

به نام خدا

پروژه: پیاده سازی ماشین مانو با یک زبان سطح بالا

زبان برنامه نویسی: پایتون

پدیدآورندگان: گروه گزماهی (علیرضا احمدی و شکی، سید عرفان اعتصامی)

در این گزارش ابتدا چیکیده ای از کد موجود در ماژول ها و سیر فکری ما در نوشتن این پروژه، و سپس به عنوان یک مثال، یک قطعه کد از برنامه ای 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، تقریباً 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 آدرس و مقدار در آن ایندکس نیز منطقاً 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، سائز 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 کامپوننت‌ها 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 ایکزیکوت 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، مثلاً 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 حتماً دستور `END` را در پایان اینستراکشن‌ها و خط‌های برنامه‌اش برای نشان دادن خانه‌ی پایان برنامه بنویسد. این دستور به صورت ماشین کد `0x00` 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. اگر 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 اینستراکشن 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 مطمئن نبودیم، پس تصمیم گرفتیم که وقتی این دستور خوانده شد، برنامه یک ارور 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 ذخیره کردن 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 ایکیزیکوت 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 وقت گیر بودند)، کنترل خطا تا حد مطلوب خودش نرسید. یعنی نوشتن اینستراکشن‌ها تا حدی که میشد انعطاف‌پذیر و دینامیک 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 متأسفانه 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                })
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
```

c.compile

"1 BSA 70", # indirect BSA to (0x70)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 0x90)

"0 ISZ 90", # -> ISZ instruction

"0 BUN 0", # -> BUN instruction

"END"

)

)

"END"

"0 BUN 0", # -> BUN instruction

"0 ISZ 90", # -> ISZ instruction

amount of looping = -(value in the 0x90)

loop making

"0 STA 60", # -> STA instruction

کلام آخر: این پروژه به صورت تعاملی انجام شد. یعنی با همدیگه ایده پردازی کردیم و یا اینکه این ایده رو چطور باید پیاده کرد فکر کردیم. برای همین مرزهای تسک هامون واضح نیست و اشتراکی هستند. اما اگر به طور کلی و غیر دقیق بخواهیم تقسیم بندی وظایف رو شرح بدیم: سید عرفان اعتصامی: ایده ی ماژول کامپوننت و پیاده سازی کلاس های مموری و رجیستر، نوشتن `__init__` از کلاس `core`، ایده ی متد `فچ` و دیکودینگ، پیاده سازی متد `رجیستر بیس` اینستراکشن و مموری `بیس` اینستراکشن (به جز `BUN BSA ISZ`)

علیرضا احمدی و شکی: پیاده سازی متدهای کلاس `ALU`، ایده و نوشتن کامپایلر، نوشتن بیسیک متدهای کلاس `core`، پیاده و هماهنگ سازی کلاس `فچ` اند دیکود، پیاده سازی (`BUN BSA ISZ`) و هماهنگ سازی رجیستر `PC` در برنامه، نوشتن مثال در تابع `main()`

-موفق باشید