

PROYECTO

González Ramos Jorge Humberto
Jiménez Vital Francisco
Velasco Ochoa Dana



30 DE MAYO DE 2020
SEMINARIO DE SOLUCION DE ARQUITECTURA DE COMPUTADORAS
Sección D14

MIPS

MIPS por las siglas de Microprocessor without Interlocked Pipeline Stages, se conoce a toda una familia de microprocesadores de arquitectura RISC desarrollados por MIPS Technologies.

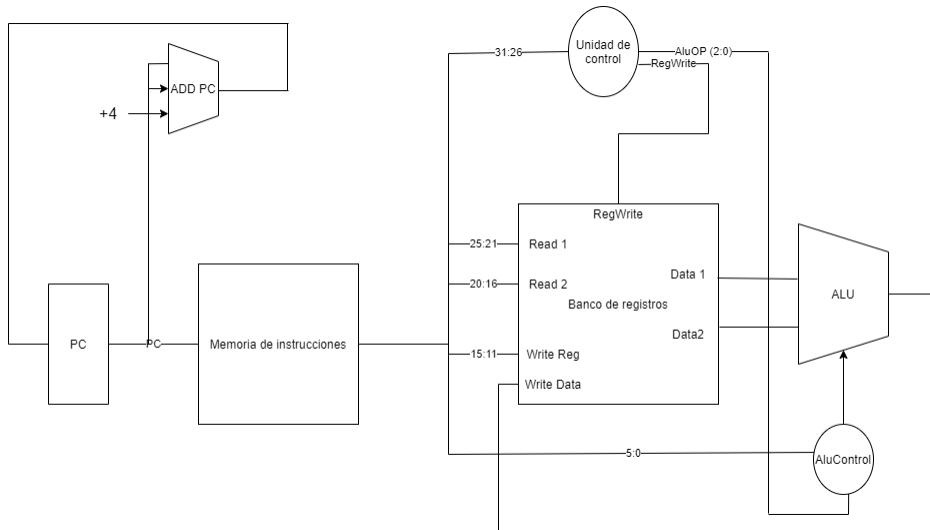
Los diseños del MIPS son utilizados en la línea de productos informáticos de SGI; en muchos sistemas embebidos; en dispositivos para Windows CE; routers Cisco; y videoconsolas como la Nintendo 64 o las Sony PlayStation, PlayStation 2 y PlayStation Portable. Más recientemente, la NASA usó uno de ellos en la sonda New Horizons.

En los últimos años gran parte de la tecnología empleada en las distintas generaciones MIPS ha sido ofrecida como diseños de "IP-cores" (bloques de construcción) para sistemas embebidos. Se ofertan los núcleos básicos de 32 y 64 bits, conocidos respectivamente como 4K y 5K respectivamente, y con licencias MIPS32 y MIPS64. Estos núcleos pueden ser combinados con unidades añadidas tales como FPUs, sistemas SIMD, dispositivos de E/S, etc.

Los núcleos MIPS han sido comercialmente exitosos, siendo empleados actualmente en muchas aplicaciones industriales y de consumo. Pueden encontrarse en los más modernos routers Cisco, TP-Link y Linksys, cablemódems y módems ADSL, tarjetas inteligentes, controladoras de impresoras láser, decodificadores de TV, robots, ordenadores de mano, Sony PlayStation 2 y Sony PlayStation Portable.

En móviles y PDA's, sin embargo, el núcleo MIPS fue incapaz de desbancar a su competidor de arquitectura ARM.

Explicación DataPath Fase 1



Para la fase 1 se juntaron solo los módulos PC, Add PC, Memoria de instrucciones, Unidad de control, Banco de registros, un controlador para la Alu y la misma Alu.

```

1  module pc (
2    input clk,
3    input [31:0] entrada,
4    output reg [31:0] salida
5  );
6
7  initial
8  begin
9    salida <= -4;
10 end
11
12 always@(posedge clk)
13 begin
14   salida <= salida+32'd4;
15 end
16
17 endmodule
21

```

En el módulo **PC** se juntó con el **Add PC** recibiendo una entrada de 32 bits y obteniendo una salida de 32bits haciendo una suma de +4Decimal en bits, reaccionando al juego del reloj mientras está en positivo.

En el módulo de memoria de instrucciones se hizo una lectura en el archivo instrucciones recibiendo un dato de 32 bits que se separan más adelante.

```

1  module instrucmemory(
2    input [31:0] direccion,
3    output reg [31:0]salida_datos
4  );
5
6  reg [7:0]mem[0:128];
7
8  initial begin
9    $readmem("instrucciones.txt", mem);
10 end
11
12 always @/*
13 begin
14   salida_datos <= {mem[direccion],mem[direccion+1],mem[direccion+2],mem[direccion+3]};
15 end
16
17 endmodule
20

```

Los módulos Unidad de control, banco de registros y Alu Control, reciben sus entradas de esta salida utilizando la separación de bits.

```

1 module UnidadControl (
2     input [5:0]entrada,
3     output [0:0]RegDst,
4     output [0:0]AluSrc,
5     output [2:0] AluOP,
6 );

```

En el módulo **Unidad de control** se reciben los últimos 6bits que tiene la salida de memoria de instrucciones la cual sirve para decidir en esta fase si serán operaciones de tipo R, otorgando la decisión al controlador de la Alu mediante 3bits y la posición donde se guardará el registro al banco de registros.

El módulo **controlAlu** recibe los primeros 6bits de la salida de memoria de instrucciones que define la operación y 3 bits que se dan del controlador para decir el tipo de operación tipo R en esta fase y ofreciendo una salida de 4 bits con la operación a realizar ya filtrada por tipo.

```

1 module controlAlu (
2     input[5:0]entrada,
3     input[2:0]op,
4     output reg[3:0]salida
5 );
6
7     always@(entrada, op)
8     begin
9         if(op==3'b010) //2 func based
10            begin
11                if(entrada[3:0] == 0) begin //sum
12                    salida <= 4'b0010;
13                end
14                else if(entrada[3:0] == 4'b0010) begin //resta
15                    salida <= 4'b0110;
16                end
17                else if(entrada[3:0] == 4'b0100) begin //and
18                    salida <= 4'b0000;
19                end
20                else if(entrada[3:0] == 4'b0101) begin //or
21                    salida <= 4'b0001;
22                end
23                else if(entrada[3:0] == 4'b1100) begin //lessThan
24                    salida <= 4'b0111;
25                end
26            end
27        else if(entrada[3:0] == 4'b0011) begin //mult
28            salida <= 4'b0000;
29        end
30    end
31 endmodule

```

```

1 module BancoRegistros (
2     input [4:0]ReadReg1,
3     input [4:0]ReadReg2,
4     input [4:0]WriteReg,
5     input [31:0]writeData,
6     input Regwrite,
7
8     output reg [31:0]ReadData1,
9     output reg [31:0]ReadData2
10 );
11
12     reg [31:0]AlmacenReg[0:31];
13
14     initial
15     begin
16         Sreadmemb("bancoreg.txt", AlmacenReg, 0, 31);
17         #10;
18     end
19
20     always @*
21     begin
22         AlmacenReg[writeData] <= writeData;
23
24         ReadData1 <= AlmacenReg[ReadReg1];
25         ReadData2 <= AlmacenReg[ReadReg2];
26
27     end
28
29 endmodule
30

```

Mientras que a los anteriores módulos se les conectaron los primeros y últimos 6bits al **banco de registros** le toca recibir todos los demás bits separados de 5 en 5 los cuales contienen una dirección para guardar un dato nuevo y las direcciones de los datos que se lean en el archivo y son mandados a ser operados por la Alu.

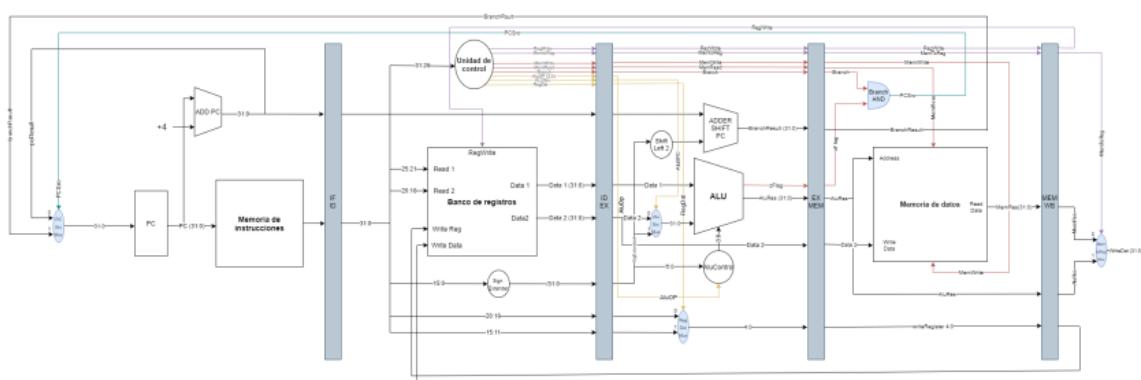
En el módulo **Alu** se reciben las 3 entradas antes mencionadas 2 datos que interactuarán con un operador y un selector que dice que operación se tendrá que hacer dando el resultado al banco de registros para ser guardado en una posición que ya se le dio anteriormente y ofreciendo una salida más para saber si el resultado queda en 0's.

```

1 module alu (
2     input [31:0]Data1,
3     input [31:0]Data2,
4     input [3:0]selector,
5     output Reg [31:0]salida,
6     output reg zflag
7 );
8
9     reg [31:0]tmp;
10
11     always @*
12     begin
13         case (selector)
14             4'b0000: salida <= Data1 & Data2;
15             4'b0001: salida <= Data1 | Data2;
16             4'b0110: salida <= Data1 + Data2;
17             4'b0101: salida <= Data1 - Data2;
18             4'b0111: salida <= Data1 < Data2 ? 1:0;
19             4'b1100: salida <= Data1 ^ Data2; // XOR
20             4'b1100: salida <= ~Data1 | Data2; // NOR
21             4'b1000: salida <= Data1 * Data2; // MUL
22             4'b1010: salida <= Data1 / Data2; // DIV
23             default: salida <= Data2;
24         endcase
25     end
26
27 endmodule
28

```

DataPath Fase 2



En esta fase se aumenta la capacidad de nuestro procesador, con la unidad de control agregamos las instrucciones: **beq, sl, sw, addi, ori, andi, slti**.

Se agregaron 4 buffers, 1 Memoria de datos, el módulo sing extender, shift left, un adder shift PC, una compuerta and llamada Branch y 4 multiplexores y fueron editadas algunas entradas y salidas de algunos módulos para la nueva fase.

Se usa el **primer multiplexor** para el módulo de pc, en el cual de entradas le llega el resultado de PC ADD y WriteData que va de la memoria de datos, el regDST decide qué salida tiene el multiplexor

```

1 module mux1(
2     input [31:0]ResBranch1,
3     input [31:0]ResPC0,
4     input Pcsrc,
5     output [31:0]BResult
6 );
7
8 assign
9     BResult = (Pcsrc)? ResBranch1 : ResPC0;
10
11 endmodule

```

```

1 module buffer1(
2     input clk;
3     input [31:0] entrada,
4     input [31:0] entr_pc;
5     output reg [31:0]salida,
6     output reg [31:0]sal_pc
7 );
8
9
10 always@(posedge clk)
11 begin
12     salida <= entrada;
13     sal_pc <= entr_pc;
14 end
15
16 endmodule

```

Modificando las salidas de los módulos existentes, el **primer buffer** recibe 2 entradas, una de pc y otra de la memoria de instrucciones, se recorren las salidas de la memoria de instrucciones al buffer y se hace la separación de bits en las instancias no en el módulo mismo. Las entradas del banco de registros quedan igual.

El módulo **Sing Extender** le llega una entrada de 16 bits del primer buffer hace una función para convertir la salida en 32 bits para ser otorgado al segundo buffer.

```

1 module sing_ext(
2     Input [15:0]entrada,
3     output [31:0] salida
4 );
5
6 assign salida = (entrada[15] == 1'b1 ? {16'b1111111111111111, entrada}
7
8
9 endmodule

```

```

1  module UnidadControl (
2    input[3:0]Entrada,
3    output reg Regdst,
4    output reg ALUsrc,
5    output reg [1:0]aluOp,
6    output reg RegMench,
7    output reg Memwrite,
8    output reg Memread,
9    output reg Jump,
10   output reg Memtoeg,
11   output reg Regwrite,
12 );
13 end
14
15 always@*
16 begin
17
18   if (Entrada == 6'b101011) // SW
19     begin
20       Regdst <= 0; // MPX (0 <= 20-16, 1=15-11) <
21       Jump <= 0;
22       ALUsrc <= 0; // usa reg 0 o extended/imm 1
23       Memtoeg <= 1;
24       Regwrite <= 0; // guardar en reg
25       MemRead <= 0; // leer mem
26       MemWrite <= 1; // escribir en mem
27       Branch <= 0; // no branch
28       ALuOp <= 0; // ALU function
29       Regost <= 0; // guarda en reg
30     end
31   else if (Entrada == 6'b000100) // BEQ
32   begin
33
34     Regdst <= 1'bX;
35     Jump <= 0;
36     ALUsrc <= 1'b0;
37     Memtoeg <= 1'b0;
38     Memread <= 0;
39     Memwrite <= 1'b0;
40     Branch <= 1'b1;
41     ALuOp <= 0;
42     $display("BEQ");
43
44   end
45   else if (Entrada == 6'b001010) // SLTI
46   begin
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
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
788
789
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
817
818
819
819
820
821
822
823
824
825
826
827
827
828
829
829
830
831
832
833
833
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1390
1391
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1400
1401
1401
1402
1402
1403
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1410
1411
1411
1412
1412
1413
1413
1414
1414
1415
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1420
1421
1421
1422
1422
1423
1423
1424
1424
1425
1425
1426
1426
1427
1427
1428
1428
1429
1429
1430
1430
1431
1431
1432
1432
1433
1433
1434
1434
1435
1435
1436
1436
1437
1437
1438
1438
1439
1439
1440
1440
1441
1441
1442
1442
1443
1443
1444
1444
1445
1445
1446
1446
1447
1447
1448
1448
1449
1449
1450
1450
1451
1451
1452
1452
1453
1453
1454
1454
1455
1455
1456
1456
1457
1457
1458
1458
1459
1459
1460
1460
1461
1461
1462
1462
1463
1463
1464
1464
1465
1465
1466
1466
1467
1467
1468
1468
1469
1469
1470
1470
1471
1471
1472
1472
1473
1473
1474
1474
1475
1475
1476
1476
1477
1477
1478
1478
1479
1479
1480
1480
1481
1481
1482
1482
1483
1483
1484
1484
1485
1485
1486
1486
1487
1487
1488
1488
1489
1489
1490
1490
1491
1491
1492
1492
1493
1493
1494
1494
1495
1495
1496
1496
1497
1497
1498
1498
1499
1499
1500
1500
1501
1501
1502
1502
1503
1503
1504
1504
1505
1505
1506
1506
1507
1507
1508
1508
1509
1509
1510
1510
1511
1511
1512
1512
1513
1513
1514
1514
1515
1515
1516
1516
1517
1517
1518
1518
1519
1519
1520
1520
1521
1521
1522
1522
1523
1523
1524
1524
1525
1525
1526
1526
1527
1527
1528
1528
1529
1529
1530
1530
1531
1531
1532
1532
1533
1533
1534
1534
1535
1535
1536
1536
1537
1537
1538
1538
1539
1539
1540
1540
1541
1541
1542
1542
1543
1543
1544
1544
1545
1545
1546
1546
1547
1547
1548
1548
1549
1549
1550
1550
1551
1551
1552
1552
1553
1553
1554
1554
1555
1555
1556
1556
1557
1557
1558
1558
1559
1559
1560
1560
1561
1561
1562
1562
1563
1563
1564
1564
1565
1565
1566
1566
1567
1567
1568
1568
1569
1569
1570
1570
1571
1571
1572
1572
1573
1573
1574
1574
1575
1575
1576
1576
1577
1577
1578
1578
1579
1579
1580
1580
1581
1581
1582
1582
1583
1583
1584
1584
1585
1585
1586
1586
1587
1587
1588
1588
1589
1589
1590
1590
1591
1591
1592
1592
1593
1593
1594
1594
1595
1595
1596
1596
1597
1597
1598
1598
1599
1599
1600
1600
1601
1601
1602
1602
1603
1603
1604
1604
1605
1605
1606
1606
1607
1607
1608
1608
1609
1609
1610
1610
1611
1611
1612
1612
1613
1613
1614
1614
1615
1615
1616
1616
1617
1617
1618
1618
1619
1619
1620
1620
1621
1621
1622
1622
1623
1623
1624
1624
1625
1625
1626
1626
1627
1627
1628
1628
1629
1629
1630
1630
1631
1631
1632
1632
1633
1633
1634
1634
1635
1635
1636
1636
1637
1637
1638
1638
1639
```

```

31   always @(posedge clk)
32     begin
33       sal_Regwrite <= Regwrite;
34       sal_MemToReg <= MemToReg;
35       sal_Jump <= Jump;
36       sal_MemWrite <= MemWrite;
37       sal_MemRead <= MemRead;
38       sal_Branch <= Branch;
39       sal_OutBranch <= OutBranch;
40       sal_Zflag <= zFlag;
41       sal_ALUres <= ALUres;
42       sal_Data2 <= Data2;
43       sal_WriteReg <= WriteReg;
44     end
45   endmodule
46 
```

En el **tercer buffer** se tienen 12 entradas y 12 salidas, mantiene y pasa los datos de las salidas anteriores para dirigirlos a la memoria de datos, a una compuerta AND llamada Branch y al cuarto buffer.

El módulo **Branch** funge como AND analizando el Zflag y una salida de la unidad de control de 1 bit que define lo que será el selector del ADD PC.

```

1 module branch(
2   input Branch,
3   input zFlag,
4   output salida
5 );
6 assign
7   salida = zFlag & Branch;
8 endmodule
9 
```

```

1 module Memoriadato(
2   input clk,
3   input [31:0]Address,
4   input [31:0]writeData,
5   input MemWrite,
6   input MemRead,
7   output reg[31:0]MemRes
8 );
9   reg [31:0]mem[31:0];
10
11   integer i;
12   initial begin
13     for(i = 0; i < 32; i = i + 1) begin
14       mem[i] = 1;
15     end
16   end
17   always@(posedge clk) begin
18     if(Memwrite == 1) mem[Address] <= writeData;
19     if(MemRead == 1) MemRes <= mem[Address];
20   end
21 endmodule
22 
```

La **memoria de datos** es una expansión del banco de registros por falta de espacio.

El **cuarto buffer** recibe los registros de la memoria de datos y el resultado de la ALU y la salida del multiplexor 3, este buffer sirve para guardar temporalmente los datos y pasarlos a sus respectivos lugares de entrada.

```

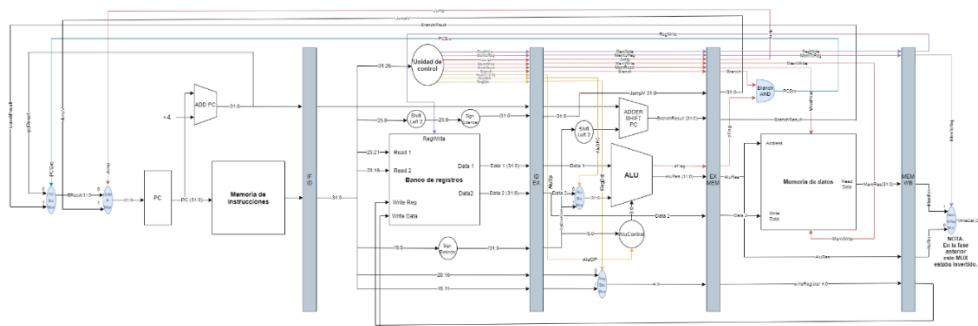
1 module buffer4 (
2   input clk,
3   input Regwrite,
4   input MemToReg,
5   input [31:0] MemRes,
6   input [31:0] ALUres,
7   input [4:0] writeRegister,
8
9   output reg sal_Regwrite,
10  output reg sal_MemToReg,
11  output reg[31:0] sal_MemRes,
12  output reg[31:0] sal_ALUres,
13  output reg[4:0] sal_WriteReg
14 );
15
16   always @(posedge clk)
17   begin
18     sal_Regwrite <= Regwrite;
19     sal_MemToReg <= MemToReg;
20     sal_MemRes <= MemRes;
21     sal_ALUres <= ALUres;
22     sal_WriteReg <= WriteRegister;
23   end
24 endmodule
25 
```

```

1 module Mux4(
2   input [31:0]ALUres0,
3   input [31:0]MemRes1,
4   input MemToReg,
5   output [31:0]writeDat
6 );
7
8   assign
9     writeDat = (MemToReg)? MemRes1: ALUres0;
10
11 endmodule
12 
```

A este **cuarto multiplexor** se le asignan las salidas de la memoria de datos y el resultado de la ALU, con una entrada de la unidad de control decide qué salida tendrá el multiplexor.

DataPath Fase 3



En esta ultima etapa, se agregaron las intrucciones de salto, (tipo J) y ya con esta ultima modificación se completa lo que seria un procesador basico y funcional. Se incluyeron 3 módulos los cuales son shift_left_J, Mux_J, Sing_ext_J, y se le añadió al alu control instrucciones de tipo J

```

1  module shift_left_J(
2    input [25:0] entrada,
3    output [25:0] salida
4  );
5    assign salida = {entrada<<2};
6  endmodule

```

En el **shift_left_J** La entrada se multiplica por 4 moviendo 2 bits a la izquierda, dato que al pasar por los buffers que adelante se dirigirá a PC ADD

El multiplexor de instrucciones J **Mux_J** se dedica a asignarle la salida de shift_left_J y el resultado del PC ADD, con una entrada de la unidad de control decide qué salida tendrá el multiplexor.

```

1  module Mux_J(
2    input [31:0] JumpV1,
3    input [31:0] BResult0,
4    input Jump,
5    output [31:0] PC
6  );
7    assign
8      PC = (Jump)? JumpV1 : BResult0;
9  endmodule

```

```

1  module sing_ext_J(
2    input [25:0] entrada,
3    output [31:0] salida
4  );
5    assign
6      salida = (6'b0+entrada);
7  endmodule

```

El módulo **sing_ext_J** hace que el resultado dado por shift_left_J tenga el mismo valor mientras que se vuelve un dato de 32 bits utilizados en PC ADD

El modulo **controlAlu** se vio modificado agregándole el condicional que revisa si es una instrucción de tipo J junto con sus selectores de operación que se darán a la Alu

```

34   else if(entrada[3:0] == 4'b1010) begin //division
35     salida = 4'b1010;
36   end
37
38   else if(entrada[3:0] == 4'b0110) begin //xor
39     salida = 4'b0101;
40   end
41
42   else if(entrada[3:0] == 4'b0111) begin //nor
43     salida = 4'b1100;
44   end
45
46   else if(entrada[3:0] == 3'b000) begin
47     salida = 4'b0010;
48     //$display("ALUC0_Suma");
49   end
50
51   else if (op == 3'b001) begin
52     salida = 4'b0110;
53     //$display("ALUC0_Resta");
54   end
55   else if (op == 3'b011) begin
56     salida = 4'b0000;
57     //$display("ALUC0_AND");
58   end
59   else if (op == 3'b100) begin
60     salida = 4'b0001;
61     //$display("ALUC0_OR");
62   end
63   else if (op == 3'b101) begin
64     salida = 4'b0111;
65     //$display("ALUC0_SLT");
66   end
67
68

```

Programa de ensamblador Selective Sort

```

main:
    li $t5, 85
    li $t4, 22
    li $t3, 94
    li $t2, 55
    li $t0, 105
    li $t6, 58
    li $t7, 43
    li $t8, 39
    li $t1, 69

    sw $t5, 0($zero)
    sw $t4, 1($zero)
    sw $t3, 2($zero)
    sw $t2, 3($zero)
    sw $t1, 4($zero)
    sw $t6, 5($zero)
    sw $t7, 6($zero)
    sw $t8, 7($zero)
    sw $t0, 8($zero)
    li $t9, 8 # total de nums

.sort:
    add $t0, $zero, $zero # i = 0

    .loop1:
        addi $t1, $t9, -1 # n - 1
        add $t2, $t0, $zero # index
= i
        addi $t3, $t0, 1 # j = i+1

        .loop2:
            nop
            lw $t4, 0($t3) # arr[j]
            lw $t5, 0($t2) #
arr[index]

            # if arr[j] < arr[index]
            blt $t4, $t5 .change
            j .addj

.change:
    add $t2, $zero, $t3
# index = j

.addj:
    addi $t3, $t3, 1 # j
= j +1
    blt $t3, $t9, .loop2
# mientras j < n

.swap:
    lw $t4, 0($t2) #
arr[index]
    lw $t5, 0($t0) # arr[i]

    add $t1, $zero, $t4 #
temp = arr[index]

    sw $t5, 0($t2) #
arr[index] = arr[i]
    sw $t1, 0($t0) # arr[i]
= temp

.add_i:
    addi $t0, $t0 1 # i = i
+ 1
    blt $t0, $t1 .loop1 #
mientras i < n - 1

.end:
    lw $t1, 0($zero)
    lw $t1, 1($zero)
    lw $t1, 2($zero)
    lw $t1, 3($zero)
    lw $t1, 4($zero)
    lw $t1, 5($zero)
    lw $t1, 6($zero)
    lw $t1, 7($zero)
    lw $t1, 8($zero)

```

Este programa tiene como objetivo hacer ordenamientos de números enteros utilizando el método selective sort, fue basado en el código de la siguiente página

<https://www.geeksforgeeks.org/selection-sort/>

Programa de ensamblador Tipos de Triángulos

```
.main:  
    li $t0 2 # lado 1  
    li $t1 35 # lado 2  
    li $t2 30 # lado 3  
    sub $t4 $t0 $t1  
    sub $t5 $t0 $t2  
    beq $t4 $zero .f1  
  
.iso:  
  
    sub $t6 $t1 $t2  
  
    beq $t4 $zero .isoceles # a ==  
b  
    beq $t5 $zero .isoceles # a ==  
c  
    beq $t6 $zero .isoceles # b ==  
c  
  
    # si no entonces es escaleno  
    j .escaleno  
  
.f1:  
    beq $t5 $zero .equilatero  
  
    j .iso  
  
.equilatero:  
    li $t3 1  
    j .end  
  
.isoceles:  
    li $t3 2  
    j .end  
  
.escaleno:  
    li $t3 3  
  
.end:  
    sw $t3 0($zero) # guardar tipo  
de triangulo
```

El objetivo de este programa es determinar que tipo de triángulo es, para eso utilizamos 3 registros donde guardamos la medida de sus lados y después hacemos sus cálculos.

Un == es el equivalente a 2 registros restados dando 0, con esto ya podemos realizar la operación igual usando un branch con el registro \$zero

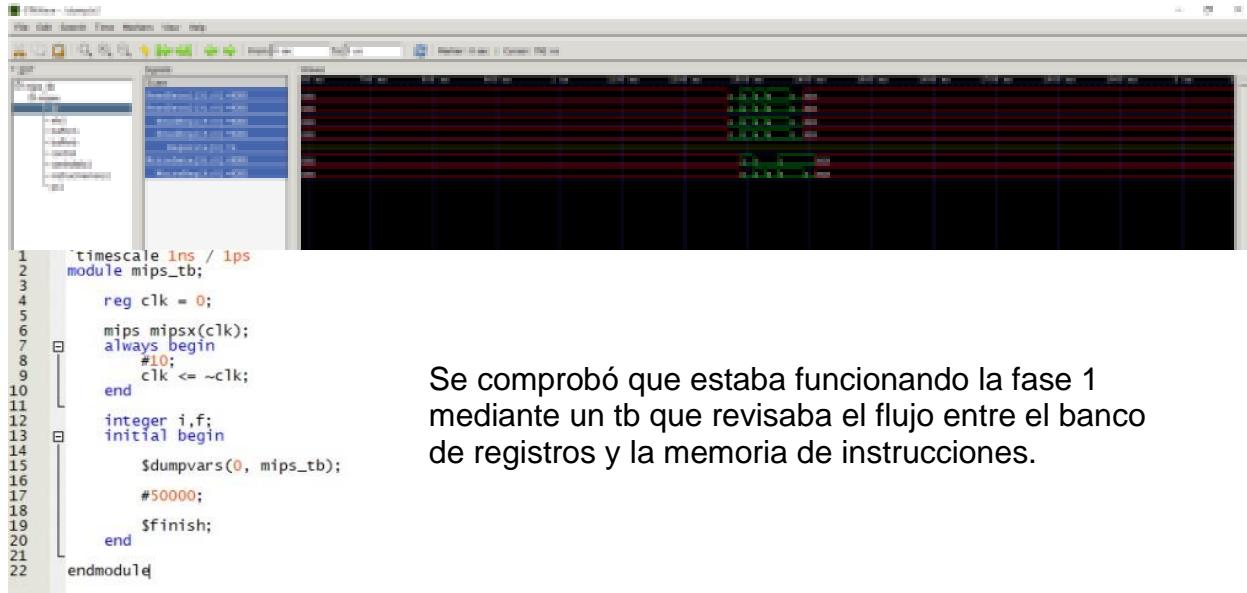
El equivalente de este código en python sería el siguiente:

```
a = 2  
b = 35  
c = 30  
if a == b and a == c:  
    resultado = 1
```

```
elif a==b or a==c or b == c:  
    resultado = 2  
else:  
    resultado = 3
```

TestBenches

TestBench Fase 1



Se comprobó que estaba funcionando la fase 1 mediante un tb que revisaba el flujo entre el banco de registros y la memoria de instrucciones.

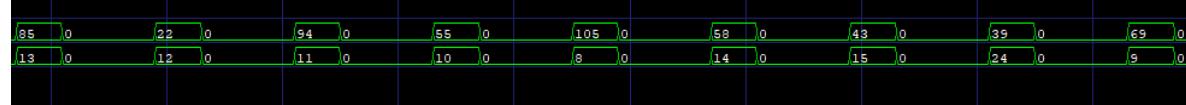
TestBench Fase3



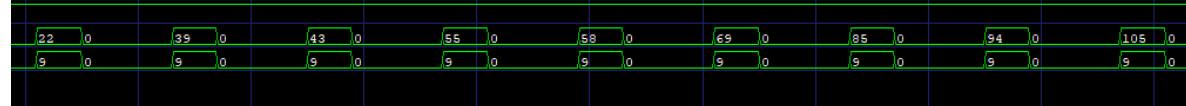
En esta tercera fase se comprobó que estuviera funcionando el salto

Testbench. Ensamblador Selective Sort

Valores ingresados: 85, 22, 94, 55, 105, 58, 43, 39, 69

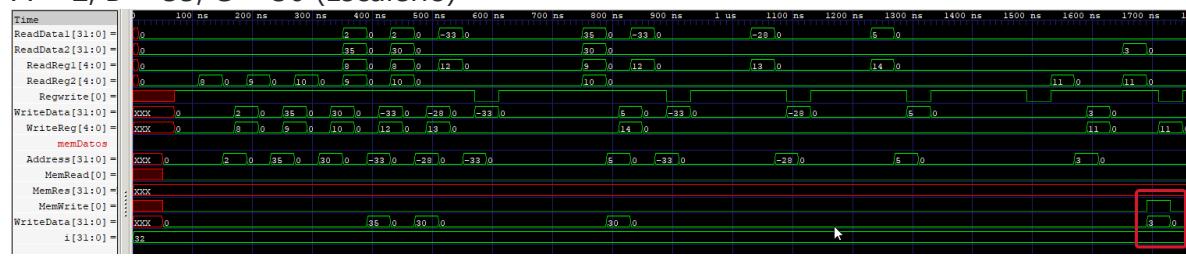


Resultados -> 22, 39, 43, 55, 58, 69, 85, 94, 105

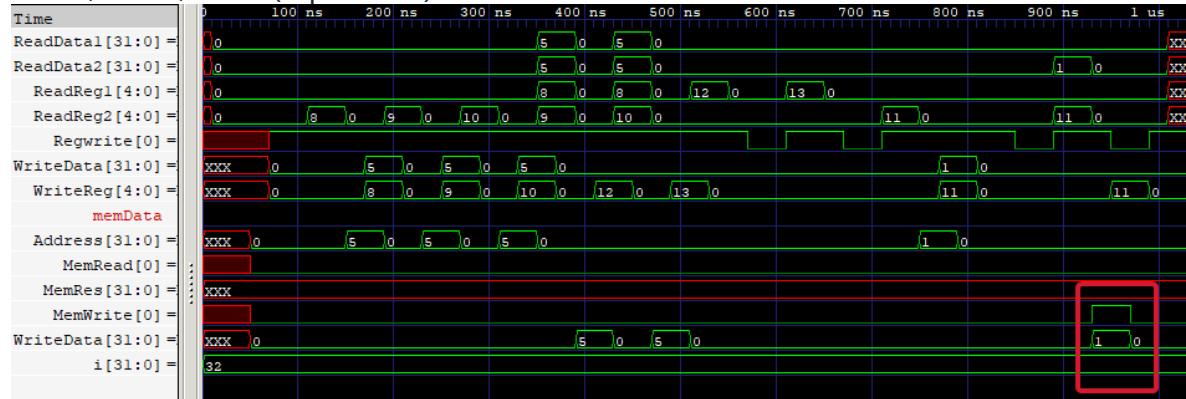


Testbench. Ensamblador Tipos de Triángulos

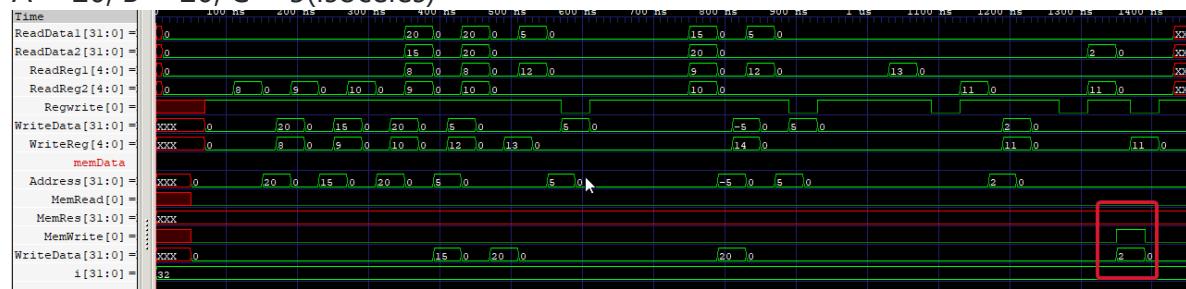
A = 2, B = 35, C = 30 (Escaleno)



A = 5, B = 5, C = 5(Equilatero)



A = 20, B = 20, C = 5(Isoceles)



DECODIFICADOR

El objetivo del decodificador es convertir las instrucciones de ensamblador al código de instrucciones el cual puede ser leído por el MIPS.

Las características que más definen al decodificador son:

- Soporte de labels o etiquetas
- Generar automáticamente los NOPs requeridos por las instrucciones para tener una ejecución correcta
- Detección de errores de escritura (parámetros, acceder a labels incorrectos, etc)
- Pseudo instrucciones para hacer más simple la escritura del código en ensamblador

Nuestro decodificador fue escrito en python, en 2 archivos, uno que es el principal y otro en el cual se declaran listas y diccionarios.

El archivo secundario es Diccionarios.py, en este archivo se almacenan los diccionarios y listas con las instrucciones y registros válidos, también una función para checar el contenido de estos registros.

```
registros = {
    "$zero": [0, 0],
    "$one": [1, 0],
    "$t0": [8, 1],
    "$t1": [9, 1],
    "$t2": [10, 1],
    "$t3": [11, 1],
    "$t4": [12, 1],
    "$t5": [13, 1],
    .
    .
    .
}
```

Archivo principal (Decoder.py) es el importante, es el que realiza todo el proceso de conversión a instrucciones del MIPS.

Está dividido en 6 funciones

- Cargado del archivo
- Eliminación de líneas sin código
- Mover los parámetros de ensamblador a los parámetros de instrucción y añadir NOPs
- Cálculo de posiciones de branch/jump
- Convertir los números con \$ a enteros y los registros también
- Convertir a binario y generar archivo de instrucciones

Cargado del archivo

```
• def selectfile():
•     global cfile
•     fName = str(input("Nombre del archivo: "))
•     print("")
•
•     try:
•         cfile = open(fName, 'r')
•     except FileNotFoundError:
•         print("> [ERROR] No se encontró el archivo indicado, ¿está en la
misma carpeta?")
•         exit()
```

Esta función simplemente pide el archivo de manera relativa el cual debe de cargar, no tiene mucha ciencia.

Eliminación de líneas sin código

```
• def torealcode():
•     global rlines
•
•     lines = cfile.readlines()
•     cline = 1
•
•     for line in lines:
•         strippedline = list(line.strip())
•
•         if strippedline:
•             while "," in strippedline: # Remover las comas de la linea
•                 strippedline.remove ","
•             strippedline = "".join(strippedline)
•             rlines.append([strippedline, cline])
•
•         cline += 1
```

En esta función se leen todas las líneas del código, se evitan las líneas que no tienen contenido y después se eliminan las , que existan para tener una lectura más simple, ya que se eliminan se une y se añade a una lista el cual contiene el código ya retocado y la linea en la que está en el archivo.

Convertir parámetros

Esta función es una de las más complicadas y tiene bastante contenido. El código tiene comentarios le cual explica prácticamente que hace línea por línea así que se explicará lo que no podría haber quedado aquí.

addinstruction

En esta función hay una parte donde se utiliza frecuentemente la siguiente función.

```
• def addinstruction(linex, params, nops=1):
•     scriptlines_1.append([linex, params])
•     for __ in range(nops):
•         scriptlines_1.append([linex, ["nop"]])
```

Esta función se encarga de agregar a una lista la instrucción con los parámetros ya ordenados y añade la cantidad de NOPs que necesita esta función para funcionar correctamente, también guarda la posición de la linea de código original para informar si hay un error.

LW y SW

También se contempla un caso especial en la escritura del ensamblador para las instrucciones sw y lw, las cuales se escriben de la siguiente manera `sw $reg decimal($reg)`. Para no complicar el asunto usando splits se decidió utilizar una expresión regular para ver el contenido.

```
• if rpam[0] == "sw":
•     reg = re.match(r"(\d+?)\((\$([^\$]+))\)", rpam[2])
•     if reg:
•
•         arg2 = str(reg.group(1))
•         arg1 = str(reg.group(2))
•         addinstruction(linex[1], ["sw", "$" + arg1, rpam[1], arg2])
•     else:
•
•         print("> [ERROR] No cumple con los parámetros esperados en la
linea %i (esperados: %s)" % (linex[1], dic.funcs[rpam[0]]))
•     exit()
```

Por ejemplo el SW. Primero hacemos un análisis en regex con `(\d+?)\((\$([^\$]+))\)` la cual busca que cumpla con `decimal(signodedinero+palabra)` y almacena los datos en grupos, si no se encuentra de esa manera entonces hay un error de escritura y se termina la decodificación.

Cálculo de posiciones

```
• def calculatepositions():
•     global scriptlines_2
•
•     PC = -1
•
•     for line in scriptlines_1:
•         PC += 1
```

```

•         fpam = line[1][0]
•
•         if(fpam[-1] == ":"):
•             scriptlines_2.append([line[0], ["nop"]])
•             jpositions.append([PC, fpam])
•
•         else:
•             if fpam == "beq":
•                 scriptlines_2.append(line)
•                 scriptlines_2.append([line[0], ["nop"]])
•                 jpositions.append([PC, fpam, line[1][3], 0])
•             else:
•                 scriptlines_2.append(line)

```

Esta función se encarga de calcular las posiciones de los labels y almacenarlas en una lista para que en la función siguiente convierta el texto al entero que corresponde a esa linea.

En realidad los labels son interpretados como NOPs en el MIPS así de esta manera evitamos más problemas.

Conversión de palabras a enteros.

En esta sección se convierten todo lo que sea una palabra a un entero, por ejemplo los registros, los labels y algunas constantes que empiezan con \$

Para las constantes se utiliza una expresión regular que es la siguiente (^-?\d+(.\d+)?\\$) lo cual busca números positivos o negativos que inician con un \$.

Para la conversión de los labels a números está esta sección del código

```

•     if dic.checkKey(pam, dic.registros):
•         templist.append(dic.registros[pam][0])
•     else:
•         if pam[0] == ".": # Ajustar posiciones de salto/branchoe
•             for zon in jpositions:
•                 if pam == zon[1][: -1]:
•                     if line[1][0] == "beq":
•                         for zonx in jpositions:
•                             if zonx[1] == "beq":
•                                 if len(zonx) > 3:
•                                     if zonx[3] == 0:
•                                         zonx[3] = 1
•                                         jpositions[jpositions.index(zonx)] =
•                                         zonx
•
•                                     templist.append(zon[0] - zonx[0])
•                                     break

```

```

•           else:
•               templist.append(zon[0])
•               break
•       else:
•           print("> [ERROR] Ingresaste un parámetro inexistente en la linea
• %i (>%s) (esperados: %s)" % (line[0], pam, dic.funcs[line[1][0]]))
•           exit()

```

Se revisa que el parámetro inicie con un . (esta forma la elegimos para que sea mucho más sencillo identificar los labels) y si no entonces ya hay algo mal escrito y se informa al usuario. Checa si es una instrucción de tipo branch, si lo es entonces hace el cálculo de las posiciones que hay de diferencia templist.append(zon[0] - zonx[0]) , si no es un branch entonces es un salto así que añade directamente el número.

Conversión a binario y guardado

Finalmente está la conversión a binario y su guardado, esta función lee todas las líneas que ya fueron procesadas y las convierte a binario. Dependiendo del tipo de instrucciones como será procesada ya que tenemos que considerar el tamaño de bits por si se utilizan números negativos.

En esta parte también se puede detectar otro error, si existe una label inexistente ya que jamás fue convertida a un número entero por no estar declarada.

Una vez terminado el proceso se convierte a binario utilizando la siguiente función que soporta tanto positivos como negativos.

```

•   def dbin(number, clen, c2=0):
•       if number < 0:
•           return bin(number % (1 << clen))[2::]
•       else:
•           return "0" * (clen - len(bin(number % (1 << clen))[2::])) +
•           bin(number % (1 << clen))[2::]

```

Esta función detecta si es positivo o negativo ya que dependiendo de esto se añaden los bits sobrantes como 1 o un 0.

Y una vez convertido todo es guardado en un archivo llamado instr.mem

Objetivo

Se busca simular y demostrar en el proceso la importancia que tiene un procesador en una computadora, aprender sobre su arquitectura y desenmarañar las instrucciones y procesos que se pueden realizar mediante esta tecnología, desarrollando un procesador básico de 32 bits y ejemplificando su funcionamiento mediante instrucciones de tipo I, R, y J.

Conclusión

El procesador es la parte más importante de un ordenador. Es el cerebro, coordina todos los datos, todas las aplicaciones, y todos los procesos que pasan por el procesador. Los dos tipos de procesador más utilizados son Intel y AMD, pero no son los únicos. Se logró aprender a lo largo del semestre la arquitectura para después implementar nuestro propio procesador dummy, logrando trabajar en equipo y apoyándonos mutuamente entre todos, teniendo como base los aprendizajes y el apoyo del profesor.

Bibliografía

MIPS.pdf	Instruction Set manual	The MIPS32
MIPS Architecture for Programmers	Volume II-A	
http://www.nec.co.jp/press/en/9801/2002.html		
Computer Organization and Design	5 th Edition	David A. Patterson/Jhon L. Hennessy
https://web.archive.org/web/20150719075343/http://www.sumagamer.com/noticias/new-horizons-mismo-procesador-playstation/		