هو العلیم

اصول سیستم‌های کامپیوتری

نیمسال اول سال تحصیلی ۱۴۰۲ – ۱۴۰۱

تمرینات ۴

مریم رضائی

۱. تابع $f(x) = \sin x$ را می‌توان به شکل این سری مک‌لورن نمایش داد:

$$f(x) = \sin x = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)!} x^{2k+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

برای تعیین مقدار تقریبی تابع $f(x) = \sin x$ می‌توانیم مجموع چند جمله اول از بی‌شمار جملات این سری را محاسبه کنیم. اگر حد بالای متغیر k را n بگذاریم، مقدار تقریبی تابع، با محاسبه $n+1$ جمله اول، به این صورت تعیین خواهد شد:

$$\sin x \approx \sum_{k=0}^{n} \frac{(-1)^k}{(2k+1)!} x^{2k+1}$$

شما از روی این فرمول، برنامه‌ای را مستقیماً به زبان RISC-V بنویسید که مقدار تقریبی تابع $f(x) = \sin x$ را محاسبه کند.

اسم (برچسب) تابعی را که می‌نویسید sin یا sine بگذارید. از آنجا که در بدنه این تابع، مکرراً محاسبه مقادیر x^{2k+1} و مقادیر $(2k+1)!$ لازم است، دو تابع نیز با نام‌های (برچسب‌های) power و fact بنویسید. (کد RISC-V تابع fact در بخش ۸.۲ کتاب ذکر شده است و شما می‌توانید همان قطعه کد را در برنامه خود استفاده کنید.)

خودتان شماره رجیسترهایی را که در برنامه RISC-V استفاده می‌کنید طبق قراردادها تعیین کنید. توجه کنید که x عددی حقیقی است و n هم یک عدد صحیح مثبت.

جواب:

از آنجا که رجیسترها ۳۲ بیتی هستند ابتدا فرض می‌کنیم اعداد تنها ۳۲ بیتی می‌شوند. مثلا فرض می‌کنیم تعداد جملات سری مک‌لورن بیش از ۵ نیست؛ زیرا اگر به فاکتوریل که بزرگ‌ترین بخش محاسبه است توجه کنیم درمیابیم فاکتوریل بیش از ۱۲ را نمی‌توان محاسبه و در رجیسترها ذخیره کرد. همچنین دقت می‌کنیم x حقیقی است پس برای انجام اعمال روی آن از دستورات ممیز شناور دقت واحد استفاده می‌کنیم. در نهایت دقت می‌کنیم ورودی و خروجی توابع را همواره در رجیسترهای f0 و x10 تعیین کرده‌ایم، پس برای فراخوانی هر تابع (ورودی دادن) و یا استفاده از خروجی آن رجیسترها را بنا بر این نکته تعیین کرده و می‌خوانیم. حال هر یک از ۳ تابع را جدا توضیح می‌دهیم:

- **تابع sine:** به طور غیر بازگشتی با استفاده از توابع فاکتوریل و توان تعریف شده، جملات سری مک‌لورن را محاسبه می‌کنیم. برای اینکار ورودی‌های زاویه (x) و تعداد جمله (n) را در f0 و x10 دریافت کرده و از آنجا که چند فراخوانی توابع دیگر را داریم، مقادیر آدرس فراخوانی و ورودی x تابع را در پشته ذخیره می‌کنیم.

  برای محاسبه ابتدا متغیرهایی برای k، حاصل کلی سیگما و حاصل بخش‌های مختلف یک جمله‌ی سیگما تعبیه می‌کنیم. از آنجا که x10 برای ورودی توابع بعد تغییر خواهد کرد n را نیز در رجیستری دیگر برای حفظ آن و

مقایسه‌ی پایان حلقه کپی می‌کنیم. همچنین دقت می‌کنیم تابع توان تعریف شده پایه را به شکل ممیز شناور در ورودی می‌گیرد، پس عدد 1 – را برای استفاده‌ی هر بار در حلقه برای محاسبه‌ی علامت جمله در رجیستر ممیز شناوری ذخیره می‌کنیم. در آخر، با توجه به اینکه دستوری برای کپی مقدار یک رجیستر ممیز شناور در دیگری نداریم، مقدار صفر را در رجیستر ممیز شناوری برای استفاده‌ی هر بار با fadd.s قرار می‌دهیم.

حال که تمامی رجیسترهای پایه برای محاسبات تعریف کرده‌ایم، با استفاده از حلقه‌ای برای هر k از صفر تا n، جمله‌ی مربوطه را با استفاده از فرمول و توابع توان و فاکتوریل به دست می‌آوریم و به حاصل کل سیگما اضافه می‌کنیم. باز دقت می‌کنیم که خروجی تابع توان به شکل عدد حقیقی در رجیستر ممیز شناور محاسبه شده و بنابراین محاسبات هر جمله نیز بایستی با رجیسترها و دستورات ممیز شناور انجام شود، پس هر بار تبدیل و کپی را با کمک رجیسترهای از قبل تعیین شده انجام می‌دهیم. در انتها آدرس را بازیابی کرده، حاصل را در f0 که خروجی است قرار داده، پشته را خالی می‌کنیم و آدرس تابع فراخوانی تابع بازمی‌گردیم.

- **تابع fact:** به طور بازگشتی هر بار از اجرای تابع بر ورودی یکی کمتر، حاصل فاکتوریل را پیدا می‌کنیم. برای اینکار ورودی (n) را در x10 دریافت کرده و با ذخیره‌ی مقدار آن برای این سطح از فراخوانی در پشته و سپس تغییر رجیستر x10 برای خروجی پیش می‌رویم. فرض می‌کنیم مقادیر بیش از ۳۲ نیستند.

  ابتدا مقدار n این سطح از فراخوانی و آدرس مکانی که فراخوانی این تابع را انجام داده بود را در پشته ذخیره می‌کنیم (زیرا در ادامه‌ی همین سطح با کم کردن از ورودی و فراخوانی دوباره‌ی تابع و تغییر x10 به خروجی آن، مقادیر رجیستر و آدرس تغییر می‌کنند). سپس اگر ورودی کوچکتر از یک بود خروجی را یک قرار داده و به آدرس فراخوانی تابع برمی‌گردیم، و در غیر این صورت به بخش فراخوانی بازگشتی رفته و از خروجی آن برای ضرب $n \times fact(n-1)$ استفاده می‌کنیم.

  لازم به ذکر است که در شرط اول (کوچکتر از صفر بودن) نیازی به بازیابی مقدار گذاشته شده در پشته نداریم و فقط پشته را پیش از اتمام تابع و خالی می‌کنیم، اما در شرط دوم (بزرگتر مساوی بودن یک) نیاز است به علت تغییر آدرس پس از فراخوانی بازگشتی و همچنین برای ضرب n این سطح در خروجی فراخوانی بازگشتی (که در x10 قرار دارد) آدرس و n را از پشته بازیابی کنیم، سپس پشته را خالی و به مکان فراخوانی بازگردیم.

- **تابع pow:** به طور غیر بازگشتی با استفاده از یک حلقه پایه را به تعداد توان در خود ضرب می‌کنیم. برای اینکار ورودی‌های پایه (x) و توان (n) را در f0 و x10 دریافت کرده، سپس f0 را به خروجی (که هر بار به روی آن ضرب انجام شده) تغییر داده و پایه را به رجیستری موقت منتقل می‌کنیم. آنگاه با استفاده از شمارنده‌ای از توان ۱ تا n را تک تک مبنای یک ضرب توسط حلقه قرار می‌دهیم.

  دقت می‌کنیم که در صورتی که توان صفر باشد خروجی ۱ است، پس ابتدا خروجی را برابر ۱ و شمارنده را برابر صفر قرار می‌دهیم، و در ورود به حلقه ابتدا در صورت برابری شمارنده با n (یعنی صفر بودن توان) به حلقه وارد نشده و با همان خروجی ۱ به آدرس فراخوانی تابع بازمی‌گردیم. در غیر این صورت وارد حلقه شده و خروجی را با ضرب تغییر می‌دهیم.

برنامه RISC-V

```
sine:                   // function for sin(x), f0 = x and x10 = n are inputs
        addi sp, sp, -8     // adjust stack for 2 items
        sw x1, 4(sp)        // save return address
        fsw f0, 0(sp)       // save the argument x
        addi x5, x0, 0      // x5 = 0 as k a.k.a sum's counter
        addi x6, x10, 0     // x6 = n as sum's limit
        fcvt.s.w f1, x0     // f1 = 0 as result of sums
        addi x7, x0, -1     // x7 = -1 to build f2 for pow(-1, k)
        fcvt.s.w f2, x7     // f2 = -1 for pow(-1, k)
        fcvt.s.w f3, x0     // f3 = 0 for  copying sth to f0 via fadd.s
s.loop: fadd.s f0, f2, f3   // f0 = -1 for pow(-1, k)
        addi x10, x5, 0     // x10 = k for pow(-1, k)
        jal x1, pow         // call pow(-1, k)
        fadd.s f4, f0, f3   // move result of pow(-1, k) to f4
        flw f0, 0(sp)       // restore argument x into x10 for pow(x, 2k+1)
        slli x10, x5, 1     // x10 = 2k to construct 2k+1
        addi x10, x10, 1    // x10 = x10 + 1 to construct 2k+1
        jal x1, pow         // call pow(x, 2k+1)
        fmul.s f4, f4, f0   // f4 = (-1)^k × pow(x, 2k+1)
        jal x1, fact        // call fact(2k+1), x10 = 2k+1 still
        fcvt.s.w f0, x10    // f0 = fact(2k+1)
        fdiv.s f4, f4, f0   // f4 = ((-1)^k × pow(x, 2k+1)) / fact(2k+1)
        fadd.s f1, f1, f4   // f1 += current term, to construct sum of terms up until now
        blt x5, x6, s.loop  // if k < n, rerun loop to calculate next term
        fadd.s f0, f1, f3   // when loop ends, put result of sum into f0 to return f0
        lw x1, 4(sp)        // restore call address into x1
        addi sp, sp, 8      // pop 2 items from stack
        jalr x0, 0(x1)      // return to caller

fact:                   // function for n! recursively, x10 = n is input
        addi sp, sp, -8     // adjust stack for 2 items
        sw x1, 4(sp)        // save return address
        sw x10, 0(sp)       // save the argument n
        addi x5, x10, -1    // x5 = n – 1
        bge x5, x0, f.else  // if (n – 1) ≥ 0, go to f.else
        addi x10, x0, 1     // put 1 into x10 to return 1
        addi sp, sp 8       // pop 2 times from stack since this level is done
        jalr x0, 0(x1)      // return to caller
f.else: addi x10, x10, -1   // since n ≥ 1, argument of next call becomes n – 1
        jal x1, fact        // call fact(n – 1) recursively
        addi x6, x10, 0     // return from jal: move result of fact(n – 1) to x6
        lw x10, 0(sp)       // restore n of this level (that was changed for recursive call)
        lw x1, 4(sp)        // restore return address (that was changed for recursion)
        addi sp, sp, 8      // pop 2 items from stack since this level is done
        mul x10, x10, x6    // put result into x10 to return n × fact(n – 1)
        jalr x0, 0(x1)      // return to caller
```

```
pow:                    // function for x^n where n ≥ 0, f0 = x and x10 = n are inputs
        addi x5, x0, 0  // x5 = 0 as counter variable
        fcvt.s.w f1, x0 // f1 = 0 to build f1 = x
        fadd.s f1, f1, f0 // f1 = x to multiply by
        addi x6, x0, 1  // x6 = 1 to build f01 return value
        fcvt.s.w f0, x6 // f0 = 1 return value to be changed
p.loop: bge x5, x10, p.end // if counter ≥ n, end loop, else continue
        fmul.s f0, f0, f1 // multiply return value by x
        addi x5, x5, 1  // increment the counter
        jal x0, p.loop  // recheck loop condition
p.end:  jalr x0, 0(x1)  // return to caller
```

مراجع: کتاب منبع درس (برای تابع fact):

Computer Organization and Design (RISC-V Edition) – Patterson & Hennessy

۲. تصور کنید که می‌خواهیم یک پردازنده RISC-V بسازیم که میزان تأخیر مؤلفه‌های آن طبق این جدول باشد:

نام واحد	نماد تأخیر	مقدار (ps)
Register Read	t_{read}	40
Register Setup	t_{setup}	50
Multiplexer	t_{mux}	30
Single Gate	t_{gate}	10
ALU	t_{ALU}	120
Control	t_{con}	25
Sign Extend	t_{ext}	35
Memory Read	t_{mem}	200
Register File Read	t_{RFread}	100
Register File Setup	$t_{RFsetup}$	60

(منظور از t_{read} مدت زمانی است که باید برای خواندن محتوای یک رجیستر صرف شود؛ و منظور از t_{setup} مدت زمانی است که باید برای نوشتن مقداری در یک رجیستر صرف شود.)

الف) اگر بخواهیم پردازنده‌ای «تک دور ساعتی» بسازیم، دور ساعت آن حداقل چقدر باید باشد؟

برای تعیین دور ساعت پردازنده، به این دو نکته توجه کنید:

- اجرای دستور lw (از میان دستوراتی که قرار است با پردازنده اجرا شوند) بیشتر از بقیه دستورات طول خواهد کشید. پس مسیر بحرانی پردازنده، مسیری است که باید برای اجرای دستور lw طی شود. برای تعیین مسیر بحرانی، شکل ۲۴.۴ کتاب درسی را ببینید.
- در همان حال که مقادیر دو رجیستر از واحد رجیسترها خوانده می‌شوند، واحد کنترل و واحد imm gen و مالتی‌پلکسری که قبل از ALU قرار گرفته است، ورودی دوم ALU را مشخص می‌کنند.

ب) اکنون فرض کنید که می‌خواهیم پردازنده «تک دور ساعتی» را با اضافه کردن رجیسترهای خط لوله‌ای به یک پردازنده «خط لوله‌ای پنج مرحله‌ای» تبدیل کنیم.

و فرض کنید طبق برنامه‌های محکی که برای مقایسه کارایی پردازنده‌ها نوشته شده‌اند، انتظار ما این است که درصد دستورات RISC-V در برنامه‌هایی که قرار است روی پردازنده خط لوله‌ای اجرا شوند نزدیک به درصدهای مذکور در این جدول باشد:

دستور	loads	stores	branches	jumps	R-type/I-type
درصد	۲۵	۱۰	۱۱	۲	۵۲

همچنین فرض کنید که بعد از ۴۰ درصد از دستورات load ، دستوری وجود دارد که از نتیجه دستور load استفاده می‌کند و همین باعث می‌شود که اجرای آن دستور، یک دور ساعت به تأخیر بیفتد. و اینکه در ۵۰ درصد از دستورات branch شرط درست است (یعنی پیش‌بینی ما نادرست است) و اجرای درست هر یک از آن ۵۰ درصد از دستورات پرش شرطی، نیازمند آن است که دو دستور بعد از آن پاک شود.

با این مفروضات، مقدار CPI پردازنده خط لوله‌ای پنج مرحله‌ای را تعیین کنید.

پ) اکنون فرض کنید که می‌خواهیم پردازنده «تک دور ساعتی» را با اضافه کردن رجیسترهای خط لوله‌ای به یک پردازنده خط لوله‌ای N مرحله‌ای تبدیل کنیم.

همچنین فرض کنید که:

- می‌توان N مرحله را به گونه‌ای تعیین کرد که میزان تأخیر همه مراحل یکسان باشد.
- هر رجیستر خط لوله‌ای، 90ps تأخیر خواهد داشت.
- تأخیر واحدهایی که برای مدیریت مخاطرات به پردازنده اضافه شده‌اند را می‌توان نادیده گرفت.
- اضافه کردن هر مرحله به پردازنده پنج مرحله‌ای، به علت بیشتر کردن تعداد مخاطرات، باعث خواهد شد که 0.1 به مقدار CPI پردازنده پنج مرحله‌ای اضافه شود.

با این مفروضات، سؤال این است: تعداد مراحل پردازنده خط لوله‌ای (مقدار N) چقدر باید باشد تا پردازنده برنامه‌ها را با بیشترین سرعت ممکن اجرا کند؟

جواب:

الف) از آنجا که می‌خواهیم دور ساعت پردازنده آنقدر طولانی باشد که تمامی دستورات را در یک دور ساعت انجام دهد، بایستی طول آن را به اندازه‌ی طول اجرای طولانی‌ترین دستور قرار دهیم. این دستور lw است، پس با توجه به شکل ۴.۲۴ کتاب، به شرح مراحل و زمان هر کدام می‌پردازیم:

۱) خواندن رجیستر PC: $40\ ps$

۲) خواندن دستور از حافظه توسط Instruction memory: $200\ ps$

۳) خواندن دو رجیستر برای ورودی ALU (که یکی در lw وجود ندارد و فقط یک رجیستر خوانده می‌شود) و به

طور موازی با عمل قبل، اجرای دنباله واحدهای واحد کنترل، واحد Imm Gen، مالتی پلکسر قبل ALU:

$$\max\{100\ ps, 25 + 35 + 30\ ps\} = 100\ ps$$

۴) اجرای ALU: $120\ ps$

۵) خواندن مقدار از حافظه در واحد Data memory: $200\ ps$

۶) اجرای مالتی پلکسر بعد از Data memory: $30\ ps$

۷) نوشتن بر روی رجیستر: $60\ ps$

در نهایت، برای جمع مقادیر و زمان کلی اجرای lw داریم: $40 + 200 + 100 + 120 + 200 + 30 + 60 = 750\ ps$

پس زمان یک دور ساعت (که آن را T_{CC} می‌نامیم) در پردازنده‌ی جدید بایستی $750\ ps$ باشد یعنی: $T_{CC} = 750\ ps$

ب) اگر فرض کنیم CC_i تعداد دور ساعت دستور iام بنابر CC (زمان یک دور ساعت)، Instruction Count یا IC_i تعداد آن نوع دستور در تمام دستورات باشد، و IC نیز تعداد کل دستورات باشد، آنگاه CPI یعنی تعداد دور هر دستور را داریم:

$$CPI = \sum \frac{IC_i \times CC_i}{IC}$$

حال مقدار فرمول را برای هر i محاسبه می‌کنیم (یعنی هر جمله‌ی سیگما را به دست می‌آوریم) و در نهایت با هم جمع می‌کنیم. بنابراین بر اساس جدول برای ۱۰۰ دستور اجرایی کلی (مقادیر درصد هستند)، هر نوع را بررسی می‌کنیم:

۱) **دستورات R-type/I-type:** در این دسته از دستورات مخاطرات خاصی موجود نبوده و در صورت به وجود آمدن مخاطره همواره با پیشرانی قابل حل هستند. بنابراین هر دستور در زمان $CC_i = 1$ CC اجرا می‌شود:

$$CPI_1 = \frac{IC_1 \times CC_1}{IC} = \frac{52}{100}\ CC$$

۲) **دستورات Jump:** در این دسته از آنجا که مقدار پرش (یعنی PC+immediate) در مرحله‌ی سوم خط لوله‌ای تعیین می‌شود که این باعث می‌شود به جای یک دور ساعت، ۳ دور ساعت باعث معطلی شوند، یعنی ۲ دستور بعدی اجرا شوند که بایستی حذف شوند، بنابراین زمان کل آن ۳ دور ساعت خواهد بود:

$$CPI_2 = \frac{IC_2 \times CC_2}{IC} = \frac{2 \times 3}{100}\ CC$$

۳) **دستورات Branch:** در دستورات شرطی فرض می‌کنیم شرط غلط است. ذکر کردیم در ۵۰ درصد حالات فرض ما غلط است؛ در این حالت باز مثل پرش، ۲ دستور بعد نیز با دستور اجرا شده‌اند که باید حذف شوند پس ۳ دور ساعت مصرف شده است. فرمول را داریم:

$$CPI_3 = \frac{IC_3 \times CC_3}{IC} = \frac{11 \times (1/2 \times 1 + 1/2 \times 3)}{100}\ CC = \frac{11 \times 2}{100}\ CC$$

۴) **دستورات Store:** مانند دسته‌ی اول، مخاطرات با عملیات پیشرانی حل شده و معطلی اضافه نداریم، پس:

$$CPI_4 = \frac{IC_4 \times CC_4}{IC} = \frac{10}{100}\ CC$$

۵) **دستورات Load:** ذکر کردیم در ۶۰ درصد این دستورات به مخاطره برنمی‌خوریم پس تنها یک دور ساعت زمان می‌برند. اما از آنجا که خواندن از حافظه بر مقدار رجیستر می‌نویسد که بنا بر فرض در ۴۰ درصد حالات اجرا باعث ایجاد مخاطره با دستور بعد از آن می‌شود، برای رفع مخاطره نیاز به یک NOP (یعنی یک دستور بی اثر) داریم که به عنوان یک دستور یک دور ساعت اضافه می‌کند. بنابراین برای فرمول داریم:

$$CPI_5 = \frac{IC_5 \times CC_5}{IC} = \frac{25 \times \left({}^{3}/_{5} \times 1 + {}^{2}/_{5} \times 2\right)}{100}\ CC = \frac{25 \times {}^{7}/_{5}}{100}\ CC$$

در نهایت برای مقدار کلی CPI بر اساس زمان یک دور ساعت یعنی یک CC مقادیر بالا را با هم جمع می‌کنیم. یعنی:

$$CPI = \left(\frac{52}{100} + \frac{2 \times 3}{100} + \frac{11 \times 2}{100} + \frac{10}{100} + \frac{25 \times {}^{7}/_{5}}{100}\right) CC = \frac{125}{100}\ CC = 1.25\ CC$$

پ) می‌خواهیم مقدار N را به گونه‌ای تعیین کنیم که بیشترین سرعت ممکن را داشته باشیم؛ یعنی زمان هر دستور کمترین مقدار ممکن باشد. می‌دانیم اگر T_{CPU} زمان کل CPU، IC تعداد دستورات، CPI تعداد دورها برای هر دستور، و T_{CC} زمان یک دور ساعت باشد، رابطه‌ی زیر را داریم. از این رابطه، زمان هر دستور یعنی T_{Inst} را به دست می‌آوریم:

$$T_{CPU} = IC \times CPI \times T_{CC} \Longrightarrow \frac{T_{CPU}}{IC} = CPI \times T_{CC} \Longrightarrow T_{Inst} = CPI \times T_{CC}$$

پس باید $CPI \times T_{CC}$ کمترین مقدار ممکن باشد. مقادیر آن‌ها را بر حسب N به دست آورده و آنگاه مقدار N مناسب برای مینیمم بودن ضرب را محاسبه می‌کنیم. بنابراین ابتدا برای CPI و T_{CC} داریم:

- **تعداد دور ساعت هر دستور**: گفتیم اضافه کردن هر مرحله به پردازنده‌ی پنج مرحله‌ای به علت افزایش تعداد مخاطرات باعث خواهد شد ۰/۱ به مقدار CPI پردازنده پنج مرحله‌ای بخش ب اضافه شود. یعنی:

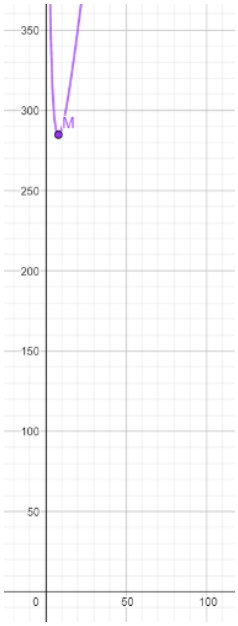
$$CPI = 1.25 + \frac{1}{10}(N - 5)$$

- **زمان هر دور ساعت:** در قسمت الف زمان یک دور ساعت را ۷۵۰ پیکو ثانیه در آوردیم و فرض کردیم با اضافه کردن N خط لوله می‌توان دور ساعت را به طور یکسان تقسیم کرد، یعنی ${}^{750}/_{N}$ پیکو ثانیه. هر خط لوله نیز ۹۰ پیکو ثانیه به زمان اضافه می‌کند، پس N خط لوله به هر یک از N دور ۹۰ پیکو ثانیه اضافه می‌کند. یعنی:

$$T_{CC} = \frac{750}{N} + 90$$

حال بایستی تابع زیر را برای محاسبه‌ی T_{Inst} که از ضرب مقادیر بالا به دست می‌آید مینیمم کنیم:

$$T_{Inst} = CPI \times T_{CC} = \left(1.25 + \frac{(N-5)}{10}\right) \times \left(\frac{750}{N} + 90\right)$$



با رسم نمودار تابع بالا به ازای $x = N$ تعداد مرحله‌های خط لوله، $y = T_{Inst}$ زمان اجرای هر دستور، و شرط مثبت بودن x، نمودار روبه‌رو را روی x از اعداد حقیقی داریم. مینیمم y این نمودار در نقطه‌ی $M = (7.91, 284.8)$ اتفاق می‌افتد، یعنی $N = 7.91$. از آنجا که تعداد مرحله‌ها بایستی اعداد طبیعی باشد، دو عدد طبیعی اطراف ۷/۹۱ را به دست می‌آوریم:

$$N = 7 \xrightarrow{T_{Inst}=CPI\times T_{CC}} T_{Inst} = \left(1.25 + \frac{(7-5)}{10}\right) \times \left(\frac{750}{7} + 90\right) = 285.857143$$

$$N = 8 \xrightarrow{T_{Inst}=CPI\times T_{CC}} T_{Inst} = \left(1.25 + \frac{(8-5)}{10}\right) \times \left(\frac{750}{8} + 90\right) = 284.8125$$

پس به ازای $N = 8$ مرحله‌ی خط لوله‌ای، پردازنده برای هر یک از دستورات دستگاه کمترین زمان ممکن را دارد. یعنی سریع‌ترین حالت ممکن برای هر برنامه‌ای است.

مراجع:

https://www.sciencedirect.com/topics/computer-science/pipeline-stage

۳. فرض کنید که پردازنده خط لوله‌ای پنج مرحله‌ای قرار است این دنباله از دستورات RISC-V را اجرا کند:

```
addi  x9,  x0,  52
addi  x8,  x9,  −4
lw    x19,  16(x8)
sw    x19,  20(x8)
xor   x18,  x8,  x19
or    x18,  x18,  x19
```

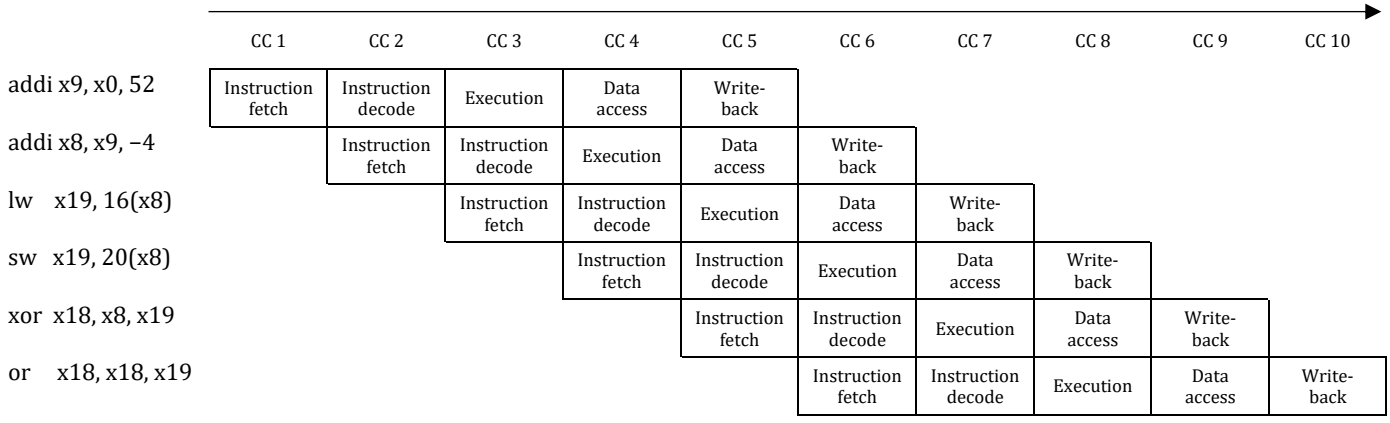
الف) با رسم نموداری (شبیه به شکل ۴۶.۴) مشخص کنید در دور ساعت پنجم از اجرای خط لوله‌ای دستورات، مقادیر کدام رجیسترها تغییر می‌کند و مقادیر کدام رجیسترها خوانده می‌شوند؟

ب) با رسم نموداری (شبیه به شکل‌های ۶۰.۴ و ۶۱.۴) مشخص کنید که پردازنده باید چه وقت عملیات پیشرانی (forwarding) انجام دهد و چه وقت باید اجرای خط لوله را موقتاً متوقف (stall) کند تا برنامه به درستی اجرا شود.

پ) چند دور ساعت لازم است تا پردازنده همه دستورات را وارد خط لوله کند؟ مقدار CPI پردازنده برای اجرای این برنامه چیست؟

جواب:

الف) با شکل کتاب که برای ۵ دستور است، شکل عادی و رفع مخاطره نشده‌ی ۶ دستور ارائه شده را رسم می‌کنیم.



بنابر شکل، مرحله‌ی هر دستور در دور پنجم را بررسی می‌کنیم تا تعیین کنیم چه اعمال تغییر و خواندنی اتفاق می‌افتد:

۱) **دستور addi اول:** در مرحله‌ی «پس نوشتن» بر رجیستر مقصد می‌نویسد، پس x9 تغییر می‌کند.

۲) **دستور addi دوم:** در مرحله‌ی «دسترسی به حافظه» نیاز به خواندن حافظه نیست پس کاری انجام نمی‌دهد.

۳) **دستور lw:** در مرحله‌ی «اجرا» جمع ۱۶ با محتوی x8 را حساب می‌کند، که محتوی x8 در مرحله قبل خود

(یعنی دور ساعت ۴) خوانده شده بوده است. پس در این مرحله رجیستری خوانده یا تغییر داده نمی‌شود.

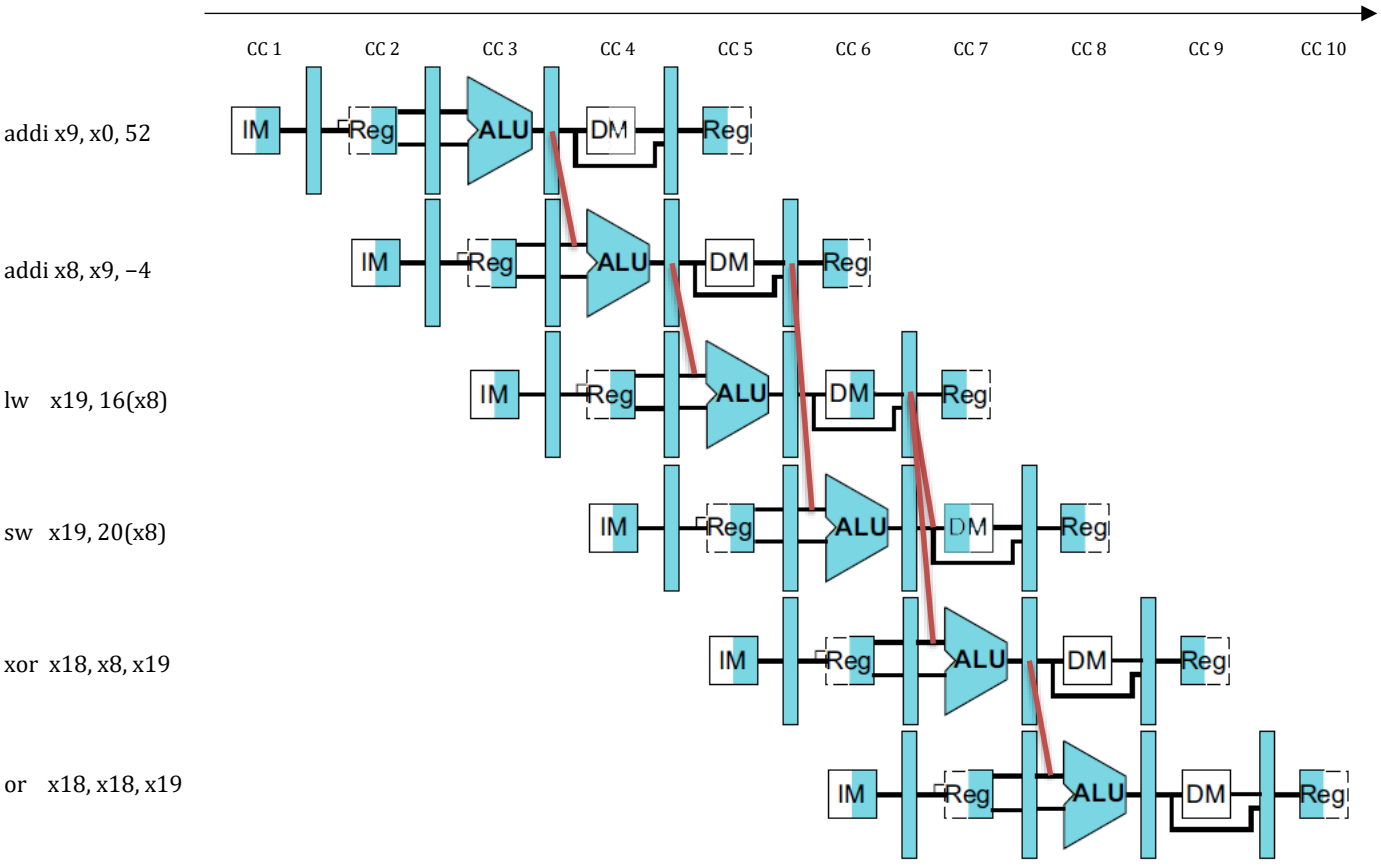
۴) **دستور sw:** در مرحله‌ی «کد برداری دستور» واحد فایل رجیستر محتوی رجیستر x8 را می‌خواند.

۵) **دستور xor:** در مرحله‌ی «گرفتن دستور» رجیستر PC هم خوانده می‌شود هم تغییر داده می‌شود. بنابر مقدار خوانده شده‌ی آن نیز با واحد حافظه‌ی دستورات، دستور از حافظه خوانده می‌شود.

پس مشاهده می‌کنیم که رجیستر x9 تغییر داده می‌شود، x8 خوانده می‌شود، و PC هم خوانده و هم تغییر داده می‌شود. دقت می‌کنیم در هر مرحله رجیستر خط لوله‌ای پیشین نیز خوانده می‌شود و بر رجیستر پسین نوشته می‌شود.

نکته: این شکل بدون رفع مخاطرات است. بنابراین رجیستر x8 از حافظه خوانده می‌شود و تغییری که دستور addi دوم برای x8 در دور ساعت چهارم محاسبه شده بر رجیستر اعمال نشده است. پس مقدار x8 خوانده شده نادرست است، مگر مقدار با پیشرانی از رجیستر خط لوله‌ای خوانده شود. در بخش بعد به پیشرانی‌های مورد نیاز می‌پردازیم.

ب) با توجه به مطالب ذکر شده در بخش قبل، اعمال پیشرانی را در شکل با خطوط قرمز رسم می‌کنیم. واحدهای فعال را با رنگ آبی رنگ کرده‌ایم؛ در صورت رنگ شدن تنها یک نیمه یعنی فقط عمل نوشتن (چپ) یا خواندن (راست) انجام می‌شود و نه هر دو. رجیسترهای خط لوله‌ای همواره کامل رنگ شده‌اند زیرا هر دو عمل بر آنها اعمال می‌شود.



بنابراین برای هر رجیستر که بایستی مقدار آن خوانده شود اما در دستوری پیشین مقداری جدید برای آن محاسبه شده است که هنوز بر آن نوشته نشده، با پیشرانی محاسبه با مقدار جدید آن انجام می‌شود. همانطور که مشاهده می‌کنیم

هیچ خط پسرویی نداریم، بنابراین نیاز به هیچ توقف و ایجاد حبابی نیست.

پ) بنابر شکل رسم شده پس از رفع مخاطره همچنان ۱۰ دور ساعت برای اجرای تمامی دستورات کافی است؛ یعنی همانطور که مشاهده می‌کنیم ۵ دور ساعت برای اجرای دستور اول صرف شده است و هر دستور پس از آن تنها در یک دور ساعت اضافه‌تر اجرا شده و به پایان رسیده‌است. این امر با استفاده از فرمول نیز قابل مشاهده است.

در تمرین قبل ذکر کردیم که تعداد دورهای هر دستور را با استفاده از فرمول زیر محاسبه می‌کنیم:

$$CPI = \sum \frac{IC_i \times CC_i}{IC}$$

از آنجا که تعداد کل دستورات $IC = 6$ است، و تعداد دستور addi، $IC_1 = 2$ است اما یکی ۵ دور ساعت و دیگری ۱ دور زمان می‌برد و بنابراین جدا محاسبه می‌شوند، و تعداد هر نوع دستور دیگر $IC_i = 1$ می‌باشد، صورت کسر (یعنی تعداد کل دورها) برابر با همان ۱۰ ذکر شده می‌باشد که بر تعداد دستورات تقسیم می‌شود. زیرا برای رابطه‌ی بالا داریم:

$$CPI = \frac{(1 \times 5 + 1 \times 1) + 1 \times 1 + 1 \times 1 + 1 \times 1 + 1 \times 1}{6} \; CC = \frac{10}{6} \; CC \approx 1.667 \; CC$$

مراجع: کتاب منبع درس:

Computer Organization and Design (RISC-V Edition) – Patterson & Hennessy

۴. در این تمرین، نحوه تخمین انرژی مصرفی پردازنده و رابطه انرژی مصرفی با کارایی پردازنده را مورد بررسی قرار می‌دهیم. برای حل مسائلی که در این تمرین مطرح می‌شوند، فرض می‌کنیم که میزان مصرف انرژی واحدهای Instruction Memory و Registers و Data Memory برابر با مقادیری باشد که در این جدول ذکر شده‌اند:

I-Mem Read	Register Read	Register Write	D-Mem Read	D-Mem Write
140pJ	70pJ	60pJ	140pJ	120pJ

(فرض کنید که میزان انرژی مصرفی دیگر مؤلفه‌های پردازنده آنقدر کم باشد که بتوان آنها را نادیده گرفت. عملیات Register Read و Register Write بیانگر خواندن و نوشتن در رجیسترهای واحد Registers هستند.)

فرض کنید که میزان تأخیر مؤلفه‌های پردازنده برابر با مقادیری باشد که در این جدول ذکر شده‌اند:

I-Mem	Control	Register Read or Write	ALU	D-Mem Read or Write
200ps	150ps	90ps	90ps	250ps

(فرض کنید که میزان تأخیر دیگر مؤلفه‌های پردازنده آنقدر کم باشد که بتوان آنها را نادیده گرفت.)

الف) یک پردازنده تک دور ساعتی، چقدر انرژی برای اجرای یک دستور add مصرف می‌کند؟ یک پردازنده خط لوله‌ای پنج مرحله‌ای، چقدر انرژی برای اجرای یک دستور add مصرف می‌کند؟

ب) بدترین دستور RISC-V از نظر مصرف انرژی چیست؟ چقدر انرژی برای اجرای آن مصرف می‌شود؟

پ) چگونه طرح پردازنده خط لوله‌ای را تغییر دهیم تا انرژی کمتری را برای اجرای بعضی از دستورات مصرف کند؟ بعد از این تغییر در طرح پردازنده، چند درصد از میزان انرژی‌ای که پردازنده، قبل از این، برای اجرای دستور lw مصرف می‌کرد کم می‌شود؟

ت) تغییر مورد نظر در قسمت پ، برای کاهش انرژی مصرفی پردازنده در طول اجرای چه دستورات دیگری از مجموعه دستورات RISC-V مفید است؟

ث) تغییر مورد نظر در قسمت پ، چگونه روی میزان کارایی پردازنده خط لوله‌ای اثرگذار خواهد بود؟

ج) ما می‌توانیم خط کنترلی MemRead را حذف کنیم؛ یعنی کاری کنیم که دائماً MemRead = 1 باشد و در هر دور ساعت، مقداری از واحد حافظه داده‌ها خوانده شود. توضیح دهید که چرا بعد از این تغییر در طرح پردازنده، باز پردازنده به درستی کار خواهد کرد. اگر ۲۵ درصد دستورات برنامه‌ها دستورات load باشند، تأثیر این تغییر روی میزان دور ساعت پردازنده و میزان انرژی مصرفی واحد Data Memory چیست؟

جواب:

الف) پردازنده چه تک دور ساعتی باشد چه خط لوله‌ای، بایستی واحدهای یکسانی را برای اجرای دستور add به کار بی‌اندازد. همانطور که می‌دانیم این واحدها به ترتیب «حافظه‌ی دستورات» برای خواندن دستور، «رجیسترها» برای خواندن دو رجیستر (یعنی دو برابر انرژی)، ALU برای محاسبه‌ی جمع (که با فرض نا چیز بودن حساب نمی‌شود)، و «رجیسترها» برای نوشتن بر یک رجیستر می‌باشد. با جمع مصرف انرژی این اعمال داریم:

$$IMem\ Read + 2 \times Register\ Read + Register\ Write = 140 + 2 \times 70 + 60\ \text{pJ} = 340\ \text{pJ}$$

ب) همانطور که می‌دانیم تمامی دستورات از حافظه‌ی دستورات می‌خوانند، و واحد رجیسترها هم در تمامی آن‌ها دو رجیستر از دستور گرفته شده کد برداری می‌کنند و می‌خوانند. پس برای تعیین دستور با بیشترین مصرف انرژی به سه عمل پر انرژی دیگر توجه می‌کنیم. واضح است که دسترسی به حافظه انرژی بالایی مصرف می‌کند، بنابراین یکی از دستورات دسترسی به حافظه بایستی پرانرژی‌ترین دستور ما باشد؛ به بررسی این دو دستور می‌پردازیم:

- **دستور sw:** به جای انرژی مصرفی برای نوشتن بر رجیستر که در add ذکر شده است، بر حافظه می‌نویسد:

$$IMem\ Read + 2 \times Register\ Read + DMem\ Write = 140 + 2 \times 70 + 120\ \text{pJ} = 400\ \text{pJ}$$

- **دستور lw:** هم از حافظه می‌خواند هم بر رجیستر می‌نویسد، پس علاوه بر انرژی مصرفی محاسبه شده برای add، انرژی خواند از حافظه را نیز دارد. یعنی دستور با بیشترین مصرف انرژی دستور lw می‌باشد، زیرا:

$$IMem\ Read + 2 \times Register\ Read + DMem\ Read + Register\ Write = 340 + 140\ \text{pJ} = 480\ \text{pJ}$$

پ) اگر دقت کنیم، کار بیهوده‌ی اصلی قابل مشاهده خواندن دو رجیستر در واحد رجیسترها است که در خیلی از دستورات مانند lw نیازی به این کار نیست. پس می‌توانیم با بسط واحد کنترل، به آن قابلیت تشخیص نیاز به خواندن هر کدام را اضافه کنیم. در این صورت در دستور lw دیگر نیازی به دو 70 pJ نیست و فقط یک رجیستر خوانده می‌شود. بنابراین از 480 pJ انرژی کلی 70 pJ کم می‌شود، یعنی $\frac{70}{480} \times 100 \approx 14.58334$، پس بیش از ۱۴ درصد کم می‌شود.

ت) تغییر بیان شده در هر دستوری که نیاز به خواندن دو رجیستر ندارد مفید است؛ یعنی دستوارتی که یا فقط یک خواندن رجیستر دارند یا هیچ خواندن رجیستری ندارند. این موارد را با توجه به ساختار دستورات به شکل زیر داریم:

- دستورات U-type و UJ-type نیاز به خواندن هیچ رجیستری ندارند.
- دستورات I-type نیاز به خواندن تنها یک رجیستر دارند.
- شبه دستورات حاصل از دستورات SB-type مانند beqz یا R-type مانند mv نیاز به خواندن یک رجیستر دارند و اگر قابلیت تشخیص x0 را نیز اضافه کنیم در آنها نیز صرفه جویی می‌شود.
- شبه دستورات I-type مانند li نیاز خواندن رجیستر ندارند و به همان شکل می‌توان در آن‌ها صرفه جویی کرد.

ث) در پیش داشتیم که عمل کد برداری در واحد کنترل به طور موازی با عمل خواندن رجیسترها در واحد رجیسترها انجام می‌شود. اما به علت وابستگی خواندن رجیسترها به کد برداری در تغییر ارائه شده، اعمال به طور موازی انجام نشده و به میزان تاخیر حاصل از واحد کنترل اضافه می‌شود. پس اگر میزان تاخیر جدید حاصل از کد برداری دستور از 250ps که دور ساعت قبلی بر اساس پر تاخیرترین عمل (نوشتن بر یا خواندن از حافظه) می‌باشد بیشتر شود به دور ساعت پردازنده‌ی خط لوله‌ای اضافه می‌شود.

ج) هر مورد را بررسی می‌کنیم:

- **درستی کارکرد تغییر:** یا دستور نیاز به خواندن حافظه داشته است که در این صورت دستور به درستی اجرا می‌شود، یا دستور نیازی به خواندن حافظه نداشته است که در این صورت به مقدار خوانده شده اجازه‌ی عبور از مالتی پلکسر مرحله‌ی آخر داده نخواهد شد. پس در هر صورت اختلالی در اجرای دستورات به وجود نمی‌آید.
- **میزان دور ساعت جدید:** از آنجا که دور ساعت قبلی بنا بر وقت‌گیرترین دستور (نوشتن بر یا خواندن از حافظه) تعیین شده بود، پس اجرای همیشگی عمل خواندن از حافظه تغییری در دور ساعت نیاز ندارد؛ به عبارت دیگر دور ساعت به شکلی طولانی در نظر گرفته شده بود که عمل خواندن از حافظه همواره در یک دور آن جا بشود، پس دور ساعت جدید برابر با دور قبلیست.
- **میزان مصرف انرژی جدید:** از آنجا که انرژی خواندن از دستور یعنی انرژی مصرفی واحد Data memory به تمامی دستورات (حتی دستوراتی که نیازی به آن ندارند) تحمیل می‌شود، باعث می‌شود واحد انرژی هدر رفته داشته باشد و در کل انرژی بیش از اندازه مصرف کند.

مراجع:

۵. الف) دو پردازنده ARM Cortex-A53 و Intel Core i7 6700 نمونه‌هایی هستند از پردازنده‌هایی که در عمل موفق بوده‌اند. با پر کردن خانه‌های خالی این جدول، این دو پردازنده را از ابعاد مختلف با هم مقایسه کنید.

پردازنده	سال	سرعت ساعت (clock rate)	تعداد مراحل خط لوله	پهنای جریان (issue width)	اجرای خارج از نوبت (out of order)	تعداد هسته‌ها
Cortex-A53						
Core i7 6700						

ب) خط لوله پردازنده ARM Cortex-A53 از چه مراحلی تشکیل می‌شود؟ مشخصه‌ها و عواملی که روی کارایی این پردازنده اثر مثبت می‌گذارند چیستند؟ مشخصه‌ها و عواملی که روی کارایی این پردازنده اثر منفی می‌گذارند چیستند؟

پ) خط لوله پردازنده Intel Core i7 6700 از چه مراحلی تشکیل می‌شود؟ مشخصه‌ها و عواملی که روی کارایی این پردازنده اثر مثبت می‌گذارند چیستند؟ مشخصه‌ها و عواملی که روی کارایی این پردازنده اثر منفی می‌گذارند چیستند؟

ت) پردازنده ARM Cortex-A53 در ساخت چه نوع رایانه‌هایی استفاده شده است؟ پردازنده Intel Core i7 6700 در ساخت چه نوع رایانه‌هایی استفاده شده است؟ آیا می‌توان انرژی مصرفی این دو پردازنده را با هم مقایسه کرد و نتیجه گرفت که میزان انرژی مصرفی یکی از این دو پردازنده، کمتر از انرژی مصرفی پردازنده دیگر است؟ آیا می‌توان سرعت این دو پردازنده را با هم مقایسه کرد و نتیجه گرفت که سرعت یکی از این دو پردازنده، بیشتر از سرعت پردازنده دیگر است؟

جواب:

الف) بر اساس مطالب کتاب و جستجو، جدول را پر می‌کنیم. مشاهده می‌کنیم i7 6700 جدیدتر و سریع‌تر است.

پردازنده	سال	سرعت ساعت (clock rate)	تعداد مراحل خط لوله	پهنای جریان (issue width)	اجرای خارج از نوبت (out of order)	تعداد هسته‌ها
Cortex-A53	2012	1.3 GHz	8	2 micro-ops (۲ ریز عمل در یک دور ساعت)	No	4
Core i7 6700	2015	3.4 GHz	14	6 micro-ops (تا ۶ ریز عمل در یک دور ساعت)	Yes	4

ب) برای A53 هر سوال را جدا بررسی می‌کنیم:

- **مراحل خط لوله:** برای دستورات غیر شرطی، هشت مرحله‌ی F1، F2، D1، D2، D3/ISS، EX1، EX2 و WB موجودند. مراحل F1 و F2 دستور را می‌گیرند، D1 و D2 عمل کد برداری (decode) ساده را انجام می‌دهند و D3 کد برداری دستورات پیچیده‌تر را انجام می‌دهد که با اولین مرحله‌ی خط لوله‌ی اجرا یعنی ISS هم‌پوشانی دارد. پس از آن مراحل EX1 و EX2 و سپس WB کار پایان و باز نوشتن برای اتمام و کامل شدن خط لوله‌ی عدد صحیح را انجام می‌دهند. دقت می‌کنیم که برای دستورات شرطی و اجرای ممیز شناور مراحل اصلی قبل بیشتر می‌شوند. برای دستورات شرطی بسته به نوع دستور چهار پیشبینی کننده موجودند که در مراحل گرفتن دستور در واحد تولید آدرس برای PC بررسی شده و می‌توانند باعث دورهای تاخیری شوند، و در اجرای ممیز شناور علاوه بر ۵ مرحله‌ی گرفتن و کد برداری دستور، ۵ دور دیگر (نه فقط ۳ دور) نیاز که باعث می‌شود مراحل اجرای کلی ۱۰ عدد شوند. شکل ۴.۷۵ کتاب مراحل را به تصویر کشیده است.
- **عوامل مثبت کارایی:** این پردازنده از خط لوله‌ی سطحی و پیش‌بینی کننده‌ی پرش پافشاری استفاده می‌کند که امکان دستیابی به دور ساعت بالا با مصرف انرژی پایین را به وجود آورده و در هدر رفتن اجرای خط لوله صرف جویی می‌کنند. این علتی است که در مقایسه با i7 با وجود ۴ هسته‌ای بودن هر دو، این پردازنده $\frac{1}{200}$ انرژی i7 را مصرف می‌کند.
- **عوامل منفی کارایی:** تاخیرات در خود خط لوله‌ی این پردازنده از ۳ منبع نشأت می‌گیرند:

۱) **مخاطرات اجرایی:** در زمانی اتفاق می‌افتد که به علت ساختار دو دستور در دور ساعت پردازنده، دو دستور مجاور برای اجرای همزمان انتخاب شده اما از یک بخش خط لوله‌ی اجرایی یکسان استفاده می‌کنند؛ کامپایلر باید از این تاخیر جلوگیری کند و دستورات باید در ابتدای خط لوله چند بخشی شده و تک تک هر یک انتخاب شوند.

۲) **مخاطرات داده‌ای:** در زمانی اتفاق می‌افتد که ایرادی در داده‌ی مورد نیاز دستورات اجرایی در ابتدای خط لوله تشخیص داده شده و باعث عدم اجرای هر دو دستور (یعنی یا داده به هر دو ربط دارد و با تنها به دستور اول ارتباط دارد و به علت توقف دستور اول دستور دوم نیز متوقف می‌شود) یا عدم اجرای تنها دومین دستور می‌شود؛ باز کامپایلر می‌تواند از این امر جلوگیری کند.

۳) **مخاطرات کنترلی:** در زمانی اتفاق می‌افتد که پیشبینی کننده‌ی پرش قضاوت اشتباه کند و از آنجا که تصمیمات پرش در لوله‌ی صفر در ALU اتفاق می‌افتند، یک تصمیم اشتباه باعث ضرر ۸ دور می‌شود؛ ساختار طراحی شده برای ۴ پیشبینی کننده‌ی A53 همواره امکان پیشبینی غلط را داشته هدر رفتن کار را به همراه دارد که باعث تاخیر می‌شود.

تاخیرات خارج از خط لوله نیز ممکن هستند، برای مثال خطا در حافظه‌ی میانجی (بافر) TLB یا در کش. این تاخیرات در ضعیف‌ترین اجراها به نسبت تاخیرات حاصل از مخاطرات خط لوله بسیار بیشتر و سنگین‌تر هستند.

پ) برای i7 6700 هر سوال را جدا بررسی می‌کنیم:

- **مراحل خط لوله:** این خطه لوله دارای ۱۴ مرحله است (شکل ۴.۷۹ کتاب). در ساختار این خط لوله، مفاهیمی جدید برای رفع پاد وابستگی‌ها و حدس غلط معرفی می‌شوند. به طور دقیق‌تر، پردازنده از یک بافر برای تغییر چینش و همچنین دسته‌ی کوچک‌تری از رجیسترهای ساختاری (که جدا از رجیسترهای فیزیکی که بیشتر هستند تعریف می‌شوند) استفاده می‌کند و نگاشتی میان آن‌ها در نظر می‌گیرد که تعیین می‌کند کدام رجیستر فیزیکی جدیدترین کپی یک رجیستر ساختاریست. آنگاه برای رفع پاد وابستگی‌ها و حدس غلط با تغییر نام رجیسترها پیش می‌رود؛ یعنی در صورت حدس غلط، به آخرین نگاشت درست میان دو نوع رجیستر ساختاری و فیزیکی بر می‌گردد. با توجه به این مفاهیم، مراحل خط لوله را به شکل خلاصه داریم:

۱) با استفاده از یک بیشبینی کننده‌ی شرطی چند سطحی سریع و دقیق، پردازنده بنا بر آدرس پیشبینی شده، ۱۶ بایت از کش دستور را می‌گیرد. به علت وجود بخش پیشبینی در این گرفتن دستور ابتدایی، در صورت پیشبینی غلط هزینه‌ی ۱۷ دور (چند دور اضافه برای تعیین دوباره‌ی پیشبینی کننده‌ی شرطی) به اجرا تحمیل می‌شود.

۲) ۱۶ بایت خوانده شده در بافر دستورات کد برداری نشده قرار می‌گیرد و با بررسی انواع طول دستورات ممکن به دستورات تکی x86 تقسیم می‌شود. سپس حاصل‌ها در صف دستورات قرار می‌گیرند.

۳) کد برداری دستورات درون صف انجام می‌شود. سه عدد از کد بردار (decoder) ها دستورات x86 را که به طور مستقیم به ریز اعمال MIPS-شکل مدیریت می‌کنند و دیگر دستورات x86 که پیچیده‌تر هستند به موتور ریزبرنامه‌سازی رفته و به ریز اعمال معادل طی هر چند دور مورد نیاز ترجمه می‌شوند. سپس ریز اعمال حاصل به بافر ۶۴-ورودی ریز اعمال وارد می‌شوند.

۴) بر دستورات بافر شناسایی جریان حلقه صورت می‌گیرد و در صورت یافتن دنباله‌ای کوچک از دستوارت (زیر ۶۴ دستور) که حلقه‌ای را شکل می‌دهند، آن ریز اعمال را منتشر می‌کند.

۵) انتشار پایه‌ی دستوارت با مراحل جستجوی مکان رجیستر در جدول رجیسترها، تغییر نام رجیسترها، تعیین ورودی بافر تغییر چینش، و گرفتن نتایج از رجیسترها یا از بافر تغییر چینش انجام می‌شود. در نهایت ریز اعمال به ایستگاه‌های نگهداری ارسال می‌شوند.

۶) ایستگاه نگهداری مرکزی ۶ واحد اجرایی، ۶ ریز عمل را در هر دور ساعت به واحدها ارسال می‌کند.

۷) هر واحد اجرایی ریز اعمال را اجرا می‌کند و نتایج را به ایستگاه‌های نگهداری منتظر و واحد بازنشستگی رجیسترها (که با آپدیت رجیستر تعیین می‌کند دستور دیگر حدسی (speculative) نیست) پس می‌فرستد. ورودی مربوط به هر دستور در بافر تغییر چینش به عنوان کامل علامت زده می‌شود.

۸) زمانی که یک یا بیشتر دستورات سر بافر تغییر چینش کامل علامت زده شدند، نوشتن‌های در حال انتظار در واحد بازنشستگی رجیسترها اجرا می‌شوند و دستورات از بافر تغییر چینش حذف می‌شوند.

- **عوامل مثبت کارایی:** حدس پافشار استفاده شده تخمین تفاوت کارایی ایده‌آل و واقعی را سخت می‌کند. اما با قرار گرفتن در کنار بافرها و صف‌های مفصل، باعث کاهش احتمال تاخیر به علت نبود ایستگاه نگهداری، تغییر نام رجیسترها، و بافرهای تغییر چینش می‌شود. در این پردازنده میزان پیشبینی غلط نیمی از میزان پیشبینی غلط ددر A53 می‌باشد. همچنین پهنای جریان بالای ممکن شده بر این اساس باعث افزایش کارایی می‌شود.
- **عوامل منفی کارایی:** اکثر تاخیرات از پیشبینی اشتباه پرش و خطاهای کش به وجود می‌آیند، اما علل دیگری نیز بر تاخیر اثر می‌گذارند. بعضی موارد دارای تاثیر منفی را به طور کل به شکل زیر داریم:

۱) **پیشبینی غلط**: پرش‌هایی که پیشبینی آنها سخت است. همانطور که گفتیم، پیشبینی غلط هزینه‌ی ۱۷ دور (چند دور اضافه برای تعیین دوباره‌ی پیشبینی کننده‌ی شرطی) را تحمیل می‌کند.

۲) **پیچیدگی دستورات:** استفاده از دستورات x86 پیچیده که به راحتی به ریز اعمال ترجمه نمی‌شوند.

۳) **وابستگی‌های طولانی:** این تاخیرات در نتیجه‌ی دستورات با اجرای طولانی یا رتبه‌بندی حافظه‌اند.

۴) **دسترسی حافظه:** تاخیر که از دسترسی به حافظه به وجود می‌آید. برای مثال در صورت خطاهای کش بسته به نوع خطا (L1، L2، L3) انواع هزینه‌های مربوطه را بر مبنای تعداد دور به همراه داریم. البته پردازنده در خطاهای L2 و L3 تلاش می‌کند تا دستورات دیگری را برای اجرا پیدا کند، اما در صورت پر شدن بافرها پردازنده از اجرای دستورات متوقف می‌شود.

همچنین به علت پیچیدگی بالای این طراحی انرژی این پردازنده بسیار بیشتر (۲۰۰ برابر A53) می‌باشد.

ت) پردازنده‌ی A53 به شکل هسته‌ی IP ارائه شده که بیشتر در سیستم‌های تعبیه شده‌ی سیار شخصی استفاده می‌شوند در حالی که پردازنده‌ی i7 6700 در سیستم‌های رایانشی کلی کاربرد دارد. تفاوت این دو نوع سیستم میزان تمرکز آن‌ها بر اعمال خاص است؛ هسته‌های مخصوص سیستم‌های تعبیه شده برای آمیختن با منطقی دیگر مانند پردازنده‌های خاص یک کار (برای مثال کد گذاری یا کد برداری فیلم)، رابط I/O و رابط حافظه طراحی و برای عمل محدود مربوطه بهینه شده‌اند، اما هسته‌های مخصوص سیستم‌های عمومی توان متنوع‌تر و کلی‌تر دارند.

همانطور که اشاره کردیم، استفاده از حدس پافشار در هر دو ساختار، تخمین تفاوت کارایی ایده‌آل و واقعی را در این پردازنده‌ها دشوار کرده است. اما می‌توان به طور عملی و آماری مصرف انرژی و سرعت میانگینی برای این پردازنده‌ها با پیاده‌سازی‌های موجود یافت. که در این صورت مشاهده می‌کنیم مصرف انرژی i7 حدود ۲۰۰ برابر A53 است (بیشتر است) اما سرعت هر ریز عمل آن نیز بیش از ۵ برابر بیشتر از A53 است (سرعت ساعت بیش از ۲ برابر بیشتر)، زیرا:

$$A53 \xrightarrow{1.36\ CPI\ and\ 1.3\ GHz\ clock\ rate} CPI \times clock\ cycle = 1.36 \times {}^{1}/_{1.3\ GHz} = 1.05\ ns$$
$$i7\ 6700 \xrightarrow{0.64\ CPI\ and\ 3.4\ GHz\ clock\ rate} CPI \times clock\ cycle = 0.64 \times {}^{1}/_{3.4\ GHz} = 0.18\ ns$$

مراجع: کتاب منبع درس: Computer Organization and Design (RISC-V Edition) – Patterson & Hennessy