Praktikum Technische Informatik

T3-1

Mikroprozessorsysteme

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

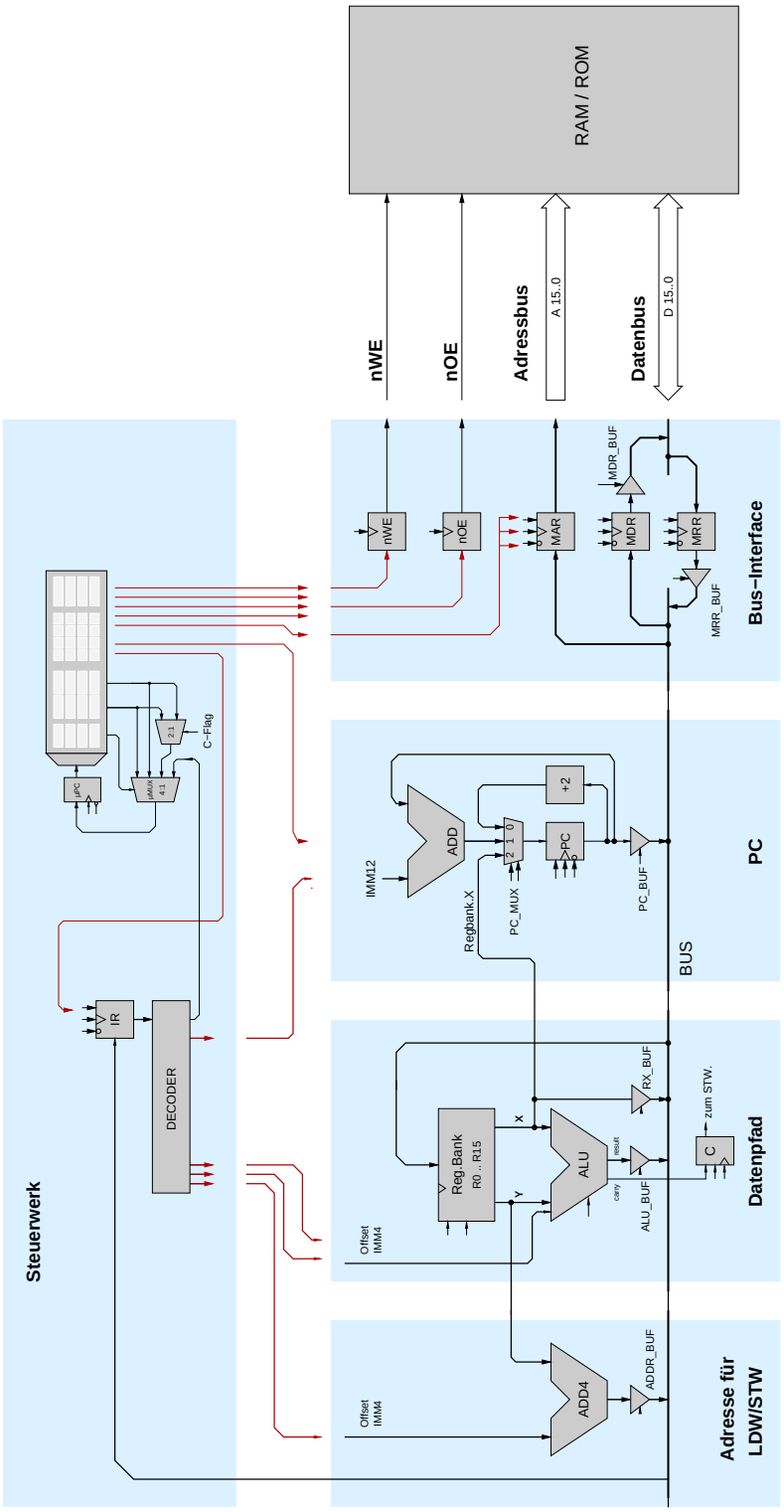


Abbildung 1: Blockschaltbild des D·CORE Prozessors

1 Einführung und Motivation

Ziel des Praktikums *technische Informatik T3* ist das Verständnis von Funktion und Struktur eines modernen Mikroprozessorsystems. Um überhaupt mit der Komplexität eines Computers umgehen zu können, hat sich die Einteilung in aufeinander aufbauende Abstraktionsebenen bewährt. In diesem Praktikum werden folgenden Ebenen des Gesamtsystems an praktischen Beispielen behandelt: *Betriebssystemebene*, *Assemblerebene*, *Ebene der Maschinensprache*, *Register-Transfer-Ebene*. Die Ebenen oberhalb der Betriebssystemebene sind dem P-Zyklus vorbehalten, die unteren Ebenen von Schaltwerkebene bis hinunter zur Physik kennen Sie bereits aus T1 und T2.

Um die Hierarchie der einzelnen Abstraktionsebenen deutlich zu machen, werden alle Aufgaben in einem *bottom-up* Vorgehen aufeinander aufbauen. Am ersten Praktikumstag wird dazu ein einfacher Mikroprozessor aufgebaut (Register-Transfer-Ebene) und schrittweise um neue Befehle erweitert (Befehlsarchitektur). Damit können dann erste Assemblerprogramme geschrieben werden, deren Komplexität sich langsam erhöht. Schließlich werden wir grundlegende Konzepte von Betriebssystemen, insbesondere die Speicherorganisation, untersuchen.

Wegen der gegenüber einem Hardwareaufbau besseren Debug-Möglichkeiten werden die folgenden Versuche zunächst mit dem HADES-Simulator durchgeführt, den Sie bereits aus dem Praktikum T1 kennen. Diesmal sind jedoch die Teilschaltungen zum größten Teil bereits fertig aufgebaut und müssen lediglich vervollständigt werden. Die späteren Aufgaben zur Assemblerprogrammierung werden dann mit einem üblichen Compiler/Debugger durchgeführt.

Aufgabe 1.1: Vorbereitung Ohne ein solides Grundwissen über digitale Logik und Rechnerorganisation sind die folgenden Versuche nicht durchführbar. Machen Sie sich daher anhand Ihrer Vorlesungs- und Praktikumsunterlagen noch einmal mit dem Thema vertraut. Versuchen Sie zum Beispiel, die folgenden Begriffe spontan zu erklären: Moore-Automat, flankengesteuertes Flipflop, ALU, Zweierkomplement-Zahldarstellung, Random-Access Memory, Programmzähler, indizierte Adressierung, Interrupt, Virtueller Speicher, LRU-Seitenersetzung, Round-Robin Scheduling.

Aufgabe 1.2: Installation Erstellen Sie zunächst ein Unterverzeichnis für Ihre HADES-Dateien unterhalb von `C:\java\hades\users` auf dem Praktikumrechner, sofern dieses nicht noch aus vorigen Semestern vorhanden ist. Verwenden Sie zum Beispiel die Namen Ihrer Unix-Accounts am Rechenzentrum, etwa `C:\java\hades\users\0gates-0mcnealy\t3`. Vermerken Sie bitte hier das von Ihnen gewählte Verzeichnis:

`C:\` _

Um die Zeit der Schaltplaneingabe zu sparen, finden Sie auf unserem Webserver unter `tams-www.informatik.uni-hamburg.de/lehre/ws2003/praktika/t3Prak/t3-hades.zip` ein Archiv mit vorbereiteten Musterdateien für alle nachfolgenden Versuche. Laden Sie das Archiv und entpacken Sie die einzelnen Dateien mit WinZip in Ihr oben erstelltes Benutzerverzeichnis.

Machen Sie sich aus den Praktikumsunterlagen noch einmal mit der Bedienung des HADES-Simulators vertraut, vor allem dem Popup-Menü und der Farbkodierung der Logikpegel (etwa cyan für nicht initialisierte Signale, grau für `0` und rot für `1`). Eine Quick-Reference Karte (hades-quick-reference.pdf) mit der Kurzbedienungsanleitung finden Sie ebenfalls auf dem Webserver.

Aufgabe 1.3: RT-Komponenten in Hades Öffnen Sie das Hades-Design `components.hds`. Es demonstriert verschiedene elementare RT-Komponenten: Schalter, Register, Register mit Enable, Tristate-Treiber, einen Bus mit drei Treibern, einen Inkrementer. Ein Register ist einfach eine „Abkürzung“ für eine Anzahl von einzelnen flankengesteuerten Flipflops mit gemeinsamer Ansteuerung aber separaten Datenleitungen. Das Register mit Enable übernimmt neue Datenwerte bei einer Taktflanke nur dann, wenn gleichzeitig der Enable-Eingang auf `1` liegt. Ein Tristate-Treiber wirkt wie ein Schalter und gibt sein Eingangssignal nur weiter, solange sein Enable-Eingang auf `1` liegt.

Die Ausgangswerte der Eingabeschalter an Bussen (`IpinVector`) lassen sich direkt durch Anklicken inkrementieren bzw. mit Shift-Klick dekrementieren. Durch Anklicken im rechten Teil ist es möglich, nacheinander die Werte `X` und `U` (undefiniert) und `Z` (hochohmig) zu erzeugen. Wenn Sie die Maus einige Zeit nicht bewegen, erscheint ein Kontext-Tooltip mit dem Wert der Komponente bzw. des Signals. Benutzen Sie das Kontextmenü, um die Edit-Dialoge der einzelnen Komponenten zu öffnen.

Versuchen Sie jetzt, durch geeignete interaktive Ansteuerung der Steuerleitungen einige Werte aus dem Eingabe-Schalter `I` in das Register `X` und umgekehrt aus `J` nach `Y` zu übertragen. Beachten Sie, dass Sie den Bus durch Ansteuerung der Tristate-Treiber auch ganz abschalten (Z) oder kurzschliessen können (X). Wählen Sie im Editor die Menü-Option `Special -> Disable Create Signals`, um bei versehentlichem Anklicken eines Anschlusses nicht gleich eine neue Leitung zu erzeugen.

2 D·CORE Prozessor

Um die Funktion eines Computersystems wirklich zu begreifen, hat sich ein *bottom-up* Vorgehen bewährt. Dazu werden Sie in den folgenden Aufgaben schrittweise erst alle Komponenten kennenlernen und dann einen vollständigen Mikroprozessor realisieren — als Simulationsmodell. Bei Interesse können Sie Ihren Mikroprozessor später aber gerne auch auf unserer FPGA-Prototypenplatine als reale Hardware austesten.

Leider sind moderne 32-bit Prozessoren einfach zu komplex, um Sie innerhalb weniger Stunden wirklich verstehen zu können. Dies gilt erst recht für die Intel x86-Architektur mit ihrem komplizierten Befehlssatz und den Spezialaufgaben der einzelnen Register. Aber auch die veralteten 8-bit Architekturen sind nicht optimal: zwar lässt sich die Hardware leicht verstehen, aber dafür wird die Programmierung sehr aufwändig.

Deshalb erscheint eine „saubere“ RISC-Architektur als guter Kompromiss. Unser D·CORE-Prozessor (*demo core*) orientiert sich dabei stark an der aktuellen M·CORE-Architektur von Motorola, die für Anwendungen in *embedded systems* mit hoher Performance bei minimalem Stromverbrauch entwickelt wurde (etwa Mobiltelefone). Diese wurde erst 1998 vorgestellt und weist gegenüber älteren Architekturen eine ganze Reihe von Vorteilen auf:

- zwei-Adress RISC-Maschine mit 16 Universalregistern
- extrem einfaches und reguläres Programmiermodell
- keine Spezialregister, keine implizit gesetzten Flags
- umfangreicher und trotzdem übersichtlicher Befehlssatz
- optimale Unterstützung von Interrupts und Exceptions

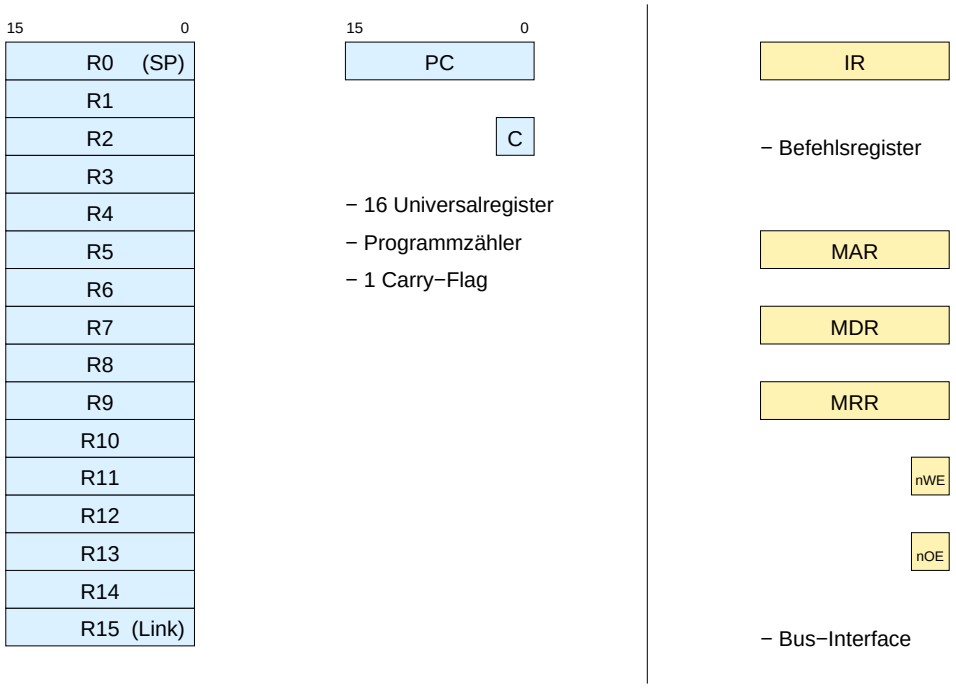


Abbildung 2: Die Architektur (Programmiermodell) des D·CORE Prozessors: Programmzähler `PC`, 16 Universalregister und 1 Carry-Flag (links). Das Befehlsregister `IR` und die zusätzlichen Register des Businterface (`MAR`, `MDR`, `MRR`, `nOE`, `nWE`) sind dagegen nicht für Programme sichtbar (rechts).

Trotz der hohen Regularität sind die M·CORE-Prozessoren immer noch viel zu komplex für ein Grundpraktikum. Deshalb verwendet der D·CORE nur 16-bit statt 32-bit Wortbreite und einen reduzierten Befehlssatz. Natürlich sind aber alle wesentlichen Befehle enthalten; und D·CORE-Programme sollten mit wenigen Änderungen auch auf dem M·CORE laufen.

2.1 Programmiermodell

Das Programmiermodell für den D·CORE ist in Abbildung 2 links dargestellt. Es besteht lediglich aus 16 Universalregistern `R0` bis `R15` mit je 16 bit Wortbreite, dem Programmzähler `PC` mit ebenfalls 16-bit, und einem einzelnen Flag-Register `C` (Carry).

Der rechte Teil der Abbildung zeigt die Register, die für die Realisierung des Prozessors zusätzlich benötigt werden, die aber für den Assemblerprogrammierer nicht direkt zugänglich sind. Es handelt sich um das Befehlsregister `IR` (instruction register), und einige Register für das Bus- und Speicherinterface des Prozessors, deren Funktion später sukzessive erläutert wird.

2.2 Befehlssatz

Die Befehls-*Architektur* eines Rechners wird durch seinen Befehlssatz definiert, der alle auf dem Rechner möglichen Operationen exakt beschreibt. Neben der eigentlichen Rechenoperation müssen dabei auch die Ziel- und Quellenoperanden, eventuelle Seiteneffekte, sowie die Befehlskodierung angegeben werden. Meistens gibt es deshalb eine kurze Tabelle zur Übersicht über alle Befehle und

Mnemonic	Kodierung	Hex	Bedeutung
			ALU-Operationen
mov	0010 0000 yyyy xxxx	20yx	R[x] = R[y]
addu	0010 0001 yyyy xxxx	21yx	R[x] = R[x] + R[y]
addc	0010 0010 yyyy xxxx	22yx	R[x] = R[x] + R[y] + C; (modifiziert C)
subu	0010 0011 yyyy xxxx	23yx	R[x] = R[x] - R[y]
and	0010 0100 yyyy xxxx	24yx	R[x] = R[x] AND R[y]
or	0010 0101 yyyy xxxx	25yx	R[x] = R[x] OR R[y]
xor	0010 0110 yyyy xxxx	26yx	R[x] = R[x] XOR R[y]
not	0010 0111 **** xxxx	27*x	R[x] = NOT R[x]
			Shift-Operationen
lsl	0010 1000 yyyy xxxx	28yx	R[x] = R[x] << R[y].<3:0>
lsr	0010 1001 yyyy xxxx	29yx	R[x] = R[x] >> R[y].<3:0>
asr	0010 1010 yyyy xxxx	2Ayx	R[x] = R[x] >>> R[y].<3:0>
lslc	0010 1100 **** xxxx	2C*x	R[x] = R[x] << 1, C=R[X].15
lsrc	0010 1101 **** xxxx	2D*x	R[x] = R[x] >> 1, C=R[X].0
asrc	0010 1110 **** xxxx	2E*x	R[x] = R[x] >>>1, C=R[X].0
			Vergleichs-Operationen
cmpe	0011 0000 yyyy xxxx	30yx	C = (R[x] == R[y]) (signed)
cmpne	0011 0001 yyyy xxxx	31yx	C = (R[x] != R[y])
cmpgt	0011 0010 yyyy xxxx	32yx	C = (R[x] > R[y])
cmplt	0011 0011 yyyy xxxx	33yx	C = (R[x] < R[y])
			Immediate-Operationen
movi	0011 0100 cccc xxxx	34cx	R[x] = cccc
addi	0011 0101 cccc xxxx	35cx	R[x] = R[x] + cccc
subi	0011 0110 cccc xxxx	36cx	R[x] = R[x] - cccc
andi	0011 0111 cccc xxxx	37cx	R[x] = R[x] AND cccc
lsli	0011 1000 cccc xxxx	38cx	R[x] = R[x] << cccc
lsri	0011 1001 cccc xxxx	39cx	R[x] = R[x] >> cccc
bseti	0011 1010 cccc xxxx	3Acx	R[x] = R[x] | (1 << cccc) (set bit)
bclri	0011 1011 cccc xxxx	3Bcx	R[x] = R[x] & !(1 << cccc) (clear bit)
			Speicher-Operationen
ldw	0100 cccc yyyy xxxx	4cyx	R[x] = MEM(R[y] + cccc<<1)
stw	0101 cccc yyyy xxxx	5cyx	MEM(R[y] + cccc<<1) = R[x]
			Kontrollfluss
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	if (C=1) then PC = PC+2+imm12 else PC=PC+2
bf	1011 iiii iiii iiii	Biii	if (C=0) then PC = PC+2+imm12 else PC=PC+2
jmp	1100 **** **** xxxx	C**x	PC = R[x]
			Interrupt (Aufgabenzettel 4)
trap	1101 **** **** iiii	D**i	PC = VT+(iiii<<1) (software interrupt)
rfi	1110 **** **** ****	E***	PC = EPC (return from interrupt)
halt	1111 **** **** ****	F***	halt andere Opcodes illegal

xxxx: 4-bit Index des Quell- und Zielregisters RX
yyyy: 4-bit Index des Quellregisters RY
cccc: 4-bit Konstante IMM4
iiii: 12-bit sign-extended Konstante IMM12
****: don't care

Tabelle 1: Befehlssatz des D·CORE

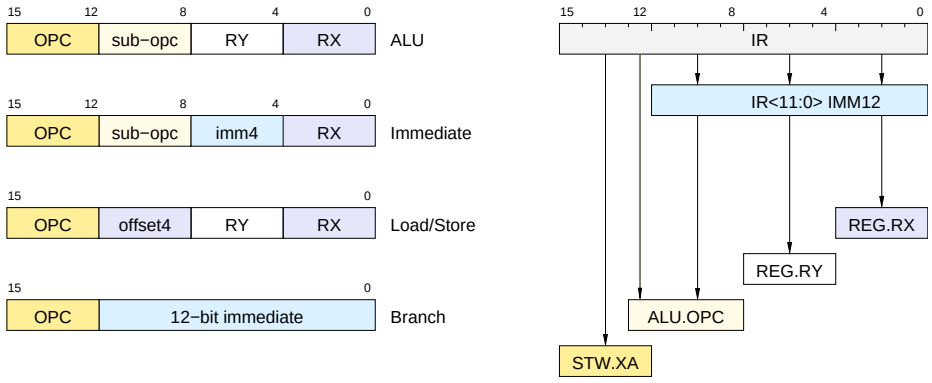


Abbildung 3: Befehlsformate (links) und Dekodierung der einzelnen Felder (rechts)

eine längere Beschreibung jedes einzelnen Befehls.

Eine möglichst reguläre Struktur des Befehlssatzes vereinfacht nicht nur die Struktur der Prozessor-Hardware, sondern erleichtert auch den Entwurf von Assembler und Compiler. Tabelle 1 enthält die Befehlsliste unseres D·CORE-Prozessors. Es zeigt sich, dass alle Befehle der obige Liste insgesamt nur vier verschiedene *Befehlsformate* verwenden, die in Abbildung 3 dargestellt sind. In allen Befehlen werden die obersten vier Bits 15..12 für den *Opcode* verwendet; das Quell- und Zielregister `RX` wird durch Bits 3..0 adressiert. Die einzelnen Felder lassen sich also trivial aus dem Befehlswort dekodieren. (Zum Vergleich: M·CORE verwendet 14 verschiedene Befehlsformate, um die 65536 möglichen Befehlsworte möglichst optimal ausnutzen zu können.)

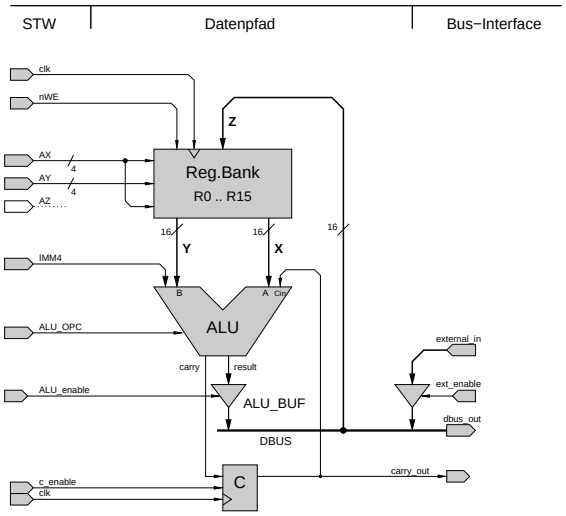
3 Datenpfad

Der in Abbildung 4 gezeigte Datenpfad unseres D·CORE-Prozessors ist für eine moderne Registermaschine typisch. Kernstück ist die Registerbank, in der die Universalregister untergebracht sind. Die Inhalte der über die *Adressports* `AX` und `AY` ausgewählten Register werden auf den *Leseports* `X` und `Y` dauernd ausgegeben. Sofern die *write-enable* Leitung aktiviert (low!) ist, wird mit der steigenden Taktflanke von `clk` der gerade am *Schreibport* `Z` anliegende Wert in das über `AZ` adressierte Register geschrieben.

Natürlich ist es wünschenswert, über möglichst viele Register zu verfügen und außerdem die Operanden und das Ziel jeder Alu-Operation unabhängig voneinander wählen zu können, zum Beispiel `R3 = R2 + R1` (eine *Drei-Adress-Maschine*). Das ist bei 32-bit Prozessoren auch kein Problem, denn die notwendigen Bits für die Registeradressen des Zielregisters und der zwei Quellregister passen problemlos in ein 32-bit Befehlswort hinein — zum Beispiel werden bei 32 Registern $3 \cdot \log_2 32 = 15$ Bits für die Adressen benötigt.

In ein 16-bit Befehlswort wie im D·CORE passt das aber nicht hinein, da nur Platz für zwei Befehle bliebe. Deshalb verfügt D·CORE nur über 16 Register und gleichzeitig wird das Zielregister immer auch als ein Quellregister verwendet; also zum Beispiel `R2 = R2 + R1`. In der Hardware sind dazu an der Registerbank einfach die Adressleitungen `AX` und `AZ` miteinander verbunden.

Die ALU kombiniert die Logik für Addition, die logischen Operationen, und einen Shifter. Sie verfügt über insgesamt 26 verschiedene Funktionen, die über den Eingang `ALU_OPC` ausgewählt werden.



ALU_OPC	Result, Carry
0 0000	B
0 0001	A + B
0 0010	A+B+Cin,update C
0 0011	A − B
0 0100	A AND B
0 0101	A OR B
0 0110	A XOR B
0 0111	NOT A
0 1000	A << B.<3:0>
0 1001	A >> B.<3:0>
0 1010	A >>> B.<3:0>
0 1011	undefined
0 1100	A << 1, C = A.15
0 1101	A >> 1, C = A.0
0 1110	A >>> 1, C = A.0
0 1111	undefined
1 0000	C = (A == B)
1 0001	C = (A != B)
1 0010	C = (A > B)
1 0011	C = (A < B)
1 0100	IMM4
1 0101	A + IMM4
1 0110	A − IMM4
1 0111	A AND IMM4
1 1000	A << IMM4
1 1001	A >> IMM4
1 1010	set bit (IMM4)
1 1011	clear bit (IMM4)
1 11**	undefined

Abbildung 4: Datenpfad des D·CORE Prozessors: Registerbank, zentrale ALU und Carry-Flag. Adressen, Alu-Opcode und Takte werden vom Steuerwerk geliefert.

Anders als in den meisten älteren Architekturen gibt es im D·CORE nur genau ein Flag-Register; und dieses wird auch nur durch bestimmte Befehle (`ADDC`, die Vergleichsbefehle, und die 1-Bit Shift-Befehle) von der ALU neu berechnet — für die übrigen Befehle reicht die ALU einfach den alten Wert von `Cin` nach `C` durch. Einmal gesetzt, wird der Wert im `C`-Register also nicht automatisch von allen folgenden Befehlen überschrieben.

Beachten Sie übrigens das gewählte Abstraktionsniveau der *Register-Transfer-Ebene* mit Komponenten wie Register, Multiplexer, ALUs, Speicher. Wichtig ist hier nur die Speicherung und der Transfer von Registerinhalten, nicht aber die eigentliche Realisierung der Komponenten aus Hunderten oder Tausenden von einzelnen Logikgattern. Den prinzipiellen Aufbau von Registern und ALU kennen Sie ja bereits aus T1.

Aufgabe 3.1: Initialisierung der Register Öffnen Sie das Hades-Design `datapath.hds`. Es enthält die Registerbank, das Carry-Register und die ALU. Ausserdem sind die Kontrollsignale der Register und der ALU direkt mit Schaltern verbunden und können interaktiv bedient werden.

Öffnen Sie über das Kontext-Menü den Editor für die Registerbank, um die Registerinhalte direkt anzuzeigen (Verkleinern Sie das Hades-Fenster, um beide Fenster nebeneinander anordnen zu können). Zu Beginn der Simulation werden diese Werte ebenso wie der Ausgabewert der ALU undefiniert sein. Schalten Sie jetzt die ALU durch geeignete Werte an `ALU_OPC` und `ALU_IMM` auf die Ausgabe `Null`, wählen Sie die Register-Zieladresse `0` aus, und aktivieren Sie das Write-Enable der Registerbank (active low!). Beim nächsten Taktimpuls wird dann `R0` auf den Wert `0` gesetzt. Initialisieren Sie auf diese Weise alle Register. Spielen Sie anschließend ein bisschen mit den Steuerleitungen herum, um die Funktion der Registerbank und der ALU zu verstehen.

Aufgabe 3.2: Einfache ALU-Operationen Überlegen Sie sich jetzt, wie Sie durch entsprechende Bedienung der Steuersignale nacheinander die einzelnen Register auf die folgenden Werte setzen können: `R0=0`, `R1=1`, `R2=2`, `R3=4`, `R4=8`, ..., `R10=0200h` (512d). Achtung: Verwenden Sie *nicht* den exernen Eingang, dieser dient nur als Platzhalter für den Rest des Prozessors, etwa zum Laden der

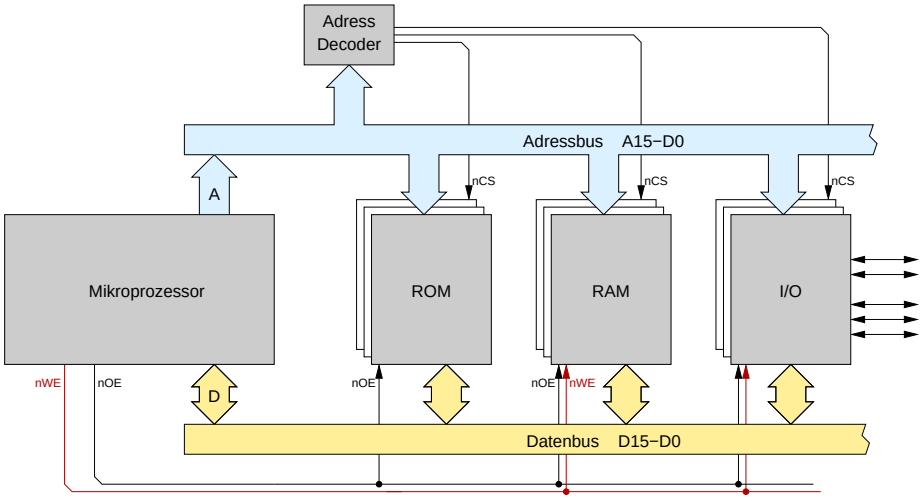


Abbildung 5: Gesamtsystem mit Prozessor, RAM, ROM und I/O Komponenten. Der Bus besteht aus Adress- und Datenbus sowie den Steuerleitungen nWE und nOE.

Register aus dem Speicher. Probieren Sie Ihr „Programm“ schrittweise am Simulator aus.

Dokumentieren Sie, welche Operationen dazu nacheinander ausgeführt werden müssen (Registeradressen `AX` uind `AY`, Alu-Operation, Takte, etc.), da das Programm später noch einmal benötigt wird. Erweitern Sie Ihr Programm anschließend soweit, daß Ihr Geburtsdatum im Format Tag/Monat/Jahr in die Register `R11` bis `R13` abgelegt wird. Tip: die Jahreszahl läßt sich mittels `MOVI` und Addition recht einfach aus den Werten in den Registern `R1` bis `R10` zusammensetzen. Benutzen Sie gegebenenfalls den Windows-Taschenrechner zur Umrechnung zwischen Dezimal- und Hex-Zahlen.

4 Speicheransteuerung

Kernstück des von-Neumann-Rechners ist der gemeinsame, einheitliche Speicher für Daten und Befehle. Praktisch realisiert wird dieses Konzept aber meistens durch Standardkomponenten für RAM (random access memory) und ROM (read only memory), die vom Prozessor über einen gemeinsamen *Bus* angesteuert werden. Häufig werden Bereiche des Speichers auch zur Ansteuerung von I/O-Komponenten verwendet (memory-mapped I/O). Damit diese Komponenten nicht für jeden Rechner vollständig neu entwickelt werden müssen, hat sich eine einheitliche Ansteuerung durchgesetzt. Neben den eigentlichen Daten- und Adressleitungen gibt es dazu noch zwei Steuerleitungen für *Read-Enable* und *Write-Enable*. Dieses Bussystem ist in Abbildung 5 skizziert.

Da die Speicher- und I/O-Bausteine alle gemeinsam an den Bus angeschlossen sind, gibt es zusätzliche *Chip-Select* Leitungen, über die sich die einzelnen Komponenten auswählen lassen. Diese werden von einem *Adressdekoder* angesteuert, der abhängig von der vom Prozessor angelegten Adresse genau (maximal) eine Komponente aktiviert. Beachten Sie, dass zumindest die Kaltstart-Programme des Prozessors (etwa das PC-BIOS) im ROM liegen müssen.

Eigentlich ist ein ROM nur ein einfaches Schaltnetz, das nach Anlegen einer Adresse nach einer gewissen Verzögerung den zugehörigen Datenwert liefert. Für das Auslesen aus dem ROM würde es also zunächst ausreichen, eine Adresse auf den Adressbus zu legen, und einige Zeit später den auf dem Datenbus vorhandenen Wert in das Befehlsregister `IR` zu speichern.

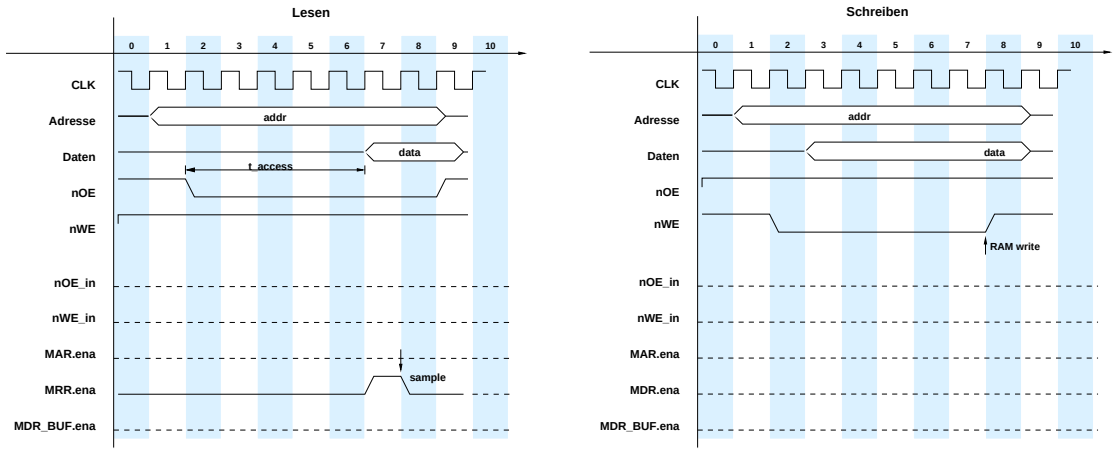


Abbildung 6: Zeitabläufe beim Speicherzugriff (Lesen und Schreiben). Der Lesezugriff wird über die `nOE` Leitung gesteuert, der Schreibzugriff über `nWE`. Nach Aktivieren des Steuersignals sind Wartezyklen notwendig, um die Zugriffszeit des Speichers einzuhalten.

Leider funktioniert diese einfache Ansteuerung aber nicht mit allen erhältlichen Speicer-ICs und I/O-Bausteinen. Statt dessen ist eine kompliziertere Ansteuerung nötig, die in Abbildung 6 gezeigt ist. Aus den Datenblättern der ICs ergibt sich, welche Mindestzeiten zwischen den einzelnen Signaländerungen eingehalten werden müssen. Da alle Zeiten in einem getakteten System immer Vielfache der Taktperiode sind, muss der Prozessor gegebenenfalls *n Wartezyklen* einlegen, bis die Bedingung $n \cdot t_{\text{clk}} > t_{\text{access}}$ erfüllt ist. Zum Beispiel beträgt die typische Zugriffzeit eines normalen RAMs oder eines I/O-Bausteins 200 ns. Falls der Prozessor mit 50 MHz Takt (Taktperiode also 20 ns) betrieben werden soll, sind also für jeden Zugriff mindestens 10 Wartezyklen notwendig, bei 1 GHz Takt bereits 200 Wartezyklen.

Aufgabe 4.1: Speicheransteuerung Laden Sie die Datei `memory.hds`, um sich mit der Speicheransteuerung vertraut zu machen. Das Design enthält je eine RAM- und ROM-Komponente und einige Schalter zur Ansteuerung der Adress-, Daten- und Steuerleitungen. Der *Adressdekoder* wertet nur das oberste Bit 15 der Adresse aus und aktiviert das ROM für die Adressen von `0x0000..0x7fff` und das RAM für `0x8000..0xffff`.

Bei fast allen Mikroprozessoren ist es üblich, Adressen als Byte-Adressen zu interpretieren. Andererseits ist der Speicher meistens wortweise organisiert, hier also mit 16-bit (2 Byte) Wortbreite. Also befinden sich die D·CORE-Adressen `0000` und `0001` im ersten Wort eines Speichers, die Adressen `0002` und `0003` im zweiten Wort, usw. Entsprechend muss die Adresse zum Zugriff auf das nächste Speicherwort immer um `2` inkrementiert werden. In der Hardware wird das einfach dadurch realisiert, dass das unterste Bit des Adressbusses nicht an die Speicherbausteine angeschlossen wird.

Typisch ist auch, dass die Speicher mehrfach in den Adressraum eingeblendet werden. Obwohl das ROM in `memory.hds` nur 1K Worte aufweist, wird es nicht nur im Bereich `0x0000..0x07ff` (Wortadressen!) aktiviert, sondern von `0x0000..0x7fff`. Zugriffe auf die Adresse `0x0002` sind in diesem Fall also äquivalent zu Zugriffen auf `0x0802`, `0x1002`, `0x2002`, usw.

Die zusätzlichen Register `MAR`, `MDR`, `MRR`, `nOE` und `nWE` stellen das Speicherinterface des Prozessors dar. Solche Register sind an allen I/O Leitungen notwendig, um *Hazards* zu vermeiden und Störimpulse vom Prozessor und Speicher fernzuhalten. Das `MAR` (memory address register) puffert die Adresse, das

MDR (memory data register) die vom Prozessor ausgegebenen Daten und das MRR (memory read register) die vom Prozessor eingelesenen Daten. Die nOE (output enable) und nWE (write enable) Register sorgen für saubere Pegel auf den (*low-aktiven!)* Steuerleitungen. Natürlich werden alle Speicherzugriffe durch die zusätzlichen Register um einen Takt langsamer. Um zum Beispiel einen Wert aus Register R6 in den Speicher zu schreiben, muss zunächst der Wert von R6 in das Register MDR übertragen werden. Erst danach kann der eigentliche Speicherzugriff stattfinden.

Öffnen Sie den Editor für das RAM und das ROM, um die Speicherinhalte sehen und ändern zu können. Da noch nichts in die Speicher geschrieben wurde, werden alle Speicherstellen einen undefinierten Wert xxx anzeigen. Sie können die Speicher aber per Menü über `Edit -> Initialize` initialisieren, in eine Datei abspeichern oder aus einer Datei laden.

Stellen Sie jetzt eine Adresse und einen Datenwert ein und aktivieren Sie in der notwendigen Reihenfolge die nCS, nOE und nWE-Leitungen, um diesen Wert in das RAM zu schreiben. Sofern Sie die Farbzuordnung nicht modifiziert haben, zeigt der Editor die zuletzt gelesene und geschriebene Adresse in grün bzw. magenta hervorgehoben an. Wiederholen Sie den Vorgang für einige Adressen und Daten, und versuchen Sie außerdem, die geschriebenen Daten wieder aus dem RAM zu lesen.

Aufgabe 4.2: Speicheransteuerung Ergänzen Sie in Abbildung 6 die notwendigen Eingangs- bzw. Steuersignale für die Signale nOE_in bis MRR.enable, um die gezeigten Verläufe zu realisieren. Als Beispiel ist im linken Diagramm die notwendige Ansteuerung für das MRR-Register gezeigt; dessen Enable-Eingang wird in Takt 7 aktiviert, damit das Register bei der nächsten Taktflanke den Wert vom Datenbus übernimmt.

5 Mikroprogrammierung

Nach der Steuerung von „Hand“ wird es jetzt Zeit für automatische Abläufe. Auch die Zeitabläufe in einem Computersystem lassen sich bequem als endliche Automaten darstellen und spezifizieren. Beim Versuch, einen Rechner wirklich zu bauen, muss diese abstrakte Beschreibung aber in eine *Implementierung* der Spezifikation aus Logikgattern und Zeitgliedern umgesetzt werden. Dazu gibt es mehrere Möglichkeiten.

Im Praktikum und den Übungen zu T1 haben Sie bereits Verfahren kennengelernt, um einfache Automaten als Schaltwerke mit Flipflops und (zweistufiger) Logik für die δ- und λ-Schaltnetze zu realisieren — per Hand mittels KV-Diagrammen oder mit Softwareunterstützung.

Im diesem Praktikum werden wir statt dessen die Technik der *Mikroprogrammierung* einsetzen, mit der komplexe Schaltwerke mit vielen Ausgängen sehr bequem realisiert werden können. Das grundlegende Prinzip eines mikroprogrammierten Schaltwerks ist in Abbildung 7 zusammen mit dem zugehörigen endlichen Automaten skizziert. Das Schaltwerk besteht aus drei Komponenten:

- einem n-bit Register, genannt Mikroprogrammzähler (μPC, micro program counter).
- einem ROM mit 2^n Adressen und m Ausgangsbits. Letztere werden oft in logisch zusammenhängende Felder eingeteilt, z.B. eine Gruppe von n bits als Eingabewert für den μPC.
- etwas Logik zur Auswahl der Eingabedaten für den μPC.

Jeder Zustand des Automaten wird dabei durch den Zustand des Mikroprogrammzählers repräsentiert,

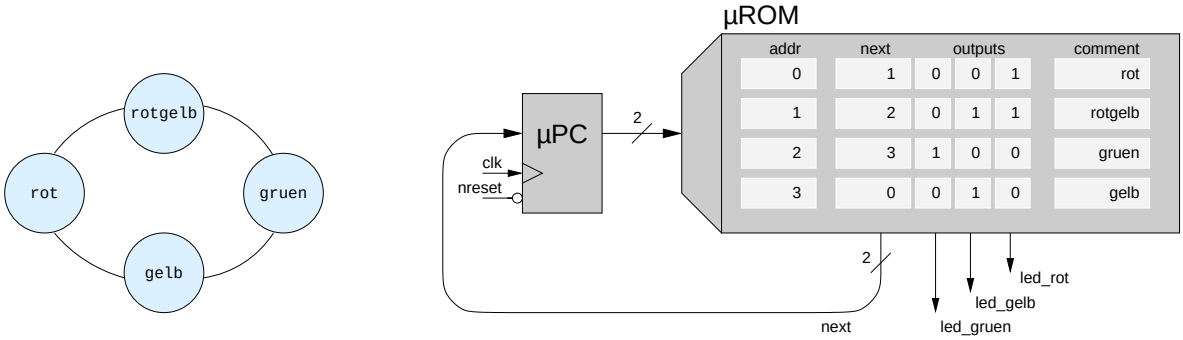


Abbildung 7: Prinzip der Mikroprogrammierung: einfacher Automat (links) und Realisierung mit linearem Mikroprogramm

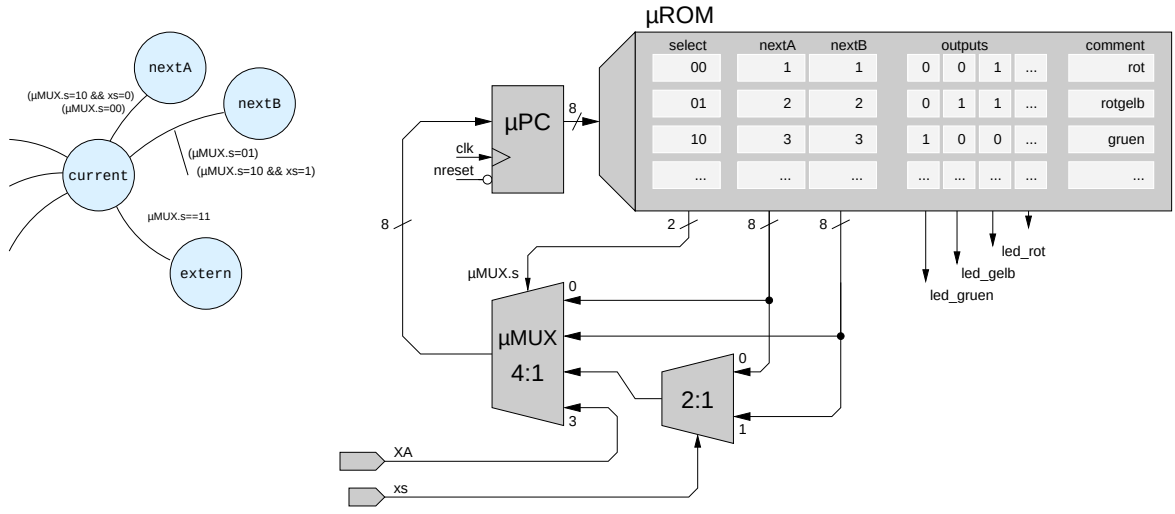


Abbildung 8: Mikroprogrammiertes Steuerwerk mit zwei externen Eingängen `xs` und `XA` und vierfacher Auswahl des Folgezustands (siehe Text).

der das zugehörige Speicherwort im Mikroprogrammspeicher adressiert. Die Ausgangswerte des Mikroprogrammspeichers liefern dann die Ausgangssignale (das λ-Schaltnetz des Automaten).

Zur Auswahl des Nachfolgezustands muss ein neuer Wert in den Mikroprogrammzähler geladen werden. Hierzu (also für das δ-Schaltnetz) sind viele Varianten möglich. Abbildung 7 zeigt die einfachste davon: hier wird der μPC bei jedem Taktimpuls mit dem Wert von μROM.next geladen; dieser Automat kann also nur lineare Zustandsfolgen ohne Verzweigungen realisieren. (Das letzte Feld μROM.comment dient nur der Veranschaulichung und wird in der Hardware natürlich nicht realisiert.)

Realistischer ist das in Abbildung 8 dargestellte Steuerwerk. Es verfügt über vier Möglichkeiten, einen Folgezustand auszuwählen. Die Auswahl erfolgt durch Ansteuerung des 4:1-Multiplexers μMUX über zwei Steuerleitungen aus dem Mikroprogramm. Für μMUX.s=00 ergibt sich der Folgezustand direkt aus dem Feld uROM.nextA des Mikroprogrammspeichers; bei 10 wird abhängig vom externen Steuersignal xs entweder uROM.nextA oder uROM.nextB in der μPC geladen. Für 11 schließlich wird der μPC mit dem externem Wert XA geladen.

Aufgabe 5.1: Lauflicht Laden Sie die Beispieldatei `sequencer.hds`. Diese enthält den Mikroprogrammspeicher µROM mit zugehörigem Mikroprogrammzähler µPC und einige LEDs. Selektieren Sie den Eintrag *Edit* aus dem Kontextmenü des Mikroprogrammspeichers, um den Mikroprogramm-Editor zu öffnen. Dort können Sie jetzt die Speicherinhalte direkt modifizieren (die Einzelbits lassen sich auch durch Doppelklicken umschalten). Alternativ können Sie auch eine Textdatei mit den gewünschten Speicherinhalten erstellen und dann in den Mikroprogrammspeicher laden.

Schreiben Sie ein Mikroprogramm zur Ansteuerung einiger LEDs als Lauflicht, indem Sie für jeden Schritt die Ansteuerung der einzelnen LEDs und den Nachfolgezustand des µPC definieren. Realisieren Sie zwei verschiedene periodische Muster, zwischen denen über den externen Eingang `xs` umgeschaltet werden kann. Damit Ihr Programm nach einem Reset sauber anläuft, muss der erste Mikroprogrammschritt fest an Adresse `0` liegen. Hinweis: Eventuell müssen Sie die laufende Simulation anhalten (⏮-Button) und neu starten (⏭-Button), damit der Simulator ein geändertes Mikroprogramm korrekt übernimmt. Speichern Sie Ihr Mikroprogramm (z.B. als `lauflicht.rom`) und tragen Sie die Daten auch in die folgende Tabelle ein:

			MUX	LED	
addr	nextA	nextB	4:1	f e d c b a 9 8 7 6 5 4 3 2 1 0	label
00	01	**	00	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1	state01
01					
02					
03					
04					
05					
06					
07					
08					
09					
0a					
0b					

6 Befehl-Holen

Nachdem alle einzelnen Komponenten jetzt vorhanden sind, 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.

Aufgabe 6.1: 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

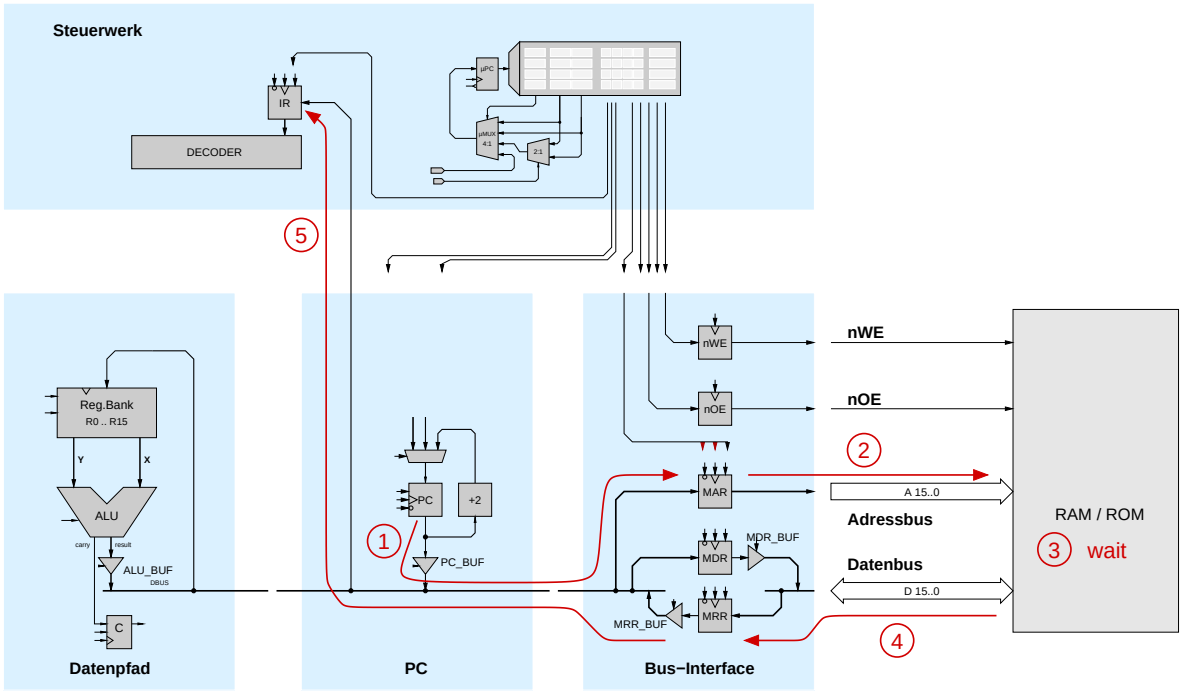


Abbildung 9: Speicherinterface und Komponenten für die Befehl-Holen Phase

ALU, den Programmzähler, das Speicherinterface und schließlich RAM und ROM.

Aufgabe 6.2: 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 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 und die beiden Register `PC` und `IR` benötigt. Dies ist in Abbildung 9 illustriert.

Öffnen Sie den Editor für den Mikroprogrammspeicher. Erstellen Sie ein Mikroprogramm für die *Befehl Holen*-Phase des von-Neumann Rechners, das den addressierten Befehl 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 μ`ROM` beginnen.

Überlegen Sie sich, welche Registertransfers für diese Operation notwendig sind, um die Zeitbedingungen aus Abbildung 6 unter Berücksichtigung der Register `MAR`, `MDR` und `MRR` einzuhalten. Verwenden Sie (mindestens) vier Wartezyklen.

Im Mikroprogramm befinden sich links die Steuersignale für das Steuerwerk selbst (`nextA`, `nextB`, μ`MUX`), 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`).

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	

7 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. 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 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 7.1: Decode Erweitern Sie Ihren Microcode aus Aufgabe 6 um die Befehlsdekodierung (μPC := XA) und das Inkrementieren des PC. 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 13 und speichern Sie das Mikroprogramm (z.B. als `fetch-decode.rom`).

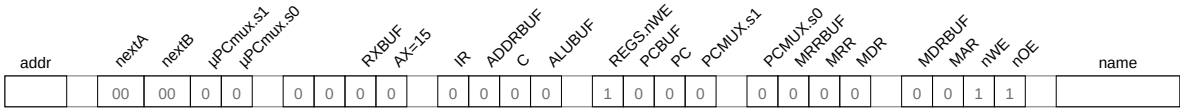
8 Befehlsausführung

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.

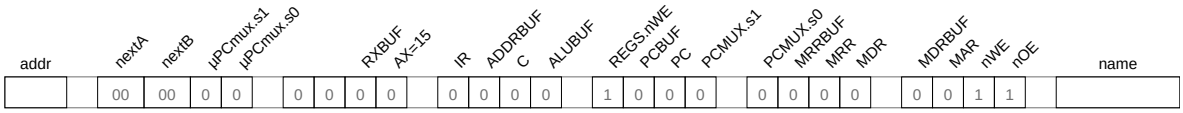
Aufgabe 8.2: HALT-Befehl Der einfachste Befehl ist der HALT Befehl, der auf den ersten Blick unsinnig erscheint. Tatsächlich verfügen aber viele moderne Prozessoren über einen solchen Befehl, um Teile des Prozessors abschalten zu können und damit Strom zu sparen. Im D·CORE dient der HALT-Befehl aber 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	



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 8.4: 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 8.5: Register-Initialisierung Verwenden Sie ihre Notizen von Aufgabe 3, 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 8.6: 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 0002 0004 0006	0x3401	movi R1, 0	R[1]=0
	0008 000a 000c 000e			
	0010 0012 0014 0016			
	0018 001a 001c 001e			

Aufgabe 8.7: 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			

Das zweite Aufgabenblatt des Praktikums dient dazu, die noch fehlenden Befehle des D·CORE zu implementieren und an kleinen Maschinenprogrammen auszutesten. Dies betrifft insbesondere die Load/Store-Befehle und die Sprungbefehle. Bitte wechseln Sie also auf Blatt T3-2 über, sobald Sie mit diesen Aufgaben fertig sind.

Literaturempfehlungen

- Tanenbaum, Goodman: *Computerarchitektur*, 4. Auflage, Prentice-Hall 1999
- Hennessy, Patterson: *Computer Organziation and Design — the Hardware-Software Interface*, Morgan Kaufmann1994
- Schiffmann, Schmitz: *Technische Informatik 2*, 3. Auflage, Springer 1999
- Vorlesungsmitschrift und Lösungen der Übungsaufgaben zu T1 bis T3
- Motorola: Mcore-Dokumentation, www.motorola.com/mcore
- Hades Dokumentation, tech-www.informatik.uni-hamburg.de/applets/hades