2022

# Computer Architecture

PHASE 3


Members:

Hossein Goli

Erfan Sadraiye

Amirhossein Barati

Alireza Foroodnia

# مقدمه :

در این فاز همانگونه که خواسته شده بود پردازنده خود را از single cycle به طراحی pipeline درآوردیم. در طراحی از معماری معروف 5 stage pipeline استفاده کردیم. در نتیجه در بهترین حالت speed up پایپلاین ما برابر 5 میشود. همچنین رفع data dependacny ها در پایپلاین وظیفه کامپایلر با قرار دادن nop است.

## کلیات پیاده سازی:

در پیاده سازی ما ماژول buff که شامل تعدادی بلوک رجیستری است را اضافه کردیم که درواقع باید دیتا های هر stage را در خود نگه دارد. بافر ها شامل:

IF/ID: Instruction fetch => Instruction decode

ID/EXE: Instruction decode => Execution unit

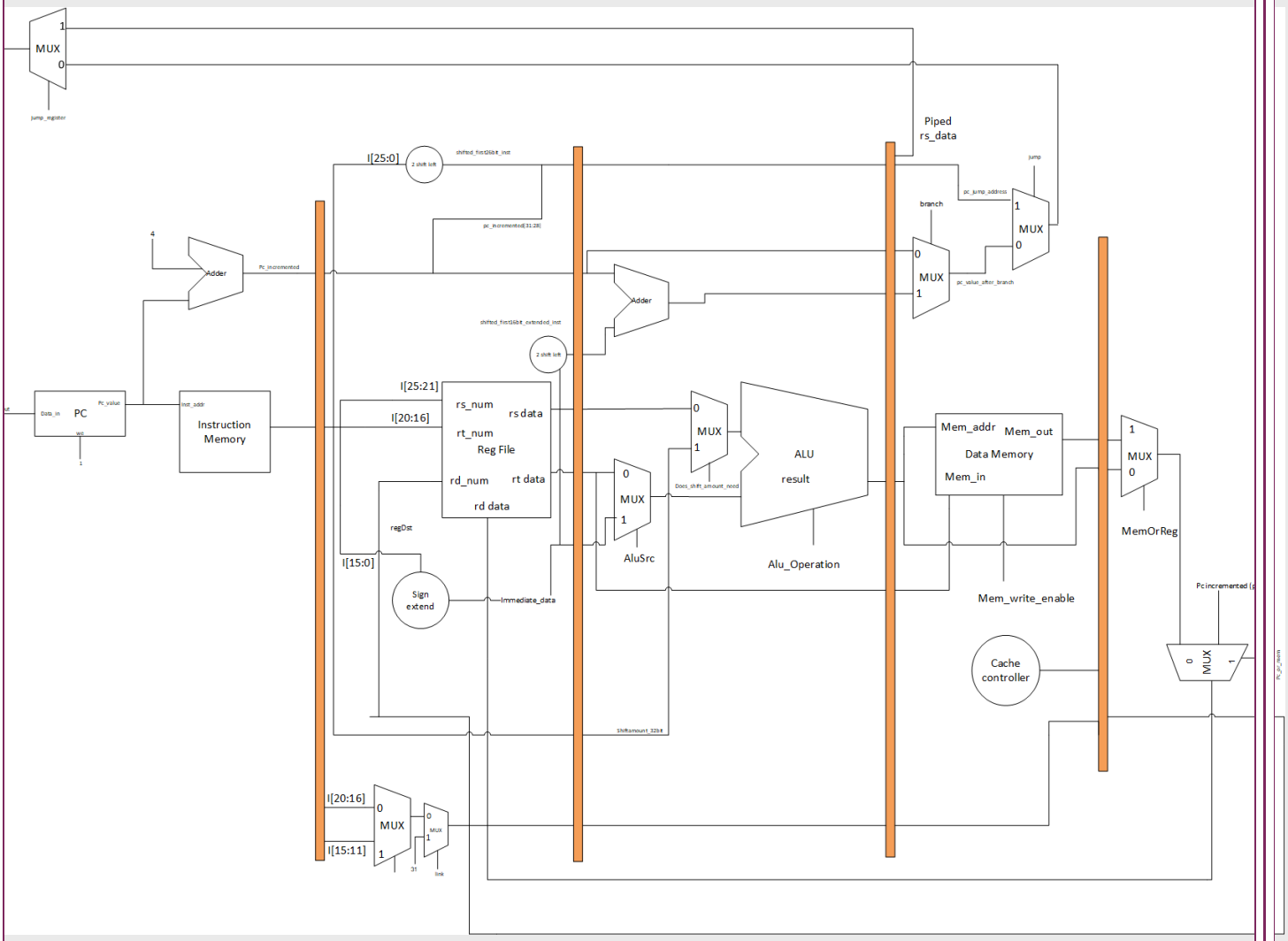
EXE/Mem : Execution Unit => memory unit

Mem/WB : memory unit => write back unit

می شوند. همچنین تغییرات لازم در مسیر داده و واحد کنترل را اعمال کردیم تا بتوانند سیگنال های کنترلی هر مرحله را به درستی تنظیم کنند و هر مرحله مستقل از مرحله های دیگر کار کند.

# جزئیات پیاده سازی:

ماژول datapath :




طراحی datapath مطابق با شکل بالا تغییر کرده است تا بتواند 5 مرحله پایپلاین را پیاده سازی کند. همانطور که مشخص است در این نوع پایپلاین ما فقط ممکن است True dependacny داشته باشیم که همانطور که در مقدمه گفته شد با nop آن را رفع میکنیم. همچنین لازم است که چندین سیگنال جدید به مسیر داده اضافه شود. (برای مثال instruction که در هر مرحله در bufferها وجود دارد).

```
module data_path (
    inst,inst_addr , reg_dest, reg_write_enable, alu_src, alu_operation,
mem_addr, mem_data_in,mem_data_out,
    mem_or_reg,clk,halted,rst_b,branch,jump,jump_register,pc_or_mem,does_shift
_amount_need,zero,negative,
    is_unsigned,pc_we,flush,stall,reg_write_enable_cache,inst_ID, inst_MEM,
inst_EX, should_branch);
```

## ماژول Buff:

وظیفه این ماژول صرفا قرار دادن یک سری buffer در مسیر داده است تا بتوانیم سیگنال های کنترلی مورد نیاز و مقادیر مسیر داده که لازم است تا مراحل بعدی pipe شوند را نگه داریم. برای مثال buff IF/ID مطابق زیر است:

```
module buff_IF_ID(pc_incremented_IF,pc_incremented_ID, inst_IF,inst_ID, clk,
rst_b, flush, stall);
    output [31:0] pc_incremented_ID;
    output [31:0] inst_ID;
    input [31:0] pc_incremented_IF;
    input [31:0] inst_IF;
    input flush, stall, clk, rst_b;
    Register pc_inc_reg (.clk(clk), .rst_b(rst_b),.data_in(pc_incremented_IF),
.data_out(pc_incremented_ID), .we(~stall));
    Register instruction_reg (.clk(clk), .rst_b(rst_b), .data_in(inst_IF),
.data_out(inst_ID), .we(~stall));
endmodule
```

طراحی بقیه buffer ها نیز به همین صورت با سیگنال های متفاوت است.

# برخی نکات پیاده سازی

### Rd_num:

در پیاده سازی ما لازم داریم تا برخی از سیگنال ها را تا مرحله آخر برای جلوگیری از bug پایپ کنیم. یکی از این موارد مهم rd_num است. چون مرحله write back در آخرین مرحله پایپلاین قرار دارد نوشتن در reg file در مرحله آخر قرار میگیرد و ما لازم داشتیم تا لاجیک decode کردن rd_num را از مرحله ID تا مرحله WB پایپ کنیم.

### Branching:

دقیقا همانطور که بالا بحث شد به علت اینکه branch و jump در مرحله 4 که همان مموری است انجام میشود ما لازم داریم تا PC+ 4 را نیز تا این stage پایپ کنیم. همچنین چون برای branchنیاز به opcode داریم تا بدانیم کدام یک از دستورات (BGE, BEQ ,...) است نیاز به پایپ کردن دستور نیز داریم.

## Stalling Cache for memory Latency:

همانطور که در فاز 2 بحث شد مموری ما latency دارد و ما برای کاهش این latency از کش استفاده کردیم. اما در این فاز چون پایپلاین قرار گرفته است و یک خط لوله مداوم با دستورات پر میشود ما نمیتوانیم در مرحله MEM چندین کلاک صبر کنیم و دستورات همچنان وارد پایپلاین شوند.

در نتیجه ایده ما برای حل این مشکل stall کردن مراحل قبلی پایپلاین (با خاموش کردن enable های بافر ها و pc) بود تا وقتی که در مرحله mem ، hit رخ بدهد یا اصلا دستور کار با مموری نباشد .

با این ایده مشکل memory latency و پایپلاین را حل کردیم. برای همین منظور یک سیگنال stall به طراحی خود اضافه کردیم که به stall~ همان enable بافر ها و pc ما است.

# نتیجه شبیه سازی:

ما علاوه بر 8 تست داده شده ، 12 تست دیگر نیز نوشتیم و در آن ها مطابق خواسته فاز 3 در cw با قرار دادن nop ، وابستگی داده ها را حذف کردیم . همانطور که مشاهده میکنیم تمامی این 20 تست پاس میشوند: