Abgabe zu Übungsblatt 3

Rechnerarchitektur und Eingebettete Systeme

Abgabedatum:	03.12.2015
Semester:	Wintersemester 2015/16
Beteiligte:	296118 Stefan Fuhrmann – Techn.Inf. HS Bremen 325585 Steffen Christiansen – Techn.Inf. HS Bremen 4140709 Daniel Tauritis – Inf. Uni Bremen
Gruppennummer:	1
Laborleitung /-Betreuung:	Oliver Keszöcze

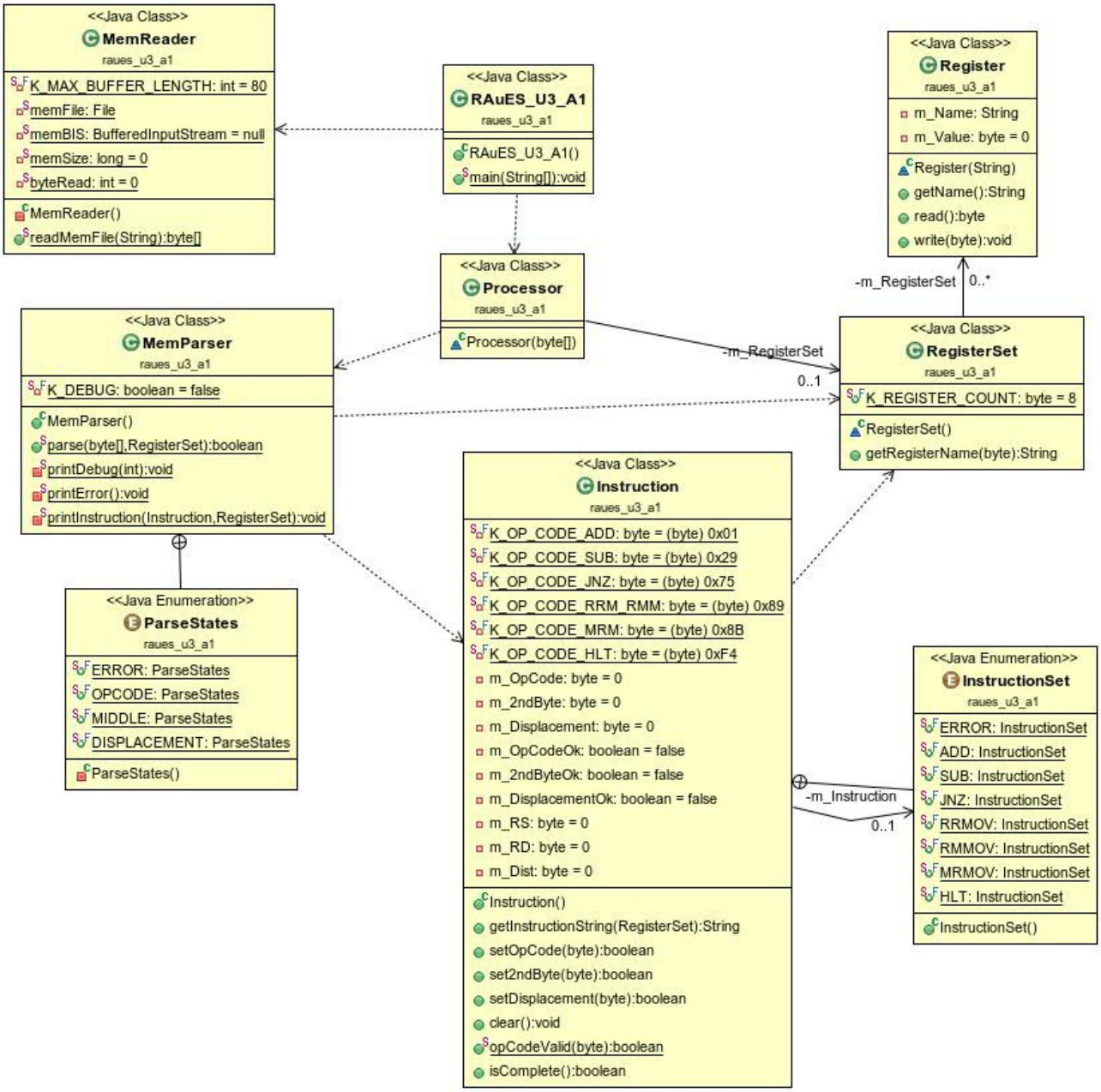
Einleitung

Dieses Dokument stellt lediglich Übersichtstexte zu der Implementierung dar. Es werden die Charakteristika der Implementierung (Eingabe, Ausgabe, Besonderheiten der Implementierung) beschrieben und der Programmablauf verdeutlicht. Der ausführlich kommentierte Quelltext wird dem Tutor in digitaler Form verfügbar gemacht.

Übung 3 Aufgabe 1

Klassendiagramm

In diesem Klassendiagramm werden die Zusammenhänge der Klassen und der Methoden deutlich, bevor sie später einzeln näher beschrieben werden.



Hauptklasse RauES_U3_A1.java

In der Main-Klasse lesen wir über die statischen Methoden der Klasse MemReader die Datei testfile.mem ein. Diese wird der Methode der Klasse als Argument übergeben. In der Klasse speichern wir den Inhalt lokal in einem Bytearray namens MemBuffer. Wir führen eine Fehlerbehandlung durch, um Exceptions abzufangen, falls kein Argument angegeben ist, mehr als ein Argument angegeben ist oder die Datei nicht gelesen werden konnte.

Wenn die Datei erfolgreich gelesen werden konnte, erstellen wir ein Objekt der Klasse Prozessor, dem wir den lokalen Array MemBuffer für die Ausführung übergeben. Dieser stellt den lokalen Speicher des Prozessorobjektes dar. Das ganze Konstrukt hüllen wir in einen try-catch Block, um auf Exceptions reagieren zu können und daraufhin das Programm automatisch zu beenden.

MemReader.java

Die Klasse MemReader stellt statische Methoden zur Verfügung, mit denen *.mem-Dateien eingelesen werden können.Dazu muss der Dateipfad übergeben werden. Es kann eine gegebene Memory-Datei mit bis zu 80 Byte Länge eingelesen und als ByteArray zurückgegeben werden.

Variablen

[private] [static] [final] (Integer) K_MAX_BUFFER_LENGTH	[Konstante] Über diesen Integerwert wird die maximale Länge des Registerspeichers angegeben. In unserem Fall ist der interne Speicher des Prozessors auf 80 Byte begrenzt.
[private] [static] (File) memFile	Eine lokale Variable, die in der Methode readMemFile() den absoluten Dateipfad der angegebenen *.mem-Datei speichert.
[private] [static] (BufferedInputStream) MemBIS	Ein Objekt der Bibliotheksklasse BufferedInputStream, die wir verwenden um die eingegebenen *.mem-Dateien zeilenweise einzulesen. Wir initialisieren es zuerst mit 'null', da wir dem Objekt erst in der Methode readMemFile() die memFile übergeben.
[private] [static] (long) memSize	Eine lokale Variable vom Datentyp 'long', dem wir später in der Methode readMemFile() die Zeilenlänge des einzulesenden Objektes übergeben. Anhand dieses Wertes führen wir eine Fehlerbehandlung durch.
[private] [static] (Integer) byteRead	Dies ist die Zählervariable, die angibt, in welcher Zeile wir uns beim Leseprozess befinden. Sie wird als Zählervariable für die Methode readMemFile() benutzt.

Methoden

[private] MemReader()	Konstruktor der Klasse. Da diese aber nur statische Methoden aufweist, ist der Konstruktor leer.
[public] [static] (byte[]) readMemFile([String] fileName)	Diese Methode wird mit einem String aufgerufen, der als Inhalt die absolute Adresse der *.mem-Datei beinhaltet. Zunächst wird über eine Fehlerbehandlung überprüft, ob die Zeilenlänge des Inhaltes die maximal zulässige Länge des Arrays memBuffer von 80 überschreitet. Sollte dies der Fall sein, wird das Programm nicht abgebrochen, sondern nur darauf hingewiesen, dass alle zusätzlichen Bytes ignoriert werden. Weiterhin wird überprüft, ob die Zeilenlänge der Datei kleiner oder gleich 'null' ist, damit wird abgefangen, ob eine leere oder korrupte [negative Zahlen] eingelesen wird. Wenn die Datei zulässig ist, wird sie zeilenweise eingelesen und der Inhalt in das lokale Array memBuffer geschrieben. Dabei wird ebenfalls noch einmal vor dem Einlesen überprüft, ob der BufferedInputStream überhaupt bereit ist. Falls nicht, wird eine Exception geworfen. Letztlich ist die gesamte Methode in einen try-catch Block gehüllt. Dabei werden NullPointerExceptions, FileNotFoundExceptions, IOExceptions und normale Exceptions abgefangen. Am Ende der Methode wird über eine 'finally'-Klausel der BufferedInputStream geschlossen und die Variablen auf die initialen Werte zurückgesetzt, um auf die nächste Leseoperation vorbereitet zu sein. Die Methode gibt den eingelesenen Inhalt der *.mem-Datei als Inhalt des Arrays memBuffer[] per return zurück.

Processor.java

In der Main-Klasse wird ein Objekt dieser Klasse erzeugt, dem ein Bytearray mit dem Inhalt der eingelesenen *.mem-Datei übergeben wird. Die Klasse Prozessor erzeugt bei seiner Initialisierung im Konstruktor ein neues Registerset-Objekt, dass die entsprechenden Y86-Register [laut Aufgabenstellung, Klasse wird später beschrieben] beinhaltet.

Variablen

[private] (RegisterSet) m_RegisterSet	Ein Objekt der Klasse RegisterSet, dass die betreffenden Register des Prozessors enthält [laut Aufgabenstellung].

Methoden

[Konstruktor] Processor	Im Konsturktor wird wie oben bereits beschrieben, das zugehörige RegisterSet als Objekt der Klasse RegisterSet() erzeugt. Anschließend wird in einer try-catch Klausel der statischen Methode 'parse' der Klasse MemParser der übergebene Bytevektor memBuffer und das erstellte RegisterSet übergeben. Der Methode 'parse' liest nun den Inhalt des Bytevektors memBuffer und versucht ihn, laut der Aufgabenstellung, zu interpretieren. Bedeutet, es wird überprüft, ob sich in der einzulesenden Datei nur reguläre Befehle befinden, die der Prozessor auch anwenden kann. Aus der Klasse RegisterSet werden hier nur die Mnemonic's für die Kommandozeileausgabe verwendet. Es wird noch nicht in die Register geschrieben. Sollte das Parsen fehlschlagen, wird dies über eine IF-Klausel abgefangen und eine Exception ausgelöst, dass die MemFile nicht geparsed werden konnte. Über den try-catch Block werden sonstige Exceptions abgefangen.

Register.java

Diese Klasse bildet ein abstraktes Register ab. Dabei können Objekte der Klasse erzeugt werden, die zwingend über einen Registernamen verfügen müssen. Ein Register ist dabei immer 1 Byte oder eben 8 Bit breit.

Variablen

[private] (String) m_Name	Eine Instantiierung eines String-Objektes, dass den Namen des Registerobjektes beinhalten soll.
[private] (Byte) m_Value	Der Speicherinhalt des Registers, dazu wird der in Java gegebene Datentyp Byte genutzt. Dieser bildet ein Byte aus acht Bit ab. Das Objekt wird mit 0 initialisiert.

Methoden

[Konstruktor] Register(String name)	Der Konstruktor der Klasse überprüft, ob bei der Erzeugung eines Registerobjektes eine Name übergeben wurde. Sollte kein Name übergeben werden, schlägt die Erzeugung des Objektes fehl.
[public] (String) getname()	Eine Getter Methode, die den Namen des betreffenden Objektes zurückgibt.
[public] (Byte) read()	Eine Getter Methode, die den Inhalt des Registers [der Speicherzelle] zurückgibt.
[public] (void) write(byte value)	Eine Setter-Methode, die den übergebenen Inhalt 'value' in das betreffende Register [die Speicherzelle] schreibt.

RegisterSet.java

Die Klasse RegisterSet erzeugt die Prozessortypischen Register des Y86-Prozessors. Dabei wird ein Array aus Objekten von Register erzeugt, die intern per Hexadezimal über einen 8-Bitvektor adressiert werden. Die Adressierung wurde hierbei der Aufgabenstellung entnommen. Über eine Methode können die Namen der einzelnen Register abgefragt werden.

Variablen

[public] [static] [final] (byte) K_REGISTER__COUNT	[Konstante] Eine Konstante, die in Byte angegeben wird, die angibt, wie viele Register das Array von Objekten des Typs Register halten können soll.
[private] (Register[]) m_RegisterSet	Das eigentliche Array des Obekttyps Register, das die typischen Register des Y-86 Prozessors enthält.

Methoden

[Konstruktor] RegisterSet()	Im Konstruktor der Klasse wird das Array von Objekten des Typs Register erzeugt. Dabei wird auf Grundlage der Konstante K_REGISTER_COUNT die Länge des Arrays angegeben. Das RegisterSet wird mit Byte-Adressen adressiert, die über Hex-Werte angegeben sind. Die einzelnen Registerobjekte werden mit den typischen Namen der Y-86 Register initialisiert. Das Ganze ist in einen try-catch Block gehüllt, damit etwaige Exceptions abgefangen werden können.
[public] (String) getRegisterName(byte register)	Diese Methode überprüft zunächst, ob der übergebene Wert in Byte größer 0 und kleiner als die maximale Größe des Registerarrays K_REGISTER_COUNT ist und liest dann über die Methode der Klasse Register 'getName()' den Namen des Registers an der betreffenden Stelle aus. Wenn der übergebene Wert nicht den erwarteten Spezifikationen entspricht, wird 'null' zurückgegeben.

MemParser.java

Diese Klasse bildet einen Zustandsautomaten ab, der überprüft, ob sich in der eingelesenen Datei MemBuffer nur ByteCode befindet, der vom Prozessor auch interpretiert werden kann. Dabei wird in der statischen Methode 'parse()' ein Objekt der Klasse 'Instruction' erzeugt, das noch einmal in einer while-Schleife genauer überprüft, ob auch alle nötigen Informationen zu den aufgerufenen Instruktion im Bytecode vorhanden sind.

Das Ergebnis der Überprüfung wird über verschiedene statische Klassenmethoden in der Kommandozeile ausgegeben. Wenn die Methode richtig interpretiert werden konnte, wird die Methode 'getInstructionString()' der Klasse 'Instruction' aufgerufen, die einen String der ausgeführten Operation und die zu verwendenden Register zurückgibt. Dieser String wird dann in der Kommandozeile ausgegeben.

Zusätzlich wird der Interpretierte Bytecode als Hex-Wert ausgegeben.

Wenn der Bytecode nicht interpretiert werden konnte, wird der Text „Error“ in der Kommandozeile ausgegeben.

Variablen

[private] [static] [final] (boolean) K_DEBUG	Ein internes Flag, das gesetzt werden kann, um zusätzliche Debug-Informationen auszugeben.
[private] (enum) ParseStates	Ein Enumerator, der die einzelnen Zustände des Zustandsautomaten beinhaltet.

Methoden

[Konstruktor] MemParser	[Konstruktor] Der Konstruktor ist hier eine leere Methode, da nur statische Methoden vorhanden sind.
[public] [static] (boolean) parse(byte memBuffer[], RegisterSet registerSet)	Diese Methode setzt einen Zustandsautomaten um, der anhand von Methoden der Klasse 'Instruction' überprüft, ob der Bytecode, der sich in MemBuffer befindet, richtig interpretiert werden kann. Dabei wird memBuffer in einer While-Scheife durchlaufen und es wird überprüft, ob zu jeder Instruktion auch die geforderten Informationen mit im ByteCode vorhanden sind. Dabei werden die Zustände OPCODE, MIDDLE, DISPLACEMENT und ERROR unterschieden. Die While-Schleife wird dabei solange durchlaufen, bis das Ende des Arrays erreicht ist. Wenn eine Operation richtig interpretiert wurde und das Ende des Arrays noch nicht erreicht wurde, wird in den Initialzustand des Zustandsautomaten zurückgesprungen. Der Initialzustand ist OPCODE. Im Zustand OPCODE wird überprüft, ob es sich um eine der bekannten Y86-Operationen handelt. Wenn dies zutrifft und es sich um eine 1 Byte lange Instruktion handelt, kann es sich nur um die HALT-Instruktion handeln und der interne Instruction-Pointer currentByte wird hochgezählt. Der Zustand bleibt unverändert in OPCODE. Wenn die Instruktion mindestens

	2 Byte lang ist, wird der Zustand geändert und es wird in den Zustand MIDDLE gewechselt. Im Zustand MIDDLE wird übeprüft, ob es sich um eine valide Operation handelt. Wenn die Instruktion nach 2 Byte komplett ist, wird zurück in OPCODE gewechselt. Handelt es sich um eine 3 Byte Instruktion, wird in den Zustand DISPLACEMENT gewechselt. Im Zustand DISPLACEMENT wird überprüft, ob es sich um eine valide Operation handelt, trifft dies zu, wird zurück in den Zustand OPCODE gewechselt. Sollte zu irgendeinem Zeitpunkt ein Fehler in der Interpretation vorliegen, wird in den Zustand ERROR gewechselt und die Methode 'printError()' aufgerufen. Der interne Instruction-Pointer wird inkrementiert, damit er auf die nächste gültige Instruktion zeigt und es wird in den Zustand OPCODE zurückgewechselt. Das ganze Methodenkonstrukt ist in einen try-catch Block gehüllt um etwaige Exceptions abfangen zu können.
[private] [static] (void) printDebug(int line)	Diese Methode gibt die betreffende Bytecodezeile aus, die gerade interpretiert wurde.
[private] [static] (void) printError()	Ein Interpretationsfehler liegt vor, der Text „Error“ wird ausgegeben.
[private] [static] (void) printInstruction(Instruction inst, RegisterSet registerSet)	Diese Methode erstellt intern einen String und ruft die Methode 'getInstructionString(registerSet)' der Klasse 'Instruction' auf. Diese Methode gibt auf Grundlage der aktuellen Instruktion nähere Informationen zu den verwendeten Registern und ggf. zum Zustand des Instruction-Pointers aus.

Instruction.java

Diese Klasse überprüft, ob es sich bei dem Bytecode um eine valide Instruktion handelt, die im Y86 Befehlssatz vorhanden ist. Dabei ist zu jeder Instruktion bekannt, über welche Informationen sie verfügen und in welcher Länge sie vorliegen muss.

Wenn eine Instruktion nicht über die geforderten Informationen verfügt, wird von der betreffenden Methode ein 'FALSE' zurückgegeben.

Die Methoden dieser Klasse dienen dem aufrufenden Objekt der Klasse MemParser als Abfrage, ob der betreffende Bytecode konsistent und valide ist oder eben nicht.

Variablen

[public] (enum) InstructionSet	Eine Typendefinition basierend auf dem Datentyp ENUM, der die Zustände der Klasse abbildet. Es kann sich entweder um eine der bekannten Instruktionen handeln oder um falschen Bytecode.
[private] [static] [final] (byte) K_OP_CODE_ADD	Der bekannte OPCODE aus der Aufgabenstellung, der als Datentyp Byte gespeichert wird und später für die Abfrage dient.
[private] [static] [final] (byte) K_OP_CODE_SUB	Der bekannte OPCODE aus der Aufgabenstellung, der als Datentyp Byte gespeichert wird und später für die Abfrage dient.
[private] [static] [final] (byte) K_OP_CODE_JNZ	Der bekannte OPCODE aus der Aufgabenstellung, der als Datentyp Byte gespeichert wird und später für die Abfrage dient.
[private] [static] [final] (byte) K_OP_CODE_RRM_RMM	Der bekannte OPCODE aus der Aufgabenstellung, der als Datentyp Byte gespeichert wird und später für die Abfrage dient.
[private] [static] [final] (byte) K_OP_CODE_MRM	Der bekannte OPCODE aus der Aufgabenstellung, der als Datentyp Byte gespeichert wird und später für die Abfrage dient.
[private] [static] [final] (byte) K_OP_CODE_HLT	Der bekannte OPCODE aus der Aufgabenstellung, der als Datentyp Byte gespeichert wird und später für die Abfrage dient.
[private] (byte) m_OpCode	Diese Variable speichert den ersten Teil der Instruktion lokal ab.
[private] (byte) m_2ndByte	Diese Variable speichert den zweiten Teil der Instruktion [falls vorhanden] lokal ab.
[private] (byte) m_Displacement	Diese Variable speichert den dritten Teil der Instruktion [falls vorhanden] lokal ab.
[private (boolean) m_OpCodeOk	Dieses Flag wird gesetzt, wenn der erste Teil der Instruktion richtig interpretiert wird.
[private (boolean) m_2ndByteOk	Dieses Flag wird gesetzt, wenn der zweite Teil der Instruktion [falls vorhanden] richtig interpretiert wird.
[private (boolean)	Dieses Flag wird gesetzt, wenn der dritte Teil der Instruktion [falls

m_DisplacementOk	vorhanden] richtig interpretiert wird.
[private] (byte) m_RS	Die lokale Variable für die Zwischenspeicherung des vom Bytecode übergebenen Inhaltes von 'edi' bei RMMOV-Instruktionen.
[private] (byte) m_RD	Die lokale Variable für die Zwischenspeicherung des vom Bytecode übergebenen Inhaltes von 'esi' bei MRMOV-Instruktionen.
[private] (byte) m_Dist	Diese Variable ist für den Menschen erstellt, da es bei JNZ-Instruktionen einfacher ist, den Code zu interpretieren, wenn man anstelle eines Registers eine 'Destination' angegeben ist.
[private] (InstructionSet) m_Instruction	Das Objekt des vorher erstellten Datentypen InstuctionSet. Hiermit wird der Zustand angegeben, ob es sich um eine Instruktion handelt, die gerade überprüft wird oder ob bereits ein ERROR vorliegt. Initial wird der Zustand ERROR initialisiert.

Methoden

[Konstruktor] Instruction	[Konstruktor] Der Konstruktor ist hier eine leere Methode, da nur ein Objekt erzeugt wird, dem aber keine Werte übergeben werden. Es werden nur die Klassenmethoden verwendet.
[public] (String) getInstructionString (RegisterSet registerSet)	Diese Methode gibt einen String zurück, der auf Grundlage des aktuellen Zustandes die verwendeten Register und die darauf ausgeführte Operation zurückgibt. Bei JNZ-Instruktionen wird die Sprungadresse ausgegeben. Bei Memory-Operationen wird zusätzlich das Displacement angegeben. Sollte ein ERROR oder ein unbekannter Zustand vorliegen, wird 'null' zurückgegeben.
[public] (boolean) setOpCode(byte opCode)	Diese Methode überprüft den ersten Teil des Bytecode, den Opcode auf Validität. Wenn es sich um eine HALT Instruktion handelt, ist die Überprüfung hiermit auch schon beendet.
[public] (boolean) set2ndByte(byte middle)	Auf Grundlage des zweiten Bytes der Instruktion, dass der Methode übergeben wird, wird überprüft, um welche Instruktion es sich handelt und die für die Instruktion verwendeten Register werden für die spätere Nutzung aus dem Bytecode [Adressen, der Y86 kann keine Immediates behandeln] extrahiert.
[public] (boolean) setDisplacement(byte displacement)	In dieser Methode wird das der Methode übergebene Displacement lokal gespeichert. Es wird nicht auf Konsistenz überprüft, da alle Werte hier möglich sind.
[public] (void) clear()	Diese Methode setzt alle Variablen, die für die Feststellung einer validen Instruktion nötig sind, auf die Initialwerte zurück, damit eine weitere Instruktion überprüft werden kann.
[public] [static] (boolean)	Diese Methode überprüft auf Grundlage der vorher initialisierten

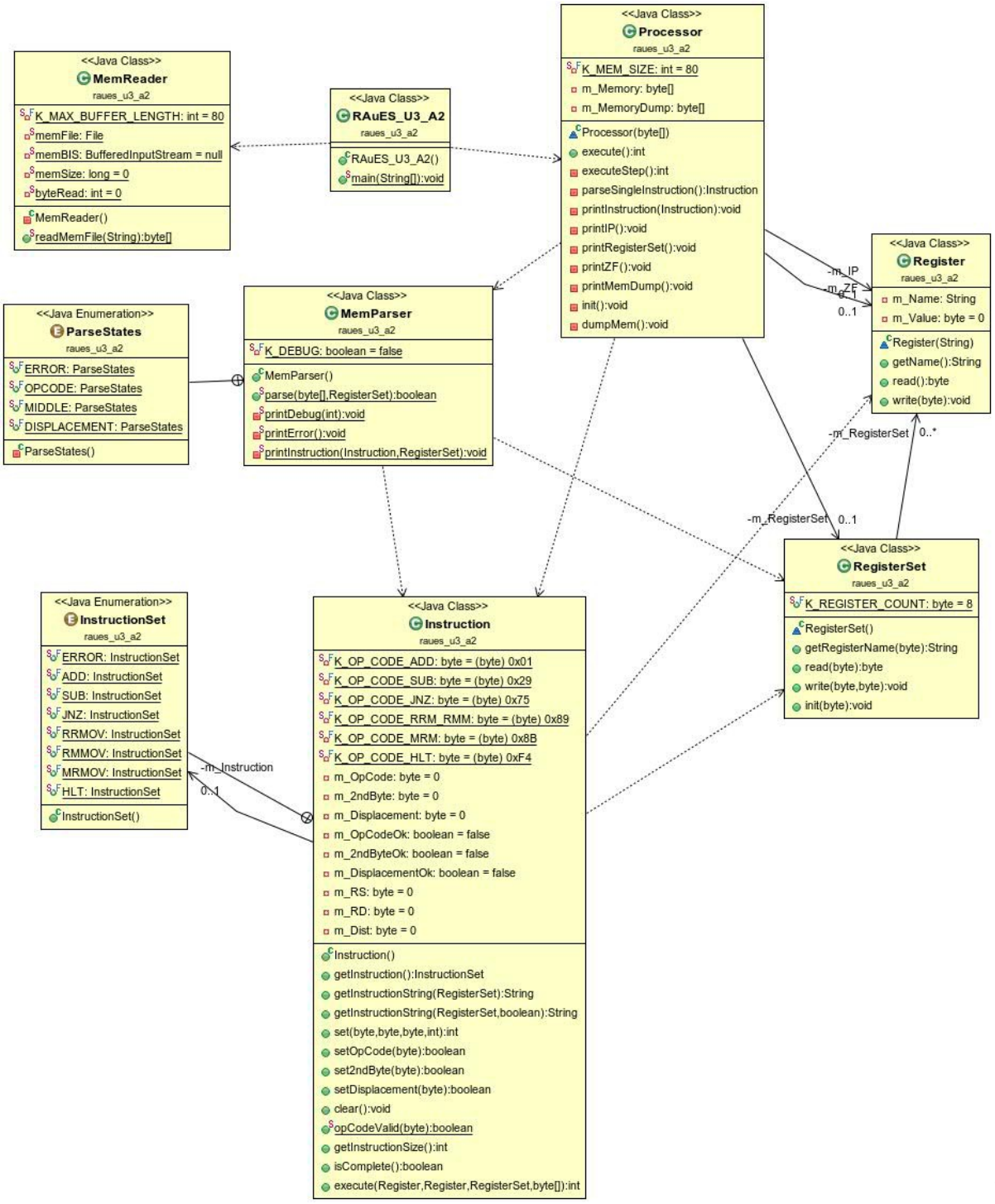
opCodeValid(byte opCode)	Bytes der Instruktionen (K_OP_CODE_ADD usw.) ob es sich um eine bekannte Instruktion handelt. Wenn nicht, wird 'FALSE' zurückgegeben.
[public] (boolean) isComplete()	Diese Methode prüft auf Grundlage des OPCODES und der Flags für m_opCodeOk, m_2ndByteOk und m_DisplacementOk ob die Instruktion bereits komplett überprüft wurde. Wenn die Instruktion komplett überprüft wurde, wird ein 'TRUE' zurückgegeben, wenn nicht, ein 'FALSE'.

Übung 3 Aufgabe 2

Diese Aufgabe baut auf der ersten Aufgabe auf. Neben dem Klassendiagramm, dass ganzheitlich gezeigt wird, werden nur die Änderungen und Erweiterungen genannt, die sich zur Umsetzung in Aufgabenteil 1 ergeben haben. Insbesondere MemReader.java, MemParser.java und Register.java sind zur vorherigen Aufgabe identisch.

Klassendiagramm

Dieses Klassendiagramm zeigt die Erweiterungen hinsichtlich Aufgabenstellung 1.



Erweiterung RAuES_U3_A2.java

Der Konstruktor der Main-Klasse wurde um einige Ausgaben und den Aufruf der Methode 'execute' der Klasse 'Processor' erweitert. Der Rückgabewert der Methode wird in der Variable 'res' gespeichert, die im Anschluss auch in der Kommandozeile ausgegeben wird.

Erweiterung RegiserSet.java

Die Klasse RegisterSet wurde um drei Methoden erweitert, die nun erlauben, die einzelnen Register auszulesen, in die Register zu schreiben und auch die Register zu initialisieren. Diese Umsetzung war nötig, da die Register nun auch wirklich verwendet werden und nicht nur die Namen der einzelnen Register im Vergleich zu Aufgabe 1 abgefragt werden.

Neue Methoden

[public] (byte) read(byte register)	Diese Methode gibt, bei gültiger Angabe für die Adresse des Registers, den Inhalt des betreffenden Registers aus. Sollte die Adresse nicht korrekt sein, wird eine Exception geworfen.
[public] (void) write(byte register, byte value)	Diese Methode schreibt bei gültiger Angabe für die Adresse des Registers, den Inhalt der übergebenen Variable 'value' in das betreffende Register. Sollte die Adresse nicht korrekt sein, wird eine Exception geworfen.
[public] (void) init(byte value)	Alle vorhandenen Register werden mit dem übergebenen Wert der Variable 'value' überschrieben.

Erweiterung Processor.java

Der Prozessor wurde um einige Variablen und einige Methoden erweitert. Da bei realer Programmausführung auch JNZ-Operationen ausgeführt werden können, benötigen wir einen Instruktion-Pointer und ein Zero-Flag. Diese beiden Elemente werden auf Grundlage eines Registers umgesetzt. Außerdem muss der Arbeitsspeicher oder auch das Memory sowie eine Möglichkeit geschaffen werden, den Inhalt des Arbeitsspeichers zu dumpen (MemoryDump). Der Inhalt des lokalen Arrays memBuffer wird nach erfolgreicher Überprüfung durch MemParser() in Memory geschrieben.

Die Methoden execute() ruft die Methode executeStep() solange auf, bis die letzte Instruktion in Memory abgearbeitet wurde. Die Methode executeStep() liest eine einzelne Instruktion aus dem Memory aus, parsed diese und führt sie aus. Wenn die Ausführung erfolgreich war, werden über verschiedene neue Print-Methoden, die aktuell ausgeführte Instruktion, der Zustand des Instruktion-Pointers, der Zustand des Registers, des Zero-Flags und des Memorys auf der Kommandozeile ausgegeben.

Neue Variablen

[private] (Register) m_IP	Ein Objekt der Klasse Register, in dem ein Byte gespeichert werden kann. In dem Register m_IP wird der Instruction-Pointer gespeichert, also die aktuelle Programmadresse.
[private] (Register) m_ZF	Ein Objekt der Klasse Register, in dem ein Byte gespeichert werden kann. In dem Register m_ZF wird das Zeroflag gespeichert. Dabei ist uns bewusst, das es sich beim Register um einen Bitvektor handelt. Die Umsetzung ist aber bewusst gewählt, da dieses Register später als Statusregister erweitert werden könnte. Bei der Zuweisung werden dabei alle Elemente entsprechend auf 0 oder 1 gesetzt [(byte)0 || (byte) 1].
[private] (byte) m_Memory	Das Bytearray für die Simulation des Memorys [Arbeitsspeicher] des Prozessors.Wird innerhalb des Konstruktors instantiiert. Wird nicht von Register abgeleitet, da Memory über keine Mnemonics verfügen muss.
[private] (byte) m_MemoryDump	Das Bytearray für die Zwischenspeicherung des Inhalts von Memory für Debugging oder Ausgabezwecke. Da sich der Memory mit jeder ausgeführten Instruktion ändert, kann der Inhalt von Memory in dieser Bytevektor gerettet werden.

Neue Methoden

[public] (int) execute()	Ruft die Methode init() auf, die die internen Register des Prozessors flusht [initialisiert]. Anschließend wird in einer Do-While-Schleife solange die Methode executeStep() ausgeführt, bis die letzte Instruktion im Memory ausgeführt wurde. Anschließend wird das Ergebnis des letzten Aufrufs der Methode executeStep() zurückgegeben.
[private] (int) executeStep()	Es wird eine einzelne Instruktion aus dem Memory gelesen und diese wird geparsed. Wenn sie den gegebenen Konventionen entspricht und auch definitiv über konsistente Daten verfügt, wird sie über den Aufruf der Methode execute() der Klasse 'Instruction' ausgeführt und das Ergebnis in der lokalen Variable 'result' gespeichert. Für die Ausführung der aktuellen Instruktion werden der Methode die Objekte des Instruktion-Pointers, der Zero-Flags, des RegisterSet's und des Memory's übergeben, bzw. werden sie intern referenziert.
[private] (Instruction) parseSingleInstruction()	Es wird ein Objekt der Klasse 'Instruction' erzeugt. Wenn der Instuktion-Pointer auf eine Adresse im Memory zeigt, die nicht das Ende des Array's ist, wird die 'set()' Methode der Klasse 'Instruction' aufgerufen. Je nachdem, wie viele Stellen man sich vor dem Ende des Array's befindet, werden die einzelnen Bytes für OPCODE, MIDDLE und DISPLACEMENT übergeben, die intern in der Methode 'set()' der Klasse 'Instruction' ausgewertet werden und in die entsprechenden Attribute des Objektes geschrieben werden. Nach dem Parsen der Instruktion wird sie per Return an den Aufrufer zurückgegeben.
[private] (void) printInstruction(Instruction inst)	Die Instruktion wird über eine Methode namens 'getInstructionString' der Klasse 'Instruction' mit entsprechenden Registern in einen String verpackt und dieser wird in der Kommandozeile ausgegeben.
[private] (void) printIP()	Der aktuelle Zustand des Instruktion-Pointers bzw. auf welche Adresse er zeigt, wird ausgegeben.
[private] (void) printRegisterSet()	Der aktuelle Inhalt des Prozessorregisters wird auf der Kommandozeile ausgegeben.
private void printZF()	Der aktuelle Inhalt des zero-Flags (oder des Statusregisters) wird in der Kommandozeile ausgegeben.
private void printMemDump()	Der Inhalt des Memory's vor Ausführung der letzten Instruktion wird ausgegeben.
private void init()	Der Instruktion-Pointer, das RegisterSet und das Zero-Flag

	werden mit mit 0 Byte initialisiert.
private void dumpMem()	Der aktuelle Inhalt des Memory's wird in MemoryDump kopiert.

Erweiterung Instruction.java

Die Klasse wurde um einige Methoden erweitert, die die Ausführung der Instruktionen eines Y-86 Prozessors nun auch gestatten. Es wurde eine neue getInstructionString()-Methode geschrieben, die nun auch neben den verwendeten Registern und der durchgeführten Operation, den Namen der Instruktion ausgibt. Dabei wurde die alte Methode beibehalten und kann weiterhin über einen Zusatz beim Aufruf genutzt werden. Die Methode wurde also überladen. Zudem wurde eine Methode implementiert, die den Zustand der Instruktion, also den Namen der Instruktion abfragt und diesen zurückgibt.

Über die neu implementierte set()-Methode wird eine einzige Instruktion geparsed und über den Rückgabewert wird der Aufrufenden Methode bekannt gegeben, ob das Parsen erfolgreich war. Die Methode getInstructionSize() prüft um welche Instruktion es sich handelt und gibt die Länge der Instruktion als Integerwert zurück.

Letztlich wurde die execute()-Methode umgesetzt, die die aktuelle Instruktion ausführt, in die entsprechenden Register schreibt und die nötigen Flags setzt.

Neue Methoden

[public] (InstructionSet) getInstruction()	Der aktuelle ENUM-Wert, also der Zustand der Instruktion, wird zurückgegeben. Dabei steht der Zustand der Instruktion für den Namen der ausgeführten Instruktion als Mnemnonic.
[public] (String) getInstructionString(RegisterSet registerSet)	Die alte Methode wurde erweitert und gibt nun neben den verwendeten Registern oder dem aktuellen Sprungziel bei JNZ-Instruktionen, die eigentlich ausgeführte Operation namentlich zurück.
[public] (String) getInstructionString(RegisterSet registerSet, boolean useHex)	Die alte Funktion aus Aufgabe 1 wurde um einen weiteren Eingabewert erweitert und überladen. Wenn man beim Methodenaufruf nun als zweites Argument ein 'TRUE' übergibt, ist es möglich, auch den alten 'InstructionString' zu erhalten.
[public] (int) set(byte opCode, byte middle, byte displacement, int argCnt)	Die set()-Methode bekommt Werte für OPCODE, MIDDLE und DISPLACEMENT übergeben und prüft diese auf Konsistenz und erfasst die betreffenden Register und um welche Operation es sich handelt. Der Rückgabewert gibt an, ob es sich um eine valide Operation handelt oder eben nicht, bzw. ob ein Error vorliegt. Der vierte übergebene Wert betrifft die Länge der übergebenen Argumente.
[public] (int) getInstructionSize()	Die Länge der aktuellen Instruktion wird als Integer zurückgegeben. Dabei bezieht sich die Länge der Operation auf ganze Bytes.
[public] (int) execute(Register ip, Register zf, RegisterSet regs, byte[] mem) throws Exception	Die Ausführung der Instruktion. Dabei werden je nach Typ der Instruktion die Werte aus den entsprechenden Registern gelesen und geschrieben, der Instruktion-Pointer wird inkrementiert bzw. bei JNZ auf die entsprechende Sprungadresse gesetzt und das

	Zero-Flag wird gesetzt. Wenn die Instruktion entweder die HALT-Instruktion oder eine andere valide Instruktion ist wird 0 oder 1 zurückgegeben. Bei einem Fehler wird -1 zurück gegeben.

Übung 3 Aufgabe 3

Die Abgabe dieser Aufgabe erfolgt derart, dass im folgenden nur die Mnemonics abgedruckt werden. Die verwendete Memory-Datei wurde als Inhalt des Ordners „RauES_U3_A3“ an den Tutor geschickt wird.

```
//The following bytes may be changed to alter the calculation:
//n in 0x07 (n>0 required!)
//f1 in 0x04 (previous fib-number)
//f2 in 0x05 (previous previous fib-number)

if !ZF then IP := IP + 8   //jnz - jump over literal values

eax := MEM[0x07]           //MRmov - save n to register 0 (MAY BE ALTERED!)
ecx := MEM[0x06]           //MRmov - save 1 to register 1

edc := MEM[0x04]           //MRmov - move f1 to register 2(now called a)(MAY BE ALTERED!)
ebx := MEM[0x05]           //MRmov - move f2 to register 3(now called b)(MAY BE ALTERED!)

//Loop begins here

eex := ebx                 //RRmov - move register 3 (b) to register 4 (temp)
eex := eex + edx           //add - add register 2 (a) to register 4 (temp)
ebx := edx                 //RRmov - move register 2 (a) to register 3 (b)
edx := eex                 //RRmov - move register 4 (temp) to register 2 (a)

eax := eax - ecx           //sub - subtract register 1 (1) from register 0 (n)

if !ZF then IP := IP -10   //jnz - loop if register 0 (n) did not reach 0

MEM[ea] := eex             //RMmov - move temp to address 0x4f (output bit)
Prozessor anhalten
```