بسم الله الرحمن الرحیم

گزارش نهایی پروژه دوم ساختار

پردازنده superscalar

محمد جوشقانی

۹۵۲۰٤٥٦۹

فهرست

الگوریتم استفاده شده و شکل کلی بلوک دیاگرام

شکل کلی بلوک دیاگرام به صورت صفحه بعد می‌باشد. الگوریتم استفاده شده به این شکل است که دستورات به صورت خارج از ترتیب و با بالاترین سرعت اجرا شده و در زمان مناسب عمل write back صورت پذیرد. مراحل پنج‌گانه اجرا به ترتیب زیر می‌باشد:

Fetch: در این مرحله ، همزمان ۸ دستور متوالی یا غیر متوالی (با توجه به تحقق branch) به ورودی فلیپ‌فلاپ رفته و به control unit تحویل داده می‌شود. مقادیر PC توسط کنترل یونیت داده شده است.

Decode: در این مرحله ، اگر منتظر دستور جدید باشیم (تمام شدن تمامی دستورها یا تحقق برنچ) دستورات جدید گرفته شده، و سپس دیکود شده و به سمت یکی از ٤ واحد عملیاتی EXE1 و EXE2 و Store و Load می‌رود. تحقق برنچ هم با ارتباط با ROB برای چک کردن آماده بودن دو طرف برنچ و در صورت نیاز استفاده از forwarding صورت می‌گیرد. مساله مهمی که در این مرحله صورت می‌گیرد این است که به همراه هر دستورالعمل، شماره ترتیبی آن نیز ارسال می‌گردد و در آینده این شماره همانند یک شناسه برای یافتن مقادیر صحیح forwarding و WB در زمان مناسب ما را یاری می‌کند.

نحوه چک کردن دستورات پرش هم به این صورت است که هر وقت یک پرش تشخیص داد شد، دستورات بعد از آن دیکود نمی‌گردند تا تکلیف آن پرش مشخص شود. هر وقت که طرفین برنچ در ROB با register file وجود داشتند سیگنال‌های مربوطه یک شده و کنترل یونیت در سریع ترین زمان ممکن آن را اعمال می‌کند، یعنی PC های بعدی را تغییر مقدار می‌دهد و دستورات جدید مطابق شکل فوق مجددا دیکود می‌گردند.

Execute: در ابتدای این مرحله ، یک بافر وجود دارد که این بافر (در صورت خالی بودن) دستورات ارسال شده از کنترل یونیت را می‌گیرد. با هر لبه بالا رونده کلاک، این بافر با کمک ROB به ترتیب از اول به آخر

دستورات را چک می‌کند که اگر آماده بودند (اینکه آماده بودن عملگر ها به چه معناست در بخش ROB توضیح داده می‌شود. با هر کلاک، بافر این بخش ۲ دستور را به سمت دو واحد اجرایی ارسال می‌کند.

Address generation: در این مرحله، مانند مرحله Executeیک بافر وجود دارد که دستورات ارسالی از کنترل یونیت را به ترتیب ذخیره کرده، با هر لبه بالارونده کلاک به ترتیب با ارتباط با ROB آماده بودن دستورات را چک کرده، آدرس ها را چک کرده و به سمت فلیپ فلاپ متصل به مموری می‌فرستد تا در کلاک بعدی از مموری خوانده یا در آن بنویسد.

نکته حائز اهمین در این بخش، اضافه شدن یک نوع hazard جدید است، به این شکل که ممکن است دو دستور save و load متوالی بخواهند در یک آدرس بنویسند که لزوما مقادیر رجیستر ها و آفست ها برابر نباشند ولی آدرس یکی باشد، در این صورت ROB باید این مشکل را با به ترتیب اجرا کردن این دستورات حل کند.

Write Back: این مرحله نیز توسط ROB انجام می‌گیرد. دستورات هنگام دیکود شدن به اطلاع این بخش می‌رسند تا بعدا به ترتیب write backصورت بگیرد.

نحوه پردازش دستورات فرمت R و I و نیز branch

دستورات نوع **R** و **I**:

این دستورات توسط بخش دیکود ، دیکود می‌شوند و در صورتی که بافرهای **ALU** و **Load**و **Save** خالی باشند این دستورات را به سمت این بافرها می‌فرستد. اگر این بافرها خالی نباشند از همانجا متوقف شده تا بافرها خالی گردند (طبیعتا این عمل لطمه ای به سرعت سیستم نمیزند چرا که وقتی بافرها پر باشند یعنی اینقدر دستورالعمل در صف وجود دارد تا بتوان دستورات مستقل را از میان آن‌ها پیدا کرد.)

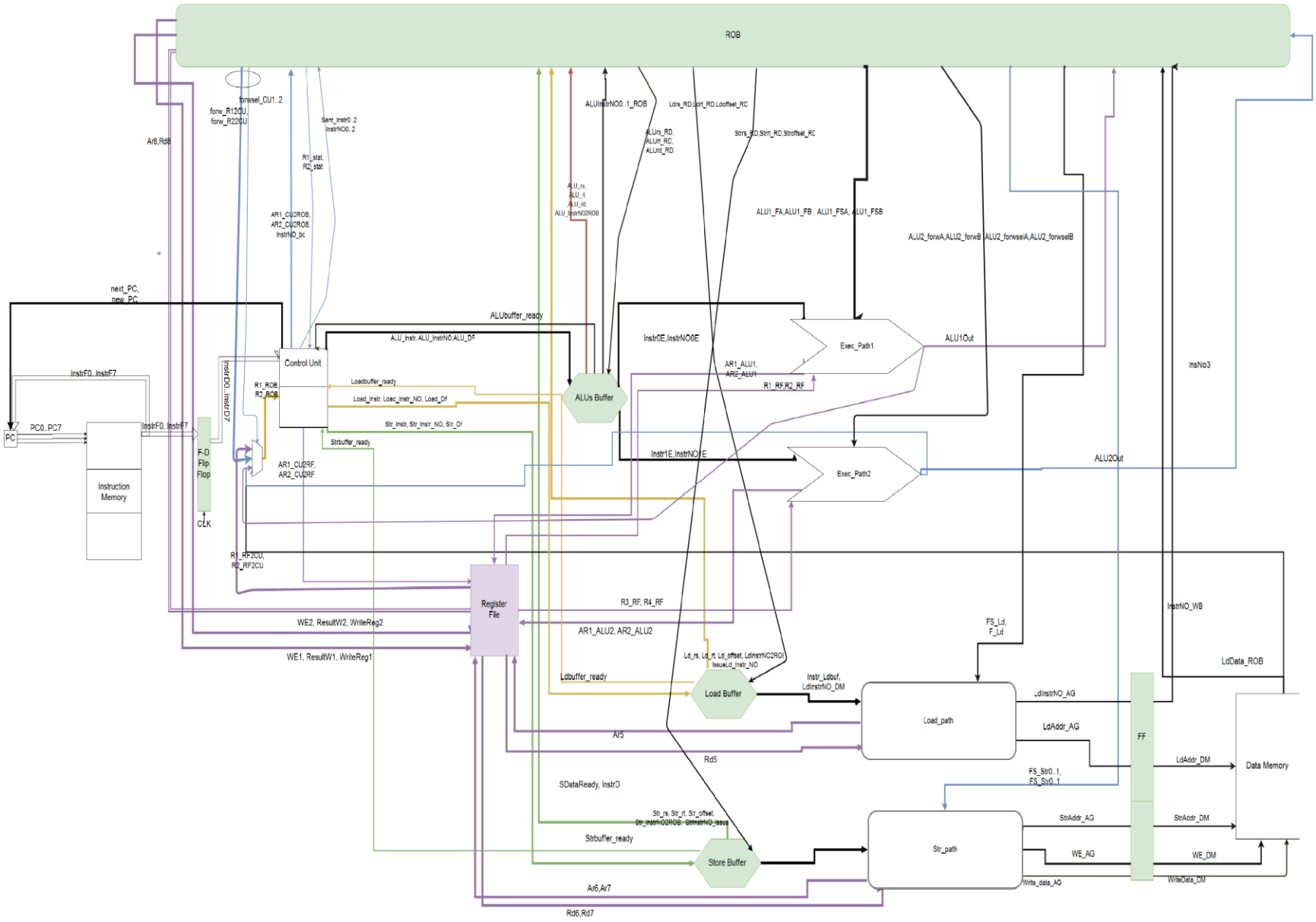
وقتی دستورات به سمت بافرها فرستاده می‌گردند، باید شماره دستورالعمل و خود دستورالعمل به **ROB**ارسال گردد تا برای به ترتیب ارسال کردن دستورالعمل‌ها و همچنین وارد کردن این شماره ها در بخش **register status** که بعدا توضیح داده می‌شود ، مشکلی وجود نداشته باشد.

دستورات پرش:

پردازش این دستورات به این نحو است که هنگامی که نوبت به دیکود کردن این دستور میرسد (دیکود به ترتیب صورت می‌گیرد) سایر دستورها دیکود نشده و کنترل یونیت با ارتباط با **ROB**، به تشخیص وضعیت دو رجیستر موجود در رجیستر ها می‌پردازد. به محض اینکه این دو رجیستر از طریق فورواردینگ و یا خود رجیسترفایل قابل دسترسی باشند **ROB** به کنترل یونیت اطلاع می‌دهد که این دو رجیستر آماده هستند و خود او مالتی‌پلکس های بیرون کنترل یونیت را راهنمایی می‌کند.

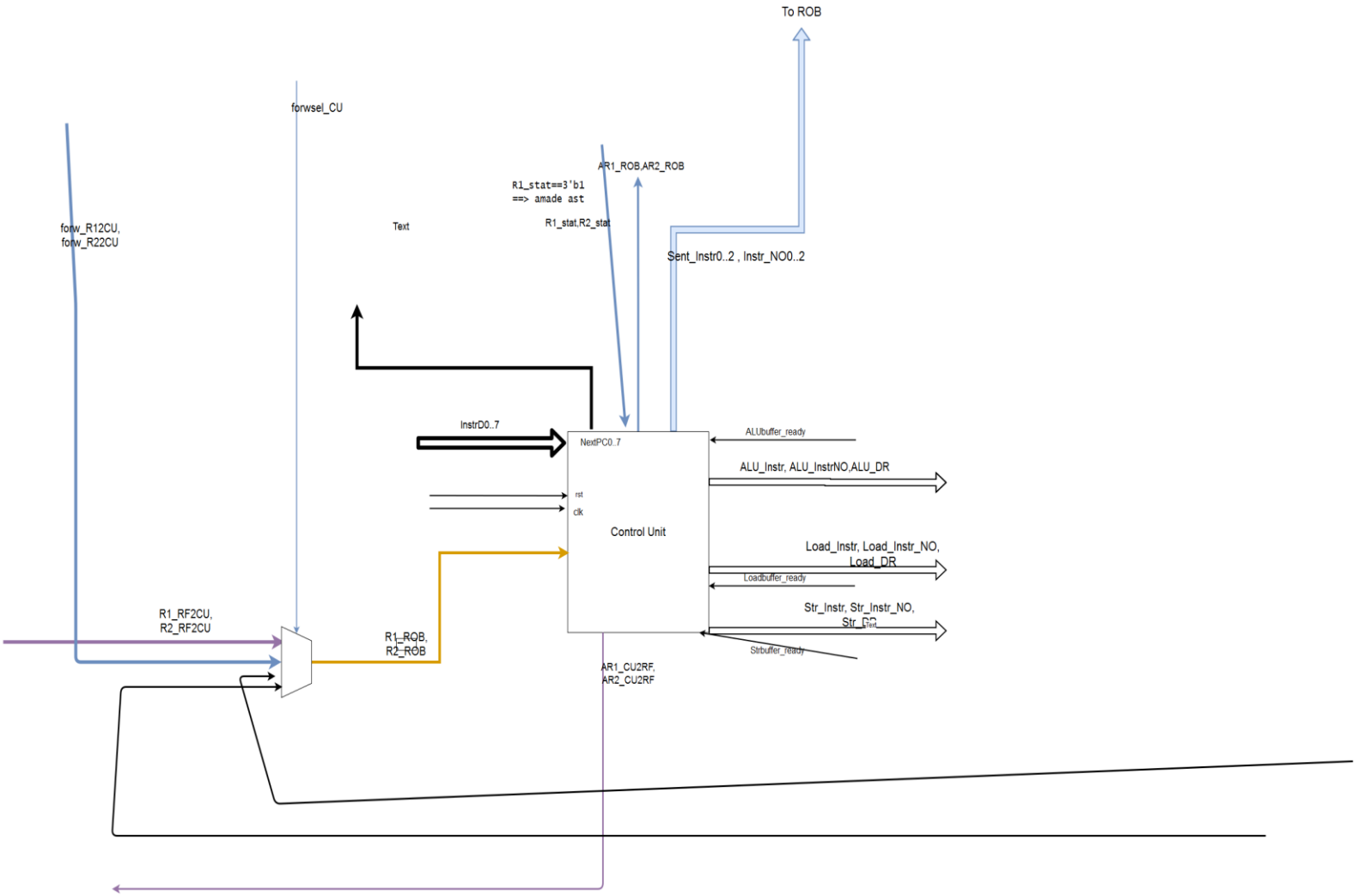
بلوک دیاگرام کامل برنامه و توضیح وظایف هر بخش

دیاگرام بلوکی پردازنده به شکل زیر است:


ROB
forwsel_CU1..2
forw_R12CU
forw_R22CU
Ar8,Rd8
Sent_Instr0..2
InstrNO0..2
R1_stat,
R2_stat
AR1_CU2ROB,
AR2_CU2ROB,
InstrNO_bc
ALUInstrNO0..1_ROB
Ldrs_RD,Ldrt_RD,Ldoffset_RD
ALUrs_RD,
ALUrt_RD,
ALUrd_RD
Strrs_RD,Strrt_RD,Stroffset_RD
ALU_rs,
ALU_rt
ALU_rd,
ALU_InstrNO2ROB
ALU1_FA,ALU1_FB , ALU1_FSA , ALU1_FSB
ALU2_forwA,ALU2_forwB ,ALU2_forwselA,ALU2_forwselB
next_PC,
new_PC
ALUbuffer_ready
ALU_Instr, ALU_InstrNO,ALU_DF
InstrOE,InstrNO0E
Exec_Path1
ALU1Out
InsNo3
Control Unit
InstrF0..InstrF7
InstrDO..InstrD7
R1_ROB,
R2_ROB
Loadbuffer_ready
Load_instr, Loac_instr_NO, Load_DF
ALUs Buffer
AR1_ALU1,
AR2_ALU1
R1_RF,R2_RF
Str_instr, Str_insrr_NO, Str_DF
Strbuffer_ready
Instr1E,InstrNO1E
Exec_Path2
ALU2Out
PC
PC0..PC7
InstrF0..InstrF7
Instruction
Memory
F-D
Flip
Flop
CLK
AR1_CU2RF,
AR2_CU2RF
R1_RF2CU,
R2_RF2CU
Register
File
R3_RF, R4_RF
InstrNO_WB
WE2, ResultW2, WriteReg2
AR1_ALU2, AR2_ALU2
FS_Ld,
F_Ld
WE1, ResultW1, WriteReg1
Ld_rs, Ld_rt, Ld_offset, LdInstrNO2ROB,
IssueLd_Instr_NO
LdData_ROB
Ldbuffer_ready
Load Buffer
Instr_Ldbuf,
LdInstrNO_DM
Load_path
LdInstrNO_AG
LdAddr_AG
FF
Ar5
Rd5
LdAddr_DM
Data Memory
FS_Str0..1,
FS_Str0..1
SDataReady, InstrD
Str_rs, Str_rt, Str_offset,
Str_InstrNO2ROB, StrInstrNO_issue
StrAddr_AG
StrAddr_DM
Str_path
Strbuffer_ready
Store Buffer
WE_AG
WE_DM
Write_data_AG
WriteData_DM
Ar6,Ar7
Rd6,Rd7

وظایف هر کدام از اجزا و بلوک دیاگرام آن به شکل زیر است:

Control unit:



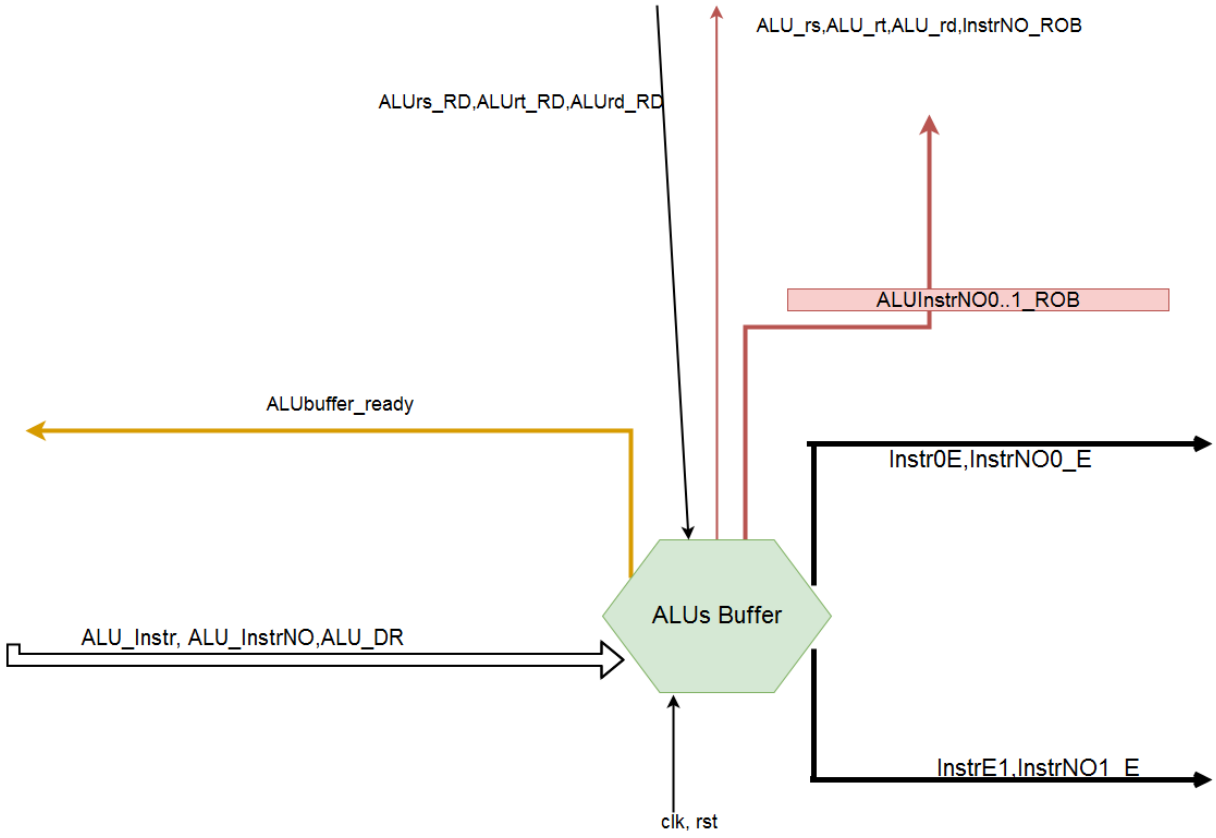
وظایف:

- دیکود کردن دستورات و ارسال آن به واحدهای مناسب
- سر پرش ها می ایستد

- هر وقت هر ۸ دستور تمام شد ۸ دستور بعدی لود می گردد. اعمال پرش هم به شکلی است که مستقل از دستورات قبل است و تغییر PC برای پرش و بعد از آن صورت می‌گیرد.
- هر ۸ تا دستور در یک forارسال میگردند، مگر اینکه یکی جا نداشته باشد یا برنچ داشته باشیم که تا سر برنچ چک میشود در هر کلاک (از اول).
- هر دستوری را که به ALU یا سایرین می‌فرستد به ROB اطلاع میدهد تا ROB آن را در لیست دستورالعمل ها بوجود آورده و وضعیت آن را بعدا در ارتباط با سایر بخش ها به روز کند.

ALUs Buffer:

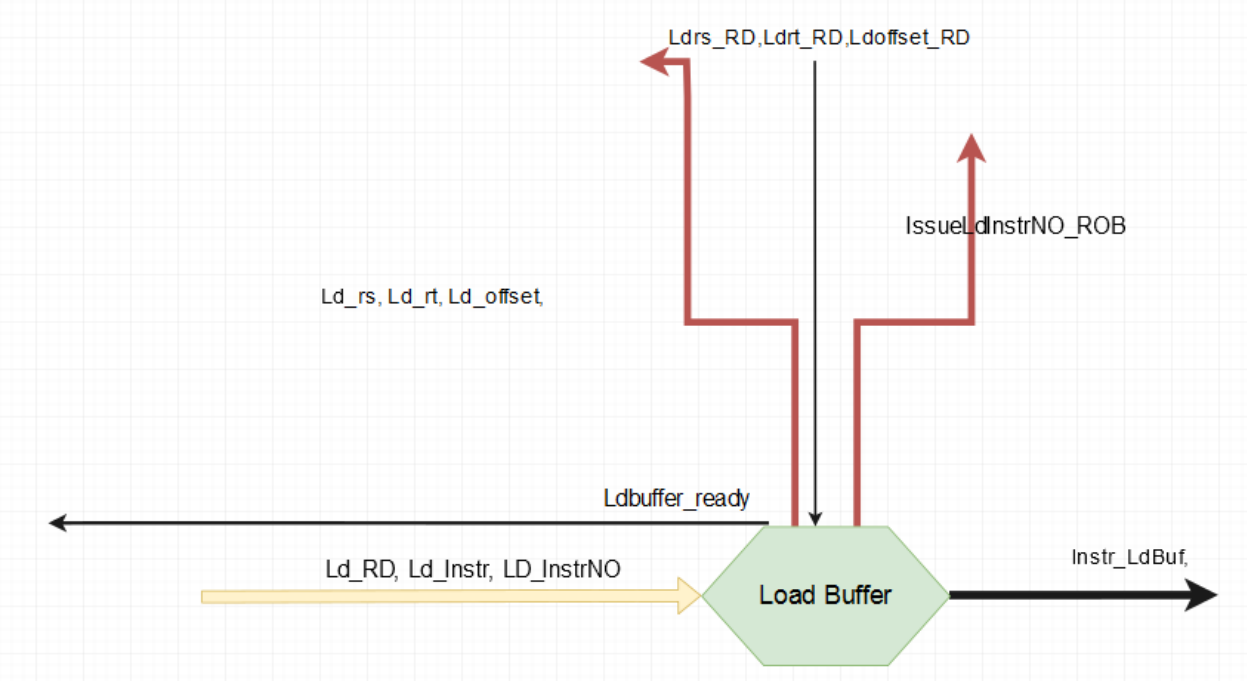
شکل آن به صورت زیر است:



وظایف:

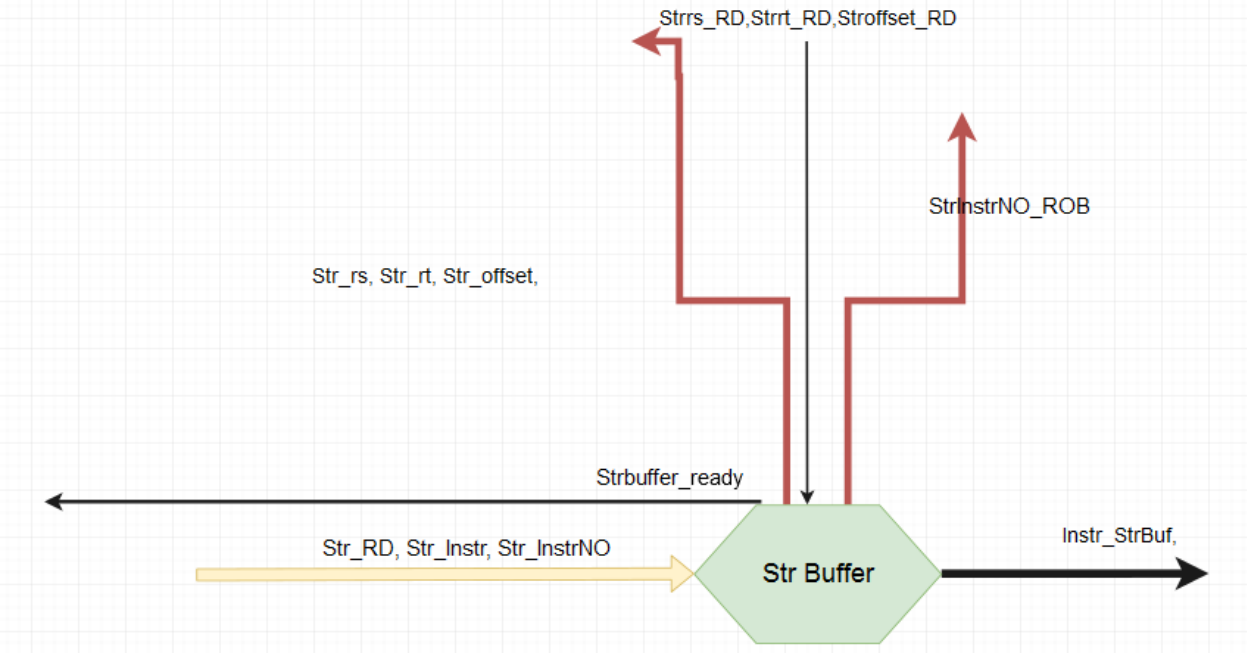
- چک کردن وضعیت آماده بودن عملگر های دستورات به ترتیب از بالا (چک کردن با ارتباط با ROB به دست می‌آید)
- با هر کلاک حداکثر دو دستور (درصورت آماده بودن) به سمت ALU ها روانه می‌شود (به ترتیب) .
- ALU Buffer با توجه به نوع دستورالعمل rs, rt, rd و addr را تشخیص داده و به همراه InstrNO به ROBمی‌فرستد و با توجه به جواب تصمیم می‌گیرد که کدام دستور را روانه مسیرهای Execution کند.
- پس از اینکه دستوری را برای ALU ها فرستاد صرفا شماره Instruction Number را به ROBاطلاع می‌دهد. همانطور که قبلا گفته شد این شماره به عنوان شناسه دستورالعمل استفاده می‌شود.
- در برنامه نوشته شده، ظرفیت این بافر ۱۵ دستورالعمل در نظر گرفته شده است.
- هر دستورالعمل به همراه شماره آن در این بافر ذخیره می‌شود.

Load Buffer



وظایف این بافر مشابه بافر ALU می‌باشد.

Store Buffer

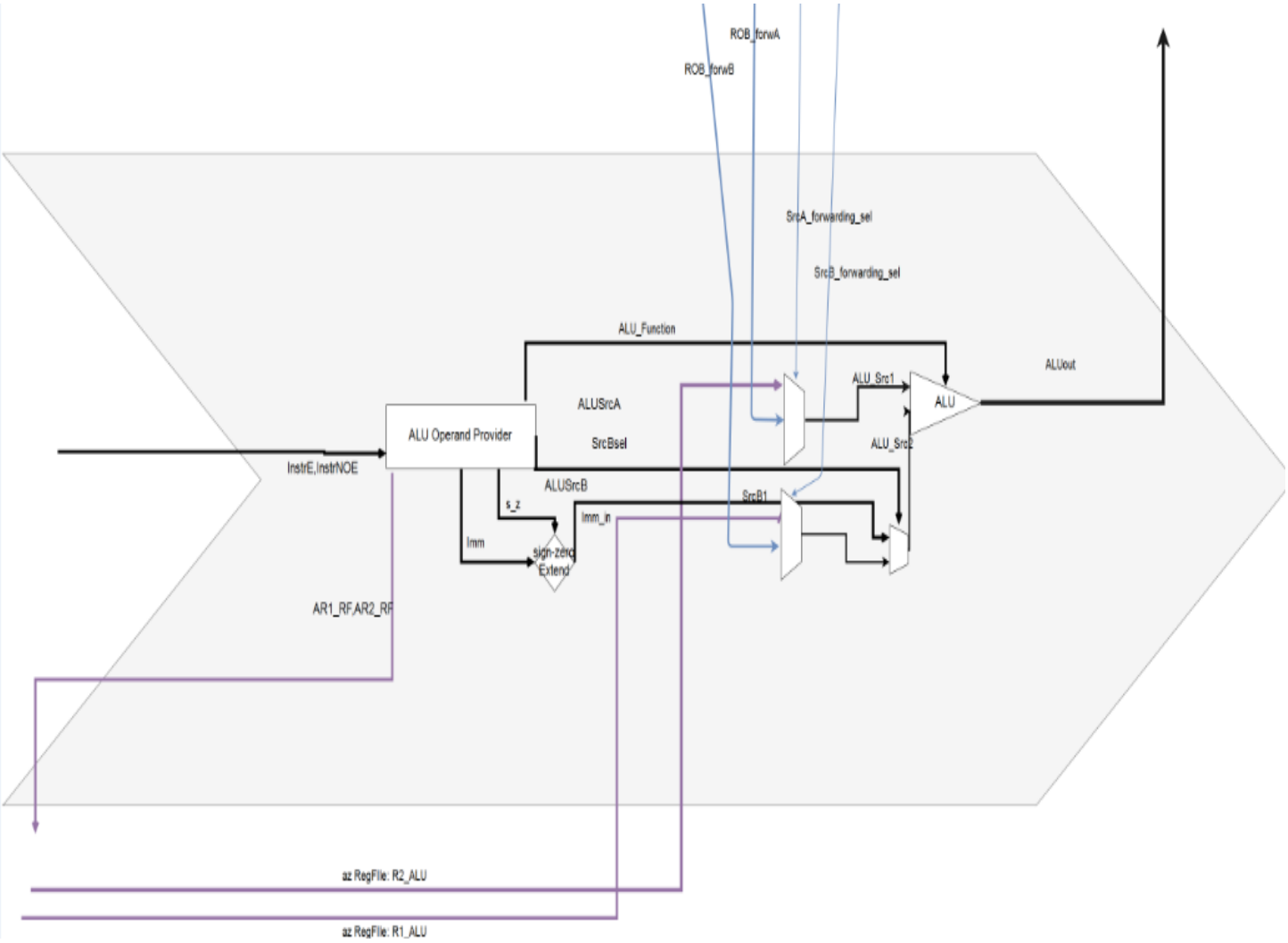


ALU Operand Provider

- در هر کلاک یک دستور برای این بخش می آید که سعی در اماده کردن آدرس برای ALU دارد.

Execution Path:

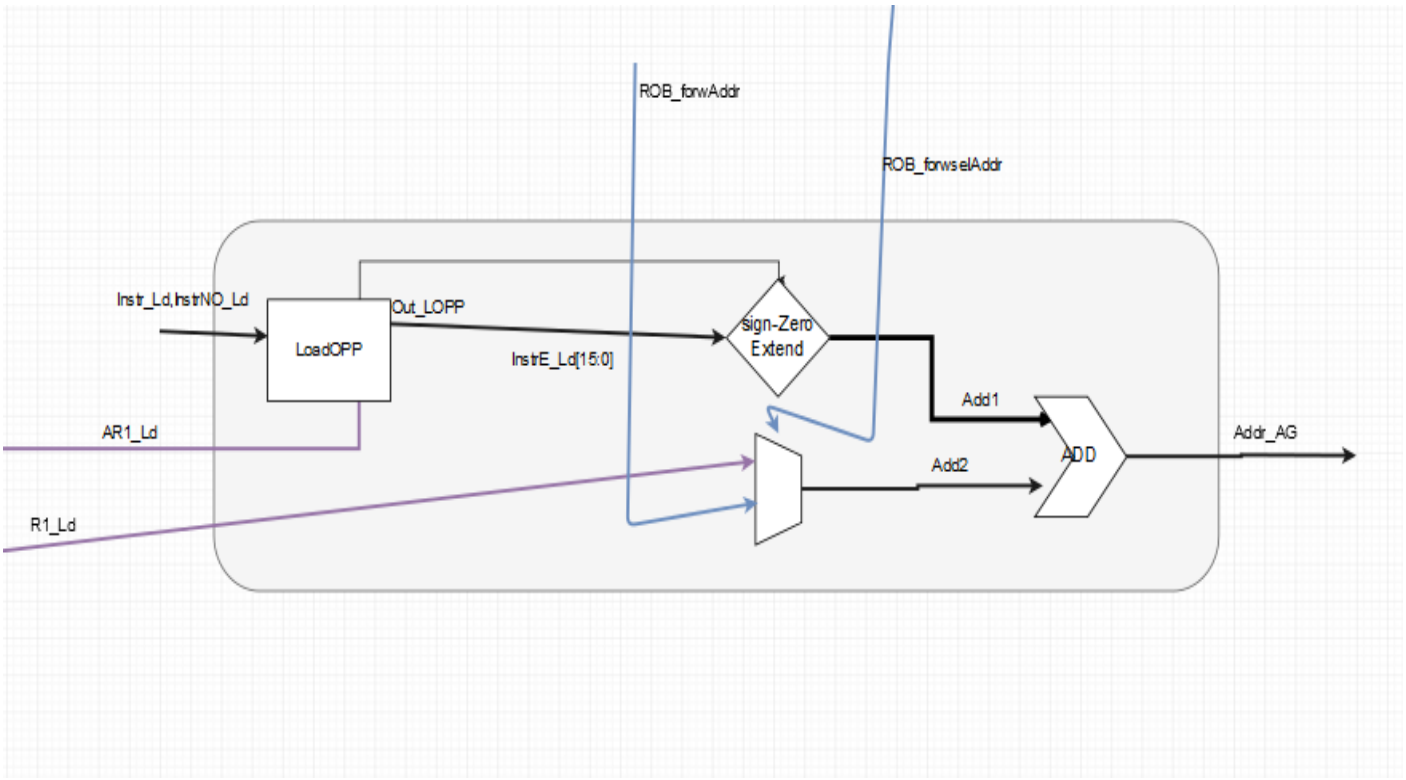
این بخش به شکل زیر است



این بخش وظیفه انجام دستورالعمل به شکل مناسب را دارد.

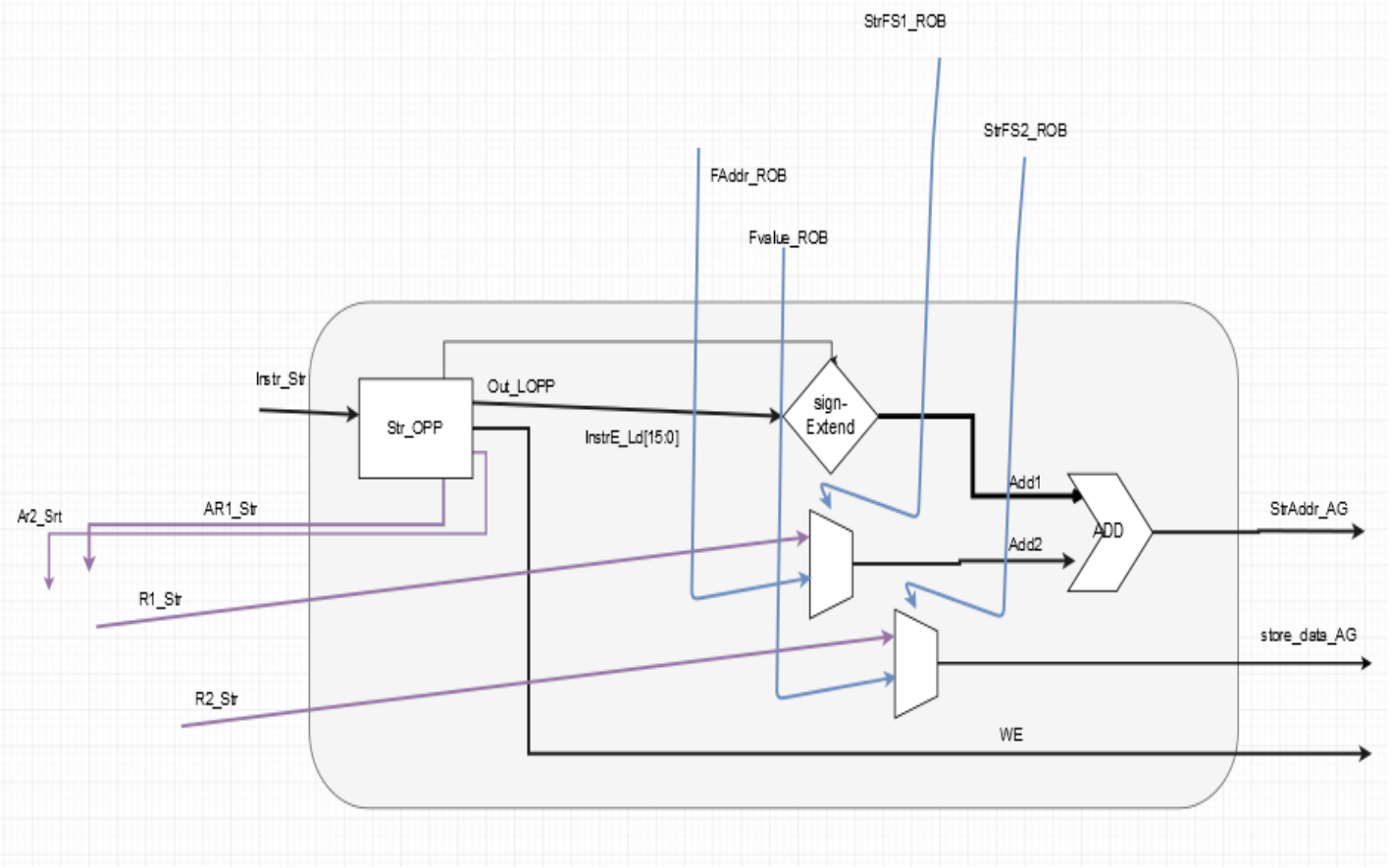
:Load Path

این بخش به شکل زیر است:



:Store Path

دیاگرام این بخش به صورت زیر است

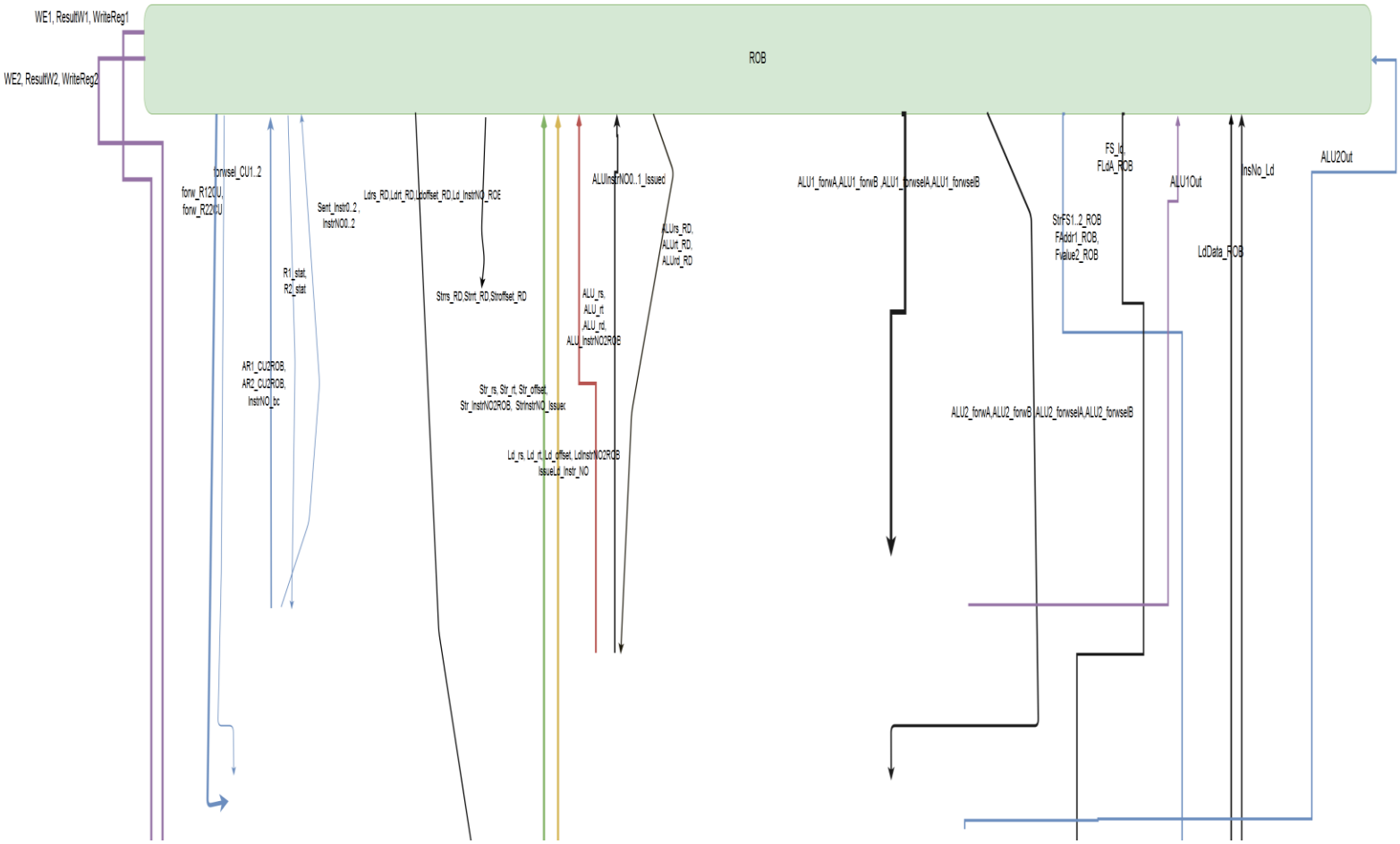


:ROB

عملکرد این بخش به صورت زیر است:

- یک رجیستر به نام Register status درون این بخش وجود دارد که شامل این است که هر رجیستر، توسط چه شماره دستورالعملی استفاده می‌شود و تخت چه عنوان (عملگر است یا قرار است در آن نوشته شود). اینکه یک رجیستر آماده است توسط بیتی در این رجیستر چک می‌شود.
- آماده بودن رجیستر ، برای عملگر به این معنی است که این دستور می‌تواند این رجیستر را استفاده کند و تغییر آن به هیچ دستوری لطمه نمی‌زند، و آماده بودن برای یک رجیستری که قرار است مقداری در آن ریخته شود این است که دستورات بالا و پایین مقادیرشان وابستکی به تغییر این مقدار ندارد.

- یک قابلیت اضافه Register status این است که وقتی یک رجیستر مقدار می‌گیرد، رجیسترهای پایین را تا جایی که عملگر هستند به روز می‌کند و مقادیر انها را به روز می‌کند لذا برای فورواردینگ بسیار مورد توجه است.
- بخش که وظیفه commitکردن دستورات را دارد، دستورات را در رجیستری به نام Commit Register قرار می‌دهد. در این رجیستر مقادیر نهایی و نیز رجیسترهایی که باید در ان نوشته شوند وجود دارد. پس از commitشدن هر دستور تمامی رجیستر های Register Status و غیره شیفت به بالا پیدا می‌کنند.
- شکل ROBبه صورت زیر است

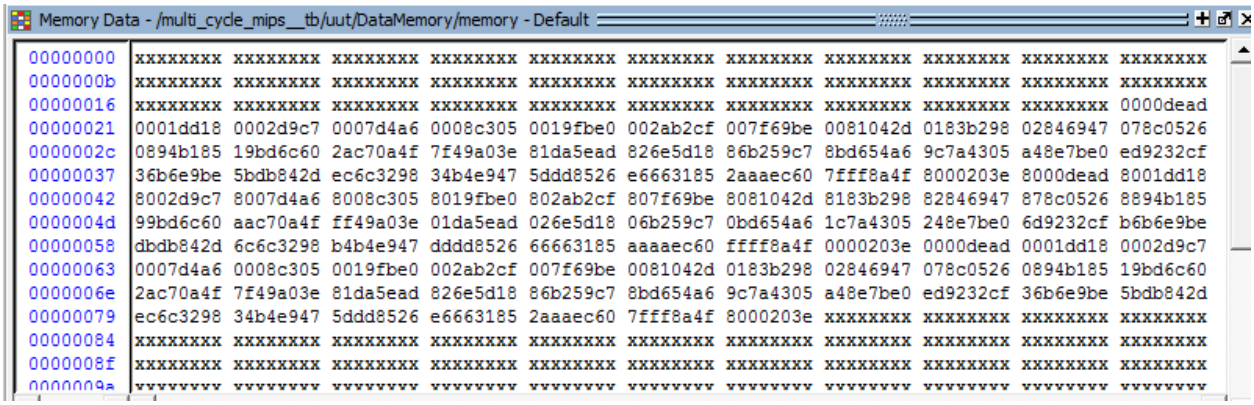


نتایج شبیه سازی

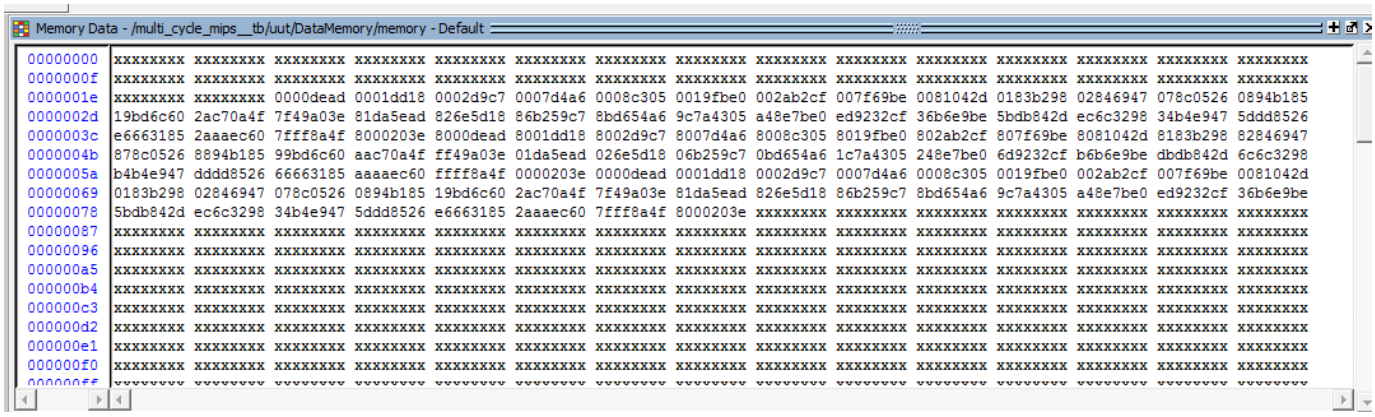
نتایج شبیه‌سازی در دو بخش مورد بررسی قرار می‌گیرد:

بخش اول: اجرای برنامه تا قبل از شروع sort کردن اعداد تولید و ذخیره شده در حافظه

خروجی برنامه تا این حالت مطابق شکل زیر است. قابل ذکر است این مرحله ۲۸۹۰ نانو ثانیه به طول می‌انجامد



مقایسه این بخش با حالت پایپ لاین: در پایپ لاین این برنامه حدود ۷۷۰۰ نانوثانیه به طول می‌انجامد.



نتیجه مقایسه این بخش: این قسمت از برنامه با سرعت ۲,۶۶ برابر حالت پایپ‌لاین عادی اجرا می‌گردد.

بخش دوم: اجرای کامل برنامه

در این بخش خروجی برنامه به شکل زیر است:

اجرای کامل برنامه حدود ۱۹۹ نانوثانیه به طول می‌انجامد که حدود ۱,۹۶ برابر بهبود در سرعت اجرای پایپ‌لاین عادی است.

قابلیت های بعدی (شامل قابلیت اضافه شدن cache و branch prediction)

در مورد قابلیت اضافه شدن **cache**، طراحی اولیه این برنامه همراه کش بود اما به دلیل ساده تر شدن کار این کش حذف شد. برای این کار صرفا نیاز به استفاده از **icache** داریم تا دستورات یک کلاک زودتر در اختیار ما قرار بگیرند. این قابلیت وجود دارد تا زیر کنترل یونت این **cache** اضافه گردد.

قابلیت **branch prediction** در این برنامه وجود ندارد اما جای آن بسیار خالی است و قطعا قابلیت اضافه شدن را دارد. اگر از **icache** استفاده کنیم این واحد می‌تواند کنار کنرل یونیت مشغول به کار شود.

اشتباهات و نکات آموزنده

در مورد نکات آموزنده، مهم ترین نکته این است که برای کارهای وسیع این چنینی حتما از دیاگرام بلوکی واضح و صریح استفاده شود تا از سردرگمی و فراموشی جلوگیری گردد. همچنین تا جای ممکن در برخی کارها که

ارتباط بین دو واحد زیاد است، اگر این دو واحد درون یک بخش قرار دارند منطقی است که این دو در دو ماژول جداگانه نوشته نگردند، چرا که برای صحت دریافت و ارسال سیگنال‌های صحیح نیاز به استفاده از تاخیر می‌گردد که می‌تواند روند اجرای برنامه را مختل کند.

در مورد اشتباهات انجام شده به طور خیلی خلاصه، اشتباهات زمانبر عبارت بودند از:

- با توجه به حساس بودن وریلاگ به بزرگی و کوچکی حروف، بسیار مختمل است که یک وایر که برای ارتباط بین دو ماژول است را سهوا تعریف نکرده باشیم و یا اسم آن را اشتباهی با یک حرف بزرگ و کوچک جابجا نوشته باشیم، در این حالت متاسفانه کامپایلر، این متغیر را یک وایر یک بعدی در نظر گرفته، که در حالتی که در ماژول کپی شده هم این متغیر یک بعدی باشد بدون هیچ خطا و هشداری برنامه شبیه سازی شده و مشغول به کار می‌شود.
- بهتر است برای هر چیزی بدترین حالت در نظر گرفته شود، مثلا برای Reg_stat اشتباها در ابتدا برای هر رجیستر ۸ خانه (مربوط به ٤ دستورالعمل) در نظر گرفته شده بود که مشکلات زیادی را به وجود آورد. با توجه به اینکه ۸ دستورالعمل حداکثر از یک رجیستر ٢٤ بار استفاده می‌کنند و ٤٨ خانه نیاز است ، لذا ٦٤ خانه برای هر رجیستر اختصاص داده شد.
- در یک حلقه ای که تعداد خاصی انجام می‌شود، اگر تاخیر استفاده می‌شود خیلی مهم است که این تاخیر ها طوری تنظیم شوند که قبل از تحریک مجدد بلاک always این حلقه تمام شده باشد.
- استفاده از متغیرهای یکسان در بلاک های always مختلف ایجاد خطای ران تایم می‌کنند.