# به نام خدا

**پروژه: پیاده‌سازی ماشین مانو با یک زبان سطح بالا**

**زبان برنامه نویسی: پایتون**

**پدیدآورندگان: گروه گزماهی(علیرضا احمدی وشکی، سید عرفان اعتصامی)**

**در این گزارش ابتدا چیکیده‌ای از کد موجود در ماژول‌ها و سیر فکری ما در نوشتن این پروژه، و سپس به عنوان یک مثال، یک قطعه کد از برنامه‌ای که طوری نوشته شده تا هم مفهوم تابع و حلقه را شامل بشود و هم تمامی دستورات هسته(به جز `INPR/OUTR instructions`) را در بر بگیرد آورده شده است.**

# چکیده‌ای از نحوه‌ی کارکرد برنامه

سوای ماژول‌های `__init__` و `errors` که به ما مربوط نیست، ما سه ماژول `core` ، `components` و `compiler` را نوشته‌ایم.

*Compiler* : در این ماژول ما برنامه‌ی نوشته شده را به زبان ماشین تبدیل میکنیم.

بعد این ماشین‌کدها را به `core` میدهیم تا در خانه‌های اول حافظه ذخیره کند.

*Components*: در این ماژول ما کلاسی از اشیای `core` مان میسازیم.

۱. `ALU`: در این کلاس، تقریبا تمام عملیات‌های منطقی و حسابی مورد نیاز `core` مان را نوشته‌ایم.
۲. `Memory`: یک حافظه که با لیست ذخیره شده است و ایندکس نشان دهنده‌ی ادرس و مقدار در آن ایندکس نیز منطقا مقدار حافظه در آن آدرس است. نوشتن در حافظه و خواندن از حافظه، ردیابی سلول‌های فعال و نشان دادن آن‌ها به صورت کلی در ترمینال نیز از قابلیتهای آبجکتهای تولید شده از این کلاس است.
۳. `Register`: اسم، سایز و مقدار یک رجیستر. با قابلیت تغییر مقدار یا خواندن مقدار آن و یا چاپ اطلاعات کلی از یک رجیستر.

*Core*: این ماژول حاوی کلاس هسته‌ی ما و در تابع `main()` حاوی برنامه‌ی نوشته شده توسط کاربر است.

مرحله‌ی اول: در همان ابتدا در تابع `main()` یک آبجکت از این کلاس میسازیم. این آبجکت موقع ساخته شدن در `__init__` توسط آبجکت‌های تولید شده از کلاس‌های موجود در ماژول کامپونتنت که ایمپورت شده است، مقداردهی اولیه میشود.

مرحله‌ی دوم: تابع‌های کار با این آبجکت‌ها را می‌نویسیم. نوشتن یا خواندن به صورت سلولی یا کلی در حافظه. و نوشتن یا خواندن داده‌ها از رجیسترها و متدی برای چاپ کردن مقادیر فعلی رجیسترها.

مرحله‌ی سوم: در متد `compile` که به عنوان یک متد در آبجکت تولید شده از این کلاس قرار دارد، ما یک لیست از دستورات که کاربر نوشته است را میگیریم و با استفاده از ماژول ایمپورت شده‌ی `compiler` شروع به کامپایل کردن این دستورات و بعد ذخیره‌ی آن در `n` خانه‌ی اول حافظه‌ی اصلی‌مان میکنیم. به طوری که `n` تعداد خطوط برنامه یا تعداد اینستراکشن‌هاست.

مرحله‌ی چهارم: متد `fetch_and_decode` را داریم که این متد به صورت خودکار تمام برنامه‌ی نوشته شده را ایکزیکیوت میکند.

- توقف و شروع شدن برنامه: در آن یک حلقه‌ی بینهایت است که فقط زمانی از کار می‌ایستد که به یک سلول خالی در حافظه برسد.(یکی از ضعف‌های برنامه همین است، مثلا اگر کاربر در یکی از سلول‌های میان برنامه مقدار صفر را بارگذاری کند، برنامه موقع رسیدن به آن از کار می‌ایستد.) اما با وجود ایستادن برنامه در خانه‌های خالی از حافظه(که به خاطر رخ دادن یک خطاست)، من تصمیم گرفتم از یک روش منطقی‌تر و تمیزتر برای پایان دادن به حلقه استفاده کنم، یک سینتکس تعریف کردم که کاربر حتما دستور `END` را در پایان اینستراکشن‌ها و خط‌های برنامه‌اش برای نشان دادن خانه‌ی پایان برنامه بنویسد. این دستور به صورت ماشین کد `0x000` کامپایل شده و برنامه وقتی به این اینستراکشن برسد از حلقه خارج میشود.

- ساختار درونی آن : ابتدا با توجه به رجیسترهای **AR** و **PC** سلول حافظه‌ی اینستراکشن را تشخیص میدهیم. بعد با این ادرس به اصطلاح اینستراکشن مورد نیاز را فچ می‌کنیم. اکر این اینستراکشن دستور پایان(**END**) بود از حلقه بریک می‌کنیم و برنامه تمام است. در غیر این صورت شروع به دیکود کردن اینستراکشن به دو بخش **Opcode** و **Operand** میکنیم. با استفاده از **Opcode** تشخیص میدهیم که دستور رجیستر بیسد است یا مموری بیسد(این برنامه اینستراکشن‌های مربوط به ورودی و خروجی هسته را شامل نمی‌شود.) بعد تابع مناسب را فراخوانی میکنیم و **Operand** را برای اینستراکشن‌های رجیستر بیسد و برای اینستراکشن‌های مموری بیسد علاوه بر آن **Opcode** را نیز به عنوان پارامتر ورودی به آن تابع میدهیم.

مرحله‌ی پنجم: در این مرحله یا تابع مموری بیسد اینستراشکن فراخوانی شده یا رجیستر بیسد.

- تابع رجیستر بیسد: دستورات ساده‌ای را انجام میدهد که فقط با رجیسترهای **AC** و **E** سر و کار دارد. با توجه به **OPERAND** تابع مناسب فراخوانی شده و مقدار این رجیسترها را و حتی در مواقع نیاز(اسکیپ‌ها) رجیستر **PC** را تغییر میدهد. دستورات منطقی و حسابی با **ALU** اجرا میشوند.
  - نکته: دستور **HLT** برایمان نامفهوم بود و از آن مطمین نبودیم، پس تصمیم گرفتیم که وقتی این دستور خوانده شد، برنامه یک اررو بدهد و روند پردازشی **CPU** متوقف شود.
- تابع مموری بیسد: این تابع از لحاظ پیاده‌سازی پیچیده‌تر بود. چندین معیار برای کنترل وجود داشت. اول با توجه به **Opcode** ایندایرکت یا دایرکت بودن اینستراکشن را مشخص میکنیم. اگر ایندایرکت باشد به یک مرحله بیشتر برای خواندن **Operand** از حافظه نیاز داریم.
  - نکته: سعی کردم تا حد مکان با منطق ماشین مانو پیش بروم(از لحاظ استفاده کردن از رجیسترها) و همینطور برای عملیات‌ها از متدهای خود **CORE** و آبجکت‌هایش استفاده کنم.
- از لحاظ نوع، این اینستراکشن‌ها را به چهار بخش تقسیم می‌کنم:
  - ✓ **AND & ADD** : این دستورات بیشتر با **ALU** سروکار دارند.
  - ✓ **LDA & STA**: برای بارگذاری یا ذخیره‌کردن مقدار در **AC**.
  - ✓ **BUN & BSA**: دستوراتی که پیچیده‌ترین اینستراکشن‌ها در تمام این پروژه از لحاظ پیاده سازی بودند(در واقع هماهنگ کردن این دستورات با تابع **fetch_and_decode** و همینطور حافظه کمی سخت بود).
  - ✓ **ISZ**: در مثال نشان داده شده در ادامه‌ی این گزارش کاربردی از این دستور برای ساختن یک حلقه را خواهیم دید.(مکمل با دستور**BUN** )

مرحله‌ی ششم: ساخت **core** تمام است. حال به تابع **main()** میرسیم تا از **core** استفاده کنیم. آبجکتی از **core** ساخته میشود، چون در این مثال در اینستراکشن‌ها **BSA** داریم، پس به صورت دستی در حافظه در خانه‌های مورد نیاز برای این دستوراتی قرار می‌دهیم تا حالت یک تابع شکل بگیرد. همچنین چند دیتا در حافظه برای استفاده در برنامه قرار داده‌ایم. بعد هم برنامه‌ای که کاربر نوشته است را به صورت لیستی از رشته‌ها (که هر رشته نشان دهنده‌ی یک اینستراکشن است) به متد **compile** ابجکت هسته‌مان می‌دهیم. این دستورات کامپایل شده و در خانه‌های اول حافظه ذخیره میشوند. بعد برای ایکیزیکیوت کردن برنامه کامپایل شده، برای آبجکت هسته‌مان متد **fetch_and_decode** را فراخوانی می‌کنیم.

**چند نکته‌ی مهم: با توجه به برنامه‌ی فشردمون به خاطر پروژه‌های درس برنامه‌سازی پیشرفته و بقیه درس‌ها(که بسیار وقت‌گیر بودند)، کنترل خطا تا حد مطلوب خودش نرسید. یعنی نوشتن اینستراکشن‌ها تا حدی که میشد انعطاف‌پذیر و داینامیک نیست(در واقع یک ایده این است که با استفاده از کتابخانه‌ی PANDAS برای برنامه‌ی نوشته شده تا حدودی در همان ابتدا از وقوع خطا جلوگیری کرد اما متاسفانه وقت نبود).**

**پس نحوه‌ی نوشته شدن بسیار مهم است:**

- **حتما برنامه با دستور END تمام شود(یعنی آخرین عنصر از لیست ورودی `c.compile` رشته‌ی "END" باشد).**
- **خانه‌های حافظه به صورت هگزادسیمال(نه دسیمال) نوشته شوند.**
- **در اینستراکشن‌های این برنامه نمیتوان از عدد منفی استفاده کرد(در واقع فقط میتوان در حافظه ذخیره کرد و بعد آن را LOAD کرد).**
- **در خروجی ترمینال، بعد از اجرای هر دستور اطلاعات آن دستور و همینطور رجیسترها و سلول‌های فعال حافظه چاپ شده است. برای دیدن خروجی و حاصل اجرای هر دستور این برنامه را در ترمینال اجرا کنید.**

```
466  def main():
467      # example
468      c = Core()
469      c.memory_write({0x70: "0x47", # 0x49 = 73
470                      0x48: "0x1132", # -> ADD instruction
471                      0x49: "0x0fff",  # -> AND instruction
472                      0x4a: "0xC047", # indirect BUN to (0x49 = 73)
473
474                      # using this variable for looping
475                      # with ISZ and BUN
476                      0x90: "-1",
477
478                      # Stored data
479                      0x100: "-1",
480                      0x101: "0xffff",
481                      0x102 : "0x5b13"
482                      })
```

```
c.compile([
            "1 BSA 70", # indirect BSA to (0×70)th cell of memory

            # ----- demonstrating skip instructions for AC ---
            "CLA",
            "INC", # AC > 0
            "SPA",
            "INC", # skiping
            "CLA",
            "SZA", # AC = 0
            "INC", # skiping
            "0 LDA 100", # AC < 0 | -> LDA instruction
            "SNA",
            "INC", # skiping

            # ---------- demonstrating E instructions ------
            # making carry
            "0 LDA 101", # -> LDA instruction
            "0 ADD 1", # -> ADD instruction
            # instructions
            "CME", # complement E
            "CLE", # clear E

            # ------ demonstrating AC instructions -------
            "0 LDA 102",
            "CMA",
            "CIR",
            "CIL",


            "0 STA 60", # -> STA instruction

            # loop making
            # amount of looping = -(value in the 0×90)
            "0 ISZ 90", #  -> ISZ instruction
            "0 BUN 0", # -> BUN instruction
            "END"
            ])
```