تمرین های تئوری

تمرین اول

معماری کامپیوتر ـ دکتر اسدی (زمستان ۱۴۰۳)

نرگس کاری

دانشکده مهندسی کامپیوتر

فهرست مطالب

سوال اول

۱.۱ (آ) محاسبه‌ی مقدار CPI میانگین

CPI میانگین از رابطه‌ی زیر محاسبه می‌شود:

$$CPI = \sum(UsageRate \times ClockCyclesperInstruction)$$

$$CPI_{C۱} = (۰/۱۶ \times ۶) + (۰/۱۰ \times ۸) + (۰/۰۸ \times ۱۰) + (۰/۶۶ \times ۳) = ۰/۹۶ + ۰/۸ + ۰/۸ + ۱/۹۸ = ۴/۵۴$$

$$CPI_{C۲} = (۰/۱۶ \times ۲۰) + (۰/۱۰ \times ۳۲) + (۰/۰۸ \times ۶۶) + (۰/۶۶ \times ۳) = ۳/۲ + ۳/۲ + ۵/۲۸ + ۱/۹۸ = ۱۳/۶۶$$

۲.۱ (ب) محاسبه‌ی مقدار MIPS

MIPS از رابطه‌ی زیر محاسبه می‌شود:

$$MIPS = \frac{ClockRate}{CPI \times ۱۰^۶}$$

با توجه به اینکه فرکانس پردازنده ۴۰۰ مگاهرتز است:

$$MIPS_{C۱} = \frac{۴۰۰}{۴/۵۴} \approx ۸۸/۱۱$$

$$MIPS_{C۲} = \frac{۴۰۰}{۱۳/۶۶} \approx ۲۹/۲۸$$

۳.۱ (ج) محاسبه‌ی زمان اجرای برنامه

$$ExecutionTime = \frac{InstructionCount \times CPI}{ClockRate}$$

چون تعداد دستورالعمل‌ها ۱۲۰۰۰ است:

$$ExecutionTime_{C۱} = \frac{۱۲۰۰۰ \times ۴/۵۴}{۴۰۰ \times ۱۰^۶} = \frac{۵۴۴۸۰}{۴۰۰ \times ۱۰^۶} = ۱۳۶/۲\ \mu s$$

$$ExecutionTime_{C۲} = \frac{۱۲۰۰۰ \times ۱۳/۶۶}{۴۰۰ \times ۱۰^۶} = \frac{۱۶۳۹۲۰}{۴۰۰ \times ۱۰^۶} = ۴۰۹/۸\ \mu s$$

۴.۱ (د) محاسبه میزان تسریع

با توجه به قسمت قبل باید از فرمول زیر استفاده کنیم:

$$\text{Speedup} = \frac{\text{time}_{\text{original}}}{\text{time}_{\text{improved}}} = \frac{CPI_{\text{original}}}{CPI_{\text{improved}}}$$

$$CPI'_{C۲} = (۰/۱۶ \times ۲۰ \times \frac{۱}{۲}) + (۰/۱۰ \times ۳۲ \times \frac{۱}{۴}) + (۰/۰۸ \times ۶۶ \times \frac{۱}{۳}) + (۰/۶۶ \times ۳)۱/۶ + ۰/۸ + ۱/۷۶ + ۱/۹۸ = ۶/۱۴$$

پس داریم:

$$\text{Speedup} = \frac{CPI_{C۲}}{CPI'_{C۲}} = \frac{۱۳/۶۶}{۶/۱۴} = ۲/۲۲$$

سوال دوم

۱.۲ (آ) محاسبه CPI برای هر کامپایلر

با استفاده از فرمول زیر:

$$CPI = \frac{CPUTime}{ClockCycleTime \times InstructionCount}$$

برای کامپایلر A داریم:

$$CPI_A = \frac{۱\text{s}}{(۱۰^{-۹}\text{s}) \times (۱ \times ۱۰^{۹})} = ۱$$

برای کامپایلر B داریم:

$$CPI_B = \frac{۱/۴\text{s}}{(۱۰^{-۹}\text{s}) \times (۱/۲ \times ۱۰^{۹})} = ۱/۱۶$$

۲.۲ (ب) مقایسه عملکرد دو کامپایلر

می دانیم که زمان اجرای برنامه از فرمول زیر محاسبه میشود.

$$ExecutionTime = InstructionCount \times CPI \times ClockCycleTime$$

با توجه که این زمان برای دو پردازنده یکسان شده است پس داریم:

$$ExecutionTime_{C_A} = ExecutionTime_{C_B}$$

$$InstructionCount_A \times CPI_A \times ClockCycleTime_{C_A} = InstructionCount_B \times CPI_B \times ClockCycleTime_{C_B}$$

$$\frac{ClockCycleTime_{C_A}}{ClockCycleTime_{C_B}} = \frac{CPI_B \times InstructionCount_B}{CPI_A \times InstructionCount_A} = \frac{۱/۱۶ \times (۱/۲ \times ۱۰^۹)}{۱ \times (۱ \times ۱۰^۹)} = ۱/۴$$

با توجه به اینکه زمان دقیق اجرای برنامه هارا نداشتیم ، تنها میتوانیم به طور نسبی بگوییم چرخه زمانی پردازنده ای که کد کامپایلر A را اجرا میکند ۴.۱ برابر پردازنده ی دیگر است و کند تر است.

۳.۲ (ج) محاسبه میزان تسریع در کامپایلر جدید

با توجه به فرمول هایی که داشتیم، داریم:

$$\text{Speedup} = \frac{\text{time}_{\text{previous}}}{\text{time}_{\text{new}}}$$

برای کامپایلر جدید داریم:

$$ExecutionTime = InstructionCount \times CPI \times ClockCycleTime = (۶۰۰ \times ۱۰^۶) \times ۱/۱ \times ۱۰^{-۹} = ۰/۶۶$$

پس نسب به کامپایلر A داریم:

$$Speedup_A = \frac{۱}{۰/۶۶} = ۱/۵۱$$

و نسبت به کامپایلر B داریم:

$$Speedup_B = \frac{۱/۴}{۰/۶۶} = ۲/۱۲$$

سوال سوم

۱.۳ (آ) مقایسه دو معماری SIMD ، MIMD با مثال

۱.۱.۳ معماری SIMD $(SingleInstruction, MultipleData)$

- در این معماری، یک دستورالعمل واحد روی چندین مجموعه داده به‌طور همزمان اجرا می‌شود.
- پردازنده‌های **گرافیکی (GPU)** و پردازش‌های **برداری** $(VectorProcessing)$ از این مدل استفاده می‌کنند.
- **مزیت:** بهره‌وری بالا در پردازش‌های داده‌محور مانند پردازش تصویر، پردازش سیگنال، و شبیه‌سازی‌های فیزیکی.

۲.۱.۳ معماری MIMD $(MultipleInstruction, MultipleData)$

- در این معماری، چندین پردازنده مستقل، دستورالعمل‌های متفاوتی را روی داده‌های مختلف اجرا می‌کنند.
- پردازنده‌های **چندهسته‌ای** $(Multi-CoreCPUs)$ و **ابررایانه‌ها** $(Supercomputers)$ از این مدل استفاده می‌کنند.
- **مزیت:** انعطاف‌پذیری بالا برای اجرای برنامه‌های عمومی و پردازش‌های پیچیده که به وظایف مستقل نیاز دارند.

۳.۱.۳ مثال در کامپیوترهای شخصی

- پردازنده‌های **های‌CPU چندهسته‌ای** مانند Intel و AMD از معماری MIMD استفاده می‌کنند تا برنامه‌های مختلف را همزمان اجرا کنند.

- پردازنده‌های **گرافیکی (GPU)** از معماری SIMD برای پردازش تصاویر و ویدئوها استفاده می‌کنند.
- در کامپیوترهای مدرن، ترکیب این دو معماری دیده می‌شود؛ CPU برای اجرای وظایف کلی و GPU برای پردازش‌های موازی سنگین.

۲.۳ (ب) بررسی Out-of-Order Execution

۱.۲.۳ تعریف

- **اجرای خارج از ترتیب** $(Out-of-OrderExecution)$ یک تکنیک در پردازنده‌هاست که به دستورالعمل‌ها اجازه می‌دهد مستقل از ترتیب برنامه اجرا شوند، به شرطی که وابستگی داده‌ای نداشته باشند.
- این روش برخلاف **اجرای ترتیبی** $(In-OrderExecution)$ است که دستورالعمل‌ها را به همان ترتیب که در کد برنامه هستند اجرا می‌کند.

۲.۲.۳ مراحل اجرای :OoOE

۱. **واگذاری و رمزگشایی دستورات:** پردازنده مجموعه‌ای از دستورات را بارگیری و رمزگشایی می‌کند.

۲. **بررسی وابستگی‌ها:** پردازنده مشخص می‌کند که کدام دستورات به داده‌های در حال آماده‌سازی وابسته نیستند.

۳. **اجرای دستورات آماده:** دستوراتی که داده‌های لازم را دارند، در واحدهای پردازشی مستقل اجرا می‌شوند.

۴. **مرتب‌سازی مجدد نتایج:** نتایج در Reorder Buffer (ROB) ذخیره شده و به ترتیب صحیح در ثبات‌ها یا حافظه ثبت می‌شوند.

۳.۲.۳ تأثیر بر عملکرد پردازنده

- افزایش **کارایی پردازنده** با کاهش زمان انتظار برای دسترسی به داده‌ها.
- بهبود **استفاده از منابع سخت‌افزاری** و کاهش **تأخیرهای ناشی از وابستگی داده‌ای**.
- باعث افزایش **میزان IPC** $(InstructionsPerCycle)$ می‌شود که منجر به اجرای سریع‌تر برنامه‌ها خواهد شد.

مثال: پردازنده‌های مدرن مانند **Intel Core i۷/i۹ و Ryzen AMD** از Out-of-Order Execution استفاده می‌کنند تا عملکرد بهتری داشته باشند، مخصوصاً در اجرای برنامه‌هایی که شامل پردازش‌های پیچیده هستند.

سوال چهارم

۱.۴ (الف) محاسبه CPI پردازنده در حالت اولیه:

$$CPI_{\text{initial}} = \sum (CPI_i \times UsageRate_i) = (۰/۳۵ \times ۳) + (۰/۲۰ \times ۵) + (۰/۱۵ \times ۴) + (۰/۲۵ \times ۲) + (۰/۰۵ \times ۱۰)$$

$$= ۱/۰۵ + ۱ + ۰/۶ + ۰/۵ + ۰/۵ = ۳/۶۵$$

۲.۴ (ب) محاسبه CPI پردازنده پس از اعمال روش‌های X و Y

روش :X کاهش ٪۵۰ در CPI دستورات Floating Point:

$$CPI_X = CPI_{\text{initial}} - ۰/۰۵ \times ۰/۵ \times ۱۰ = ۳/۶۵ - ۰/۲۵ = ۳/۴$$

روش :Y تقسیم دستورات Store:

$$CPI_{\text{Store،new}} = (۰/۶۰ \times ۲) + (۰/۴۰ \times ۵) = ۱/۲ + ۲ = ۳/۲$$

$$CPI_Y = CPI_{\text{initial}} - ۰/۱۵ \times (۴ - ۳/۲) = ۳/۶۵ - ۰/۱۲ = ۳/۵۳$$

۳.۴ (ج) بررسی تأثیر هر روش و مقایسه آن‌ها:

$$\text{بهبود روش X} = \frac{۳/۶۵}{۳/۴} = ۱/۰۷$$

$$\text{بهبود روش Y} = \frac{۳/۶۵}{۳/۵۳} = ۱/۰۳$$

بنابراین، روش X تأثیر بیشتری در بهبود عملکرد پردازنده دارد.

سوال پنجم

۱.۵ (الف) محاسبه CPI اولیه و زمان اجرای پردازنده:

$$CPI_{\text{initial}} = \frac{(۲۰۰۰۰ \times ۱) + (۱۵۰۰۰ \times ۲) + (۵۰۰۰ \times ۳)}{۲۰۰۰۰ + ۱۵۰۰۰ + ۵۰۰۰} = \frac{۶۵۰۰۰}{۴۰۰۰۰} = ۱/۶۲۵$$

زمان کل اجرای برنامه:

$$T_{\text{initial}} = \text{(تعداد کل دستورالعمل‌ها)} \times CPI_{\text{initial}} \times T_{\text{cycle}} = ۴۰۰۰۰ \times ۱/۶۲۵ \times ۰/۵ = ۳۲۵۰۰ \text{نانوثانیه}$$

تأثیر جایگزینی ٪۳۰ از دستورات B با کلاس D $(CPI = ۰/۵)$:

- **تعداد دستورات جایگزین‌شده:** $۱۵۰۰۰ \times ۰/۳ = ۴۵۰۰$
- **دستورات باقی‌مانده در B:** $۱۵۰۰۰ - ۴۵۰۰ = ۱۰۵۰۰$

محاسبه CPI جدید:

$$CPI_{\text{new}} = \frac{(۲۰۰۰۰ \times ۱) + (۱۰۵۰۰ \times ۲) + (۵۰۰۰ \times ۳) + (۴۵۰۰ \times ۰/۵)}{۴۰۰۰۰} = \frac{۵۸۲۵۰}{۴۰۰۰۰} = ۱/۴۵$$

محاسبه زمان اجرای جدید:

$$T_{\text{new}} = ۴۰۰۰۰ \times ۱/۴۵ \times ۰/۵ = ۲۹۱۲۵ \text{نانوثانیه}$$

محاسبه میزان بهبود

$$speedup = \frac{۳۲۵۰۰}{۲۹۱۲۵} = ۱/۱۱۶$$

بنابراین در عمل بهبود داشتیم.

۲.۵ (ب) بررسی بهینه‌سازی با کاهش تعداد دستورالعمل‌ها و افزایش

$$speedup = \frac{T_{initial}}{T_{new}} = \frac{IC_{initial} \times CPI_{\text{initial}} \times T_{\text{cycle}}}{IC_{new} \times CPI_{\text{new}} \times T_{\text{cycle}}} = \frac{۱ \times ۱}{۰/۸ \times ۱/۱} = ۱/۱۳$$

بنابراین این بهینه سازی سودمند است.

سوال ششم

۱.۶ (آ) محاسبه speedup با ۵ برابر شدن سرعت

$$T_{F-initial} = ۰/۴ \times T_{initial}$$

پس داریم:

$$speedup = \frac{T_{initial}}{T_{new}} = \frac{T_{initial}}{T_{initial} - T_{F-initial} + \frac{۱}{۵} \times T_{F-initial}} = \frac{۱}{۰/۶ + ۰/۲ \times ۰/۴} = ۱/۴۷$$

۲.۶ (ب) محاسبه speedup با x برابر شدن

$$speedup = \frac{T_{initial}}{T_{new}} = \frac{T_{initial}}{T_{initial} - T_{F-initial} + \frac{۱}{x} \times T_{F-initial}} = \frac{۱}{۰/۶ + \frac{۰/۴}{x}}$$

$$\lim_{x \to \infty} \frac{۱}{۰/۶ + \frac{۰/۴}{x}} = ۱/۶۶$$

۳.۶ (ج) محاسبه speedup نهایی سیستم

$$speedup = \frac{T_{initial}}{T_{new}} = \frac{T_{initial}}{(۱ - ۰/۳۰ - ۰/۲۵)T_{initial} + (\frac{۰/۳۰}{۴} + \frac{۰/۲۵}{۳})T_{initial}} = \frac{۱}{۰/۶۰۳} = ۱/۶۵$$

سوال هفتم

۱.۷ (آ) تحلیل کد

خط به خط کدداده شده را تحلیل میکنیم.

```
array: .word 15, -19, 17, 20, -10, 12, 100, -5
   la $a0, array # $a0 = 0x10010000
   addi $a1, $a0, 28
   move $v0, $a0
   lw $v1, 0($v0)
   move $t0, $a0
loop: addi $t0, $t0, 4
   lw $t1, 0($t0)
   bge $t1, $v1, skip
   move $v0, $t0
   move $v1, $t1
skip: bne $t0, $a1, loop
```

۱. یک آرایه از اعداد صحیح در حافظه تعریف می‌شود که شامل مقادیر زیر است:

$$\text{array:}\{۱۵, -۱۹, ۱۷, ۲۰, -۱۰, ۱۲, ۱۰۰, -۵\}$$

۲. مقدار آدرس ابتدایی این آرایه در ثبات a۰ ذخیره می‌شود:

$$a۰ = ۰x۱۰۰۱۰۰۰۰$$

۳. مقدار a۱ برابر با آدرس آخرین عضو آرایه (یعنی عضو هشتم، ۵−) تنظیم می‌شود:

$$a۱ = a۰ + ۲۸ = ۰x۱۰۰۱۰۰۱C$$

این مقدار از آنجایی به دست می‌آید که هر عدد صحیح ۴ بایت است و ۲۸ = ۴ × ۷ مقدار آفست آخرین عضو نسبت به ابتدای آرایه است.

۴. مقدار a۰ در v۰ کپی می‌شود.

۵. مقدار اولین عنصر آرایه (عدد ۱۵) در v۱ ذخیره می‌شود:

$$v۱ = \text{lw}۰(a۰) = ۱۵$$

۶. مقدار a۰ در t۰ کپی می‌شود.

۷. حلقه شروع می‌شود. ابتدا مقدار t۰ به اندازه ۴ بایت افزایش می‌یابد تا به عنصر بعدی آرایه اشاره کند.

۸. مقدار عنصری که آدرس آن در t۰ قرار دارد در t۱ بارگذاری می‌شود.

۹. مقدار درون t۱ با مقدار درون v۱ مقایسه می‌شود. اگر مقدار t۱ بزرگتر یا مساوی v۱ باشد، پرش به برچسب skip انجام می‌شود.

۱۰. اگر مقدار کوچکتری یافت شود، مقدار t۰ که آدرس آن مقدار را دارد، در v۰ ذخیره می‌شود.

۱۱. مقدار t۱ که کوچک‌ترین مقدار تا آن لحظه است، در v۱ ذخیره می‌شود.

۱۲. بررسی می‌شود که آیا به انتهای آرایه رسیده‌ایم یا نه. اگر هنوز به انتها نرسیده باشیم، به ابتدای حلقه بازمی‌گردیم و پردازش ادامه می‌یابد.

در پایان اجرای برنامه، v۰ شامل آدرس کوچک‌ترین مقدار در آرایه خواهد بود و v۱ مقدار کوچک‌ترین عضو آرایه را ذخیره خواهد کرد. در این مثال، مقدار کوچک‌ترین عضو آرایه برابر با ۱۹− و آدرس آن ۱۰۰۱۰۰۰۴x۰ است.

۲.۷ (ب) تحلیل فرم نهایی آرایه

کد را خط به خط تحلیل میکنیم.

۱. آدرس ابتدای آرایه در رجیستر a۰ ذخیره می‌شود.

۲. مقدار ۶ در a۱ ذخیره می‌شود که نشان‌دهنده تعداد تکرار حلقه است.

۳. مقدار a۰ (آدرس اولین عنصر آرایه) در t۰ کپی می‌شود.

۴. مقدار a۰ + ۱۲ در t۱ ذخیره می‌شود که نشان‌دهنده آدرس عنصر ششم آرایه است.

۵. مقدار ۱۶ بیتی موجود در آدرس t۰ در t۳ بارگذاری می‌شود.

۶. مقدار ۱۶ بیتی موجود در آدرس t۱ در t۴ بارگذاری می‌شود.

۷. مقدار ذخیره‌شده در t۳ (عدد اولیه در t۰) در آدرس t۱ (نیمه دوم آرایه) ذخیره می‌شود.

۸. مقدار ذخیره‌شده در t۴ (عدد اولیه در t۱) در آدرس t۰ (نیمه اول آرایه) ذخیره می‌شود.

۹. آدرس t۰ را دو بایت افزایش می‌دهد، به‌طوری‌که به عنصر بعدی آرایه اشاره کند.

۱۰. آدرس t۱ را دو بایت افزایش می‌دهد، تا به عنصر بعدی در نیمه دوم آرایه اشاره کند.

۱۱. مقدار a۱ را یک واحد کاهش می‌دهد تا تعداد تکرارها را کنترل کند.

۱۲. اگر a۱ صفر نشده باشد، به ابتدای حلقه پرش می‌کند تا پردازش ادامه یابد.

این کد عناصر نیمه اول آرایه را با نیمه دوم آن جابه‌جا می‌کند. بنابراین، پس از اجرای برنامه، آرایه به شکل زیر تغییر می‌کند

$$۷,۸,۹,۱۰,۱۱,۱۲,۱,۲,۳,۴,۵,۶$$

۳.۷ (ج) کد میپس برای شمارش تعداد ۱ ها

در برنامه زیر ما به عنوان مثال عدد ۲۱ را در a۰ ریختیم و تعداد ۱ های ان را شمرده و در v۰ ریختیم

```
1 .globl main
2 .text
3 main:
4     li $a0 , 21
5     li $v0, 0
6 loop:   andi $t1, $a0, 1
7     add $v0, $v0, $t1
8     srl $a0, $a0 , 1
9     bne $a0, $zero , loop
```

حال کد را خط به خط توضیح میدهیم

۱. مقدار ۲۱ در رجیستر $a۰ ذخیره می‌شود.

۲. مقدار اولیه $v۰ صفر شده و برای شمارش بیت‌های ۱ استفاده خواهد شد.

۳. حلقه به ازای هر بیت از راست به چپ اجرا می‌شود:

(آ) کم‌ارزش‌ترین بیت (LSB) با عملگر AND استخراج شده و در $t۱ ذخیره می‌شود.

(ب) مقدار استخراج‌شده به $v۰ اضافه می‌شود.

(ج) مقدار $a۰ یک بیت به راست شیفت داده می‌شود.

(د) اگر مقدار $a۰ صفر نشده باشد، حلقه تکرار می‌شود.

از خط ۵ تا ۹ دقیقا زیر برنامه ی خواسته شده ی سوال را داریم