

Lab 1.1

1. Step by step algorithm

(1) Variable definition in *.data* section

First, I need to define the 3 variables *var1*, *var2* and *var3*. Since the type and initial value is not specified, I just set them as *.word* type, and initialize them to 0 (explicitly, for clarity).

Also, since we need to print out the calculated values of *var1*, *var2* and *var3* line by line, we have to print out the new line character. So I additionally defined the new line character *newline* of type *.asciz* and of value `"\n"`.

(2) Read variables from terminal

By loading service number 5 into register *a7* by *li* and executing *ecall*, *ReadInt* is performed and the integer input from terminal is stored in register *a0*. Then I use *la* instruction to store the address of *var1*, *var2* and *var3* in register *t1*, *t2* and *t3* respectively, and use *sw* instruction to store the 3 integers read from terminal to the address *(0)t1*, *(0)t2* and *(0)t3*, so that the value of *var1*, *var2* and *var3* become the value read from terminal.

(3) Increase *var1* by 3

To change the value of *var1*, first I have to load the value of *var1* from the memory to the register, which is done by the *lw* instruction, storing the value at *(0)t1* (*t1* stores the address of *var1* since step (2)) to *t4*. Then I use the *addi* to add an immediate value 3 to *t4*. Finally, the value in *t4* is stored back to memory at *(0)t1* using the *sw* instruction.

(4) multiply *var2* by 2

To change the value of *var2*, first I have to load the value of *var2* from the memory to the register, which is done by the *lw* instruction, storing the value at *(0)t2* (*t2* stores the address of *var2* since step (2)) to *t5*. Since integers are stored in binary in computer, multiply a number by 2 or 0b10 is the same with shift every digit in the number left for 1 digit, which could be done by the *slli* instruction. Finally, the value in *t5* is stored back to memory at *(0)t2* using the *sw* instruction.

(5) Increase *var3* by *var1* + *var2*

To change the value of *var3*, first I have to load the value of *var3* from the memory to the register, which is done by the *lw* instruction, storing the value at *(0)t3* (*t3* stores the address of *var3* since step (2)) to *t6*. Then I use *add* instruction twice to first add *t6* by *t4* (which is the value of increased *var1* since step (3)), then add *t6* by *t5* (which is the value of increased *var2* since step (4)). Finally, the value in *t6* is stored back to memory at *(0)t3* using the *sw* instruction.

(6) Print variables to terminal

By loading service number1 into register *a7* by *li* and executing *ecall*, *PrintInt* is performed and the integer in *a0* will be printed in terminal. So to print the value of *var1*, *var2* and *var3*, I use *lw* instruction to load the value of them to *a0* and perform *ecall* one at a time.

Between printing each variable, we have to print an additional newline character. This is done by loading service number4 (*PrintString*) into register *a7* by *li*, and loading the address of variable *newline* to *a0* by *la*, and finally execute the *ecall* instruction.

(7) Exit program

By loading service number10 into register *a7* by *li* and executing *ecall*, the program will exit with code 0.

2. Main Code:

(1) Variable declaration:

```
.data
var1: .word 0
var2: .word 0
var3: .word 0
newline: .asciz "\n"
```

(2) Variable input

```
### load variable from console
# read input1
li a7, 5
ecall
# put input into var1
la t1, var1      # t1 is the address of var1
sw a0, 0(t1)

# read input2
li a7, 5
ecall
# put input into var2
la t2, var2      # t2 is the address of var2
sw a0, 0(t2)

# read input3
li a7, 5
ecall
# put input into var3
la t3, var3      # t3 is the address of var3
sw a0, 0(t3)
```

(3) Variable update

```

### increase var1 by 3
# addi t4, var1, 3 # add
lw t4, 0(t1)
addi t4, t4, 3
sw t4, 0(t1)      # t4 is value of var1

### multiply var2 by 2
lw t5, 0(t2)
add t5, t5, t5    # multiply
sw t5, 0(t2)      # t5 is value of var2

### increase var3
lw t6, 0(t3)      # t0 is temp sum, now var3
add t6, t6, t4
add t6, t6, t5
sw t6, 0(t3)

```

(4) Variable output

```

### print vars
# print var1
li a7, 1    # set to print integer
lw a0, var1 # set int to print
ecall      # print

# print new line
li a7, 4
la a0, newline
ecall

# print var2
li a7, 1    # set to print integer
lw a0, var2 # set int to print
ecall      # print

# print new line
li a7, 4
la a0, newline
ecall

# print var3
li a7, 1    # set to print integer
lw a0, var3 # set int to print
ecall      # print

```

(5) Program exit

```

# exit
li a7, 10
ecall

```

3. Console results

OneDrive/OneDrive - The Chinese University of Hong Kong/CNHK/y22/CENG420/lab1/lab1_1new.asm - RARS 1.5

File Edit Run Settings Tools Help

Run speed at max (no interaction)

Text Segment

Skpt	Address	Code	Basic	Source
	0x00400000	0x00500893 addi x17,x0,5	14:	li x7, 5
	0x00400004	0x00500072 ecall	15:	ecall
	0x00400008	0x00f01017 auipc x6,64528	17:	la t1, var1 # t1 is the address of v...
	0x0040000c	0x00f03019 addi x6,x6,-9		
	0x00400010	0x00a32023 sw x10,0(x6)	19:	sw a0, 0(t1)
	0x00400014	0x00500093 addi x17,x0,5	21:	li x7, 5
	0x00400018	0x00800073 ecall	23:	ecall
	0x0040001c	0x00f01017 auipc x7,64528	24:	la t2, var2 # t2 is the address of v...
	0x00400020	0x00a38393 addi x7,x7,-24		
	0x00400024	0x00a3a073 sw x10,0(x7)	25:	sw a0, 0(t2)
	0x00400028	0x00900093 addi x17,x0,5	29:	li x7, 5
	0x0040002c	0x00000073 ecall	29:	ecall
	0x00400030	0x00f01017 auipc x20,64528	31:	la t3, var3 # t3 is the address of v...
	0x00400034	0x00f0e013 addi x20,x20,-40		
	0x00400038	0x00a32023 sw x10,0(x20)	32:	sw a0, 0(t3)

Data Segment

Address	Value (x0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0	0	0	0	0	0	0	0
0x10010020	0	0	0	0	0	0	0	0
0x10010040	0	0	0	0	0	0	0	0
0x10010060	0	0	0	0	0	0	0	0
0x10010080	0	0	0	0	0	0	0	0
0x100100a0	0	0	0	0	0	0	0	0
0x100100c0	0	0	0	0	0	0	0	0
0x100100e0	0	0	0	0	0	0	0	0
0x10010100	0	0	0	0	0	0	0	0
0x10010120	0	0	0	0	0	0	0	0
0x10010140	0	0	0	0	0	0	0	0
0x10010160	0	0	0	0	0	0	0	0
0x10010180	0	0	0	0	0	0	0	0

Registers

Name	Number	Value
x0	0	0
x1	1	0
x2	2	2147479548
x3	3	268446824
x4	4	0
x5	5	0
x6	6	268500992
x7	7	268500992
x8	8	0
x9	9	0
x10	10	11
x11	11	0
x12	12	0
x13	13	0
x14	14	0
x15	15	0
x16	16	0
x17	17	10
x18	18	0
x19	19	0
x20	20	0
x21	21	0
x22	22	0
x23	23	0
x24	24	0
x25	25	0
x26	26	0
x27	27	0
x28	28	268501000
x29	29	4
x30	30	4
x31	31	11
x32	32	4194452

Message Window

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
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
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099

```

Lab 1.2

1. Problem analysis

In lab 1.2, we are given a prestored array named *array1* that has 10 elements -1, 22, 8, 35, 5, 4, 11, 2, 1, 78. We want to partition the array with 8 (the 3rd element) as pivot, so that all the numbers smaller than 8 will be stored at the left of it, and all the numbers greater than 8 will be stored at the right of it. In other words, the program output should be -1, 5, 4, 2, 1, 8, 11, 35, 22, 78.

2. Algorithm overview

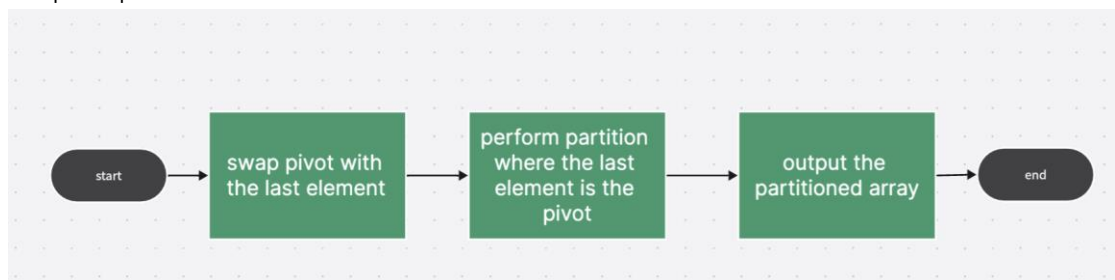
Since in practice, the pivot of the partition mission could be different each time, it is tedious to develop a different partitioning algorithms for different pivots.

So first, no matter where the pivot originally is, I swap it with the last element of the array.

Then, we only need to develop an algorithm to perform partition where the last element is the pivot.

Finally, we output the partitioned array.

Graph representation:



3. Function declaration

In order to increase the make the program more efficient and portable, I declare functions *swap* and *partition*.

Since these functions only involve a limited amount of registers, I think there is no needs to use a stack to store the original values of these registers and load them back.

(1) *Swap*

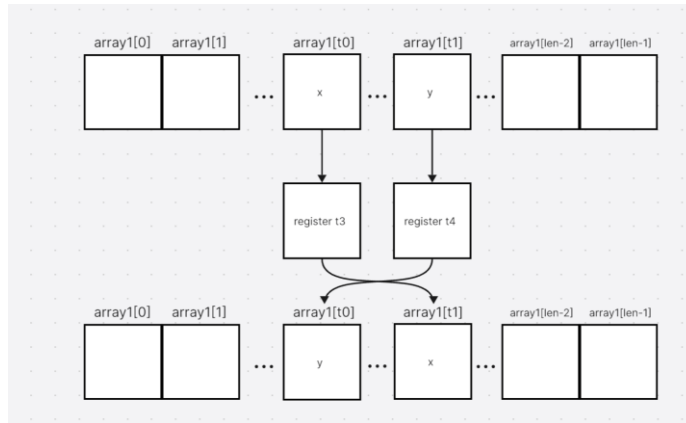
Since in both swapping pivot and partitioning array parts, we need to perform a lot of swapping, I declare a swap function to avoid tedious code.

The function input is the indices of the 2 elements that will be swapped, stored in *t0* and *t1*.

Since the number with index *i* will be in address $array1 + i * 4$, I first left shift *t0*, *t1* by 2 to get the quadruple of them, then I add them with the address of the first element to get the address of *array1[t0]* and *array1[t1]*.

I then load *array1[t0]* and *array1[t1]* to registers *t3* and *t4* respectively using *lw*, then I use *sw* to load the value of *t3* and *t4* to *array1[t1]* and *array1[t0]*, thus done swapping.

The graph representation is shown below:



(2) *partition*

The function *partition* follows the partition pseudo code given in lab 1.2 power point. Throughout the partition process, it maintains two index parameters i and j , so that it always satisfies that (a) all the elements with index $\leq i$ are smaller than pivot (b) all the elements with index $> i$ and $\leq j$ are bigger than pivot. Every iteration, j increases by 1, and the new element $array1[j]$ is compared with pivot. If $array1[j] > \text{pivot}$, no further manipulation is needed to satisfy (a) and (b); else if $array1[j] \leq \text{pivot}$, we increase i by 1 and swap $array1[i]$ and $array1[j]$ to satisfy (a) and (b). Finally, when j reaches the last element before pivot, swap $array1[i + 1]$ with pivot to make all the numbers smaller than pivot stored at the left of it, and all the numbers greater than pivot stored at the right of it.

The pseudo code is shown below:

```
function PARTITION(A, lo, hi)
    pivot  $\leftarrow$  A[hi]
    i  $\leftarrow$  lo-1;
    for j = lo; j  $\leq$  hi-1; j  $\leftarrow$  j+1 do
        if A[j]  $\leq$  pivot then
            i  $\leftarrow$  i+1;
            swap A[i] with A[j];
        end if
    end for
    swap A[i+1] with A[hi];
    return i+1;
end function
```

The function in assembly language is explained below in detail.

(a) It takes the address of array $A(array1)$, the index of the element before pivot (which is the last element) hi , and the index of the first element (which is the 0 in this situation) lo as parameters, read from register $a0$, $a1$, $a2$ respectively. The value of pivot is stored into $a3$, and i , j are initialized and stored in $a4$ and $a5$ respectively.

(b) Then there is a for loop marked by label *start_for* as start and *end_for* as stop.

Inside the for loop, it first checks whether $a5 > a2 - 1$ ($j > hi - 1$) using

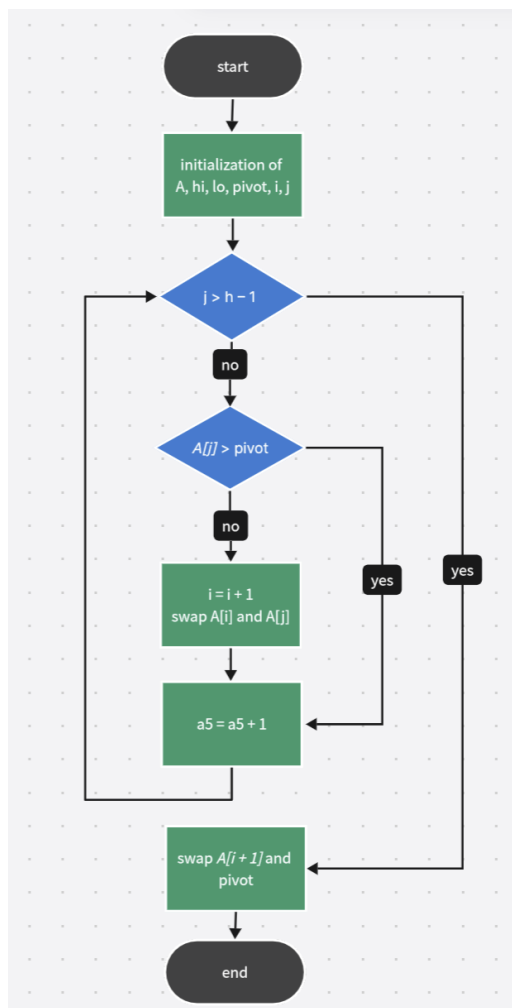
bgt. If the condition is true, it will jump to *end_for* to end the for loop. Else the loop continues and goes into an if-block marked by label *end_if* as stop. It first checks whether $A[j] > a3$ (pivot) using *bgt*. If the condition is true, the if-block is skipped by branching to *end_if*, else it will increase i by 1 and swap the elements with index $a4$ ($A[i]$) and $a5$ ($A[j]$) by calling the *swap* function.

After the if-block, $a5$ increases by 1 and *pc* jumps to *start_for* again.

(c) After the for loop, it will swap elements with index $a4 + 1$ ($A[i + 1]$) and $a2$ (pivot) by calling the *swap* function.

(d) Finally it will return to the main function using *jr*.

The graph representation is shown below:



4. Using functions declared to perform partition

(1) Variable definition in *.data* section

First, to store the elements of *array1*, I put the numbers into consecutive memories by only defining a single *.word* type variable *array1* and put all 10 numbers after it. By this method, the first number will be in address *array1*, and the i^{th} number will be in address $\text{array1} + (i - 1) * 4$. I also created a

variable named *len* to store the length of the array, which is 10, to make the program aware of where is the last element of the array.

Then, we can define a variable *original_pivot_index* to store the index of the pivot, which is 2.

Also, since we need to print out the array line by line, we have to print out the new line character. So I additionally defined the new line character *newline* of type *.asciz* and of value `"\n"`.

- (2) Swap pivot to the last element

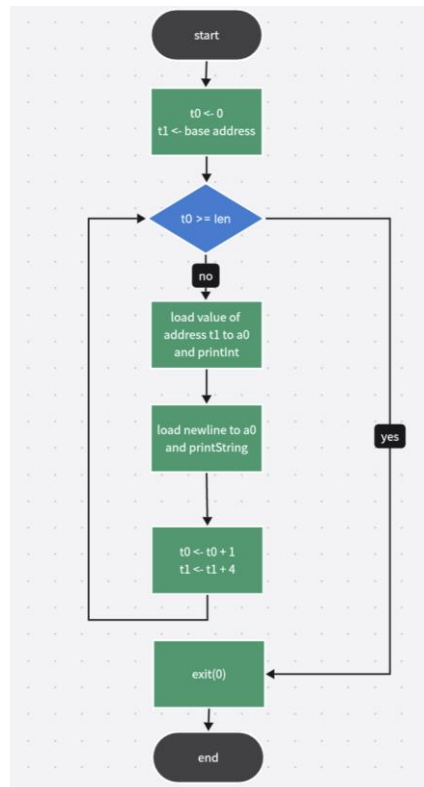
I first set *t0* as *original_pivot_index* and set *t1* as *len* - 1, then I call *swap* function to swap pivot to the last element.

- (3) Perform partition where the last element is the pivot

- (4) I first set *a0*, *a1*, *a2* to *array1*, 0 and *len* - 1 respectively, then I call *partition* to perform partition where the last element is the pivot.

- (5) Output the partitioned array

Using a for loop similar to the one in *partition* function, we can print out the element of the array one by one, each followed with a newline character.



5. Main code

- (1) Variable declaration

```
.data
array1: .word -1, 22, 8, 35, 5, 4, 11, 2, 1, 78
len: .word 10
original_pivot_index: .word 2
newline: .string "\n"
```

- (2) Calling swap and partition function


```

# swap pivot to the last element
lw t0, original_pivot_index      # a0 set to original_pivot_index

lw t1, len
addi t1, t1, -1                  # a1 set to last element index

jal ra, swap                     # swap

# perform partition
la a0, array1                   # a0: address of the array
li a1, 0                        # a1: lo = 0
lw a2, len
addi a2, a2, -1                 # a2: hi = len - 1

jal a6, partition

```

(3) Output the partitioned array using for loop and exit

```

# print
li t0, 0                        # init, t0 == 0
la t1, array1                   # init t1 is the address of current int to print
lw s1, len

## for_loop to output
start_for_out:
bge t0, s1, end_for_out        # check for loop

## load the int to print to a0 and output
lw a0, 0(t1)
li a7, 1                        # printInt
ecall

## output a newline
la a0, newline
li a7, 4                        # printString
ecall

## update
addi t0, t0, 1                  # t0 ++
addi t1, t1, 4                  # t1 = t1 + 4

## jump
j start_for_out
end_for_out:

# exit
li a7, 10
ecall

```

6. Console results

(1) Input

```

array1: .word -1, 22, 8, 35, 5, 4, 11, 2, 1, 78
len: .word 10
original_pivot_index: .word 2

```

(2) Output

D:\OneDrive\OneDrive - The Chinese University of Hong Kong\CUHK\y2s2\CEG3420\lab1\lab1-2.asm - RARS 1.5

File Edit Run Settings Tools Help

Run speed at max (no interaction)

Control and Status

Register	Value
zero	0
ra	4194552
sp	2147479548
gp	268463224
tp	0
t0	5
t1	6
t2	7
a0	8
a1	9
a2	10
a3	11
a4	12
a5	13
a6	14
a7	15
s0	16
s1	17
s2	18
s3	19
s4	20
s5	21
s6	22
s7	23
s8	24
s9	25
s10	26
s11	27
t3	28
t4	29
t5	30
t6	31
pc	4194428

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	-1	5	4	2	1	0	11	35
0x10010020	22	78	10	2	10	0	0	0
0x10010040	0	0	0	0	0	0	0	0
0x10010060	0	0	0	0	0	0	0	0
0x10010080	0	0	0	0	0	0	0	0
0x100100a0	0	0	0	0	0	0	0	0
0x100100c0	0	0	0	0	0	0	0	0

Messages Run I/O

1

Clear

1

5

4

2

1

8

11

35

22

78

-- program is finished running (0) --

-1

5

4

2

1

8

11

35

22

78

-- program is finished running (0) --

Lab 1.3

1. Problem analysis

In lab 1.3, we are given an array from console input. We need to sort it into ascending order and print it out line by line.

The task mainly consists of 3 parts: input, sort and output. The input and output part could be easily done with a for loop similar to the one implemented in lab 1.2. The sorting part can be done with quicksort, and the sorting is a recursion process of first partition the array based on the pivot and recursively sort the 2 subarrays, and the partition part is also similar to the partition function implemented in lab 1.2.

2. Recursive implementation of *qsort* function.

(1) Algorithm overview

Quick sort algorithm follows the divide and conquer principle. Given any array, we can pick a pivot and partition the array into 2 subarrays where the lower array has all the elements that are smaller than pivot and the higher array has all the elements that are bigger than pivot. Then we can do the same quick sort to the 2 subarrays just as what is done to the original array to generate more partitioned subarrays until each subarray has only 1 element. Since all the subarrays are the result of partition, the elements in different subarrays must follow the ascending order. So, when all the subarrays have only one element, the sorting of the whole array is finished.

C code representation of *qsort*:

```
void qsort(int lo, int hi)
{
    // check if end recursion
    if (lo >= hi)
    {
        return;
    }

    int pivot = array[hi];
    int i = lo - 1;

    // partition
    for (int j = lo; j < hi; j++)
    {
        if (array[j] <= pivot)
        {
            i++;
            int temp = array[i];
            array[i] = array[j];
            array[j] = temp;
        }
    }

    int temp = array[i + 1];
    array[i + 1] = array[hi];
    array[hi] = temp;

    // qsort sub-arrays
    qsort(array, lo, i);
    qsort(array, i + 2, hi);
}
```

(2) Detailed implementation in assembly language

(a) Variables used in *qsort*

Throughout *qsort* process, register *a0* always stores the base address of the array to sort. *a1* is the index of the smallest address of the array to partition (*lo* in C code), *a2* is the index of the smallest address of the array to partition (*hi* in C code). *a1* and *a2* are parameters defined by caller. In *qsort*, I always see the last element in the array to partition as pivot, and store its value in *a3*. Similar to the partition algorithm in lab 1.2 *a4*, *a5* stores the index *i*, *j* that marks the end of the partitioned elements smaller and bigger than pivot respectively. *ra* stores the return address of the callee. Besides, some temporary registers are also used in the function but are just temporary such that modifying their values by callee won't have an effect on the caller.

(b) Push the used parameters into stack

Since *qsort* is a recursive function and uses a lot of variables, we need to push the useful values in the registers that *qsort* might use into stack before running the *qsort* function, in order to maintain these values unchanged after the function finishes.

Throughout the code, only 6 registers *ra*, *a1* to *a5* are modified and might contain important values that might be used by caller, so we only need to push these 6 values into stack. This is done by subtracting *sp* by 24 and storing these values one by one to the corresponding address.

(c) Read *a1* and *a2*

a1 and *a2* are variables that are read from caller. Since *qsort* is a recursive function, the caller shouldn't directly store the *a1* and *a2* of the callee directly into *a1* and *a2* because *a1* and *a2* of the caller also have meaning, and modifying them might cause an error in the subsequent instructions. One of the ways to solve this problem is to first store the *a1* and *a2* of the callee in temporary register *t0* and *t1*, and let callee load them to *a1* and *a2* after the callee has already stored the *a1* and *a2* of the caller into stack. By this method, *a1* and *a2* won't change in caller's view since the value is pushed into stack by callee and popped back when callee finishes its program, and thus won't cause an error.

Therefore, after pushing the used parameters into stack, I copy the value of *t0* and *t1* into *a1* and *a2*.

(d) Check if end recursion

If $a2(hi) \leq a1(lo)$, it means that there are at most 1 element in the subarray, thus the partition is finished. So we don't need to do partition and recursive quicksort, we can directly branch to the step to pop the registers out of stack.

(e) Partition the array between *array[lo]* and *array[hi]*

Since we always set the pivot to be the last element of the array, the partition process is exactly the same with the *partition* process done in

lab 1.2.

- (f) Perform *qsort* on 2 subarrays

After partition, all the elements who are smaller than pivot is at the left of pivot and all the elements who is bigger than pivot is at the right of pivot. So we only need to recursively quick sort the 2 subarrays to finish *qsort*.

From the algorithm above, we know that at the end of the partition, the index of *i* is stored in *a4*, so the index of pivot is $a4 + 1$. Therefore, the *lo* of smaller array is *lo* of original array, the *hi* of smaller array is $a4 + 1 - 1 = a4$, the *lo* of bigger array is $a4 + 1 + 1 = a4 + 2$, the *hi* of bigger array is *hi* of original array.

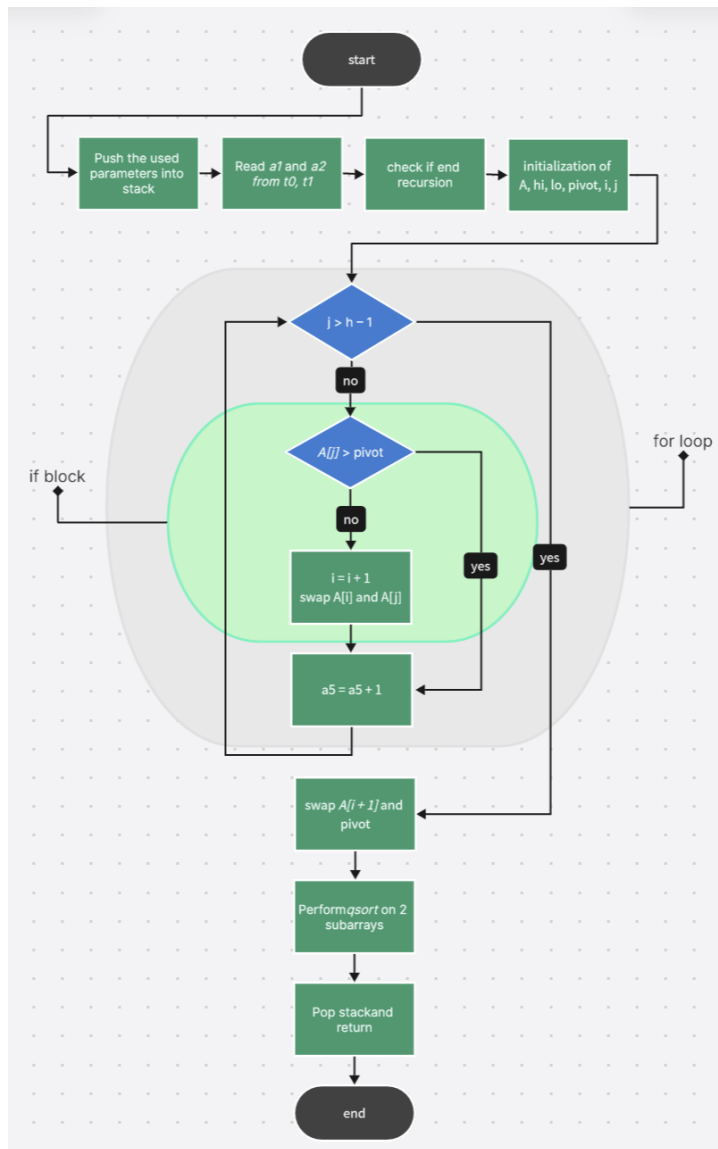
By setting these values to *t0* and *t1*, *qsort* on 2 subarrays can be performed.

- (g) Pop stack and return

After all the instructions of *qsort* are performed, we should pop the values of the registers stored in stack out. This can be done by first use *lw* to load back all the values of registers in (b) and then adding *sp* by 24.

Finally, we can jump to the address stored by *ra* to hand the control to caller.

The graph of *qsort* algorithm is shown below:



3. Main function

(1) Variable definition in `.data` section

Since we need to print out the array line by line, we have to print out the new line character. So I additionally defined the new line character `newline` of type `.asciz (.string)` and of value `"\n"`.

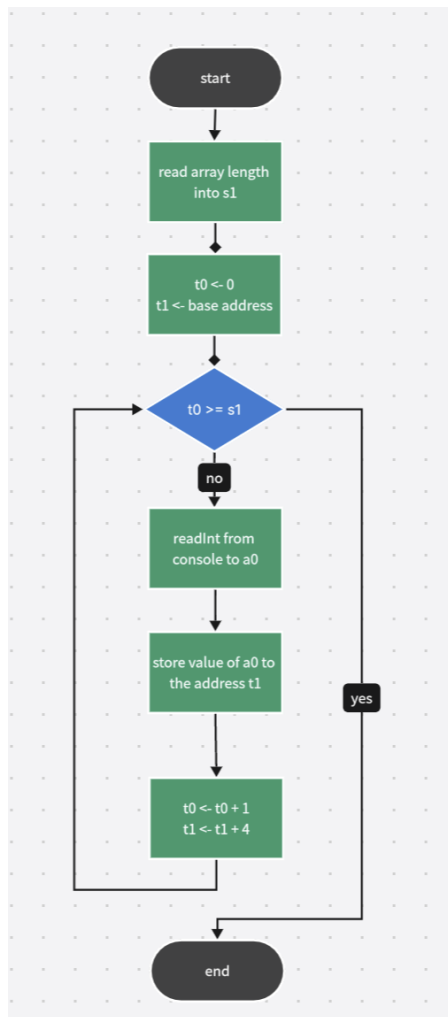
Similar to lab 1.2, we can just declare a single `.word` variable `array` as the first element of the array, then the address of the rest of the elements can be calculated based on the base address and the index of the element.

(2) Input the array from console

I use `s1` to store the length of the array, which is the first input of the console. Similar to lab 1.1, this can be done by loading service number 5 (`readInt`) into register `a7` by `li` and executing `ecall`.

Then with the number of elements that will be inputted from the console known, we can use a for loop to input the array elements also using `readInt`.

The graph representation is shown below:



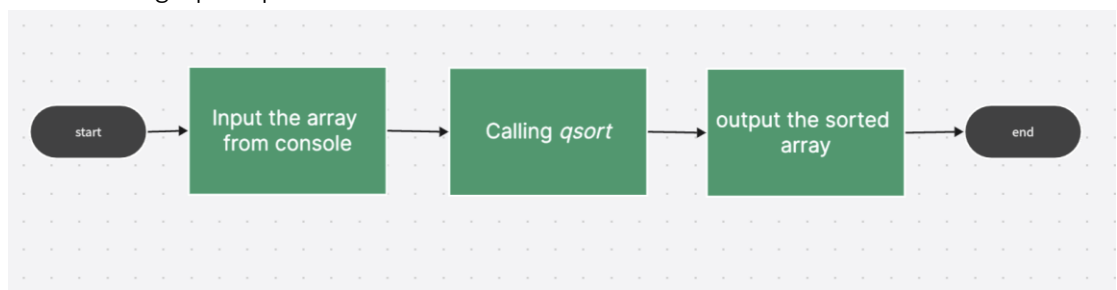
(3) Calling *qsort*

First initialize *a0* to the base address of the array, and initialize the required input parameters *t0* (*lo*), *t1* (*hi*) to 0 and *s1* – 1 respectively. Then call the function *qsort* by *jal* command.

(4) Output the sorted array

This step should be the same with lab 1.2: output the partitioned array.

The overall graph representation is shown below:



4. Main code:

(1) *qsort* function

```

# qsort(hi, lo)
qsort:
    # description: given lo, hi INDEX of an array, sort it in ascending order (assume last element is pivot)
    # input
    ## t0: lo idx
    ## t1: hi idx
    # var_used:
    ## a0: address of array
    ## a1: (from input param t0) lo -- index
    ## a2: (from input param t1) hi -- index
    ## a3: pivot
    ## a4: i -- index
    ## a5: j -- index
    ## temporary registers

    # push
    addi sp, sp, -24
    sw ra, 20(sp)
    sw a1, 16(sp)
    sw a2, 12(sp)
    sw a3, 8(sp)
    sw a4, 4(sp)
    sw a5, 0(sp)

```

```

    # read input param
    mv a1, t0
    mv a2, t1

    # check if end recursion
    ble a2, a1, return_qsort

    # partition
    ## init i
    addi a4, a1, -1

    ## init j
    addi a5, a1, 0

    ## load pivot
    slli t0, a2, 2
    add t0, t0, a0
    lw a3, 0(t0)

    ## for loop
start_for_par:

    ## for loop check
    addi t3, a2, -1
    bgt a5, t3, end_for_par    # j > hi - 1

    ## if check
    slli t0, a5, 2
    add t0, t0, a0
    lw t3, 0(t0)              # t3 = A[j]
    bgt t3, a3, end_if_par    # A[j] > pivot

    ## inside if
    addi a4, a4, 1            # i = i + 1

    mv t0, a4                 # call swap function
    mv t1, a5
    jal ra, swap

end_if_par:
    ## end if
    addi a5, a5, 1
    j start_for_par

```

```

end_for_par:
    ## swap
    addi t0, a4, 1
    mv t1, a2
    jal ra, swap

    # qsort 2 subarrays
    addi t2, a4, 1            # t2 <- pivot_idx = i + 1
    addi t3, t2, -1           # t3 <- the array that is smaller than pivot's hi = t0 - 1
    addi t4, t2, 1            # t4 <- the array that is bigger than pivot's hi = t0 + 1

    ## sort lower array
    mv t0, a1                 # lo_lo = lo
    mv t1, t3                 # lo_hi = pivot_idx - 1
    jal ra, qsort

    ## sort higher array
    mv t0, t4                 # hi_lo = pivot_idx + 1
    mv t1, a2                 # lo_hi = hi
    jal ra, qsort

return_qsort:
    # pop stack
    lw a5, 0(sp)
    lw a4, 4(sp)
    lw a3, 8(sp)
    lw a2, 12(sp)
    lw a1, 16(sp)
    lw ra, 20(sp)
    addi sp, sp, 24

    # return
    jr ra

```


(2) variable declaration

```
.data
space: .string " "
array: .word 0
```

(3) input array

```
# input
## input lenth and store in len
li a7, 5
ecall
mv s1, a0

## input the array
### t0: for_loop counter
### t1: the address of current spot for the array
la t1, array

start_for:
bge t0, s1, end_for      # check for loop, initially t0 == 0

### input, a7 == 5 already
ecall

### store
sw a0, 0(t1)

### update
addi t0, t0, 1           # t0 ++
addi t1, t1, 4           # t1 = t1 + 4

### jump
j start_for
end_for:
```

(4) calling *qsort* function

```
# qsort
la a0, array
li t0, 0                # index of lo
addi t1, s1, -1         # index of hi = s1 - 1
jal ra, qsort           # quick sort
```

(5) output array and exit

```
# output
## vars:
### t0: for_loop counter
### t1: the address of current spot for the array
li t0, 0                # init, t0 == 0
la t1, array            # init t1 is the address of current int to print

## for_loop to output
start_for_out:
bge t0, s1, end_for_out  # check for loop

## load the int to print to a0 and output
lw a0, 0(t1)
li a7, 1                # printInt
ecall

## ouput a space
la a0, space
li a7, 4                # printString
ecall

## update
addi t0, t0, 1          # t0 ++
addi t1, t1, 4          # t1 = t1 + 4

## jump
j start_for_out
end_for_out:

# exit
li a7, 10
ecall
```

5. Console results

DA\OneDrive\OneDrive - The Chinese University of Hong Kong\CUHK\y2s2\CENG3420\lab1\lab1-3\lab1-3.asm - RARS 1.5

File Edit Run Settings Tools Help

Run speed at max (no interaction)

Text Segment

Skpt	Address	Code	Basic	Source
<input type="checkbox"/>	0x00400000	0x00500893	addi x17,x0,5	14: li a7, 5
<input type="checkbox"/>	0x00400004	0x00000073	ecall	15: ecall
<input type="checkbox"/>	0x00400008	0x00a004b3	add x9,x0,x10	16: mv s1, a0
<input type="checkbox"/>	0x0040000c	0x0fc10317	auipc x6,64528	21: la t1, array
<input type="checkbox"/>	0x00400010	0xffe30313	addi x6,x6,-8	
<input type="checkbox"/>	0x00400014	0x0092dc63	bge x5,x5,24	24: bge t0, s1, end for # check for loop, init.
<input type="checkbox"/>	0x00400018	0x00000073	ecall	27: ecall
<input type="checkbox"/>	0x0040001c	0x00a32023	mv x10,0(x6)	30: mv a0, 0(t1)
<input type="checkbox"/>	0x00400020	0x00128293	addi x5,x5,1	33: addi t0, t0, 1 # t0 ++
<input type="checkbox"/>	0x00400024	0x00430113	addi x6,x6,4	34: addi t1, t1, 4 # t1 = t1 + 4
<input type="checkbox"/>	0x00400028	0xfed4ff04f	jai x0,-20	37: j start for
<input type="checkbox"/>	0x0040002c	0x0fc10517	auipc x10,64528	41: la a0, array
<input type="checkbox"/>	0x00400030	0xfdd50513	addi x10,x10,-40	
<input type="checkbox"/>	0x00400034	0x00000293	addi x5,x0,0	42: li t0, 0 # index of lo
<input type="checkbox"/>	0x00400038	0xffff4813	addi x6,x5,-1	43: addi t1, s1, -1 # index of hi
<input type="checkbox"/>	0x0040003c	0x044000ef	jai x1,68	44: jai rs, qsort # quick sort
<input type="checkbox"/>	0x00400040	0x00000293	addi x5,x0,0	50: li t0, 0 # init, t0 == 0
<input type="checkbox"/>	0x00400044	0x0fc10317	auipc x6,64528	51: la t1, array # init t1 is th.
<input type="checkbox"/>	0x00400048	0xfcd30313	addi x6,x6,-64	
<input type="checkbox"/>	0x0040004c	0x0292dc63	bge x5,x5,44	55: bge t0, s1, end for out # check for loop
<input type="checkbox"/>	0x00400050	0x00032503	lw x10,0(x6)	58: lw a0, 0(t1)
<input type="checkbox"/>	0x00400054	0x00100831	addi x17,x0,1	59: li a7, 1 # printInt

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	32	-1	1	2	4	5	8	11
0x10010020	22	35	78	0	0	0	0	0
0x10010040	0	0	0	0	0	0	0	0
0x10010060	0	0	0	0	0	0	0	0
0x10010080	0	0	0	0	0	0	0	0
0x100100a0	0	0	0	0	0	0	0	0
0x100100c0	0	0	0	0	0	0	0	0
0x100100e0	0	0	0	0	0	0	0	0

Registers

Name	Number	Value
zero	0	0
ra	1	4194369
sp	2	2147479548
gp	3	268468224
tp	4	0
t0	5	10
t1	6	268501036
t2	7	0
t3	8	0
t4	9	10
a0	10	268500992
a1	11	0
a2	12	0
a3	13	0
a4	14	0
a5	15	0
a6	16	0
a7	17	10
s0	18	0
s1	19	0
s2	20	0
s3	21	0
s4	22	0
s5	23	0
s6	24	0
s7	25	0
s8	26	0
s9	27	0
s10	28	8
s11	29	10
t5	30	0
t6	31	0
pc		4194432

Messages

```

10
-1
22
8
35
5
4
11
2
1
78
-1 1 2 4 5 8 11 22 35 78
-- program is finished running (0) --

```

```

10
-1
22
8
35
5
4
11
2
1
78
-1 1 2 4 5 8 11 22 35 78
-- program is finished running (0) --

```

Reference:

TextBook -Computer Organization and Design_ The Hardware Software Interface
[RISC-V Edition]