Praktikum Rechnerstrukturen

3

Mikroprogammierung II

Name, Vorname	
Bogen bearbeitet	abzeichnen lassen

Universität Hamburg – MIN – Fachbereich Informatik
Arbeitsbereich Technische Aspekte Multimodaler Systeme

https://tams.informatik.uni-hamburg.de

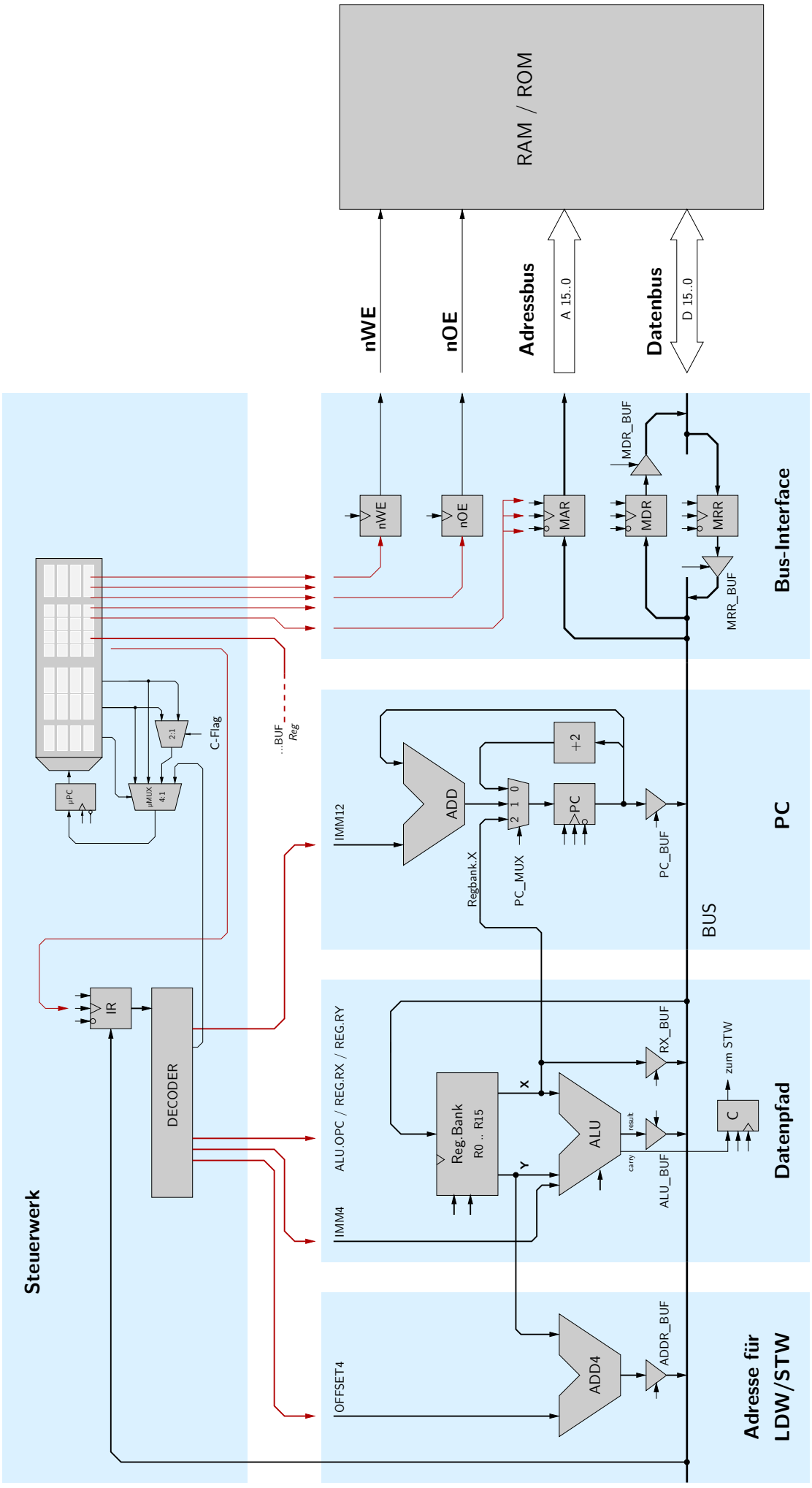

Abb. 1: Blockschaltbild des D·CORE Prozessors

Ziel der Versuche dieses Bogens ist es, die Mikroprogramme für die noch fehlenden Befehle des D·CORE Prozessors nach Tabelle 1 aus Bogen 1 zu implementieren. Dazu wird wieder der vollständige Prozessor (**`processor.hds`**) und das bisher von Ihnen entwickelte Mikroprogramm benötigt.

1 Lade- und Speicherbefehle

Aufgabe 3.1 Load-Befehl

Der Befehl `LDW` (*load word*) dient dazu, Datenwerte aus dem Speicher in ein Register zu übertragen. Als Pseudocode formuliert lautet der Ladebefehl des D·CORE Prozessors:
`R[x] = MEM(R[y] +` ⟨*cccc*⟩`<<1)` mit einer 4-bit Konstanten ⟨*cccc*⟩, die aus den Bits <11:8> des Befehlswortes gespeist wird (vgl. Bogen 1, Tabelle 1). Über das Feld ⟨*xxxx*⟩ im Befehlswort wird das Zielregister `RX` des `LDW`-Befehls ausgewählt. Als Basisadresse dient der Inhalt des Registers `RY`.

Im D·CORE werden, wie bei fast allen RISC-Architekturen, die noch freien Bits im Befehlswort des `LDW`-Befehls ausgenutzt, um einen vier Bit Offset zu dem Inhalt von `RY` zu addieren. Dies erleichtert unter anderem den indizierten Zugriff auf die Elemente in zusammengesetzten Datentypen (etwa eine C `struct`). Zur Adressberechnung aus Basisadresse und Offset dient dabei ein eigener Addierer — im Schaltbild des D·CORE liegt dieser ganz links im Operationswerk (vgl. Abbildung 1).

Ein Beispiel für die Adressberechnung ist in Abbildung 2 für eine einfache `struct Point3D` mit drei Elementen `x`, `y` und `z` dargestellt. Register `R2` und `R4` dienen dabei als Pointer auf zwei dieser Strukturen. Mit Hilfe des Offsets bei der Adressierung ist es jetzt möglich, direkt auf (bis zu 16) Elemente innerhalb der Strukturen zuzugreifen, ohne die Adresse separat berechnen zu müssen. Zum Beispiel laden die Befehle `ldw R6,2(R4)` und `ldw R5,4(R4)` direkt die Werte von `target.y` und `target.z` in die Register `R6` und `R5`.

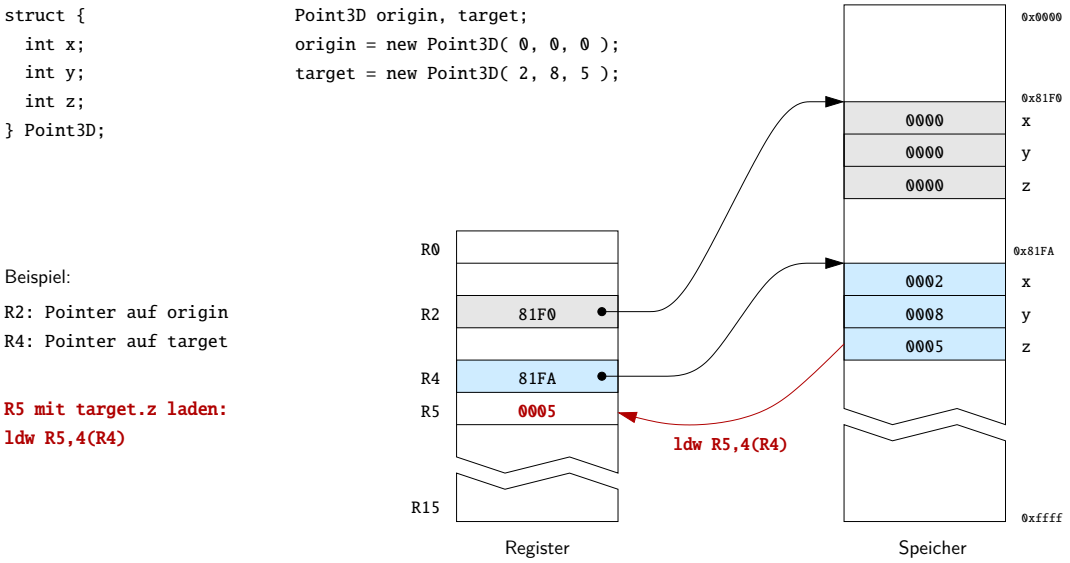

Abb. 2: Adressierung mit Basisadresse und Offset zum direkten Zugriff auf Elemente zusammengesetzter Datentypen

Für den eigentlichen Speicherzugriff ist das gleiche komplizierte Timing erforderlich wie in der Befehl-Holen Phase (siehe Abbildung 10 in Aufgabenblatt 2). Implementieren Sie den `LDW`-Befehl und schreiben Sie ein kleines Testprogramm, um die Funktion zu demonstrieren. Tragen Sie den Microcode zusätzlich in die folgende Tabelle ein:

addr		nextA	nextB	µPCmux.S1	µPCmux.S0	RXBUF	AX=15	IR	ADDRBUF	C	ALUBUF	REGS.nWE	PCBUF	PC	PCMUX.s1	PCMUX.s0	MRRBUF	MRR	MDR	MDRBUF	MAR	nWE	nOE	annotation
		00	00	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	inactive
		00	00	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	inactive

Aufgabe bearbeitet	abzeichnen lassen

Aufgabe 3.2 Store-Befehl

Mit dem Befehl `STW` (*store word*) können Registerinhalte in den Speicher übertragen werden. Auch für `STW` verwendet die D·CORE-Architektur die bereits bei `LDW` erläuterte Adressierung `MEM(R[y] +` $\langle cccc \rangle$`<<1) = R[x]` mit einem Basisregister `RY` und einem positiven Offset $\langle cccc \rangle$.

Die notwendige Ansteuerung des Speicherinterface ist in Abbildung 6 von Aufgabenblatt 2 dargestellt. Zunächst wird das `MAR`-Register mit der Adresse geladen. Diese muss während des gesamten Schreibzyklus unverändert bleiben. Einen Takt danach wird das Write-Enable Signal aktiviert (`nWE` ist low-active!). Gleichzeitig werden die zu schreibenden Daten aus der Registerbank in das Register `MDR` übertragen (nutzen Sie dazu den direkten Datenpfad über den `RXBUF` Treiber). Danach muss der Ausgangstreiber hinter dem `MDR`-Register aktiviert werden, um die Daten aus `MDR` auf den Datenbus zu legen. Sobald die Daten auf dem Datenbus liegen, werden Wartezyklen eingefügt, um die Zugriffszeit des Speichers einzuhalten. Schließlich wird das `nWE`-Signal deaktiviert (auf 1) gesetzt, wobei der Speicher die aktuellen Daten übernimmt. Im nächsten Takt wird der Treiber hinter `MDR` wieder deaktivert, um den Datenbus für nachfolgende Datenübertragungen frei zu machen. Notieren Sie Ihren Microcode zur Dokumentation in folgender Tabelle:

addr		nextA	nextB	µPCmux.S1	µPCmux.S0			RXBUF	AX=15	IR	ADDRBUF	C	ALUBUF	REGS.nWE	PCBUF	PC	PCMUX.s1	PCMUX.s0	MRRBUF	MRR	MDR	MDRBUF	MAR	nWE	nOE	annotation
		00	00	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	inactive
		00	00	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	inactive

Schreiben Sie jetzt ein Testprogramm, das mit möglichst wenig Anweisungen die ersten vier Befehle ihres Programms aus dem ROM in das RAM kopiert. Nutzen Sie hierfür die in Abbildung 2 demonstrierte indizierte Adressierung.

Label	Adresse	Befehlscode	Mnemonic	Kommentar
`testLDWSTW:`	`0000`	`0x3482`	`movi R2, 8`	`R[2]=8`
	`0002`			
	`0004`			
	`0006`			
	`0008`			
	`000a`			
	`000c`			
	`000e`			
	`0010`			
	`0012`			
	`0014`			
	`0016`			
	`0018`			
	`001a`			
	`001c`			
	`001e`			

Aufgabe bearbeitet	abzeichnen lassen

2 Sprungbefehle

Sprungbefehle sind ein essentieller Bestandteil aller von-Neumann Rechner, um die sequentielle Abarbeitung der Befehle unterbrechen und beeinflussen zu können. Alle Kontrollstrukturen wie Blöcke, Bedingungen, Schleifen und Unterprogrammaufrufe werden auf der Ebene der Maschinensprache mit Sprungbefehlen realisiert, die direkt den Programmzähler PC modifizieren. Die D·CORE-Architektur definiert die folgenden Sprungbefehle (vergleiche Tabelle 1 im Aufgabenblatt 1):

Mnemonic	Codierung	Hex	Bedeutung
br	1000 ⟨*iiii*⟩ ⟨*iiii*⟩ ⟨*iiii*⟩	8iii	PC = PC+2+⟨*imm12*⟩
jsr	1001 ⟨*iiii*⟩ ⟨*iiii*⟩ ⟨*iiii*⟩	9iii	R[15] = PC+2; PC = PC+2+⟨*imm12*⟩ (call)
bt	1010 ⟨*iiii*⟩ ⟨*iiii*⟩ ⟨*iiii*⟩	Aiii	(C=1) ? PC = PC+2+⟨*imm12*⟩ : PC=PC+2
bf	1011 ⟨*iiii*⟩ ⟨*iiii*⟩ ⟨*iiii*⟩	Biii	(C=0) ? PC = PC+2+⟨*imm12*⟩ : PC=PC+2
jmp	1100 ⟨****⟩ ⟨****⟩ ⟨*xxxx*⟩	C**x	PC = R[x]

Auf den ersten Blick mag die Definition dieser Befehle ungewöhnlich erscheinen. Aber wie in Aufgabenblatt 1 angedeutet, verwenden die meisten Rechnerarchitekturen eine Byte-Adressierung des Speichers. Für den D·CORE muss daher der PC nach jedem Befehl um den Wert 2 inkrementiert werden, um das nächste Befehlswort zu adressieren. Mit der Konvention, dass der PC für jeden Befehl bereits in der Decode-Phase inkrementiert wird, ist auch die Berechnung der Sprungadressen für die relativen Sprünge verständlich: erst wird der PC in der Decode-Phase inkrementiert, dann wird in der Execute-Phase noch eine (sign-extended) 12-bit Konstante aus dem Befehlswort zum Wert des PC addiert.

Die notwendige Hardware für die Realisierung der Sprungbefehle ist in Abbildung 3 skizziert. Ein Inkrementierer (um den Wert 2) sowie ein separater Addierer sorgen für die ständige Berechnung der Werte `(PC + 2)` und `(PC + sign_extend(IR.<11:0>))`. Über den Multiplexer vor dem Dateneingang des PC erfolgt die Auswahl, welcher dieser Werte in den PC geladen wird. Sie finden diese Komponenten auch einzeln im Hades-Design **`next-pc.hds`**.

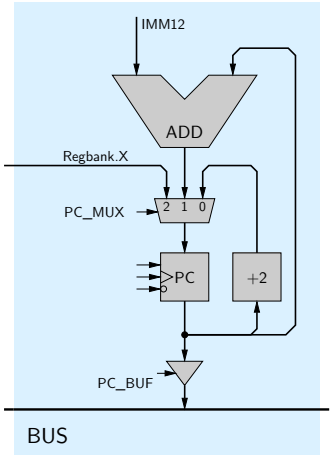

Abb. 3: Realisierung der Sprungbefehle: Über den Multiplexer werden die Werte PC+2, PC+IMM12 oder RX ausgewählt und in den PC geladen.

Aufgabe 3.3 Jump-Befehl

Der `JMP`-Befehl (*jump*) dient dazu, einen *absoluten Sprung* an eine bestimmte absolute Adresse durchzuführen, wobei der Wert des `PC` aus einem Register der Registerbank stammt. Erweitern Sie das Mikroprogramm um den `JMP`-Befehl:

addr		nextA	nextB	µPCmux.s1	µPCmux.s0	RXBUF	AX=15	IR	ADDRBUF	C	ALUBUF	REGS.nWE	PCBUF	PC	PCMUX.s1	PCMUX.s0	MRRBUF	MRR	MDR	MDRBUF	MAR	nWE	nOE	annotation
–		00	00	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	inactive

Erstellen Sie ein kurzes Programm `test-jmp.rom`, um den Befehl zu testen. Inkrementieren Sie zum Beispiel den Wert von `R3` in einer Endlosschleife:

Label	Adresse	Befehlscode	Mnemonic	Kommentar
`testJMP:`	`0000`	`0x3403`	`movi R3, 0`	`R[3]=0`
	`0002`			`R[2]=⟨????⟩`
`loop:`	`0004`			`R[3]++`
	`0006`			`jmp R[2] (goto loop)`
	`0008`			

Aufgabe bearbeitet	abzeichnen lassen

Aufgabe 3.4 Branch-Befehl

Mit dem `BR`-Befehl (*branch*) werden *relative Sprünge* realisiert, bei denen sich die Zieladresse aus dem aktuellen Wert des `PC` und einem Offset ergibt. Der 12-bit Offset aus dem Befehlswort wird dabei als Zweierkomplement interpretiert und mit Vorzeichen auf 16-bit erweitert (aus `0x123` wird also `0x0123`, aus `0xffc` bzw. $(-4)_{10}$ entsprechend `0xfffc`), damit der `PC` beim Sprung auch verkleinert werden kann. Das wird zum Beispiel bei Schleifen benötigt, wenn der Test der Schleifenbedingung am Ende der Schleife durchgeführt wird, also gegebenenfalls ein Rücksprung erfolgt.

Vervollständigen Sie zunächst folgende Tabelle. Alle Angaben sind hexadezimal zu verstehen:

alter `PC`	`br`-Befehl	neuer `PC`
`0100`	`8008`	
`010c`	`8ff6`	
`0020`		`0042`
`0020`		`001a`

Realisieren Sie jetzt den Mikrocode für den BR-Befehl:

addr		nextA	nextB	µPCmux.s1	µPCmux.s0	RXBUF	AX=15	IR	ADDRBUF	C	ALUBUF	REGS.nWE	PCBUF	PC	PCMUX.s1	PCMUX.s0	MRRBUF	MRR	MDR	MDRBUF	MAR	nWE	nOE	annotation
–		00	00	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	inactive

Schreiben Sie zum Test ein neues Programm `test-br-clear-ram`, das in einer Endlosschleife das gesamte RAM ab Adresse `0x8000` löscht:

Label	Adresse	Befehlscode	Mnemonic	Kommentar
`testBR:`	`0000`	`0x3480`	`movi R0, 8`	`R[0]=8`
	`0002`			
	`0004`			
	`0006`			
	`0008`			
	`000a`			
	`000c`			
	`000e`			
	`0010`			
	`0012`			
	`0014`			
	`0016`			
	`0018`			
	`001a`			
	`001c`			
	`001e`			

Aufgabe bearbeitet	abzeichnen lassen

2.1 Bedingte Sprünge

Programmverzweigungen erfordern bedingte Sprungbefehle. Der Befehlssatz des D·CORE verfügt hierfür über die beiden Befehle `BT` (*branch if true*) und `BF` (*branch if false*), wobei der Wert des `C`-Registers für die Bedingung ausgewertet wird. Dazu ist der Ausgang des `C`-Registers mit dem Select-Eingang des 2:1-Multiplexers im Steuerwerk verschaltet, was im Mikroprogramm die Auswahl von μ`ROM.nextA` (C= **0**) oder μ`ROM.nextB` (C= **1**) als Folgeadresse erlaubt. Dies ist auch in Abbildung 1 skizziert und Sie haben diese Verwendung des `C`-Registers bereits in den Aufgaben **??** und 2.3 des zweiten Aufgabenblattes kennengelernt. Im Grunde müssen Sie also zur Implementation der Befehle `BF` und `BT` nur noch den 4:1-Multiplexer richtig ansteuern sowie μ`ROM.nextA` und μ`ROM.nextB` geeignet setzen.

Auf der Assembler-Ebene muss vor einem bedingten Sprung natürlich das `C`-Register z.B. durch einen Vergleichsbefehl entsprechend gesetzt werden. Da das `C`-Register anders als bei den meisten älteren Architekturen aber nicht von allen ALU-Befehlen beeinflusst wird, muss der Vergleichsbefehl nicht unbedingt direkt vor dem Sprungbefehl stehen.

Realisieren Sie den `BT`- und den `BF`-Befehl im Mikroprogramm. Je nach Realisierung wird die zweite Zeile nicht zwangsläufig benötigt.

addr		nextA	nextB	µPCmux.S1	µPCmux.S0	RXBUF	AX=15	IR	ADDRBUF	C	ALUBUF	REGS.nWE	PCBUF	PC	PCMUX.s1	PCMUX.s0	MRRBUF	MRR	MDR	MDRBUF	MAR	nWE	nOE	annotation
		00	00	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	inactive

addr		nextA	nextB	µPCmux.S1	µPCmux.S0	RXBUF	AX=15	IR	ADDRBUF	C	ALUBUF	REGS.nWE	PCBUF	PC	PCMUX.s1	PCMUX.s0	MRRBUF	MRR	MDR	MDRBUF	MAR	nWE	nOE	annotation
		00	00	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	inactive

Mit erfolgreich implementierten `BT`- bzw. `BF`-Befehlen lassen sich jetzt auf dem D·CORE auch Schleifen mit einer Abbruchbedingung ausführen.

Das Programm der folgenden Aufgabe demonstriert neben der Umsetzung einer `while`-Schleife auch noch die indizierte Adressierung für den Zugriff auf Arrays.

Aufgabe 3.5 While-Schleife

Schreiben Sie ein Programm, um ein Array (Feld) mit n Elementen auf die Werte 0...n-1 vorzubesetzen. Das Feld soll ab der Adresse `base` im Speicher liegen. Hier ein C-Pseudocode für das Programm:

```
int length = 5;
int base[] = 0x8010;  // Startadresse

int i = 0;
do {
  base[i] = i;
  i++;
} while( i < length );
```

Label	Adresse	Befehlscode	Mnemonic	Kommentar
testWhile:	0000	0x3480	movi R0, 8	R[0]=8
	0002			
	0004			
	0006			
	0008			
	000a			
	000c			
	000e			
	0010			
	0012			
	0014			
	0016			
	0018			
	001a			
	001c			
	001e			

Testen Sie Ihr Programm auf Ihrem Prozessor.

Aufgabe bearbeitet	abzeichnen lassen

2.2 Unterprogrammaufrufe

Üblicherweise stellen Prozessoren, wie auch der D·CORE, spezielle Maschinenbefehle für Unterprogrammaufrufe, wie JSR/RET, zur Verfügung.

Der Maschinenbefehl **JSR**
Die Abkürzung `JSR` steht für *Jump to Subroutine*. Der eigentliche Sprung erfolgt genau wie beim `BR`-Befehl; allerdings wird der aktuelle Wert des `PC` vorher im Register `R15` abgespeichert. Für diesen ersten Schritt des `JSR`-Befehls ist zusätzliche Logik im Prozessor erforderlich, da die Schreib-Adresse der Registerbank für alle anderen Befehle direkt aus dem Befehlsregister kommt, hier aber fest auf den Wert 15 gesetzt werden muss. Dies erledigt ein kleiner Block von OR-Gattern (Komponente `AX-or-15`), der zwischen Befehlsdecoder und die Schreibadresse `AZ` der Registerbank gesetzt ist und über die Steuerleitung `ax=15` aus dem Mikroprogramm aktiviert wird. Da das Abspeichern des `PC` erfolgt, nachdem dieser in der Decode-Phase bereits um 2 inkrementiert wurde, zeigt Register `R15` nach einem `JSR` direkt auf den nach einem Rücksprung auszuführenden Befehl.

Aufgabe 3.6 Implementierung und Test der Unterprogrammunterstützung

Erweitern Sie Ihr Mikroprogramm um den letzten noch fehlenden Befehl `JSR`:

addr		nextA	nextB	µPCmux.S1	µPCmux.S0	RXBUF	AX=15	IR	ADDRBUF	C	ALUBUF	REGS.nWE	PCBUF	PC	PCMUX.s1	PCMUX.s0	MRRBUF	MRR	MDR	MDRBUF	MAR	nWE	nOE	annotation
		00	00	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	inactive

Begründen Sie, warum die D·CORE-Architektur im Gegensatz zu dem o. a. Befehlspärchen (JSR/RET) keinen expliziten Return-Befehl (return from subroutine) bereitstellt.

In Bogen 2 Aufgabe 2.3 hatten wir ein Mikroprogramm geschrieben, das das Quadrat einer positiven Zahl berechnen kann. Dieses lässt sich mühelos in ein Assembler-Programm umsetzen. Um Ihnen das Leben etwas zu erleichtern, haben wir Ihnen diese Aufgabe bereits abgenommen. Der Code für ein entsprechendes Unterprogramm steht in der Datei `Quadrat.rom`, die sich in das ROM unseres Prozessors laden lässt.

Die Startadresse liegt dabei auf `0x0040`. Als Eingabe erwartet das Unterprogramm die Zahl, deren Quadrat berechnet werden soll, im Register `R4`. Das Ergebnis wird dann im Register `R5` zurückgeliefert. Weiterhin wird das Register `R6` modifiziert.

Erweitern Sie den gegebenen Code (**`Quadrat.rom`**) um ein Hauptprogramm, das das Unterprogramm zur Berechnung von a^4 für eine gegebene (kleine!) Zahl a nutzt.

Label	Adresse	Befehlscode	Mnemonic	Kommentar
	0000			
	0002			
	0004			
	0006			
	0008			
	000a			
⋮	⋮	⋮	⋮	⋮
	003e			
quadrat:	0040	0x3405	movi R5, 0	
	0042	0x3406	movi R6, 0	
	0044	0x3046	cmpe R6,R4	
⋮	⋮	⋮	⋮	⋮

Aufgabe bearbeitet	abzeichnen lassen

Aufgabe 3.7 Ein letzter Test

Laden Sie jetzt die vorgegebene Datei `bigtest.rom` in das ROM und starten Sie den Prozessor. Das Testprogramm überprüft noch einmal alle bisher vorhandenen Befehle (ALU, Immediate, Compare, Load, Store, Jump, Branch, Jump to Subroutine, Halt). Wenn alles funktioniert, schreibt das Programm den Wert `0xaffe` in das Register `R7`.

Aufgabe bearbeitet	abzeichnen lassen

3 Zusammenfassung

Machen Sie sich noch einmal die folgenden Punkte klar:

- das Modell aufeinander aufbauender, zunehmend abstrakterer Schichten zur Beschreibung (und zum Verständnis) eines Computersystems — von der Algorithmenebene über die logische Ebene bis hinunter zur physikalischen Ebene
- den grundlegenden Aufbau eines von-Neumann-Rechners mit Steuerwerk, Operationswerk mit Registern und ALU, dem Speicher und den I/O-Komponenten
- den Befehlszyklus mit den Phasen *fetch*, *decode* und *execute*

- Alle Rechenwerke des System sind jederzeit aktiv und berechnen ununterbrochen Ausgangswerte. Aber von all diesen Werten werden nur die für den aktuellen Befehl benötigten Ergebnisse mit der nächsten Taktflanke abgespeichert.
- Mikroprogrammierung als direkte Umsetzung von endlichen Automaten in Hardware
- die Trennung zwischen Befehlsarchitektur (z.B. x86), die für den Programmierer sichtbar ist, und der Struktur des Rechners (z.B. Core-i.. als RISC-Registermaschine)
- Speicherzugriffe und I/O sind langsame Operationen. Cache-Speicher dienen dazu, die Zugriffszeiten zu verstecken.
- die Adressierung mit Basisadresse und Offset als effiziente Möglichkeit zum Zugriff auf zusammengesetzte Datentypen