Miniprojekt: Minimax-Maschine Dokumentation

Maximilian Lumpe, Niklas Blume, Jan Feuchter, Phu Bac Duong

Hardware Projekt 2017
24. Januar 2017

Inhaltsverzeichnis

1 Einleitung

In diesem Miniprojekt im Rahmen des Hardware-Praktikums beschäftigen wir uns mit der Minimax-Maschine, welche uns grundlegend aus der Vorlesung "Grundlagen der Rechnerarchitektur" bekannt ist. Zur Lösung der Aufgaben ist es hierbei notwendig, die vorgegebene Grundstruktur der Maschine geeignet zu erweitern, um die Algorithmen zu realisieren.
Die Vorbereitung auf unser Projekt wird dokumentiert und strukturiert durch das von uns erstellte Pflichtenheft. Das Pflichtenheft wird nur unsere Vorbereitung beinhalten. Die Ergebnisse werden in einer weiteren Dokumentation enthalten sein.

2 Aufgabe: Paketanalyse

Nach unserem Verständnis ist das Ziel der Aufgabenstellung das Implementieren des Algorithmus "Paketanalyse" auf der Minimax-Maschine. Dieser Algorithmus wertet die Länge des Nutzdatenteils der Datenpakete aus dem Speicher der Maschine aus.
Jedes Paket besteht aus einem Header mit 80 Bits, gefolgt von dem Datenteil mit variabler Länge. Ein Paket beginnt mit dem festgelegten Bitmuster 1110. Der Header enthält eine 2 Bytes lange Kanalnummer, die bei der Bitstelle 32. beginnt. Zu einem Kanal gehören mehrere Datenpakete mit einer eindeutigen Kanalnummer. Die Anzahl der Bits, die in den Speicher geladen werden, wird als bekannt vorausgesetzt und wird in ein entsprechendes Register vorgeladen.
Nun soll der "Paketanalyse"-Algorithmus eine Tabelle, die Kanalnummern und zugehörige Datenlängen (in Bits) enthält, anlegen. Haben mehrere Pakete dieselbe Kanalnummer, so werden die Längen des Nutzdatenteils addiert. Die Tabelle soll ab einer beliebigen Speicheradresse außerhalb des Paketfeldbereichs im Hauptspeicher der Maschine abgelegt werden.
Diese Aufgabenstellung soll mit dem gegebenen Minimax-Simulator simuliert und getestet werden. Die Maschine kann durch vorgegebene Bauteile erweitert werden, was sich jedoch auf die Bewertung auswirkt. Der Algorithmus wird in Form der Steuertabelle implementiert und soll außerdem als Flussdiagramm abgegeben werden.

3 Ist-Analyse der Basismaschine

Im Rahmen der Vorlesung "Grundlagen der Rechnerarchitektur" haben wir die Minimax-Maschine als Beispiel kennengelernt. Diese Minimax-Maschine ist ein Rechensystem, welches auf dem Grundprinzip der Von-Neumann-Architektur basiert. Im Wesentlichen besteht das System aus einigen Registern und einer arithmetisch-logischen Einheit (ALU), welche den Datenpfad bilden, und einem Hauptspeicher (HS), in welchem Code und Daten gemeinsam liegen.

3.1 Register der Basismaschine

Die Basismaschine besteht zunächst nur aus fünf Registern, welche für Grundfunktionen der Maschine benötigt werden. Diese sind ACCU (zum Speichern von Ergebnissen der ALU), PC (zum Speichern der Nummer der aktuellen Programmzeile), MDR (zum Speichern von aus dem HS gelesenen Daten), IR (zum Speichern des Befehls der aktuellen Programmzeile) und MAR (zum Anlegen bestimmter Speicheradressen).
Die Register sind als zweiflankengesteuerte Master-Slave-Flipflops ausgelegt. Damit kann ein Register während eines Taktimpulses zunächst als Quelle und dann als Ziel dienen. Der Befehlsablauf dieses Systems wird über ein Mikroprogramm festgelegt. Für unser Projekt kann die Architektur der Minimax-Maschine um zusätzliche Register erweitert werden.

3.2 Operationen der ALU

In der Basismaschine kann die ALU zunächst nur vier Operationen ausführen (bezeichnet mit: ADD, SUB, TRANS.A, TRANS.B). Die ALU kann aber durch zusätzliche Operationen, wie z.B. dem DIV-Befehl, ergänzt werden.

- ADD: Addiert zwei Register
- SUB: Subtrahiert zwei Register
- TRANS.A: Leitet den Wert A durch
- TRANS.B: Leitet den Wert B durch

Das Ergebnis kann an eines der oben genannten Register geleitet werden.

Symbol	ALU-Operation	ALU Ctrl
ADD	ALUresult $\leftarrow A + B$	00
SUB	ALUresult $\leftarrow -A + B$	01
Trans.A	ALUresult $\leftarrow A$	10
Trans.B	ALUresult $\leftarrow B$	11

Table 1: ALU-Operationen der Basisimaschine

3.3 Ein- und Ausgabewerte der ALU

Die Eingangswerte A und B der ALU können über zwei Multiplexer bestimmt werden. A kann die Werte 0, 1 und ACCU annehmen, B die Werte MDR, PC, IR und ACCU.
Das Ergebnis einer Operation wird als ALU-result auf einen Bus gegeben, wodurch es an jedes beliebige Register geleitet werden kann. Eine weiter Datenleitung führt zur CU, wodurch bestimmte Flags gesetzt werden können(z.B. wenn das Ergebnis der Operation 0 ist)

3.4 CPU-Befehle

Die CPU-Befehle sind 32 Bit breit, wobei das höchstwertige Byte immer den Opcode enthält. Die verbleibenden 3 Bytes bestimmen dann den Adressteil. Der Opcode und die Operanden werden zunächst vollständig aus dem Hauptspeicher geladen, bevor ein Befehl ausgeführt wird.

4 Finale Implementierung

Bei der Implementierung der Paketanalyse muss zwischen mehreren Teilschritten unterschieden werden. Zunächst wird nach dem Startbitmuster des ersten Paketes gesucht (1). Ist ein Paket identifiziert, können die ersten 32 Bits (dies entspricht einer Speicherzelle) übersprungen werden. Somit beginnt mit der zweiten Speicherzelle die Kanalnummer des Paketes, diese wird extrahiert und abgespeichert (2). Nach 32 weiteren Bits kommt nun der Datenteil, welcher Bit für Bit ausgelesen wird, während gleichzeitig ein Zählregister inkrementiert wird (3). Während dieses Vorgangs wird permanent überprüft, ob ein neues Startbitmuster vorliegt, sodass ein neues Paket anfängt (4). Ist dies der Fall, wird das Zählregister auf den momentanen Wert für die Kanalnummer addiert. Die Kanalnummer dient dabei als Offset für die Speicherzelle (5).

4.1 Identifizierung des Startbitmusters

Da für das Startbitmuster nur die vier niedrigwertigsten Bits untersucht werden, müssen zunächst für einen Vergleich die restlichen ausgeblendet werden. Dies erfolgt durch eine bitwise-and Verknüpfung mit Nullen. Anschließend wird das Ergebnis mit dem Startbitmuster verglichen. Sind sie nicht identisch, wird die Speicherzelle geshiftet. Dies wird solange wiederholt, bis das Ergebnis identisch ist, dann wird mit Schritt 2 fortgefahren.

4.2 Kanalnummer extrahieren

In der ersten Speicherzelle stehen nun die 4 Bits des Startmusters, sowie 28 Bits die übersprungen werden. Also enthält die erste Speicherzelle keine weiteren Informationen, sodass mit der nächsten Speicherzelle fortgefahren wird. Nun müssen die ersten 16 Bits, die die Kanalnummer beinhalten, extrahiert werden. Hierfür sorgt wieder die bitwise-and Verknüpfung der restlichen Bits mit Nullen. Die Kanalnummer muss für die spätere Nutzung abgespeichert werden.

4.3 Datenbits zählen

Auch die zweite Speicherzelle enthält nun keine Informationen mehr. Da in der dritten Speicherzelle noch 16 Bits sind, die nicht zum Datenteil gehören,

werden im ersten Durchlauf nur 16 auf das Zählregister für die Länge des Datenteils addiert. Für jeden weiteren Schleifendurchlauf werden dann 32 addiert.

4.4 Auf neues Startbitmuster überprüfen

Während die Datenbits nach und nach durch shiften gezählt werden, wird permanent auch der Test aus (1) durchgeführt. Sobald das Startbitmuster erkannt ist, bricht die Schleife aus (3) ab und es folgt Schritt (5).

4.5 Zählregister zur Tabelle hinzufügen

Um die Tabelle zur Speicherung der Längen von den eigentlichen Daten zu trennen, darf diese erst in einer Speicherzelle nach der maximalen Datenlänge beginnen. Ab dieser Speicherzelle repräsentieren immer zwei aufeinanderfolgende Speicherzellen eine Reihe dieser Tabelle. So beinhaltet die erste Speicherzelle die Kanalnummer 0, die zweite Speicherzelle den entsprechenden Wert für diesen Kanal. Die dritte Speicherzelle beinhaltet wieder die Kanalnummer der nächsten Zeile, in diesem Fall die Nummer 1. Danach kommt wieder der Wert für Kanal 1 und so weiter. Die Adresse der ersten Speicherzelle für die Adresse soll dabei in einem Register gespeichert sein. Nun wird die doppelte Kanalnummer auf dieses Register addiert, um die Speicherzelle für eine bestimmte Kanalnummer zu erhalten. Eine Speicherzelle weiter ist somit der entsprechende Wert. Auf den Inhalt dieser Speicherzelle wird dann der Inhalt des Zählregisters addiert. Anschließend werden die temporären Register zurückgesetzt und der Algorithmus beginnt erneut mit Schritt (2).

5 Angestrebte Projektergebnisse

- Ein Pflichtenheft, in dem wir unsere Vorüberlegungen festhalten.
- Eine vollständige und funktionsfähige Lösung mit allen Testdateien.
- Einen Schaltplan, bei dem wir die gegebene Minimax-Maschine um die notwendigen Elemente erweitert haben.
- Eine vollständige Dokumentation, mit der wir unseren Algortihmus und unsere implementierte Lösung beschreiben und erläutern.

6 Arbeitsaufteilung

Name	**Aufgabe**
Maximilian Lumpe	Ausarbeitung der Dokumentation
Niklas Blume	Anfertigung der Minimax-Maschine
Jan Feuchter	Anfertigung der Minimax-Maschine
Phu Bac Duong	Ausarbeitung der Dokumentation

7 Projektdurchführung

7.1 Initialisierung und erstes Startbit-Muster

	0	01011	-	-	0	-	010	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	-	1	MaxData ← 280
	1	-	-	1	1	1	-	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	-	2	MDR ← M[MAR]
start	2	01000	0000	-	0	-	100	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	-	3	Tmp ← 15 & MDR
	3	01001	0100	-	0	-	001	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	-	4	Tmp ← Tmp - 14
	4	-	0100	-	0	-	011	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1 0	7 5	Tmp == 0?
shift	5	-	0000	-	0	-	101	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	-	6	MDR ← MDR[31]@MDR[31..1]
	6	00001	1001	-	0	-	000	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	-	2	OffsetKanal ← 1 + OffsetKanal OffsetData ← 1 + OffsetKanal

Zunächst wird die erste Speicherzelle in das Register MDR geladen. Durch eine bitwise-and Verknüpfung werden nur die vier LSBs betrachtet. Sind diese gleich 14, startet der Algorithmus. Ansonsten shiftet das MDR und die neuen LSBs werden betrachtet, solange bis das Startbitmuster gefunden wurde. Gleichzeit wird das Offset vom Speicherzellenanfang mitgespeichert.

7.2 Kanalnummer extrahieren

kanal	7	00001	0001	-	0	-	000	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	-	8	PC ← 1 + PC MAR ← 1 + PC
	8	00001	1000	-	0	-	001	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1 0	6 9	MaxData ← MaxData - 1
	9	-	-	1	1	1	-	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	-	10	MDR ← M[MAR]
kanal-shift	10	-	1001	-	0	-	011	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1 0	13 11	OffsetKanal == 0?
	11	-	0000	-	0	-	101	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	-	12	MDR ← MDR[31]@MDR[31..1]
	12	00001	1001	-	0	-	001	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	-	10	OffsetKanal ← OffsetKanal - 1
	13	01010	0000	-	0	-	100	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	-	14	Kanal ← 65535 & MDR

Nun wird die nächste Speicherzelle geladen und diese um das Offset geshiftet. Dadurch erhält man in den unteren 16 Bit die Kanalnummer, welche durch bitwise-and mit 65.535(`0xFFFF`) extrahiert wird.

7.3 Datenanfang ermitteln

data	14	00001	0001	-	0	-	000	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	-	15	PC ← 1 + PC MAR ← 1 + PC
	15	00001	1000	-	0	-	001	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1 0	7 16	MaxData ← MaxData - 1
	16	-	-	1	1	1	-	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	-	17	MDR ← M[MAR]
	17	00000	-	-	0	-	010	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	-	18	CNT ← 0
	18	00111	1010	-	0	-	000	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	-	19	CNT2 ← 16 + OffsetData
	19	10000	1100	-	0	-	001	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	-	20	OffsetData ← 16 - OffsetData
	20	00001	1110	-	0	-	001	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1 0	21 25	flag - 1 == 0?
	21	00001	1110	-	0	-	001	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	-	22	flag ← flag - 1
	22	00001	0001	-	0	-	000	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	-	23	PC ← 1 + PC MAR ← 1 + PC
	23	-	-	1	1	1	-	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	-	24	MDR ← M[MAR]
	24	00001	1000	-	0	-	001	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1 0	6 25	MaxData ← MaxData - 1

Nun wird wieder die nächste Speicherzelle betrachtet. Auf das Offset wird zusätzlich 16 addiert, da im ersten Schleifendurchlauf 16 Bit übersprungen werden müssen. Die verbleibenden Bits bis zum Speicherzellenende werden

in OffsetData mitgezählt. Sollte das Offset größer als 16 sein (Flag vom Ende des Algorithmus ist gesetzt) ist eine weitere Speicherzelle ohne Information und kann übersprungen werden.

7.4 Datenbits zählen

data-shift	25	-	0110	-	0	-	011	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1 0	28 26	CNT2 == 0?
	26	-	0000	-	0	-	101	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-	27	MDR ← MDR[31]@MDR[31..1]
	27	00001	0110	-	0	-	001	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	-	25	CNT2 ← CNT2 - 1
	28	01000	0000	-	0	-	100	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	-	29	Tmp ← 15 & MDR
	29	01001	0100	-	0	-	001	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1 0	38 30	Tmp ← Tmp - 14
	30	-	0000	-	0	-	101	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-	31	MDR ← MDR[31]@MDR[31..1]
	31	00001	0101	-	0	-	000	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	-	32	CNT ← 1 + CNT
	32	00001	1010	-	0	-	001	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	-	33	OffsetData ← OffsetData - 1
	33	-	1010	-	0	-	011	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1 0	34 28	OffsetData == 0?
	34	00001	0001	-	0	-	000	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	-	35	PC ← 1 + PC MAR ← 1 + PC
	35	-	-	1	1	1	-	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-	36	MDR ← M[MAR]
	36	01101	-	-	0	-	010	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	-	37	OffsetData ← 32
	37	00001	1000	-	0	-	001	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1 0	63 28	MaxData ← MaxData - 1

Zunächst wird diese Speicherzelle um das verbleibene Offset geshiftet. Anschließend wird überprüft, ob nun die vier LSBs wiederum das Startbitmuster sind. Ist dies der Fall, springt der Algoritmus zum Schritt. Ist dies nicht der Fall, wird die Speicherzelle geshiftet, der Counter der Datenbits erhöht sowie der Counter der verbleibenden Bits bis Speicherzellenende erniedrigt. Sind keine Bits mehr bis Speicherzellenende vorhanden, wird in die nächste Speicherzelle gewechselt und der Wert wieder auf 32 gesetzt. Die Speicherzelle wird jedoch in jedem Fall solange geshiftet, bis das Startbitmuster erkannt wurde.

7.5 Abspeichern

save	38	-	0001	-	0	-	011	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	-	39	Tmp ← PC
	39	01011	0111	-	0	-	000	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-	40	PC ← 280 + Kanal
	40	01100	0001	-	0	-	000	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	-	41	PC ← Kanal + PC MAR ← Kanal + PC
	41	-	0111	-	0	-	011	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-	42	MDR ← Kanal
	42	-	-	-	1	0	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-	43	M[MAR] ← MDR
	43	00001	0001	-	0	-	000	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	-	44	PC ← 1 + PC MAR ← 1 + PC
	44	-	-	1	1	1	-	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-	45	MDR ← M[MAR]
	45	01110	0101	-	0	-	000	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-	46	MDR ← MDR + CNT
	46	-	-	1	1	-	-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-	47	M[MAR] ← MDR
	47	-	0100	-	0	-	011	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-	48	PC ← Tmp

Als nächstes wird die aktuelle Speicherzelle temporär zwischen gespeichert. Auf die maximale Datenlänge des Pakets wird nun das zweifache des Kanals addiert, da für jeden Kanal zwei Speicherzellen benötigt werden. In der ersten Speicherzelle wird die Kanalnummer abgelegt, in der zweiten die Anzahl der Datenbits. Anschließen wird die Speicheradresse wieder auf ihren

ürsprunglichen Wert zurückgesetzt.

7.6 Offset-Manipulation

offset>16?	48	10000	1011	-	0	-	001	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	-	49	OffsetKanal ← 32 - OffsetData OffsetData ← 32 - OffsetData
	49	01000	-	-	0	-	010	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	-	50	Dec ← 15
	50	-	1101	-	0	-	011	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1 0	55 51	Dec == 0?
	51	00001	1101	-	0	-	001	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	-	52	Dec ← Dec - 1
	52	-	1010	-	0	-	011	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1 0	54 53	OffsetData == 0?
	53	00001	1010	-	0	-	001	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	-	50	OffsetData ← OffsetData - 1
	54	00000	1011	-	0	-	010	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	-	7	OffsetData ← 0
	55	10001	-	-	0	-	110	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	-	56	OffsetData ← ¬ OffsetKanal
	56	00001	1010	-	0	-	000	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	-	57	OffsetData ← 1 + OffsetData
	57	00001	-	-	0	-	010	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	-	7	flag ← 1

War das Offset beim Entdecken des Startbismuster größer gleich 16, so führt dies während "Datenanfang ermitteln" zu einem Offset, das größer ist als die Speicherzelle selber. Anstelle dessen wird das Offset bei Werten größer gleich 16 um 32 verringert und ein Flag, was das Überspringen einer Speicherzelle zufolge hat.

8 Bewertung der Lösung

Wir führen die Berechnung unseres Algorithmus durch, um ein Vergleichswert für andere Lösungen zu haben. Zur Realisierung benötigten wir:

- **reg (Anzahl der ergänzten Register):** 9
- **const (Anzahl der ergänzten Konstanten):** 6
- **alu_add (Anzahl der ergänzten ALU-Befehle):** 17
- **alu_use (Anzahl der verwendeten ALU-Befehle):** 58

$t_{bewertet} = t_{bench} * (1 + 0,1 * reg + 0,015 * alu\_add + 0,05 * const)$
$n_{bewertet} = n_{algorithmus} + 5 * reg + 3 * alu\_use + 5 * const$

8.1 Laufzeit

8.1.1 Benchmark 1

$t_{bewertet} = 12365 * (1 + 0,1 * 9 + 0,015 * 17 + 0,05 * 6) = 30356,075$

8.1.2 Benchmark 2

$t_{bewertet} = 31966 * (1 + 0,1 * 9 + 0,015 * 17 + 0,05 * 6) = 78476,530$

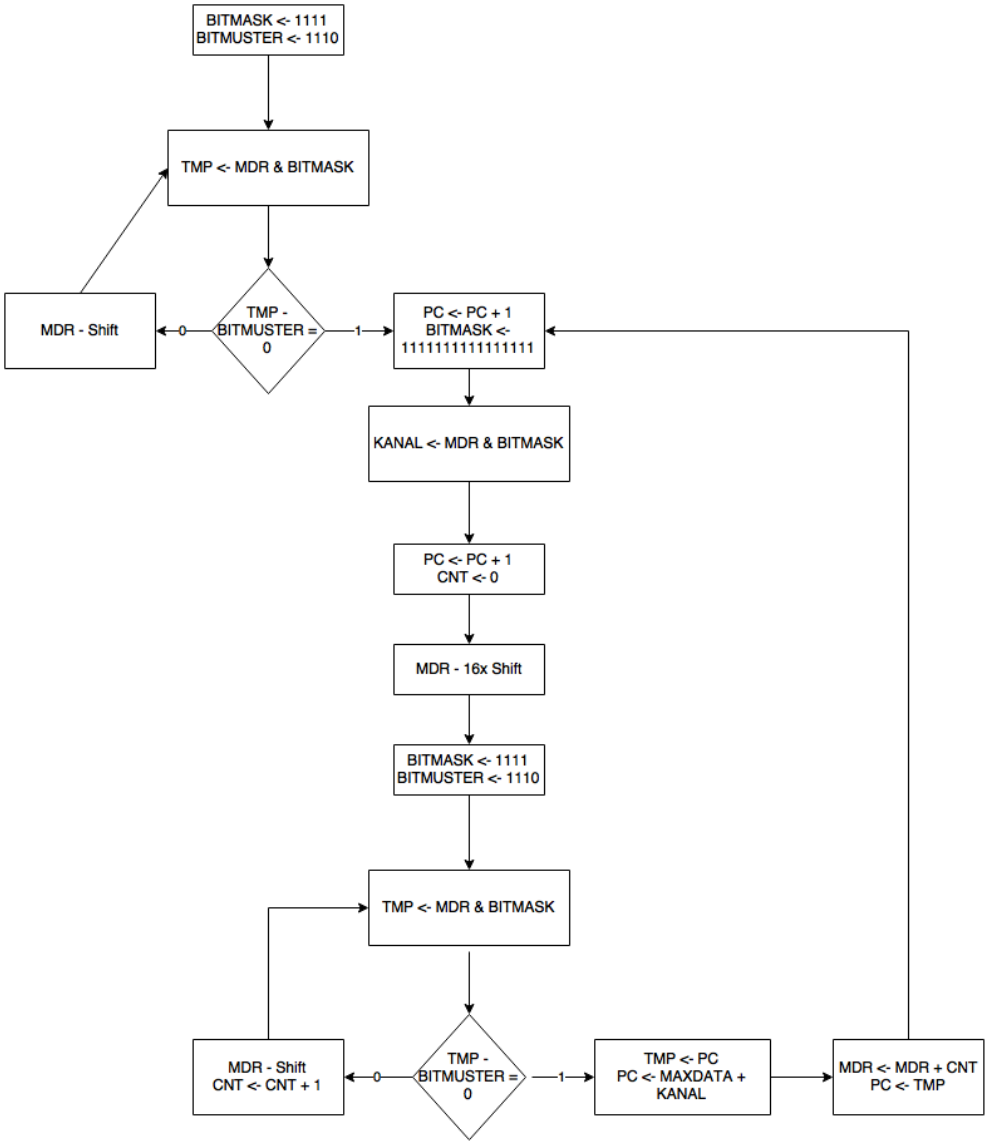
8.1.3 Benchmark 3

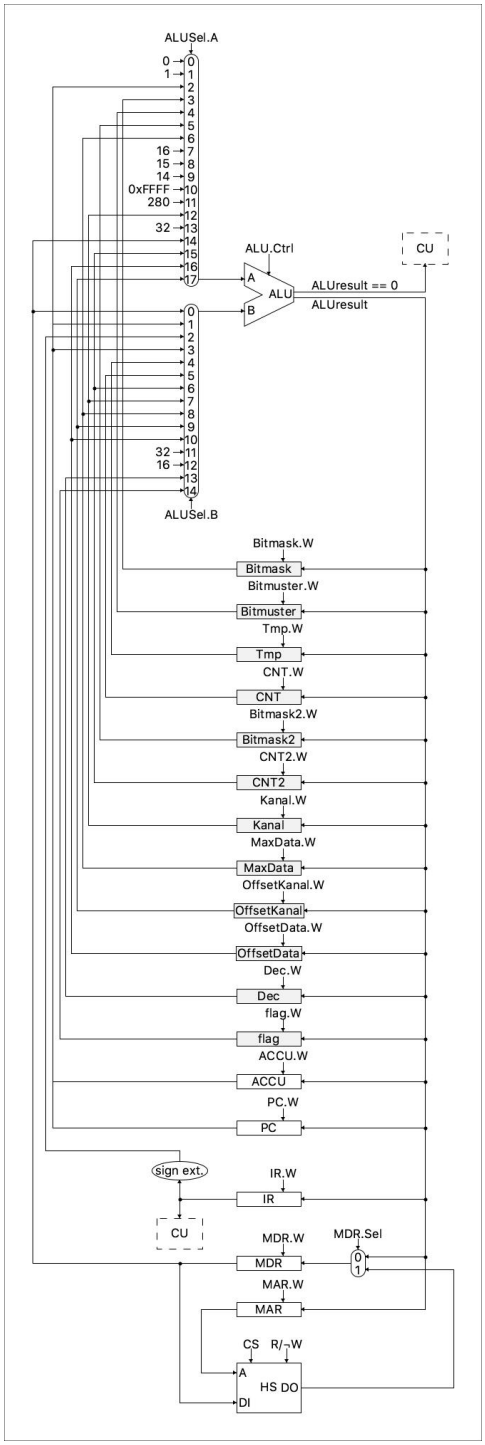
$t_{bewertet} = 38510 * (1 + 0,1 * 9 + 0,015 * 17 + 0,05 * 6) = 94542,050$

8.2 Algorithmuslänge

$n_{bewertet} = 59 + 5 * 9 + 3 * 58 + 5 * 6 = 308$

9 Anhang: Flussdiagramme, Maschinenübersicht




ALUSel.A
0
1
16
15
14
0xFFFF
280
32
ALU.Ctrl
CU
A
B
ALU
ALUresult == 0
ALUresult
32
16
ALUSel.B
Bitmask.W
Bitmask
Bitmuster.W
Bitmuster
Tmp.W
Tmp
CNT.W
CNT
Bitmask2.W
Bitmask2
CNT2.W
CNT2
Kanal.W
Kanal
MaxData.W
MaxData
OffsetKanal.W
OffsetKanal
OffsetData.W
OffsetData
Dec.W
Dec
flag.W
flag
ACCU.W
ACCU
PC.W
PC
sign ext.
IR.W
IR
CU
MDR.W
MDR.Sel
MDR
MAR.W
MAR
CS
R/¬W
A
HS DO
DI

10 Anhang: Verwendete Hilfsmittel

- Vorlesung: Grundlagen der Rechnerarchitektur
- Umdruck Paketanalyse 2016/17
- GitHub zur Versionskontrolle und Organisation
- https://www.draw.io zur Erstellung der Flussdiagramme