.1

آ) من توی شکل *Data* های مشترک و زمان ساخت و استفادشون رو علامت زدم (خونه هایی که توشون پر شده داده های درست رو دارن و خونه هایی که دورشون دایره مشکی داره داده رو نیاز دارن):

Instructions	Cycles															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
MOVI R1, X	F	D	E1	E2	E3	M	W									
MOVI R2, Y		F	D	E1	E2	E3	M	W								
MUL R4, R1, R1			F	D	-	E1	E2	E3	M	W						
MUL R1, R1, R2				F	-	D	E1	E2	E3	M	W					
ADD R4, R5, R6						F	D	E1	E2	E3	M	W				
ADD R5, R2, R4							F	D	-	-	E1	E2	E3	M	W	
SUBI R3, R1, 2048								F	-	-	D	E1	E2	E3	M	W
JNZ L1											F	D	-	-	E1	...
...	...															

همانطور که در تصویر مشاهده میکنید، *4* کلاک مستعد *Forwarding* داده هستند. در دستوراتی که از *Data* رجیستر ها استفاده میکنند، یا باید داده در مرحله *Decode* که داده ها از رجیستر خوانده میشوند، به ما برسند(یا *Write back* قبلا انجام شده باشه یا داده پاس داده شود) و یا باید در کلاک بعد به *E1* پاس داده شوند. پس اگر فرض کنیم که *Data Forwarding* بین قسمت های مرحله فعلی که داده را دارند انجام میشود، داریم:

- میان *E1, M* *Data Forwarding*وجود دارد چون در کلاک *6* ام به *R1* ، در کلاک *7* ام به *R2* و در کلاک *11* ام به *R4* نیاز داریم.
- میان *E1, W* و یا *D,M* *Data Forwarding*وجود دارد چون در دستور *4* ام به *R1* نیاز داریم و این نیاز را یا در کلاک *7* ام و یا در کلاک *6* ام باید هندل کنیم.
- میان *D,W* *Data Forwarding*وجود دارد (اگر *latch* داشته باشیم نیازی نیست) چون در کلاک *8* ام به *R2* و در کلاک *11* ام به *R1* نیاز داریم.

پ.ن: در دستور آخر پرش به پرچم *z* نیاز دارد ولی من معتقدم که این فلگ در مرحله *E3* در کلاک *14* ام ست شده است و نیازی به چک کردن دستی *R13* و *Data Forwarding* برای آن نیست...

ب) این پردازنده از *interlocking* به صورت سخت افزاری استفاده میکند زیرا *Hazard* هارا با *stall* و *Data Forwarding* (و حتی شاید با *latch*) حل کرده است (اگر با لحاظ کردن تغییراتی در کد مانند استفاده نکردن دستورات وابسته، استفاده از *no-op* و یا *code reordering* آن هارا هندل کرده بود، نرم افزاری حساب میشد)

ج) چون *X=4* است پس در ابتدای این کد *R1* برابر با *4* شده است و سپس وارد حلقه ای شده است که با ضرب *R2* (که چون *Y=4* بوده و در جای دیگر تغییری نکرده است، درین کد برابر با *2* است) در آن، آن را تغییر میدهد و با رسیدن آن به *2048* از حلقه خارج میشود. برای اینکه $R1=1024 = 4 \times 2^8$ شود باید *8* بار *MUL R1, R1, R2* اجرا شود. همچنین از آنجا که در حال واکشی *MUL R4, R1, R1* هستیم، پس وارد دور بعدی حلقه و اجرای *9* ام آن شدیم. اولین واکشی دستور گفته شده در کلاک *3*ام است و دومین واکشی آن در کلاک *12* ام است. اما در اجرای دوم حلقه بخاطر استفاده دستور *JNZ* از *ALU* ، *2* کلاک کند تر خواهد شد و از طرفی چون دو دستور اول برنامه قبلا اجرا شده اند، *stall* کلاک ششم اجرا نخواهد شد، پس ازین به بعد *10* کلاک یکبار انجام میشود. پس: *T=12 + 7*10= 82*

د) *2* دستور قبل از حلقه، *8* بار اجرای کامل حلقه (*6* دستور در هر حلقه) و یک بار دستور اول حلقه : *N=2+8*6+1=51*

ه) برای خارج شدن از حلقه باید *R1* برابر *2048* شود که یعنی از زمانی که *R1=1024* است، تنها یک بار دیگر باید حلقه انجام شود. پس در چرخه *T+10* دستور آخر واکشی میشود و برای اجرای کامل آن بخاطر *2* تا *stall* دستور قبل و وابسته نبودن داده خودش به *8=6+2* کلاک دیگر نیاز داریم. پس کلاک *100 = 8+10+82* ام آخرین چرخه برنامه است.

2. میدانیم *throughput* به تعداد دستورات یا عملیات‌هایی اطلاق می‌شود که پردازنده قادر است در یک واحد زمان انجام دهد. پس برای بیشینه شدن آن کافیست زمان هرکلاک را کمینه کنیم:

$$argmin\, T_p = argmin\, \frac{20}{n} + 0.05n \xRightarrow{\frac{d}{dn}} -\frac{20}{n^2} + 0.05 = 0 \Rightarrow n^2 = 400 \Rightarrow n = 20$$

$$Throughpu_{max} = \frac{number\ of\ cores}{latency} = \frac{1}{1 \times \left(\frac{20}{20} + 0.05 \times 20\right) \times 10^{-9}} = 500Mhz$$

3. بدون دور زدن:

Instruction	1	2	3	4	5	6	7	8	9	10	11	12
Sub $2, $3, $1	F	D	X	M	W							
Lw $5, 0($2)		F	d*	d*	D	X	M	W				
Addi $4, $5, 1					F	d*	d*	D	X	M	W	
Add $5, $3, $1								F	D	X	M	W

دور زدن کامل:

Instruction	1	2	3	4	5	6	7	8	9	10	11	12
Sub $2, $3, $1	F	D	X	M	W							
Lw $5, 0($2)		F	D	X	M	W						
Addi $4, $5, 1			F	D	d*	X	M	W				
Add $5, $3, $1				F	d*	D	X	M	W			

$$SpeedUp = \frac{ET_1}{ET_2} = \frac{12}{9} = 1.33 \Rightarrow 33\%$$

4. آ) کد a:

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14
LW R2, 0(R2)	F	D	X	M	W									
BEQ R2, R0, Label1		F	D	-	X	M	W							
LW R2, 0(R2)			F	-	D	X	M	W						
BEQ R2, R0, Label1					F	D	-	X	M	W				
OR R2, R2, R3									F	D	X	M	W	
SW R2, 0(R5)										F	D	X	M	W

کد b:

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14
LW R1, 0(R1)	F	D	X	M	W									
BEQ R2, R0, Label2		F	D	-	X	M	W							
LW R3, 0(R2)						F	D	X	M	W				
BEQ R3, R0, Label1							F	D	-	X	M	W		
BEQ R2, R0, Label2								F	-	D	X	M	W	
SW R1, 0(R2)										F	D	X	M	W

ب) کد a:

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14
LW R2, 0(R2)	F	D	X	M	W									
BEQ R2, R0, Label1 (delay slot)		F	D	-	X	M	W							
OR R2, R2, R3			F	-	D	X	M	W						
LW R2, 0(R2)					F	D	X	M	W					
BEQ R2, R0, Label1						F	D	-	X	M	W			
OR R2, R2, R3 (delay slot)							F	-	D	X	M	W		
SW R2, 0(R5)										F	D	X	M	W

کد b:

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14
LW R1, 0(R1)	F	D	X	M	W									
BEQ R2, R0, Label2		F	D	-	X	M	W							
LW R3, 0(R2) (delay slot)			F	-	D	X	M	W						
BEQ R3, R0, Label1						F	D	X	M	W				
ADD R1, R3,R1							F	D	X	M	W			
BEQ R2, R0, Label2								F	D	X	M	W		
LW R3, 0(R2) (delay slot)									F	D	X	M	W	
SW R1, 0(R2)										F	D	X	M	W

```
toupper:
        101 lb $t2, 0($a0)
        101 beq $t2, $0, exit        # stop at end of string
        100 blt $t2, 97, next        # not lowercase
        100 bgt $t2, 122, next       # not lowercase
        100 sub $t2, $t2, 32         # convert to uppercase
        100 sb $t2, 0($a0)           # store in string
next:
        100 addi $a0, $a0, 1
        100 j toupper
exit:
          1 jr \$ra
```

5. آ) در تصویر تعداد اجرای هر دستور را نوشتم. پس سرجمع 803 دستور اجرا میشود.

ب) در معماری *single cyvle* همواره *CPI* برابر *1* است.

$$ExecuteTime = IC \times CPI \times ClockCycleTime$$
$$= 803 \times 1 \times 8 = 6424\, ns$$

ج) دستور دوم هر بار *2* تا *stall* میخورد چون داده *t2* در پایان مرحله *Memory* تازه میرسد و در کلاک بعد قبل از آنکه *Write back* شود میتوانیم آن را با *DataForwarding* به *ID* برسانیم. در کل *3* عدد دستور پرش شرطی داریم که سر جمع *301* بار اجرا میشوند، بنابراین در آن *10٪* مواقع که اشتباه پیشبینی میکنند به طور تقریبی *30.1* کلاک حروم میکنند. پس داریم:

$$Cycle\ Count: 4 + 803 + (2 \times 101) + 30.1 = 1039.1$$

د) خط لوله ای نسبت به تک چرخه ای *209٪* بهبود دارد.

$$speedup = \frac{ExecTime_{single}}{ExecTime_{pipeline}} = \frac{6424}{1039.1 \times 1 \times 2} = 3.09$$

6. آ) مکانیزم پیش‌بینی‌کننده‌ی ۲ بیتی(*2-bit saturating counter*) یکی از روش‌های پیش‌بینی پرش *(branch prediction)* در معماری کامپیوتر است. برای هر دستور پرش شرطی، یک شمارنده‌ی ۲ بیتی نگه داشته می‌شود. این شمارنده می‌تواند ۴ مقدار داشته باشد:

پیش‌بینی	حالت	مقدار شمارنده
Not Taken	*Strongly Not Taken*خیلی احتمال عدم پرش	***00***
Not Taken	*Weakly Not Taken*احتمال کم عدم پرش	***01***
Taken	*Weakly Taken*احتمال کم پرش	***10***
Taken	*Strongly Taken*خیلی احتمال پرش	***11***

عملکرد:

1. در ابتدا برای هر پرش، شمارنده در یک مقدار تنظیم می‌شود (مثلاً *Strongly Taken* = *11*)
2. در هر بار اجرای پرش:
   - اگر پیش‌بینی درست باشد ← شمارنده تغییر نمی‌کند یا به سمت همان جهت حرکت می‌کند.
   - اگر پیش‌بینی غلط باشد ← شمارنده به سمت جهت مخالف یکی کم یا زیاد می‌شود.
3. به این ترتیب، سیستم در مقابل تغییرات ناگهانی در رفتار پرش‌ها مقاوم است.

یک پیش‌بینی‌کننده ۱ بیتی با کوچکترین تغییر رفتار سریعاً پیش‌بینی را عوض می‌کند اما ۲ بیتی با دو مرحله‌ی میانی *(weakly taken / weakly not taken)*پایداری بیشتری دارد.

مزایا:

- پایداری بیشتر از پیش‌بینی ۱ بیتی
- مقاوم در برابر نوسان‌های کوچک در رفتار پرش
- بسیار ساده برای پیاده‌سازی در سخت‌افزار (فقط یک شمارنده ۲ بیتی برای هر آدرس پرش)