

Praktikum Rechnerstrukturen

WiSe2023/2024

3

Mikroprogrammierung II

Name, Vorname	
Bogen bearbeitet	abzeichnen lassen

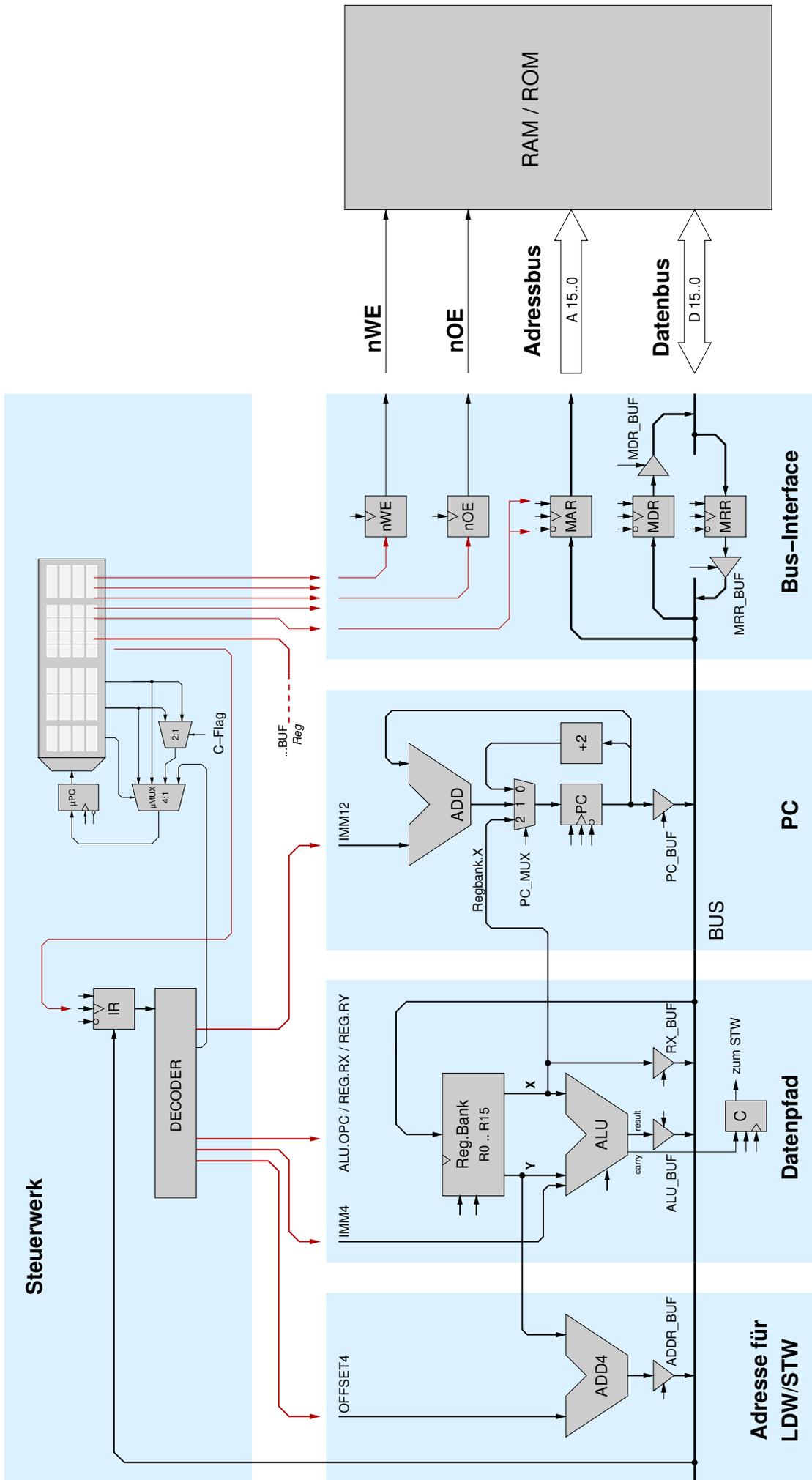


Abbildung 1: Blockschaltbild des D-CORE Prozessors

3 - Mikroprogrammierung II

Der Bogen 2 dieses Praktikums endete mit den ersten Aufgaben zur Implementierung des D-CORE Prozessors. So haben Sie bereits die Fetch-, die Decode-Phase und die ersten Befehle der Execute-Phase implementiert.

Ziel dieses Bogen 3 ist es, die Mikroprogramme für die noch fehlenden Befehle des Prozessors entsprechend Tabelle 1 aus Bogen 1 zu implementieren und zum Test des Prozessors kurze Maschinenprogramme zu schreiben.

3-7.3 Implementierung des D-CORE Prozessors (Fortsetzung)

Für die Fortführung der Implementierung der noch fehlenden Befehle des Prozessors benötigen Sie wieder den vollständigen Prozessor (**processor.hds**) und das von Ihnen **bisher entwickelte Mikroprogramm**, das Sie nun vervollständigen werden.

3-7.3.3 Lade- und Speicherbefehle

Im Aufgabenblatt 2 im Abschnitt 6 haben Sie bereits die Grundprinzipien der Ansteuerung des externen Speichers kennengelernt und in Aufgabe 2.4 Werte in den Speicher schreiben und ausgelesen. Implementieren Sie hierauf aufbauend, wie bereits bei der Implementierung der Befehl-Holen-Phase (siehe Aufg. 2.5) geschehen, den Load- und Store-Befehl des Prozessors.

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:

$R[x] = \text{MEM}(R[y] + \langle \text{cccc} \rangle \ll 1)$ mit einer 4-bit Konstanten $\langle \text{cccc} \rangle$, die aus den Bits $\langle 11:8 \rangle$ des Befehlswortes gespeist wird (vgl. Bogen 1, Tabelle 1). Über das Feld $\langle \text{xxxx} \rangle$ im Befehlswort wird das Zielregister RX des LDW-Befehls ausgewählt. Das Register RY enthält die Adresse, die für den Speicherzugriff benötigt wird.

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).

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 $MEM(R[y] + \langle cccc \rangle \ll 1) = R[x]$ mit einem Basisregister *RY* und einem positiven Offset $\langle cccc \rangle$.

Die notwendige Ansteuerung des Speicherinterface ist in Abbildung 9 des Aufgabenblattes 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 in das *WE*-Flipflop geladen (Achtung, das Signal *nWE* der Speicherbausteine 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 dem *MDR* auf den Datenbus des Speichers zu legen. Sobald die Daten auf dem Bus und damit am Speicher anliegen, werden Wartezyklen eingefügt, um die Zugriffszeit des Speichers einzuhalten. Schließlich wird das *nWE*-Signal deaktiviert (auf 1) gesetzt, wobei der Speicher mit der steigenden Flanke des *nWE*-Signals die aktuellen Daten übernimmt. Im nächsten Takt wird der Treiber hinter *MDR* wieder deaktiviert, um den Datenbus am Speicher für nachfolgende Datenübertragungen frei zu machen. Notieren Sie Ihren Microcode zur Dokumentation in folgender Tabelle:

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

Schreiben Sie jetzt ein Testprogramm, das mit möglichst wenigen Anweisungen die ersten vier Befehle des gerade im ROM stehenden Programms auf die ersten vier Adressen des RAMs kopiert. Nutzen Sie hierfür die in Abbildung 2 demonstrierte indizierte Adressierung.

Die erste Anweisung der Tabelle 1 demonstriert das zu verwendende Format. Diese Anweisung könnte beispielsweise für die Generierung der Schreibadresse (0x8000) in Register 2 verwendet werden.

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			

Tabelle 1: Testprogramm: Kopieren der ersten vier Speicherworte aus dem ROM in das RAM

Aufgabe bearbeitet	abzeichnen lassen
--------------------	-------------------

3-7.3.4 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 (vgl. Tabelle 1 im Aufgabenblatt 1):

Mnemonic	Codierung	Hex	Bedeutung
br	1000 <iii> <iii> <iii>	8iii	$PC = PC+2 + \langle imm12 \rangle$
jsr	1001 <iii> <iii> <iii>	9iii	$R[15] = PC+2; PC = PC+2 + \langle imm12 \rangle$ (call)
bt	1010 <iii> <iii> <iii>	Aiii	$(C=1) ? PC = PC+2 + \langle imm12 \rangle : PC=PC+2$
bf	1011 <iii> <iii> <iii>	Biii	$(C=0) ? PC = PC+2 + \langle imm12 \rangle : PC=PC+2$
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 mit seiner 16-bit Speicherwortbreite 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**.

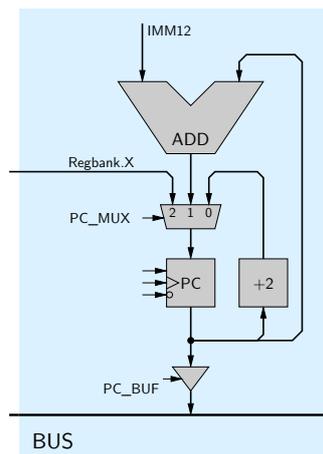


Abbildung 3: Realisierung der Sprungbefehle: Über den Multiplexer werden die Werte PC+2, PC+IMM12 oder RX (Regbank_X) ausgewählt und in den PC geladen.

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	hPCmux.s1	hPCmux.s0	nextA	nextB	RXBUF	AX=15	IR	ADDRBUF	C	ALUBUF	REGS.nWE	PCBUF	PC	PCMUX.s1	PCMUX.s0	MRRBUF	MRR	MIDR	MDRBUF	MAR	nWE	nOE	annotation	
-	0	0	00	00	0	0	0	0	0	0	0	0	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			

Tabelle 2: Testprogramm: Schleife mit Jump-Befehl

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 `0xffffc`), 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	μ PCmux.s1	μ PCmux.s0	nextA	nextB	RXBUF	AX=15	IR	ADDRBUF	C	ALUBUF	REGS.nWE	PCBUF	PC	PCMUX.s1	PCMUX.s0	MRRBUF	MRR	MDR	MDRBUF	MAR	nWE	nOE	annotation
-	0	0	00	00	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	inactive

Schreiben Sie zum Test ein neues Programm `test-br-clear-ram`, das in einer Endlosschleife das gesamte RAM ab Adresse `0x8000` löscht. Eine Abbruchbedingung nach Löschen des gesamten RAMs brauchen Sie nicht zu implementieren:

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			

Tabelle 3: Testprogramm: clear RAM

Aufgabe bearbeitet	abzeichnen lassen
--------------------	-------------------

3-7.3.5 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 $\mu\text{ROM.nextA}$ ($C=0$) oder $\mu\text{ROM.nextB}$ ($C=1$) als Folgeadresse erlaubt. Dies ist auch in Abbildung 1 skizziert und Sie haben diese Verwendung des C-Registers bereits im Kapitel 5.1 und der Aufgabe 2.3 des Aufgabenblattes 2 kennengelernt. Im Grunde müssen Sie also zur Implementierung der Befehle BF und BT nur noch den 4:1-Multiplexer richtig ansteuern sowie $\mu\text{ROM.nextA}$ und $\mu\text{ROM.nextB}$ 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 auch 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	$\mu\text{PCmux.s1}$	$\mu\text{PCmux.s0}$	nextA	nextB	RXBUF	AX=15	IR	ADDRBUF	C	ALUBUF	REGS.nWE	PCBUF	PC	PCMUX.s1	PCMUX.s0	MRRBUF	MRR	MDR	MDRBUF	MAR	nWE	nOE	annotation	
-	0	0	00	00	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	inactive

addr	$\mu\text{PCmux.s1}$	$\mu\text{PCmux.s0}$	nextA	nextB	RXBUF	AX=15	IR	ADDRBUF	C	ALUBUF	REGS.nWE	PCBUF	PC	PCMUX.s1	PCMUX.s0	MRRBUF	MRR	MDR	MDRBUF	MAR	nWE	nOE	annotation	
-	0	0	00	00	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	inactive

Mit den 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 $0 \dots n-1$ vorzubereiten. 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			

Tabelle 4: Testprogramm: Initialisierung eines Array

Testen Sie Ihr Programm auf Ihrem Prozessor.

Aufgabe bearbeitet	abzeichnen lassen
--------------------	-------------------

3-7.3.6 Unterprogrammaufrufe

Üblicherweise stellen Prozessoren, wie auch der D-CORE, spezielle Maschinenbefehle für Unterprogrammaufrufe, wie JSR/RET (Jump SubRoutine/RETurn), 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 Schreibadresse der Registerbank für alle anderen Befehle direkt aus dem Befehlsregister, das Feld RX bzw. die Bits[3..0] des Befehlswortes (siehe Abb. 3 des Bogens 1), kommt, hier aber per Konvention (Register 15 hält die Rücksprungadresse) 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	hPCmux.s1	hPCmux.s0	nextA	nextB	RXBUF AX=15	IR	ADDRBUF C	ALUBUF	REGS.nWE PCBUF	PC	PCMUX.s1	PCMUX.s0	MRRBUF MRR	MDR	MDRBUF MAR	nWE	nOE	annotation	
-	0	0	00	00	0	0	0	0	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 des Unterprogramms liegt dabei auf `0x0040`. Als Eingabe erwartet das Unterprogramm im Register R4 die Zahl, deren Quadrat berechnet werden soll. 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	
⋮	⋮	⋮	⋮	⋮

Tabelle 5: Maschinenprogramm zur Berechnung von a^4

Aufgabe bearbeitet	abzeichnen lassen
--------------------	-------------------

3-7.4 Validierung des vollständigen Prozessors

Aus Zeitgründen wird auf eine gründliche Validierung des Prozessors verzichtet, die selbst bei diesem einfachen Prozessor entschieden zu aufwendig wäre. Stattdessen werden die implementierten Befehle lediglich mit typischen Befehlsfolgen auf ihre Wirkung und ggf. unerwünschte Seiteneffekte getestet.

Aufgabe 3.7 Ein letzter Test des D-CORE Prozessors

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. Falls das nicht geschehen sollte, nutzen Sie das Programm `bigtest` zum Debuggen Ihres Prozessors.

Aufgabe bearbeitet	abzeichnen lassen
--------------------	-------------------

- Wie wurde die 2-stufige Befehlspipeline realisiert? Welche Zusatzhardware ist hierfür erforderlich?
- Welche Veränderungen mussten am bisherigen Prozessor zusätzlich vorgenommen werden, damit das Befehlspipelining überhaupt sinnvoll eingesetzt werden kann? Was bedeutet der Begriff *Harvard-Architektur*?
- Welche Geschwindigkeitsunterschiede der Architekturen sind zu erwarten?
Sie können Ihre Hypothese durch einen abschätzenden Laufzeitvergleich zumindest grob validieren. Verwenden Sie beispielsweise hierfür das Programm `hanoi2mem.rom` aus den Vorlagen unter `programs`, welches das Knobelspiel *Türme von Hanoi* mit drei Scheiben löst. Die Scheibenbewegungen werden ins RAM ab Adresse `0x8000` geschrieben, wobei beispielsweise der Eintrag auf Adresse `0x8000 = 0x1003` bedeutet, eine Scheibe wurde von Stab 1 auf Stab 3 verschoben. Zum Abschluss wird noch die Anzahl der Scheibenbewegungen in den Speicher geschrieben.
Möchten Sie nur die Laufzeitänderung durch die Parallelisierung testen, verwenden Sie bitte das Mikroprogramm `pip2w3.mic`, da hier das Speichermodell (drei Waitstates) des D-CORE sowohl für Data-RAM als auch für das Instruction-ROM verwendet wird.
- Ist durch den identischen Befehlssatz vollständige Binärkompatibilität garantiert, so dass alle Programme unter den beiden Architekturen austauschbar sind? Oder lässt die durch den Befehlssatz des D-CORE (siehe Bogen 1, Tabelle 1) definierte Struktur Spielraum für verschiedene Umsetzungen der ISA (Instruction Set Architecture)? Testen sie beide Architekturvarianten mit dem Testprogramm `bigtestPI2.rom` als auch mit dem bereits bekannten Programm `bigtest.rom`.
- Gibt es im Schematic des `Processor_PI2` noch überflüssige Baugruppen und lässt sich durch Entfernen ggf. noch die Performance steigern?
- Schauen Sie sich das Modell `Processor_PI2_hcCU.hds` an, und vergleichen Sie es mit dem Modell `Processor_PI2.hds` bzw. `Processor_PI2c.hds`.

Aufgabe bearbeitet

abzeichnen lassen

3-8 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