

Dokumentation PIC16F84-Simulator

Einleitung

Das vorliegende Dokument beschreibt die Umsetzung des Projektes der Erstellung einer Simulationssoftware des Microcontrollers des Typs PIC16F84 der Firma Microchip. Das Projekt im Rahmen der Vorlesung Systemnahe Programmierung 2 erfordert das Anwenden bereits erlernter Kenntnisse aus den vorigen Semestern wie beispielsweise binäre Schaltoperationen aus der Digitaltechnik (Bool'sche Algebra, binäre Rechenverfahren, Grundgatter, etc.). Zusätzlich sind Vorkenntnisse der Rechnerarchitekturen notwendig, um die Funktionsweise eines Microcontrollers zu verstehen und im Simulator umsetzen zu können.

Das Projekt wird in einem Zwei-Personen-Team umgesetzt, zur Implementation ist Java mit der Entwicklungsumgebung Eclipse die Programmiersprache der Wahl; Hauptgrund für diese Entscheidung waren die vielen vorgefertigten und einfach zu importierenden Tools und Bibliotheken zur Variablen- und GUI-Bearbeitung. Für die Konstruktion des GUI wird das Eclipse-Plugin Windowbuilder verwendet, das mit einem Baukastensystem die GUI-Erstellung erheblich vereinfacht.

Was ist eine Simulation?

Der Verein Deutscher Ingenieure definiert eine Simulation folgendermaßen:

„[Eine Simulation ist die] Nachbildung eines Systems mit seinen dynamischen Prozessen in einem experimentierfähigen Modell, um zu Erkenntnissen zu gelangen, die auf die Wirklichkeit übertragbar sind“
[VDI 3633, (2013)]

Übersetzt bedeutet das in etwa „In einer Simulation wird die reale Welt so nachgestellt, dass man durch Experimente Wissen über Prozesse der realen Welt erlangen kann.“ Sie werden meistens dann eingesetzt, wenn Experimente und Messungen in der Realität zu langwierig (z.B. Klimamodelle), zu schnell (Explosionen), zu gefährlich (u.A. Crashtests), unmöglich (Urknallforschung) oder einfach zu teuer (z.B. Stadtplanung) wären.

Pro/Kontra Simulation

Das wichtigste Argument für eine Simulation wurden bereits angesprochen: sie sind ein sehr gutes Mittel, wenn Messungen in der echten Welt aus verschiedenen Gründen nicht möglich sind. Dafür allerdings unterliegen sie der Problematik, dass sie die Realität nicht exakt abbilden können und daher deutlich anfälliger für Fehler und Ungenauigkeiten sind. Zusätzlich bergen Simulationen die Gefahr, dass anfängliche Fehler wie falsch gewählte Parameter sich im Laufe der Simulation fortsetzen und verstärken, sodass am Ende ein vollkommen verfälschtes Ergebnis steht.

Unsere Implementation

Da bei der Umsetzung des Projektes die Funktionsweise des originalen Controllers möglichst exakt nachgebildet werden soll, ist dieses Projekt keine Emulation, sondern eine vollwertige Simulation, äquivalent zu Flug- oder Fahrschulsimulatoren. Bei einer Emulation steht lediglich das Ergebnis im Vordergrund, die Funktionsweise muss nicht unbedingt mit dem emulierten System übereinstimmen, wie zum Beispiel bei der Emulation von Smartphone-Betriebssystemen auf dem PC oder zu Emulation nicht mehr verfügbarer Spielekonsolen für Retro-Videospiele.

File

Input

0

+0

0x

+0

+1

+2

+3

+4

+5

+6

+7

0

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

14

Das Interface besteht aus mehreren Teilen zur Darstellung der einzelnen Simulator-Funktionsbereiche. Direkt unterhalb der Menüleiste am oberen Rand des Fensters, mit der sich die Simulation beenden oder eine LST-Datei einlesen lässt, befindet sich die Übersicht über die General Purpose Register (GPR) mit Bearbeitungsfunktion für einzelne Register. Das Textfeld dient zur Eingabe des Wertes, der in das gewählte Register geschrieben werden soll, die beiden Dropdown-Menüs dienen zur Auswahl der Registeradresse über Tabellenzeile und -spalte. Direkt darunter befindet sich eine kleine Tabelle, die die Special Function Register in hexadezimal sowie binär anzeigt; führende Nullen werden dabei ausgeblendet.

Der markanteste Teil des grafischen Nutzerinterface (engl. Graphical User Interface, GUI) ist die große Tabelle in der Mitte, in die der eingelesene Code zeilenweise angezeigt wird. Dabei wird jede Codezeile aufgeteilt in Program Counter, den Befehl (in Hexadezimal-Schreibweise), Codezeile, Label (falls vorhanden) und Mnemonic, also das für Menschen lesbare Kürzel des Befehls. In der ganz linken Spalte der Tabelle lassen sich (sofern es eine Zeile mit gültigem Code ist) durch Anklicken Breakpoints für die jeweilige Zeile setzen.

Im rechten Drittel des Fensters findet sich der Teil, bei dem der User den Simulator steuern kann. Mit je einem Knopf am oberen Rand lässt sich die Simulation Starten/Fortsetzen, Stoppen/Pausieren, auf Start zurücksetzen und zeilenweise durchlaufen. Im Bereich darunter findet sich rechts eine kleine Tabelle mit der Virtualisierung des Stacks, hier allerdings in Dezimalcodierung. Links von dieser Tabelle befindet sich ein mit „Timing“ betitelter Slider zur Anpassung der Verzögerung zwischen der Ausführung von Befehlen; er ist also für die Simulations-„Geschwindigkeit“ verantwortlich. Im selben Feld direkt unter dem Slider ist zum einen der Laufzeitzähler und zum anderen eine Anpassungsmöglichkeit per Dropdown für die Quarz-Frequenz. Auch der Button für den Master Clear (MCLR) befindet sich hier.

Direkt unterhalb der Simulatorsteuerung findet sich schließlich die I/O-Funktion über Port A und Port B mit einer Visualisierung des TRIS-Registers mit einem i für Input und einem o für Output. Wenn Output für das entsprechende Bit ausgewählt ist, wird der Radiobutton ausgegraut und ist nicht mehr klickbar. Wenn am entsprechenden Pin eine 1 ausgegeben wird, wird der Radiobutton als ausgewählt angezeigt.

Ganz unten rechts befindet sich erneut die Special Function Register, hier allerdings auf jedes Bit einzeln aufgeteilt und mit kurzem Erklärungstext für jedes einzelne Bit.

Konzeptionierung

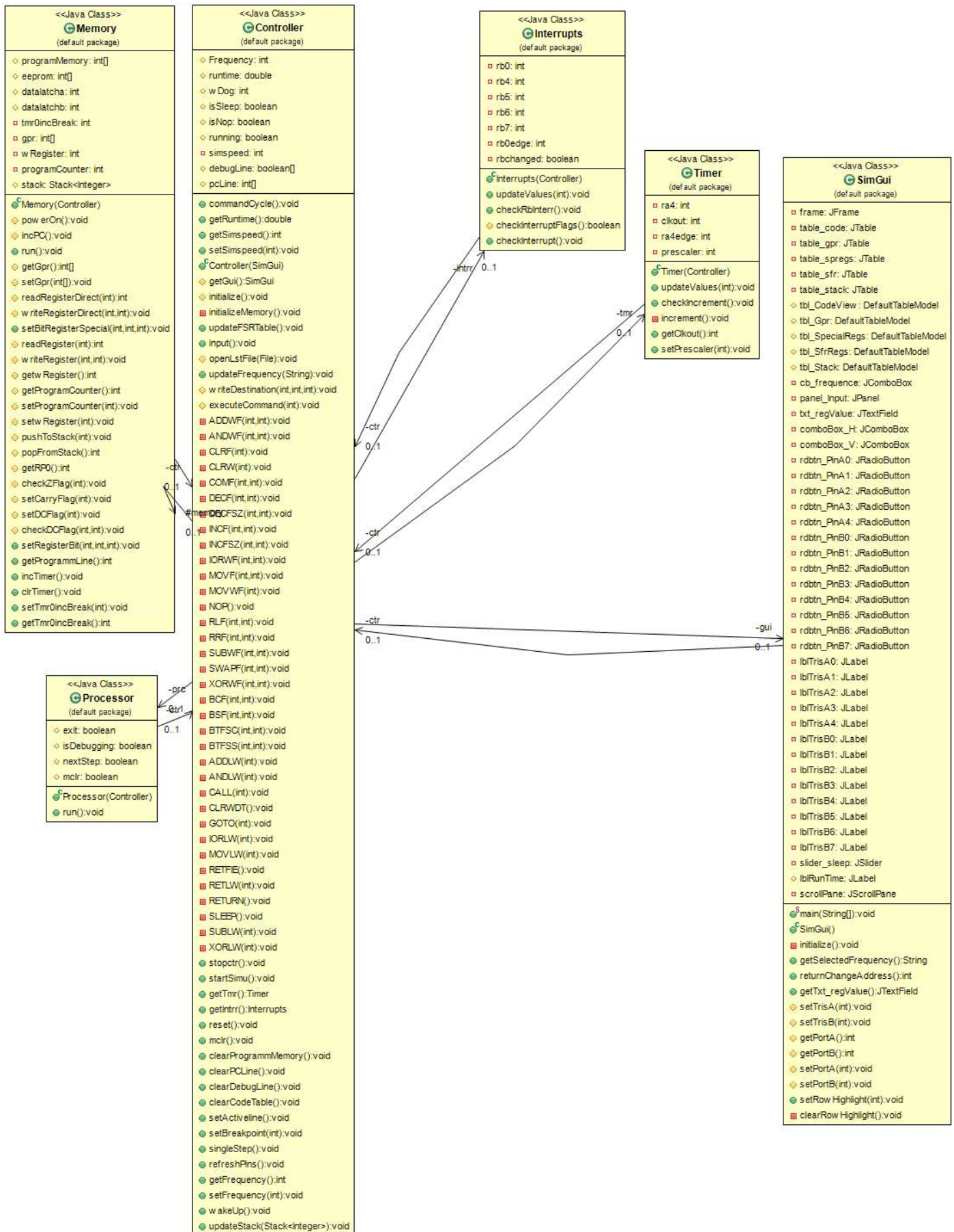


Abbildung 2: Vollständiges Klassendiagramm des Simulators

Die Hauptklasse des Projekts ist SimGui, da sie die main()-Methode enthält; sie ist außerdem für Aufbau und Handling des GUI verantwortlich. Für die Funktionsweise allerdings wurden die verschiedenen Aspekte des PIC auf Autoren: Mohammad Mehjazi, David Perk

mehrere Klassen aufgeteilt. Für Speicherfunktionen ist Memory verantwortlich, um die richtige Ausführung der Simulatorsteuerung (z.B. Befehlsausführung starten oder Breakpoints/Debugmodus) kümmert sich Processor, die Klassen Timer und Interrupts tun exakt das, was ihre Namen andeuten (Interrupt- und Timerhandling). Die optisch größte und auch die für den Simulator wohl funktionell wichtigste Klasse ist der Controller. Sie ist unter Anderem verantwortlich für das Dekodieren und ausführen der Befehle, das Handling der Codetabelle im GUI und das Berechnen der Runtime mithilfe der Quarzfrequenz.

Die Realisierung des General Purpose Registers erfolgt mithilfe eines Arrays vom Typ Integer in der Memory-Klasse. Die Adresse eines Registers entspricht der Adresse im Array und sein Wert dem Wert an der entsprechenden Stelle im Array. Über eine dedizierte Schreibmethode wird sowohl die Plausibilitätsprüfung des zu schreibenden Werts ($0 \leq x \leq 255$) als auch die Berücksichtigung des RPO-Bits und einer eventuellen indirekten Adressierung gewährleistet. Ebenfalls gibt es eine Funktion, die auf Basis der Registeradresse, die als Parameter übergeben werden muss, den dazugehörigen Registerwert zurückgibt.

Ebenfalls in der Klasse Memory befinden sich die Methoden, die bei Bedarf die Status-Bits 0, 1 und 2 setzen (Carry, Digit-Carry und Zero-Flag). Wenn eines dieser Bits bei der Ausführung eines Befehls berücksichtigt werden muss, werden diesen Methoden das Ergebnis beziehungsweise die beiden Ausgangswerte per Methodenparameter zur Verfügung gestellt und auf Basis dessen werden die Bits entsprechend gesetzt.

Die Befehle des PIC-Mikrocontrollers werden in einem 14-Bit Opcode codiert, der neben dem Befehl an sich auch die Parameter wie Destinationbits und Speicher-Adressen beinhaltet. Für den Decoder innerhalb des Controllers muss dieser Opcode in seine Bestandteile aufgeteilt. Laut Spezifikation des PIC16F84 wird dabei der Befehl mit den höchstwertigsten Bits codiert, diese sind also bei erster Analyse die wichtigsten Bits. Der Opcode wird bitweise aufgeteilt und analysiert:

Anhand der zwei höchstwertigsten Bits (dem Precommand) wird eine Vorsortierung vorgenommen, mithilfe derer die Bedeutung der folgenden zwölf Bits erfasst werden kann, da diese sich abhängig vom Precommand unterscheiden.

Bei Befehlen aus der Gruppe der bitweisen Adressierung, also jenen mit dem Precommand 01, wird der Befehl anhand der Bits 11 und 10 identifiziert und per Fallunterscheidung ausgewählt; die Adresse des Bits ist codiert in den Bits 9-7, im letzten Teil des 14-stelligen Opcodes (also den Bits 6-0) steht die Adresse des Registers, in dem das Bit gesetzt werden soll, beide Adressen werden der jeweiligen Java-Methode als Parameter mitgegeben.

Äquivalent zu dieser Bit-Aufteilung wird auch für die anderen Precommands eine Aufteilung vorgenommen. Falls es, wie beim Bit 7, das sowohl ein Destinationbit als auch eine Unterscheidung zwischen CLRF und CLRW bedeuten kann, eine Bedeutungsüberschneidung bei einzelnen oder mehreren Bits gibt, werden mehrere Variablen mit Werten bestückt. Eine leicht verkürzte Übersicht über die gesamte Dekoderstruktur bietet die Code-Zusammenfassung auf der nächsten Seite.


```

int precommand = (line >> 12) & 0x0003;
if (precommand == 0) {
    int command = (line >> 8) & 0x000f;
    int payload = line & 0x00ff;
    int d = payload >> 7 & 0x0001;
    int f = payload & 0b01111111;
    if (f == 0x00 | f == 0x80) {f = this.memory.readRegisterDirect(0x04);}
    int w = this.memory.getwRegister();
    if (command == 7) {this.ADDWF(f, d);}
    else if (command == 5) {this.ANDWF(f, d);}
    else if (command == 1)
    {
        if (d == 1) {this.CLRf(f);}
        else {this.CLRW(w);}
    }
    else if (command == 9) {this.COMF(f, d);}
    else if (command == 3) {this.DECF(f, d);}
    else if (command == 11) {this.DECFSZ(f, d);}
    else if (command == 10) {this.INCF(f, d);}
    else if (command == 15) {this.INCFSZ(f, d);}
    else if (command == 4) {this.IORWF(f, d);}
    else if (command == 8) {this.MOVF(f, d);}
    else if (command == 0)
    {
        if (d == 1) {this.MOVWF(f);}
        else if (payload == 0b01100100) {this.CLRWDT();}
        else if (payload == 0b00001001) {this.RETFIE();}
        else if (payload == 0b00001000) {this.RETURN();}
        else if (payload == 0b01100011) {this.SLEEP();}
        else {this.NOP();}
    }
    else if (command == 13) {this.RLF(f, d);}
    else if (command == 12) {this.RRF(f, d);}
    else if (command == 2) {this.SUBWF(f, d);}
    else if (command == 14) {this.SWAPF(f, d);}
    else if (command == 6) {this.XORWF(f, d);}
} else if (precommand == 1) {
    int command = (line >> 10) & 0x0003;
    int payload = line & 0x03ff;
    int b = (payload >> 7) & 0x0007;
    int f = payload & 0x007f;
    if (f == 0x00 | f == 0x80) {f = this.memory.readRegisterDirect(0x04);}
    if (command == 0) {this.BCF(f, b);}
    else if (command == 1) {this.BSF(f, b);}
    else if (command == 2) {this.BTFSC(f, b);}
    else if (command == 3) {this.BTFSS(f, b);}
} else if (precommand == 2) {
    int command = (line >> 11) & 0x0001;
    int k = line & 0x07ff;
    if (command == 0) {this.CALL(k);}
    else if (command == 1) {this.GOTO(k);}
    else
        System.out.println("Neither CALL nor GOTO");
} else if (precommand == 3) {
    int command = (line >> 8) & 0x000f;
    int k = line & 0x00ff;
    if (command == 8) {this.IORLW(k);}
    else if ((command >> 1) == 7) {this.ADDLW(k);}
    else if (command == 9) {this.ANDLW(k);}
    else if ((command >> 2) == 0) {this.MOVLW(k);}
    else if ((command >> 2) == 1) {this.RETLW(k);}
    else if (command == 10) {this.XORLW(k);}
    else if ((command >> 1) == 6) {this.SUBLW(k);}
    else {this.NOP();}
}
}

```

Konkrete Implementation

Da bei der Implementation der einzelnen Befehle die für den Befehl nötigen Parameter wie zum Beispiel die Registeradresse oder die Bit-Position per Methoden-Parameter direkt aus dem Decoder übergeben werden, war es möglich, die Methoden ohne Eingabeüberprüfung zu schreiben. Ein Beispiel dafür ist die Methode BTFSS, die die Registeradresse *f* und die Bitposition *b* übergeben bekommt. Eine in der Speicherklasse integrierte Methode erlaubt das Abrufen des Registerwertes. Durch gezieltes Shifting eines einzelnen gesetzten Bits wird eine Maske erzeugt, mithilfe derer bei Verundung das zu prüfende Bit isoliert werden kann. Durch Zurückshifting wird dieses isolierte Bit an die Position 0 gebracht, von wo aus ein simpler numerischer Vergleich ($=1$?) die Bitüberprüfung ermöglicht. Im Falle eines gesetzten Bits wird hier der Programcounter inkrementiert und ein NOP ausgeführt. In jedem Falle, unabhängig ob eine Anweisung übersprungen wird oder nicht, wird zum Abschluss ein erneuter Programcounter-Inkrement durchgeführt, sodass am Ende entweder der nächste oder der übernächste Befehl ausgeführt wird.

```
private void BTFSS(int f, int b) {
    System.out.println("BTFSS");
    int fValue = this.memory.readRegister(f);
    int bitMask = 0x01;
    bitMask = bitMask << b;
    fValue = fValue & bitMask;
    if ((fValue >> b) == 1) {
        this.memory.incPC();
        this.isNop = true;
    }
    this.memory.incPC();
}
```

Abbildung 4: Code-Ausschnitt Methode BTFSS

Eine weitere wichtige Funktion des PIC sind Interrupts, für die eine eigene Klasse angelegt wurde. In dieser werden zum einen die Interrupt-Flags überprüft (ob ein Interrupt überhaupt ausgegeben wird), zum anderen aber auch die entsprechenden Interrupts an den dazugehörigen Pins ausgegeben. Mit verschiedenen Überprüfungs- und Update-Methoden und -Funktionen wird so ein zuverlässiges Interrupthandling ermöglicht.

Die I/O-Funktionen der Port-Register wurden wie oben bereits beschrieben mit Radiobuttons ermöglicht, das grafische Handling derer wird von der Klasse SimGui übernommen, die Wertsteuerung erfolgt allerdings über die Controller-Klasse. In einer Refresh-Methode liest dieser die TRIS- und PORT-Register sowie die Radiobuttons des GUI aus und durchläuft diese Bit für Bit, bis am Ende der neue Wert feststeht und am GUI ausgegeben werden kann. Der Code für diese Refresh-Methode befindet sich auf der nächsten Seite.

```

// Refresh I/O Pins
public void refreshPins() {
    int trisa = this.memory.readRegisterDirect(0x85);
    int trisb = this.memory.readRegisterDirect(0x86);
    int dataa = this.memory.datalatcha;
    int datab = this.memory.datalatchb;
    this.getGui().setTrisA(trisa);
    this.getGui().setTrisB(trisb);
    int ra = getGui().getPortA();
    int rb = getGui().getPortB();
    for (int i = 0; i < 8; i++) {
        if ((trisa & 0x01) == 1) {
            this.memory.setBitRegisterSpecial(0x05, i, ra & 0x01);
        } else {
            this.memory.setBitRegisterSpecial(0x05, i, dataa & 0x01);
        }
        dataa = dataa >> 1;
        trisa = trisa >> 1;
        ra = ra >> 1;
        if ((trisb & 0x01) == 1) {
            this.memory.setBitRegisterSpecial(0x06, i, rb & 0x01);
        } else {
            this.memory.setBitRegisterSpecial(0x06, i, datab & 0x01);
        }
        datab = datab >> 1;
        trisb = trisb >> 1;
        rb = rb >> 1;
    }
    getGui().setPortA(this.memory.readRegisterDirect(0x05));
    getGui().setPortB(this.memory.readRegisterDirect(0x06));
}

```

Abbildung 5: Refresh-Methode für TRIS- und PORT-Register

Konklusion

Rückblickend haben sich vor allem die Wahl der Programmiersprache und der Programmierstil negativ auf den Projektverlauf ausgewirkt. Unter anderem die „else if“-Struktur im Dekoder, aber auch viele ähnliche Strukturen an anderen Stellen, hätten im Nachhinein betrachtet einfacher und besser mit switch case Anweisungen abgearbeitet werden können. Solche Fehler lassen sich in erster Linie auf mangelnde Erfahrung mit der Programmiersprache zurückführen.

Ebenfalls ist an vielen Stellen aufgefallen, dass ein wenig mehr Arbeit in die Planung und Konzeptionierung hätte fließen müssen, da viele Methoden und Funktionsweisen an falschen Stellen im Code oder in den falschen Klassen gemanagt wurden. So waren Speicherfunktionen im Controller zu finden und eigentlich beim Prozessor angesiedelte Aufgaben wurden direkt in der GUI-Klasse bearbeitet. Ebenfalls gab es vor allem beim Timer und den Interrupts Rückschlüsse, die sich mit vorheriger Planung zum größten Teil hätten vermeiden lassen können.

Ebenfalls war eine Problematik das Nicht-Verwenden von Alias-Variablen, sodass eine Adresse stets als Integer-Literal notiert und nicht per Ersatztext direkt ersichtlich war. Eine externe Wartbarkeit ist dadurch sehr erschwert, weil zum Beispiel eine Statusregister-Adressierung nicht mit „STATUS“ und einem hinterlegten Integer-Wert erfolgt, sondern immer direkt mit 0x03. Dies hat einige vermeidbare Verwirrung beim Code-Schreiben und vor allem -Debuggen verursacht.

Autoren: Mohammad Mehjazi, David Perk