به نام خدا

تمرین کامپیوتری ۳

معماری کامپیوتر – دکتر صفری

اعضای گروه : پریسا محمدی ۸۱۰۱۰۱۵۰۹ / حامد گلکار ۸۱۰۱۰۱۴۹۹

مقدمه :

در این تمرین قصد داریم یک پردازنده مبتنی بر استک را به صورت مولتی سایکل طراحی کنیم.. این پردازنده طبق خواسته صورت پروژه دارای یک باس داده ی ۸ بیتی و یک باس داده ی ۵ بیتی می باشد. این پردازنده همانند پردازنده های مولتی سایکل دیگر به منظور استفاده ی بهینه از المان های سخت افزاری، یک حافظه هم برای دسترسی به دستورات و هم برای دسترسی به دیتا دارد. حافظه ی پردازنده ما ۳۲ در ۸ بیتی است یعنی دارای گنجایش ۳۲ کلمه ی ۸ بیتی می باشد. فرمت دستورات بدین صورت هستند که سه بیست پرارزش به opcode اختصاص دارد که طبق جدول زیر دسته بندی شده اند.

همان طور که در جدول هم مشخص است بعضی از دستورات نیازمند دسترسی به حافظه هستند که در نتیجه ادرس خانه حافظه ی مد نظر خود را در بدنه ی دستور دارند. عملکرد هر یک از این دستور ها به جزییات جلو تر می بینیم.

المان های مورد نیاز برای ساخت مسیر داده پردازنده :

۱) استک : از این المان به عنوان حافظه ی پردازنده و در واقع رجیستر فایل استفاده می شود. این استک نیز همانند هر استک دیگری دارای قابلیت های push,pop می باشد و از انجایی که همواره ما به دیتای بالای استک می خواهیم دسترسی پیدا کنیم ، می باستی ادرسی بالای استک را در رجیستری ذخیره داشته باشیم که اسم انرا طبق خواسته ی صورت پروژه tos گذاشته ایم که یک رجیستر ۵ بیتی است که چرا که فرض کرده ایم استک امان می تواند ۳۲ خانه ی ۸ بیتی داشته باشد ولی برای بررسی راحت تر در بررسی نمونه ، استک را ۱۶ خانه ی ۸ بیتی فرض کرده

ایم. در کد وریلاگ استک هم ابتدا در یک بلوک initial فایل استک را در رجیستری که برای ان تعبیه دیده ایم ذخیره می کنیم. دستورات pop, push با کلاک اعمال می شوند پس برای اجرای انها را در یک بلاک always می گذاریم. در این بلاک اگر که سیگنال pop یک شده باشد، دیتای بالای استک در رجیستر خروجی قرار می گیرد. و اگر سیگنال push یک شده باشد دیتایی که در وردی خوانده شده در بالای استک قرار می گیرد.

۲) ممووری : این المان همان طور که قبلا اشاره شد، هم دیتا و هم دستورات در خود ذخیره دارد ولی این که از کجا تا کجا را داده و چه خانه هایی را دستورات به خود اختصاص داده اند را در عمل با کمک کامپایلر و نرم افزاری هماهنگ می شود ولی اینجا ما در پروژه خود در دستورات نمونه ای که برای تست پروژه داده ایم خودمان حواسمان بوده که کدام خانه ها شامل داده و کدام خانه ها دستورات را در خود جای داده اند. در کد وریلاگ ان در ابتدا در یک بلوک initial از مموری فایل مقادیر را خوانده و در رجیستر ای ۸ در ۳۲ بیتی که طبق خواسته سوال ایجاد کرده ایم می گذاریم. برای خواندن از حافظه بدون توجه به کلاک یعنی به صورت اسنکرون به المان های حافظه دسترسی پیدا می کنیم و یعنی می توانیم خواندن از حافظه را به کمک دستور assign به راحتی انجام دهیم. ولی نوشتن در حافظه را به صورت سنکرون با کلاک و در یک بلاک always انجام می دهیم. در این بلاک هرگاه write_en یک شود دیتای ورودی را در ادرسی که ان هم از ورودی دریافت شده است قرار می دهیم.

۳) alu : در یک پردازنده ی مولتی سایکل به علت استفاده بهینه از ماژول های سخت افزاری فقط یک ماژول محاسباتی در قالب alu داریم که در هر بخشی از برنامه لازم بود عملیات ای بر روی مقادیر رجیستر ها انجام شود با رعایت حق تقدم و در کلاک سایکل کنترل شده توسط واحد کنترلی alu این کار را انجام می دهد. Alu مورد نیاز ما لازم است عملیات های جمع، تفریق ، and و Not را انجام دهد. بدین منظور دارای سیگنال کنترلی op_code است که تعیین می کند کدام عملیات انجام شود و همچنین در ورودی دارای دو مقدار عملوند نیز می باشد. در خروجی حاصل عملیات را نتیجه می دهد و همچنین سیگنال zero را برای وقتی که مقدار عملوند اول برابر با صفر است را نتیجه می دهد. برای پیاده سازی ان نیز در کد وریلاگ از بلوک always استفاده شده است که در ان نیز از ساختار case استفاده می شود. با استفاده از ساختار

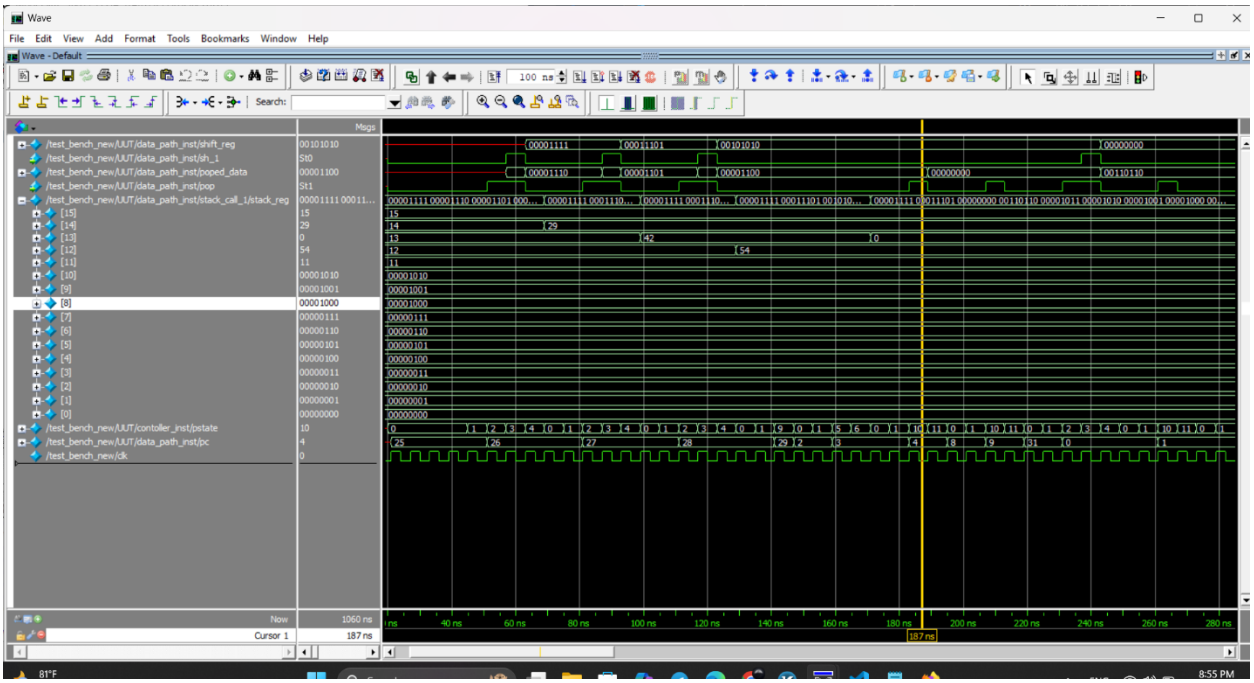
case و طبق مقدار op_code خروجی مورد نظر را به دست می اوریم. سیگنال zero را نیز بربا nor کردن bitwise مقدار عملوند اول به دست می اوریم.

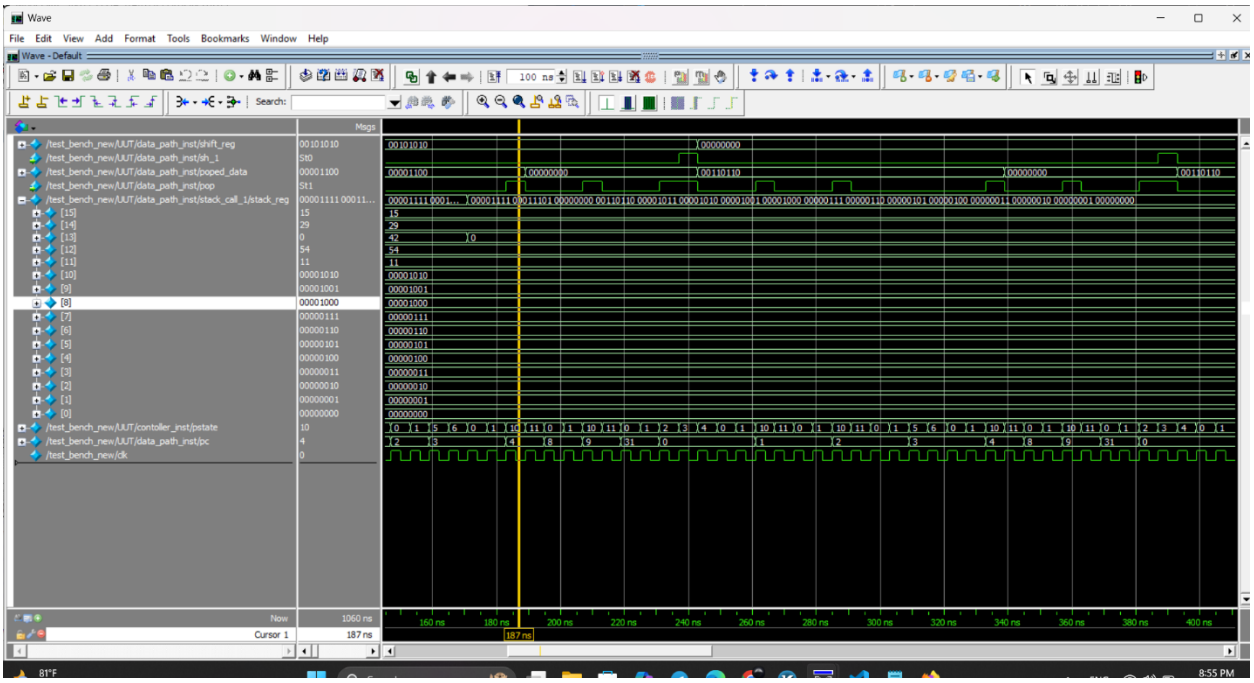
data_path : از انجایی که برای پیاده سازی ساختار پردازنده خود از ساختار هافمن استفاده کرده ایم دارای یک دیتاپت کلی و کنترلر کلی هستیم که با یکدیگر در ارتباط هستند و دیتاپت بر اساس دستورات کنترلر در زمان مناسب مقادیر مناسب را محاسبه کرده و خروجی می دهد و یا در ساختار استک و یا مموری ذخیره می کند. در ورودی دیتاپت سیگنال های کنترلی که توسط کنترلر اماده شده اند را داریم و با توجه به مقادیر انها در دیتا پت خروجی و تغییرات مناسب را ایجاد می کنیم.

روند کلی و هدف از بخش های مختلف کد به شرح زیر است : در ابتدا یک instance از استک می گیریم که بدین وسیله با ان در ارتباط باشیم. دیتایی که در استک بایستی Push شود یا از خروجی مموری و یا خروجی alu است پس یک Mux بر سر ورودی استک می گذاریم. همچنین یک instance از مموری داریم که با ان نیز در ارتباط باشیم. تنها چیزی که در مموری نوشته می شود ان دیتایی است که از استک پاپ شده در نتیجه همان را مستقیم به d_in مموری وصل می کنیم. ادرسی که از مموری می خواهیم بخوانیم یا ادرسی است که در دستور قبلی ان را جدا کرده و در رجیستر address ذخیره اش کرده ایم و یا از رجیستر pc به عنوان ادرس بعدی دسترسی استفاده می کنیم که این انتخاب را هم قطعا به راحتی می توانیم به کمک یک مولتی پلکسر استفاده می کنیم. خود pc هم می تواند در خود در استیت هایی می تواند خروجی alu و یا در استیت های دیگری ادرس را ذخیره کند که در استیت بعدی بتوانیم از محل حافظه شروع کرده و ادامه دستورات یا دیتا را بخوانیم. و در اخر باید یک Mux ای هم بر سر ورودی alu داشته باشیم تا بتوانیم محاسبات بخش های مختلف را همگی به کمک این یک alu هندل کنیم.

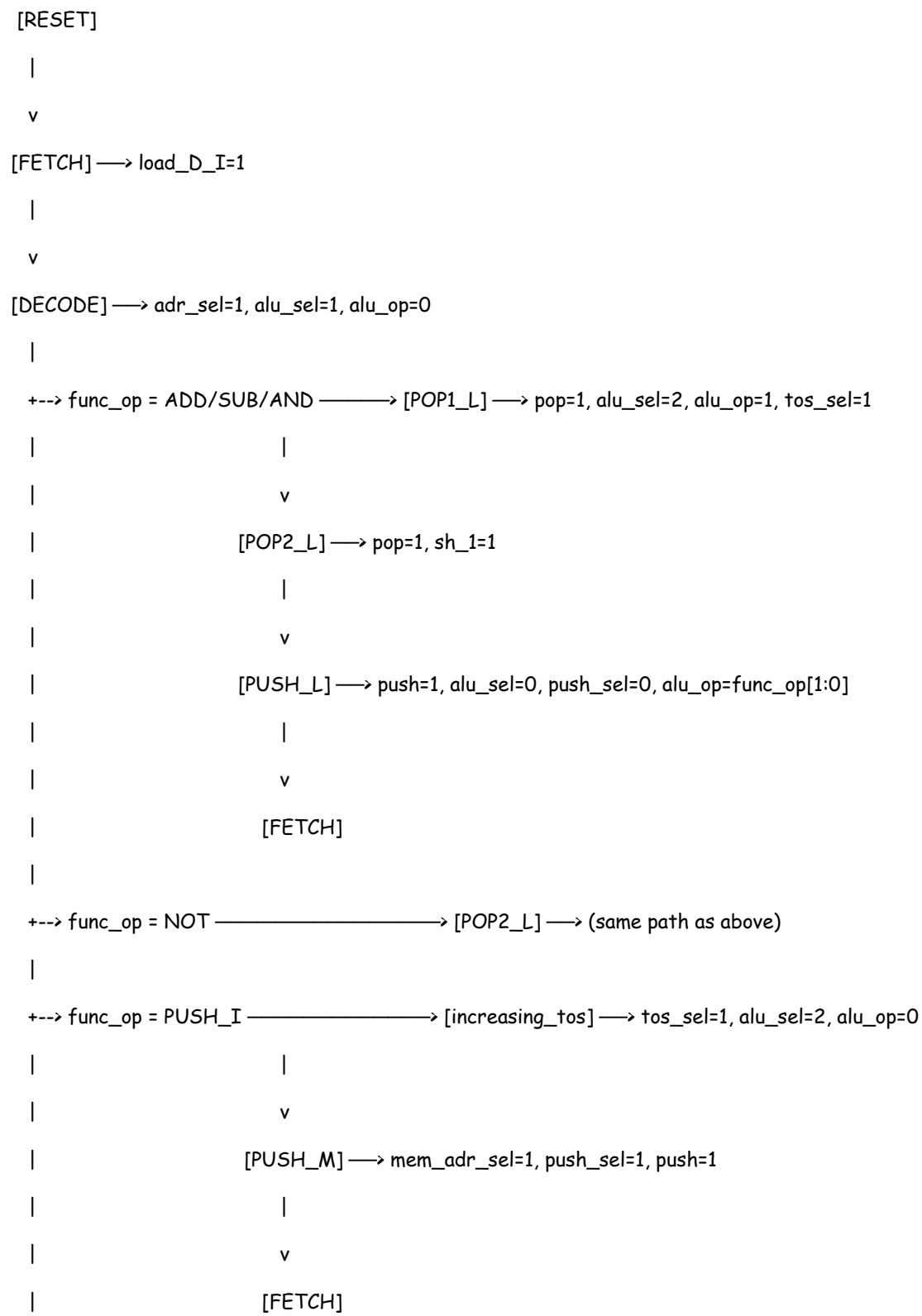
Controller : به عنوان سیگنال های خروجی تمام سیگنال های کنترلی بخش دیتا پت را داریم. دارای سه بلوک always هستیم که دو تا از انها compinational هستند که استیت بعدی را با توجه به استیت کنونی و بلوک محاسباتی بعدی سیگنال های مربوط به هر استیت را اعلام می کند. در بلوک ترتیبی هم استیت بعدی را در هر تیغه بالارونده ی کلاک به استیت کنونی منتقل می کند. شکل کنترلرهم به صورت زیر است :

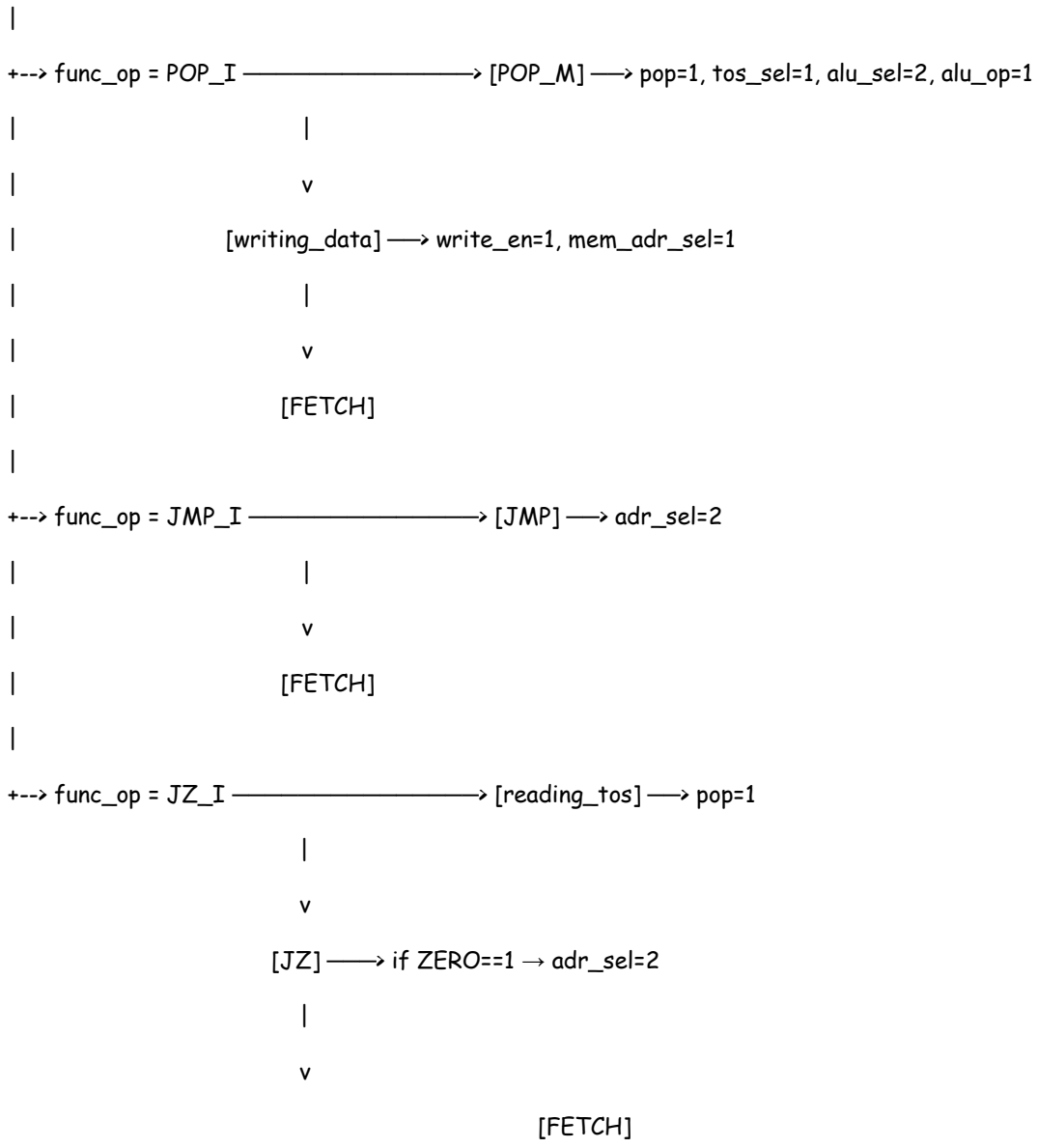
تست بنچ : تست بنچ برای مموری و استک زیر خروجی گرفته شده است. خود تاپ ماژول به صورت مستقیم خروجی ندارد بلکه تغییرات را می توانیم در رجیستر هایی که برای استک و مموری اختصاص دادیم دنبال کنیم .



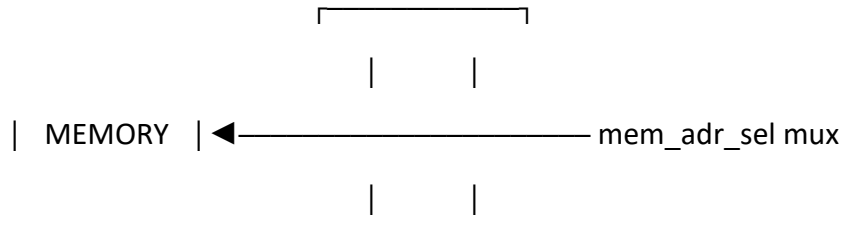


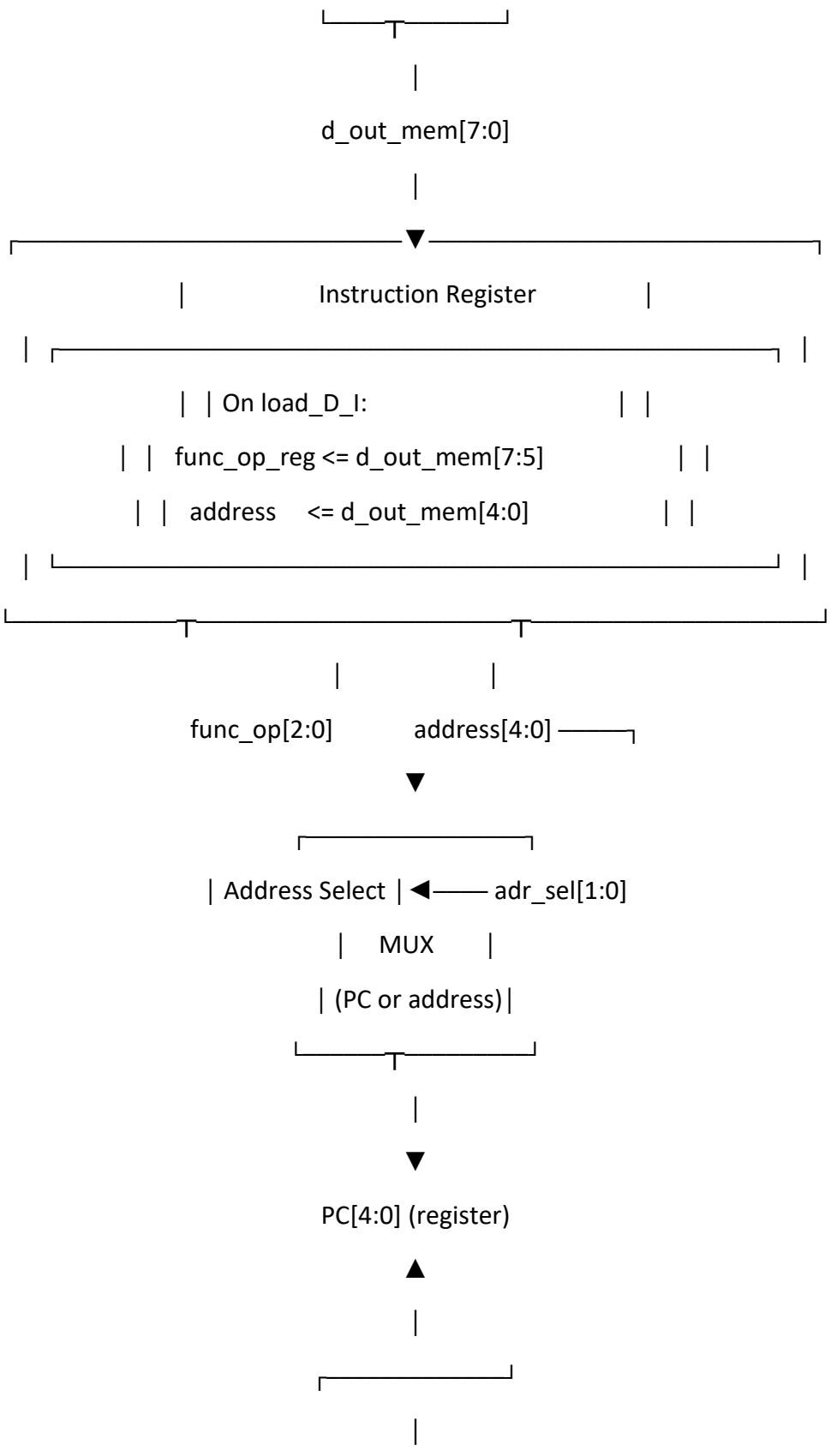
controller

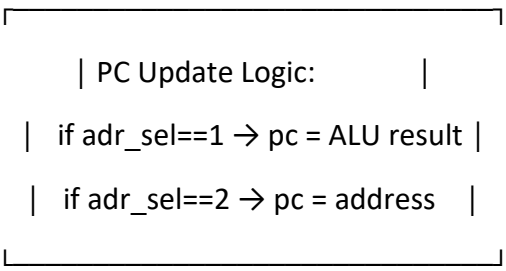

[RESET]
|
v
[FETCH] ⟶ load_D_I=1
|
v
[DECODE] ⟶ adr_sel=1, alu_sel=1, alu_op=0
|
+--> func_op = ADD/SUB/AND ⟶ [POP1_L] ⟶ pop=1, alu_sel=2, alu_op=1, tos_sel=1
| |
| v
| [POP2_L] ⟶ pop=1, sh_1=1
| |
| v
| [PUSH_L] ⟶ push=1, alu_sel=0, push_sel=0, alu_op=func_op[1:0]
| |
| v
| [FETCH]
|
+--> func_op = NOT ⟶ [POP2_L] ⟶ (same path as above)
|
+--> func_op = PUSH_I ⟶ [increasing_tos] ⟶ tos_sel=1, alu_sel=2, alu_op=0
| |
| v
| [PUSH_M] ⟶ mem_adr_sel=1, push_sel=1, push=1
| |
| v
| [FETCH]

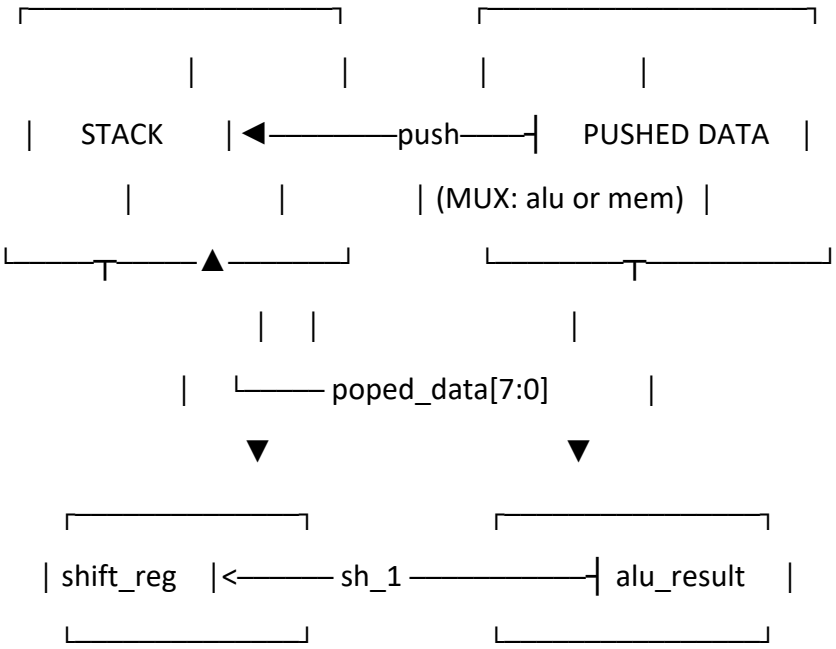

|
+--> func_op = POP_I ——————————> [POP_M] ——> pop=1, tos_sel=1, alu_sel=2, alu_op=1
|
v
[writing_data] ——> write_en=1, mem_adr_sel=1
|
v
[FETCH]
|
+--> func_op = JMP_I ——————————> [JMP] ——> adr_sel=2
|
v
[FETCH]
|
+--> func_op = JZ_I ——————————> [reading_tos] ——> pop=1
|
v
[JZ] ——> if ZERO==1 → adr_sel=2
|
v
[FETCH]

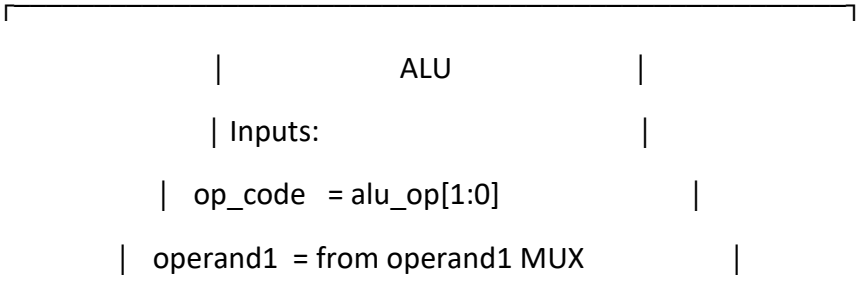
datapath

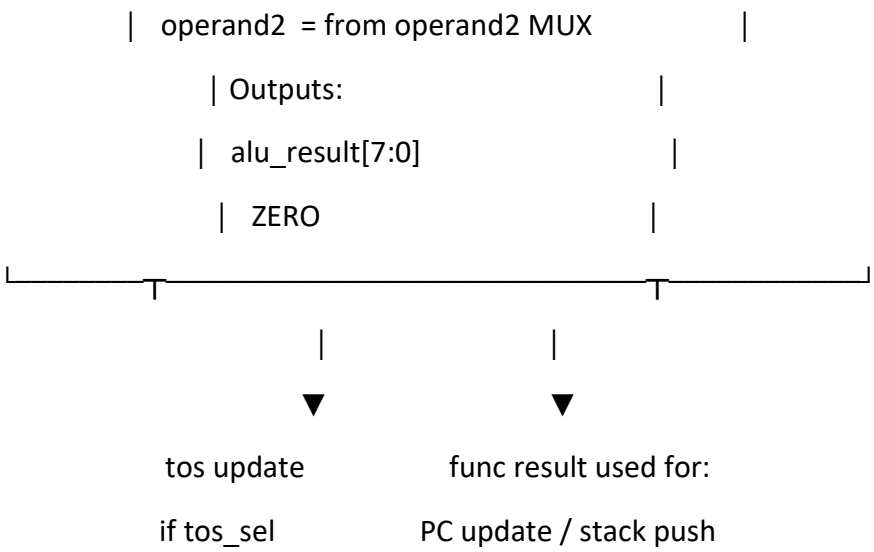

MEMORY
mem_adr_sel mux

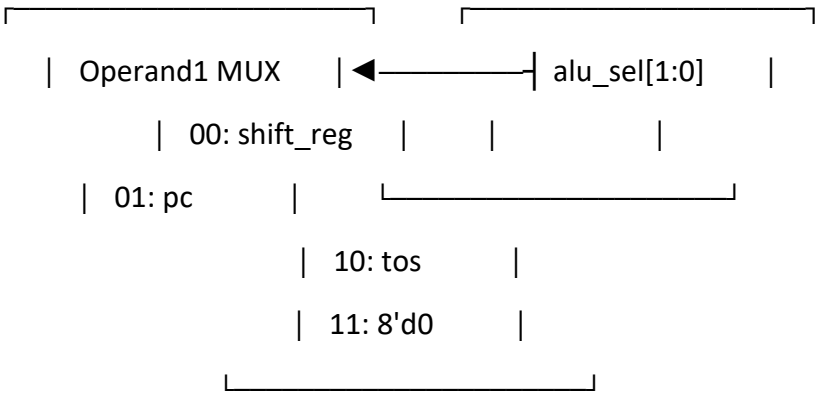

d_out_mem[7:0]
Instruction Register
On load_D_I:
func_op_reg <= d_out_mem[7:5]
address <= d_out_mem[4:0]
func_op[2:0]
address[4:0]
Address Select
MUX
(PC or address)
adr_sel[1:0]
PC[4:0] (register)

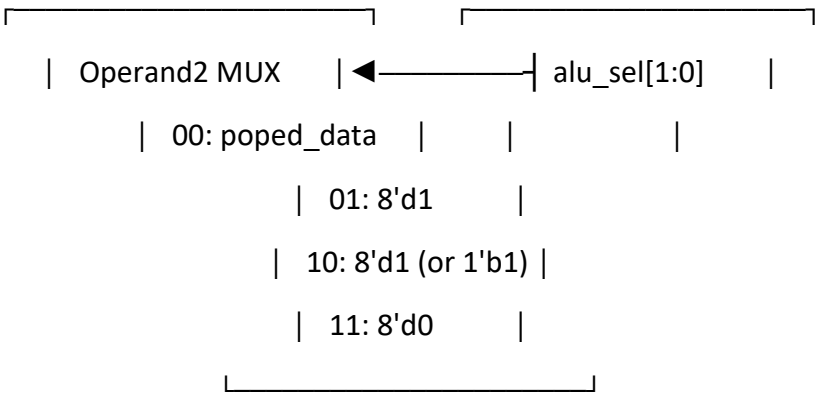

PC Update Logic:
if adr_sel==1 → pc = ALU result
if adr_sel==2 → pc = address

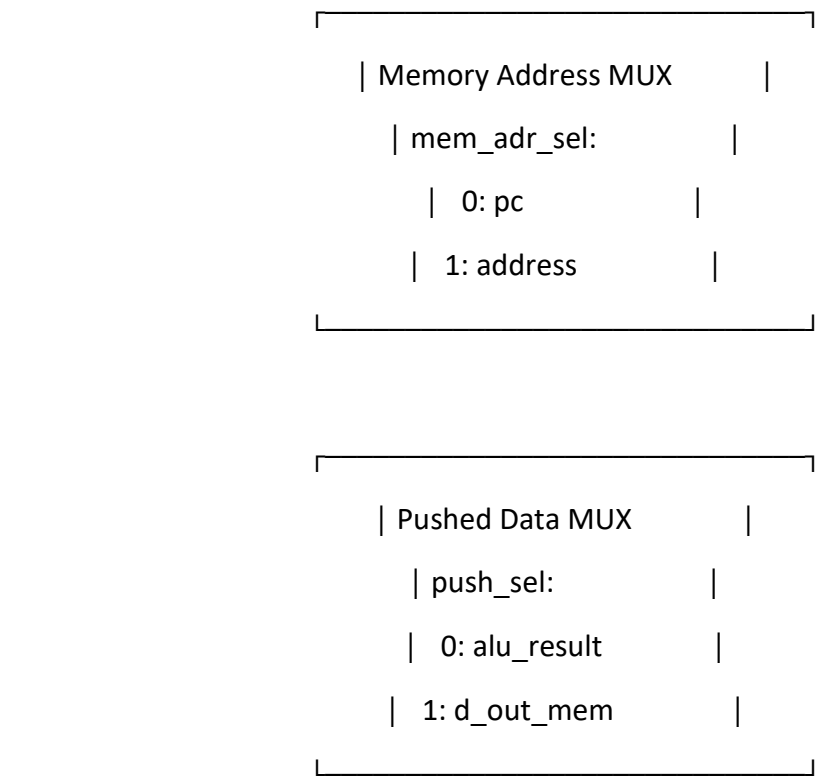

STACK
push
PUSHED DATA
(MUX: alu or mem)
poped_data[7:0]
shift_reg
sh_1
alu_result


ALU
Inputs:
op_code = alu_op[1:0]
operand1 = from operand1 MUX


| operand2 = from operand2 MUX |
| Outputs: |
| alu_result[7:0] |
| ZERO |
tos update
if tos_sel
func result used for:
PC update / stack push


Operand1 MUX
alu_sel[1:0]
00: shift_reg
01: pc
10: tos
11: 8'd0


Operand2 MUX
alu_sel[1:0]
00: poped_data
01: 8'd1
10: 8'd1 (or 1'b1)
11: 8'd0


Memory Address MUX
mem_adr_sel:
0: pc
1: address
Pushed Data MUX
push_sel:
0: alu_result
1: d_out_mem