Verstehen und Anwenden des IJVM-Befehlssatzes auf einer MAL Mikroarchitektur anhand ausgewählter Algorithmen mithilfe des Mic-1 MMV Simulators

Niklas Lampe – 5017616
Thade Struckhoff – 5016730

19. Juli 2021

Inhaltsverzeichnis

Aufgabe 1: Einführung

In dieser Aufgabe wird sich mit den Funktionalitäten des Mic-1 Simulators auseinandergesetzt. Zudem soll der Aufbau der IJVM-Programme verstanden werden.

1.1) Für die Bearbeitung dieser Aufgabe muss zunächst der Simulator gestartet werden. Dort wird das Programm 'echo.ijvm' geladen. Dabei lernen die Studierenden die Benutzung der Ein- und Ausgabekonsole kennen. Zudem lernen Sie die verschiedenen Ausführungsmöglichkeiten für Programme kennen. Für diese Teilaufgabe gibt es keine konkret nennbare Lösung, die Studierenden setzen sich lediglich mit dem Simulator auseinander.

1.2) In der zweiten Aufgabe soll nun die Entwicklung des Stacks händisch durchgeführt werden. Eine Lösung dazu ist in der nachfolgenden Abbildung 1 zu sehen.

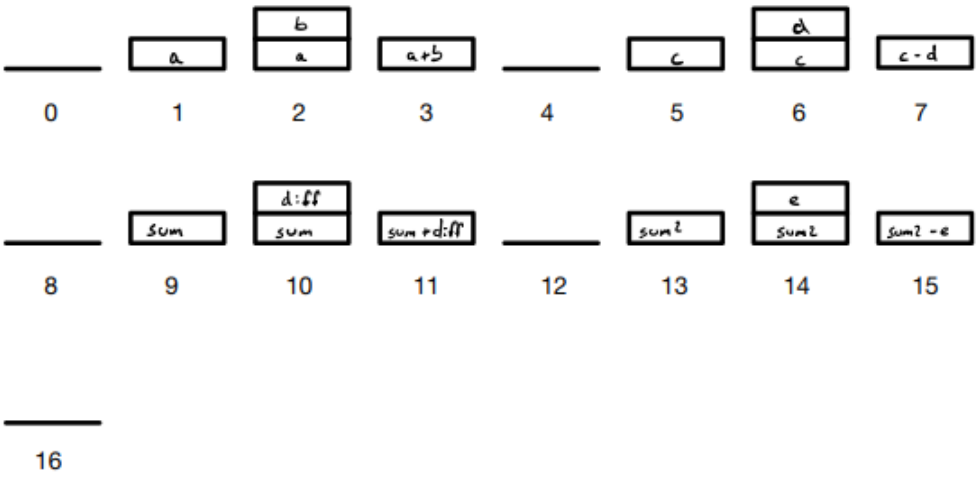

Abbildung 1: Entwicklung des Stacks der Rechenoperation

1.3) Bei der dritten Aufgabe lernen die Studierenden den typischen Aufbau eines Mikroprogrammes kennen. Dazu sollen sie das Programm 'rechenOperation.jas' in einem Editor öffnen und die Funktionalitäten beschreiben.

P1: Programmstart: Aufruf der main-Methode durch den Simulator

P2: Deklaration der lokalen Variablen

P3: Beginn der Ausführung des Methodenrumpfs

P4: Initialisierung der lokalen Variablen.

P5: Besonderheit für Variable 'c': Es kann nicht einfach der Wert 'AE' über `BIPUSH` geladen werden, da das angegebene Byte signed ist und somit aus dezimaler

Sichtweise den Wert '-82' repräsentiert. Deswegen wird die Methode 'fillVariable' aufgerufen, die zwei angegebene Bits entsprechend im Byte aufeinander schiebt. Dadurch kann der Wert 'AE' korrekt als '147' (dezimal) im Byte angegeben werden.

P6: Laden der Variablen a und b sowie addieren der Variablen von a und b und abspeichern in der Variablen sum

P7: Laden der Variablen c und d, subtrahieren der Variablen und abspeichern in Variable diff

P8: Laden der Variablen sum und diff, addieren der Werte und abspeichern in sum2

P9: Laden der variablen sum2 und e, subtrahieren der Werte und abspeichern in total

P10: Laden der Variablen total und Ausgabe in der Konsole mithilfe einer weiteren Untermethode, print.

P11: Ende der main-Methode. Durch den Befehl `HALT` erkennt der Simulator, dass das Programm zu ende ist.

Dies beschreibt ausreichend die Funktionalität des Programms. Auf die print-Methode muss nicht weiter eingegangen werden, da sie in diesem Fall nur ein Mittel zum Zweck ist und die Gesamtkomplexität dieser Aufgabe zu diesem Zeitpunkt zu sehr sprengen würde.

1.4) In der vierten Aufgabe soll das vorherige Ergebnis nun mit der Ausgabe im Simulator verglichen werden. Dazu muss zunächst das Programm 'rechenOperation.jas' assembliert und geladen sein. Beim Vergleich sollte auffallen, dass die händisch erarbeitete Stackentwicklung in dem Programm wiedergefunden werden kann. Allerdings ist dies nur ein Teil des gesamten Stackverlaufs. Vor der eigentlichen Berechnung wird nämlich noch eine Initialisierung der Variablen vorgenommen, was sich ebenfalls auf den Stack auswirkt. Außerdem wird nach der Berechnung ein Methodenaufruf zur Ausgabe des berechneten Wertes auf der Ausgabekonsole durchgeführt. Auch dieser ist auf dem Stack bei der Nachverfolgung zu sehen.

Aufgabe 2: Fehleranalyse

In dieser Aufgabe soll eine Fehleranalyse auf das IJVM-Programm 'ggT.jas' durchgeführt werden, wodurch sich die Studierenden ein besseres Verständnis über die Funktionsweise und das zielgerichtete Benutzen des Mic-1 Simulators erarbeiten. Das Programm beinhaltet fünf Fehler, die von den Studierenden gefunden und behoben werden sollen.

Abzugeben sind für diese Aufgabe die fehlerfreie Datei 'ggT.jas' sowie das durch den Simulator assemblierte Programm 'ggT.ijvm'.

2.1) Vorbereitend für diese Aufgabe muss der Simulator gestartet und mit dem Mikroprogramm 'mic1ijvm.mal' geladen sein, was im Standardfall (sofern vom Studierenden nicht verändert) automatisch gegeben ist. Nachfolgend werden die zu findenden Fehler erläutert und beschrieben, wie diese gefunden werden können. Eine vollständige Musterlösung bietet das IJVM-Programm 'ggT-musterloesung.jas'.

F1: Die globale Konstante 'OBJREF' ist nicht deklariert. Dies macht sich bei der Assemblierung in der Assembleransicht bemerkbar, wie in der Abbildung 2 für die Speicherstellen 1010, 1029, 1048, 1053 und 1071 zu sehen. Dies veranlasst die Studierenden, sich einen ersten Gesamtüberblick über das Programm zu verschaffen. Sie müssen ermitteln, welche Konstante nicht definiert ist und was diese für einen besonderen Sinn hat. Der besondere Sinn besteht in der Notwendigkeit für Methodenaufrufe. Wichtig ist dabei, dass die Studierenden erkennen, dass die Konstante global definiert werden muss, da sie gleichermaßen in verschiedenen Methoden Verwendung findet.

F2: Methode ggT - Die lokale Variable 'rest' ist nicht definiert. Dies macht sich bemerkbar durch die übrig bleibende Fehlermeldung beim assemblieren. Dies ist in Abb. 2 an der Speicherstelle '1076' abzulesen. Dieser Fehler veranlasst die Studierenden, sich auch wirklich mit der Methode zur Berechnung des ggT tiefer auseinanderzusetzen. Die Studierenden müssen erkennen, dass die benötigte Variable nur lokal definiert werden muss.

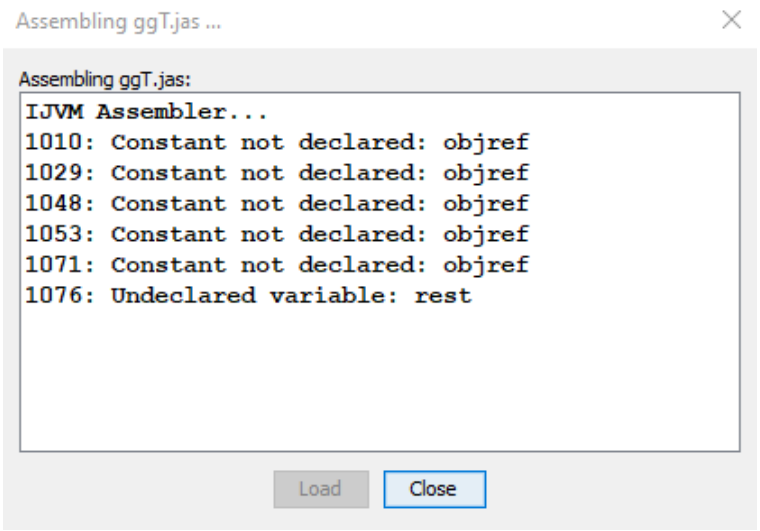

Abbildung 2: Fehlermeldung des Assemblierers für Fehler F1 und F2

Bis hierhin sollte die Fehleranalyse eher einfach sein und zügig vonstatten gehen, da der Assemblierer Hinweise gibt, was in etwa falsch sein könnte. Für die nachfolgenden Fehler muss eine dynamische Fehleranalyse durchgeführt werden, um zu verstehen, wie der Kern des Programms tatsächlich funktioniert und wo die Fehlerwirkungen ihren Ursprung haben. Dazu sollen die Ablaufsteuerung sowie die Breakpoints zielgerichtet eingesetzt werden.

F3: Methode mod - Die Abbruchbedingung der While-Schleife ist falsch. Es wird auf '$= 0$' anstatt '≤ 0' geprüft. Dadurch entsteht eine Endlosschleife zusammen mit der übergeordneten While-Schleife (in Methode ggT) für alle Berechnungseingaben, für die '$\neq 0$' gilt. Durch sinnvoll gesetzte Breakpoints an den entsprechenden Befehlen ist die Fehlerwirkung, der Loop, deutlich zu erkennen. Behoben wird dies durch den Austausch des Befehls 'IFEQ' mit 'IFLT', wie in der Musterlösungsdatei in Zeile 143 zu sehen ist.

F4: Methode ggT - Das Ablegen der Eingangsparameter auf den Stack vor dem Aufruf der mod-Methode ist nicht vorhanden. Die Studierenden erkennen dies bei der Sichtung des Quelltextes. Die (auch beschriebenen) Methodenparameter müssen erkannt werden. Dann muss geschlussfolgert werden, dass für einen Methodenaufruf natürlich ggf. auch zur Vorbereitung die entsprechenden Parameterwerte in korrekter Reihenfolge auf den Stack gelegt werden müssen. In der Musterlösungsdatei ist dies in den Zeilen '110' und '111' eingefügt.

F5: Methode ggT - Der berechnete Rückgabewert (gespeichert in der Variable 'a') wird vor dem Methodenrücksprung nicht auf den Stack gelegt. Dies zeigt sich an der fehlerhaften und immer gleichen Ausgabe '00008000', was dem Wert aus dem LV Register von vor dem Methodenaufruf darstellt. In der Musterlösungsdatei ist dies durch das Einfügen des Befehls 'ILOAD a' in der Zeile '119' behoben.

Aufgabe 3: Implementierung

In dieser Aufgabe sollen die Studierenden eigenständig (pro Laborgruppe) ein IJVM-Programm eines vorgegebenen Sortieralgorithmus erstellen. Es ist eine Schablone 'sortieralgorithmus.jas' vorgegeben, die die Studierenden als Grundlage verwenden sollen. Darin werden über die Eingabekonsole fünf Werte eingelesen, dann erfolgt die zu implementierende Sortierung und abschließend wird das 'Ergebnis' in der Ausgabekonsole ausgegeben.

Abzugeben sind für diese Aufgabe die erweiterte Datei 'sortieralgorithmus.jas' sowie das durch den Simulator assemblierte Programm 'sortieralgorithmus.ijvm'.

3.1) In dieser Aufgabe soll der vorgegebene Java-Algorithmus zunächst gesichtet werden. Die Antwort auf die gestellte Frage ist, dass der vorgegebene Sortieralgorithmus den *Selection Sort*-Algorithmus abbildet.

3.2) Eine mögliche Implementierung des Algorithmus ist in der Datei 'selectionsort-musterloesung.jas' enthalten. Der vorgegebene Java-Code ist durch Kommentare an entsprechenden Stellen im IJVM-Programm angegeben.

3.3) Es gibt viele verschiedene Testziele und daraus resultierende Testfolgen. Eine Möglichkeit ist in der Tabelle 1 aufgeführt.

Nr.	**Eingabe**	**Soll Ausgabe**[1]	**Ist-Ausgabe**[1]	**Status**
1	FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF	Eingabe: FFFFFFFF, FFFFFFFF, FFFFFFFF, FFFFFFFF, FFFFFFFF Ausgabe: FFFFFFFF, FFFFFFFF, FFFFFFFF, FFFFFFFF, FFFFFFFF	Eingabe: FFFFFFFF, FFFFFFFF, FFFFFFFF, FFFFFFFF, FFFFFFFF Ausgabe: FFFFFFFF, FFFFFFFF, FFFFFFFF, FFFFFFFF, FFFFFFFF	OK
2	1 5 A000 A00A B001	Eingabe: 1, 5, A000, A00A, B001 Ausgabe: 00000001, 00000005, 0000A000, 0000A00A, 0000B001	Eingabe: 1, 5, A000, A00A, B001 Ausgabe: 00000001, 00000005, 0000A000, 0000A00A, 0000B001	OK
3	1 FFFFFFF8 FFFFFAF8 0 FFFBFFF8	Eingabe: 1, FFFFFFF8, FFFFFAF8, 0, FFFBFFF8 Ausgabe: FFFBFFF8, FFFFFAF8, FFFFFFF8, 00000000, 00000001	Eingabe: 1, FFFFFFF8, FFFFFAF8, 0, FFFBFFF Ausgabe: FFFBFFF8, FFFFFAF8, FFFFFFF8, 00000000, 00000001	OK
4	A 8 6 4 2	Eingabe: A, 8, 6, 4, 2 Ausgabe: FFFBFFF8, FFFFFAF8, FFFFFFF8, 00000000, 00000001	Eingabe: A, 8, 6, 4, 2 Ausgabe: FFFBFFF8, FFFFFAF8, FFFFFFF8, 00000000, 00000001	OK

Tabelle 1: Testfolge zum *Selection Sort*-Algorithmus

[1]Beschreibt die visuelle Ausgabe in der Ausgabe-Konsole. Der Text 'Eingabe' mit seinen entsprechenden Werten und der Text 'Ausgabe' mit den sortierten Werten sind im Simulaotr jeweils in einer Zeile dargestellt.