Praktikum Rechnerstrukturen

Bogen 2

Mikroprogrammierung I

Department Informatik, AB TAMS
MIN Fakultät, Universität Hamburg
Vogt-Kölln-Str. 30
D22527 Hamburg

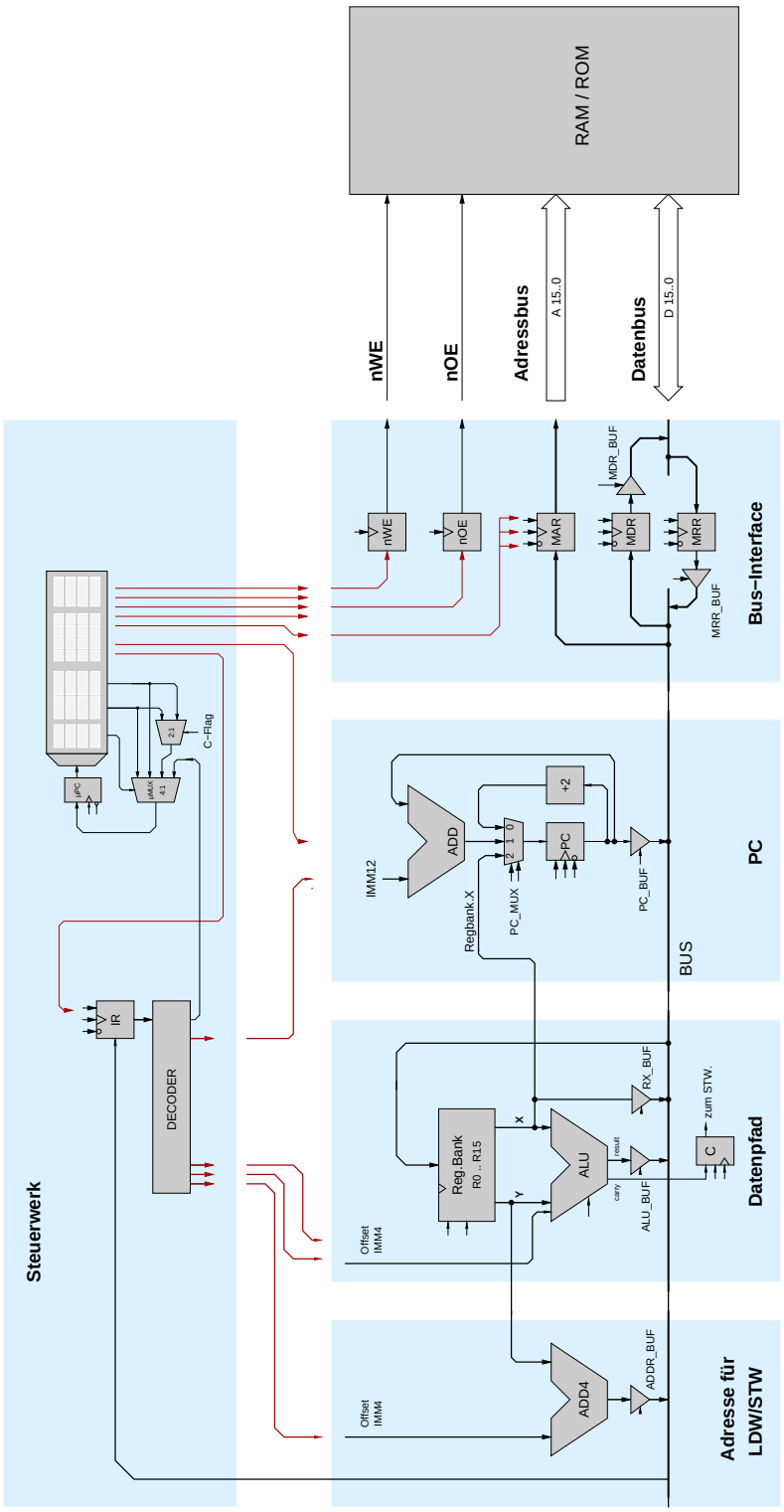


Abbildung 1: Blockschaltbild des D·CORE Prozessors

1 Befehl-Holen

Nachdem wir im ersten Bogen die Komponenten unseres Prozessors einzeln einzeln betrachtet haben, ist es an der Zeit, den kompletten D·CORE-Prozessor zu untersuchen. Dabei ist die Hardwarestruktur mit allen Registern, der ALU und dem Steuerwerk fertig vorgegeben — es fehlt jedoch das Mikroprogramm. Ziel der nächsten Aufgaben ist es, schrittweise ein Mikroprogramm zu erstellen, um einen funktionsfähigen Rechner zu erhalten, der die Befehle in der Tabelle 1 aus Bogen 1 abarbeiten kann.

Aufgabe 1.1: Der D·CORE Prozessor Öffnen Sie das Design `processor.hds` mit der vollständigen Logik des D·CORE-Prozessors inklusive Datenpfad, Speicherinterface, Befehlsdekoder und Programmzähler. Verwenden Sie gegebenenfalls die Zoom-Funktion des Hades-Editors, um die gesamte Schaltung sehen zu können (vergleichen Sie mit Abbildung 1).

Der obere Teil des Schaltplans enthält das Steuerwerk mit dem Befehlsregister `IR`, Mikroprogrammzähler `µPC` und dem Mikroprogramm-ROM. Links liegt der Schalter für den D·CORE-Takteingang, mit dem ein Einzelschrittbetrieb möglich ist. Mit einem zweiten Schalter kann auf den Taktgenerator umgeschaltet werden. Der untere Teil der Schaltung besteht aus dem Operationswerk und dem Speicher. Von links nach rechts finden Sie das Adressrechenwerk, die Registerbank und die ALU, den Programmzähler, das Speicherinterface und schließlich RAM und ROM.

Aufgabe 1.2: Mikroprogramm für Befehl Holen Der erste Schritt im Befehlszyklus des von-Neumann Rechners ist die *Befehl holen* Phase, in der ein Befehl aus dem Speicher (ROM oder RAM) in das Befehlsregister `IR` übertragen wird. Die Adresse kommt dabei aus dem Programmzähler `PC`. Von der gesamten Hardware werden für diese Schritte also nur das mikroprogrammierte Steuerwerk, der Speicher mit seiner Ansteuerung, die beiden Register `PC` und `IR` und einige der Tristate-Treiber benötigt. Dies ist in Abbildung 2 illustriert.

Öffnen Sie den Editor für den Mikroprogrammspeicher. Im Mikroprogramm befinden sich links die Steuersignale für das Steuerwerk selbst (`nextA`, `nextB`, `µMUX`), dann folgen die Steuersignale für die ALU und den Datenpfad, den Programmzähler und ganz rechts liegen die Steuersignale für die Speicheransteuerung (`MAR`, `MDR`, `MRR`, `nOE`, `nWE`).

Die Aufgabe ist jetzt die Erstellung eines Mikroprogramms für die *Befehl Holen*-Phase des von-Neumann Rechners, das den Befehl dessen Adesse im `PC` steht aus dem Speicher liest und in das Befehlsregister `IR` überträgt, `IR := MEM[PC]`. Damit diese Operation nach einem Reset als erste ausgeführt wird, sollte das Mikroprogramm an der Adresse `0` im `µROM` beginnen.

Wie schon in Abbildung 2 angedeutet ist, sind folgende Schritte nötig:

a) Der Wert des PCs muss in das MAR gebracht werden.
b) Dem Speicher muss mitgeteilt werden, dass man lesend auf ihn zugreifen möchte. Dabei ist zu beachten, dass der Speicher (mindestens) drei Waitstates braucht.
c) Der Wert, der nach den Waitstates aus dem Speicher kommt, muss in das MRR geschrieben werdsen.
d) Der Wert muss aus dem MRR in das IR gebracht werden.

Überlegen Sie sich für jeden dieser Schritt, welche Steuersignale aktiviert werden müssen und tragen Sie sie in Ihren Mikroprogrammspeicher ein.

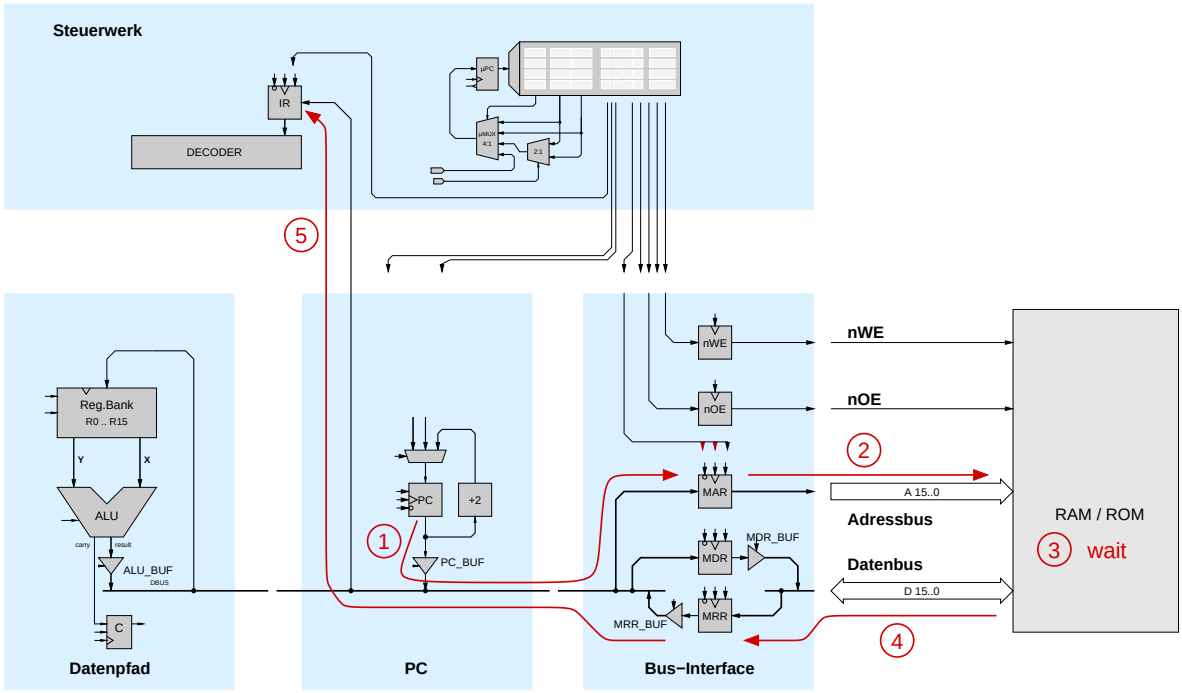


Abbildung 2: Speicherinterface und Komponenten für die Befehl-Holen Phase

.

Speichern Sie das Mikroprogramm, z.B. als Datei `fetch.rom`. Hinweis: Eventuell müssen Sie wieder die laufende Simulation anhalten (⏮-Button) und neu starten (⏭-Button), damit der Simulator ein geändertes Mikroprogramm korrekt übernimmt. Tragen Sie Ihren Mikrocode für die Befehl-Holen Phase 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	name
00	00	00	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	reset
01	00	00	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	fetch_1
02	00	00	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	fetch_2
03	00	00	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	fetch_3
04	00	00	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	fetch_4
05	00	00	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	fetch_5
06	00	00	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	fetch_6
07	00	00	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	fetch_7
08	00	00	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	
09	00	00	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	
0a	00	00	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	
0b	00	00	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	
0c	00	00	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	
0d	00	00	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	decode
0e	00	00	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	
0f	00	00	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	

2 Befehlsdekodierung

Am Ende der Befehl-Holen Phase steht der auszuführende Befehl im Befehlsregister `IR`. Abhängig vom jeweiligen Befehl (z.B. einer Addition, einem Speicherzugriff, oder einem unbedingten Sprung) muss der Prozessor aber völlig unterschiedliche Aktionen durchführen. Dazu muss im Mikroprogramm für jeden einzelnen Befehl die zugehörige Folge von Mikroinstruktionen kodiert sein. Für einige Befehle, zum Beispiel eine einfache Addition, kann dabei ein einzelner Mikroprogrammschritt ausreichen, andere Befehle wie Multiplikation oder Speicherzugriff können durchaus auch Dutzende oder Hunderte von Mikoprogrammschritten erfordern.

Wichtigste Aufgabe der *Decode*-Phase ist es, den Opcode im Befehlswort in `IR` zu analysieren und im Mikroprogramm an die richtige Stelle zu springen — die erste Mikroinstruktion des zugehörigen Mikroprogramms, dsas dann den Befehl wirklich ausführt. Die Zerlegung des Befehls in die einzelnen Teile wie Opcode, ALU-Opcode, die Register-Adressen, oder Offsets für die Sprungbefehle erfolgt in der als Decoder bezeichneten Hardware-Komponente, die Sie sich auch einzeln als Design `decoder.hds` anschauen können.

Ein Sprung im Mikroprogramm wird einfach dadurch realisiert, dass der Mikroprogrammzähler `µPC` auf den entsprechenden Wert gesetzt wird. Im Steuerwerk aus Abbildung 8 aus Bogen 1 dient dazu der externe Eingang `XA`, über den ein externer Wert direkt in den `µPC` geladen werden kann. In der Decode-Phase muss also der Multiplexer `uMUX` so angesteuert werden, dass der externe Eingang `XA` in den `µPC` geladen wird. Das Steuerwerk des D·CORE ist genau so aufgebaut: die obersten vier Opcode-Bits des Befehlsregisters `IR` werden mit vier Nullen erweitert (`XA = IR.<15:12>|0000`) an den Eingang `XA` angeschlossen. Daher beginnt zum Beispiel das Mikroprogramm für den `JMP`-Befehl mit Opcode `1100 **** **** xxxx` ab Mikroprogrammadresse `1100 0000`, das Mikroprogramm für den `HALT`-Befehl mit Opcode `1111 **** **** ****` ab Adresse `1111 0000`, usw.

Die hier gewählte Lösung ist nicht optimal, da viel Platz im Mikroprogrammspeicher ungenutzt bleibt.

Im allgemeinen Fall wird man versuchen, den Mikroprogrammspeicher besser auszunutzen.

Häufig wird die Decode-Phase zusätzlich dazu benutzt, den Programmzähler zu inkrementieren. Einerseits wird der Wert des PC für den aktuellen Befehl nicht mehr benötigt, andererseits wird der Datenpfad in der Decode-Phase nicht für Datenoperationen benutzt und ist daher frei für weitere Operationen. Da die Speicheradressen in Bytes angegeben werden, ein D·CORE Befehl aber 16 Bit breit ist, muss der PC um 2 inkrementiert werden, um auf das nächste Speicherwort zu zeigen.

Aufgabe 2.1: Decode Erweitern Sie Ihren Microcode aus Aufgabe 1 um die Befehlsdekodierung (μPC := XA) und das Inkrementieren des PC um 2. Beide Operationen können parallel in einem einzigen Mikroprogrammschritt realisiert werden. Tragen Sie jetzt einige Opcodes in das ROM ein, und testen Sie, ob die Fetch- und Decode-Phasen korrekt ausgeführt werden. Ergänzen Sie dann die Tabelle auf Seite 3 und speichern Sie das Mikroprogramm (z.B. als `fetch-decode.rom`).

3 Befehlsausführung

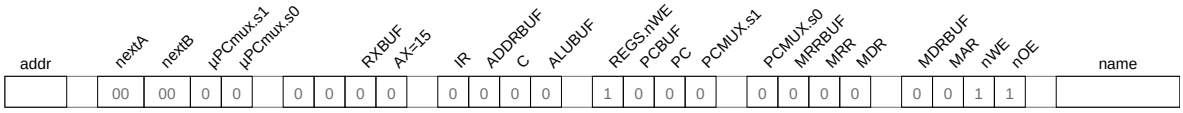
Nach Abschluss der Decode-Phase folgt die Befehlsausführung oder *Execute*-Phase, in der die eigentlichen Datenoperationen des Prozessors vorgenommen werden. Für jeden einzelnen Maschinenbefehl ist dabei eine Folge von Mikroprogrammschritten notwendig, die den Datenpfad ansteuert. Am Ende dieser einzelnen Mikroprogramme erfolgt der Rücksprung zur Mikroprogrammadresse 0, um den nächsten Befehl zu holen.

In den folgenden Aufgaben werden Sie sukzessive die einzelnen Befehle implementieren und mit diesen die ersten Programme für den D·CORE schreiben. Die Hardware des Prozessors ist dabei fest vorgegeben, es fehlen nur noch die Mikrorprogramme zur Ansteuerung der verschiedenen Enable-Leitungen für die Register und Tri-State-Treiber.

Aufgabe 3.2: HALT-Befehl Der einfachste Befehl ist der HALT Befehl, der auf den ersten Blick unsinnig erscheint. Tatsächlich verfügen aber viele moderne Prozessoren über einen solchen Befehl, um Teile des Prozessors abschalten zu können und damit Strom zu sparen. Im D·CORE dient der HALT-Befehl aber zunächst nur dazu, ein Programm gezielt beenden zu können, ohne dass der Prozessor durch den gesamten Speicher „Amok läuft“. Realisieren Sie den Befehl zum Beispiel durch eine Endlosschleife im Mikroprogramm: μPC.nextA = μPC. Tragen Sie den entsprechenden Mikroprogrammschritt hier ein:

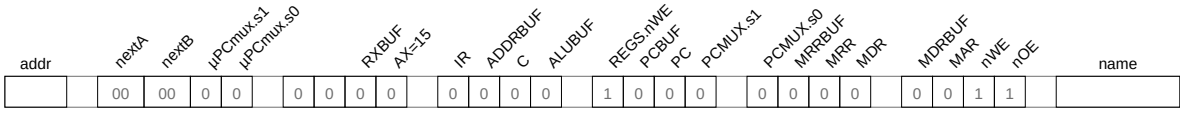
addr	nextA	nextB	µPCmux.s1	µPCmux.s0	RXBUF	AX=15	IR	ADDRBUF	C	ALUBUF	REGS.nWE	PCBUF	PC	PCMUX.s1	PCMUX.s0	MRRBUF	MRR	MDR	MDRBUF	MAR	nWE	nOE	name
	00	00	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	

des Carry-Registers, und für den Tristate-Puffer am Ausgang der ALU erzeugen. Tragen Sie den notwendigen Mikroprogrammschritt hier ein:



Die ALU ist so entworfen, dass der Carry-Ausgang nur für die Befehle gesetzt wird, die diesen Wert verändern. Für alle anderen Befehle reicht die ALU den Eingangswert von CarryIn bis zum Carry-Ausgang durch. Ihr Mikroprogramm für die ALU-Befehle sollte daher das C-Register aktivieren.

Aufgabe 3.4: Immediate- und Vergleichsbefehle Erweitern Sie das Mikroprogramm um die Vergleichsbefehle `CMPE` (*compare equal*), `CMPNE` (*compare not equal*), `CMPGT` (*compare greater*), und `CMPLT` (*compare less than*) und die ALU-Befehle mit Immediate-Operanden. Wie bei den ALU-Rechenbefehlen reicht wiederum ein einziger Mikroprogrammschritt aus, da die Auswahl der eigentlichen ALU-Funktion intern in der ALU erfolgt:



Speichern Sie Ihr Mikroprogram. Mit nur drei Mikroprogrammschritten stehen jetzt alle ALU-Befehle sowie der `HALT`-Befehl zur Verfügung. Damit können bereits die ersten vollständigen Programme geschrieben werden.

Aufgabe 3.5: Register-Initialisierung Verwenden Sie ihre Notizen aus Bogen 1, Aufgabe 3.1, um ein Programm zu schreiben, das die ersten fünf Register initialisiert, danach ein 1900er Datum in die Register `R11` bis `R13` ablegt und schließlich mit `HALT` stoppt. Ihr erstes vollständiges Programm für den D·CORE! Bitte dokumentieren Sie die einzelnen Maschinenbefehle Ihres Programms hier:

Label	Adresse	Befehlscode	Mnemonic	Kommentar
reginit:	0000	0x3400	movi R0, 0	R[0]=0
	0002			
	0004			
	0006			
	0008			
	000a			
	000c			
	000e			
	0010			
	0012			
	0014			
	0016			
	0018			
	001a			
	001c			
	001e			

Aufgabe 3.6: Vergleichsoperationen Schreiben Sie ein kleines Programm, um alle vier Vergleichsoperationen zu demonstrieren. Setzen Sie etwa `R[0] = 0, R[1] = 1, R[2] = -1` und vergleichen Sie diese Werte so miteinander, dass das C-Register abwechselnd gesetzt und rückgesetzt wird.

Label	Adresse	Befehlscode	Mnemonic	Kommentar
testcompare:	0000	0x3401	movi R1, 0	R[1]=0
	0002			
	0004			
	0006			
	0008			
	000a			
	000c			
	000e			
	0010			
	0012			
	0014			
	0016			
	0018			
	001a			
	001c			
	001e			

Aufgabe 3.7: Pseudoinstruktionen Vielleicht vermissen Sie im Befehlssatz zusätzliche Befehle, um das Carry-Register zur Initialisierung direkt setzen und zurücksetzen zu können. Warum müssen diese Befehle nicht separat mit zusätzlichen Rechenwerken und Mikroprogrammen realisiert werden?

Manchmal werden solche nützlichen Befehle als sogennante *Pseudoinstruktionen* im Assembler für einen Rechner zusätzlich zur Verfügung gestellt, obwohl sie bereits vom Befehlssatz abgedeckt werden. Der Programmierer kann dann einfacher zu merkende Befehle wie *set carry* und *clear carry* benutzen, und der Assembler setzt diese in die entsprechenden Maschinenbefehle um. Geben Sie die äquivalenten D·CORE-Befehle an:

	Befehlscode	Mnemonic	Kommentar
set carry			
clear carry			

4 Lade- und Speicherbefehle

Aufgabe 4.8: Load-Befehl Der Befehl `LDW` (*load word*) dient dazu, Datenwerte aus dem Speicher in ein Register zu übertragen. Als Pseudocode formuliert lautet der Ladebefehl im D·CORE `R[x] = MEM(R[y] + cccc<<1)` mit einer 4-bit Konstante `cccc`. Über das Feld `xxxx` im Befehlswort wird das Zielregister `RX` der `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 3 für eine einfache `struct Point3D` mit drei Elementen `x`, `y`, `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`.

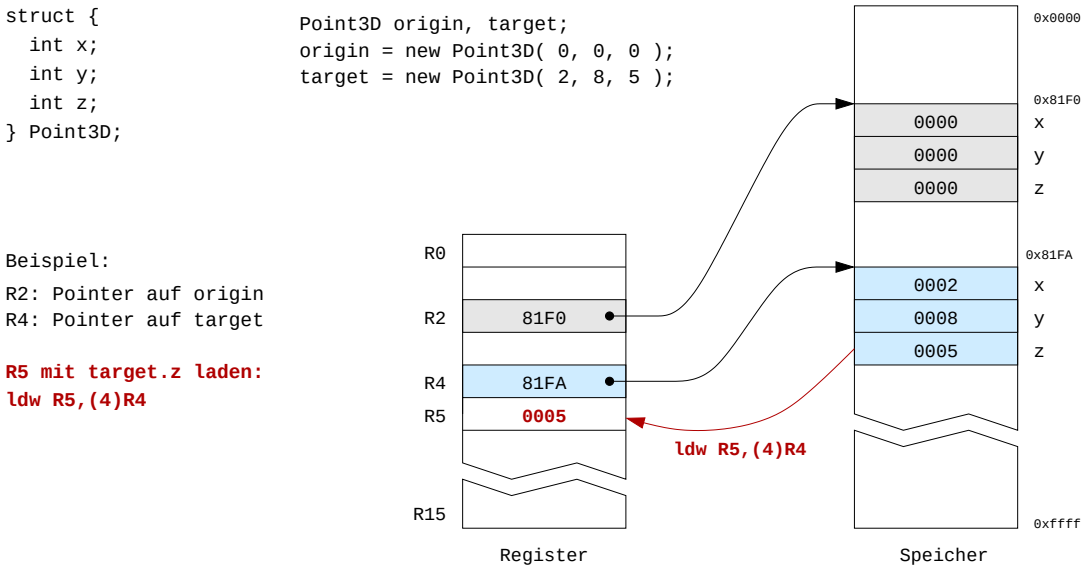


Abbildung 3: Adressierung mit Basisadresse und Offset zum direkten Zugriff auf Elemente zusammengesetzter Datentypen

Für den eigentlichen Speicherzugriff ist das gleiche komplizierte Timing erforderlich wie in der Befehl-Holen Phase (siehe Abbildung 6 in Aufgabenblatt 1). Implementieren Sie den `LDW`-Befehl und schreiben Sie ein kleines Testprogramm, um die Funktion zu demonstrieren. Tragen Sie den Microcode 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		name
		00	00	0	0		0	0	0	0		0	0	0	0		1	0	0	0		0	0	0	0		0	0	1	1		
		00	00	0	0		0	0	0	0		0	0	0	0		1	0	0	0		0	0	0	0		0	0	1	1		
		00	00	0	0		0	0	0	0		0	0	0	0		1	0	0	0		0	0	0	0		0	0	1	1		
		00	00	0	0		0	0	0	0		0	0	0	0		1	0	0	0		0	0	0	0		0	0	1	1		
		00	00	0	0		0	0	0	0		0	0	0	0		1	0	0	0		0	0	0	0		0	0	1	1		
		00	00	0	0		0	0	0	0		0	0	0	0		1	0	0	0		0	0	0	0		0	0	1	1		
		00	00	0	0		0	0	0	0		0	0	0	0		1	0	0	0		0	0	0	0		0	0	1	1		
		00	00	0	0		0	0	0	0		0	0	0	0		1	0	0	0		0	0	0	0		0	0	1	1		

Aufgabe 4.9: 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] + cccc<<1) = R[x]` mit einem Basisregister `RY` und einem positiven Offset `cccc`.

Die notwendige Ansteuerung des Speicherinterface ist in Abbildung 6b von Aufgabenblatt 1 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 `RX_BUF` 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:

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		name
		00	00	0	0		0	0	0	0		0	0	0	0		1	0	0	0		0	0	0	0		0	0	1	1		
		00	00	0	0		0	0	0	0		0	0	0	0		1	0	0	0		0	0	0	0		0	0	1	1		
		00	00	0	0		0	0	0	0		0	0	0	0		1	0	0	0		0	0	0	0		0	0	1	1		
		00	00	0	0		0	0	0	0		0	0	0	0		1	0	0	0		0	0	0	0		0	0	1	1		
		00	00	0	0		0	0	0	0		0	0	0	0		1	0	0	0		0	0	0	0		0	0	1	1		
		00	00	0	0		0	0	0	0		0	0	0	0		1	0	0	0		0	0	0	0		0	0	1	1		
		00	00	0	0		0	0	0	0		0	0	0	0		1	0	0	0		0	0	0	0		0	0	1	1		
		00	00	0	0		0	0	0	0		0	0	0	0		1	0	0	0		0	0	0	0		0	0	1	1		

Schreiben Sie jetzt ein Testprogramm, das mit möglichst wenig Anweisungen die ersten vier Befehle ihres Programms aus dem ROM in das RAM kopiert.

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			

5 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	Kodierung	Bedeutung
		Kontrollfluss
br	1000 iiii iiii iiii	PC = PC+2+imm12
jsr	1001 iiii iiii iiii	R[15] = PC+2; PC = PC+2+imm12 (call)
bt	1010 iiii iiii iiii	if (C=1) then PC = PC+2+imm12 else PC=PC+2
bf	1011 iiii iiii iiii	if (C=0) then PC = PC+2+imm12 else PC=PC+2
jmp	1100 **** **** xxxx	PC = R[x]

Auf den ersten Blick mag die Definition dieser Befehle ungewöhnlich erscheinen. Aber wie bereits in Aufgabenblatt 1 angedeutet wurde, verwenden die meisten Rechnerarchitekturen eine Byte-Adressierung des Speichers. Für den D·CORE muss daher der PC nach jedem Befehl um den Wert 2 imkrementiert 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 4 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`.

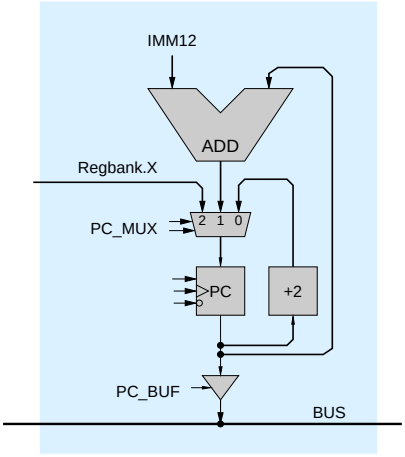


Abbildung 4: Realisierung der Sprungbefehle: Über den Multiplexer werden die Werte `PC+2`, `PC+IMM12` oder `RX` ausgewählt und in den `PC` geladen.

Aufgabe 5.10: 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		name
		00	00	0	0		0	0	0	0		0	0	0	0		1	0	0	0		0	0	0	0		0	0	1	1		

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
test_jmp:	0000	0x3403	movi R3, 0	R[3] = 0
	0002			R[2] = ????
loop:	0004			R[3] ++
	0006			jmp R[2] (goto loop)
	0008			

Aufgabe 5.11: 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 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.

Realisieren Sie 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	name
	00	00	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	

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
test_array_init:	0000	0x3480	movi R0, 8	R[0] = 8
	0002			
	0004			
	0006			
	0008			
	000A			
	000C			
	000E			
	0010			
	0012			
	0014			
	0016			
	0018			
	001A			
	001C			
	001E			