تکالیف درس برنامه نویسی سیستمی

استاد احمدزاده

دانشجو محمد زارعی

۱)انواع لینکر را نام ببرید.

لینکرها در برنامه نویسی سیستم می‌توانند متنوع باشند از جمله:

1. لینکر برنامه: که فایل‌های شیوه پوشه‌ها را به یک فایل اجرایی ترکیب می‌کند.
2. لینکر کاربر: که برنامه‌های کاربر را به کتابخانه‌های مورد استفاده آن‌ها متصل می‌کند.
3. لینکر پویا: که به کتابخانه‌هایی که در زمان اجرا بارگذاری می‌شوند ارتباط برقرار می‌کند.
4. لینکر استاتیک: که تمام کتابخانه‌ها را در زمان کامپایله کردن فایل اجرایی ترکیب می‌کند.
5. لینکر جداساز: که به برنامه‌ها اجازه می‌دهد از چندین نسخه کتابخانه مختلف استفاده کنند.
6. لینکر هلپر (همچنین به عنوان loader شناخته می‌شود): که وظیفهٔ بارگذاری فایل‌ها را بر عهده دارد.
7. لینکر برگردان (Relocating Linker): که مسئول تخصیص مجدد آدرس‌های حافظه برای برنامه‌های قابل اجرا در زمان اجرا می‌باشد.
8. لینکر فشرده‌ساز: که برای کاهش اندازه فایل‌های اجرایی استفاده می‌شود.
9. لینکر امنیتی: که به منظور اعمال اصول امنیتی از جمله رمزنگاری یا امضاء دیجیتال بر روی فایل‌های اجرایی استفاده می‌شود.
10. لینکر ایستا: که کتابخانه‌ها را در زمان اجرا توسط کاربر انتخاب می‌کند و آن‌ها را به برنامه متصل می‌کند.

۲)فرآیند کامپایل را توضیح دهید.(C,C#,RUST,GO)

فرایند کامپایل زبان‌های کامپایلری معمولاً شامل چندین مرحله است که به ترتیب اجرا می‌شوند. در اینجا به توضیح مراحل عمومی کامپایلرها می‌پردازیم:

1. پیش‌پردازش (Preprocessing):

- این مرحله شامل پردازش دستورات خاص پیش‌پردازنده است، مانند تعریف ماکروها و شامل کردن فایل‌های هدر.

2. تحلیل لغوی (Lexical Analysis):

- در این مرحله، کد منبع به توکن‌ها (tokens) تقسیم می‌شود. توکن‌ها شامل کلمات کلیدی، شناسه‌ها، عملگرها و نشانه‌های مختلف دیگر هستند.

3. تحلیل نحوی (Syntax Analysis):

- در این مرحله، توکن‌ها به ساختارهای نحوی زبان تبدیل می‌شوند. این کار با استفاده از قواعد گرامری زبان صورت می‌گیرد و معمولاً منجر به تشکیل درخت نحوی (parse tree) می‌شود.

4. تحلیل معنایی (Semantic Analysis):

- این مرحله شامل بررسی معنای کد و اطمینان از سازگاری آن با قواعد معنایی زبان است، مانند بررسی نوع داده‌ها و انطباق با اصول دامنه متغیرها.

5. تولید کد میان‌مرحله‌ای (Intermediate Code Generation):

- در این مرحله، کامپایلر کد منبع را به یک فرم میانی تبدیل می‌کند که مستقل از ماشین است. این فرم می‌تواند بهینه‌سازی شده و به کد ماشین تبدیل شود.

6. بهینه‌سازی کد (Code Optimization):

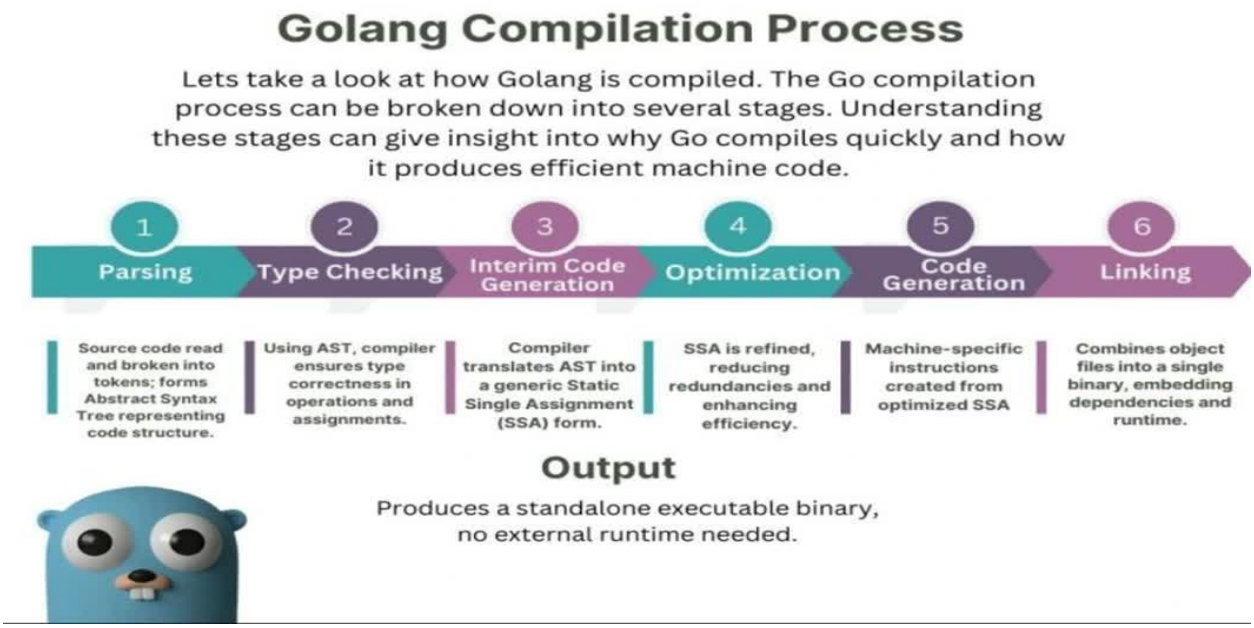
- در این مرحله، کد میانی بهینه‌سازی می‌شود تا کارایی برنامه افزایش یابد، مانند کاهش تعداد دستورات یا بهبود استفاده از حافظه.

7. تولید کد ماشین (Code Generation):

- این مرحله شامل تبدیل کد میانی به کد ماشین قابل اجرا بر روی سخت‌افزار هدف است.

8. لینک کردن (Linking):

- در نهایت، بخش‌های مختلف کد کامپایل شده با کتابخانه‌ها و سایر فایل‌های باینری ترکیب می‌شوند تا فایل اجرایی نهایی ایجاد شود.



فرآیند کامپایل یک برنامه C# شامل چندین مرحله است که هر کدام نقش خاصی در تبدیل کد منبع به کد ماشین ایفا می‌کنند. در زیر مراحل این فرآیند به تفصیل توضیح داده شده است:

1. نوشتن کد منبع:

- برنامه‌نویس کد C# را در یک فایل با پسوند .cs می‌نویسد.

2. پیش‌پردازش (Preprocessing):

- در این مرحله، preprocessors (پیش‌پردازنده‌ها) دستوراتی که با # شروع می‌شوند، مانند #define و #include را پردازش می‌کنند. این مرحله در C# به نسبت C یا C++ کمتر مشهود است.

3. تجزیه و تحلیل نحوی (Lexical Analysis و Syntax Analysis):

- کد منبع تجزیه و تحلیل می‌شود تا ساختار نحوی آن بررسی شود. تجزیه‌گر (Parser)، کد را به درخت ساختاری (Syntax Tree) تبدیل می‌کند.

4. ایجاد درخت معنایی (Semantic Tree):

- بعد از تجزیه، درخت نحوی به درخت معنایی تبدیل می‌شود که اطلاعات بیشتری درباره نوع داده‌ها و سایر ویژگی‌ها دارد.

5. تبدیل به IL (Intermediate Language):

- کد به زبان میان‌افزار (Intermediate Language یا IL) تبدیل می‌شود که یک زبان سطح پایین است و مستقل از نوع سخت‌افزار است.

6. کامپایلر JIT (Just-In-Time Compilation):

- زمانی که برنامه اجرا می‌شود، کامپایلر JIT IL را به کد ماشین مخصوص پلتفرم مقصد تبدیل می‌کند. این مرحله فقط در زمان اجرای برنامه انجام می‌شود.

7. اجرای برنامه:

- بعد از اینکه کد در زمان اجرا به کد ماشین تبدیل شد، برنامه شروع به اجرا می‌کند و می‌تواند در صورت نیاز به منابع دیگر (مانند پایگاه‌داده‌ها) دسترسی پیدا کند.

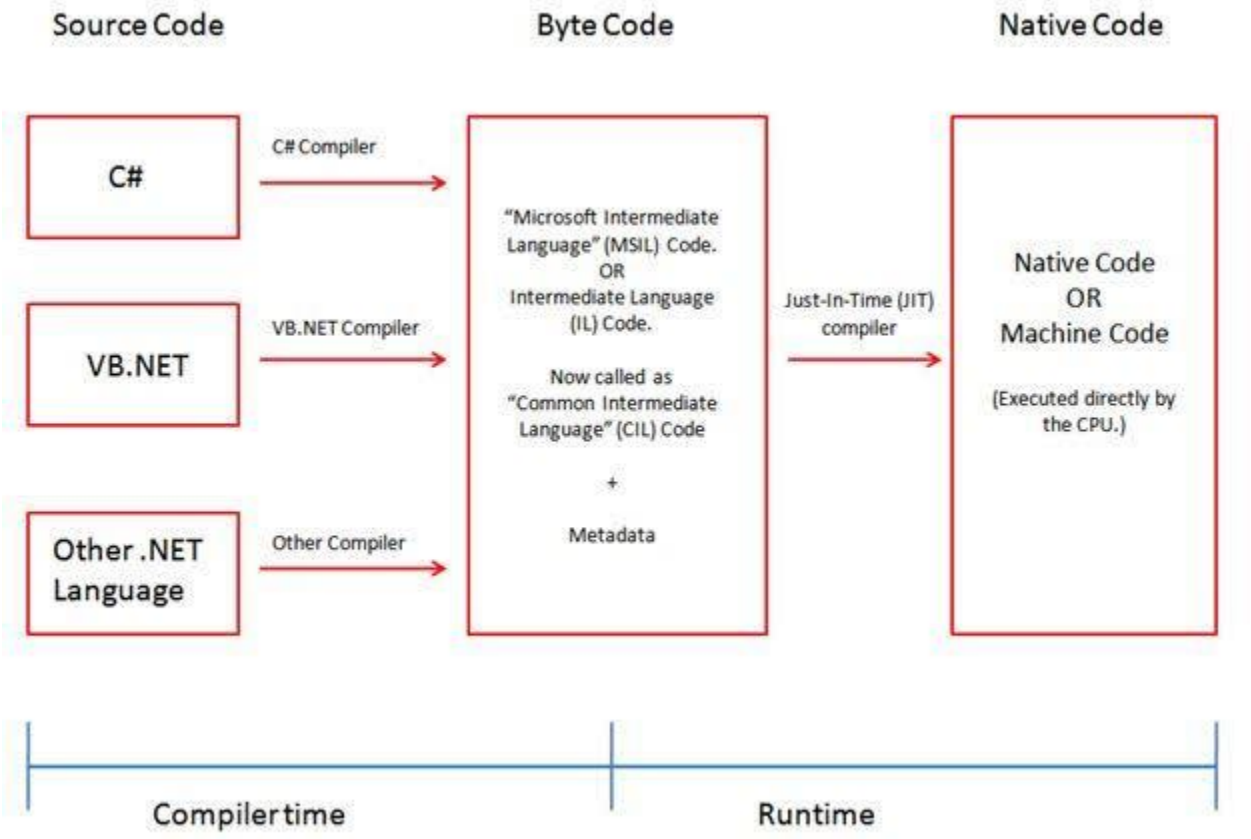
8. مدیریت حافظه:

- CLR (Common Language Runtime) به مدیریت حافظه و garbage collection کمک می‌کند تا از نشت حافظه جلوگیری شود.

9. پیغام‌های خطا و اشکال‌زدایی:

- اگر خطاهایی در زمان کامپایل یا اجرا وجود داشته باشد، کامپایلر پیغام‌های خطا را به برنامه‌نویس گزارش می‌دهد تا مشکلات شناسایی و اصلاح شوند.

این مراحل به فرآیند کلی کامپایل و اجرای برنامه‌های C# کمک می‌کند و به برنامه‌نویس این امکان را می‌دهد که کد خود را به صورت مؤثر و کارآمد توسعه دهد.



۳)فرآیند ترجمه و اجرا در زبان های مفسری را بررسی کنید.

زبان‌های مفسری (Interpreted Languages) زبان‌هایی هستند که کد آن‌ها به صورت مستقیم اجرا می‌شود، بدون این که ابتدا به زبان ماشین یا کد ماشین تبدیل شوند. این فرآیند شامل چندین مرحله است که به طور خلاصه به شرح زیر است:

1. تحلیل لغوی (Lexical Analysis):

- در این مرحله، کد منبع به توکن‌ها (Tokens) تقسیم می‌شود. هر توکن می‌تواند شامل یک کلمه کلیدی، یک شناسه، یک عملگر یا یک نشانه‌گذاری (مانند پرانتز) باشد. مفسر با استفاده از یک تحلیل‌گر لغوی (Lexical Analyzer) این کار را انجام می‌دهد.

2. تحلیل نحوی (Syntax Analysis):

- تحلیل نحوی یا تجزیه (Parsing) ساختار توکن‌های تولید شده را بررسی می‌کند تا مطمئن شود که کد مطابق با قواعد نحوی زبان است. این مرحله معمولاً شامل ساختن یک درخت نحوی (Syntax Tree) است.

3. تحلیل معنایی (Semantic Analysis):

- در این مرحله، مفسر بررسی می‌کند که آیا دستورات کد معنای منطقی دارند یا خیر. این شامل بررسی سازگاری نوع داده‌ها، متغیرهای تعریف نشده و دیگر قوانین معنایی است.

4. تولید کد میانی (Intermediate Code Generation):

- برخی مفسرها ممکن است کد منبع را به یک فرم میانی تبدیل کنند که ساده‌تر قابل اجرا باشد. این کد میانی ممکن است بهینه‌سازی شود تا اجرای بهتری داشته باشد.

5. اجرای کد (Code Execution):

- در نهایت، کد میانی یا مستقیماً کد منبع توسط یک ماشین مجازی یا مفسر اجرا می‌شود. این مرحله شامل تفسیر مستقیم دستورات و اجرای آن‌ها توسط پردازنده است.

6. مدیریت حافظه و خطایابی (Memory Management and Debugging):

- مفسرها اغلب شامل مدیریت حافظه هستند که تخصیص و آزادسازی حافظه را کنترل می‌کند. همچنین، ابزارهای خطایابی برای شناسایی و رفع خطاهای زمان اجرا ارائه می‌شوند.

۴)ساختمان داده AST(Abstract Syntax Tree) را با ذکر مثال توضیح دهید.

ساختمان داده AST (Abstract Syntax Tree) یا درخت نحوی انتزاعی، یک نمایش درختی از ساختار نحوی یک کد منبع است. این ساختار به طور گسترده در کامپایلرها و مفسرهای زبان‌های برنامه‌نویسی برای تحلیل و پردازش کد استفاده می‌شود.

ویژگی‌های کلیدی AST:

1. نمایش انتزاعی: برخلاف درخت‌های نحوی معمولی که تمام جزئیات نحوی را حفظ می‌کنند، AST فقط ساختار معنایی کد را نشان می‌دهد.

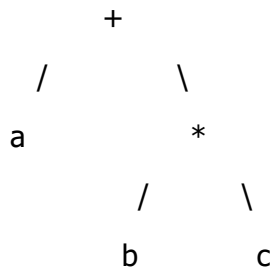
2. ساختار درختی: گره‌های درخت نشان‌دهنده ساختارهای نحوی و عملیاتی هستند، و شاخه‌ها نشان‌دهنده رابطه بین این ساختارها.

مثال:

فرض کنید عبارت زیر را داریم:

a + b * c

ساختار AST برای این عبارت:



- گره ریشه (+): نشان‌دهنده عملیات جمع است.
- گره‌های فرزندان (a و *): نشان‌دهنده عملوندها هستند.
- گره *: خود یک عملیات است که دو عملوند (b و c) دارد.

کاربردهای AST:

- تجزیه و تحلیل کد: کامپایلرها از AST برای بررسی صحت نحوی و معنایی کد استفاده می‌کنند.
- بهینه‌سازی کد: به کمک AST می‌توان کد را بهینه‌سازی کرد، مانند حذف عملیات‌های تکراری یا ساده‌سازی عبارات.
- ایجاد ابزارهای توسعه: مانند تحلیلگرهای استاتیک یا ابزارهای بازسازی کد.

۵)ابزار LLVM چیست ؟ مراحل پیاده سازی یک زبان برنامه نویسی درآن را توضیح دهید.

LLVM (Low-Level Virtual Machine) یک زیرساخت برای بهینه‌سازی کامپایلر و مجموعه‌ای از ابزارها است که برای توسعه زبان‌های برنامه‌نویسی مورد استفاده قرار می‌گیرد. این زیرساخت به توسعه‌دهندگان اجازه می‌دهد تا کامپایلرهایی بسازند که کد را برای معماری‌های مختلف بهینه‌سازی کنند. LLVM به دلیل طراحی ماژولار و انعطاف‌پذیر آن، بسیار محبوب است.

مراحل پیاده‌سازی یک زبان برنامه‌نویسی در LLVM به طور کلی شامل مراحل زیر است:

1. طراحی زبان:

- ابتدا باید نحو و معناشناسی زبان جدید را طراحی کنید. این شامل تعریف قواعد گرامری و معنای هر ساختار زبانی است.

2. نوشتن تجزیه‌گر (Parser):

- تجزیه‌گر کد منبع را به ساختار داده‌ای تبدیل می‌کند که به عنوان Abstract Syntax Tree (AST) شناخته می‌شود. ابزارهایی مانند ANTLR یا Flex/Bison می‌توانند در این مرحله کمک کنند.

3. تولید AST:

- بعد از تجزیه، باید AST تولید شود که نمایشی از ساختار نحوی برنامه است.

4. تولید IR (Intermediate Representation):

- LLVM از IR به عنوان یک نمایش میانی استفاده می‌کند که مستقل از معماری است. باید کد AST را به IR تبدیل کنید. این مرحله شامل نوشتن کدهایی است که هر ساختار زبانی را به معادل آن در IR تبدیل می‌کند.

5. بهینه‌سازی:

- یکی از نقاط قوت LLVM توانایی آن در بهینه‌سازی کد است. می‌توانید از بهینه‌سازهای داخلی LLVM استفاده کنید یا بهینه‌سازی‌های خاص خود را پیاده‌سازی کنید.

6. تولید کد ماشین:

- در نهایت، باید IR به کد ماشین مناسب برای معماری هدف تبدیل شود. LLVM این امکان را به صورت پیش‌فرض فراهم می‌کند.

7. لینک و اجرا:

- کد ماشین تولید شده باید با استفاده از لینک‌کننده به یک فایل اجرایی تبدیل شود. این فایل اجرایی سپس می‌تواند روی معماری هدف اجرا شود.

۶)رویکرد طراحی CISC, RISC را توضیح دهید و مزایا و معایب و کاربرد هر یک را بیان کنید.

رویکرد طراحی RISC (Reduced Instruction Set Computer) به یک معماری پردازنده اشاره دارد که بر استفاده از یک مجموعه دستورات محدود و ساده متمرکز شده است. این طراحی با هدف بهینه‌سازی کارایی و سادگی در پردازش داده‌ها توسعه یافته است. در ادامه به جنبه‌های مختلف طراحی RISC پرداخته می‌شود:

۱. ویژگی‌های اصلی RISC

- مجموعه دستورالعمل ساده: معماری RISC به مجموعه‌ای از دستورالعمل‌ها با عملکردهای ساده و مشخص متکی است. این دستورات معمولاً به گونه‌ای طراحی شده‌اند که در یک چرخه ساعت پردازش شوند.

- تعدد ثبات‌ها: استفاده از تعداد بیشتری ثبات (Registers) به پردازنده RISC اجازه می‌دهد تا داده‌ها را به سرعت پردازش کند و نیاز به دسترسی مکرر به حافظه را کاهش دهد. این امر به اجرای سریع‌تر دستورالعمل‌ها کمک می‌کند.

- معماری با طول ثابت: دستورالعمل‌ها معمولاً دارای طول ثابت هستند که این امر تجزیه و تحلیل و پردازش آن‌ها را برای پردازنده آسان‌تر می‌کند.

۲. مزایای طراحی RISC

- عملکرد بالا: با کاهش تعداد چرخه‌های ساعتی برای اجرای هر دستور، RISC می‌تواند عملکرد بالاتری را فراهم کند. بسیاری از دستورات می‌توانند در یک زمان اجرا شوند.

- سازگاری با کامپایلرها: طراحی ساده قابلیت تعریف و بهینه‌سازی آسان‌تری را برای کامپایلرها فراهم می‌کند. این امر به برنامه‌نویسان اجازه می‌دهد تا بدون نگرانی از جزئیات پردازنده، برنامه‌های بهینه‌ای ایجاد کنند.

- کاهش پیچیدگی سخت‌افزاری: به دلیل استفاده از دستورات ساده، طراحی سخت‌افزار RISC به طور معمول ساده‌تر و با هزینه‌های پایین‌تر همراه است.

۳. معایب طراحی RISC

- بزرگ‌تر شدن برنامه‌ها: به دلیل اینکه هر وظیفه ممکن است نیاز به چندین دستور داشته باشد، اندازه کلی برنامه‌ها ممکن است افزایش یابد.

- پیچیدگی در برنامه‌نویسی: برای اجرای برخی از وظایف پیچیده، ممکن است لازم باشد که برنامه‌نویسان از چندین دستورالعمل استفاده کنند، که این امر می‌تواند باعث افزایش زمان توسعه نرم‌افزار شود.

۴. کاربردهای RISC

- پردازنده‌های موبایل: RISC به دلیل کارایی و مصرف کم انرژی در پردازنده‌های موبایل مانند ARM رایج است.

- سیستم‌های توکار: کاربردهای RISC در سیستم‌های توکار (Embedded Systems) به دلیل نیاز به پردازش سریع و مصرف کم انرژی بسیار شناخته شده است.

- زیرساخت‌های رایانش ابری: برخی از سرورهای ابری نیز از معماری RISC برای اجرای بارهای کاری بهینه استفاده می‌کنند.

رویکرد طراحی RISC با تمرکز بر سادگی و کارایی بالا باعث ایجاد پردازنده‌هایی می‌شود که قادر به اجرای سریع و مؤثر دستورات هستند. این معماری به ویژه در حوزه‌های فناوری اطلاعات و ارتباطات به دلیل مصرف پایین انرژی و کارایی بالا بسیار محبوب است.

رویکرد طراحی CISC (Complex Instruction Set Computer) به معماری پردازنده‌هایی اشاره دارد که از مجموعه‌ای غنی و پیچیده از دستورالعمل‌ها استفاده می‌کنند. این طراحی به گونه‌ای است که می‌تواند وظایف پیشرفته‌تری را با استفاده از تعداد کمتری از دستورالعمل‌ها نسبت به RISC انجام دهد. در ادامه به بررسی جزئیات این رویکرد پرداخته می‌شود:

۱. ویژگی‌های اصلی CISC

- مجموعه دستورالعمل گسترده: CISC دارای مجموعه‌ای از دستورالعمل‌ها با عملکردهای مختلف و پیچیده است. این دستورات می‌توانند به صورت مستقیم عملیات پیچیده‌تری مانند ضرب و تقسیم و یا عملیات ریاضی و منطقی را انجام دهند.

- طول متغیر دستورات: در معماری CISC، دستورات ممکن است طول‌های مختلفی داشته باشند. این ویژگی به برنامه‌نویسان امکان می‌دهد تا با نوشتن دستورات پیچیده در یک خط، از حجم کد کاسته و کارایی را افزایش دهند.

- اجرای چندین عمل: CISC می‌تواند چندین عمل را در یک دستور انجام دهد. به عنوان مثال، یک دستور می‌تواند علاوه بر بارگذاری داده، آن را پردازش کرده و نتایج را ذخیره کند.

۲. مزایای طراحی CISC

- کاهش فضای کد: با استفاده از تعداد کمتری از دستورالعمل‌های پیچیده، می‌توان کدهای طولانی را با دستورات کوتاه‌تر و کارآمدتر نوشت. این امر منجر به کاهش نیاز به حافظه می‌شود.

- ساده‌تر بودن برنامه‌نویسی: وجود دستورات پیچیده و توابع از پیش تعریف شده، برنامه‌نویسی را برای توسعه‌دهندگان تسهیل می‌کند و می‌تواند زمان توسعه نرم‌افزار را کاهش دهد.

- انعطاف‌پذیری بیشتری: CISC به برنامه‌نویسان اجازه می‌دهد تا از دستورات خاص و پیچیده به آسانی استفاده کنند و موارد پیچیده را در یک خط برنامه‌نویسی انجام دهند.

۳. معایب طراحی CISC

- پیچیدگی طراحی: طراحی یک سیستم CISC به دلیل گستردگی و پیچیدگی دستورالعمل‌ها و نیاز به سخت‌افزار بیشتر برای پردازش آنها، به مراتب پیچیده‌تر است.

- زمان بیشتر برای اجرا: برخی از دستورات CISC ممکن است نیاز به چندین چرخه ساعت داشته باشند که می‌تواند عملکرد کلی پردازنده را کاهش دهد.

- مشکلات با پیش‌بینی دستگاه: به دلیل وجود دستورالعمل‌های غیرقابل پیش‌بینی، پردازنده‌های CISC ممکن است در بهینه‌سازی استفاده از حافظه و مخازن دچار چالش شوند.

۴. کاربردهای CISC

- پردازنده‌های شخصی و سرورها: معماری CISC معمولاً در پردازنده‌های دسکتاپ و سرور، مانند پردازنده‌های Intel و AMD، به کار می‌رود. این پردازنده‌ها به دلیل توانایی در انجام عملیات پیچیده و قدرت پردازش بالا در بین کاربران محبوب هستند.

- سیستم‌های ویندوز و بی‌سیم: از آنجا که عموماً سیستم‌عامل‌های پیچیده و برنامه‌های کاربردی سنگین به پردازش در سطح بالا نیاز دارند، CISC به عنوان یک گزینه مناسب در این زمینه شناخته می‌شود.

رویکرد طراحی CISC بر استفاده از یک مجموعه بزرگ و پیچیده از دستورالعمل‌ها متمرکز است تا بتواند عملیات پیچیده را به راحتی انجام دهد. این معماری به ویژه در پردازنده‌هایی که به عملکرد بالا و قابلیت پردازش فعالیت‌های پیچیده نیاز دارند، بسیار مؤثر است .