Projektbericht im Modul Entwurf, Realisierung und Programmierung eines Mikrorechners im Wintersemester 2015/2016

Universität Hamburg

Fachbereich Technische Aspekte Multimodaler Systeme (TAMS)

Verfasser
Jammer, Tim
Bartels, Oliver
Günther, Finn
Klinger, Thomas
Joost, Sandro

3. Juni 2016

Inhaltsverzeichnis

1 Die Vision

Ziel des Projektes war der Entwurf und die Programmierung eines eigenen Mikrorechners. Wir haben uns entschieden, eine Lisp Machine zu realisieren, also einen Prozessor, dessen Befehlssatz und Befehlsausführung möglichst nah an der Semantik der Programmiersprache Lisp sind. Der Entwurf eines solchen Rechners war in den 1980er Jahren ein aktuelles Thema und hat verschiedene Realisierungen hervorgebracht [5]. Dennoch hat sich das Konzept nie bewähren können, u.a wegen hoher Kosten und Ineffizienz. Dennoch war es im Rahmen des Projektes eine interessante Alternative zu etablierten Ansätzen wie 1-,2- oder 3-Adressmaschinen. Zur Realisierung des Projektes gab es mehrere Alternativen im Entwurf, z.B als Stackmaschine oder angelehnt an die Lisp Maschinen der 1980er von Symbolics [5]. Als Ziel der Umsetzung sollten ein Compiler (Software) sowie die Beschreibung der Hardwarekomponenten in einer Hardwarebeschreibungssprache entstehen Hierzu wurden zwei Teilgruppen gebildet. Während sich die Softwaregruppe mit den Grundlagen der Kompilierung in Maschinencode beschäftigt hat, hat die andere Gruppe die Grundlagen für die Rechnerarchitektur ausgearbeitet. Am Anfang des Projektes stand die gemeinsame Diskussion zum allgemeinen Konzept, d.h. erste Ideen zur Umsetzung in Hard- und Software. Die Entwicklung war experimentell und prototypisch, d.h. es wurden viele Ideen entwickelt und verworfen. Beide Gruppen haben sich durch produktive Diskussionen in der ganzen Gruppe gegenseitig stark beeinflusst und somit direkten Anteil an der Umsetzung des gesamten Projektes und der Umsetzung der jeweils anderen Gruppe. Schlussendlich wurde sich darauf geeinigt, einen eigenen Weg einzuschlagen und sich lediglich an bereits vorhandenen Ideen und Umsetzungen zu orientieren anstatt ein Design zu kopieren. Dies hat insbesondere den Effekt gehabt, dass wir kaum auf bewährte Konzepte der Rechnerarchitektur zurückgreifen konnten. Ein Beispiel hierfür ist die nicht-lineare Abarbeitung der Programme. Am Ende steht ein rudimentärer Hardwareinterpreter, der die Eigenschaften von Lisp auf der Ebene der Rechnerarchitektur implementiert.

Der Projektbericht gestaltet sich folgendermaßen. Zunächst wird in Kapitel 2 ein Überblick über die Programmiersprache Lisp und die von uns gewählte Untermenge gegeben. Anschließend wird in Kapitel 3 die von uns gewählte Rechnerarchitektur mit den wichtigsten Komponenten beschrieben. Hierfür wird die Organisation des Speichers und des einzelnen Speicherwortes im Detail beschrieben. Anschließend wird der Befehlssatz vorgestellt. Kapitel 4 umfasst den gesamten

Softwareentwicklungsprozess. Zunächst werden die Phasen des Prozesses beschrieben und die gesetzten Ziele reflektiert. Schließlich werden die Funktionsweise des Compilers und weiterer Werkzeuge erörtert. Zum Abschluss dieses Kapitels werden Schwierigkeiten während der Umsetzung und Ansätze für eine Weiterentwicklung aufgezeigt.

2 Lisp

Die Programmiersprache Lisp ist eine der ältesten noch in Gebrauch befindlichen Sprachen, die auch heute noch in verschiedenen Varianten vorkommt und oft als Lehrsprache für funktionale Programmierung verwendet wird. Moderne Lispvarianten, z.B Clojure [1], Scheme [2] oder Racket [3] unterscheiden sich dabei zum Teil erheblich in ihrer Syntax und Semantik. Einen Überblick über die Geschichte und die Konzepte von Lisp findet sich in „The Roots of Lisp“ [2]. Der Grund ist, dass Lisp keinen Standard besitzt, sondern lediglich als Famile von Implementierungen einigen Gemeinsamkeiten unter allen Derivaten wahrgenommen wird. Als Urheber von Lisp gilt John McCarthy, der in seinem Artikel „Recursive Functions of Symbolic Expressions and Their Computation By Machine, Part I“ im Jahre 1960 die erste Version von Lisp definiert hat [4]. Lisp steht für List processing, also die Verarbeitung von Listen. Listen sind die grundlegende Datenstruktur und jedes Programm in Lisp besteht aus einer oder mehreren Listen. Somit sind Daten und Programm auf konzeptueller Ebene das selbe. Verwendet wurde Lisp anfänglich vorallem zur Implementierung von Programmen für künstliche Intelligenz. Dies zeigt auch der Ursprung: das AI Lab des Massachussets Institute of Technology. Jede Lisp Variante hat dabei das Konzept der *S-Expression* als Grundlage sowohl von Syntax als auch Semantik. Dabei steht das S für symbolisch, da symbolische Auswertung wie sie in der künstlichen Intelligenz benötigt wird, die Hauptmotivation zur Entwicklung von Lisp war.

Jede S-Expression besteht dabei aus folgenden syntaktischen Elementen [4]: Spezialelemente, das sind "(", ")"und "·". Offene Klammern beginnen eine S-Expression, geschlossene Klammern beenden sie. Der Punkt trennt den sogenannten Listenkopf vom Rest. Da S-Expressions rekursiv definiert sind, sind beliebige Verschachtelungen möglich. Entweder enthalten sie einen weiteren symbolischen Ausdruck oder ein Terminal, d.h. eine Zeichenkette. Um anzuzeigen, dass es kein weiteres Element hinter dem aktuellen gibt, wurde das Atom *Nil* gewählt. Die Liste

$$(m_1, m_2, \ldots, m_n) \tag{2.1}$$

[1] `https://clojure.org/`
[2] `https://www.gnu.org/software/mit-scheme/`
[3] `https://racket-lang.org/`

```
1          (lambda (x)
2              (* x x))
```

Abbildung 2.1: Anonyme Funktion zur Quadrierung des Symbols x

```
1          (if
2              (< 3 x)
3              (3)
4              (x))
```

Abbildung 2.2: Beispiel für eine If Anweisung

wird dabei nach McCarthy [4] als rekursive S-Expression

$$(m_1 \cdot (m_2(\ldots(m_n \cdot NIL)\ldots))) \tag{2.2}$$

dargestellt.

Um praktische Programme, d.h. Programme, die ein Ergebnis liefern, schreiben zu können, sind symbolische Variablen jedoch nicht genug. Daher beinhalten (die meisten) Lisp Implementierungen neben den primitiven Datentypen Liste und Atom zusätzlich noch folgende Elemente:

- *Nil*, ähnlich Null in C oder Java
- $\#T$ und $\#F$, Boolsches Wahr und Falsch
- Zahlen (z.B Ganz- oder Gleitkommazahlen)
- Zeichenketten
- Primitive Funktionen, z.B Artithmetik oder Vergleichsoperationen

Dazu kommen sog. *Special forms*, die einer Sonderbehandlung bedürfen. Dazu zählen anonyme Funktionen und Varianten der If-Anweisung. Da es sich um eine funktionale Programmiersprache handelt, können Funktionen als Parameter an andere Funktionen übergeben werden.

Ein Lambda, welches das Symbol x, welches als Parameter übergeben wird, quadiert findet sich in Abb. 2.1 während Abb. 2.2 eine mögliche Variante der If-Anweisung zeigt. Es fällt auf, dass in jeder S-Expression die Funktion bzw. Primitive stets am Anfang der Liste steht. Dies wird auch als polnische Notation bezeichnet. Der Ausdruck (¡ 3 4) ist dann äquivalent zu 3 ¡ 4.

Einige Grundfunktionen eines Lisps sind im Folgenden benannt und aus [4] entnommen. Neben arithmetischen und logischen Operationen (u.a. +, -, /, *, ¡, ¿

und =) existieren Funktionen zur Listenmanipulation. *CAR[x]* gibt den Listenkopf von x, *CDR[x]* den Rest zurück. *APPLY[f; args]* wendet die funktion f mit der Umgebung, d.h. der Variablenbelegung args (als Liste gegeben) an. *QUOTE[x]* verhindert die Auswertung der folgenden S-Expression x. *CONS[x;y]* erstellt eine neue S-Expression mit x als Kopf und y als Rest. Schließlich definiert McCarthy noch *ATOM[x]*, das wahr zurückgibt, wenn x ein Atom ist, sonst falsch. Eine ausführliche Behandlung findet sich im Artikel von McCarthy.

Lispvarianten haben oft einen sog. Interpreter, ein Programm, das interaktiv eingegebenen Lisp Code auswerten kann und über eine virtuelle Maschine (eine Abstraktion der Hardware) verfügt, werden also nicht kompiliert.

3 Hardware

Um eine Speicherstruktur zu erhalten, die möglichst nah an Lisp ist, haben wir den Speicher in Cons Zellen organisiert, inspiriert von der CADR Architektur [3]. Hierfür versehen wir jedes Wort im Speicher mit einem Tag, das anzeigt, ob es sich um eine Cons-Zelle oder ein Datum, also einen numerischen Wert handelt. Eine detailierte Erläuterung der Tagbedeutung findet sich in Abschnitt 3.3. Eine Übersicht der Speicherorganisaton wird in Abschnitt 3.1 gegeben. Zentrale Komponenten werden in Abschnitt 3.2 vorgestellt.

3.1 Speicherorganisation

Der Aufbau eines Speicherworts ist in Abb. 3.1 und Abb. 3.2 dargestellt. Eine Cons Zelle ist aus einem 2 Bit Tag, der anzeigt, dass es sich um eine Cons Zelle handelt und zwei 15 Bit Pointern auf den Listenkopf respektive Listenrest in absoluten Speicheradressen. Ein Listenelement besteht aus einem 2 Bit Tag, der anzeigt, dass es sich um ein Datum handelt sowie 30 Bit Daten.

Der Vorteil einer solchen Organisation besteht darin, dass sie sehr nah an Lisp ist, Listen leicht modifizierbar sind, sowohl in ihrer Wertebelegung als auch beim Ändern der Listenlänge (z.B zusammenfügen oder entfernen von Elementen) und kaum Speichermanagement benötigt wird. Dies ermöglich u.a. eine einfache Implementierung von Garbage Collection.

Aus diesem Entwurf ergeben sich allerdings auch Nachteile. Die Auswertung von verschachtelten Listen ist, im Gegensatz zum Aufbau solcher, komplex. Zudem entsteht ein erheblicher Speicherverbrauch und Performancenachteil durch die vielen Pointer, die gespeichert und später traversiert werden müssen.

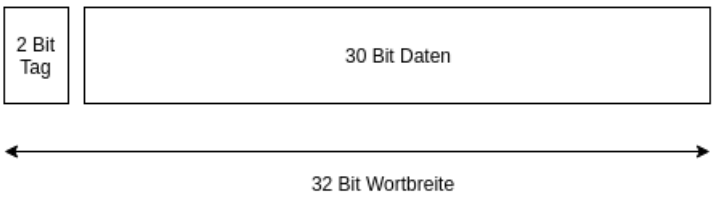


Abbildung 3.1: Ein Listenelement auf Wortebene

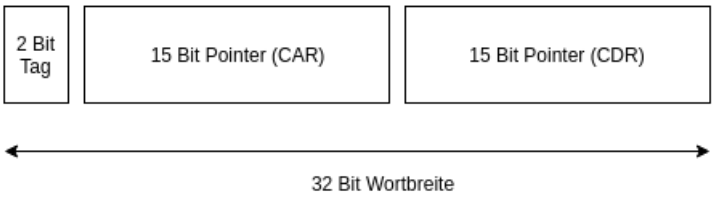


Abbildung 3.2: Eine Cons Zelle auf Wortebene

Daher haben wir uns zur Reduktion dieser Komplexität für eine Stackmaschine entschieden. Hierfür haben wir uns nun an dem Ansatz der SECD-Maschine orientiert wie in [1] beschrieben und von diesem das Konzept mehrerer Stacks übernommen. Weitere Aspekte wie den Environment Stack zur Speicherung von Variablenbelegungen werden stattdessen vom Compiler übernommen, indem besondere Speicherbereiche zugewiesen werden.

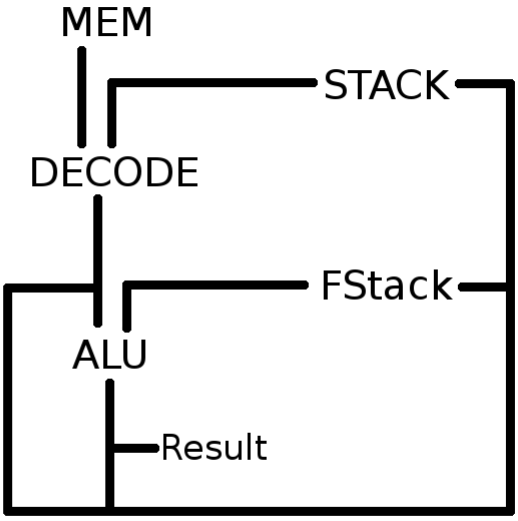
Um eine einfache Auswertung von Listen zu gewährleisten und die speicherorganisatorischen Vorteile auszunutzen, haben wir die ALU erweitert. Diese kann die selbe arithmetische Operation auf bis zu 4 Eingaben gleichzeitig durchführen. Dies entspricht einer SIMD (Single Instruction Multiple Data) Vektoreinheit. Die ALU übernimmt eine Liste als Eingabe und reduziert diese auf ein Ergebnis.

Zu diesem Zeitpunkt haben wir den Code im Speicher nach und nach durch sein Ergebnis substituiert, indem wir jeden Teilausdruck durch sein Teilergebnis ersetzten. Diese Methode verhindert allerdings das Definieren von Funktionen, da diese bei der Berechnung überschrieben werden. Eine Möglichkeit das zu verhindern wäre gewesen, die Funktionsdefinition vor jeder Ausführung zu kopieren. Da dies zu aufwändig wäre und insbesondere noch mehr Speicher verschwendet, reduzieren wir stattdessen die Teilausdrücke nicht mehr im Speicher, sondern auf dem Stack. Das stellt uns zufrieden, da wir dies auch mit baumrekursiven Funktionen erfolgreich testen konnten.

3.2 Aufbau der Maschine

Wie Abbildung 3.3 zu entnehmen ist, besteht unsere Rechnerarchitektur aus mehreren Komponenten. Sämtliche Komponenten sind im Quelltext detaillierter beschrieben und im Folgenden wird ein grober Überblick über die einzelnen Komponenten gegeben. *Mem* dient als Speicher und beinhaltet das Programm, welches zur Laufzeit neue Teile an die Decodeeinheit sendet. *Decode* fasst sämtliche nicht eingezeichneten Steuerleitungen zusammen und dient als zentrale Steuereinheit. Die Einheit beinhaltet das Instruktion-Fetch-Register. Dieses kann entweder durch Instruktionen vom Stack oder neuen Instruktionen vom Speicher befüllt werden.

Abbildung 3.3: Komponenten der Maschine



Diese Einheit ist von zentraler Bedeutung beim Ansteuern der *ALU*. Ergebnisse der *ALU* werden durch diese Einheit an die richtige Stelle geleitet und der Stack bekommt Daten durch diese Einheit zugeteilt. Sie dient ausschließlich dem Kontrollfluss, um bspw. Funktionsrücksprünge ohne die *ALU* auszuführen. Die *ALU* verwertet den Opcode (das Befehlswort) und berechnet abhängig von diesem das Ergebnis der aktuellen Instruktion. Die *ALU* ist ausschließlich für arithmetische oder logische Berechnungen und Auswertungen zuständig und beeinflusst den Kontrollfluss nicht. Die *ALU* gibt am Ende ein Ergebnis aus, welches sich aus einem Tag und einem Datenwort oder ganzem Frame (siehe unten). Falls nötig können an entsprechender Stelle die Platzhalter die Werte vom Fstack genutzt werden. *Result* ist ein Register welches das zuletzt berechnete Ergebnis enthält und dieses als eine Ausgabenschnittstelle bereitstellt. *Stack* ist der Speicherort für teilweise berechnete Ergebnisse und dient auch dem Kontrollfluss indem der Stack z.B. die Daten für die *ALU* als Eingang bereitstellt. *Fstack* ist der Funktionsaufrufstack, dieser enthält Parameter für Funktionsaufrufe. Die Parameter sind als sog. Environments angelegt und sind rechnerintern als Cons Listen dargestellt. Der Fstack beinhaltet allerdings keine Informationen vom normalen Stack. wenn beim Kontrollfluss festgestellt wird, dass ein Funktionsaufruf oder ein Rücksprung erfolgt, wird nur das aktuelle Environment (die Belegung der Parameter) auf dem Fstack festgehalten.

Tag	Bedeutung
00	*Nil* Liste (Keine Daten enthalten)
01	Datenwort
10	Pointer zu anderer Speicheradresse
11	*Error*

Abbildung 3.4: 2 Bit Tags pro Speicherwort und ihre Bedeutung

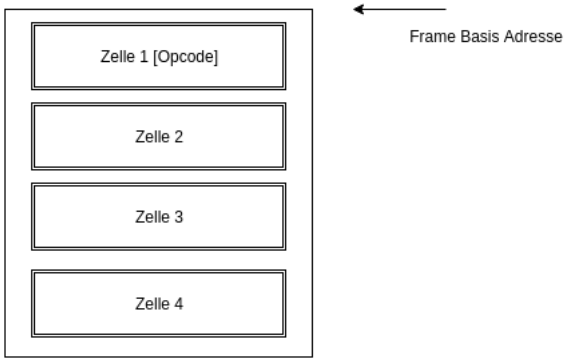


Abbildung 3.5: Ein Frame fasst 4 Zellen zu einer logischen Einheit zusammen

3.3 Tags

Um der speziellen Struktur von Lisp Operationen Rechnung zu tragen, werden analog zu den früheren Lisp Maschinen sogenannte Tags eingeführt. Diese dienen dem Kennzeichnen des zugehörigen Speicherworts. Hierfür werden die obersten zwei Bits jedes Speicherworts benutzt. Somit bleiben für Adressen oder Werte 30 Bit übrig. Eine Übersicht findet sich in Abbildung 3.4. Der *Nil* Tag ist besonders für Algorithmen, die den Speicher nicht-linear traversieren (ähnlich einer Baumsuche) gedacht sowie als Kennzeichnung, dass das entsprechende Speicherwort leer ist. Die Traversierung mittels Tags wird als Hardwaretracing bezeichnet. Somit lässt sich erkennen, wann nicht mehr weiter gesucht werden soll. Als Datenwort können Werte im Bereich von 0 bis 2^{30} benutzt werden. Ein Pointer zeigt auf eine andere physikalische Adresse. Erneut ist dies bedingt durch die nicht-lineare Struktur von Lisp. Zuletzt existiert ein Tag, das Fehler anzeigt. Dies kann zum Debugging oder für eine eventuelle Fehlerbehandlung verwendet werden. Außerdem wird der Error Tag für spezielle reservierte Konstanten benutzt.

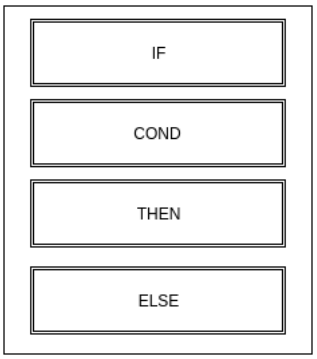


Abbildung 3.6: Ein If-Frame mit Opcode und Pointern auf Listen für Bedingung und Aktionen bei wahrheitsgemäßer und nicht-wahrheitsgemäßer Auswertung

3.4 Frames

Neben den Speicherworten (Zellen) verfügt der Mikroprozessor auf Speicherebene zusätzlich über das Konzept der Frames. Der Aufbau ist in Abb. 3.5 dargestellt. Ein Frame fasst dabei 4 Zellen zu einer logischen Einheit zusammen. Dabei ist die Adresse des ersten Frames die Basisadresse und die folgenden 3 Frames können durch einen Offset adressiert werden. Die erste Zelle ist dabei üblicherweise eine Anweisung (z.B ADD) während die folgenden Zellen Argumente darstellen. Eine kontroverse Entwurfsentscheidung war es, die Framegröße statisch zu definieren. Diese Entscheidung wird in Abschnitt 4.4 mit Bezug auf den Compiler erneut beleuchtet. Eine Alternative wäre es gewesen, die Listen variabel lang zu machen und auszunutzen, dass meistens flache Listen vorliegen und selten(er) verschachtelte. Hier wurde die Größe bewusst so gewählt, um die If-Anweisung möglich zu machen und einfach umzusetzen. In Abb. 3.6 ist die Umsetzung zu sehen. Dabei ist der erste Frame die Anweisung, während die folgenden jeweils einen Pointer auf die Bedingung (COND), die THEN Liste, die die Aktion bei einer wahrheitsgemäßen Auswertung der Bedingung und die ELSE Liste, d.h. das was bei Auswertung zu Nil getan werden soll, besitzen. Zusätzlich ist die Größe an die verfügbaren Eingänge der ALU angepasst.
Außerdem ermöglicht eine feste Framegröße eine bessere Abstraktion im physikalischen Speicher nur noch von Frames als Datenworte (dann in der Breite von $4 * 32 = 128$ Bit) zu sprechen.

Als Konsequenz ergibt sich, dass lediglich das erste Element jedes Frames adressiert werden kann. Inkrementiert man eine Adresse, so ergibt sich als Resultat die Adresse des nächsten Frames.

3.5 Stack

Der Stack ist eine nach unten wachsende LIFO-Datenstruktur mit einer Wortbreite von 130 Bit pro Stackelement. Da immer ein ganzer Frame auf dem Stack gespeichert wird und zusätzlich 2 Bit für den Offset innerhalb des Frames benötigt wird. Er verfügt über zwei Operationen: Push und Pop. Nach der Push Operation wird der Stackpointer (SP) dekrementiert und der Bitvektor, der am Eingang angelegt ist, wird auf das aktuelle Stackelement gelegt. Beim Ausführen der Operation Pop wird das Element, auf das der SP zeigt auf den Ausgang gelegt und der Stackpointer inkrementiert. Nach Pop wird das Element nicht physisch vom Stack entfernt, sondern der SP wird verändert. Das Element verbleibt demnach auf dem Stack.

Sollte der Stackpointer auf das erste Element (d.h. die oberste Adresse) zeigen, so wird der Ausgang *Stack_Empty* aktiviert. Dieser zeigt an, dass der Stack leer ist und nichts vom Stack entfernt werden kann. Sollte der Stackpointer auf das letzte Element (d.h. die unterste Adresse) zeigen, so wird der Ausgang *Stack_Full* aktiviert. Somit kann nichts mehr auf den Stack gelegt werden. Der zweite Stack, der Funktionsstack ist inspiriert durch den Dump der SECD Maschine. Während der Datenstack für die Lagerung Berechnungsergebnissen zuständig ist, erweitert der Funktionsstack (FStack) die Fähigkeiten des Mikroprozessors zur Auswertung von Funktionen. Im Folgenden wird der Datenstack lediglich Stack genannt.

Der Stack verfügt über 4 Eingänge. Diese sind:

Clk
: Die Clock, die den Takt vorgibt.

Enable
: Zeigt an, ob der Stack benutzt werden kann (Wert '1') oder deaktiviert ist (Wert '0').

Data_In
: Ein 32 Bit Vektor, zeigt an, welches Element als nächstes auf den Stack gelegt werden soll.

Push_Or_Pop
: Zeigt an, ob PUSH oder POP ausgeführt werden soll. Wert '1', falls PUSH, Wert '0' falls POP.

Enable_Inplace
: Zeigt an, ob der Inplace Modus genutzt wird.

sowie über 3 Ausgänge:

Konstante bzw. Größe	Bedeutung
0x0000	*Null Vector*
0x0000	Boolesches Falsch
0x0001	Boolesches Wahr

Abbildung 3.7: Weitere Konstanten und ihre Bedeutung

Data_Out
: Ein 32 Bit Vektor. Dieser zeigt an, welches Element als nächstes vom Stack entfernt werden soll.

Stack_Full
: Zeigt an, ob der Stack voll ist (Wert: '1'), d.h. falls der Stackpointer auf die unterste Adresse zeigt. Elemente können nicht mehr auf den Stack gelegt werden.

Stack_Empty
: Zeigt an, ob der Stack voll ist (Wert: '1'), d.h. falls der Stackpointer auf die oberste Adresse zeigt. Elemente können nicht mehr vom Stack entfernt werden.

Außerdem stehen 4 Modi zur Verfügung. Jeder Modus bestimmt, wie der Stack operiert und ob bzw welche Daten entfernt bzw. hinzugefügt werden. Grunsätzlich wird zwischen *Normal* und *Inplace* Operationen entschieden. Erster Modus versetzt den Stack entweder in den Push oder Pop Modus, die dem Verhalten der Datenstruktur entspricht. Daneben existiert der Inplace Modus. Dieser hat zwei Ausprägungen, *Inplace 1* und *Inplace 2*. Der erste ersetzt das oberste Element des Stacks, was der Ausführung von Pop, gefolgt von Push entspricht. Der zweite ersetzt das zweite Element des Stacks, den Next of Stack. Dabei wird das oberste Element gelöscht (entspricht der Ausführung von Pop, Pop und Push). Diese Inplace-Operationen sind für eine Klassische 0-Adress Machine von Vorteil. In unserem Konzept finden sie jedoch keine Anwendung.

3.6 Sonstige Konstanten

Die Lisp Maschine verfügt neben den bereits vorgestellten Befehlen und organisatorischen Details auf Wortebene über weitere Konventionen. Diese sind Abb. 3.7 zu entnehmen. Das Nullwort beschreibt dabei den leeren Vektor, also *Nil* in Lisp. Die boolschen Werte wahr und falsch werden durch 1 bzw. 0 dargestellt. Allerdings werden alle Werte größer 0 dennoch als Wahr behandelt. Dies ist konform mit Lisp

Bitmuster	Bedeutung
001111	Fehlerhafter Eingabeparameter (z.B ein leerer Frame)
011111	Undefinierter Opcode
101111	Fehler in der Behandlung eines *If* Ausdrucks
111111	Die Operation benötigt 2 Operanden

Abbildung 3.8: 5 Bit Fehlercodes und ihre Bedeutung

Bitmuster	Bedeutung
0001	X1 Der erste Funktionsparameter
0010	X2 Der erste Funktionsparameter
0011	X3 Der erste Funktionsparameter

Abbildung 3.9: Spezielle Konstanten in den 4 unteren Bits eines Fehlercodes

Implementierungen, da es dort (meistens) keinen entsprechenden Wert für falsch gibt (z.B in Scheme). Falls es während der Auswertung eines Programms zu einem Fehler kommt, so ist das Ergebnis der entsprechenden Operation ein Fehlercode. Man beachte, dass bei allen Fehlercodes die letzten 4 Bit gesetzt sind. Sind die letzten 4 Bits nicht gesetzt, so handelt es sich nicht um einen Fehlercode, sondern um eine spezielle Konstante. Diese werden zum Aufruf von Funktionen verwendet und repräsentieren die Parameter. Die entsprechenden Bitmuster können Abb. 3.9 entnommen werden.

3.7 Befehlsauswertung

Der Ablauf des Programs wird maßgeblich vom Stack bestimmt. Dabei ist erwähnenswert, dass es keinen Programmcounter gibt, da dieser zur Abarbeitung von Befehlen nicht benötigt wird. Dies unterscheidet die Lisp Maschine von etablierten Architekturen, bei denen die nächste Instruktion durch den Programmcounter angezeigt wird. Stattdessen kommt das nächste Befehlswort entweder vom Stack oder die Adresse ist ein Pointer. Als Beispiel diene dabei der Ausdruck, der in Abbildung 3.10 zu sehen ist.

```
1          (ADD 1
2              (SUB 2 1))
```

Abbildung 3.10: Ein einfacher arithmetischer Lisp Ausdruck

```
1        (defun faculty (x)
2            (if (= x 1)
3            1
4            (* x (faculty
5                    (- x 1 )))
```

Abbildung 3.11: Fakultätsfunktion in Lisp.

Ein Befehlswort (d.h. eine Liste mit 4 Elementen) wird wie folgt abgearbeitet

1. Das Wort ist im Instruction-Fetch-Register, es wird zunchst geprüft, ob es bereits vollständig berechenbar ist, oder ob es noch weitere Teilausdrücke gibt, die erst ausgewertet werden müssen

2. Zunächst muss (- 2 1) Berechnet werden, da der Rest der Liste kein Datenwort ist Falls der Rest noch nicht berechnet ist wird der gesamte Ausdruck auf den Stack gelegt, dabei wird der Offset des Resultats mit gespeichert

3. Der Offset ist die Stelle, an der der sich der nicht berechnete Teilausdruck befindet

4. Die Nächste Instruktion ist dann der noch nicht vollständig berechnete Teilausdruck. Dieser Wird vom Speicher an der gegebenen Adresse geladen, hier also (- 2 1)

5. Im Code steht (+ 1 "Pointer auf (- 2 1)"). Daran lässt sich zum Einen erkennen, dass (- 2 1) noch nicht berechnet ist [1] und zum Anderen ist dies der Zeiger zur nächsten Anweisung

6. Falls der Ausrduck bereits volständig berechnet ist, wird er nun zur Berechnung an die ALU gegeben

7. Das Result der ALU ist dann der nächste Befehl. Dieser vom Stack geladen und ist der nächste Befehl, dabei wird beim gegebenen Offset der Pointer durch das berechnete Ergebnis ersetzt. Das heißt insbesondere, dass der Tag ein Datenwort anzeigt. Im Beispiel wird (+ 1 "Pointer auf (- 2 1)") also zu (+ 1 1)

Funktionsaufrufe haben das selbe Prinzip, jedoch einige Eigenheiten. Bei Funktionsaufrufen ist der Kontrollfluss besonders wichtig. Daher illustrieren wir den

[1] wenn (- 2 1) schon ausgerechnet ist wäre dort ja bereits der Wert 1 als Datenzelle und nicht der Pointer zu finden

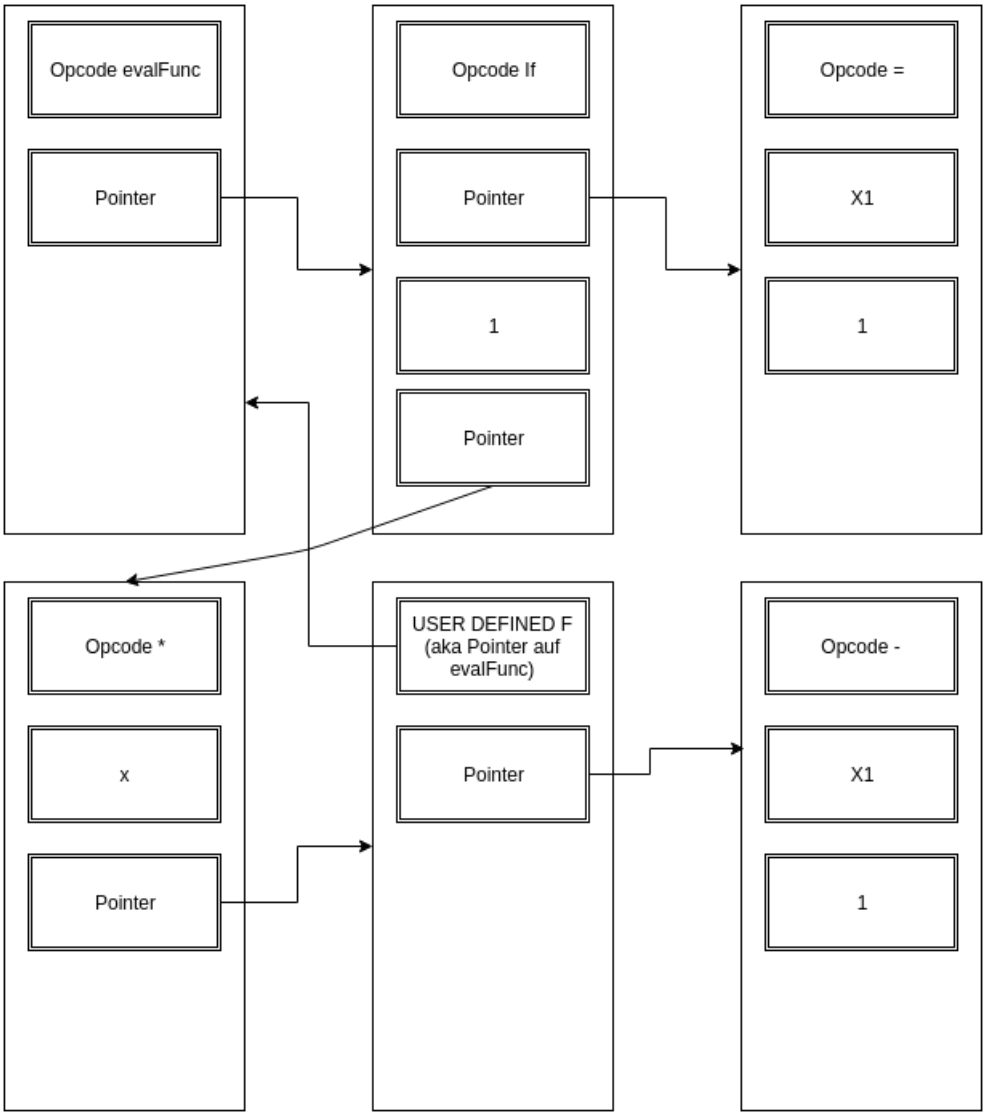


Abbildung 3.12: Organisation der Funktion *faculty* im Speicher

Ablauf anhand des Beispielprogramms aus Abb. 3.11. $faculty(x)$ berechnet offensichtlich die Fakultät von x. Im Speicher ist die Funktion wie in Abb. 3.12 zu sehen organisiert. Dies ist lediglich eine mögliche Repräsentation. Dabei wird auf Abbildung von leeren Zellen verzichtet, jedes Frame ist dennoch physikalisch 4 Zellen groß. Pointer sind Zeiger auf das erste Element des verwiesenen Frames.

Wir Berechnen z.B. f(2). Dabei lassen wir die enzelnen Schritte bei der Auswertung einfacher Arithmetik aus, da diese bereits im vorherigen Beispiel illustriert wurden. Zunächst wird der Ausdruck P aus dem Speicher geladen. Es wird festgestellt, dass es sich bei $faculty$ um eine benutzerdefinierte Funktion handelt und nicht um einen Opcode, daher wird die Funktion im nächsten Schritt aufgerufen. Der Ausdruck $faculty(2)$ wird auf dem Funktionsparameter Stack abgelegt. Der nächste Ausdruck ist durch die Adresse von $faculty$ gegeben ($faculty$ selber ist der Pointer). Bei der Ausführung von $evalFunc$ wird festgestellt, dass erst dem Pointer auf den eigentlichen Funktionsrumpf gefolgt werden muss [2] Beim If wird festgestellt, das der als Bedingung angegebene Ausdruck (der an 2. stelle direkt nach dem If) noch nicht ausgewertet wurde. Die ALU wertet nun die Bedingung ganz normal aus. Lediglich der Platzhalter $X1$ wird vor der Auswertung durch den entsprechenden Wert vom FStack ersetzt. Nun kann das If ausgewertet werden, da die Bedingung zu False evaluiert wurde, ist der wert des If Ausdrucks der Wert des an letzter Stelle angegebenen Ausdrucks. Nun wird der Ausdruck *(* X1 (f (- X1 1))* ausgewertet. Wir überspringen den rekursiven Aufruf, da er genauso abläuft, wie der gerade gezeigte. Der nun berechnete Wert ist 2, daher wird nun $evalFunc(2)$ geladen.

Anschließend wird der $evalFunc$ Ausdruck ausgewertet. Da der Stack im nach Auswertung leer ist, wird hier terminiert. Man Beachte, dass auf FStack Pop ausgeführt wird, da man mit dem Auswerten des $evalFunc$ die Funktion vollständig ausgewertet hat und nun in die aufrufende Funktion zurückspringt.

Funktionsaufrufe haben das selbe Prinzip, jedoch einige Eigenheiten. Bei Funktionsaufrufen ist der Kontrollfluss besonders wichtig. Daher illustrieren wir den Ablauf anhand des Beispielprogramms aus Abb. 3.11. $faculty(x)$ berechnet offensichtlich die Fakultät von x. Im Speicher ist die Funktion wie in Abb. 3.12 zu sehen organisiert. Dies ist lediglich eine mögliche Repräsentation. Dabei wird auf Abbildung von leeren Zellen verzichtet, jedes Frame ist dennoch physikalisch 4 Zellen groß. Pointer sind Zeiger auf das erste Element des verwiesenen Frames.

Wir Berechnen z.B. f(2). Dabei lassen wir die enzelnen Schritte bei der Auswertung einfacher Arithmetik aus, da diese bereits im vorherigen Beispiel illustriert wurden. Zunächst wird der Ausdruck P aus dem Speicher geladen. Es wird festgestellt, dass es sich bei $faculty$ um eine benutzerdefinierte Funktion handelt und

[2] In die Abbildungen hat sich leider ein Fehler eingeschlichen: Auf dem Stack muss 2 (evalFunc P (if...)) liegen, da der Ausdruck an Stelle 2 noch nicht ausgewertet wurde

nicht um einen Opcode, daher wird die Funktion im nächsten Schritt aufgerufen. Der Ausdruck $faculty(2)$ wird auf dem Funktionsparameter Stack abgelegt. Der nächste Ausdruck ist durch die Adresse von $faculty$ gegeben ($faculty$ selber ist der Pointer). Bei der Ausführung von $evalFunc$ wird festgestellt, dass erst dem Pointer auf den eigentlichen Funktionsrumpf gefolgt werden muss [3] Beim If wird festgestellt, das der als Bedingung angegebene Ausdruck (der an 2. stelle direkt nach dem If) noch nicht ausgewertet wurde. Die ALU wertet nun die Bedingung ganz normal aus. Lediglich der Platzhalter $X1$ wird vor der Auswertung durch den entsprechenden Wert vom FStack ersetzt. Nun kann das If ausgewertet werden, da die Bedingung zu False evaluiert wurde, ist der wert des If Ausdrucks der Wert des an letzter Stelle angegebenen Ausdrucks. Nun wird der Ausdruck *(* X1 (f (- X1 1))* ausgewertet. Wir überspringen den rekursiven Aufruf, da er genauso abläuft, wie der gerade gezeigte. Der nun berechnete Wert ist 2, daher wird nun $evalFunc(2)$ geladen.

Anschließend wird der $evalFunc$ Ausdruck ausgewertet. Da der Stack im nach Auswertung leer ist, wird hier terminiert. Man Beachte, dass auf FStack Pop ausgeführt wird, da man mit dem Auswerten des $evalFunc$ die Funktion vollständig ausgewertet hat und nun in die aufrufende Funktion zurückspringt.

Detailierte Erklärungen zur Auswertung mit grafischer Illustration finden sich im Anhang.

3.8 Befehlssatz

Jeder Befehl besteht aus 4 Zellen, das erste ist der Opcode mit Datentag, die anderen 3 Operanden unterscheiden sich von Befehl zu Befehl. Im Allgemeinen werden die Operanden zunächst ausgewertet, wenn es sich um Pointer handelt. Das bedeutet, der Ausdruck auf den der Pointer zeigt wird ausgewertet und das Ergebnis wird als Operand an Stelle des Pointers verwendet. Die Befehle werden dabei im folgenden zu Gruppen zusammengefasst und jede einzeln erläutert. Zusätzlich bietet Abb. 3.13 einen Überblick über alle verfügbaren Befehle und den zugehörigen Opcode.

3.8.1 Arithmetik

Verfügbar: ADD, SUB, MUL, DIV
Arithmetische Befehle arbeiten auf zwei oder drei Operanden und folgen dem Format Operand 1 $\star$ Operand 2 [$\star$ Operand 3], wobei $\star$ eine arithmetische Operation

[3] In die Abbildungen hat sich leider ein Fehler eingeschlichen: Auf dem Stack muss 2 (evalFunc P (if...)) liegen, da der Ausdruck an Stelle 2 noch nicht ausgewertet wurde

Opcode (Hexadezimal)	Mnemonic
0x0	Noop bzw. Eval
0x1	If
0x2	Negate
0x3	Add
0x4	Sub
0x5	Mul
0x6	Div
0x7	And
0x8	Or
0x9	XOR
0xA	NAND
0xB	Shift Right
0xC	Shift Left
0xD	Rotate Right
0xE	Rotate Left
0xF	Equal
0x11	Not Equal
0x12	Less Than
0x13	Greater Than
0x14	Logical Shift Right
0x15	Logical Shift Left
0x1FFFFFFF	Evaluate Function

Abbildung 3.13: Überblick über das Instruction Set

ist. Falls nur zwei Operanden addiert werden sollen, so muss der dritte Operand das Nullwort sein.

3.8.2 Logische Operationen

Verfügbar: AND, OR, XOR, NAND, NEG
Logische Befehle verhalten sich ähnlich wie arithmetische. Für sie gelten die selben Aufrufkonventionen und Einschränkungen. Eine Ausnahme bildet der Befehl NEG, welcher lediglich einen Parameter entgegebennimmt und ihn negiert.

3.8.3 Bitoperationen

Verfügbar: ShiftL, ShiftR, LogicalShiftR
Bitoperationen arbeiten auf den Bits einer Zelle und können diese manipulieren. Dabei gilt, dass der erste und zweite Operand die Eingaben sind, es wird also der erste Operand um den zweiten verschoben während der dritte ein Nullwort sein muss.

3.8.4 Vergleichsoperationen

Verfügbar: LessThan, GreaterThan, Equal, NotEqual
Vergleichsoperationen umfassen die mathematischen Relationsoperationen $\leq$, $\geq$, $=$, $\neq$, sind allerdings im Gegensatz zu anderen Befehlen nicht als Listenoperationen, sondern als binäre Operatoren (d.h. es es werden stets zwei Operanden verglichen). Es wird erwartet, dass der dritte Operand ein Nullwort ist. Alle Resultate ungleich dem Nullwort werden als wahr interpretiert. Als Konvention gilt: sollte lediglich Operand 1 Daten enthalten, Operand 2 jedoch nicht, sowerden diese Daten als Ergebnis zurückgegeben, da ein Vergleich mit dem Nullwort stets wahr zurückgibt.

3.8.5 EvalFunc

Evalfunc dient zur Definition von eigenen bzw. benutzerdefinierten Funktionen. Eine Funktionsdefinition muss immer mit EvalFunc beginnen. Der Aufbau des EvalFunc Befehls ist folgendermaßen:

- Operand1: Ein Pointer auf den Funktionsrumpf
- Operand2: Dieser Operand muss ein Nullwort sein
- Operand3: Dieser Operand muss ein Nullwort sein

Funktionsparameter	Bitmuster (Hexadezimal)
X_1	0xC0000001
X_2	0xC0000002
X_3	0xC0000003

Abbildung 3.14: Bitmuster für die Funktionsparameter X_1, X_2 und X_3

Innerhalb des Funktionsrumpfes können die 3 Argumente der Funktion mit ihren entsprechenden Platzhaltern (X_1, X_2, X_3) aufgerufen werden. Die Bitmuster zur Erkennung der Parameter sind aus Abb. 3.14 ersichtlich.

Der Aufruf einer Funktion geschieht folgendermaßen: als Opcode muss der Pointer auf den evalFunc Ausdruck sein, die anderen 3 Operanden für diesen Befehl sind dann die Parameter, die an die Funktion übergeben werden und innerhalb der Funktion mit den entsprechenden Platzhaltern für X_1, X_2 und X_3 genutzt werden können.
Dabei ist zu Beachten, das das erste Bit gesetzt sein muss, um anzuzeigen, dass es sich um einen Funktionsaufruf und nicht einen normalen Opcode handelt. Diese 1 ist nicht teil der adresse, sodass für Funktionsadressen nur noch 29 Bit zur Verfügung stehen Außerdem ist zu zu beachten, dass bei einem Funktionsaufruf nicht überprüft wird, ob der Pointer wirklich auf einen evalFunc Ausdruck zeigt. Dies muss durch den Compiler oder Programmierer geschehehen. Ebenso wird beim Erreichen eines EvalFunc nicht validiert, ob tatsächtlich mit einem Funktionsaufruf oder eines anderen Sprunges diese Adresse erreicht wurde. Die Nichtbeachtung dieser Konvention führt zu einem Fehler. Die Nutzung von EvalFunc innerhalb von Funktionen oder bzw. nicht als erstes Element eines Frames führt zu undefiniertem Verhalten und Fehlern. Für solche Situationen sollte Eval benutzt werden.

EvalFunc hat das gleiche Verhalten wie Eval und Noop: Mit dem Ergebnis der Funktion wird keine weitere Operation außer einem Funktionsrücksprung durchgeführt.

3.8.6 If

Ein If-Ausdruck hat stets die Form (IF THEN ELSE), wobei jedes Wort ein Element des If-Frames darstellt. Die einzelnen Teilbefehle können dabei entweder Pointer oder Datenworte sein. Die Bedeutung der Operanden ist dabei folgendermaßen:

- Operand 1: Die Bedingung, die überprüft wird, bevor Op2 und Op3 evaluiert werden

- Operand 2: Then-Expression, wird evaluiert, wenn Op1 nicht explizit zu false (0) evaluiert
- Operand 3: Else-Expression, wird evaluiert, wenn Op1 zu false evaluiert wurde

Falls Op2 oder Op3 Pointer zu Ausdrücken sind, werden diese nur dann evaluiert, wenn die Bedingung gilt bzw. nicht gilt. Die Bedingung wird stets als erstes ausgewertet.

3.8.7 Noop/Eval

Es wird nichts berechnet oder ausgeführt. Diese Operation verzögert die Ausführung um einen Takt.

3.8.8 Microcode Instructionset

Ein wichtiger Bestandteil von Lisp Maschinen ist es normalerweise, sog. Mikrocode, der primitive Funktionen zusammenfasst und ähnlich einer Funktion in herkömmlichen Programmiersprachen dient, zu implementieren. Diese Idee wurde anfänglich verfolgt, auf Grund von Änderungen der Befehlsausführung und der Unterstützung von Cons Listen allerdings verworfen. Ein Beispiel aus einer frühen Phase der Entwicklung findet sich im folgenden Pseudocode. Dieser dient lediglich illustrativen Zwecken.

```
1   function sub
2     sp = bp
3     OPCODE = STACK[bp-1]
4     OPCODE -> ALU
5     ACC : STACK[bp-2]
6     while sp != top
7       ACC = ACC - STACK[bp]
8       sp = sp + 1
9     end while
10  end function
```

Am Ende einer Funktion sollten immer folgende Register und Funktionen gesetzt bzw. ausgeführt werden:

```
1   sp = bp
2   pc = pop()
3   bp = pop()
4   push(ACC)
```

4 Software

Für die Lisp Maschine galt es zunächst, einen funktionierenden Compiler zu entwickeln. Dieser sollte einige grundlegende Funktionen von Lisp, wie in Kapitel 2 beschrieben, implementieren und die Programme in Maschinencode kompilieren. Abschnitt 4.1 beschreibt die Entwicklung des Compilers. Die Implementierung wird in Abschnitt 4.2 beschrieben. Probleme und fehlende Features werden in Abschnitt 4.4 aufgegriffen. Insbesondere die unklaren Abläufe in der Hardware und diverse Iterationen über das Design werden hier aufgegriffen. Weitere Tools, die im Laufe entwickelt wurden, werden in Abschnitt 4.3 kurz erläutert.

4.1 Entwicklungsprozess

Die Kopplung von Hard- und Software, insbesondere bei einer Architektur, die ein besonderes Programmier- und Ausführungsmodell wie Lisp besitzt, ist sehr eng. Daher konnte Software, die auf der Lisp Maschine aufbaut, erst spät im Entwicklungsprozess geschrieben werden. Zunächst war unklar, ob und wie die Maschine den Programmcode ausführen würde. Nach einigen Iteration über das Ausführungs- und Speichermodell war schnell klar, dass die Software bis zu einem späteren Zeitpunkt warten muss. Darum wurde von der Softwaregruppe zunächst der Lispdialekt, *NanoLisp*, definiert. Die unterstützen Sprachkonstrukte wurden so ausgewählt, dass sie 1) möglichst rudimentär (d.h. Arithmetik, logische Ausdrücke und Datentypen) und 2) nur wenig spezielle Befehle enthalten. In der frühren Phase des Entwicklungsprozesses haben wir deshalb das Ziel gesetzt, lediglich statische Programme ohne Variablen, Funktionen und Kontrollfluss umzusetzen. Dies betrachteten wir als *Proof of Concept* und als minimale Implementation. In späteren Iterationen sollten dann zunächst Atome, die keine Schlüsselwörter (siehe Abschnitt 4.2) sind, unterstüzt werden. Spätere Erweiterungen sollten dann Unterstützung für Funktionen (ein Aspekt, dessen Umsetzung lange ungeklärt war) und Kontrollfluss beinhalten. Diese Sprachkonstrukte erfordern spezielle Verarbeitung, weswegen sie auch als *special forms* bezeichnet werden. Als Datentypen sollten lediglich Ganzzahlen sowie Atome dienen, Unterstützung für Strings oder Kommazahlen war nicht geplant. Eine Übersicht über den Entwicklungsprozess findet sich in Abschnitt 4.1.

Phase	Fokus der Umsetzung
I	Definition der Teilmenge von Lisp, Compiler Frontend
II	Arithmetische Ausdrücke
III	Logischen Ausdrücke
IV	Variablen
V	Lambda, If/Then/Else
VI	Binärer und menschenlesbarer Output

Abbildung 4.1: Phasen des Compilerdesigns

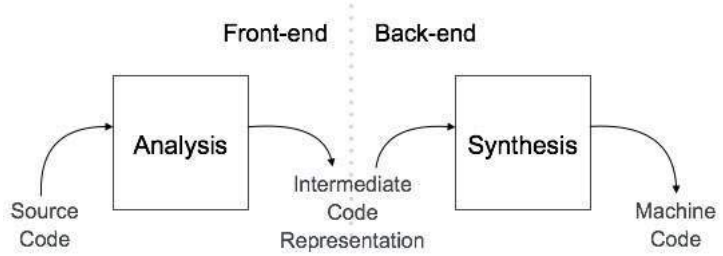


Abbildung 4.2: Klassische Architektur eines Compilers

4.2 Compiler

Zur Umsetzung des Compilers sind wir der klassischen Architektur gefolgt. Diese besteht, wie zu sehen in Abschnitt 4.2, aus dem Frontend, das den Code entgegennimmt und analysiert und anschließend eine Zwischendarstellung (Intermediate Representation, IR) erzeugt. Dies ist meist ein abstrakter Syntaxbaum (AST). In Lisp ist der Code bereits der AST, somit ist dieser Schritt trivial. Anschließend wird der AST im Syntheseschritt in tatsächtlichen Maschinencode umgewandelt. Das Frontend wurde in den verschiedenen Phasen in Python, Javascript (für einen grafischen Simulator) [1] und ML implementiert. Die ML Version diente dabei als Prototyp und zur Exploration, da die Sprache sich besonders zur Entwicklung von Compilern und Parsern eignet. Der grafische Simulator wurde auf Grund eingestellt, da fortan der Fokus auf dem Compiler lag und diser bereits über weiterführende Analysemöglichkeiten verfügte. Der finale Compiler ist in Python geschrieben und ist zusammen mit der Spezifikation der Hardware in das Projekt eingepflegt.

Abschnitt 4.2 zeigt die Definition des Syntaxbaums anhand des ML Codes. Es werden 4 Datentypen unterstützt, Atome (entweder Variablen oder Schlüsselwörter), Ganzzahlen, Cons Listen (S-Expressions) und Nil. Damit lassen sich auch die Ma-

[1] Das Repository befindet sich unter `https://github.com/teekaay/lisp-typescript`

```
1  type t =
2      | Atom of string
3      | Numeric of int
4      | Cons of t * t
5      | Nil
```

Abbildung 4.3: Darstellung des Syntaxbaums in ML Notation

```
1  let car t =
2      match t with
3      | Cons (hd,_) -> hd
4      | _ -> raise InvalidAccess
```

Abbildung 4.4: Interne Implementierung des CAR Befehls

schinenbefehle sowie die Speicherstruktur nahezu identisch übersetzen.

Beispielhaft soll hier die Implementierung des CAR Befehls gezeigt werden, die sich in Abschnitt 4.2 findet (analog für CDR, das statt dem Kopf den Rest wiedergibt). Da die rechnerinterne Speicherstruktur ähnlich wie ein Knoten im AST aufgebaut ist, lässt sich CAR einfach übersetzen und sowohl auf dem AST als auch später in der Übersetzung äquivalent wiedergeben. So kann dann einfach das erste Element des aktuellen Frames benutzt werden, ähnlich wie hier das erste Element der Cons Liste benutzt wird.

4.3 Weitere Tools

Im Laufe der Implementierung und Entwicklung wurde neben dem eigentlichen Compiler auch über einen Weg zur Speicherbereinigung nachgedacht. Diese Softwarekomponente soll die automatische Speicherallozierung sowie -freigabe übernehmen. Da Garbage Collection (GC) ein sehr komplexes Thema ist, wurde aber ein einfacher Weg entwickelt, wie solch ein hypothetischer GC aussehen kann. Der Gargabe Collector verwaltet dabei eine Liste von Speicheradressen, die alle einen Tag bekommen. Dieser markiert die Zellen als nicht referenziert (z.B wenn eine Funktion abgearbeitet ist oder der Stack leer wird), als referenziert oder unklar (z.B verschachtelte Listen). Tritt dann ein Ereignis ein, bspw. der Stapel oder der Speicher fast voll sind, verrichtet der GC seine Arbeit. Dabei folgt er Pointern und markiert Zellen dabei als verwendet und alle nicht verwendeten Zellen werden freigegeben.

Zusätzlich gibt es ein einen Assembler und einen Disassembler, der Maschinen-

code regenerieren kann.

4.4 Probleme und fehlende Features

Die Umsetzung des Compilers war eine Herausforderung. Da Lisp in den meisten Fällen eine interpretierte Sprache ist, es also keinen Compiler gibt (abgesehen von Bytecode Compilern für virtuelle Maschinen), hatten wir keine Referenz. Anders als bei Architekturen, die sequentielle Ausführungsmodelle haben, konnten wir auch auf keine bestehenden Materialen zurückgreifen. Deshalb wurde ein Großteil der Entwicklung experimentell verbracht. Dies hatte den negativen Effekt, dass wenig produktiver Quellcode entstand, sondern viele gescheiterte Versuche. Andererseits hatte das Vorgehen und die enge Zusammenarbeit mit der Hardwaregruppe den Vorteil, dass beide Seiten von den Diskussionen profitiert haben. So entstand die Idee zur Umsetzung von Funktionsaufrufen und der besonderen Speicherstruktur Überlegungen zur Umsetzung von Lisp Quellcode in einen abstrakten Syntaxbaum.

Wesentliche Aspekte, die nicht bearbeitet worden sind, sind z.B Kommazahlen und Strings. Eine grundsätzliche Idee, die wir verworfen haben, war das Aufgreifen der Ideen der SECD Maschine, deren Implementierung in ML in [1] beschrieben wird. Allerdings war zu dem Zeitpunkt die Architektur bereits in einem Stadium, das bereits als festgelegt gelten kann.

Ein Aspekt, der der praktischen Nutzung widerspricht ist fehlende Garbage Collection. Da bei ungeschickter Nutzung oder Rekursion der Speicher schnell voll werden kann (z.B rekursive Berechnung von Fibonacci Zahlen), können größere Berechnungen nicht durchgeführt werden.

Eine Umsetzung von Funktionsaufrufen ist durch die Hardware direkt möglich, wird aber vom Compiler nur für vorher festgelegte Funktionen ermöglicht. Vom Benutzer definierte Funktionen sind daher nicht möglich, aber könnten einfach eingebaut werden.

5 Literatur

[1] Olivier Danvy. „A Rational Deconstruction of Landin's SECD Machine". In: *Basic Research in Computer Science (BRICS)* 03.33 (Okt. 2003).

[2] Paul Graham. „The Roots of Lisp". In: *Retrieved October* (2002). URL: `http://www.cs.uml.edu/ecg/pub/uploads/OrganizationProgrammingLanguagesFall2008/genesis-of-lisp-jmc.pdf`.

[3] Thomas F. Jr. Knight u. a. *CADR*. AI Memo 528. Massachusetts Institute of Technology, Artificial Intelligence Labroratory, März 1981. URL: `http://www.unlambda.com/cadr/aimemo528_75pgs.pdf`.

[4] John McCarthy. „Recursive Functions of Symbolic Expressions and Their Computation By Machine, Part I". In: *Communications of the ACM* (1960).

[5] Symbolics. *Symbolics Announces the First True Single-Chip Lisp CPU*. Techn. Ber. Eleven Cambridge Center, Cambridge, 1987. URL: `http://home.rbcarleton.com/rbc/symbolics/announcements/19870519-ivory.txt`.

Abbildungsverzeichnis

6 Anhang

6.1 Ablauf bei einfachem arithmetischem Ausdruck

Der Ablauf des Programs wird maßgeblich vom Stack Bestimmt. Dabei ist erwähnenswert, dass es keinen Programmcounter gibt, dieser wird nicht benötigt. Stattdessen kommt das nächste Befehlswort entweder vom Stack oder die Adresse ist ein Pointer. Als Beispiel diene der Ausdruck (+ 1 2 (- 5 2)). Die Abarbeitung eines Ausdrucks wird im folgenden detailiert dargestellt.

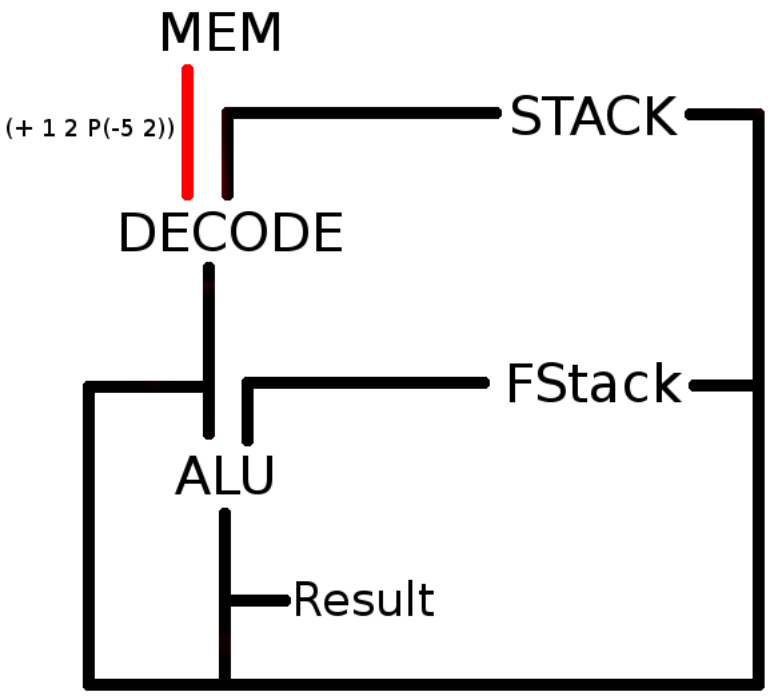


Abbildung 6.1: Als erstes wird der Ausdruck geladen. P(- 5 2) steht dabei für den Pointer auf den Teilausdruck (- 5 2)

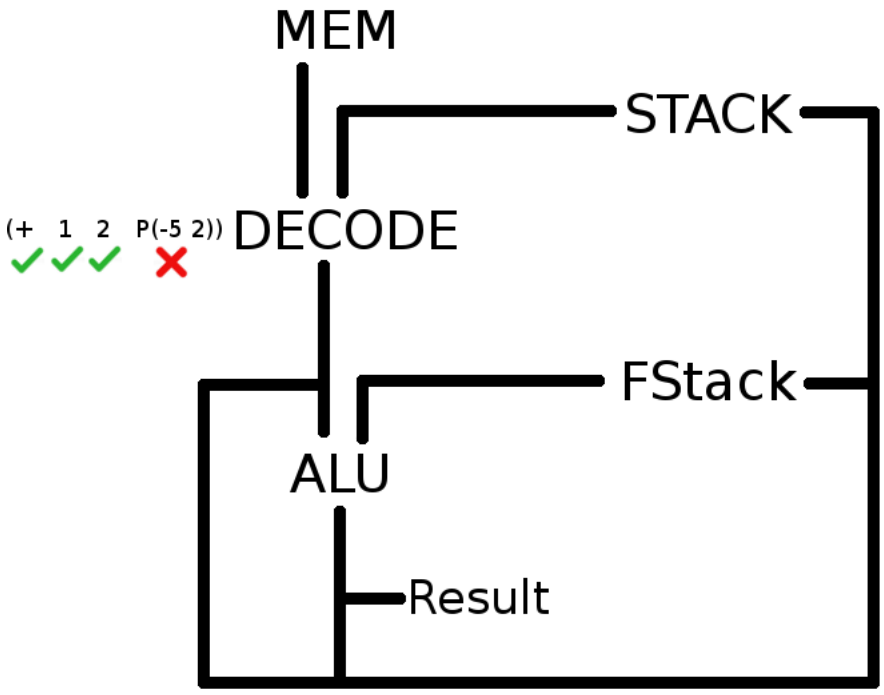


Abbildung 6.2: Als nächstes stellt die Decode Einheit fest, ob die ALU den Ausdruck bereits auswerten kann. Dies ist hier nicht der Fall, da der Pointer noch nicht ausgewertet wurde

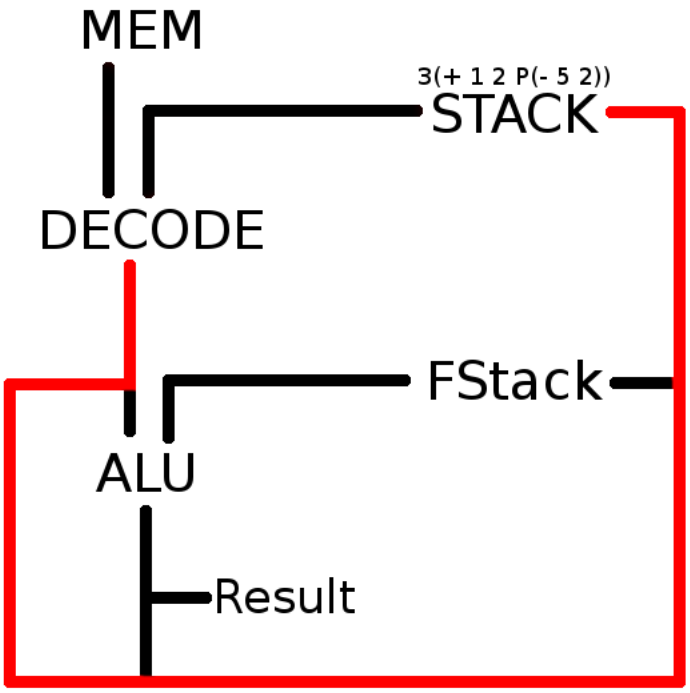


Abbildung 6.3: Daher wird dieser Ausdruck zurückgestellt, da erst dem Pointer gefolgt werden muss. Die 3, die beim Speichern auf dem Stack hinzugefügt wird, ist der Result-Offset, der anzeigt an welcher Stelle der Pointer - durch den Wert des Ausdrucks auf den er referenziert - ersetzt werden muss

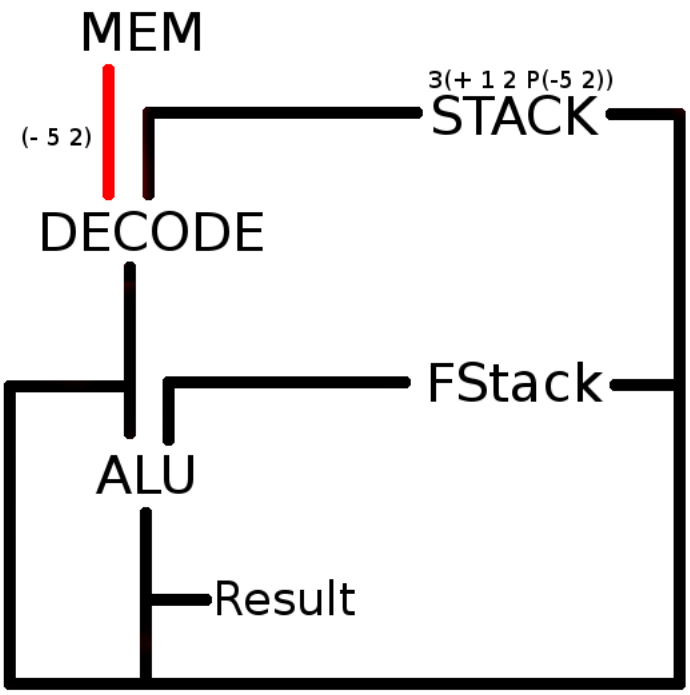


Abbildung 6.4: Es wird an der durch den Pointer gegebenen Adresse der nächste Ausdruck geladen

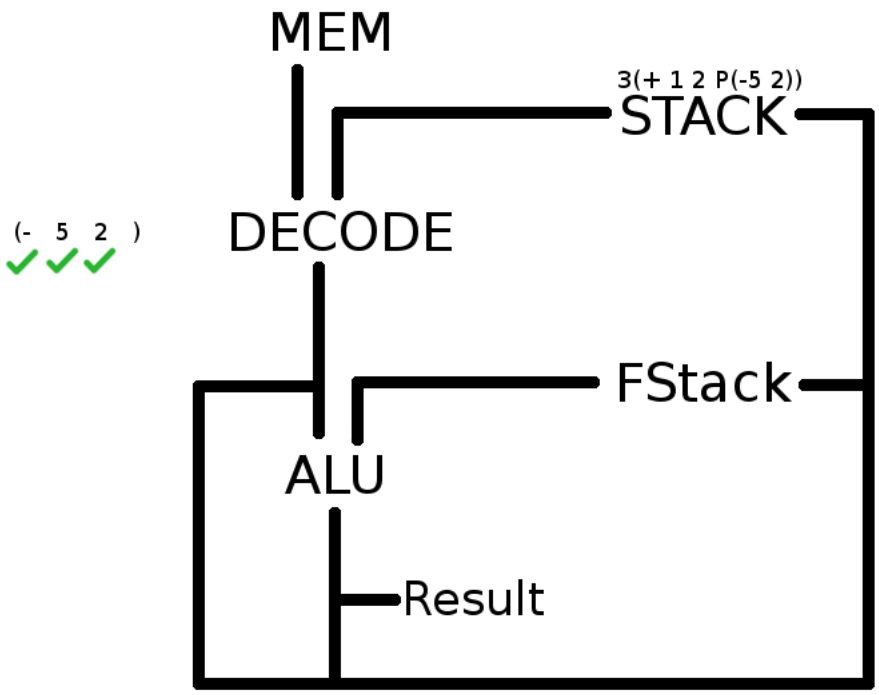


Abbildung 6.5: Diesmal wird festgestellt, dass die ALU diesen Ausdruck auswerten kann

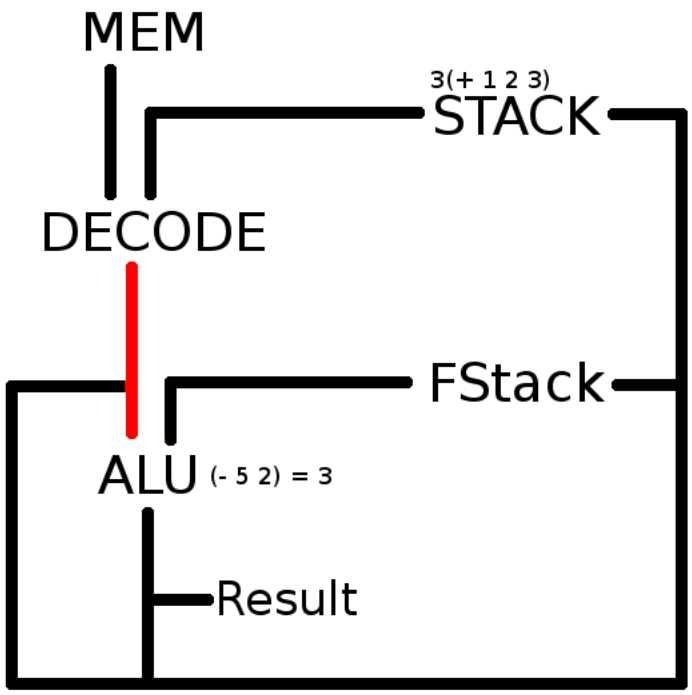


Abbildung 6.6: Daher wird der Ausdruck an die ALU gegeben, die seinen Wert berechnet

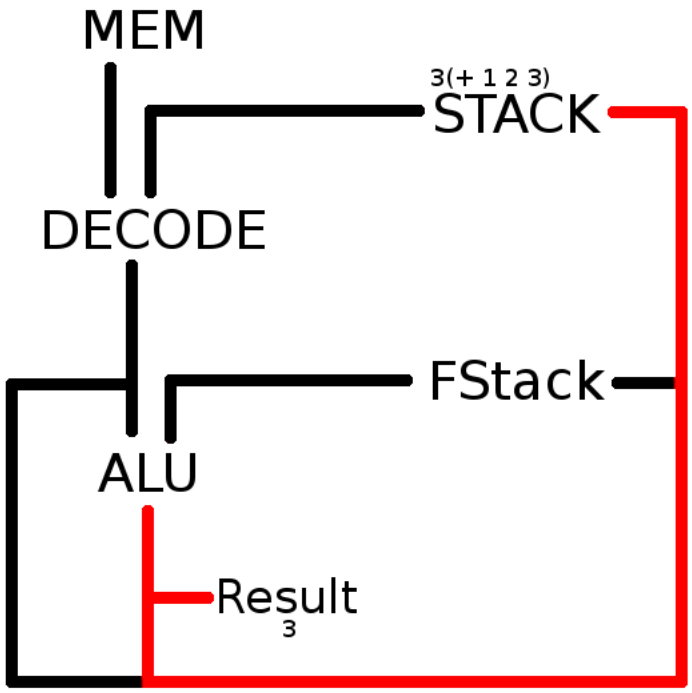


Abbildung 6.7: Das Ergebnis landet zum Einen im Speziellen Result Register, zum Anderen wird es auf den Stack gelegt. Dabei wird die durch den Offset gegebene Stelle durch das berechnete Ergebnis ersetzt – Eigentlich wird der nächste Ausdruck erst vom Stack geholt, und bei diesem Holen die entsprechende Stelle ersetzt

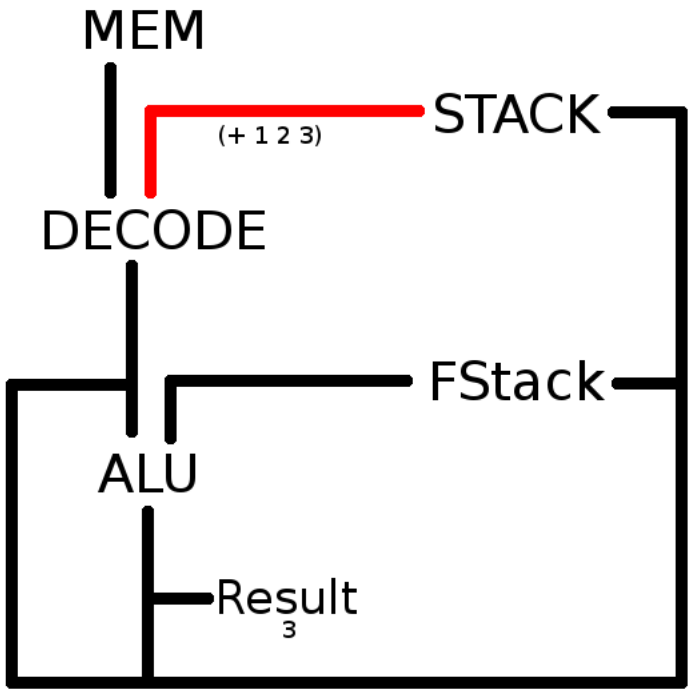


Abbildung 6.8: Der als nächstes auszuwertende Ausdruck kommt nun vom Stack

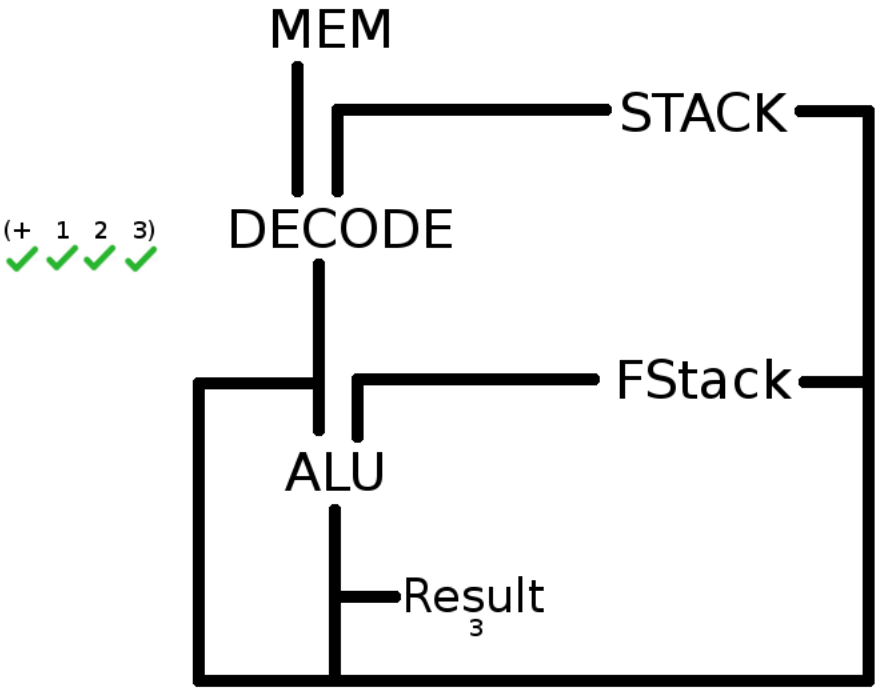


Abbildung 6.9: Diesmal wird festgestellt, dass die ALU den Ausdruck berechnen kann

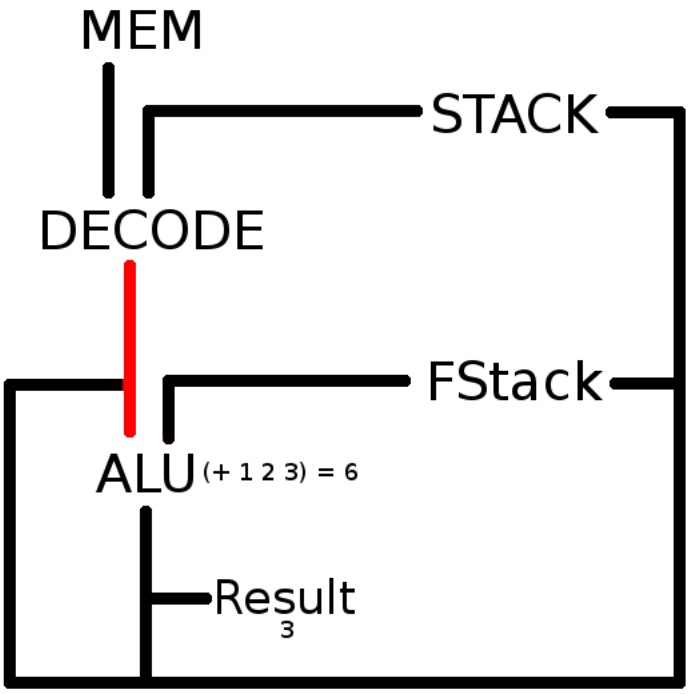


Abbildung 6.10: Daher berechnet die ALU den Wert des Ausdrucks

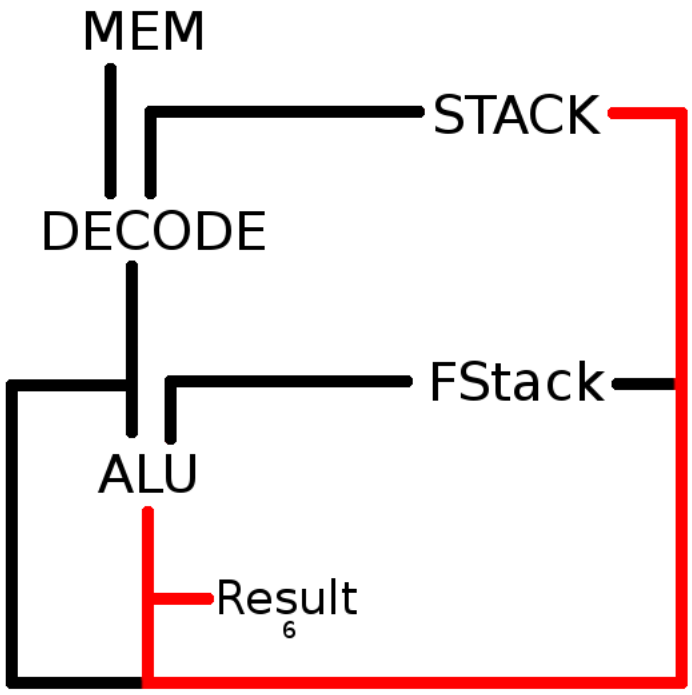


Abbildung 6.11: Der Wert landet wieder im Result Register. Im Prinzip wird beim Schreiben auf den Stack erkannt, dass der Stack Leer ist und daher terminiert werden kann. Tatsätchlich liegt als Bottom Of Stack immer (eval "P auf Adresse 0") mit adresse 0 als Programmbegin, sodass der Pointer durch das Endergebnis ersetzt werden kann, und danach festgestellt werden kann, das der Stack leer ist

6.2 Ablauf bei Auswertung einer Funktion

Nun wird der Ausdruck (* X_1 (f (– X_1 1)) veranschaulicht, wobei * die Multiplikation und – die Subtraktion sei. In den Abbildungen haben wir (* X_1 (f (– X_1 1)) durch (f ...) ersetzt,um anzudeuten, dass es sich im Prinzip nur um den Rekursiven Aufruf handelt. Wir überspringen den rekursiven Aufruf, da er genauso abläuft, wie der vorher gezeigte.

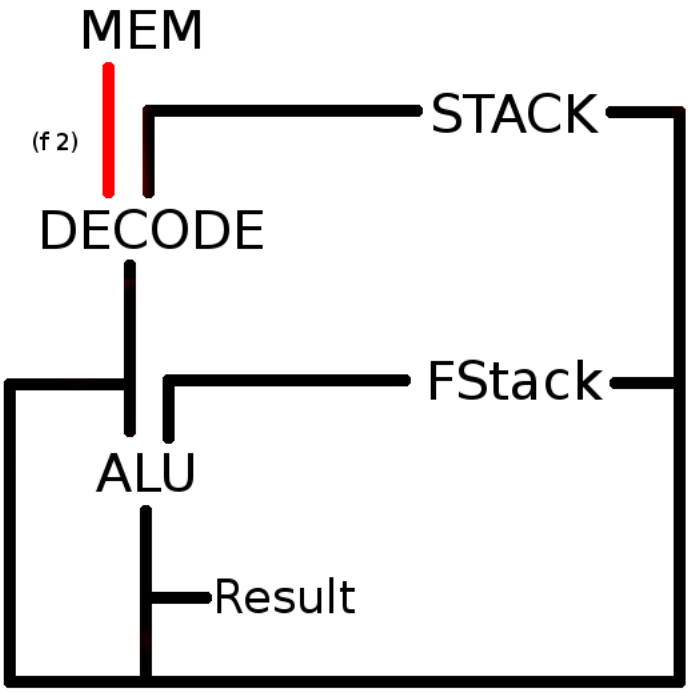


Abbildung 6.12: Wie üblich wird der Ausdruck aus dem Speicher geladen

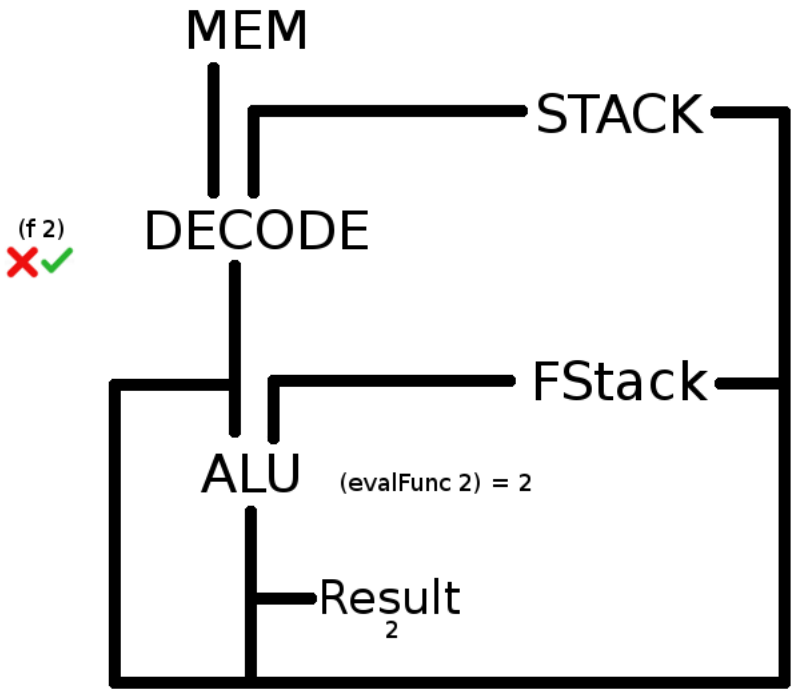


Abbildung 6.13: Es wird festgestellt, dass es sich bei f um eine benutzerdefinierte Funktion handelt und nicht um einen Opcode, daher wird die Funktion im nächsten Schritt aufgerufen

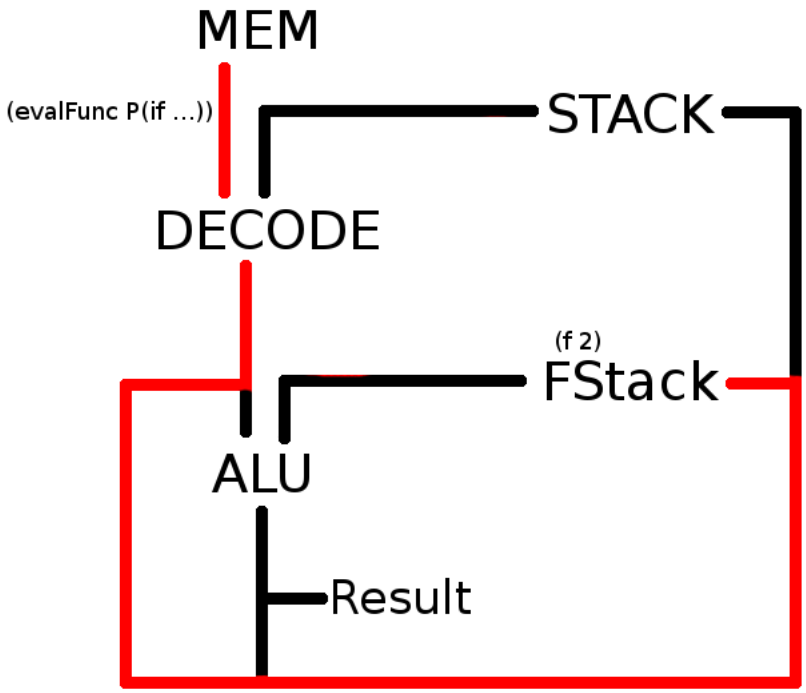


Abbildung 6.14: Der Ausdruck (f 2) wird auf dem Funktionsparameter Stack abgelegt. Der nächste Ausdruck ist durch die Adresse von f gegeben (f selber ist der Pointer)

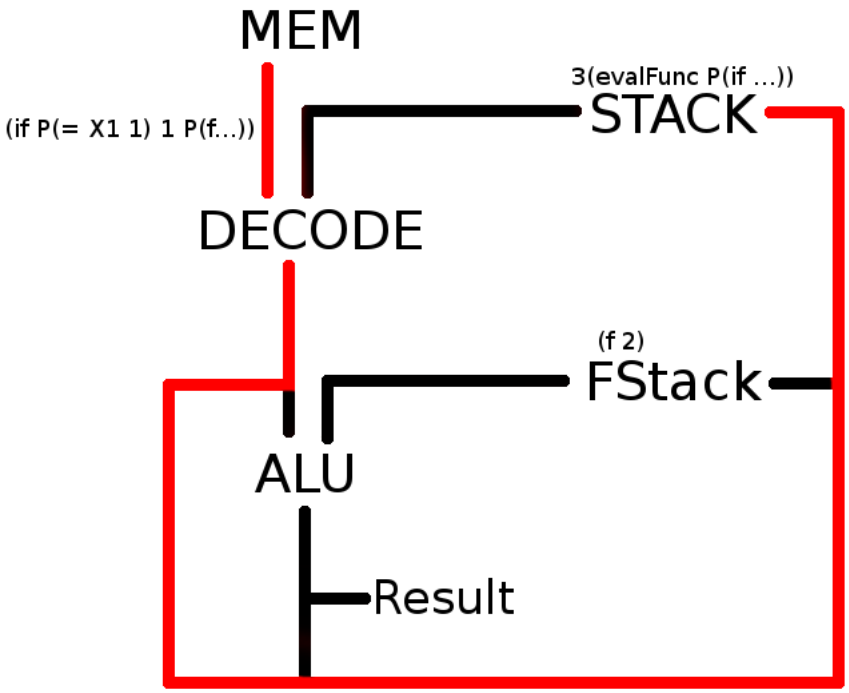


Abbildung 6.15: Beim Ausführen von EvalFunc wird festgestellt, dass erst dem Pointer auf den eigentlichen Funktionsrumpf gefolgt werden muss [1]

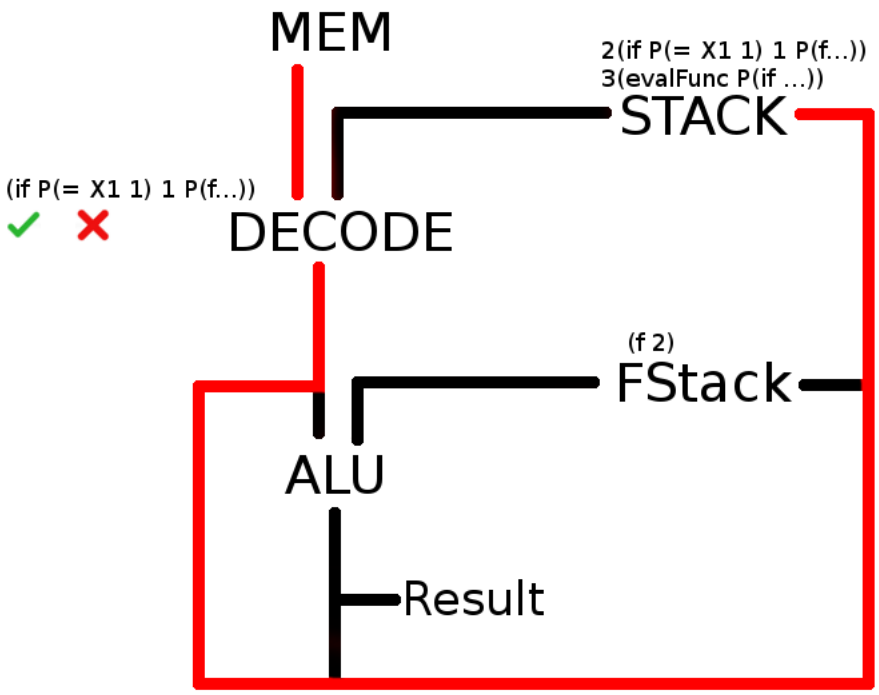


Abbildung 6.16: Beim Ausführen von If wird festgestellt, dass der als Bedingung angegebene Ausdruck P(= X_1 1) zuerst ausgewertet werden muss

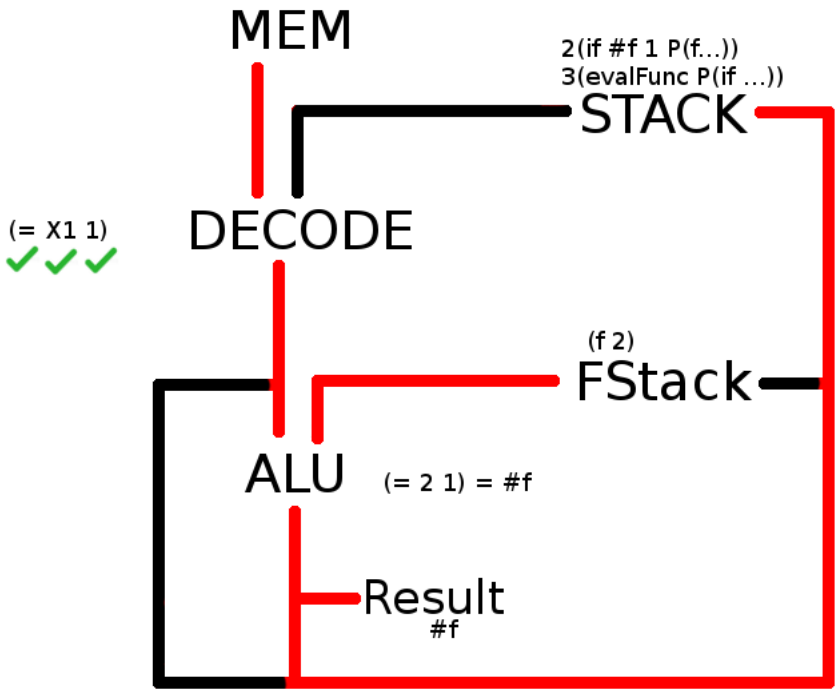


Abbildung 6.17: Die ALU wertet nun die Bedingung aus. Lediglich der Platzhalter X_1 wird vor der Auswertung durch den entsprechenden wert vom FStack ersetzt

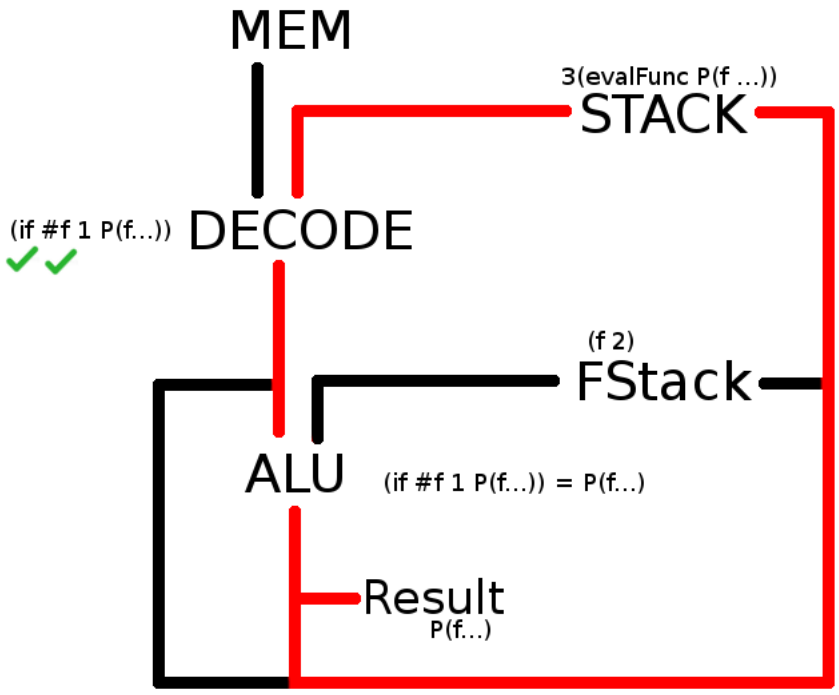


Abbildung 6.18: Nun kann das If ausgewertet werden, da die Bedingung zu False ausgewertet wurde. Deswegen, ist der Wert des If Ausdrucks der Wert des an letzter Stelle angegebenen Ausdrucks

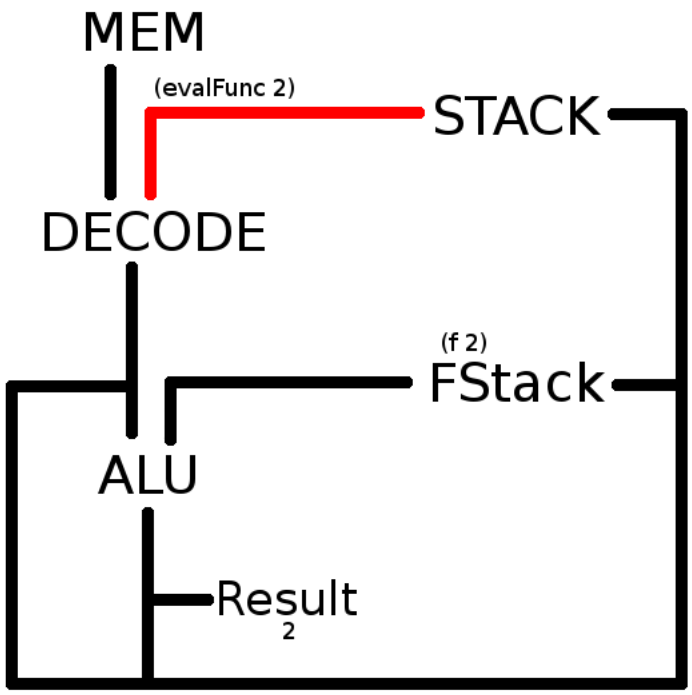


Abbildung 6.19: Der berechnete Wert ist 2, daher wird nun EvalFunc mit dem Parameter 2 geladen

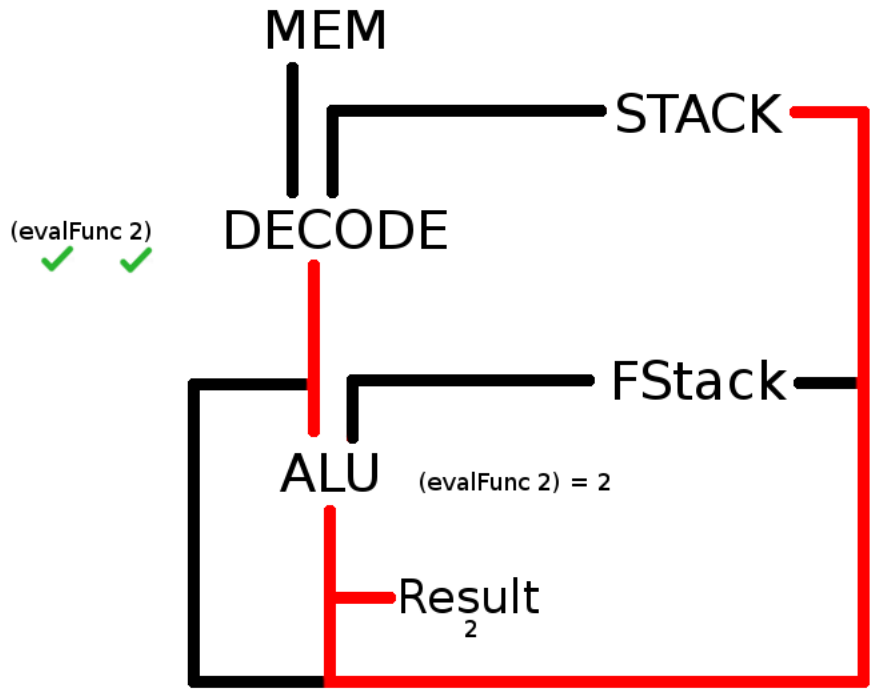


Abbildung 6.20: Nun wird der Ausdruck, auf den EvalFunc zeigt, ausgewertet. Da der Stack dann leer ist, wird hier terminiert. Man Beachte, dass auf FStack Pop ausgeführt wird, da man mit dem Auswerten des EvalFunc Ausdrucks die Funktion vollständig ausgewertet hat und nun in die aufrufende Funktion zurückspringt