = pos_ctr_reg_s2 >= {2'b10,
pos_min_reg_s2};
  assign Pulse_s1 = pulse_reg_s1;
endmodule // servo_port
```

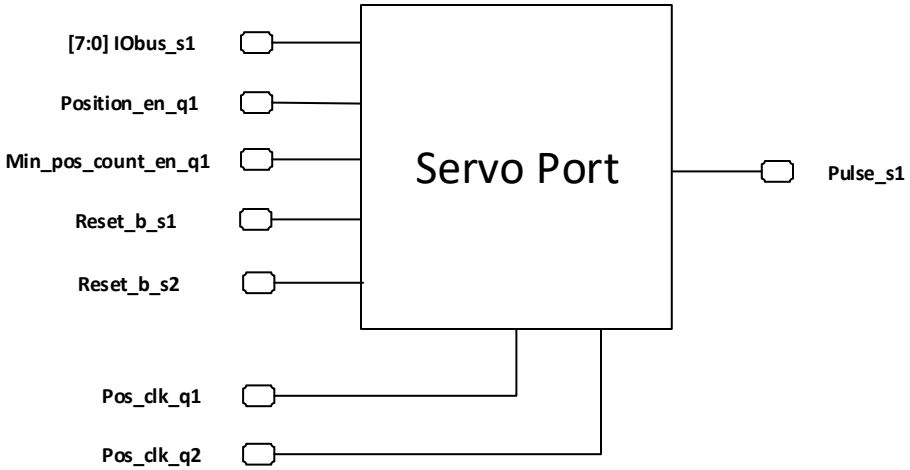
Priority to Binary Module Diagram



Register File Module Diagram

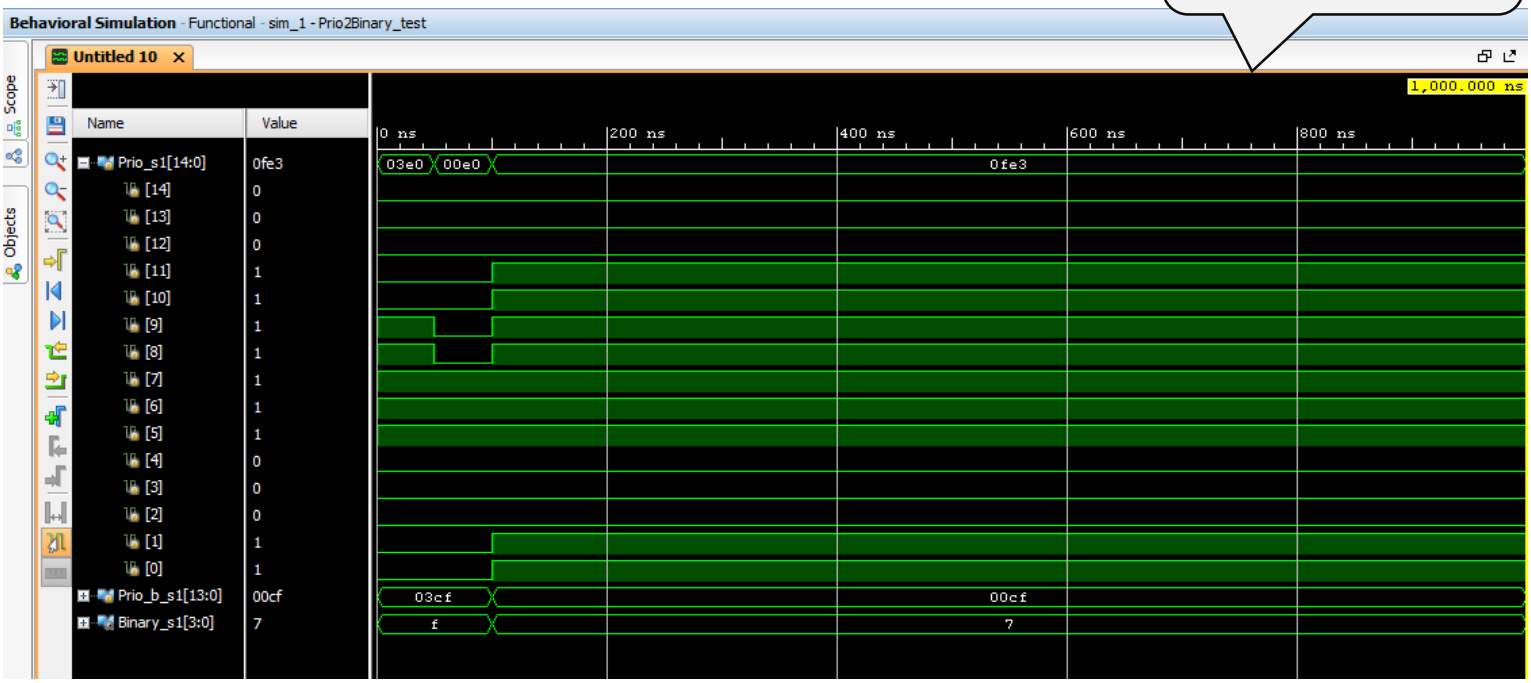


Servo Port Module Diagram

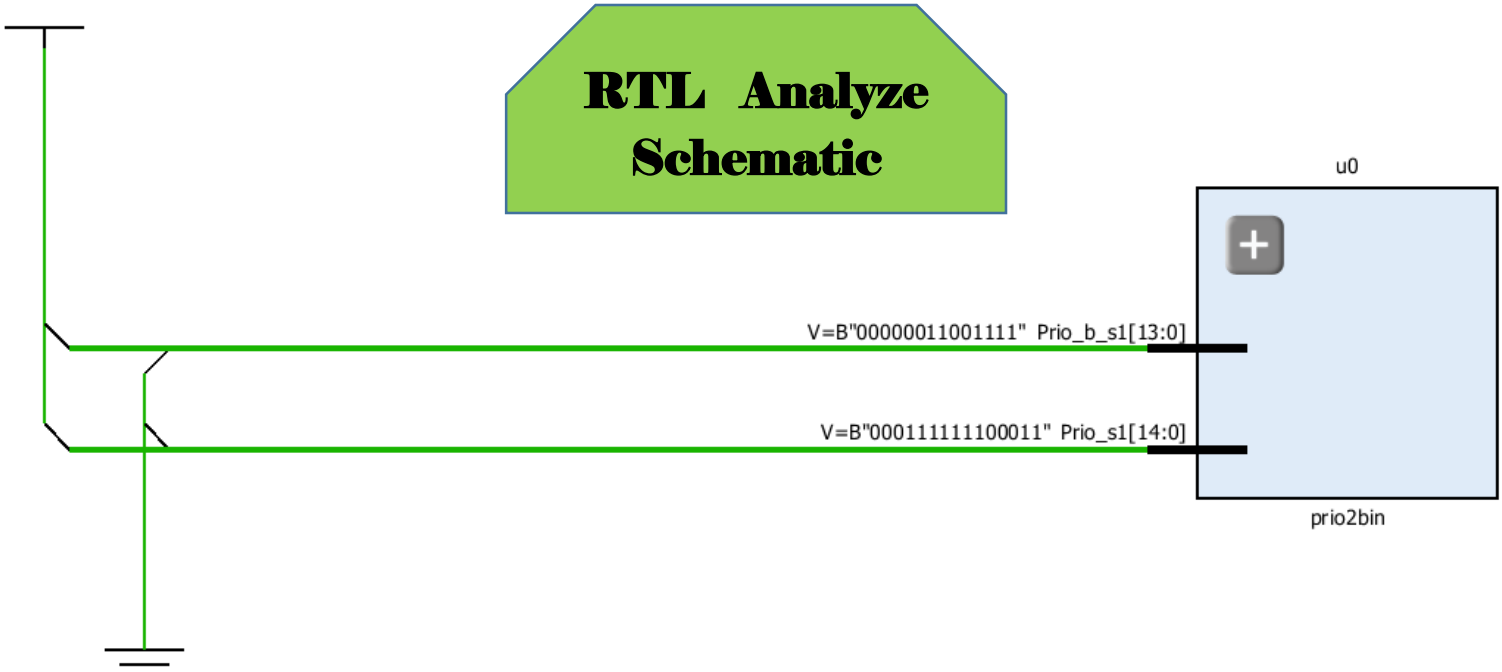


Simulation Waveform

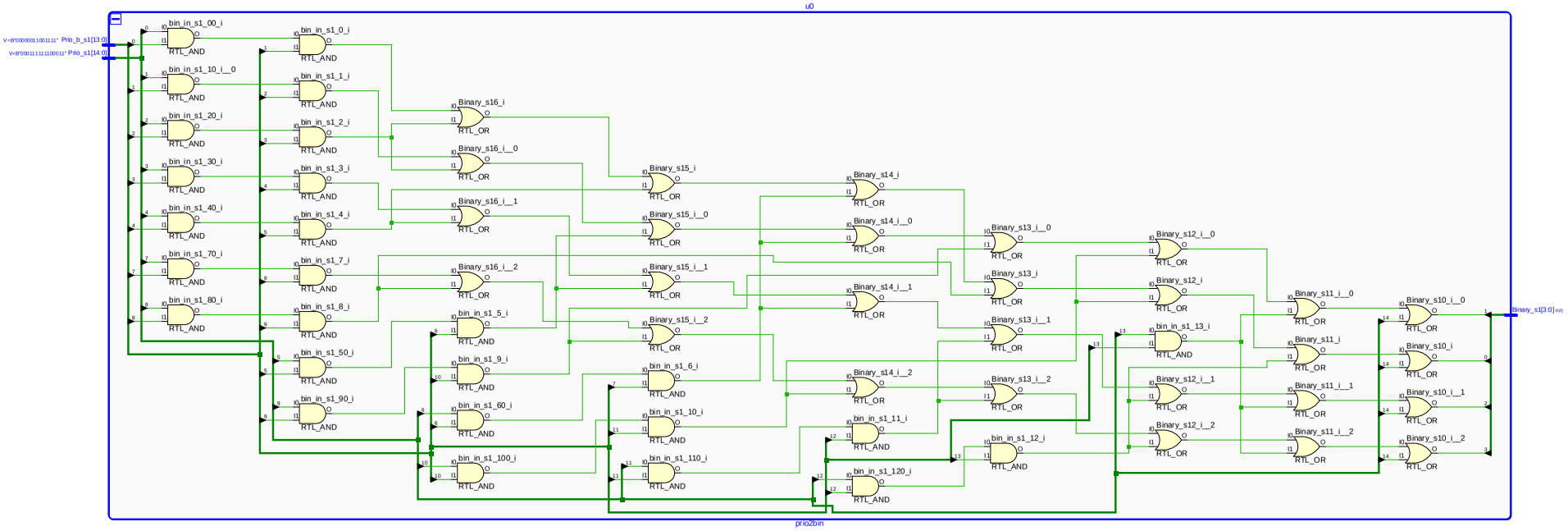
Prior 2 Binary



RTL Analyze Schematic

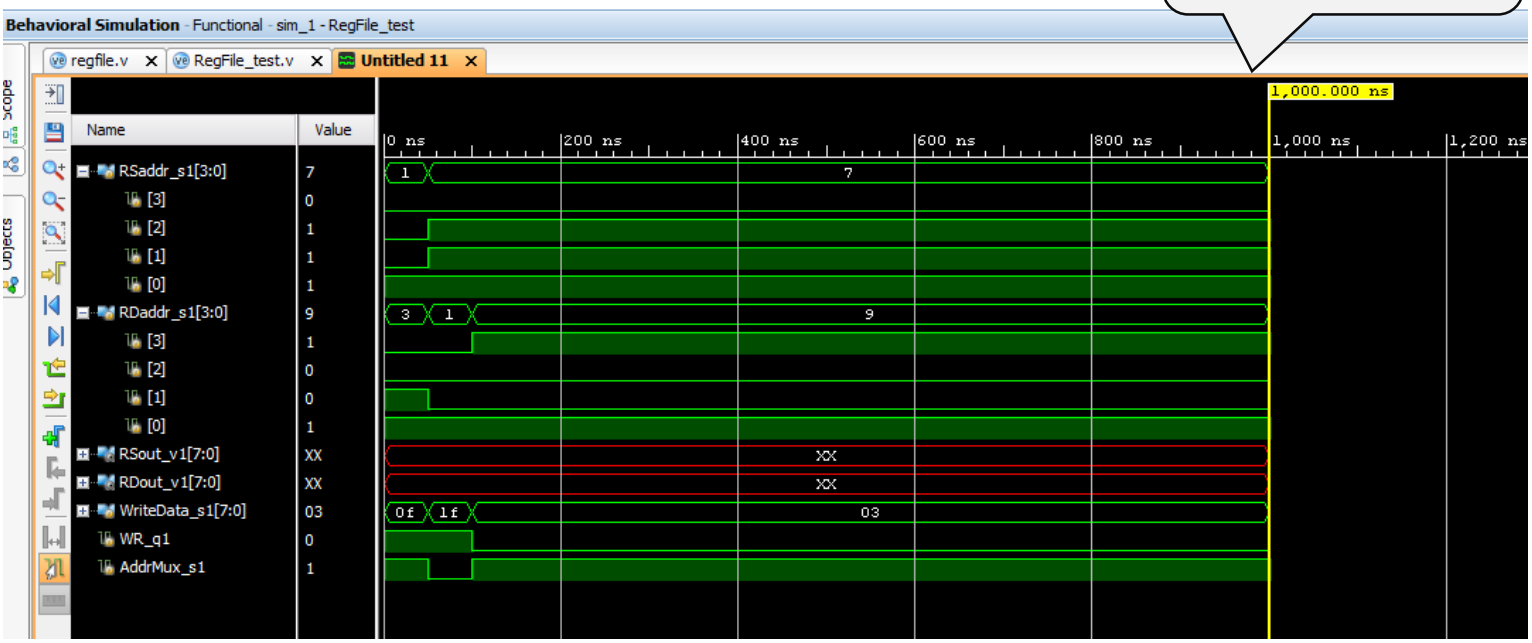


For view details, you can zoom in

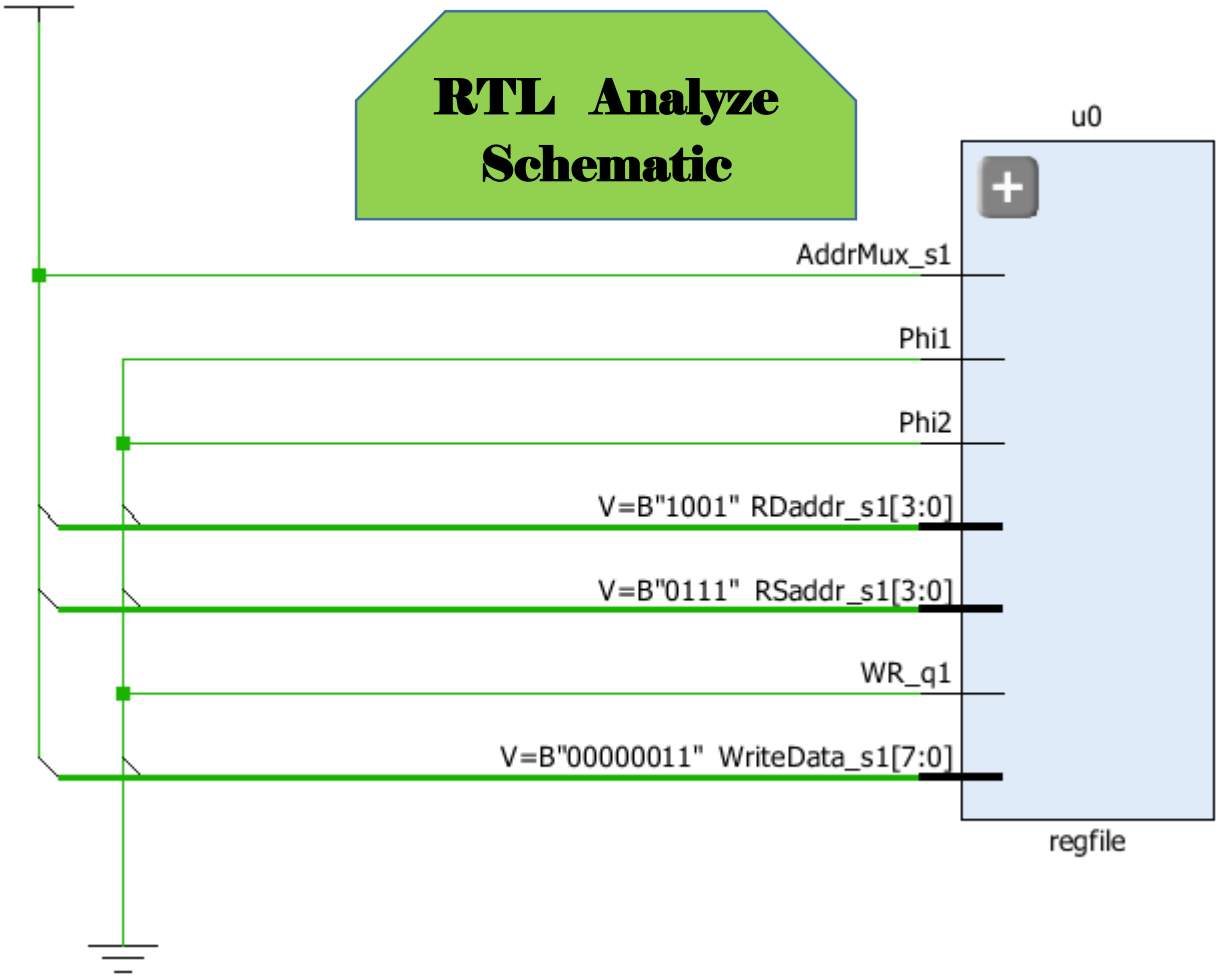


Simulation Waveform

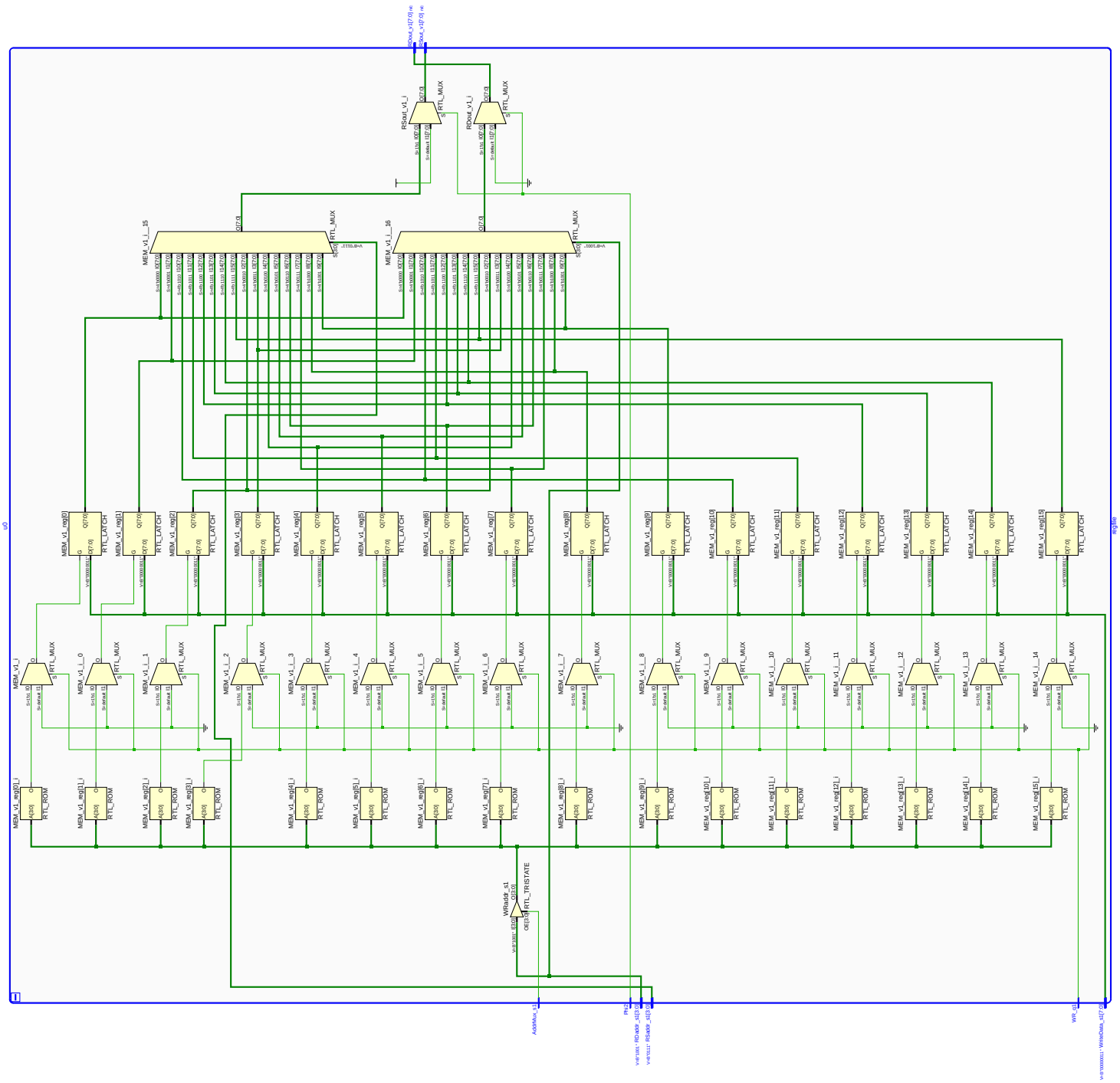
RegisterFile



RTL Analyze Schematic

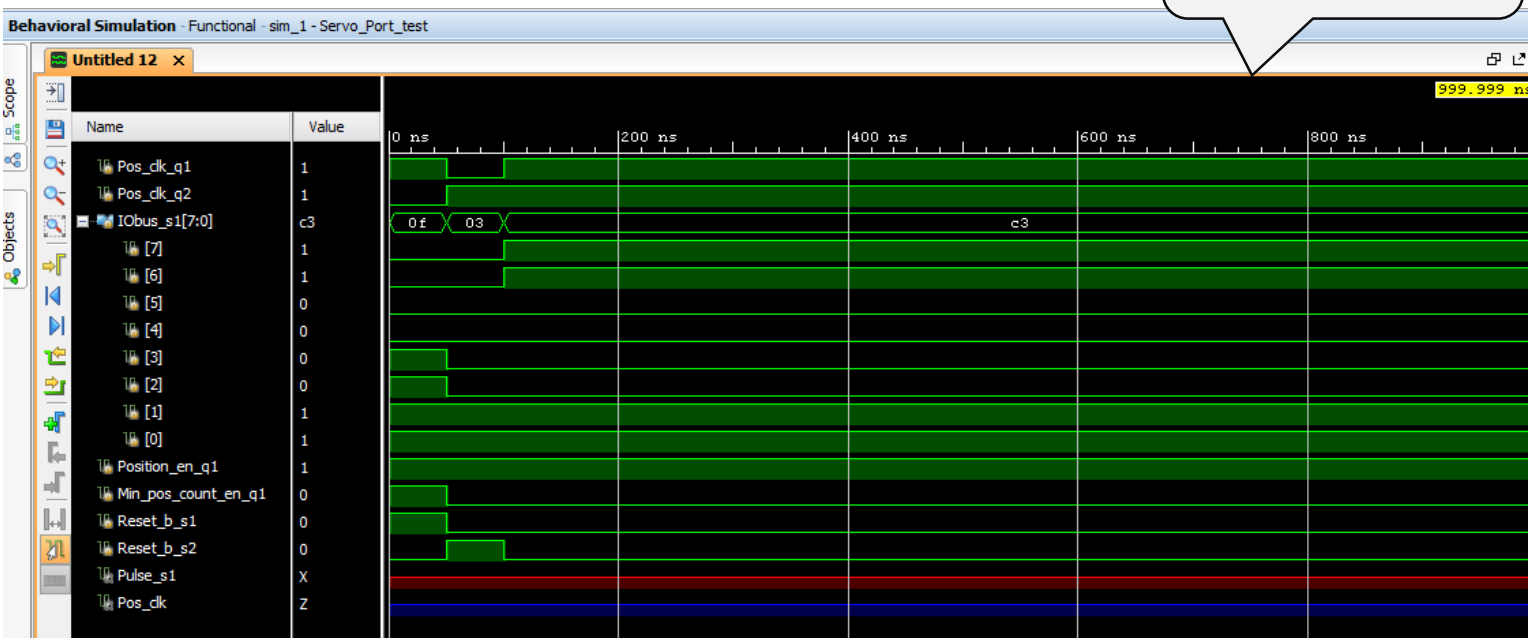


For view details, you can zoom in

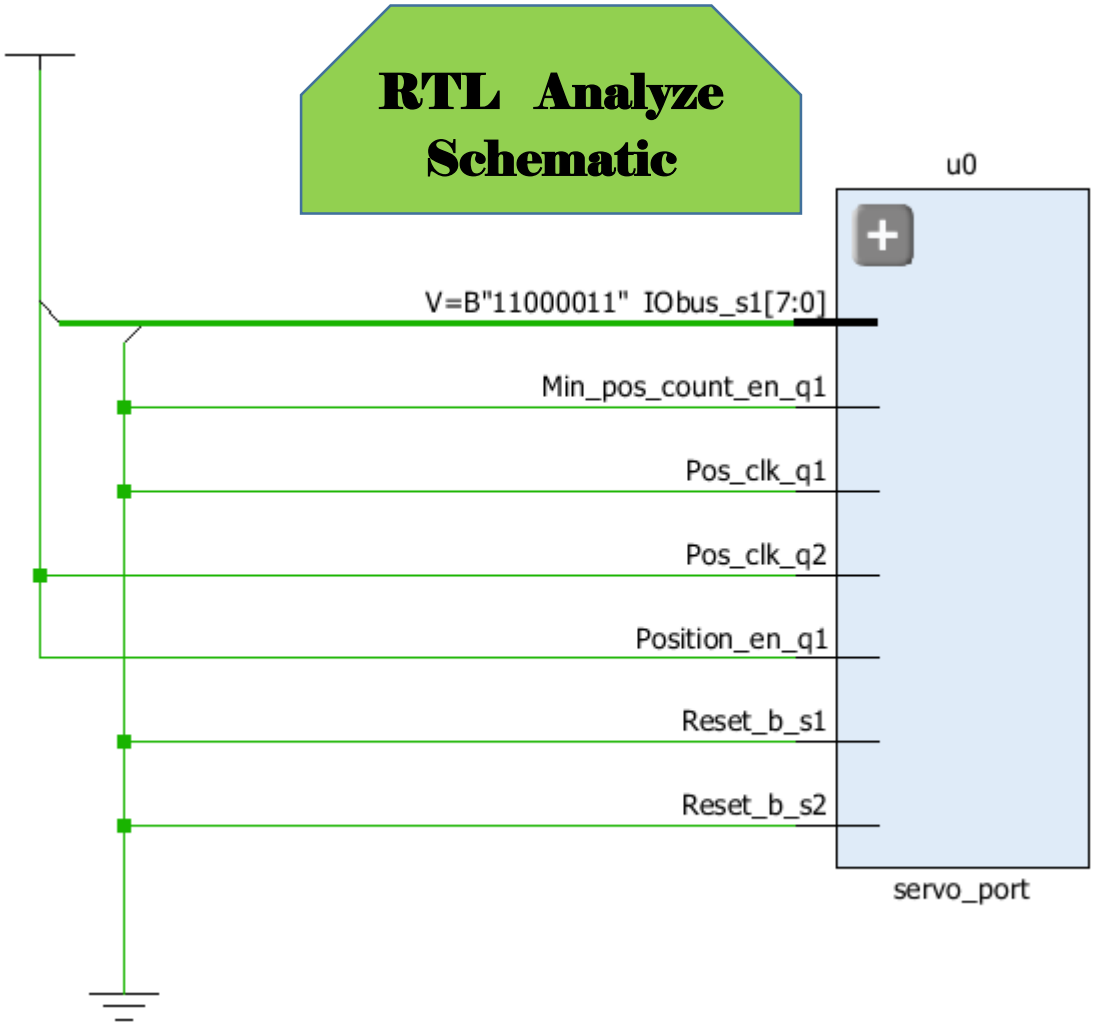


Simulation Waveform

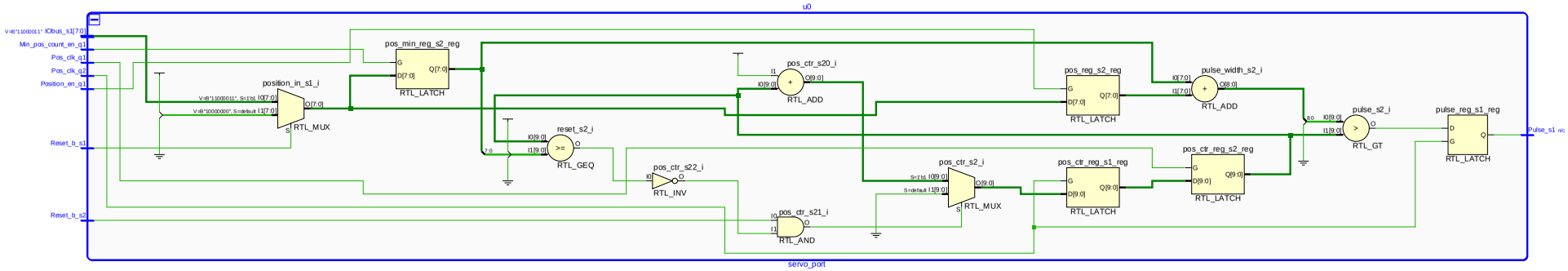
Servo Port



RTL Analyze Schematic



For view details, you can zoom in



Source Code (with Test Initial)

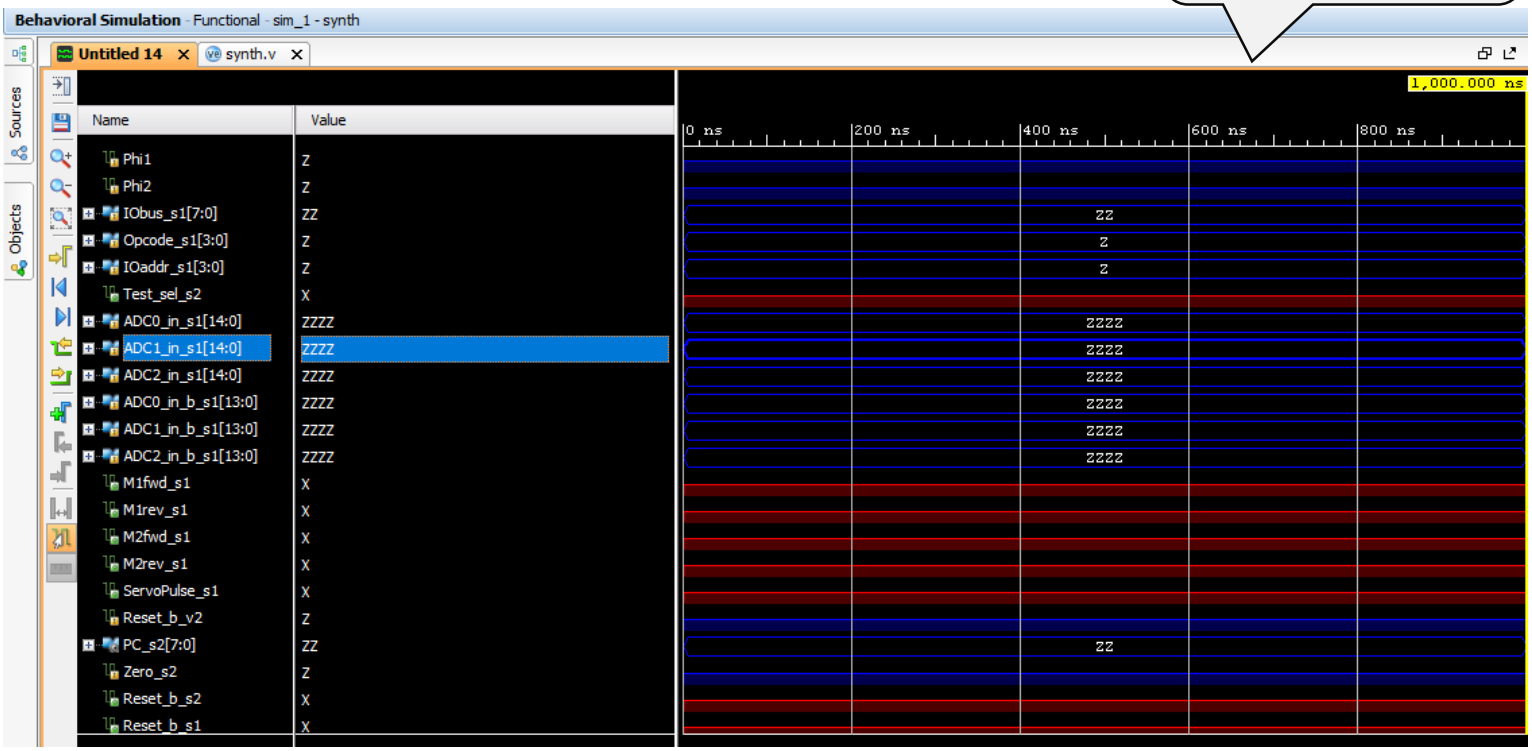
```
module synth(Phi1, Phi2, Reset_b_s1, Reset_b_s2, WR_q1, A0_s2, CE_b_q2,
             IObus_s1, Opcode_s1, IOaddr_s1, Zero_s2,
             IRL_enable_q2, IRH_enable_q2, IOaddr_sel_s1,
             PC_enable_q1, PC_enable_q2, Load_PC_s2,
             Operands_enable_q1, ALU_enable_q2, IMM_sel_s1,
             BusReg_enable_q1, Test_sel_s2, AddrMux_s1,
             ADC0_in_s1, ADC0_in_b_s1, ADC0_out_s1, ADC0_read_s1,
             ADC1_in_s1, ADC1_in_b_s1, ADC1_out_s1, ADC1_read_s1,
             ADC2_in_s1, ADC2_in_b_s1, ADC2_out_s1, ADC2_read_s1,
             PortB_data_s2, PortB_dir_s2, PortB_read_s1,
             WRbus_s1, ADClk_q2, M1fwd_s1, M1rev_s1, M2fwd_s1, M2rev_s1,
             ServoPulse_s1, ReadBusHigh_s1, OpA_sel_s1, Reset_b_v2);

  input  Phi1, Phi2;
  output         Reset_b_s1;
  output     Reset_b_s2;
  output         WR_q1,
                 A0_s2,
                 CE_b_q2;
  input  [7:0]   IObus_s1;
  input  [3:0]   Opcode_s1,
                 IOaddr_s1;
  input  Zero_s2;
  output     PortB_read_s1,
                 WRbus_s1;
  output         IRL_enable_q2,
                 IRH_enable_q2,
                 IOaddr_sel_s1,
                 PC_enable_q1,
                 PC_enable_q2,
                 Load_PC_s2,
                 Operands_enable_q1,
                 ALU_enable_q2,
                 IMM_sel_s1,
                 BusReg_enable_q1,
                 Test_sel_s2,
                 AddrMux_s1;
  input [14:0] ADC0_in_s1,
                 ADC1_in_s1,
                 ADC2_in_s1;
  input [13:0] ADC0_in_b_s1,
                 ADC1_in_b_s1,
                 ADC2_in_b_s1;
  output [3:0]   ADC0_out_s1,
                 ADC1_out_s1,
                 ADC2_out_s1;
  output         ADC0_read_s1,
                 ADC1_read_s1,
                 ADC2_read_s1;
```

```
  output [3:0]   PortB_data_s2,
                 PortB_dir_s2;
  output         M1fwd_s1,
                 M1rev_s1,
                 M2fwd_s1,
                 M2rev_s1,
                 ServoPulse_s1,
                 ADClk_q2,
                 ReadBusHigh_s1,
                 OpA_sel_s1;
input            Reset_b_v2;
// Local signals
  wire [7:0]     PC_s2;
  wire           Zero_s2,
                 Reset_b_s2,
                 Reset_b_s1,
                 WR_q1,
                 A0_s2,
                 CE_b_q2,
                 AddrMux_s1,
                 IRL_enable_q2,
                 IRH_enable_q2,
                 IOaddr_sel_s1,
                 PC_enable_q1,
                 PC_enable_q2,
                 Load_PC_s2,
                 Operands_enable_q1,
                 ALU_enable_q2,
                 IMM_sel_s1,
                 Bstatus_enable_q1,
                 Bdata_enable_q1,
                 M1speed_enable_q1,
                 M1status_enable_q1,
                 M2speed_enable_q1,
                 M2status_enable_q1,
                 Servo_enable_q1,
                 Smin_enable_q1,
                 Clock_enable_q1,
                 PortB_read_s1,
                 WRbus_s1,
                 ADC0_read_s1,
                 ADC1_read_s1,
                 ADC2_read_s1,
                 BusReg_enable_q1,
                 Test_sel_s1;
  wire [3:0]   PortB_data_s2,
                 PortB_dir_s2;
  wire   MotorClk_q1,
                 MotorClk_q2,
                 ServoClk_q1,
                 ServoClk_q2,
                 ADClk_q1,
                 ADClk_q2,
                 ReadBusHigh_s1,
                 OpA_sel_s1;
  wire [3:0]   ADC0_out_s1,
```

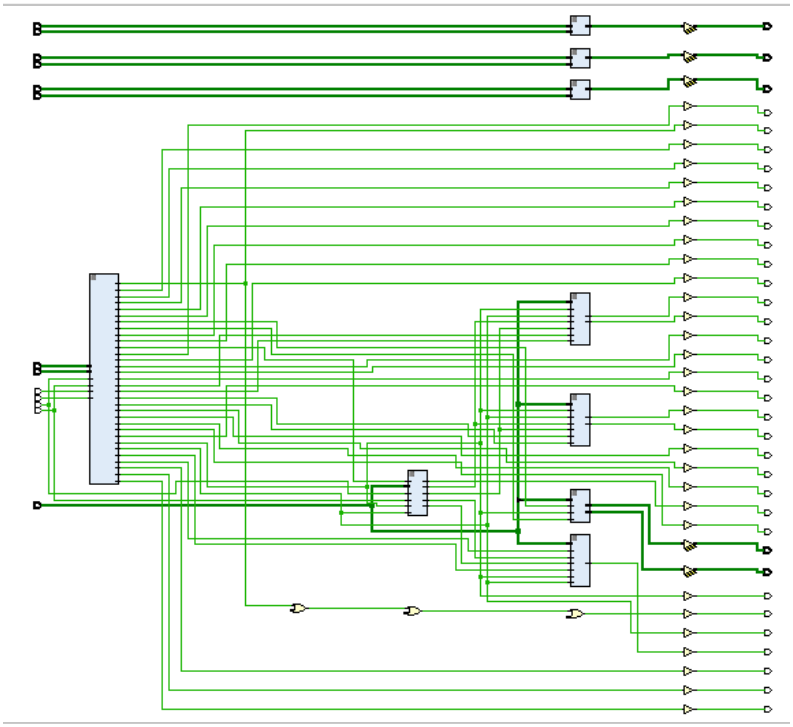
```
                    ADC1_out_s1,
                    ADC2_out_s1;
 // CONTROL MODULE
control control(Phi1, Phi2, Reset_b_s1, Reset_b_s2, Opcode_s1, IOaddr_s1,
                    Zero_s2, A0_s2, WR_q1, CE_b_q2, IRL_enable_q2, IRH_enable_q2,
              IOaddr_sel_s1, PC_enable_q1, PC_enable_q2, Load_PC_s2,
              Operands_enable_q1, ALU_enable_q2, IMM_sel_s1,
              Bstatus_enable_q1, Bdata_enable_q1, M1speed_enable_q1,
              M1status_enable_q1, M2speed_enable_q1, M2status_enable_q1,
              Servo_enable_q1, Smin_enable_q1,
              Clock_enable_q1, PortB_read_s1, WRbus_s1,
          ADC0_read_s1, ADC1_read_s1, ADC2_read_s1,
              BusReg_enable_q1, Test_sel_s2, AddrMux_s1, OpA_sel_s1,
              Reset_b_v2);
 // I/O MODULES
portb portb(IObus_s1[3:0], Bdata_enable_q1, Bstatus_enable_q1,
             PortB_data_s2, PortB_dir_s2, Reset_b_s1);
motor_port motor1(IObus_s1, M1speed_enable_q1, M1status_enable_q1,
                    MotorClk_q1, MotorClk_q2, Reset_b_s1, Reset_b_s2,
                    M1fwd_s1, M1rev_s1);
motor_port motor2(IObus_s1, M2speed_enable_q1, M2status_enable_q1,
                    MotorClk_q1, MotorClk_q2, Reset_b_s1, Reset_b_s2,
                    M2fwd_s1, M2rev_s1);
servo_port servo_port(IObus_s1, Servo_enable_q1, Smin_enable_q1,
                      Reset_b_s1, Reset_b_s2, ServoClk_q1, ServoClk_q2,
                      ServoPulse_s1);
clk_divider clk_divider(Phi1, Phi2, Reset_b_s1, Reset_b_s2, IObus_s1,
                        Clock_enable_q1, MotorClk_q1, MotorClk_q2,
                            ServoClk_q1, ServoClk_q2, ADClk_q1, ADClk_q2);
prio2bin ADC0(ADC0_in_s1, ADC0_in_b_s1, ADC0_out_s1);
prio2bin ADC1(ADC1_in_s1, ADC1_in_b_s1, ADC1_out_s1);
prio2bin ADC2(ADC2_in_s1, ADC2_in_b_s1, ADC2_out_s1);
assign ReadBusHigh_s1 = (ADC0_read_s1 || ADC1_read_s1 || ADC2_read_s1 ||
                             PortB_read_s1);
endmodule // synth
```

Simulation Waveform

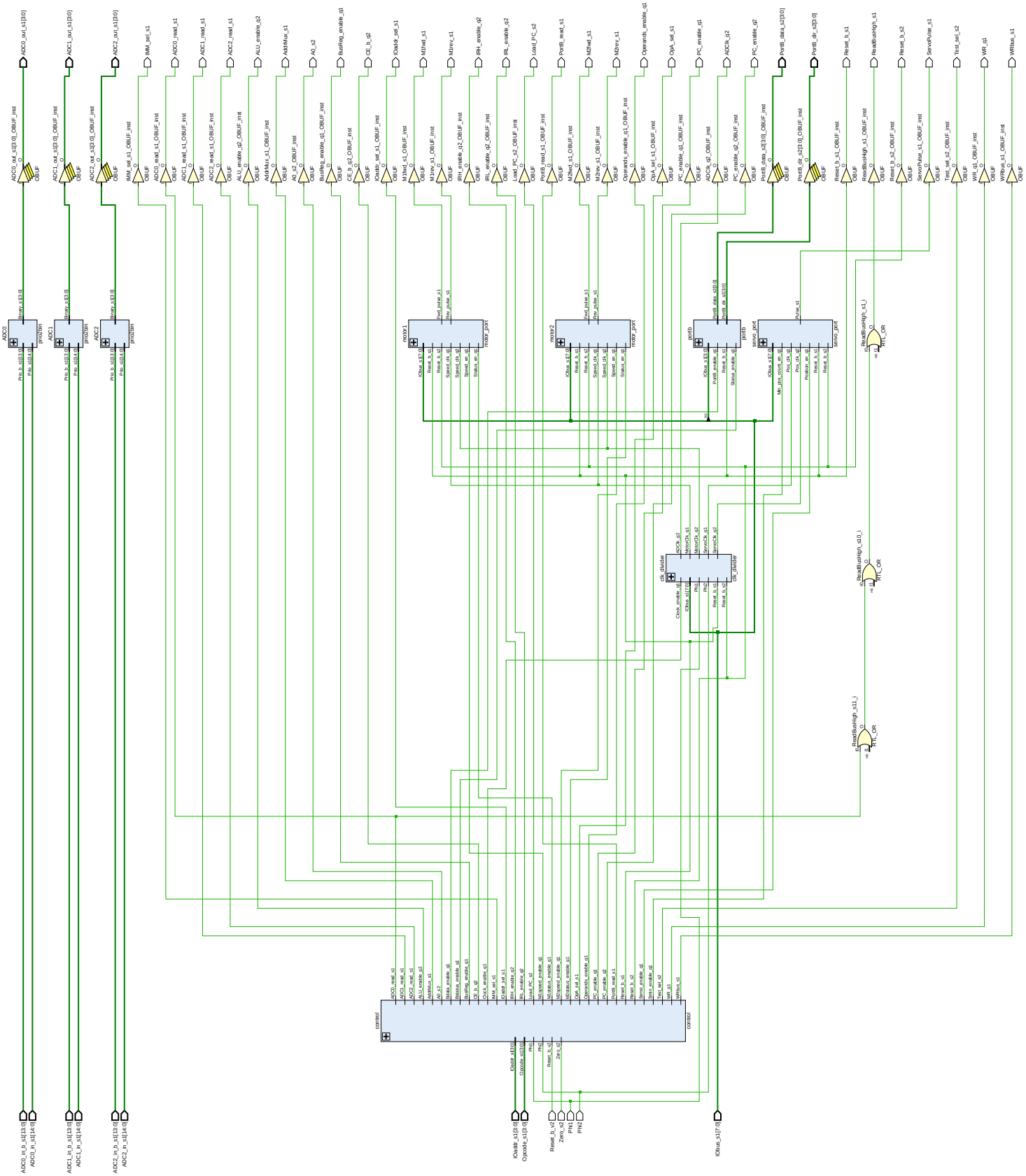
Synth



RTL Analyze Schematic



For view details, you can zoom in



SuCra

Source Code

```
module sucra(Phi1, Phi2, Reset_b_v2, Data_s2, Address_s2,
A0_s2, CE_b_q2,Bpins_in_s1, Bpins_out_s2, ADC0_in_s1,
ADC1_in_s1, ADC2_in_s1, M1fwd_s1, M1rev_s1, M2fwd_s1,
M2rev_s1,   ServoPulse_s1);
  input  Phi1, Phi2;                  // 2-Phase clocks
  input  Reset_b_v2;                  // Global reset
  input [7:0]      Data_s2;                    // Memory data bus
  output [7:0] Address_s2;                     // Memory address
bus
  output           CE_b_q2,                    // Memory chip
enable
                   A0_s2;                      // LSB of memory
address
  input [3:0]      Bpins_in_s1;                // Port B input
  output [3:0] Bpins_out_s2;                   // Port B output
  input [14:0]     ADC0_in_s1,
                   ADC1_in_s1,
                   ADC2_in_s1;
  output           M1fwd_s1,
                   M1rev_s1,
                   M2fwd_s1,
                   M2rev_s1,
                   ServoPulse_s1;
  wire [7:0]       Address_s2;
  wire             A0_s2,
                   CE_b_q2;
  wire [3:0]       Bpins_out_s2;
// Local signals
  wire [7:0]       IObus_s1;
  wire [7:0]       ALUresult_s1;
  reg  [7:0]       BusCapture_s2;
  wire [7:0]       PC_s2;
  wire [3:0]       Opcode_s1,
                   IOaddr_s1;
  wire             Zero_s2,
                   Reset_b_s2,
                   WR_q1,
                   IRL_enable_q2,
                   IRH_enable_q2,
                   IOaddr_sel_s1,
                   PC_enable_q1,
                   PC_enable_q2,
                   Load_PC_s2,
                   Operands_enable_q1,
                   ALU_enable_q2,
                   IMM_sel_s1,
                   PortB_read_s1,
                   WRbus_s1,
                   ADC0_read_s1,
                   ADC1_read_s1,
```

Test Bench

```
module Sucra_test;
reg       Phi1, Phi2;                  // 2-Phase clocks
 reg    Reset_b_v2;       // Global reset
 reg [7:0]     Data_s2;       // Memory data bus
 wire [7:0] Address_s2;       // Memory address bus
 wire    CE_b_q2,       // Memory chip enable
    A0_s2;          // LSB of memory address
 reg [3:0]     Bpins_in_s1;       // Port B input
 wire [3:0] Bpins_out_s2;       // Port B output
 reg [14:0]    ADC0_in_s1,
    ADC1_in_s1,
    ADC2_in_s1;
 wire     M1fwd_s1,
    M1rev_s1,
    M2fwd_s1,
    M2rev_s1,
    ServoPulse_s1;
sucra( .Phi1(Phi1) , .Phi2(Phi2) , .Reset_b_v2(Reset_b_v2) ,
.Data_s2(Data_s2) , .Address_s2(Address_s2) ,
.CE_b_q2(CE_b_q2),
.A0_s2(AO_s2) , .Bpins_in_s1(Bpins_in_s1) ,
.Bpins_out_s2(Bpins_out_s2) ,
.ADC0_in_s1(ADCO_in_s1) , .ADC1_in_s1(ADC1_in_s1) ,
.ADC2_in_s1(ADC2_in_s1) ,
.M1fwd_s1(M1fwd_s1) , .M1rev_s1(M1rev_s1) ,
.M2fwd_s1(M2fwd_s1) ,
.M2rev_s1(M2rev_s1), .ServoPulse_s1(ServoPulse_s1) );
initial begin
#0
Phi1=1'b0;
Phi2=1'b1;
 Reset_b_v2=1'b0;   // Global reset
  Data_s2=8'b00001111;  // Memory data bus
   Bpins_in_s1=4'b0001;   // Port B input
   ADC0_in_s1=4'b0011,
     ADC1_in_s1=1'b0;
     ADC2_in_s1=1'b1;
     #50
     Phi1=1'b0;
     Phi2=1'b1;
      Reset_b_v2=1'b0;   // Global reset
       Data_s2=8'b00000011;  // Memory data bus
        Bpins_in_s1=4'b0111;   // Port B input
        ADC0_in_s1=4'b0001,
           ADC1_in_s1=1'b1;
           ADC2_in_s1=1'b1;
           #50
           Phi1=1'b1;
           Phi2=1'b0;
            Reset_b_v2=1'b0;   // Global reset
```

```
                ADC2_read_s1,
                BusReg_enable_q1,
                Test_sel_s1,
                OpA_sel_s1;
  wire [3:0]   PortB_data_s2,
                PortB_dir_s2;
  wire   ADClk_q2,
                BusReadHigh_s1;
  wire [14:0]  ADC0_in_b_s1,
                ADC1_in_b_s1,
                ADC2_in_b_s1;
  wire [3:0]   ADC0_out_s1,
                ADC1_out_s1,
                ADC2_out_s1;
  wire [3:0]       RSaddr_s1,
                RDaddr_s1;
  wire [7:0]       RSout_v1,
                RDout_v1;
  wire        Reset_b_s1;
// DATAPATH MODULE
datapath datapath(Reset_b_s1, Reset_b_s2, Data_s2, PC_s2,
                ALUresult_s1, Opcode_s1, IOaddr_s1,
Zero_s2, IRL_enable_q2,
                IRH_enable_q2, PC_enable_q1,
PC_enable_q2, Load_PC_s2,
                IOaddr_sel_s1, Operands_enable_q1,
ALU_enable_q2,
                IMM_sel_s1, RSaddr_s1, RDaddr_s1,
RSout_v1, RDout_v1,
                OpA_sel_s1);
// REGFILE MODULE
regfile regfile(RSaddr_s1, RDaddr_s1, RSout_v1, RDout_v1,
                IObus_s1, WR_q1, AddrMux_s1);
synth synth(Phi1, Phi2, Reset_b_s1, Reset_b_s2, WR_q1,
A0_s2, CE_b_q2,
            IObus_s1, Opcode_s1, IOaddr_s1, Zero_s2,
            IRL_enable_q2, IRH_enable_q2, IOaddr_sel_s1,
            PC_enable_q1, PC_enable_q2, Load_PC_s2,
            Operands_enable_q1, ALU_enable_q2,
IMM_sel_s1,   BusReg_enable_q1, Test_sel_s2,
AddrMux_s1,         ADC0_in_s1, ADC0_in_b_s1[14:1],
ADC0_out_s1, ADC0_read_s1,    ADC1_in_s1,
ADC1_in_b_s1[14:1], ADC1_out_s1, ADC1_read_s1,
ADC2_in_s1, ADC2_in_b_s1[14:1], ADC2_out_s1,
ADC2_read_s1,    PortB_data_s2, PortB_dir_s2,
PortB_read_s1,   WRbus_s1, ADClk_q2, M1fwd_s1,
M1rev_s1, M2fwd_s1, M2rev_s1,
            ServoPulse_s1, ReadBusHigh_s1, OpA_sel_s1,
Reset_b_v2);
  assign          ADC0_in_b_s1 = ~ADC0_in_s1;
  assign          ADC1_in_b_s1 = ~ADC1_in_s1;
  assign          ADC2_in_b_s1 = ~ADC2_in_s1;
// I/O Bus Capture Latch (for testing purposes)
always @(BusReg_enable_q1 or IObus_s1)
 if (BusReg_enable_q1)
  BusCapture_s2 = IObus_s1;
```

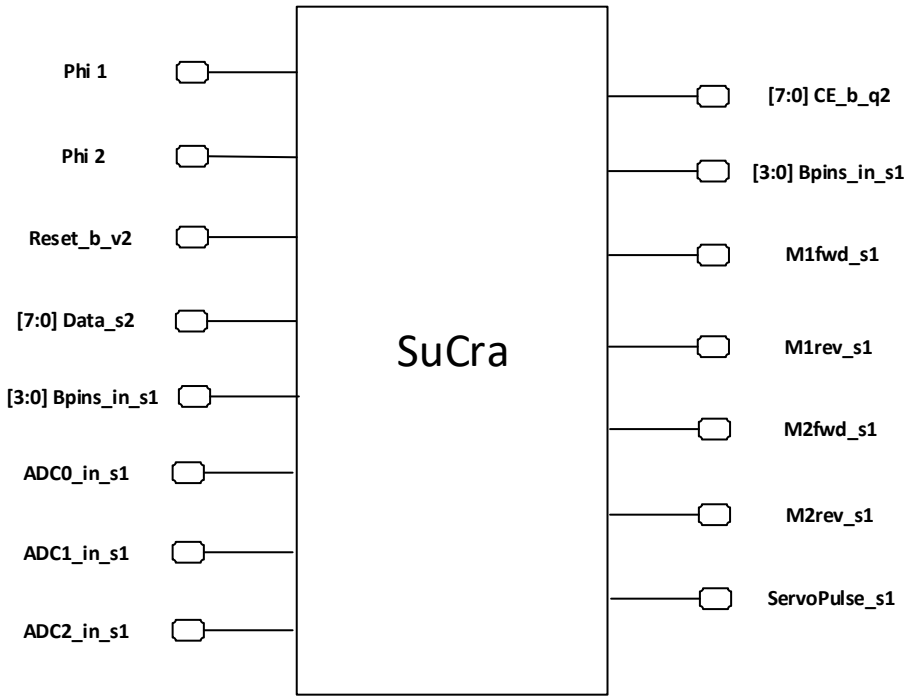
```
          Data_s2=8'b00000011;  // Memory data bus
           Bpins_in_s1=4'b0001;   // Port B input
           ADC0_in_s1=4'b1111,
              ADC1_in_s1=1'b1;
              ADC2_in_s1=1'b0;
end
endmodule
```

ادامه سورس کد Sucra :

```
assign   Bpins_out_s2[3] = (PortB_dir_s2[3]) ?
PortB_data_s2[3] : 1'bZ;
// Tristates for IO Bus
  assign IObus_s1 = (PortB_read_s1) ? {4'b0,Bpins_in_s1} :
8'bZ;
  assign IObus_s1 = (WRbus_s1) ? ALUresult_s1 : 8'bZ;
  assign IObus_s1 = (ADC0_read_s1) ? {4'b0,ADC0_out_s1} :
8'bZ;
  assign IObus_s1 = (ADC1_read_s1) ? {4'b0,ADC1_out_s1} :
8'bZ;
  assign IObus_s1 = (ADC2_read_s1) ? {4'b0,ADC2_out_s1} :
8'bZ;
endmodule // sucra
```

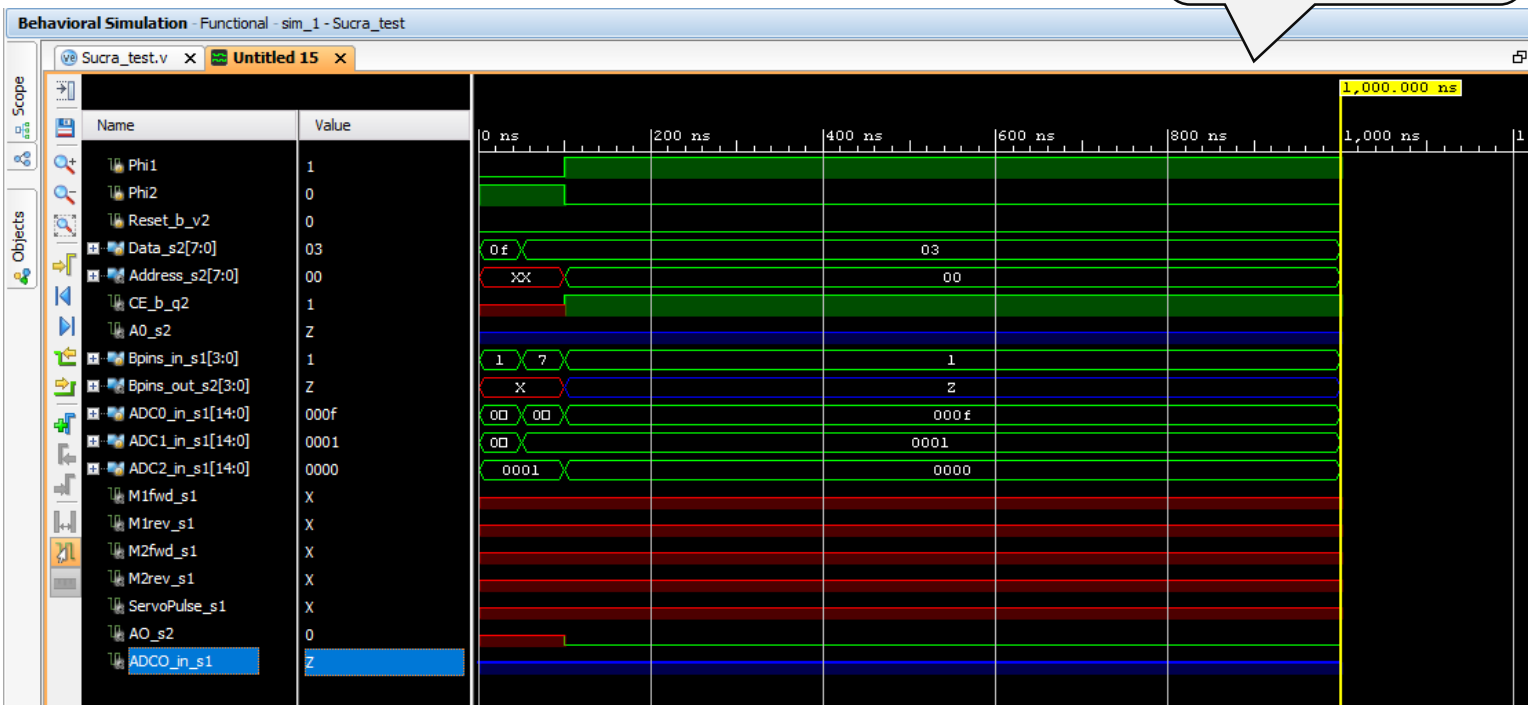
```
assign   Address_s2 = (Test_sel_s2) ? BusCapture_s2 : PC_s2;
// Glue logic
// Tristates for Port B output
  assign          Bpins_out_s2[0] = (PortB_dir_s2[0]) ?
PortB_data_s2[0] : 1'bZ;
  assign          Bpins_out_s2[1] = (PortB_dir_s2[1]) ?
PortB_data_s2[1] : 1'bZ;
  assign          Bpins_out_s2[2] = (PortB_dir_s2[2]) ?
PortB_data_s2[2] : 1'bZ;
```

Sucra Module Diagram

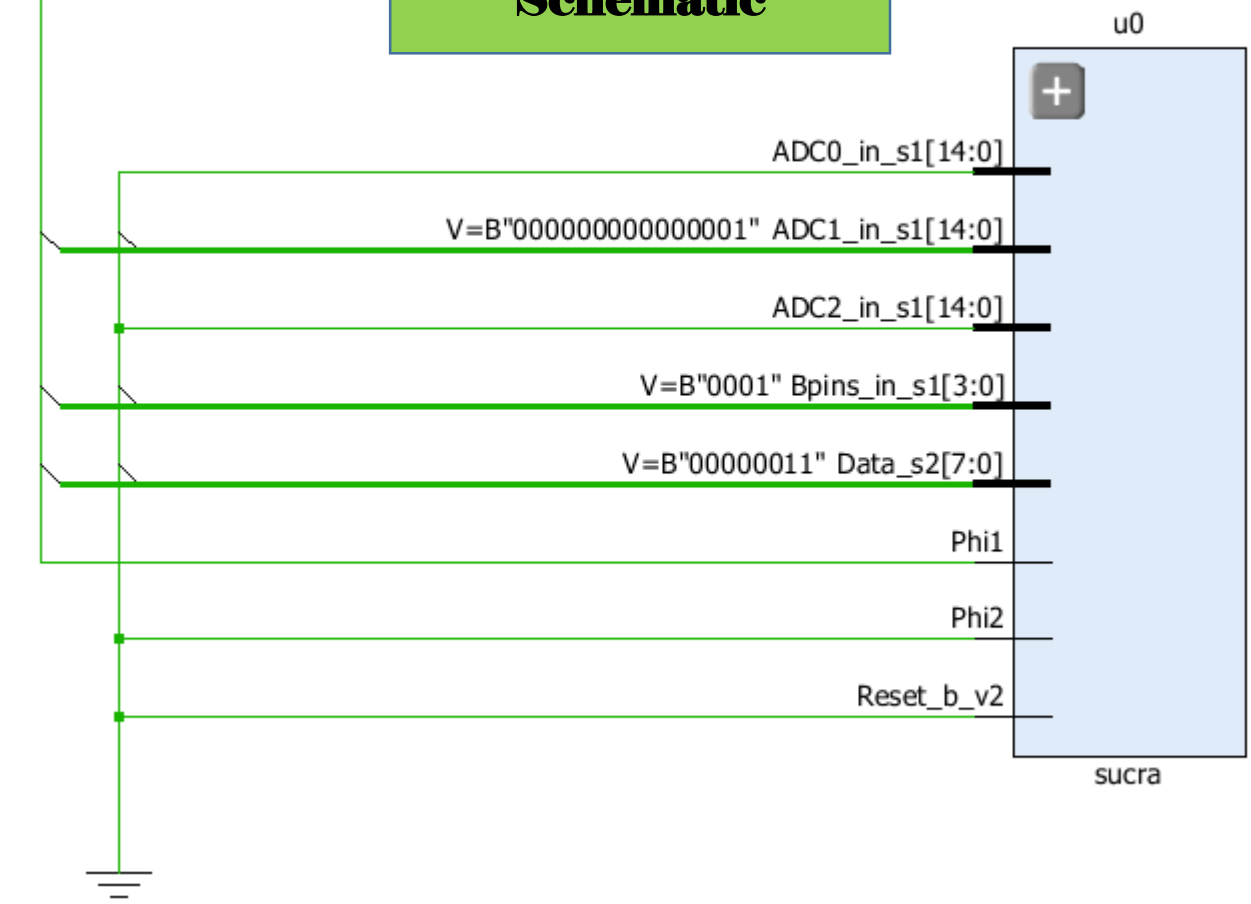


Simulation Waveform

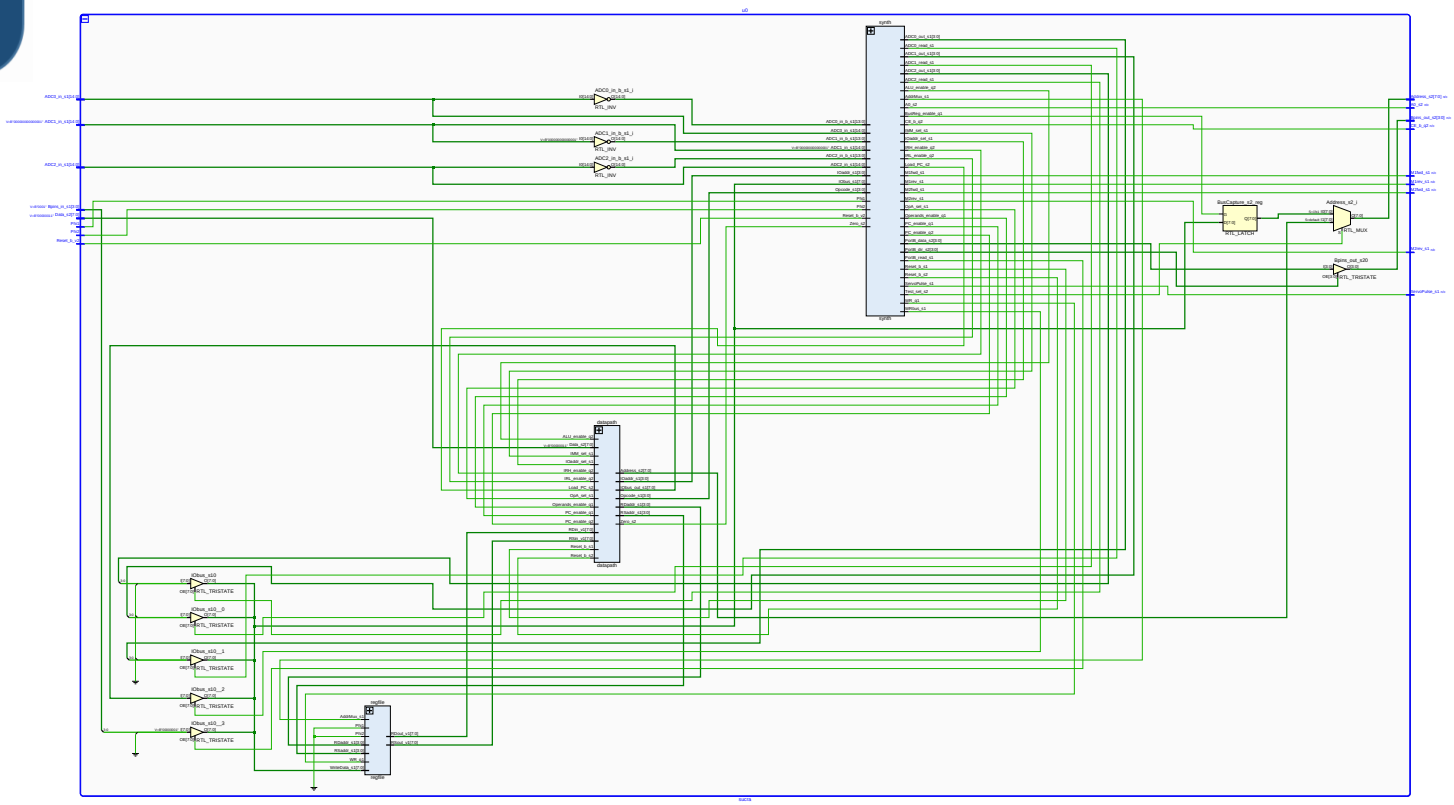
Sucra



RTL Analyze Schematic



For view details, you can zoom in

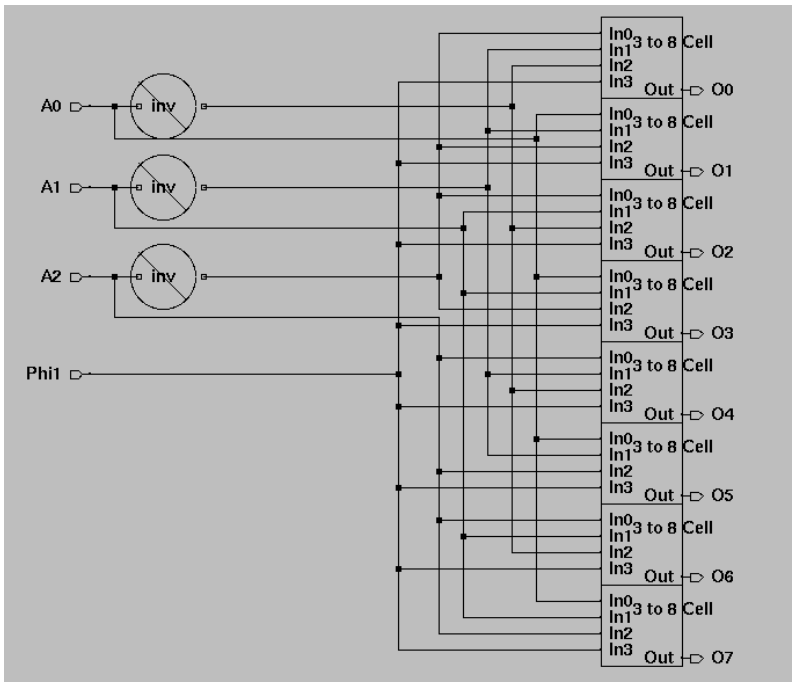


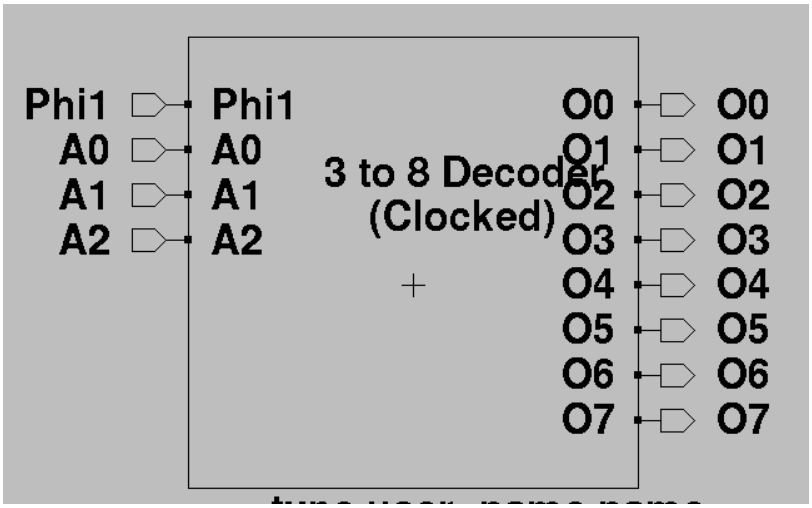
شبیه سازی فایل های sue.

در این قسمت باتوجه به فایل های sue. موجود در فولدر پروژه SuCra ، اقدام به شبیه سازی و تولید کد وریلاگ مورد نظر از آنها می کنیم .

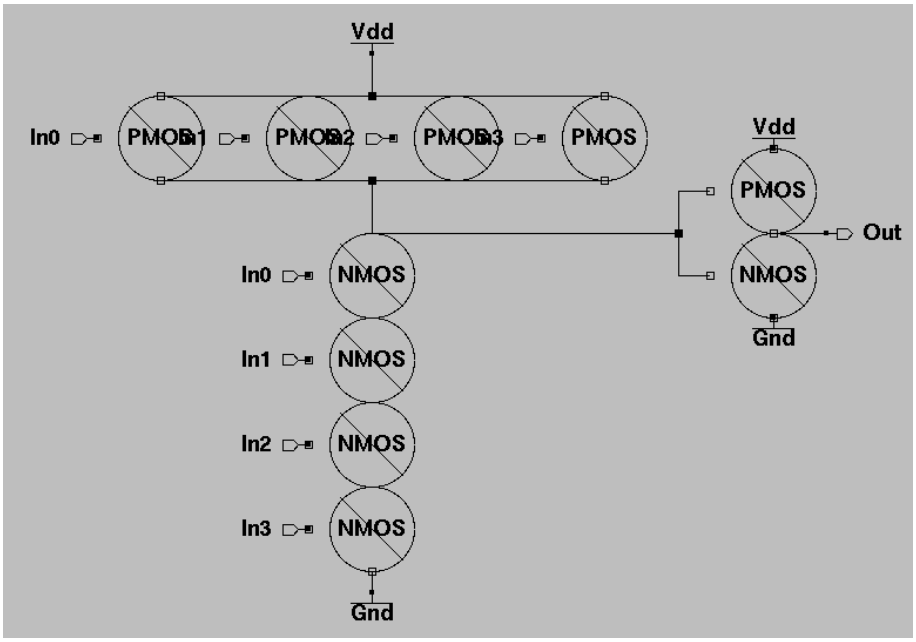
(ترتیب قرار گیری فایل ها براساس حروف الفبا می باشد .)

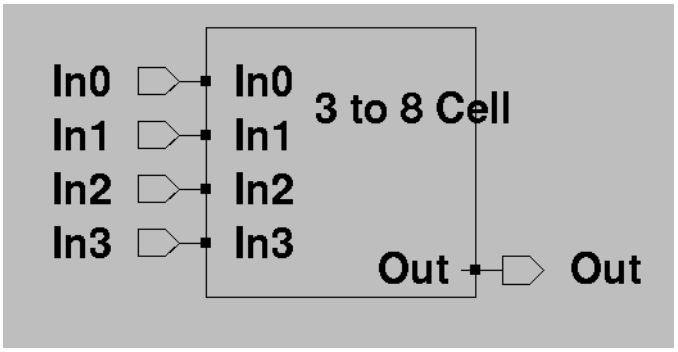
3 to 8 Decoder



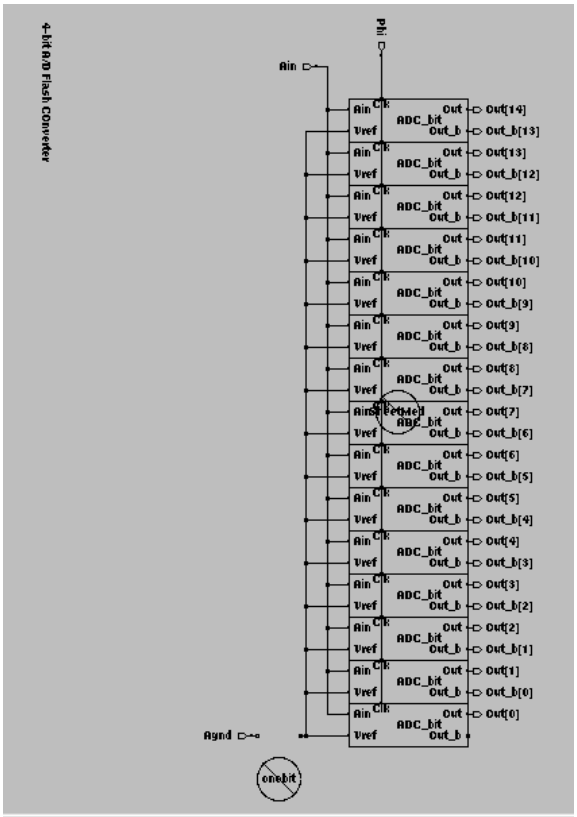


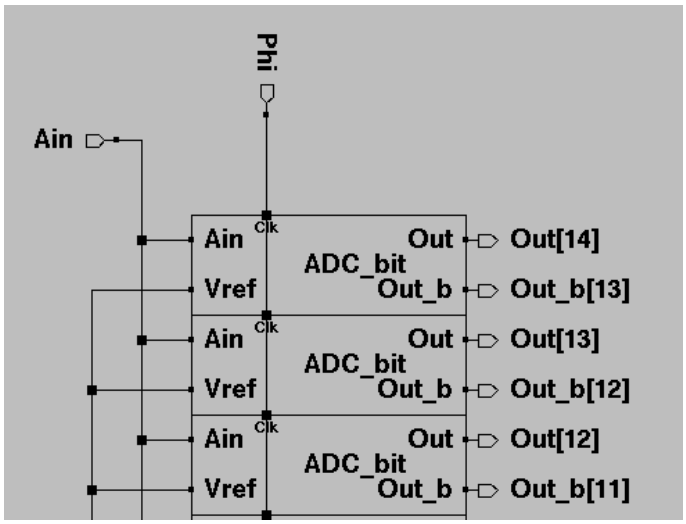
3 to 8 Decoder cell

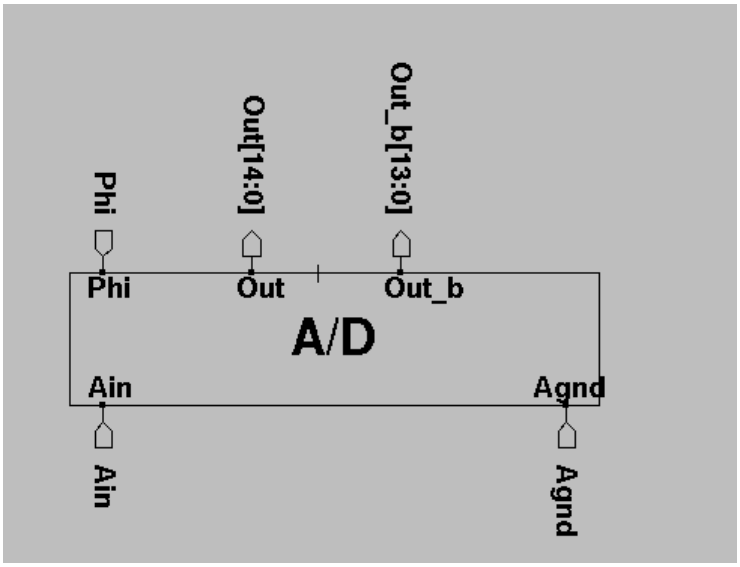




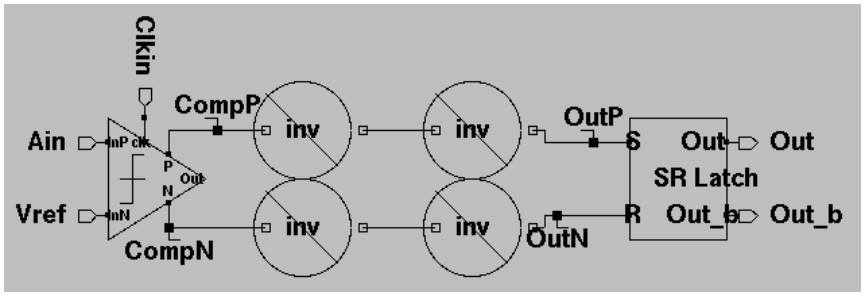
ADC

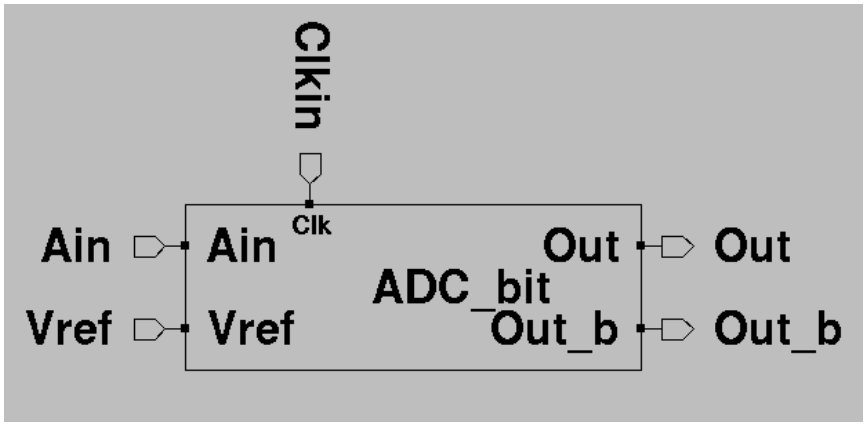



Phi
Ain
Clk
Ain
Vref
Out
ADC_bit
Out_b
Out[14]
Out_b[13]
Out[13]
Out_b[12]
Out[12]
Out_b[11]

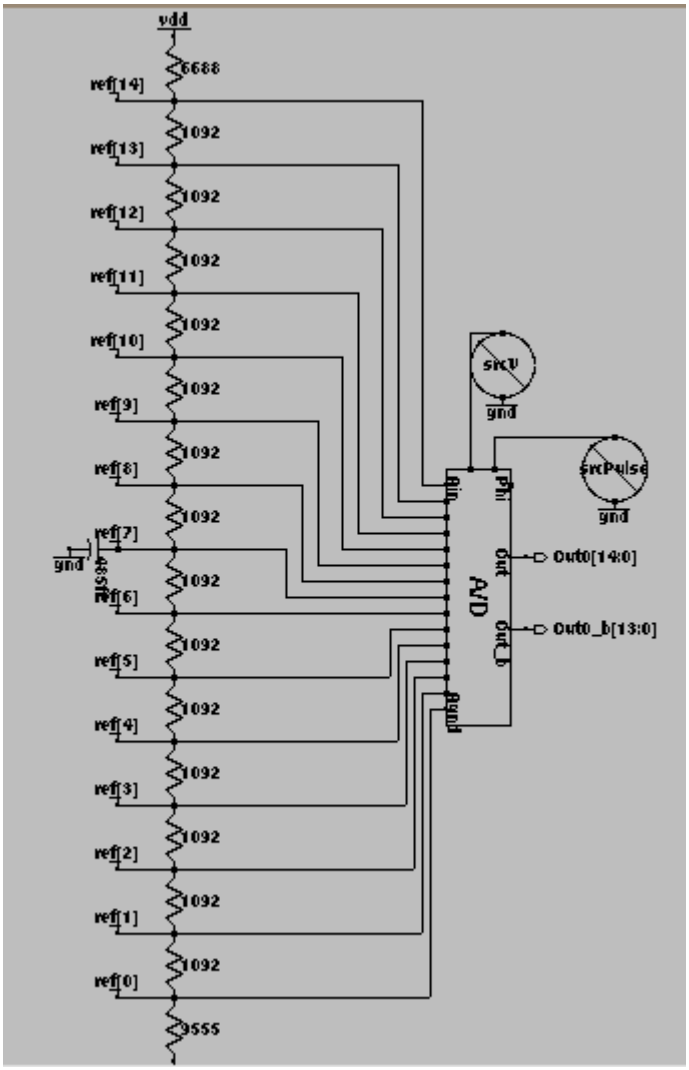

Phi
Out[14:0]
Out_b[13:0]
Phi
Out
Out_b
A/D
Ain
Agnd
Ain
Agnd

ADC Slice

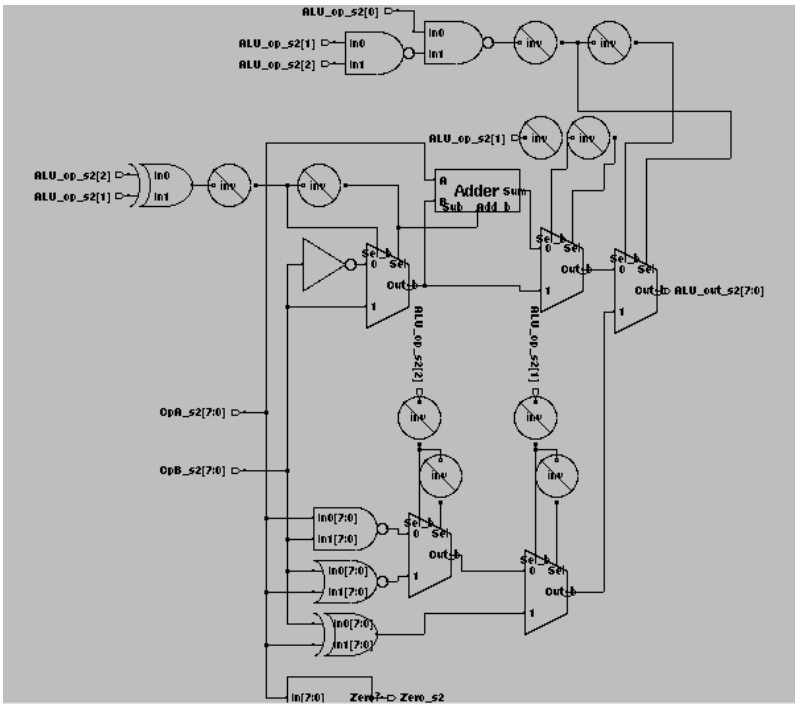

Clkin
Ain
Vref
CompP
CompN
inv
inv
inv
inv
OutP
OutN
S
R
Out
SR Latch
Out_b
Out
Out_b

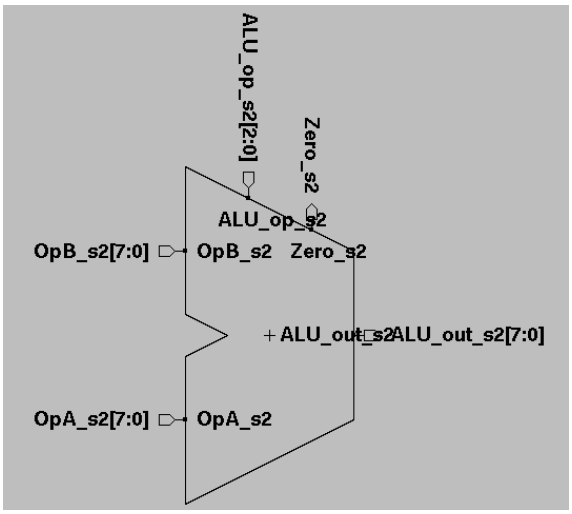

Clkin
Clk
Ain
Vref
Ain
Vref
Out
ADC_bit
Out_b
Out
Out_b

ADC test

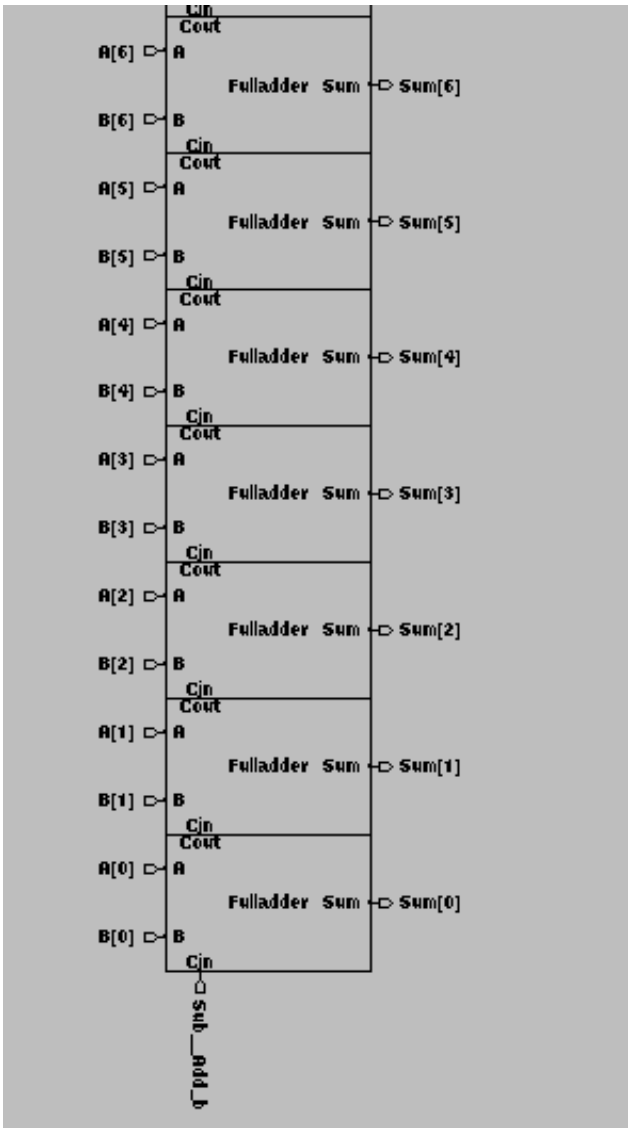

vdd
6688
ref[14]
1092
ref[13]
ref[12]
ref[11]
ref[10]
ref[9]
ref[8]
ref[7]
gnd
ref[6]
ref[5]
ref[4]
ref[3]
ref[2]
ref[1]
ref[0]
9555
srcV
srcPulse
Vin
Phi
Out
A/D
Out_b
Agnd
Out0[14:0]
Out0_b[13:0]

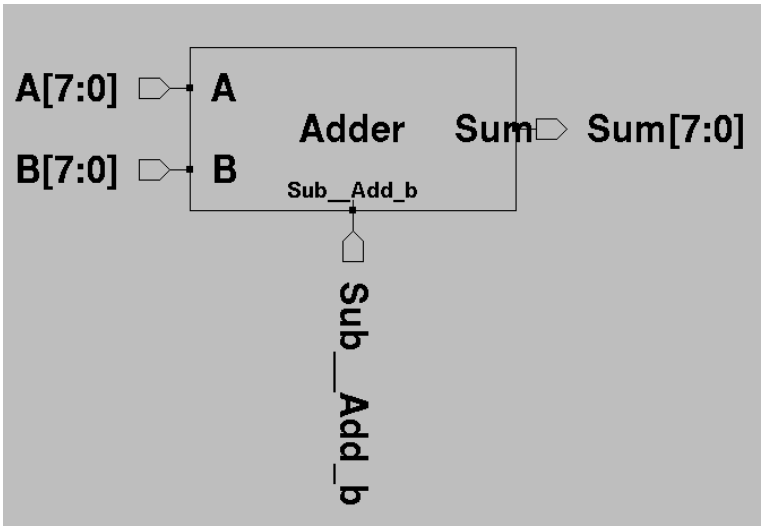
ALU


ALU_op_s2[0]
ALU_op_s2[1]
ALU_op_s2[2]
in0
in1
inv
Adder
Sum
Sub
Add_b
Sel_b
Out_b
ALU_out_s2[7:0]
OpA_s2[7:0]
OpB_s2[7:0]
in0[7:0]
in1[7:0]
Zero_s2

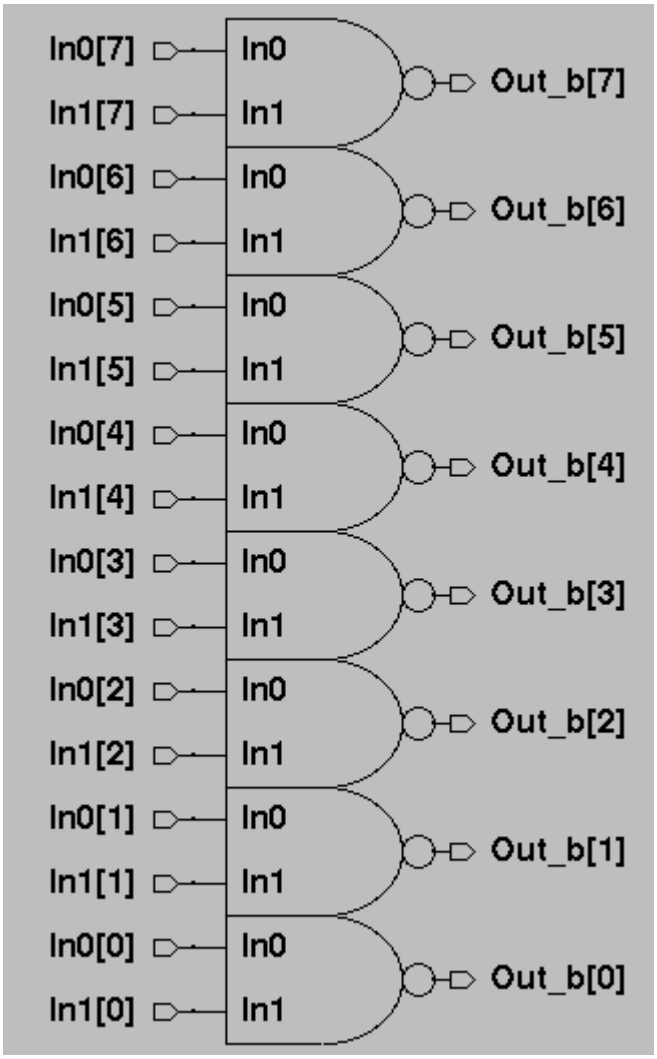

ALU_op_s2[2:0]
Zero_s2
ALU_op_s2
OpB_s2[7:0]
OpB_s2
Zero_s2
+
ALU_out_s2
ALU_out_s2[7:0]
OpA_s2[7:0]
OpA_s2

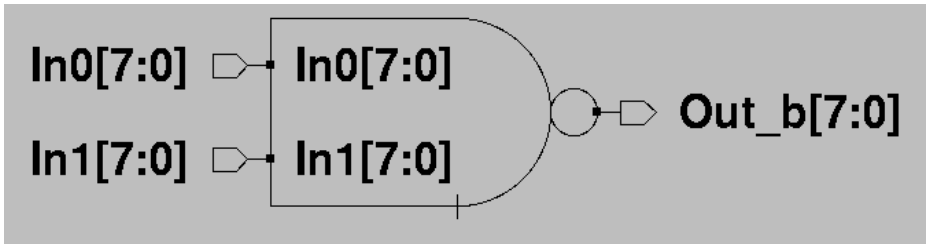
ALU Adder


Cin
Cout
A[6]
A
Fulladder
Sum
Sum[6]
B[6]
B
Cin
Cout
A[5]
A
Fulladder
Sum
Sum[5]
B[5]
B
Cin
Cout
A[4]
A
Fulladder
Sum
Sum[4]
B[4]
B
Cin
Cout
A[3]
A
Fulladder
Sum
Sum[3]
B[3]
B
Cin
Cout
A[2]
A
Fulladder
Sum
Sum[2]
B[2]
B
Cin
Cout
A[1]
A
Fulladder
Sum
Sum[1]
B[1]
B
Cin
Cout
A[0]
A
Fulladder
Sum
Sum[0]
B[0]
B
Cin
Sub_Add_b

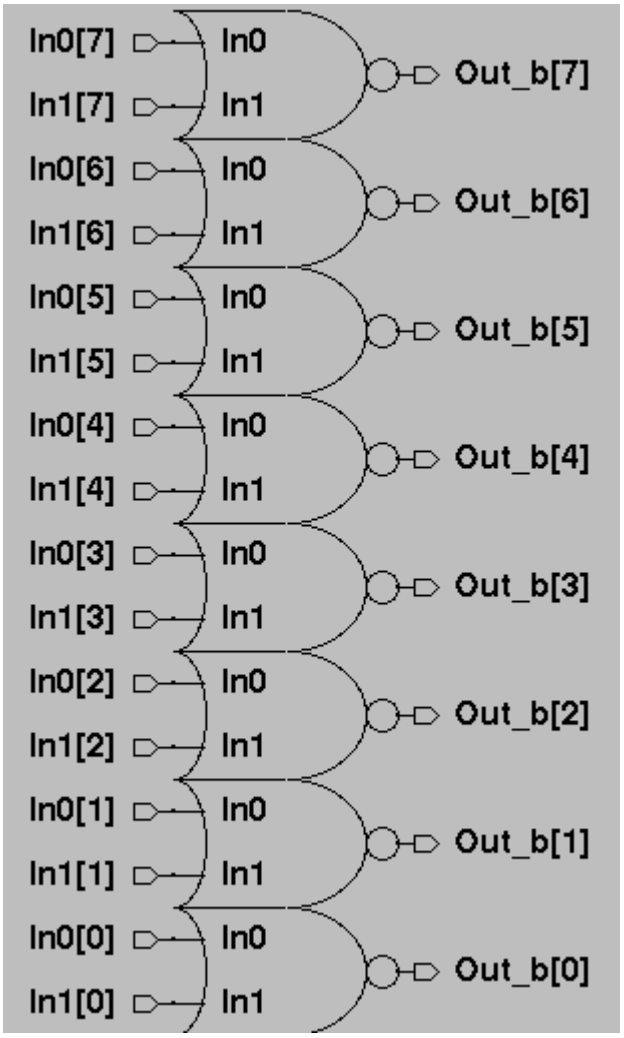

A[7:0]
A
Adder
Sum
Sum[7:0]
B[7:0]
B
Sub__Add_b
Sub__Add_b

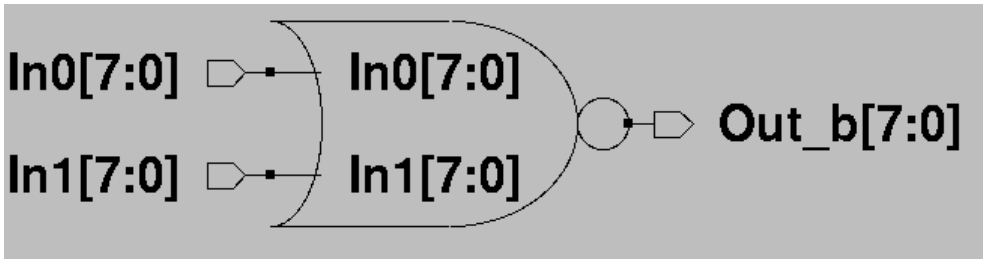
ALU nand


In0[7]
In1[7]
In0
In1
Out_b[7]
In0[6]
In1[6]
In0
In1
Out_b[6]
In0[5]
In1[5]
In0
In1
Out_b[5]
In0[4]
In1[4]
In0
In1
Out_b[4]
In0[3]
In1[3]
In0
In1
Out_b[3]
In0[2]
In1[2]
In0
In1
Out_b[2]
In0[1]
In1[1]
In0
In1
Out_b[1]
In0[0]
In1[0]
In0
In1
Out_b[0]

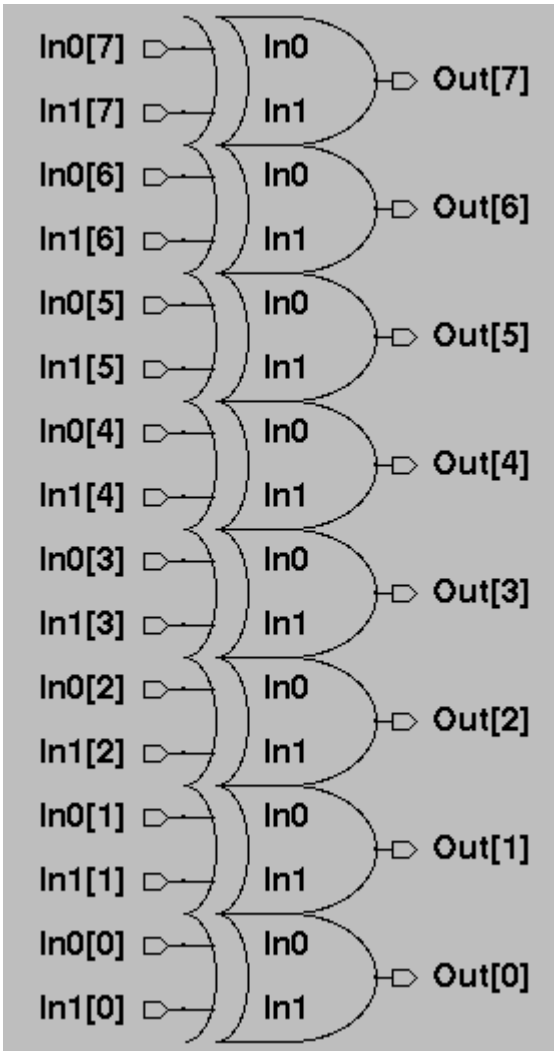

In0[7:0]
In1[7:0]
In0[7:0]
In1[7:0]
Out_b[7:0]

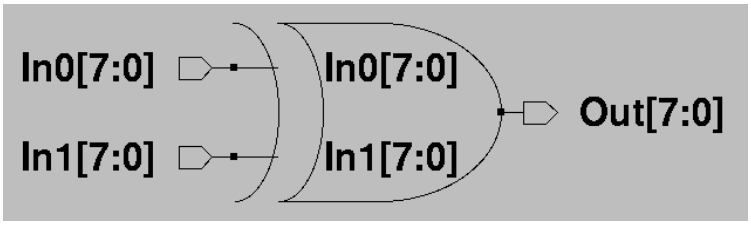
ALU nor


In0[7]
In1[7]
In0
In1
Out_b[7]
In0[6]
In1[6]
In0
In1
Out_b[6]
In0[5]
In1[5]
In0
In1
Out_b[5]
In0[4]
In1[4]
In0
In1
Out_b[4]
In0[3]
In1[3]
In0
In1
Out_b[3]
In0[2]
In1[2]
In0
In1
Out_b[2]
In0[1]
In1[1]
In0
In1
Out_b[1]
In0[0]
In1[0]
In0
In1
Out_b[0]

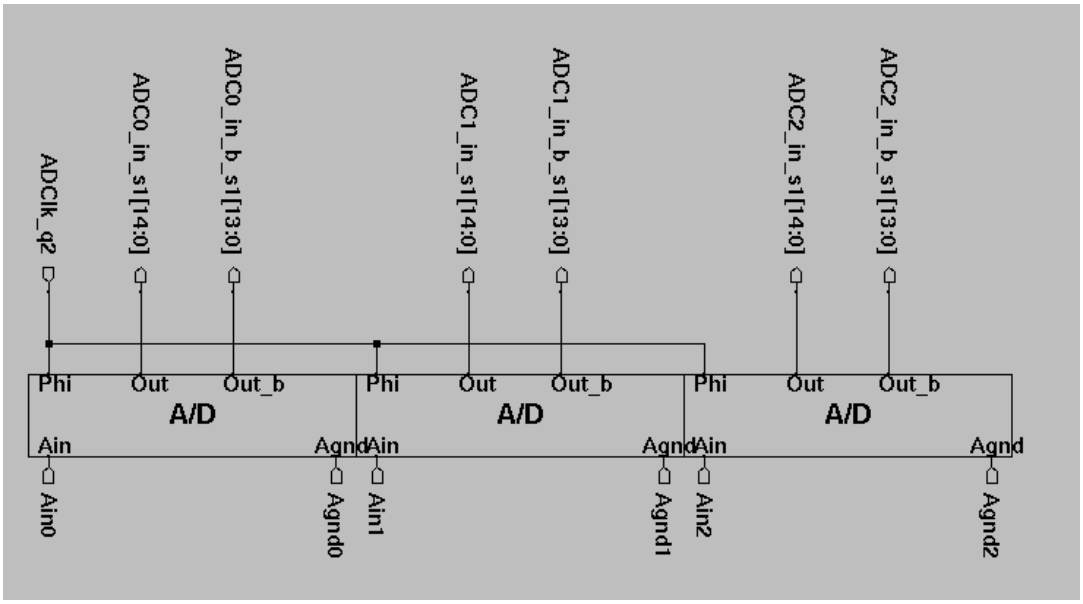

In0[7:0]
In1[7:0]
In0[7:0]
In1[7:0]
Out_b[7:0]

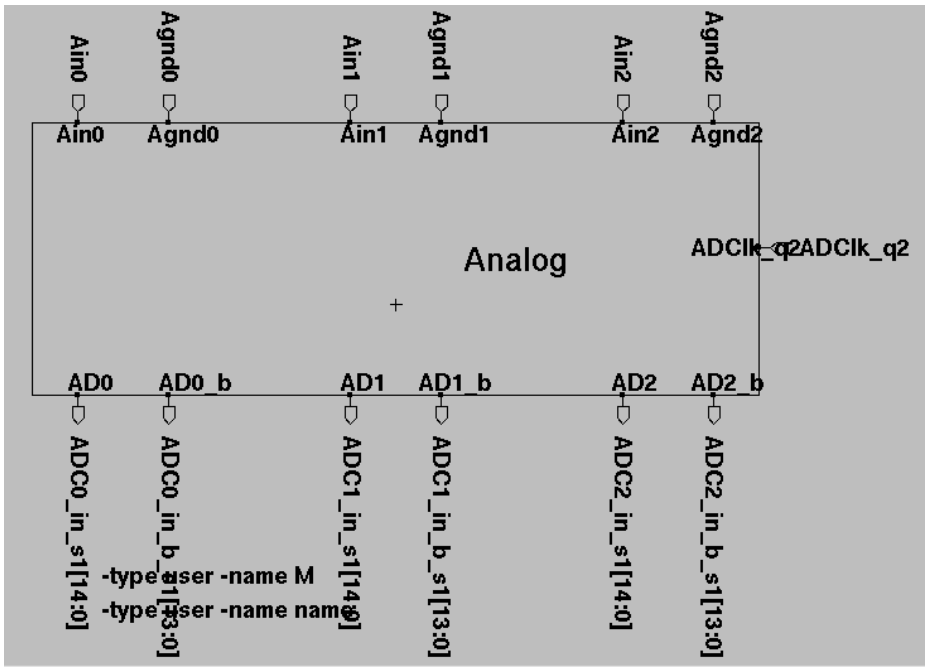
ALU xor

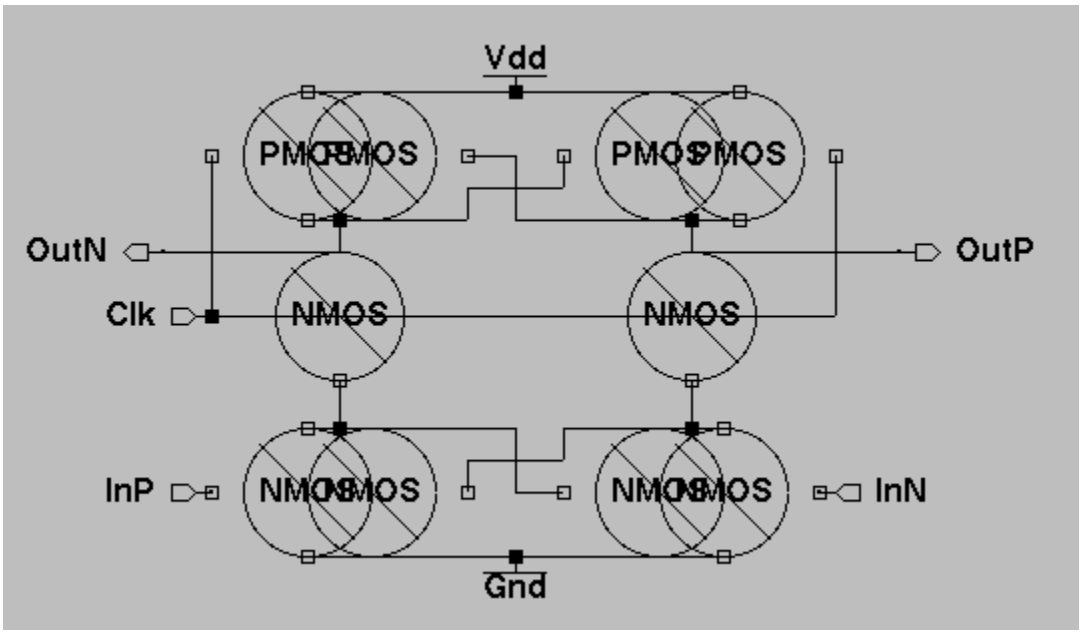




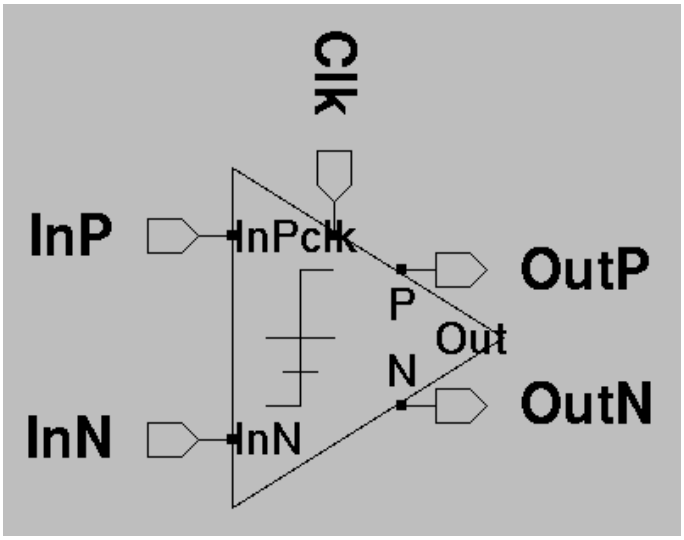
Analog



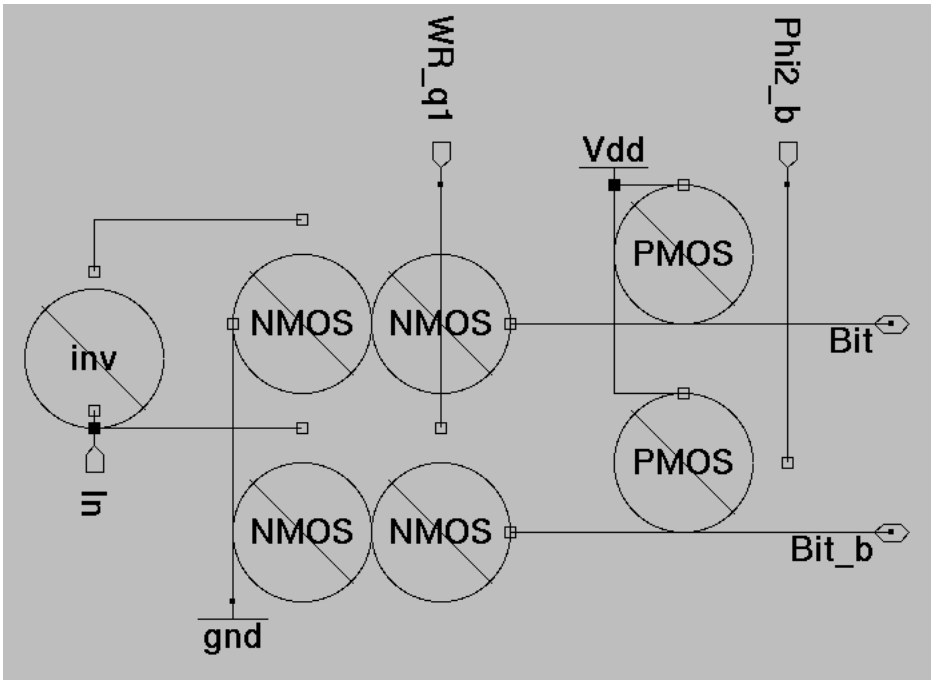


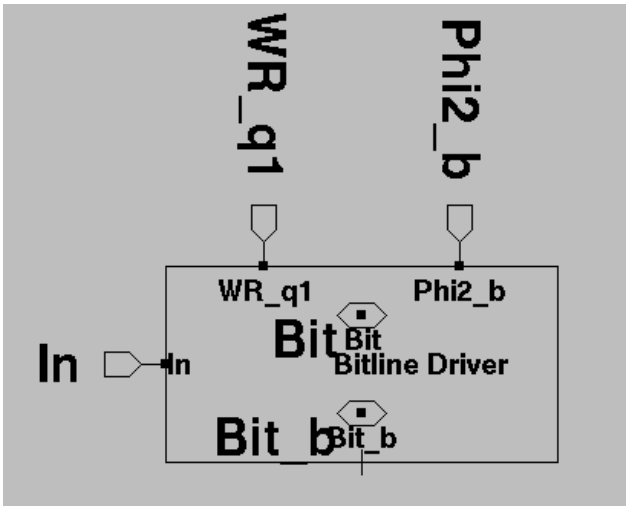


Analog Comparator

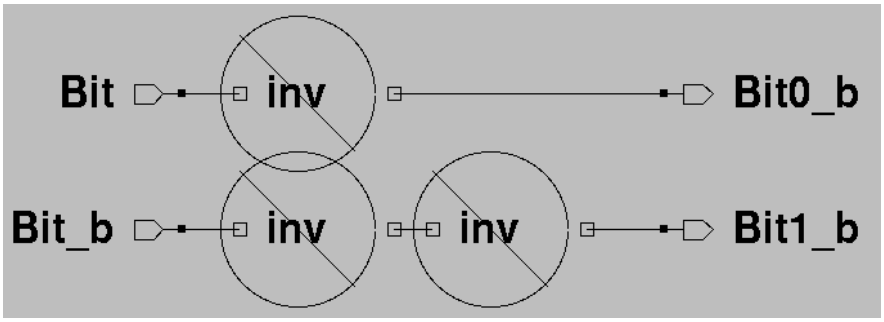


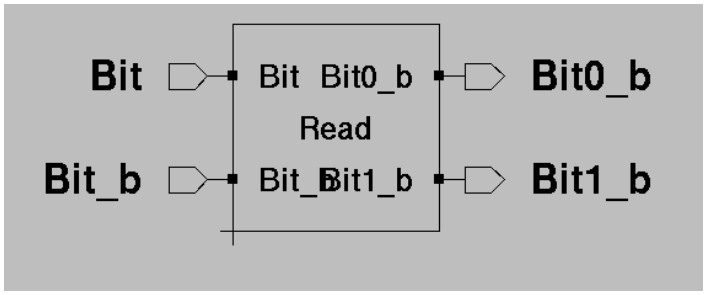
Bitline driver



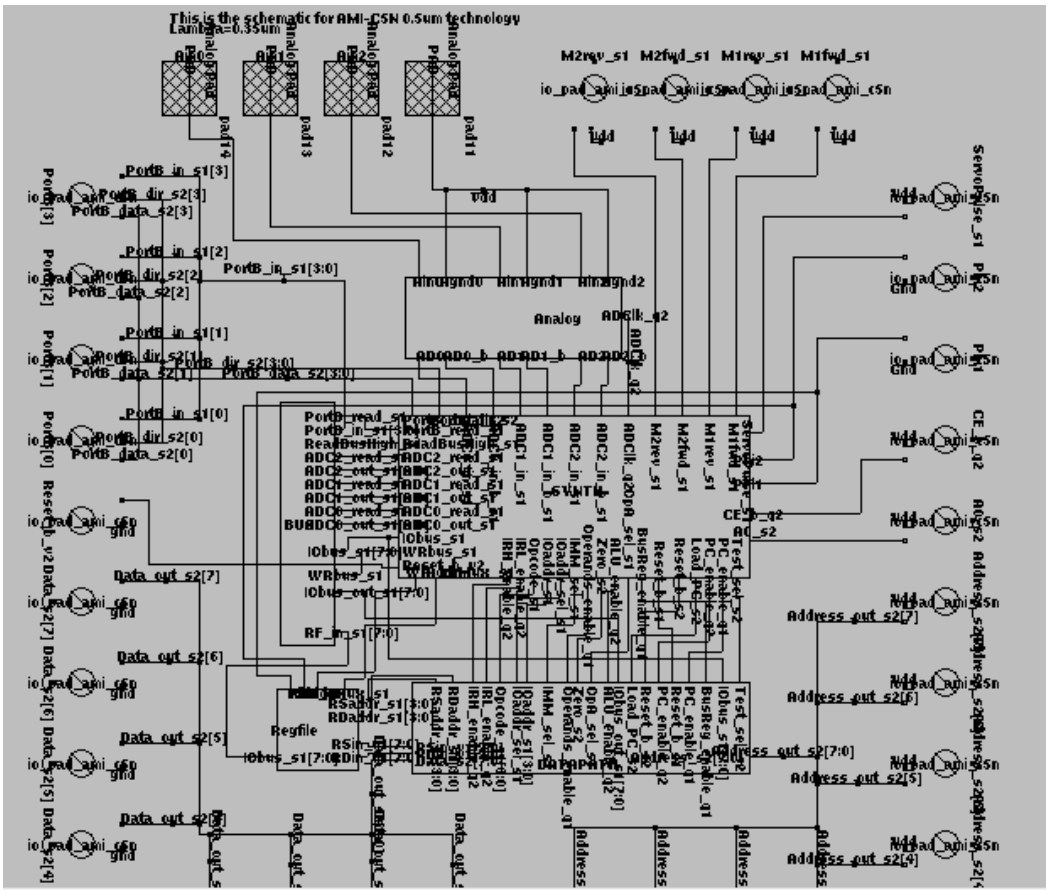


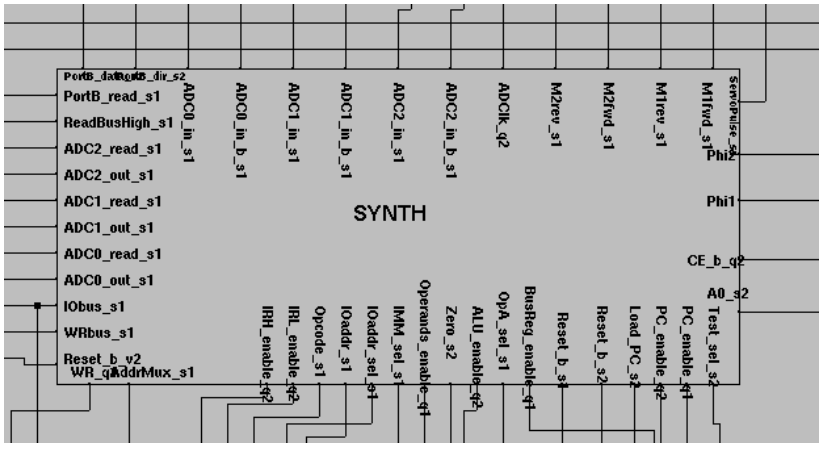
Bitline Reader

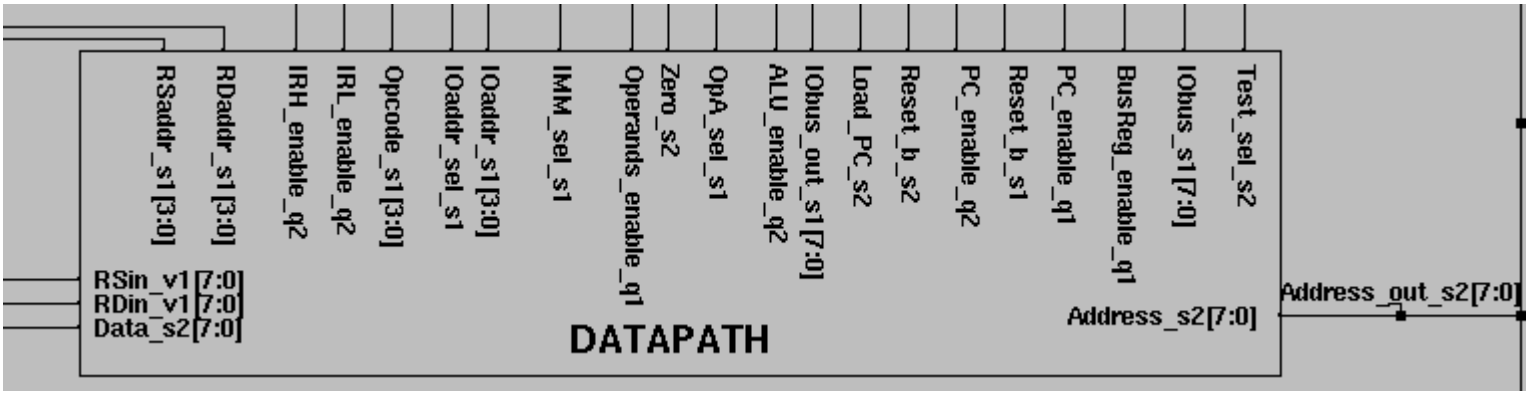


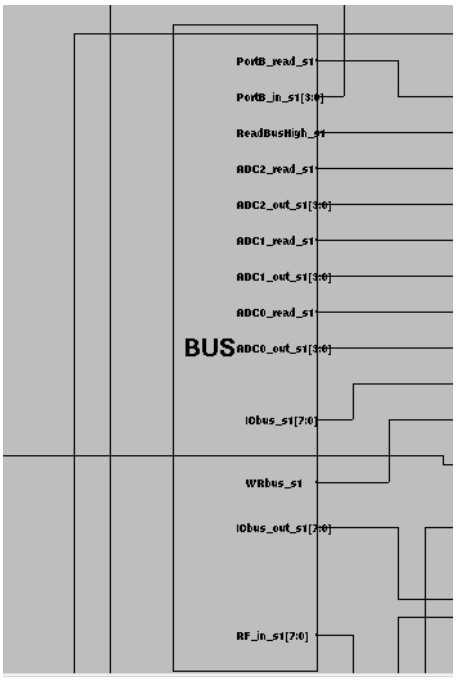


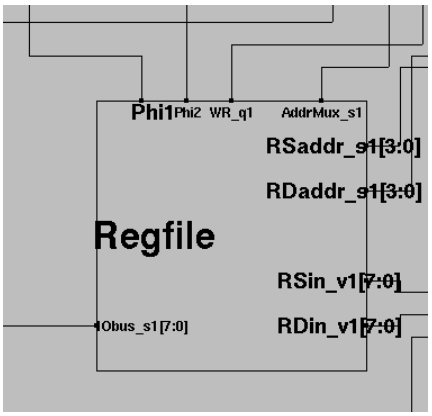
Chip

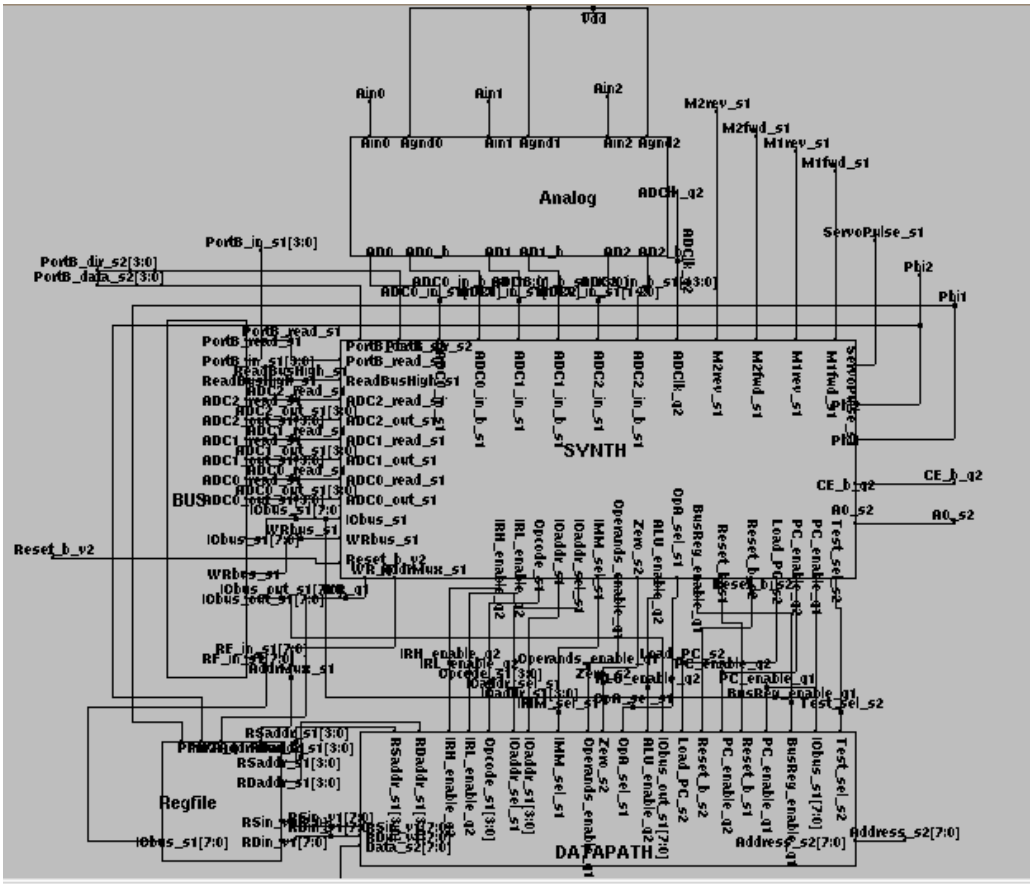

This is the schematic for AMI-CSN 0.5um technology
Lambda=0.35um
M2rev_s1 M2fwd_s1 M1rev_s1 M1fwd_s1
Vdd
Analog
ADClk_q2
Regfile
SYNTH


PortB_read_s1
ReadBusHigh_s1
ADC2_read_s1
ADC2_out_s1
ADC1_read_s1
ADC1_out_s1
ADC0_read_s1
ADC0_out_s1
IObus_s1
WRbus_s1
Reset_b_v2
WR_qAddrMux_s1
ADC0_in_s1
ADC0_in_b_s1
ADC1_in_s1
ADC1_in_b_s1
ADC2_in_s1
ADC2_in_b_s1
ADClk_q2
M2rev_s1
M2fwd_s1
M1rev_s1
M1fwd_s1
Phi2
Phi1
CE_b_q2
A0_s2
Test_sel_s2
PC_enable_q1
Load_PC_s2
Reset_b_s2
Reset_b_s1
BusReg_enable_q1
Operands_enable_q1
IMM_sel_s1
IOaddr_sel_s1
Opcode_s1
IRL_enable_q2
IRH_enable_q2
Zero_s2
OpA_sel_s1
ALU_enable_q2
SYNTH

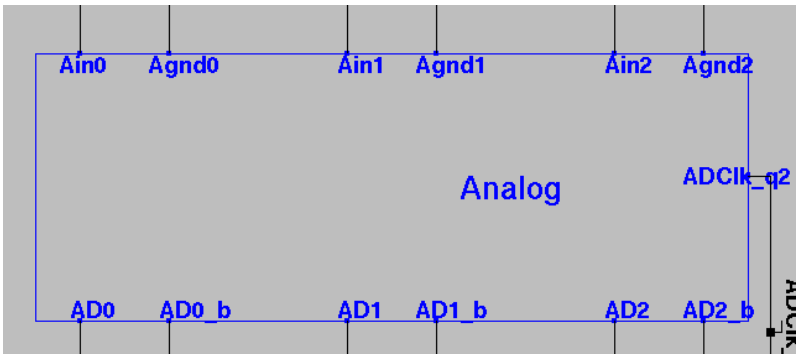

RSaddr_s1[3:0]
RDaddr_s1[3:0]
IRH_enable_q2
IRL_enable_q2
Opcode_s1[3:0]
IOaddr_s1[3:0]
IOaddr_sel_s1
IMM_sel_s1
Operands_enable_q1
Zero_s2
OpA_sel_s1
ALU_enable_q2
IObus_out_s1[7:0]
Load_PC_s2
Reset_b_s2
Reset_b_s1
PC_enable_q2
PC_enable_q1
BusReg_enable_q1
IObus_s1[7:0]
Test_sel_s2
RSin_v1[7:0]
RDin_v1[7:0]
Data_s2[7:0]
Address_s2[7:0]
Address_out_s2[7:0]
DATAPATH

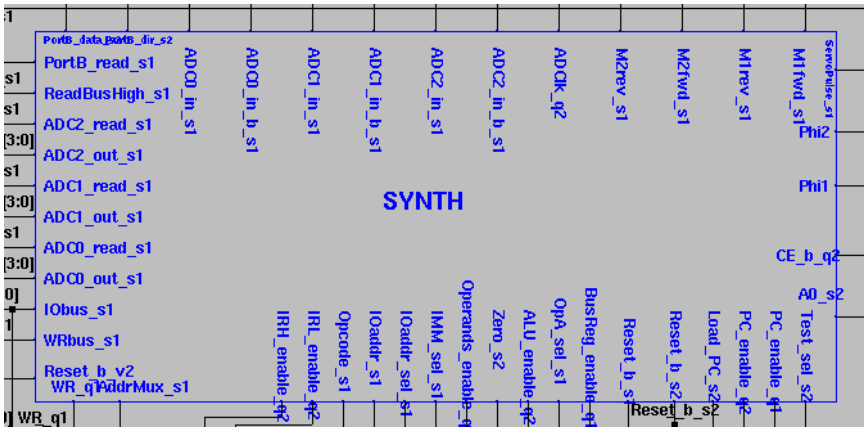

PortB_read_s1
PortB_in_s1[3:0]
ReadBusHigh_s1
ADC2_read_s1
ADC2_out_s1[3:0]
ADC1_read_s1
ADC1_out_s1[3:0]
ADC0_read_s1
BUS
ADC0_out_s1[3:0]
!Obus_s1[7:0]
WRbus_s1
!Obus_out_s1[7:0]
RF_in_s1[7:0]

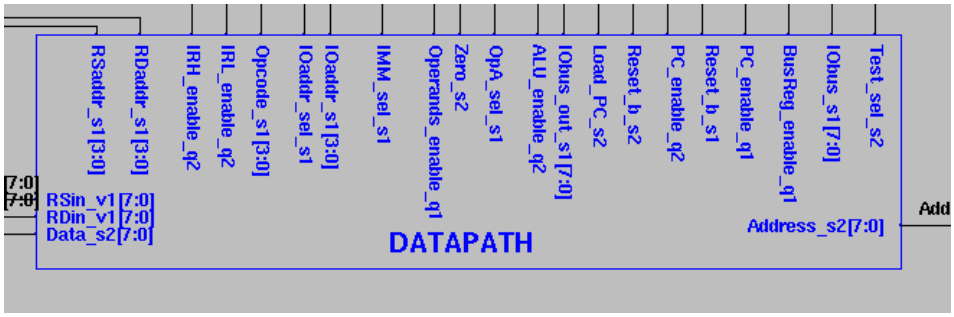

Phi1
Phi2
WR_q1
AddrMux_s1
RSaddr_s1[3:0]
RDaddr_s1[3:0]
Regfile
RSin_v1[7:0]
!Obus_s1[7:0]
RDin_v1[7:0]

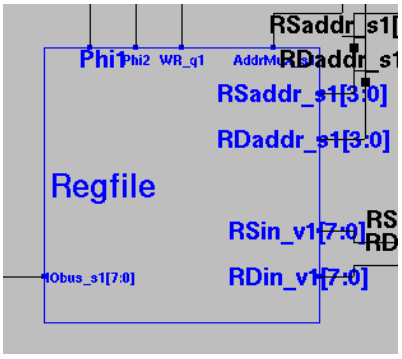

Vdd
Ain0
Ain1
Ain2
M2rev_s1
M2fwd_s1
M1rev_s1
M1fwd_s1
Analog
ADClk_q2
ServoPulse_s1
PortB_in_s1[3:0]
PortB_div_s2[3:0]
PortB_data_s2[3:0]
Phi2
Phi1
SYNTH
BUS
CE_b_q2
A0_s2
Reset_b_v2
RF_in_s1[7:0]
Regfile
DATAPATH
Address_s2[7:0]

Chip Core

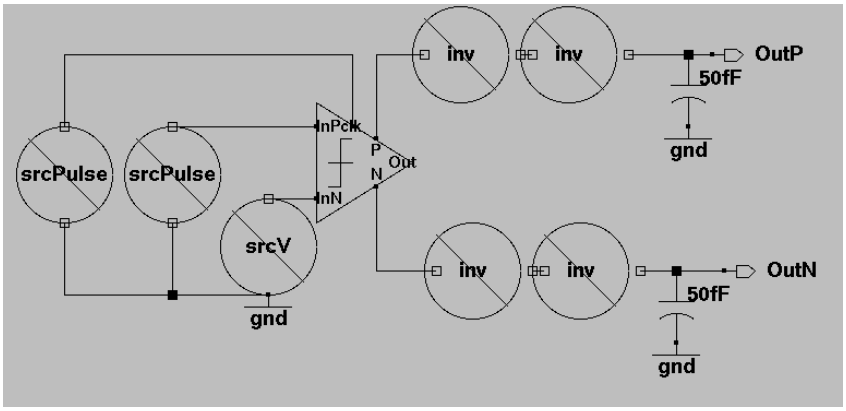




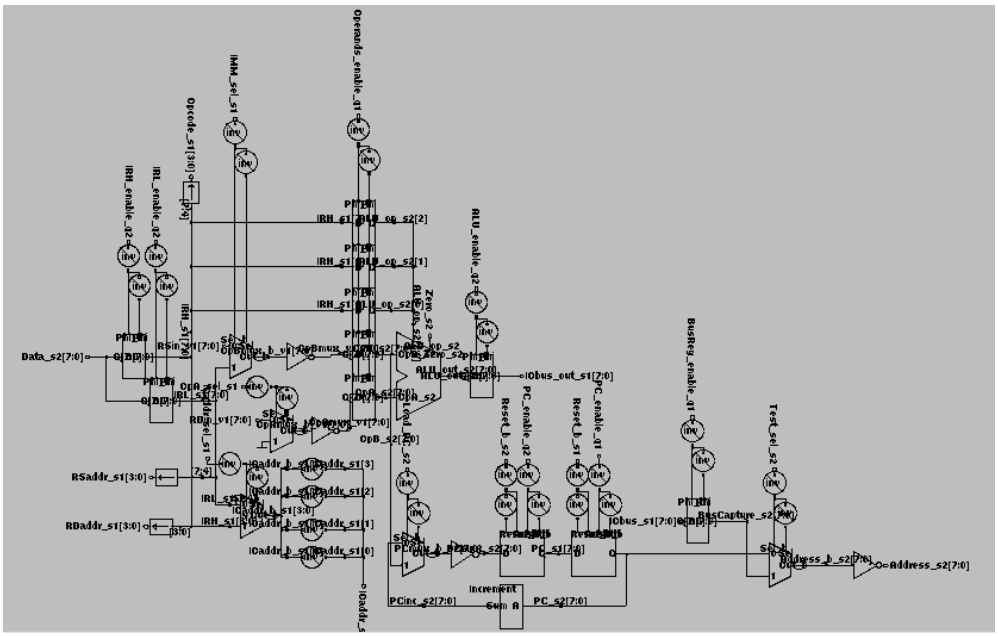


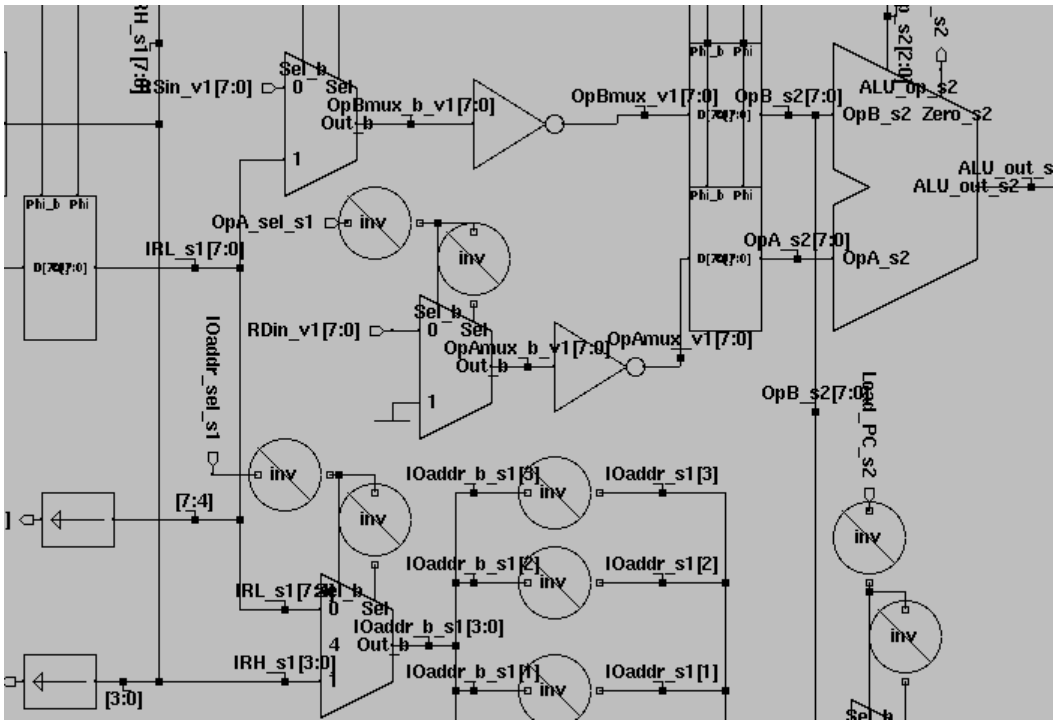


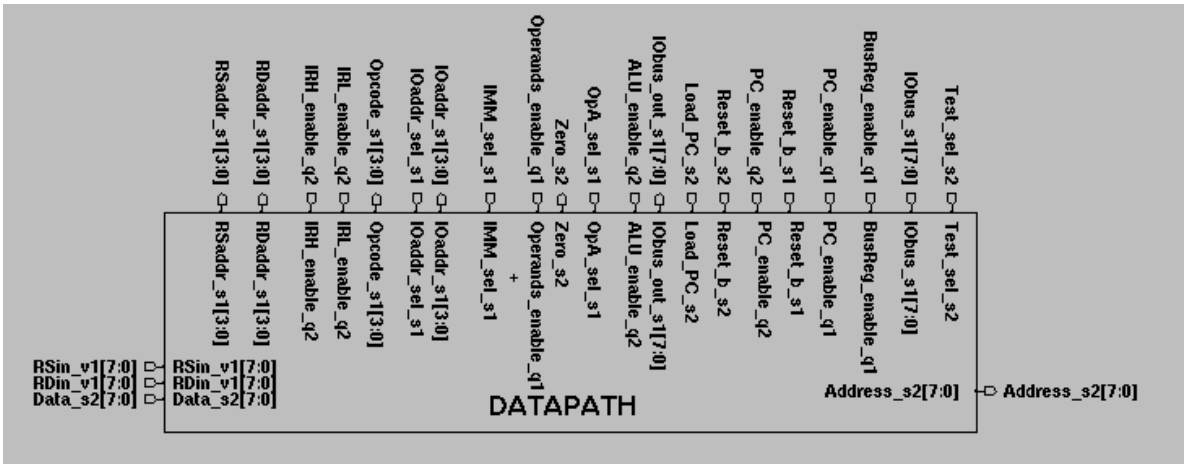
Comparator test



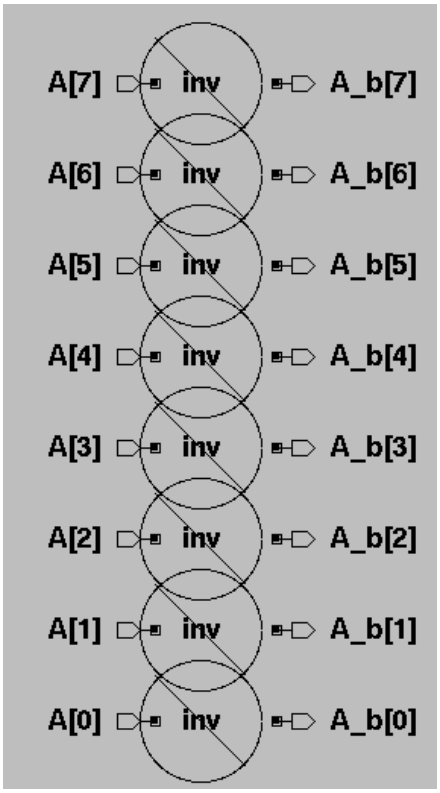
DataPath

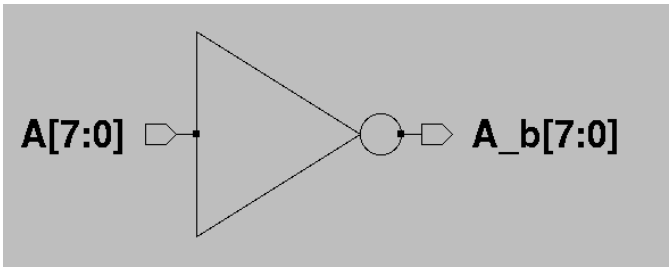



RSin_v1[7:0]
Sel_b
0 Sel
OpBmux_b_v1[7:0]
Out_b
1
OpBmux_v1[7:0]
OpB_s2[7:0]
ALU_op_s2
OpB_s2 Zero_s2
ALU_out_s
ALU_out_s2
Phi_b Phi
OpA_sel_s1
inv
IRL_s1[7:0]
OpA_s2[7:0]
OpA_s2
RDin_v1[7:0]
OpAmux_b_v1[7:0]
OpAmux_v1[7:0]
OpB_s2[7:0]
Load_PC_s2
IOaddr_sel_s1
IOaddr_b_s1[3]
IOaddr_s1[3]
[7:4]
IOaddr_b_s1[2]
IOaddr_s1[2]
IRL_s1[7:4]
IOaddr_b_s1[3:0]
4
IRH_s1[3:0]
IOaddr_b_s1[1]
IOaddr_s1[1]
[3:0]

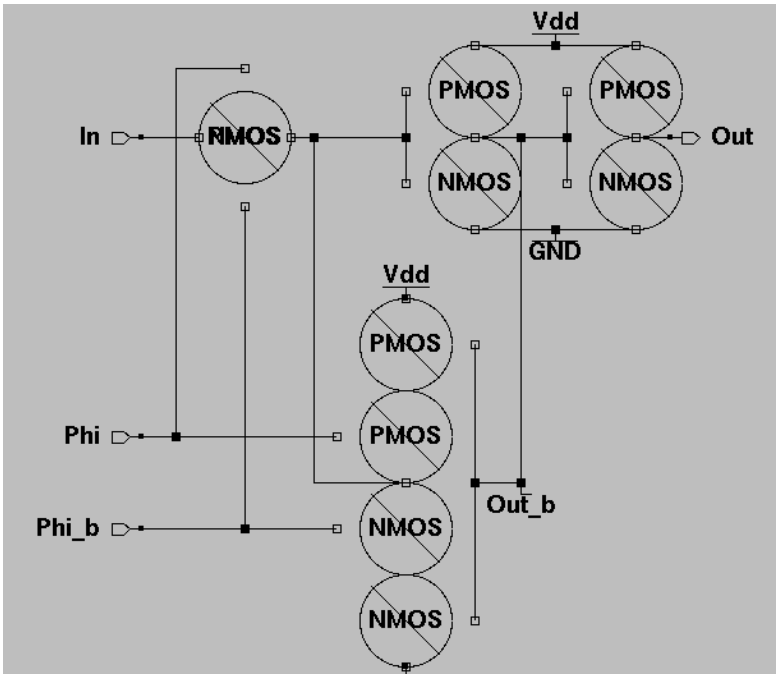

RSaddr_s1[3:0]
RDaddr_s1[3:0]
IRH_enable_q2
IRL_enable_q2
Opcode_s1[3:0]
IOaddr_sel_s1
IOaddr_s1[3:0]
IMM_sel_s1
Operands_enable_q1
Zero_s2
OpA_sel_s1
ALU_enable_q2
IObus_out_s1[7:0]
Load_PC_s2
Reset_b_s2
PC_enable_q2
Reset_b_s1
PC_enable_q1
BusReg_enable_q1
IObus_s1[7:0]
Test_sel_s2
RSin_v1[7:0]
RDin_v1[7:0]
Data_s2[7:0]
DATAPATH
Address_s2[7:0]

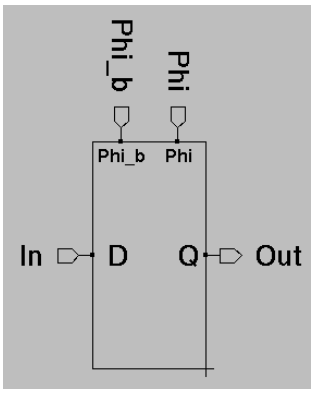
Datapath inverse



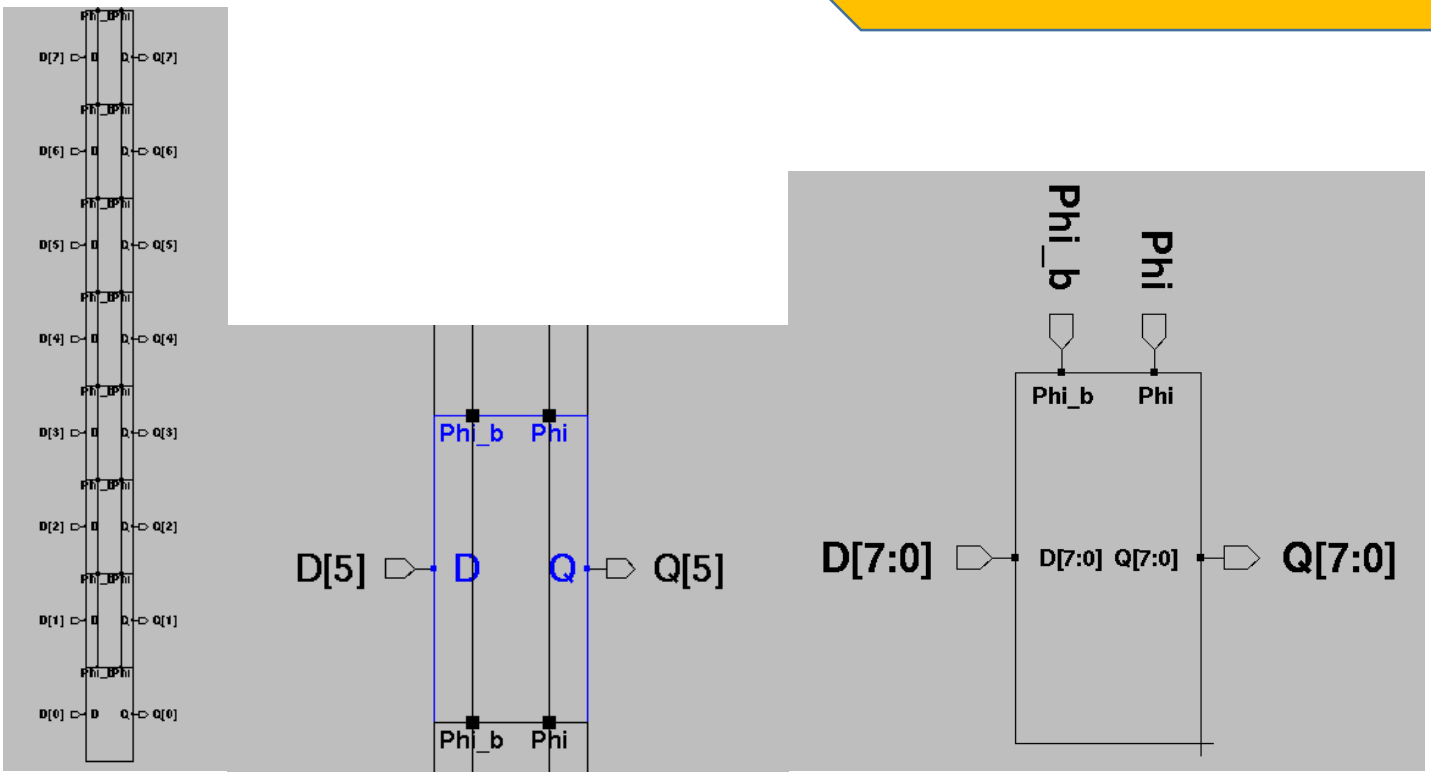


Datapath Latch

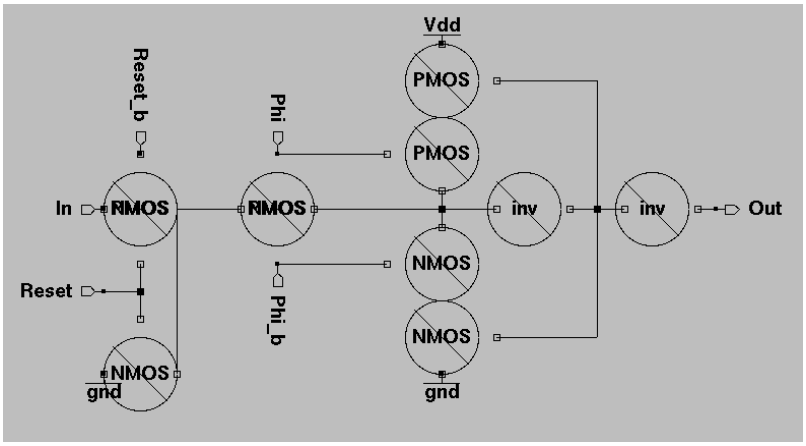


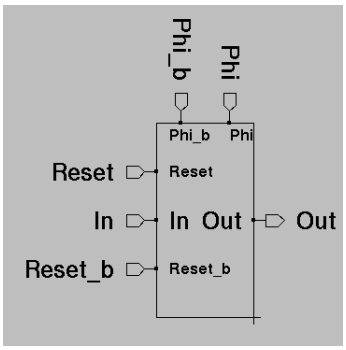


Datapath reset 8بیتی

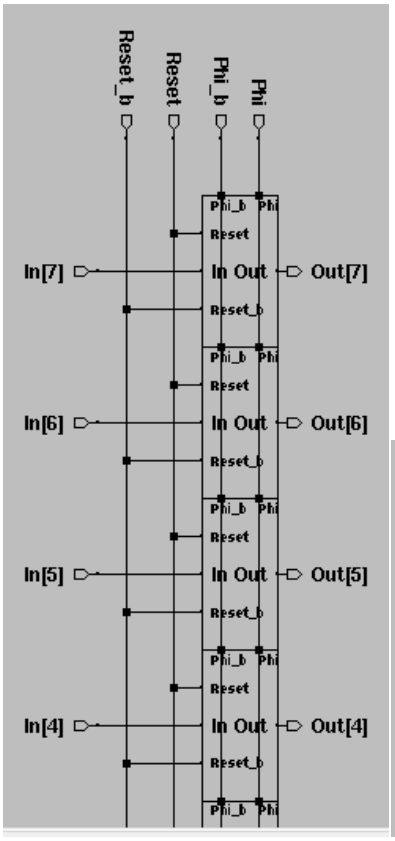


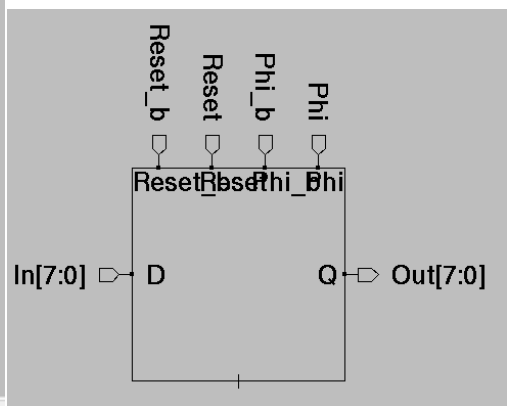
Datapath reset



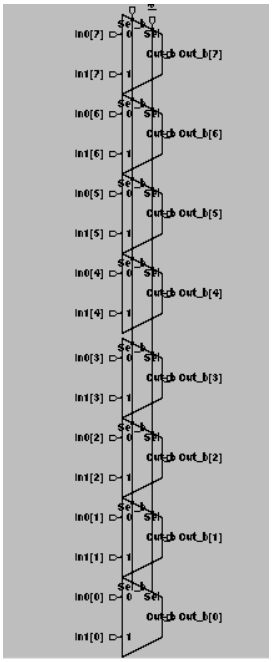


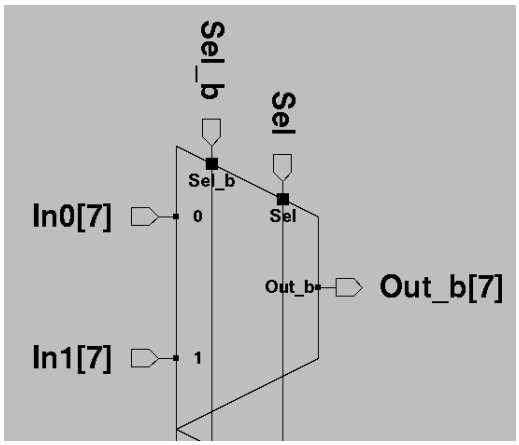
بیتی 8 Datapath reset

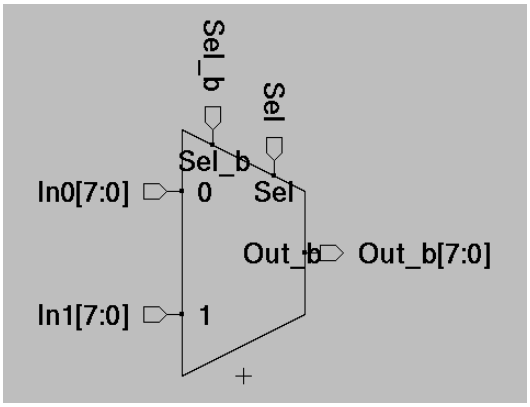




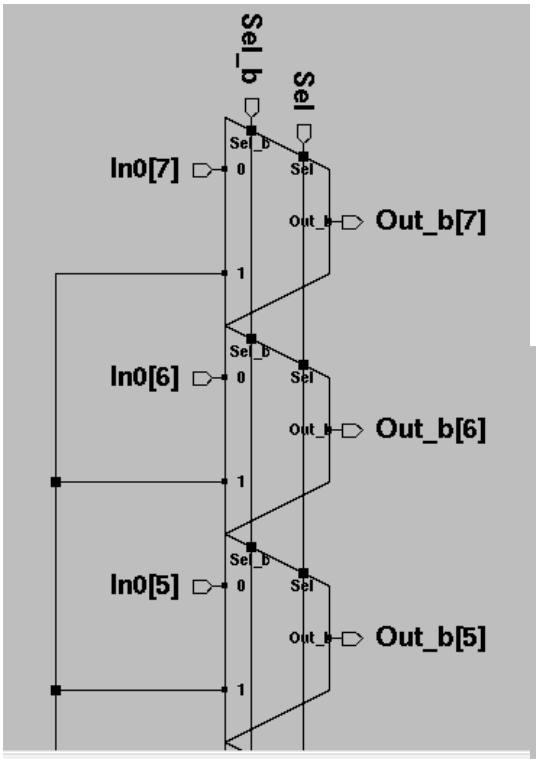
Datapath Multiplexer

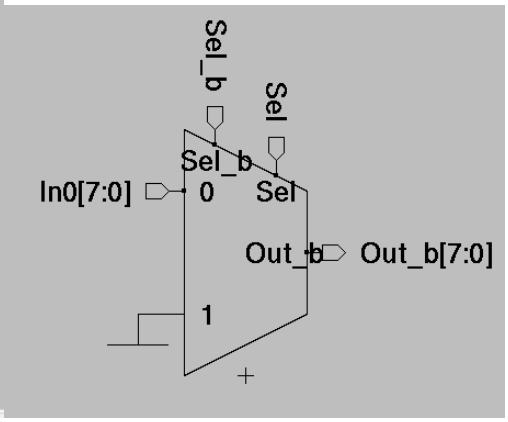




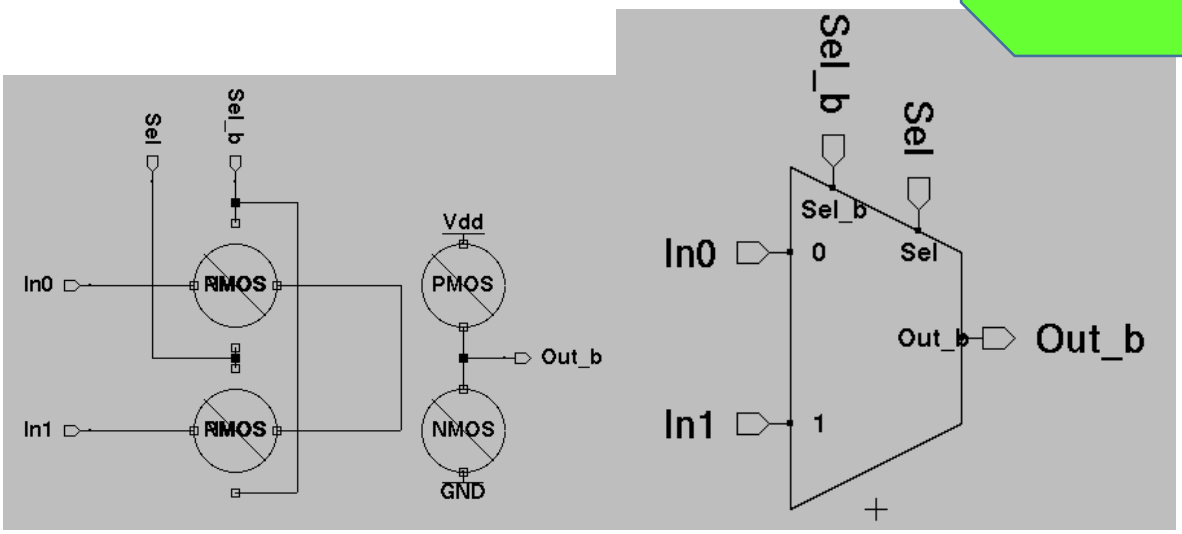


Datapath Multiplexer (GND)

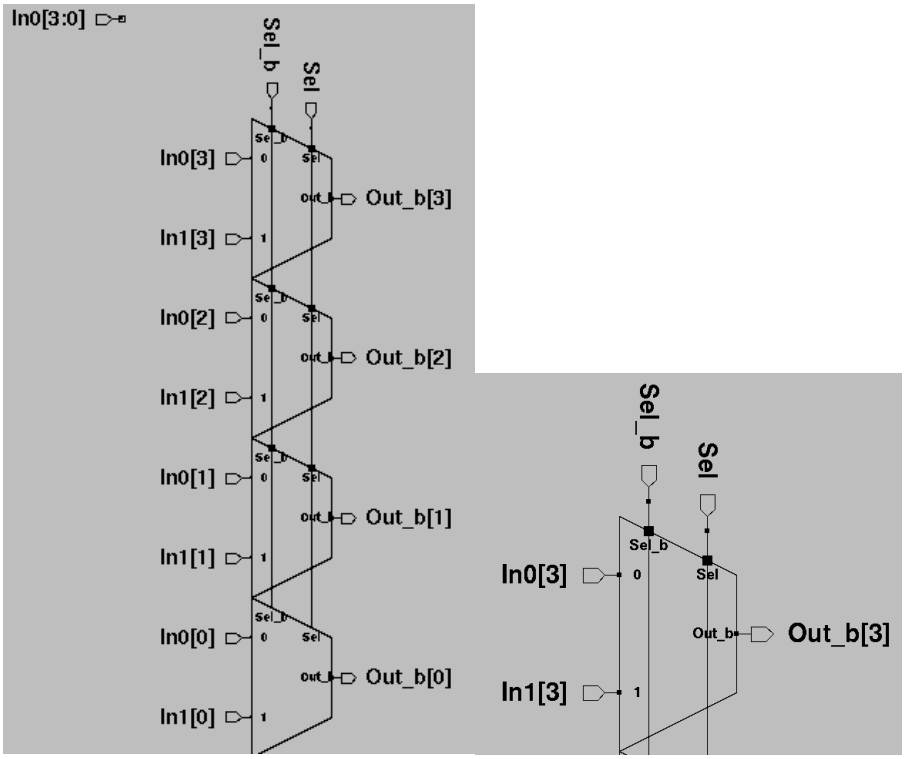


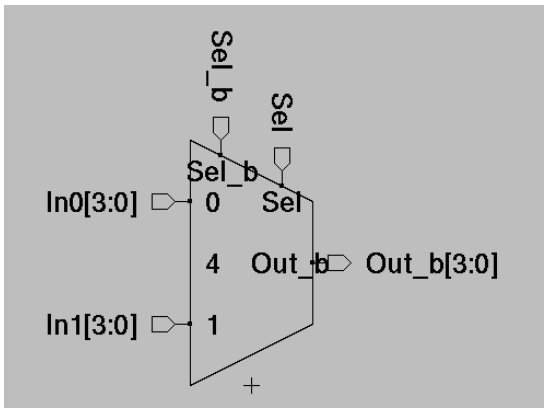


Datapath Multiplexer
2 بیتی

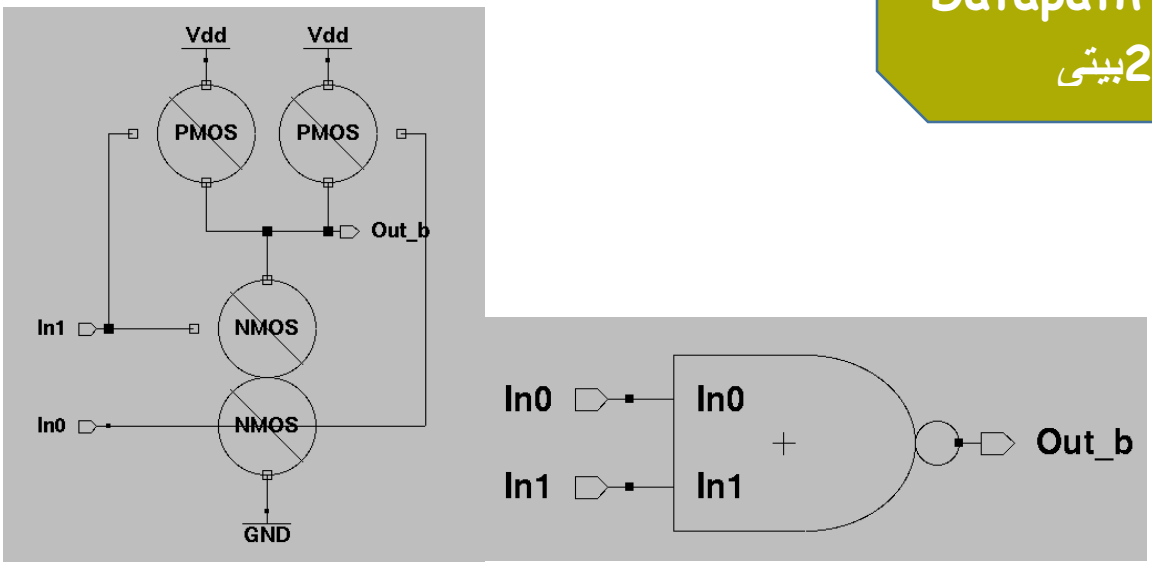


Datapath Multiplexer
3 بیتی

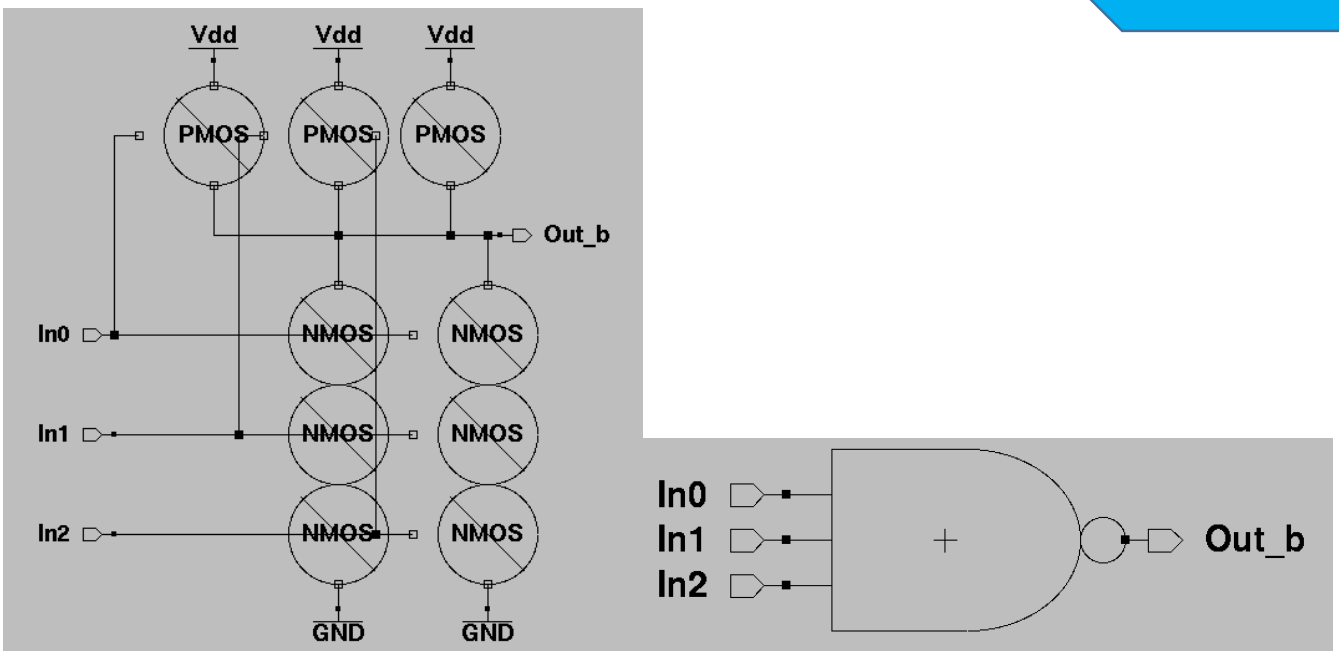




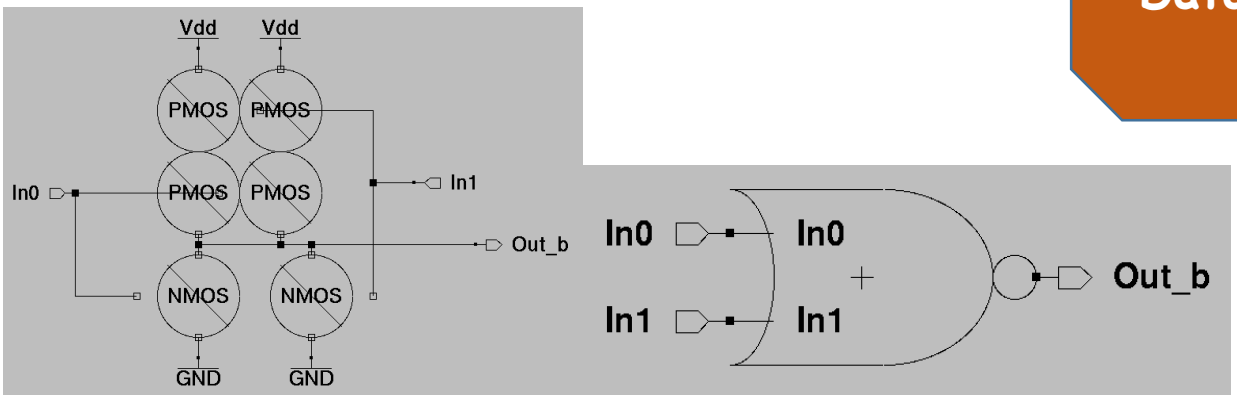
Datapath nand 2بیتی



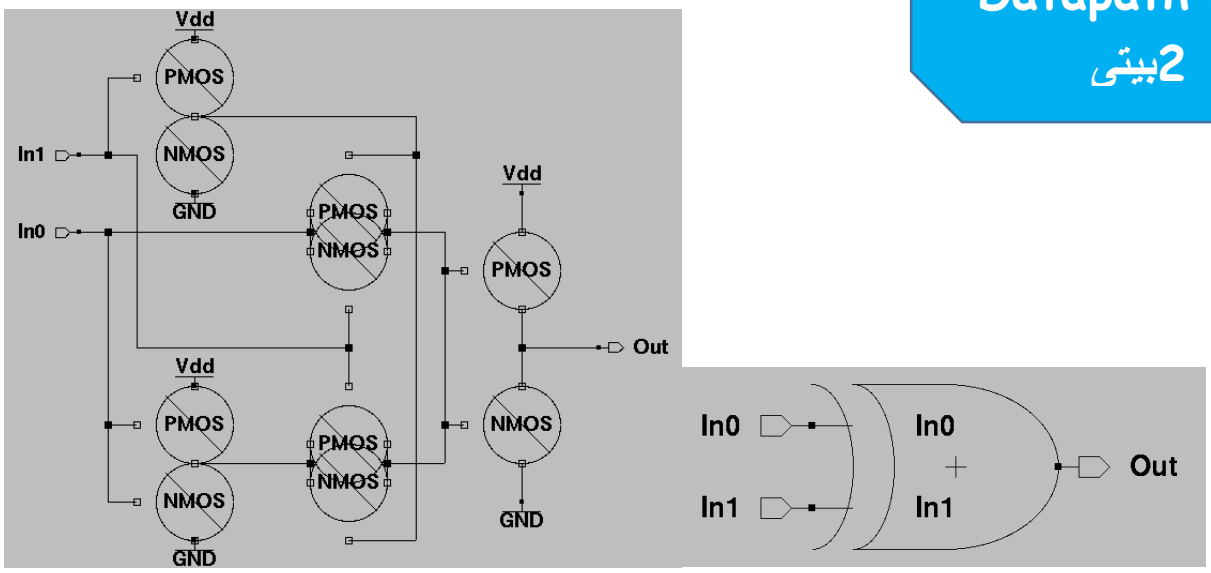
Datapath nand 3بیتی



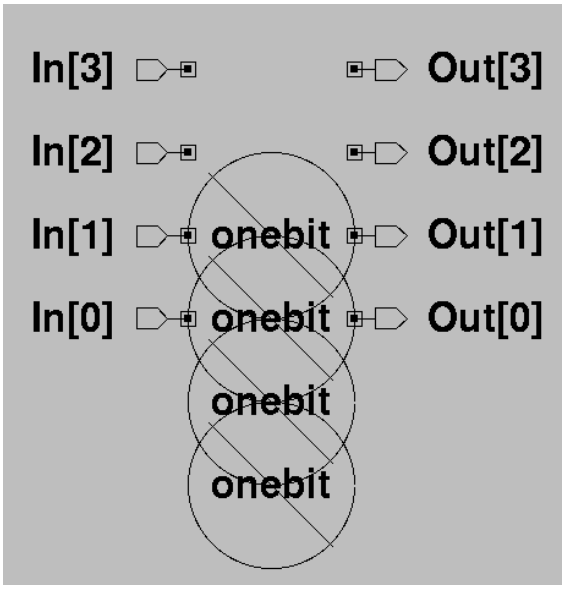
Datapath nor 2بیتی



Datapath xor 2بیتی

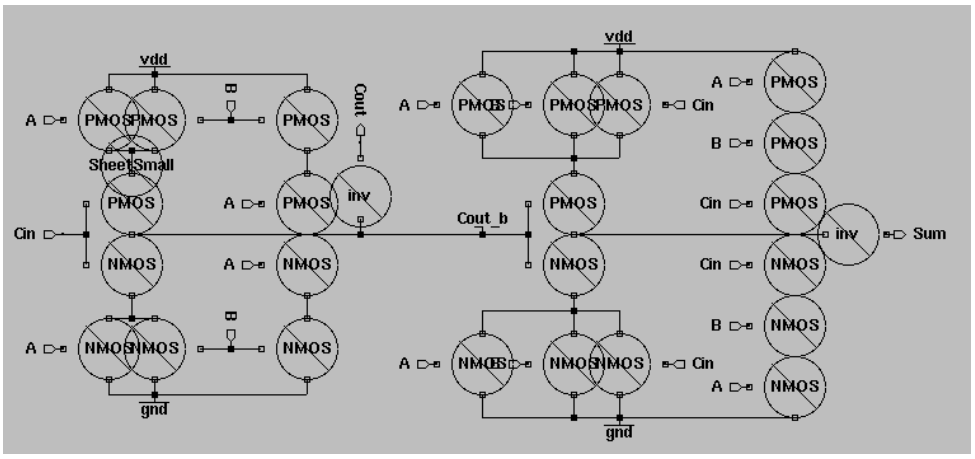


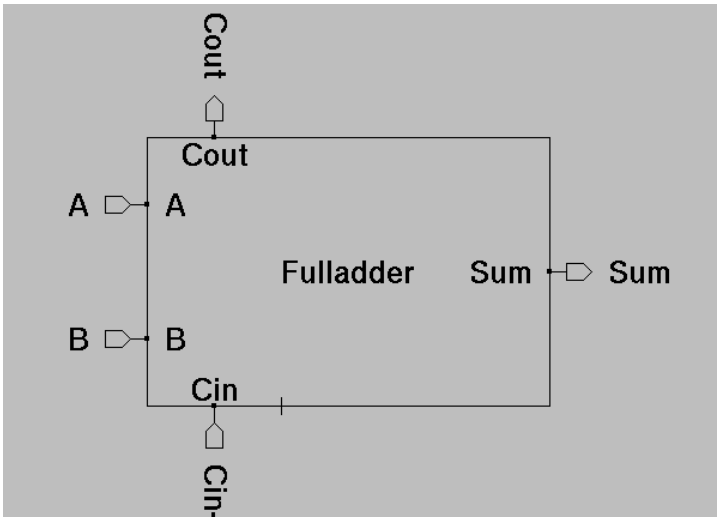
Four Bits

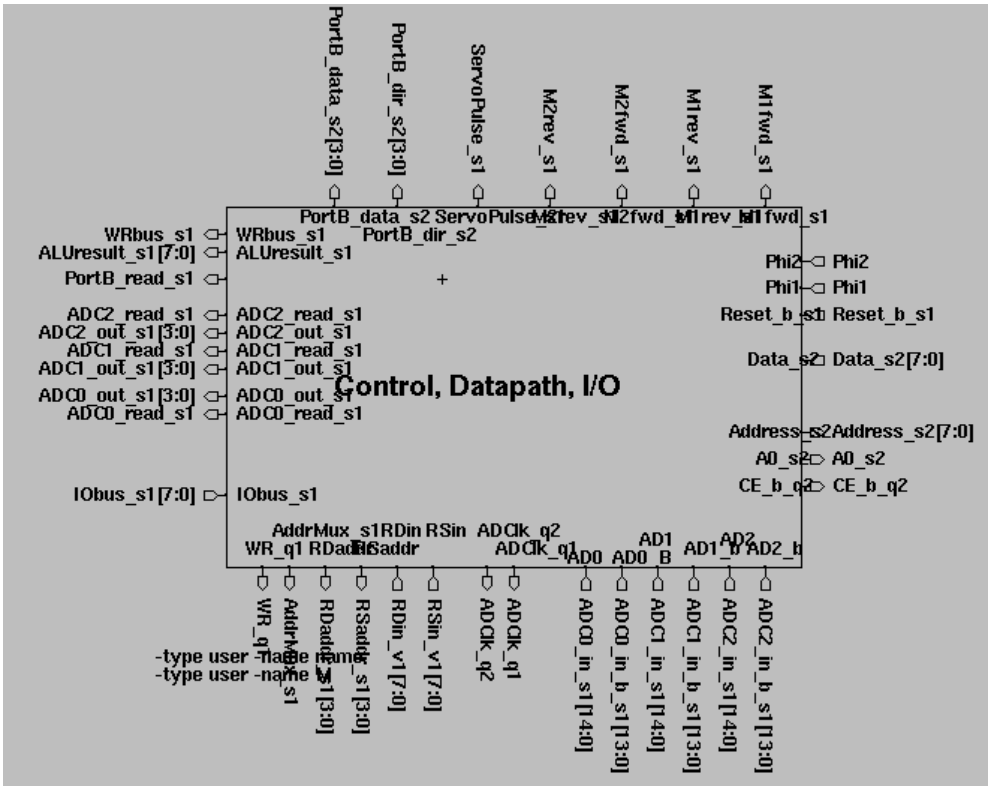


Eight Bits

Full adder

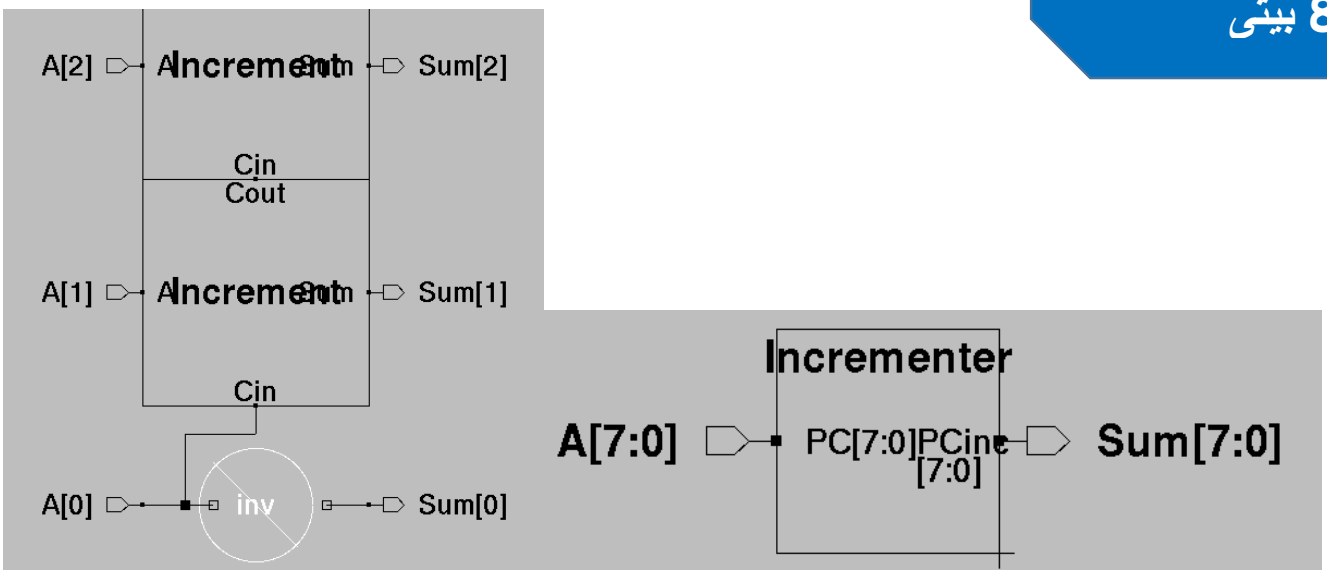




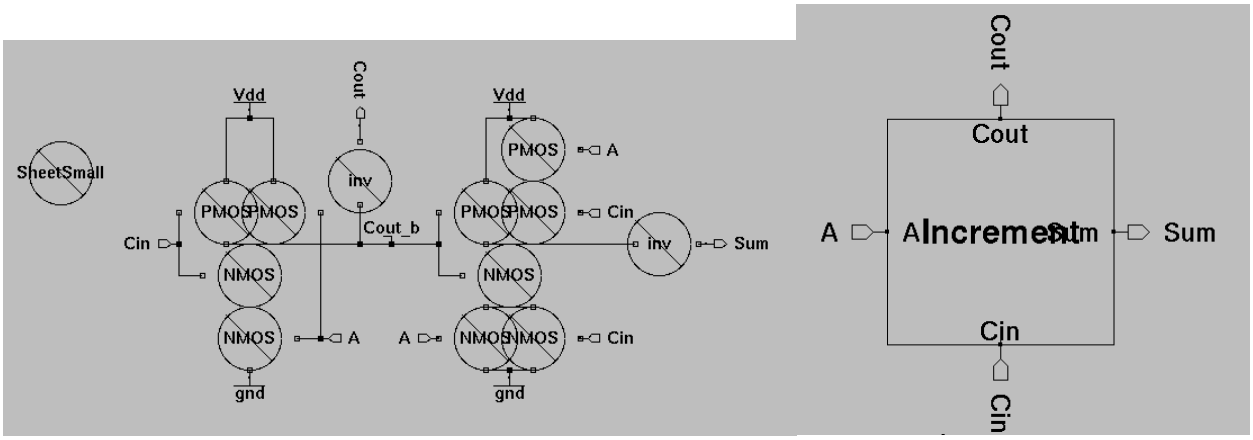


Group of Control / Datapath / IO

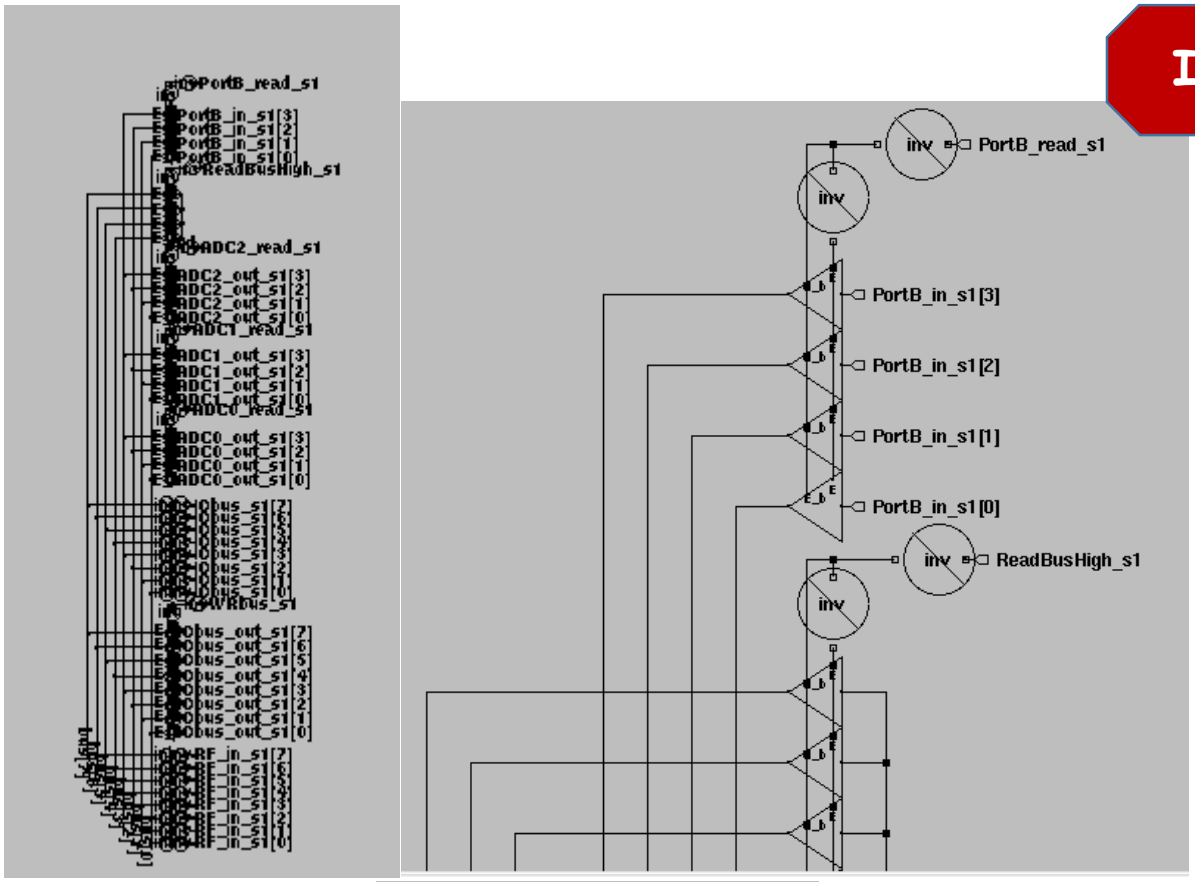
Incrementer
8 بیتی

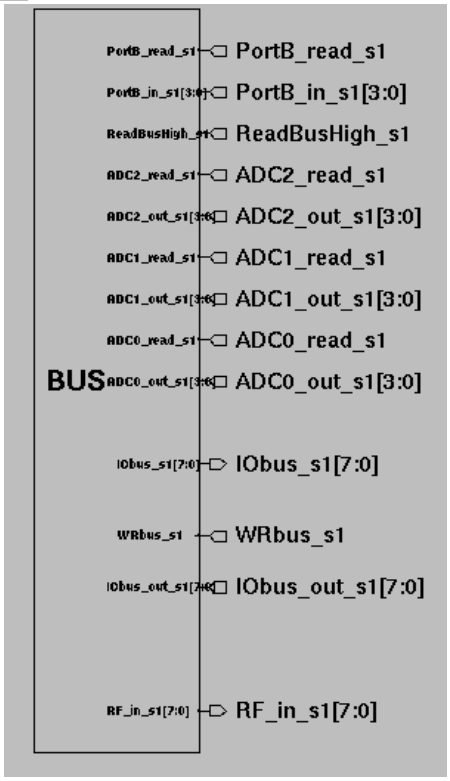


Incrementer
1 بیتی

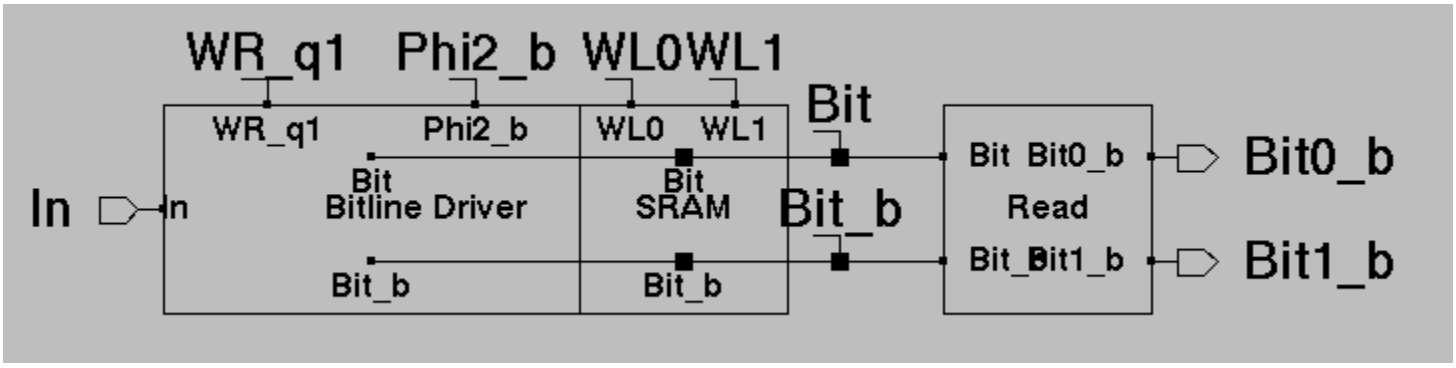


IO Bus

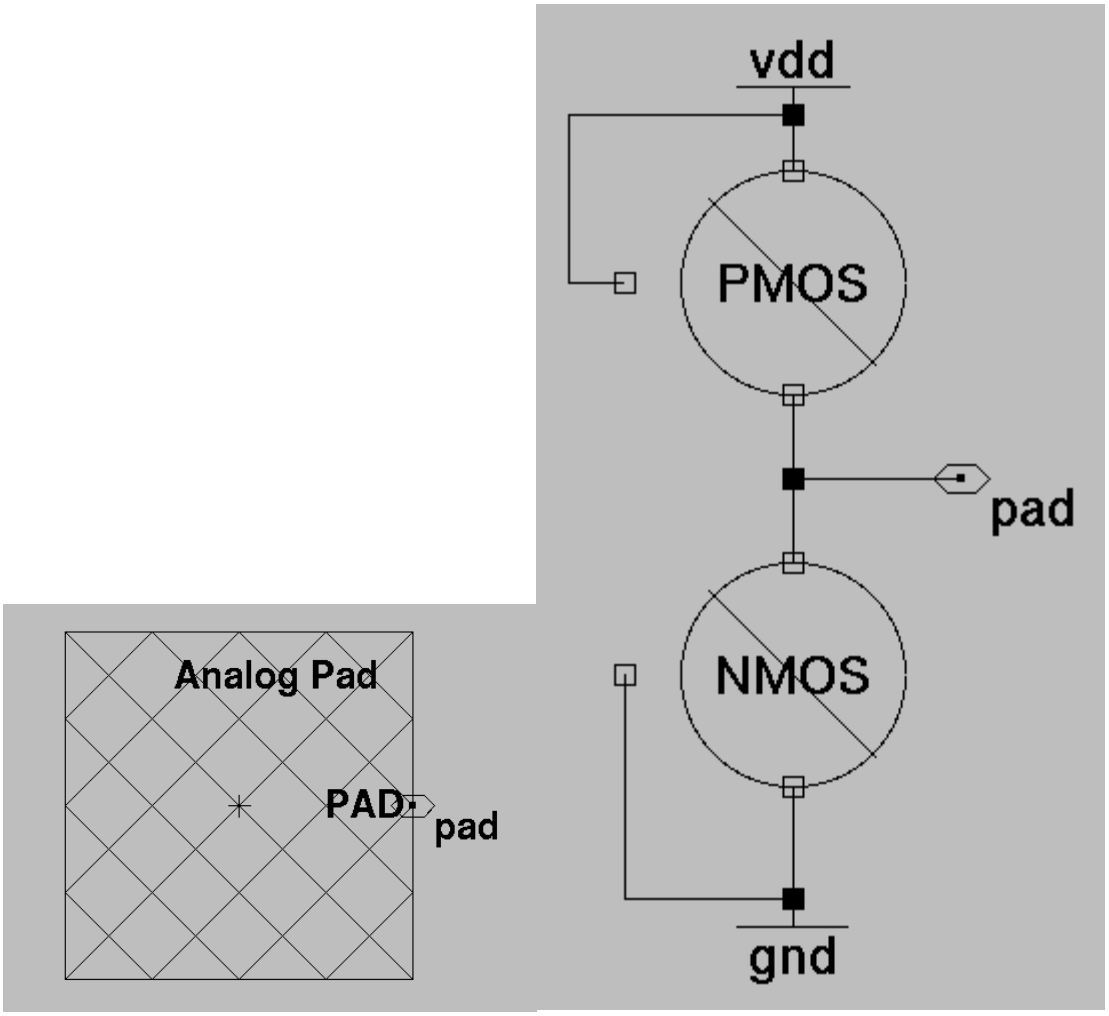

PortB_read_s1
PortB_in_s1[3]
PortB_in_s1[2]
PortB_in_s1[1]
PortB_in_s1[0]
ReadBusHigh_s1
ADC2_read_s1
ADC2_out_s1[3]
ADC2_out_s1[2]
ADC2_out_s1[1]
ADC2_out_s1[0]
ADC1_read_s1
ADC1_out_s1[3]
ADC1_out_s1[2]
ADC1_out_s1[1]
ADC1_out_s1[0]
ADC0_read_s1
ADC0_out_s1[3]
ADC0_out_s1[2]
ADC0_out_s1[1]
ADC0_out_s1[0]
IObus_s1[7]
IObus_s1[6]
IObus_s1[5]
IObus_s1[4]
IObus_s1[3]
IObus_s1[2]
IObus_s1[1]
IObus_s1[0]
WRbus_s1
IObus_out_s1[7]
IObus_out_s1[6]
IObus_out_s1[5]
IObus_out_s1[4]
IObus_out_s1[3]
IObus_out_s1[2]
IObus_out_s1[1]
IObus_out_s1[0]
RF_in_s1[7]
RF_in_s1[6]
RF_in_s1[5]
RF_in_s1[4]
RF_in_s1[3]
RF_in_s1[2]
RF_in_s1[1]
RF_in_s1[0]
inv
PortB_read_s1
inv
PortB_in_s1[3]
PortB_in_s1[2]
PortB_in_s1[1]
PortB_in_s1[0]
inv
ReadBusHigh_s1
inv


BUS
PortB_read_s1
PortB_read_s1
PortB_in_s1[3:0]
PortB_in_s1[3:0]
ReadBusHigh_s1
ReadBusHigh_s1
ADC2_read_s1
ADC2_read_s1
ADC2_out_s1[3:0]
ADC2_out_s1[3:0]
ADC1_read_s1
ADC1_read_s1
ADC1_out_s1[3:0]
ADC1_out_s1[3:0]
ADC0_read_s1
ADC0_read_s1
ADC0_out_s1[3:0]
ADC0_out_s1[3:0]
IObus_s1[7:0]
IObus_s1[7:0]
WRbus_s1
WRbus_s1
IObus_out_s1[7:0]
IObus_out_s1[7:0]
RF_in_s1[7:0]
RF_in_s1[7:0]

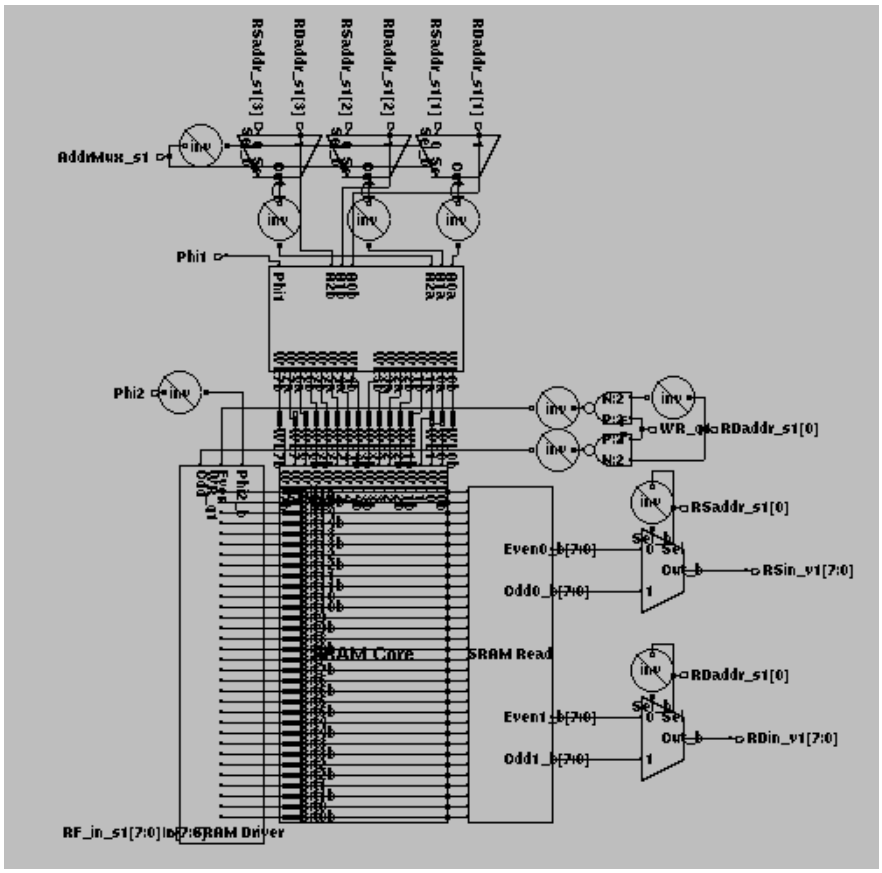
Memory Test

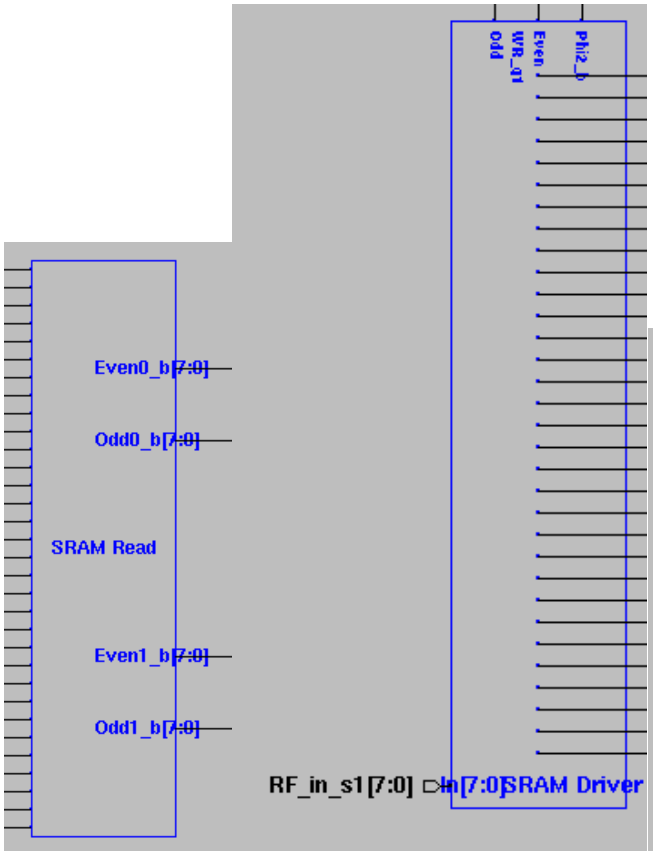


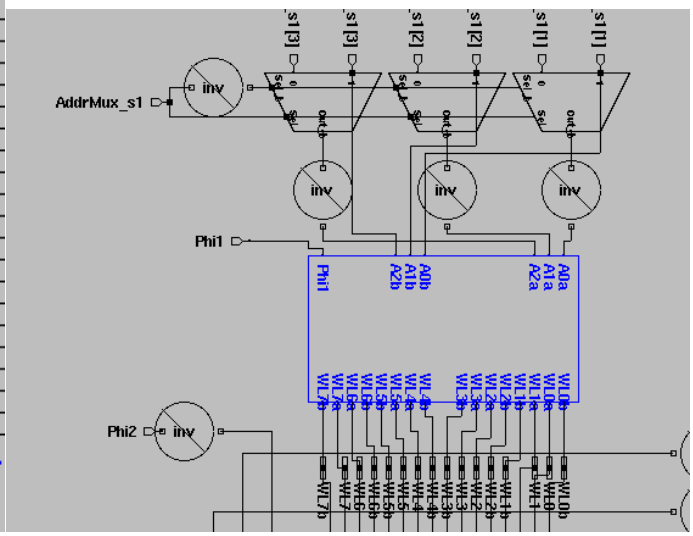
Analog Pad

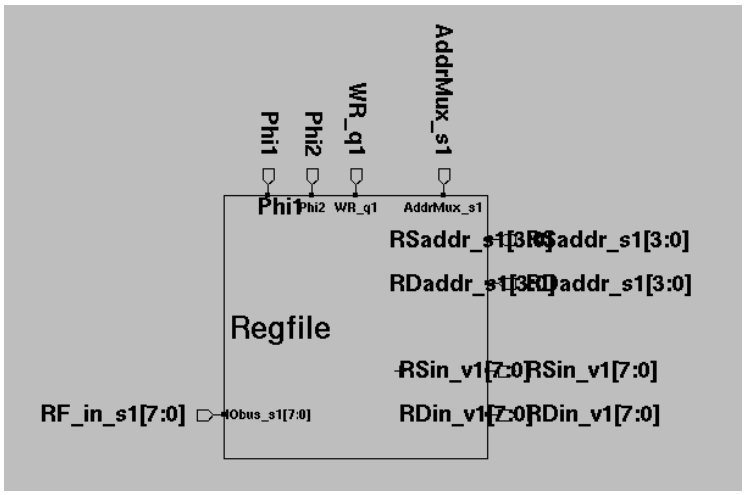


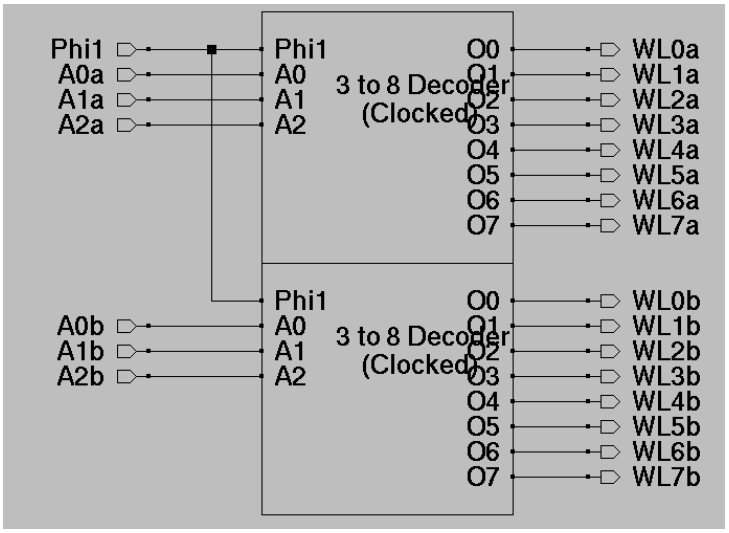
Register File



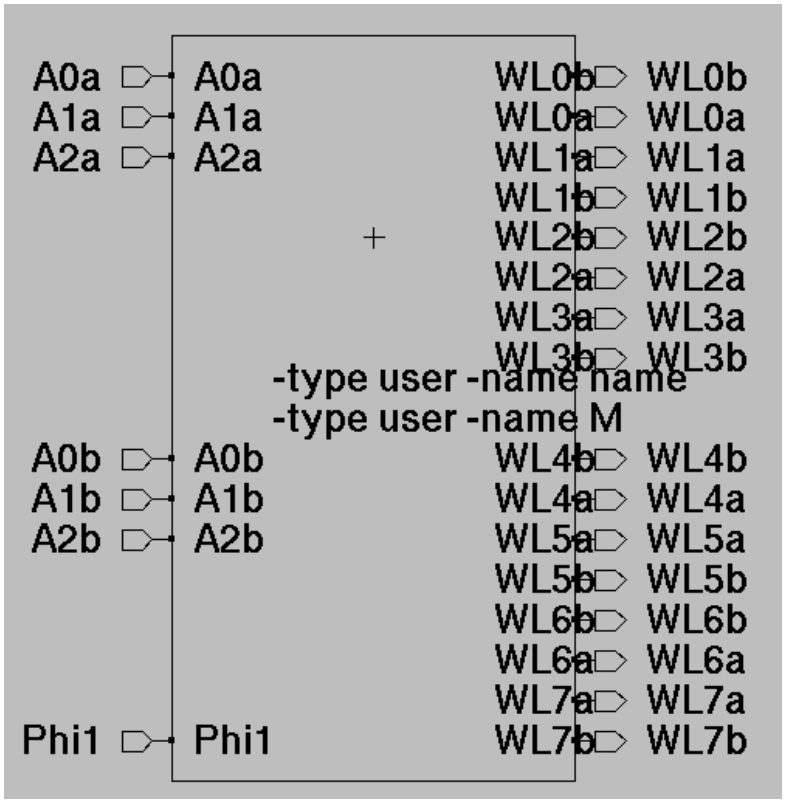




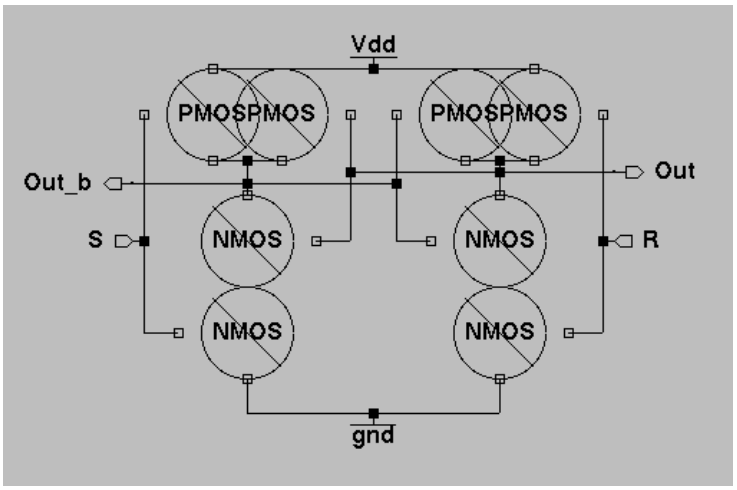


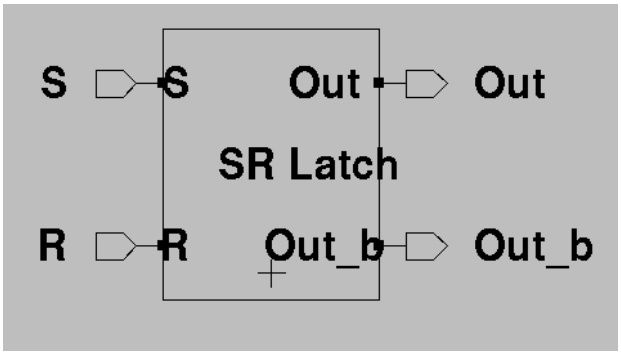


Register File Decoder

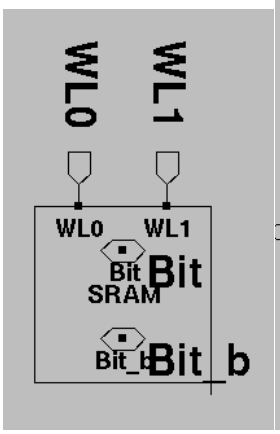


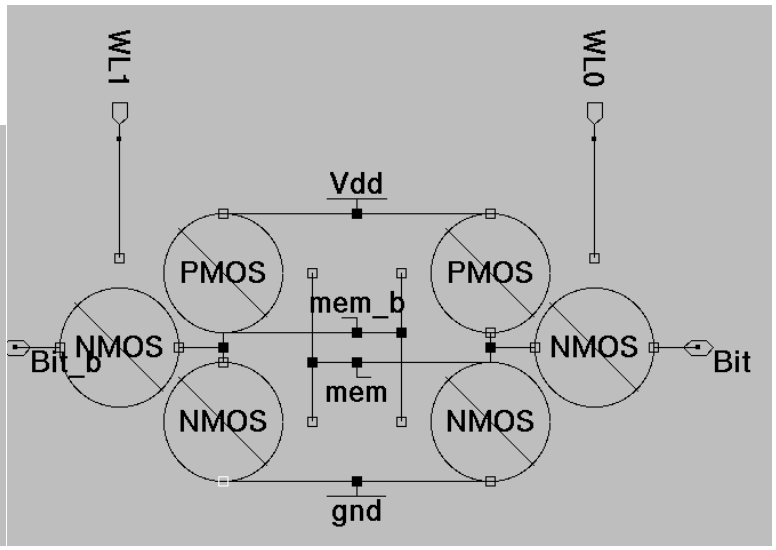
SR Latch



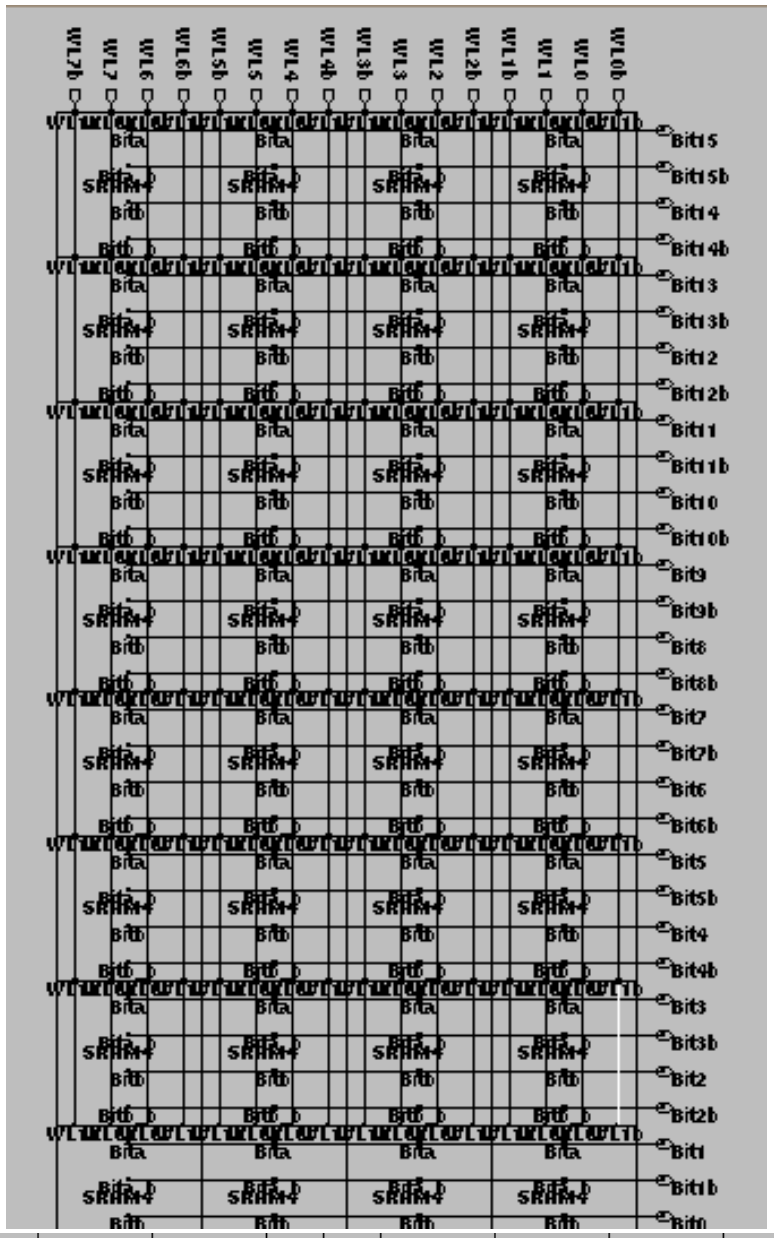


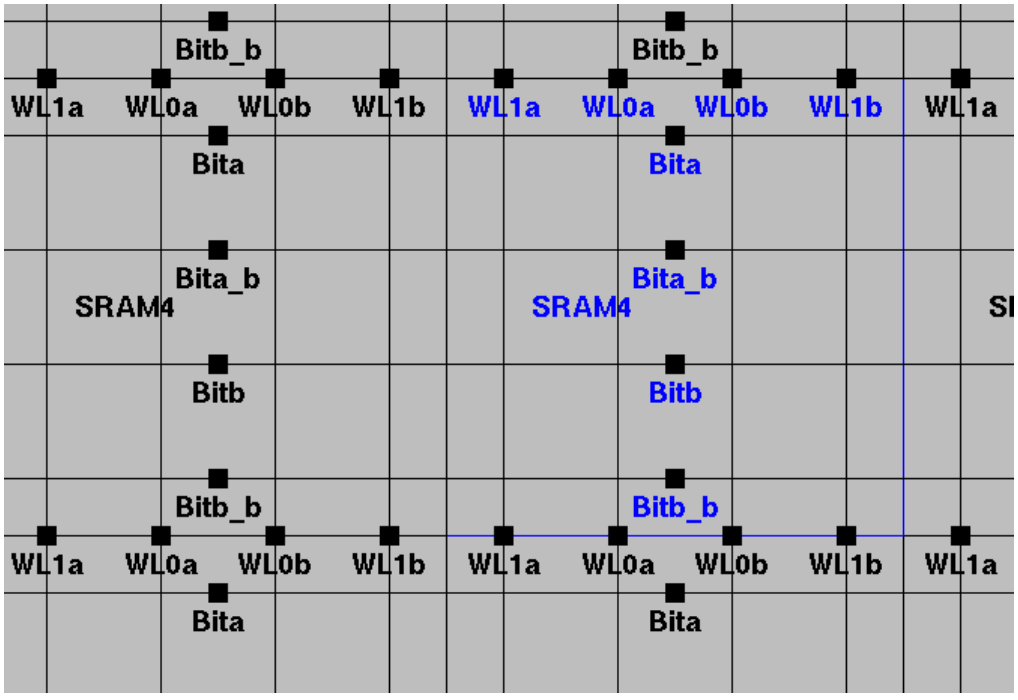
SRAM

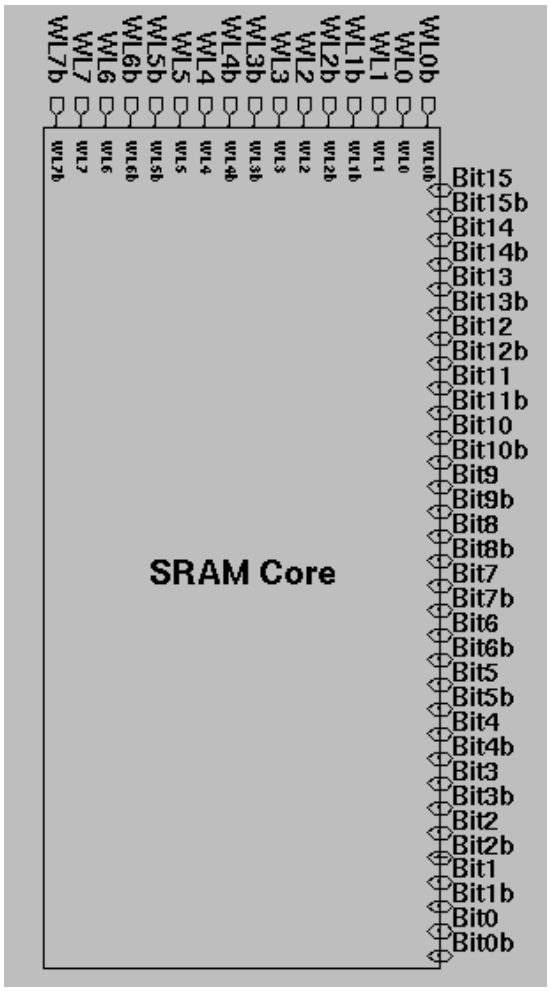




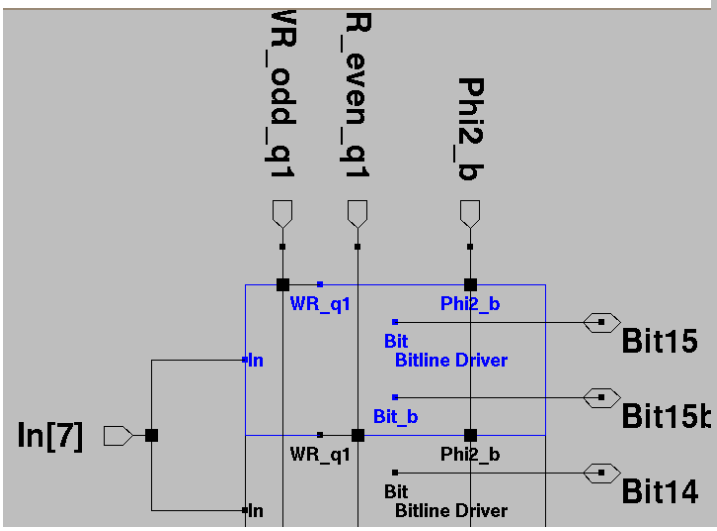
SRAM Core

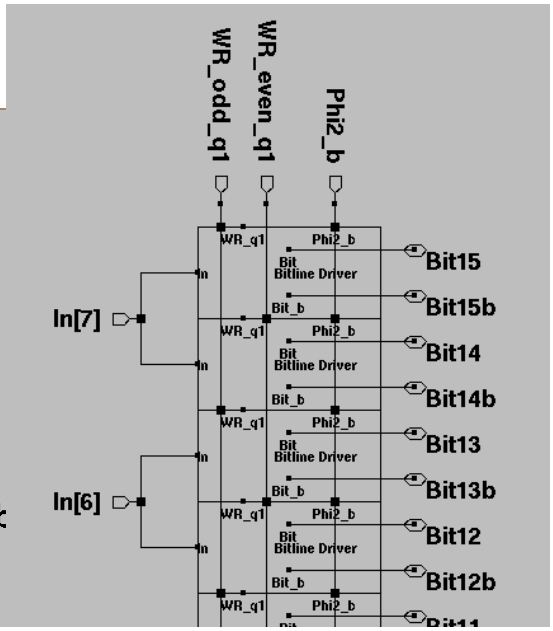

WL7b WL7 WL6 WL6b WL5b WL5 WL4 WL4b WL3b WL3 WL2 WL2b WL1b WL1 WL0 WL0b
Bit15
Bit15b
Bit14
Bit14b
Bit13
Bit13b
Bit12
Bit12b
Bit11
Bit11b
Bit10
Bit10b
Bit9
Bit9b
Bit8
Bit8b
Bit7
Bit7b
Bit6
Bit6b
Bit5
Bit5b
Bit4
Bit4b
Bit3
Bit3b
Bit2
Bit2b
Bit1
Bit1b
Bit0

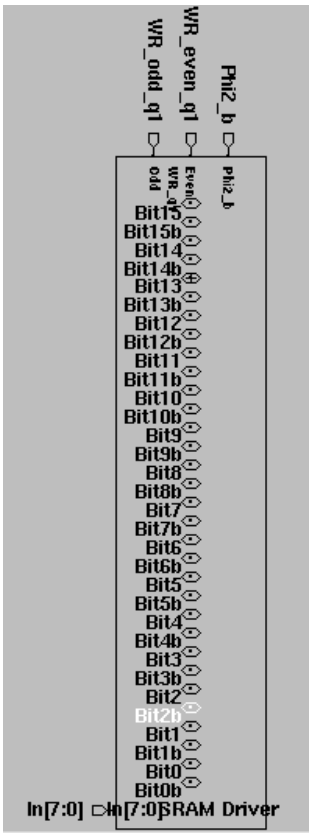

Bitb_b
Bitb_b
WL1a WL0a WL0b WL1b WL1a WL0a WL0b WL1b WL1a
Bita
Bita
Bita_b
Bita_b
SRAM4
SRAM4
Bitb
Bitb
Bitb_b
Bitb_b
WL1a WL0a WL0b WL1b WL1a WL0a WL0b WL1b WL1a
Bita
Bita


WL7b
WL7
WL6
WL6b
WL5b
WL5
WL4
WL4b
WL3b
WL3
WL2
WL2b
WL1b
WL1
WL0
WL0b
SRAM Core
Bit15
Bit15b
Bit14
Bit14b
Bit13
Bit13b
Bit12
Bit12b
Bit11
Bit11b
Bit10
Bit10b
Bit9
Bit9b
Bit8
Bit8b
Bit7
Bit7b
Bit6
Bit6b
Bit5
Bit5b
Bit4
Bit4b
Bit3
Bit3b
Bit2
Bit2b
Bit1
Bit1b
Bit0
Bit0b

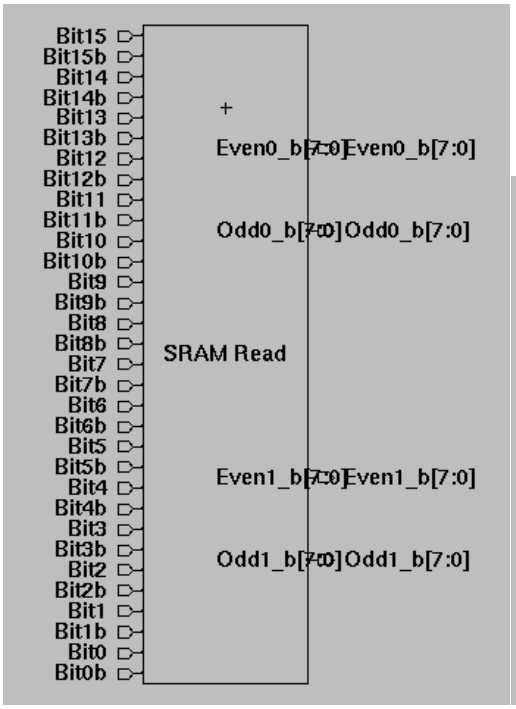
SRAM Driver

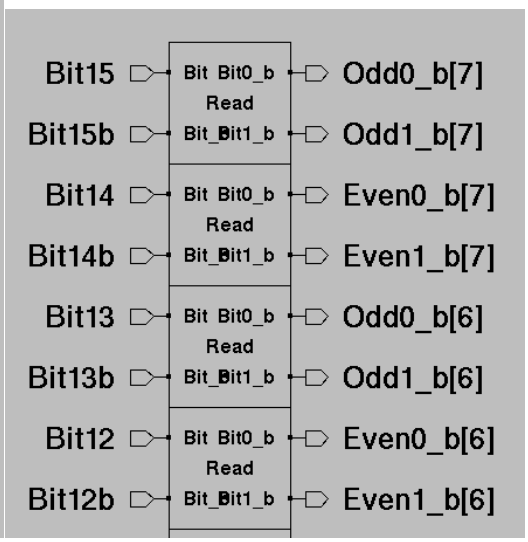

WR_odd_q1
R_even_q1
Phi2_b
WR_q1
Phi2_b
Bit
Bitline Driver
In
Bit_b
In[7]
Bit15
Bit15b
Bit14


WR_odd_q1
WR_even_q1
Phi2_b
In[7]
In[6]
Bit15
Bit15b
Bit14
Bit14b
Bit13
Bit13b
Bit12
Bit12b

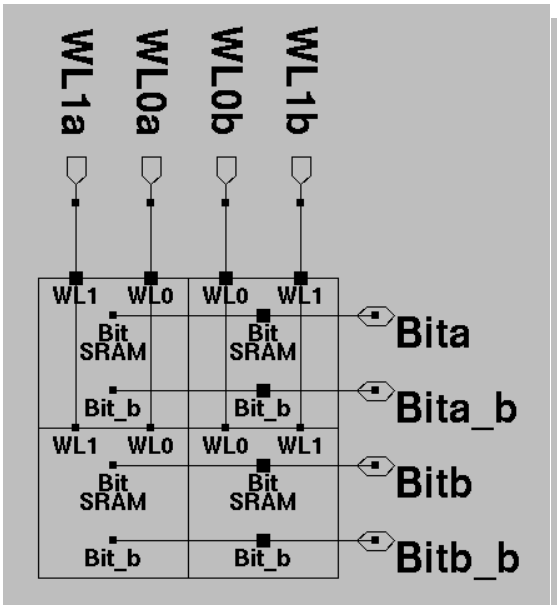


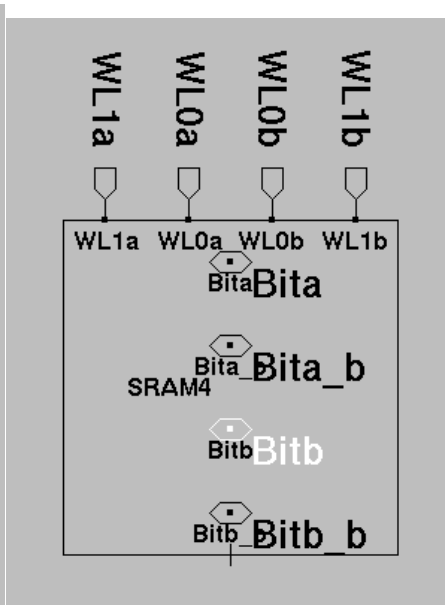
3 to 8 Decoder



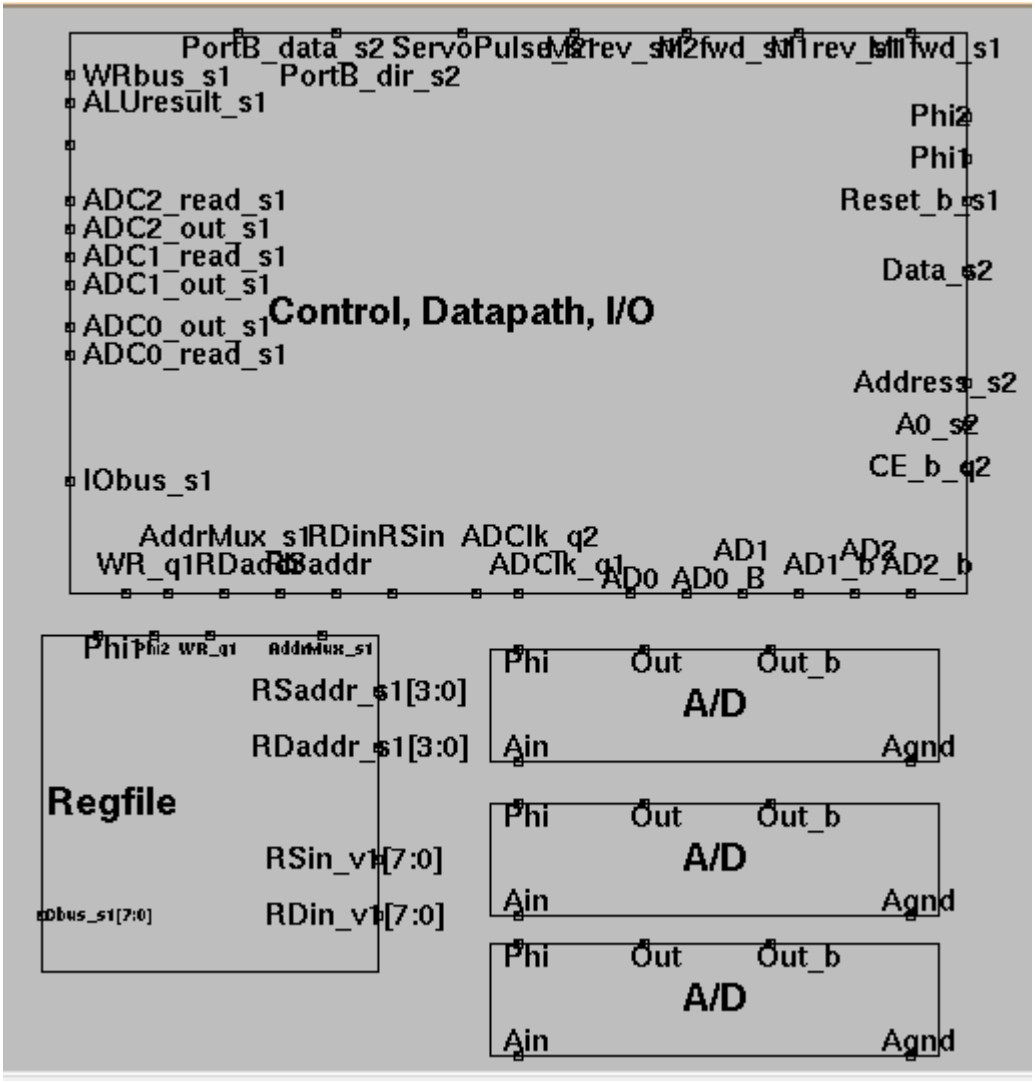


SRAM 4 bit

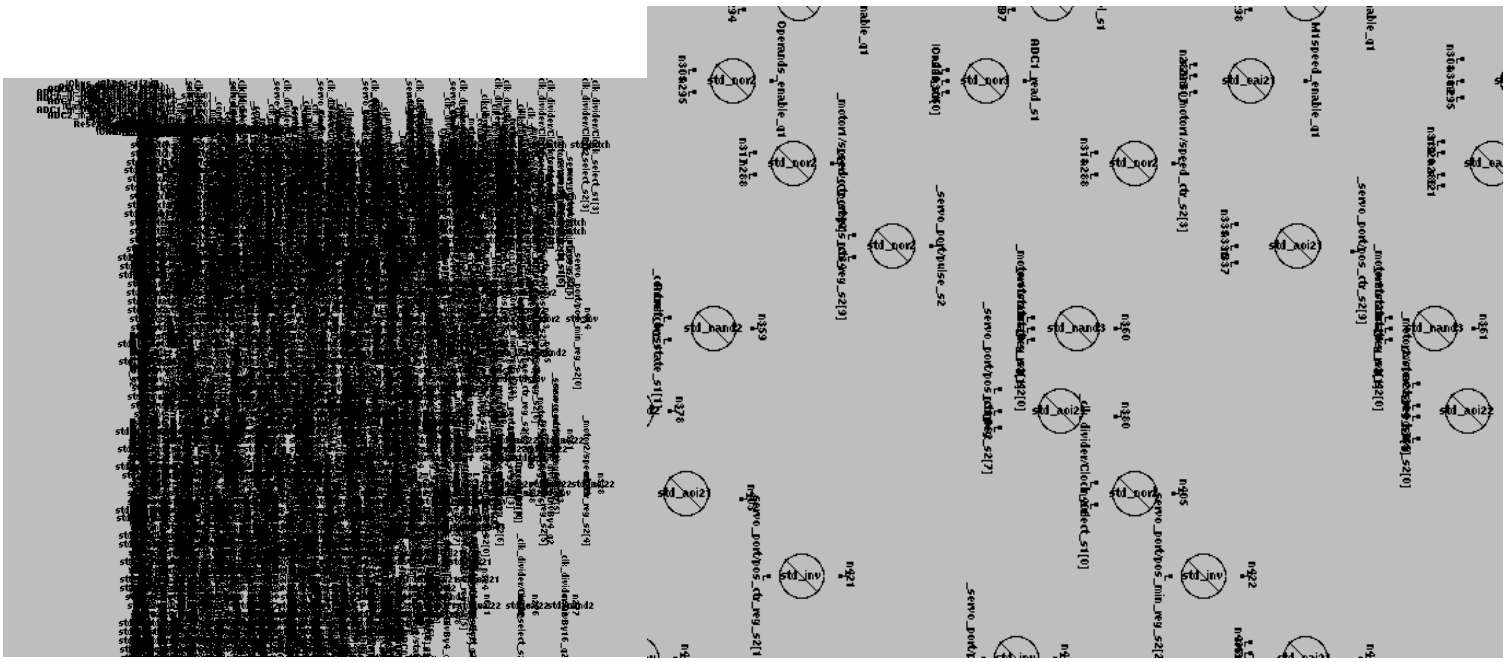


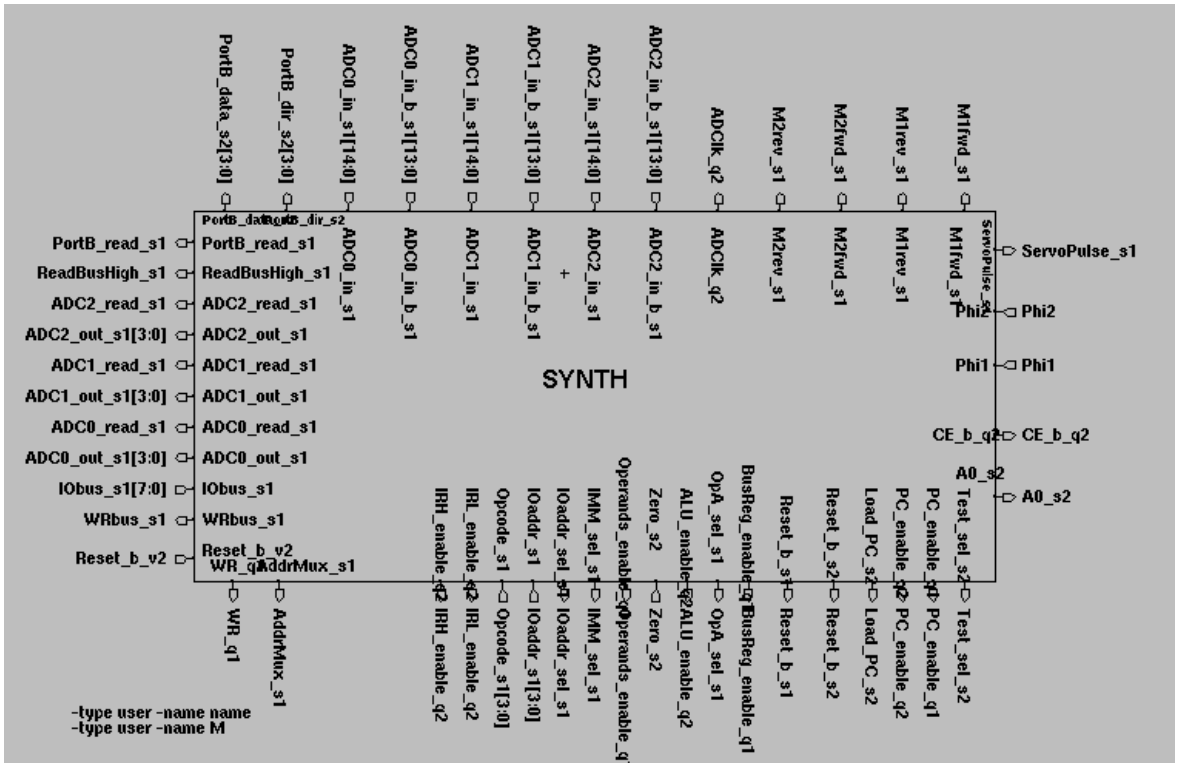


SUCRA

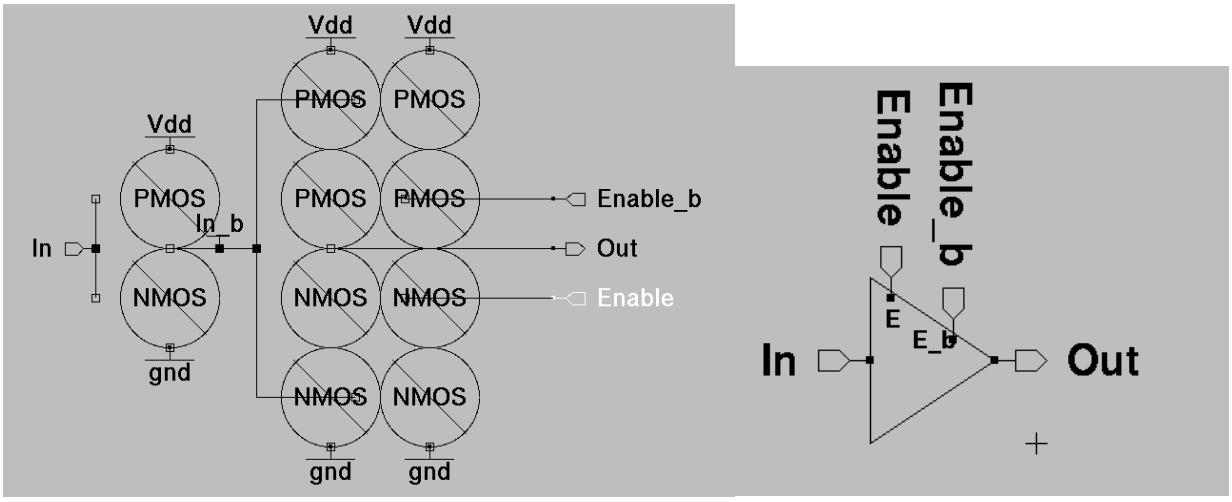


Synthesizer





Tri state



Zero detector