Praktikum Technische Informatik

T3-2

Maschinenprogrammierung

Fachbereich Informatik
Universität Hamburg
Vogt-Kölln-Str. 30
D22527 Hamburg

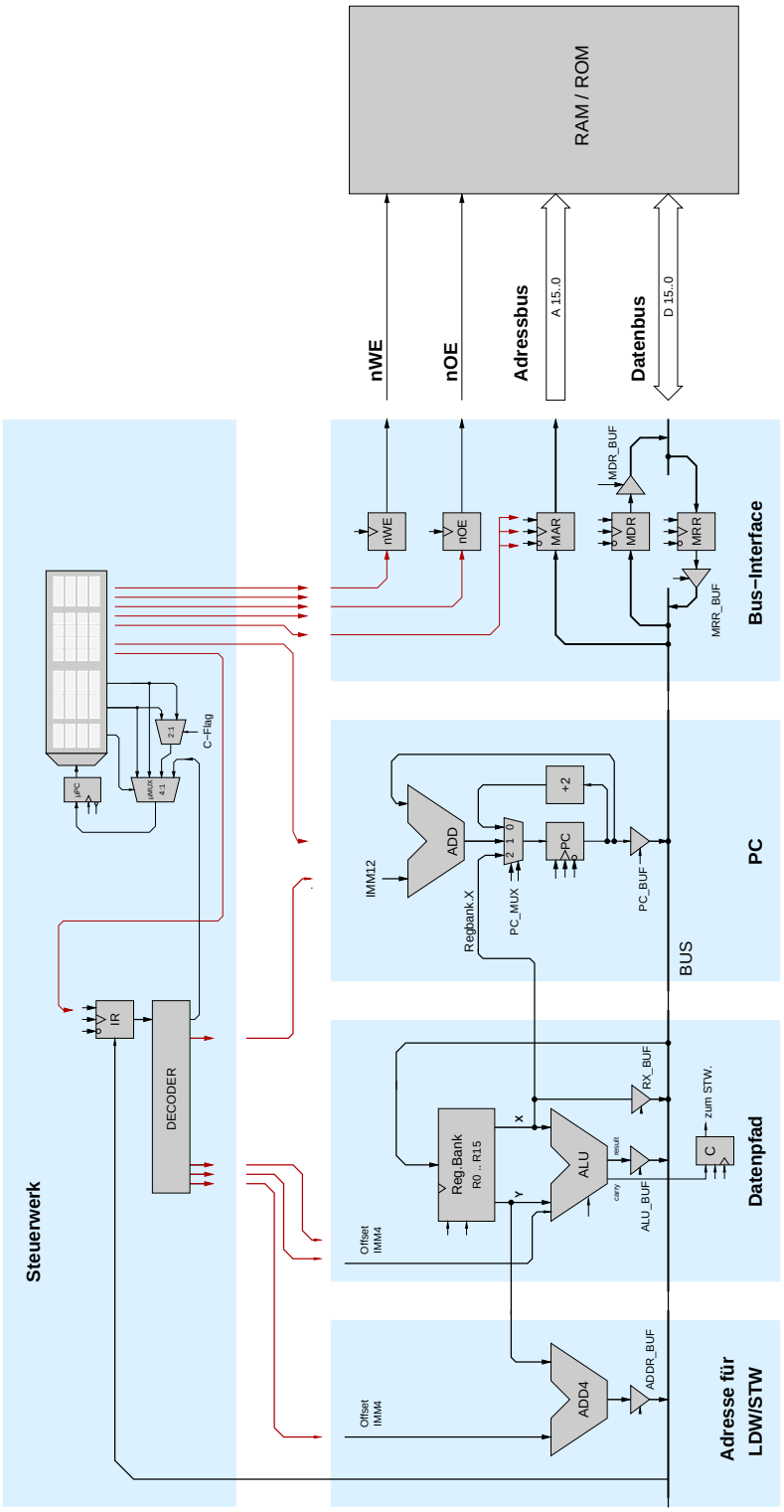

Abbildung 1: Blockschaltbild des D·CORE Prozessors

1 Lade- und Speicherbefehle

Die Aufgaben dieses Versuches schließen sich nahtlos an das erste Aufgabenblatt T3-1 an. Zunächst wird unser D·CORE-Prozessor sukzessive um die noch fehlenden Load/Store- und Sprungbefehle erweitert. Damit können dann vollwertige Programme in Maschinensprache geschrieben werden. Als grundlegende Techniken werden Schleifen sowie indirekte und indizierte Adressierung erprobt.

Für den ersten Test neuer Mikroprogrammschritte und neuer Testprogramme empfiehlt sich natürlich der Einzelschrittmodus durch direktes Anklicken des Schalters `clk` am Takteingang. Für längere Programme ist es aber bequemer, die Takte nicht einzeln durch Anklicken erzeugen zu müssen. Stellen Sie dazu die gewünschte Taktfrequenz am Taktgenerator (links oben im Design `processor.hds`) über das Edit-Menü ein, und schalten Sie dann mit dem Schalter `clk-select` auf den Taktgenerator um.

Zur Fehlersuche in Mikroprogramm und Maschinenprogramm ist zunächst der Einzelschrittmodus beim aktivem *glow mode* empfehlenswert. Verwenden Sie die „Disassemble“-Darstellung der Speicher, um den Ablauf des Programms zu verfolgen. Zur Überprüfung der Zeitabhängigkeiten — oder falls ein Fehler erst nach vielen Takten auftritt — sind auch Impulsdiagramme eine wichtige Hilfe. Selektieren Sie dazu im Editor-Menü die Funktion `Signals->Add probes->toplevel signals`, oder wählen Sie die zu beobachtenden Signale einzeln über das Kontextmenü mit `Popup->wire->add probe` aus. Sobald die Impulsdiagramme aktiv sind, werden alle zukünftigen Wertewechsel auf den Signalen protokolliert und gespeichert.

Achtung: Die Simulation der gesamten Hardwarekomponenten des Prozessors inklusive aller Zeitbedingungen ist recht aufwendig. Dies gilt erst recht für die Animation und das Neuzeichnen der Speicherinhalte in den Editor-Fenstern für den Mikroprogramm, die Registerbank und die Speicher. Mit den folgenden Einstellungen können Sie bei Bedarf die Simulation beschleunigen:

- Deaktivieren Sie im Menü `Display->glow mode`.
- Deaktivieren Sie im Menü `Display->rtlib animation`.
- Schließen Sie die Editor-Fenster der Speicher, um das ständige Neuzeichnen der Speicherinhalte zu vermeiden. Die Funktion der Speicher wird dadurch natürlich nicht beeinflusst.
- Wählen Sie unter `Window->repaint frequency` einen niedrigen Wert.

Aufgabe 1.1: Load-Befehl Der Befehl `LDW` (*load word*) dient dazu, Datenwerte aus dem Speicher in ein Register zu übertragen. Als Pseudocode formuliert lautet der Ladebefehl 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 2 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

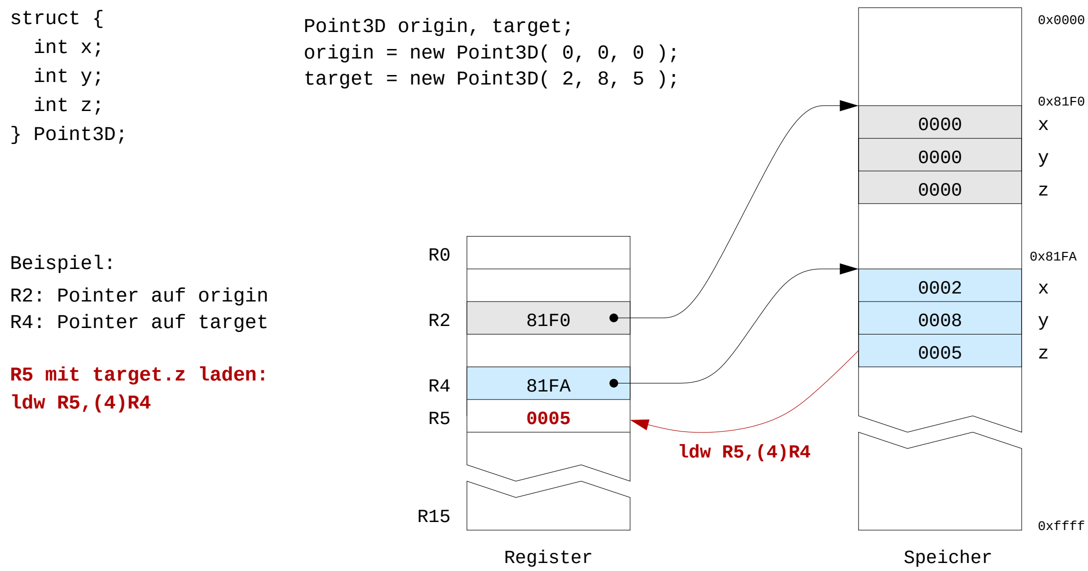

Abbildung 2: Adressierung mit Basisadresse und Offset zum direkten Zugriff auf Elemente zusammengesetzter Datentypen

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`.

Für den eigentlichen Speicherzugriff ist das gleiche komplizierte Timing erforderlich wie in der Befehl-Holen Phase (siehe Abbildung 6 in Aufgabenblatt T3-1). 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 1.2: 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] + 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		

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 T3-1):

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 T3-1 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.

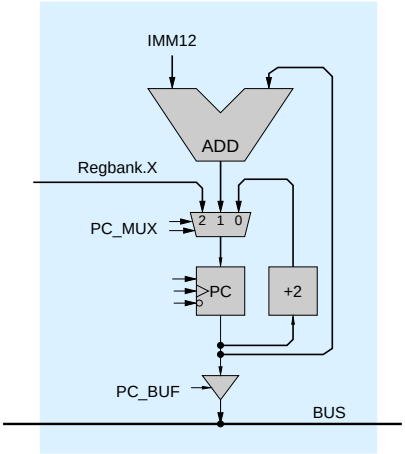

Abbildung 3: Realisierung der Sprungbefehle: Über den Multiplexer werden die Werte `PC+2`, `PC+IMM12` oder `RX` ausgewählt und in den `PC` geladen.

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`.

Aufgabe 2.3: 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	name
	00	00	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 2.4: 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 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			

Aufgabe 2.5: Bedingte Sprünge Programmverzweigungen erfordern bedingte Sprungbefehle. Im D·CORE dienen dazu 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 einfach an den Select-Eingang des 2:1-Multiplexers im Steuerwerk geschaltet, was im Mikroprogramm die Auswahl von `uROM.nextA` oder `uROM.nextB` als Folgeadresse erlaubt. Dies ist auch in Abbildung 1 skizziert, und Sie haben diese Umschaltung bereits in Aufgabe 5.1 vom Aufgabenblatt T3-1 benutzt.

Vor einem bedingten Sprung muss daher das C-Register 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 die beiden BT- und BF-Befehle im Mikroprogramm:

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	
	00	00	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	

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		

Mit den eben implementierten `BT`- bzw. `BF`-Befehlen lassen sich jetzt auf dem D·CORE auch Schleifen mit einer Abbruchbedingung ausführen. Das folgende Programm demonstriert neben der Umsetzung einer `while`-Schleife auch noch die indizierte Adressierung für den Zugriff auf Arrays.

Aufgabe 2.6: 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 = 10;
int base[] = 0x8010;  // Startadresse

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

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			

Aufgabe 2.7: While-Schleife Erweitern Sie das Programm so, dass es in einer zweiten Schleife die Summe aller Elemente im Array berechnet:

Label	Adresse	Befehlscode	Mnemonic	Kommentar
	0000			
	0002			
	0004			
	0006			
	0008			
	000A			
	000C			
	000E			
	0010			
	0012			
	0014			
	0016			
	0018			
	001A			
	001C			
	001E			

Aufgabe 2.8: JSR-Befehl 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 ein wenig 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 Befehlsdekoder 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.

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

Aufgabe 2.9: Return Begründen Sie, warum die D·CORE-Architektur keinen expliziten Return-Befehl bereitstellt:

Laden Sie jetzt die vorgegebene Datei `bigtest.rom` in das ROM und starten Sie den Prozessor. Das Testprogramm überprüft 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 2.10: Hex-To-ASCII Schreiben Sie eine Funkion, die einen Hex-Zahlenwert im Bereich 0..15 in den zugehörigen ASCII-Wert ’0’..’9’ (=0x30..0x39) bzw. ’A’..’F’ (=0x41..0x46) umsetzt. Verwenden Sie Register `R10` für das Argument und `R11` für das Resultat. Für Eingabewerte größer als 15

soll die Funktion das Zeichen '*' (=0x2A) zurückliefern:

Label	Adresse	Befehlscode	Mnemonic	Kommentar
	0000			
	0002			
	0004			
	0006			
	0008			
	000A			
	000C			
	000E			
	0010			
	0012			
	0014			
	0016			
	0018			
	001A			
	001C			
	001E			

Aufgabe 2.11: Test von JSR und Return Schreiben Sie ein Testprogramm, um obige Funktion nacheinander mit den Eingabewerten 0..16 aufzurufen und so die Befehle JSR und Return zu testen:

Label	Adresse	Befehlscode	Mnemonic	Kommentar
	0000			
	0002			
	0004			
	0006			
	0008			
	000A			
	000C			
	000E			
	0010			
	0012			
	0014			
	0016			
	0018			
	001A			
	001C			
	001E			

3 Leistungsbewertung

Die in den bisherigen Aufgaben vorgestellte Realisierung des D·CORE-Befehlssatzes mit Mikroprogramm, konservativem Speichertiming und ohne Befehlspipeline ist typisch für (ältere) Prozessoren der 10 MHz-Klasse, wie dem Motorola 68000 oder Intel 80286. Es zeigt sich, dass insbesondere der Speicherzugriff die Leistung des Prozessors begrenzt — die Mikroprogramme für Befehl-Holen und die `LDW` und `STW`-Befehle erfordern jeweils fast 10 Schritte, während die Decode-Phase und die meisten

Befehle mit nur 1 bis 2 Schritten realisiert werden können.

Aufgabe 3.1: Massnahmen zur Leistungssteigerung Aus der Vorlesung kennen Sie eine Vielzahl von Massnahmen zur Leistungsverbesserung von Rechensystemen: I-Cache, D-Cache, mehrstufige Caches, Befehlspipeline, zusätzliche Rechenwerke (z.B. Hardware-Multiplizierer), mehrfache parallele Rechenwerke (superskalar), usw.

Berechnen Sie, wieviele Takte der D·CORE mit Ihrem Mikroprogramm im Mittel für eine Instruktion benötigt (clocks per instruction, CPI). Nehmen Sie dazu die folgenden Befehlshäufigkeiten an, die in Hennessy & Patterson für einen 32-bit Prozessor (DLX) gemessen wurden:

Befehl	Mnemonic	Häufigkeit (% Prozent aller Befehle)
Register-Laden	`LDW`	28
Register-Speichern	`STW`	10
Rechnen	`ADD`, etc.	30
Vergleiche	`CMPE`, etc.	13
Sprünge	`BT`, `BF`	17
Call, Return	`JSR`	2

Zum Beispiel gilt bei (fetch: 8 Takte), (decode: 1 Takt), (ldw: 8 Takte), (stw: 8 Takte):

$$CPI = 8\ \text{Takte}(\text{fetch}) + 1\ \text{Takt}(\text{decode}) + 0.28 \cdot 8\ \text{Takte}(\text{load}) + ... + 0.02 \cdot 2\ \text{Takte}(\text{jsr}) = 12.68$$

Berechnen Sie unter der Annahme, dass die Caches ideal sind (keine Misses), welcher CPI-Wert sich bei den folgenden Varianten ergibt:

I-Cache mit 1-Takt Zugriffszeit, CPI =
D-Cache mit 1-Takt Zugriffszeit, CPI =
separate I/D-Caches, je 1 Takt Zugriffszeit, CPI=

Maximale Taktfrequenz und Übertakten

Für viele Anwendungen wird eine optimale Performance des Gesamtsystems gefordert. Die Taktfrequenz des Prozessors wird daher so eingestellt, dass die Rechenergebnisse für die langsamste Operation (den *kritischen Pfad*) gerade noch rechtzeitig vor Ende der Taktperiode fertig vorliegen.

Damit das funktioniert, muss das Zeitverhalten aller einzelnen Operationen mit allen möglichen Eingabedaten bekannt sein oder abgeschätzt werden, denn im allgemeinen sind die Verzögerungen datenabhängig. Zum Beispiel dauert die Addition von `-4 + 4` wegen des Übertrags durch alle Stellen wesentlich länger als etwa die Addition von `1 + 4`. Wegen der grossen kommerziellen Bedeutung gibt es ausgefeilte Analyseprogramme, die für einen gegebene Hardwarestruktur automatisch die kritischen Pfade ermitteln und auswerten.

Übertakten funktioniert daher überhaupt nur, weil die Hersteller meistens noch einen Sicherheitsfaktor für Alterung und besonders heiße Umgebung einrechnen. Das Risiko dabei ist natürlich, dass die meisten Operationen scheinbar noch funktionieren, für bestimmte Daten aber bereits falsche Ergebnisse berechnet werden, weil die ALU noch nicht fertig war. Heruntertakten ist dagegen harmlos.

Komponente	Verzögerung [ns]
ALU	100
Multiplexer	20
Register	10
μROM	50

Tabelle 1: Ausführungszeiten verschiedener Komponenten

Aufgabe 3.2: Übertakten Überlegen Sie sich die Ausführungszeiten verschiedener Mikroprogrammschritte. Legen Sie dazu die in Tabelle 1 genannten Verzögerungszeiten der Komponenten zugrunde. Für welche Operation ergibt sich die längste Ausführungszeit? Welche maximale Taktfrequenz ergibt sich daraus für einen sicheren Betrieb des Prozessors?

Maximale Verzögerungszeit:

Versuchen Sie jetzt, Ihren Prozessor zu übertakten. Öffnen Sie dazu das Kontextmenü des Taktgenerators und tragen Sie dort unter `period` sukzessive immer kleinere Werte ein. Starten Sie dann die Simulation mit dem Programm aus Aufgabe 3.9 (Speicher- und Registerinitialisierung). Bis zu welcher Taktfrequenz funktioniert der Rechner einwandfrei?

4 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 hinunter zur logischen Ebene und physikalischen Ebene.
- Den grundlegenden Aufbau eines von-Neumann-Rechners mit Steuerwerk, Operationswerk mit Registern und ALU, dem Speicher und 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. Pentium-II 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.