

# Praktikum Rechnerstrukturen

2

## Mikroprogrammierung Speicheransteuerung

Name, Vorname	
Bogen bearbeitet	abzeichnen lassen

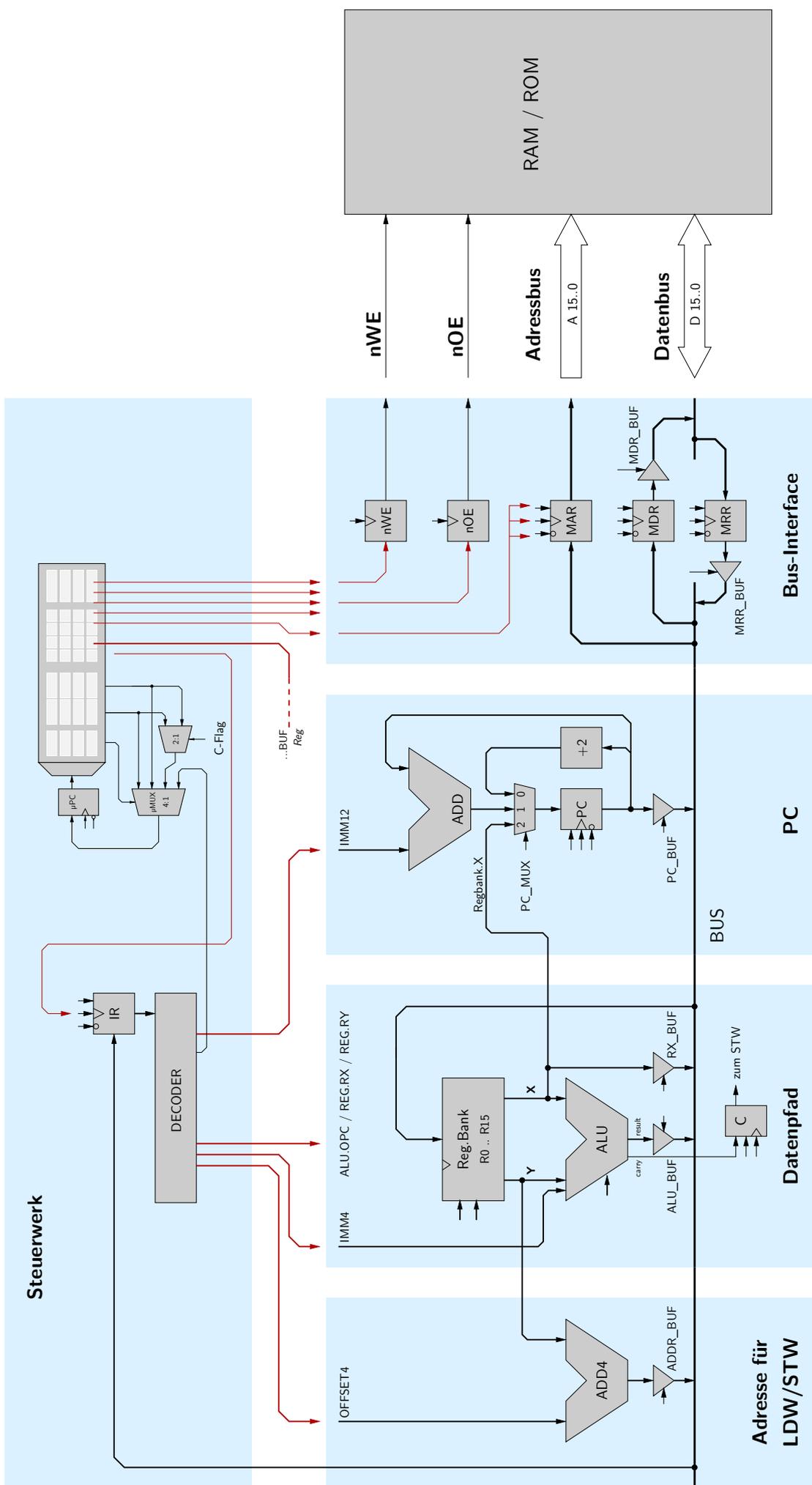


Abb. 1: Blockschaltbild des D-CORE Prozessors

## 1 Mikroprogrammierung

Auf Dauer ist die Steuerung „von Hand“ natürlich unbefriedigend, so dass sich die Frage stellt, ob sich diese Abläufe nicht auch automatisieren lassen. Eine solche Möglichkeit zu schaffen, ist das Thema dieses Abschnitts.

Die Zeitabläufe in einem Computersystem lassen sich bequem als sog. 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.

In Vorlesung und den Übungen haben Sie vielleicht bereits Verfahren kennengelernt, um einfache Automaten als Schaltwerke mit Flipflops und (zweistufiger) Logik für das Schaltnetz zur Berechnung der Folgezustände (das Zustandsüberführungsschaltnetz oder kurz das  $\delta$ -Schaltnetz) und das Schaltnetz zur Berechnung der Ausgaben (das Ausgabeüberführungsschaltnetz oder kurz  $\lambda$ -Schaltnetz) zu realisieren. Dieses kann per Hand z. B. mittels KV-Diagrammen oder mit Softwareunterstützung geschehen.

In diesem Praktikum werden wir den Automaten, der das Steuerwerk implementiert, nicht wie oben angedeutet als feste Schaltung (hard-wired) aufbauen, sondern 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 2 zusammen mit dem zugehörigen endlichen Automaten skizziert (Kreise für die Zustände und ihre Ausgabe und Pfeile für die Zustandsübergänge). Das entsprechende Schaltwerk besteht aus drei Komponenten:

- einem  $n$ -bit Register, genannt Mikroprogrammzähler ( $\mu$ PC, 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  $\mu$ PC.
- etwas Logik zur Auswahl der Eingabedaten für den  $\mu$ PC.

Jeder Zustand des Automaten wird dabei durch den Zustand des Mikroprogrammzählers repräsentiert, der das zugehörige Speicherwort im Mikroprogramm Speicher adressiert. Die Ausgangswerte des Mikroprogramm Speichers liefern dann die Ausgangssignale (das  $\lambda$ -Schaltnetz des Automaten).

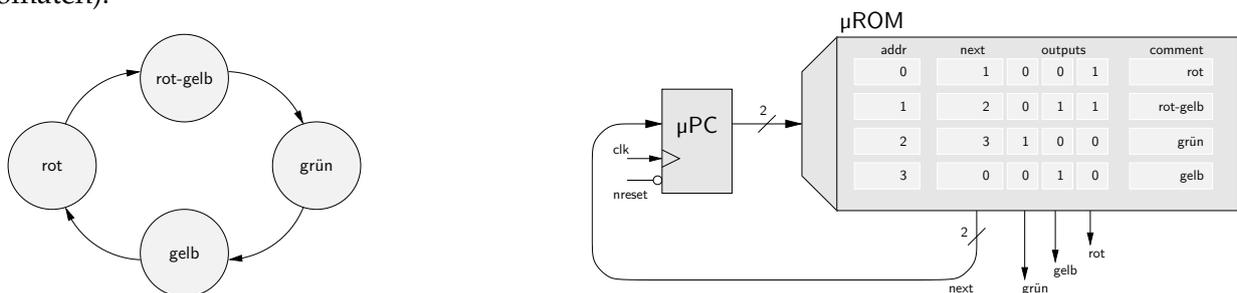


Abb. 2: Prinzip der Mikroprogrammierung: einfacher Automat (links) und Realisierung mit linearem Mikroprogramm

Zur Auswahl des Nachfolgezustands muss ein neuer Wert in den Mikroprogrammzähler geladen werden. Hierzu (also für das  $\delta$ -Schaltnetz) sind viele Varianten möglich. Abbildung 2 zeigt die

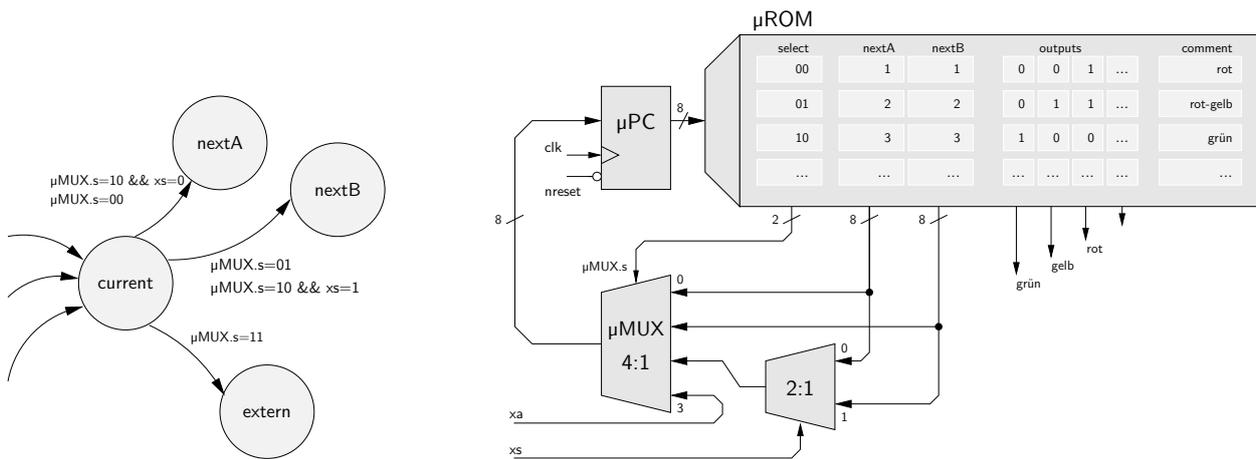


Abb. 3: Mikroprogrammiertes Steuerwerk mit zwei externen Eingängen xs und xa und vierfacher Auswahl des Folgezustands (siehe Text).

einfachste davon: hier wird der  $\mu PC$  bei jedem Taktimpuls mit dem Wert von  $\mu ROM.next$  geladen; dieser Automat kann also nur lineare Zustandsfolgen ohne Verzweigungen realisieren. (Das letzte Feld  $\mu ROM.comment$  dient nur der Veranschaulichung und wird in der Hardware natürlich nicht realisiert.)

Realistischer ist das in Abbildung 3 dargestellte Steuerwerk, das sich so auch in dem Steuerwerk des D-CORE befindet. Es verfügt über vier Möglichkeiten einen Folgezustand auszuwählen. Dazu wird der 4:1-Multiplexer  $\mu MUX$  über zwei Steuerleitungen aus dem Mikroprogramm angesteuert:

- $\mu MUX.s=00$  Der Folgezustand ergibt sich direkt aus dem Feld  $\mu ROM.nextA$  des  $\mu ROMs$ .
- $\mu MUX.s=01$  Der Folgezustand ergibt sich direkt aus dem Feld  $\mu ROM.nextB$  des  $\mu ROMs$ .
- $\mu MUX.s=10$  Es wird, abhängig vom externen Steuersignal  $xs$  entweder  $\mu ROM.nextA$  oder  $\mu ROM.nextB$  in den  $\mu PC$  geladen.
- $\mu MUX.s=11$  Der Mikroprogrammzähler  $\mu PC$  wird mit dem externen Wert  $xa$  geladen.

Machen Sie sich diese Zusammenhänge anhand Abb. 3 klar.

**Aufgabe 2.1** Mikroprogrammierung

Geben Sie jeweils den Wert an, der als nächster in den  $\mu PC$  geladen wird. Nehmen Sie hierzu das Schaltwerk aus Abb. 3 zu Hilfe und machen Sie sich die Funktion anhand des Schaltbildes klar:

$\mu MUX.s$	nextA	nextB	xs	xa	neuer $\mu PC$
00	01	0a	0	02	
01	01	0a	1	ff	
10	01	0a	0	00	
10	10	0a	0	01	
10	10	0b	1	11	
11	aa	bb	0	cc	

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

**Aufgabe 2.2** Ampelschaltung

Laden Sie die Beispieldatei **sequencer.hds**. Diese enthält den Mikroprogrammspeicher  $\mu$ ROM mit zugehörigem Mikroprogrammzähler  $\mu$ PC gemäß Abbildung 3 und einige LEDs für die Ausgabe des mikroprogrammierten Automaten. 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 Ampelschaltung an einer Kreuzung. Den Dioden D0, D1, D2 bzw. D4, D5, D6 sollten bereits die richtigen Farben rot, gelb, grün haben.

Wenn der externe Eingang *xs* mit den Wert 0 belegt ist ( $xs = 0$ ), soll der Automat zyklisch die rechts skizzierte Folge durchlaufen und die Ampel-LEDs dementsprechend ansteuern.

Falls der externe Eingang  $xs = 1$  ist, sollen beide Ampeln synchron gelb blinken.

Ampel 1	Ampel 2
rot	rot
rot/gelb	rot
grün	rot
grün	rot
gelb	rot
rot	rot
rot	rot/gelb
rot	grün
rot	grün
rot	gelb

**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 *ampel.mic*) und tragen Sie die Daten auch in die folgende Tabelle ein:

addr	nextA	nextB	$\mu$ MUX.s (4:1)	LED	Kommentar
				7 6 5 4 3 2 1 0	
00				0 0 0 1 0 0 0 1	
01					
02					
03					
04					
05					
06					
07					
08					
09					
0a					
0b					

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

**Aufgabe 2.3** Automatisierte Berechnung

Jetzt haben wir auch eine Möglichkeit, die ALU-Operationen aus Bogen 1, Aufgabe 1.9 zu automatisieren. Laden Sie dazu die Datei **datapathMR.hds** in HADES.

**datapathMR.hds** realisiert die Verbindung von Registerbank und ALU. Über das mikroprogrammierten Steuerwerk lassen sich die Abläufe steuern, indem der ALU-Opcode und SELY, SELX und das Enable-Signal der Registerbank beeinflusst werden. Im Prinzip hätte man auch noch das Enable des Tristate-Treibers am Ausgang der ALU und das Enable des Carry-Registers mit in das Mikroprogramm aufnehmen können. Darauf wurde aber verzichtet, weil sie für unsere Zwecke immer auf 1 bleiben können.

Man beachte, dass der Ausgang des Carry-Registers auf den Select-Eingang des 2-1-Multiplexers geführt ist, sodass sich auch bedingte Sprünge realisieren lassen, wenn man, wie bei der Ampelschaltung aus der letzten Aufgabe, den geeigneten Eingang des 4-1-Multiplexers auswählt. Man bedenke dabei, dass das Ergebnis eines Vergleichs erst einen Takt später wirklich abgeprüft werden kann, weil es durch das Carry-Flipflop verzögert wird. Ein Vergleich und seine Auswertung dauern also immer zwei Takte bzw. benötigen zwei Zeilen im Mikroprogramm.

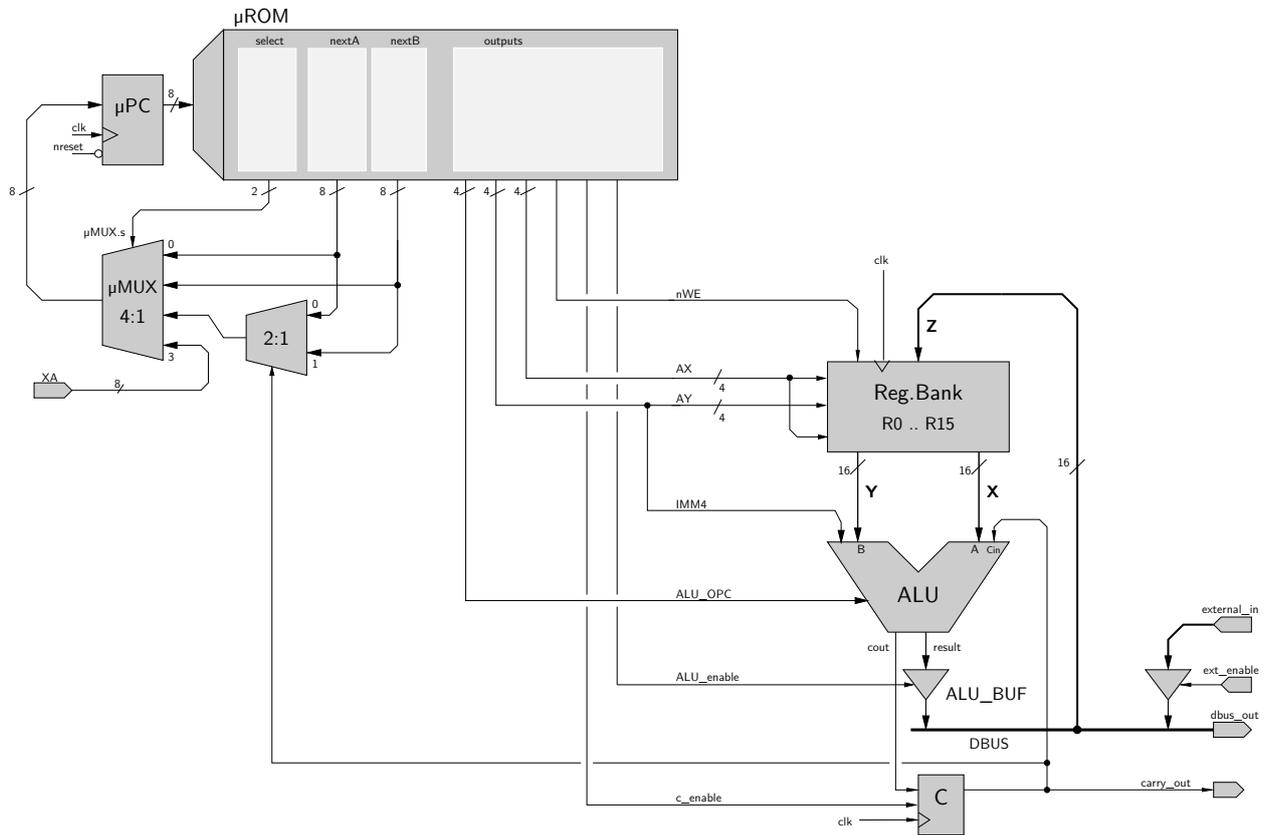


Abb. 4: Steuerwerk als Mustergenerator für Operationswerk

Die Tabelle aus Bogen 1, Aufgabe 1.9 sollte sich z.B. direkt als Mikroprogramm übernehmen und abarbeiten lassen, wenn man noch in jedem Schritt das Enable-Signal der Registerbank (nWE) auf 0 setzt.

**Beispiel:** Bedingter Sprung

Der Betrag einer Zahl, die im Register R0 steht, soll berechnet und in das Register R2 geschrieben werden. Das lässt sich zum Beispiel wie folgt realisieren:

addr	nextA	nextB	$\mu$ MUX.s (4:1)	ALUOP	SelX	SelY	REGs nWE	Kommentar
00	01	00	00	14	0	0	0	R0=0 (zum Test)
01	02	00	00	16	1	0	0	R0=R0-1 (zum Test R0=-1)
02	03	00	00	14	0	2	0	R2=0 (zum Vergleich mit 0)
03	04	00	00	13	2	0	0	Prüfen, ob R0<R2
04	06	05	10	**	*	*	1	Auswerten des Vergleichs
05	07	00	00	03	0	2	0	R2=R2-R0=-R0 (Zahl war < 0)
06	07	00	00	00	0	2	0	R2=R0 (Zahl war $\geq$ 0)
07								

Man beachte, dass auf Adresse 04 der Inhalt einiger Felder ohne Bedeutung ist, weil nichts in die Registerbank geschrieben wird (REGs.nWE=1!)

**Aufgabe 2.4** Realisierung einer Programmschleife auf Maschinenebene

Schreiben Sie ein Mikroprogramm, das den rechts stehenden Ablauf realisiert und testen Sie es aus. Als Ergebnis sollten Sie 9 erhalten. Lassen Sie Ihr Programm auch mit anderen kleinen Startwerten für R0 laufen und stellen Sie eine Vermutung auf, was das Programm berechnet.

```

R0= 3
R1= 0
R2= 0
while ( R2 != R0) do
{ R1= R1 + R2
  R1= R1 + R2
  R2= R2 + 1
}
R1= R1 + R0
stop /* Totschleife */
    
```

addr	nextA	nextB	$\mu$ MUX.s (4:1)	ALUOP	SelX	SelY	REGs nWE	Kommentar
00								
01								
02								
03								
04								
05								
06								
07								
08								
09								
0a								
0b								

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

## 2 Speichermanagement

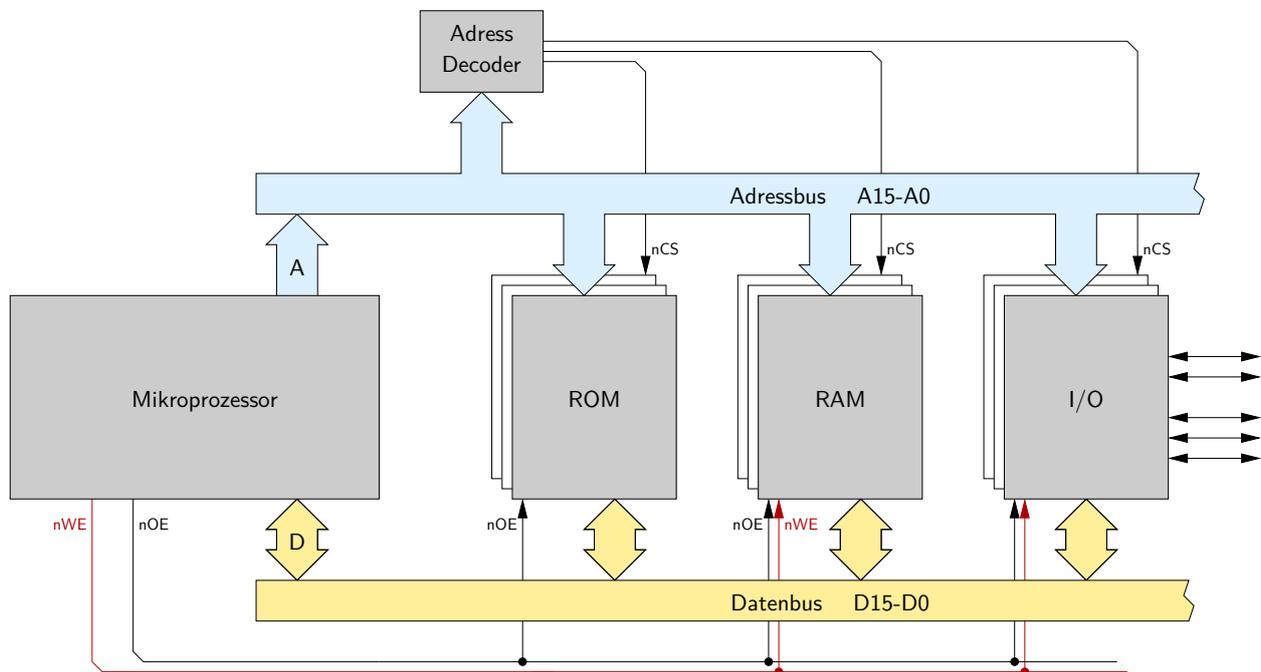


Abb. 5: Gesamtsystem mit Prozessor, RAM, ROM und I/O Komponenten. Der Bussystem zur Ansteuerung des Speichers besteht aus Adress- und Datenbus sowie den Steuerleitungen nWE und nOE.

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 (**R**andom **A**ccess **M**emory) und ROM (**R**ead **O**nly **M**emory), 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* (nOE) und *Write-Enable* (nWE). Dieses Bussystem ist in Abbildung 5 skizziert.

Da die Speicher- und I/O-Bausteine alle gemeinsam an den Bus angeschlossen sind, gibt es zusätzliche, üblicherweise low-aktive *Chip-Select* Leitungen (nCS), über die sich die einzelnen Komponenten auswählen lassen. Diese werden von einem *Adressdecoder* 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.

Leider funktioniert diese einfache Ansteuerung aber nicht mit allen erhältlichen Speicher-ICs und

I/O-Bausteinen. Statt dessen ist eine kompliziertere Ansteuerung nötig, die in Abbildung 7 (oberer pink hinterlegter Bereich der Diagramme) 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  $n \cdot t_{clk} > t_{access}$  erfüllt ist. Zum Beispiel beträgt die typische Zugriffszeit 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 einem 1 GHz Takt bereits 200 Wartezyklen.

**Aufgabe 2.5 Speicheransteuerung**

Laden Sie die Datei **memoryMR.hds**, um sich mit der Speicheransteuerung vertraut zu machen. Sie sollten folgendes Bild zu sehen bekommen:

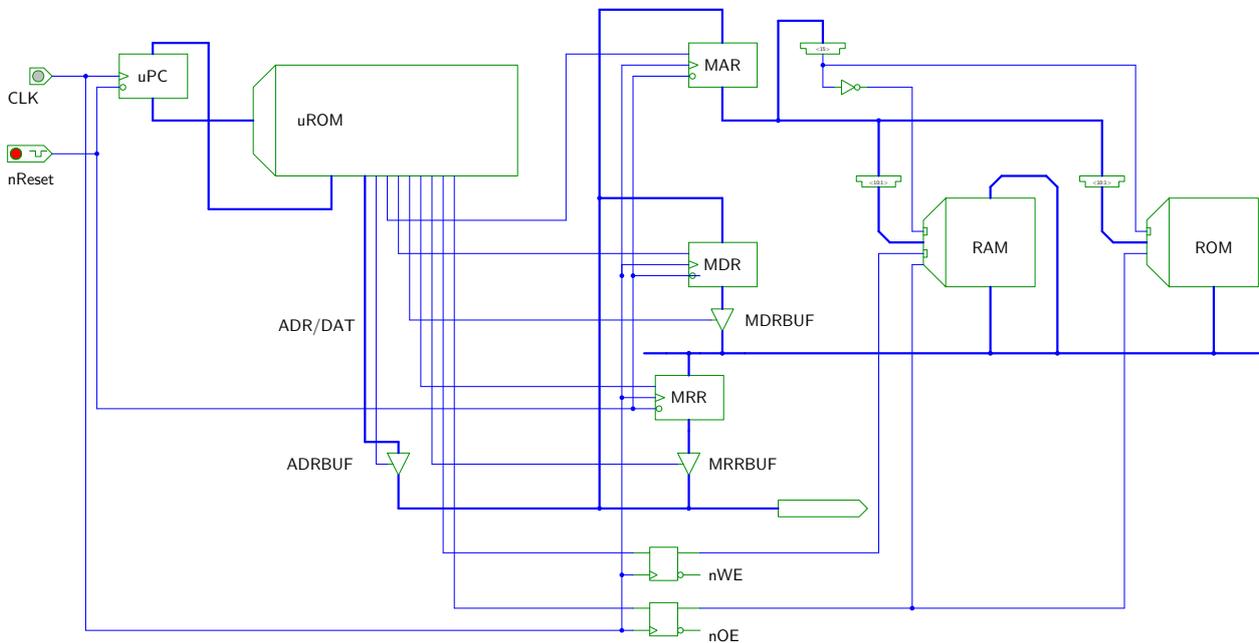


Abb. 6: Hades-Modell

Das Design enthält je eine RAM- und ROM-Komponente und ein Mikroprogramm-ROM zur Ansteuerung der Adress-, Daten- und Steuerleitungen. Der *Adressdecoder* 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.

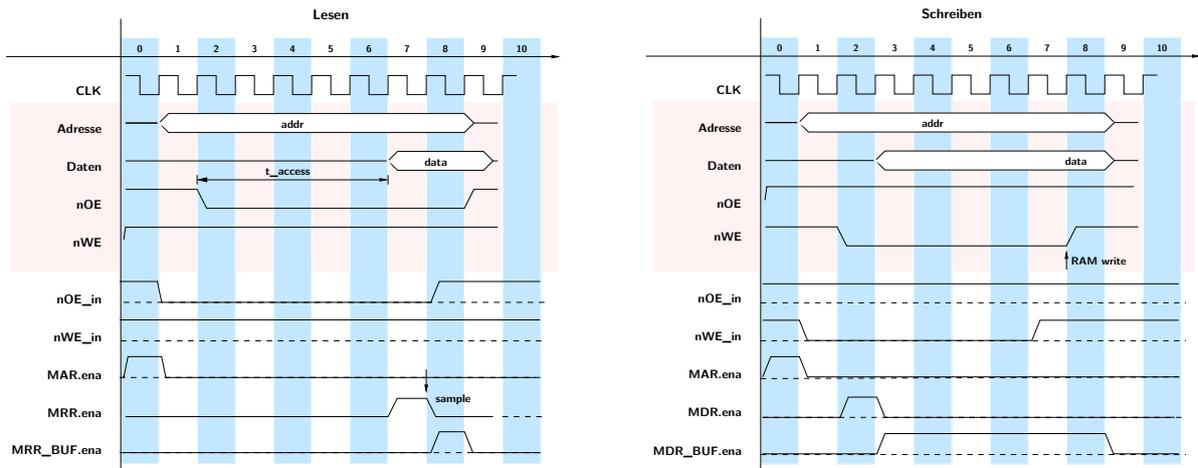


Abb. 7: Zeitabläufe beim Speicherzugriff (Lesen und Schreiben). Der Lesezugriff wird über die nOE Leitung gesteuert, der Schreibzugriff über nWE. Nach Aktivieren des Steuersignals sind Wartezyklen notwendig, um die Zugriffszeit des Speichers einzuhalten. Adresse, Daten, nOE und nWE sind Signale, die direkt am Speicherbaustein liegen, während nOE\_in, nWE\_in, MAR.ena, MDR.ena und MDR\_Buf.ena Signale sind, die an den entsprechenden Bauteilen (Register, Flip-Flops, Buffer, siehe Bogen 1 und vgl. auch Abb. 6) des Speicherinterfaces anliegen.

Typisch ist auch, dass die Speicher mehrfach in den Adressraum eingeblendet werden. Obwohl das ROM in **memoryMR.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** (**M**emory-**A**ddress-**R**egister), **MDR** (**M**emory-**D**ata-**R**egister), **MRR** (**M**emory-**R**ead-**R**egister), **nOE** (**n**ot-**O**utput-**E**nable) und **nWE** (**n**ot-**W**rite-**E**nable) 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 puffert dabei die Adresse, das MDR die vom Prozessor ausgegebenen Daten, die in den Speicher geschrieben werden sollen und das MRR die Daten, die aus dem Speicher ausgelesen wurden. Die nOE (output enable) und nWE (write enable) Register sorgen für saubere Pegel auf den (*low-aktiven!*) Steuerleitungen. Das heißt für eine 0 auf der Leitung nOE wird der Wert, der an der im MAR gespeicherten Adresse im Speicher steht, ausgelesen. Für eine 0 auf der Leitung nWE wird entsprechend der Wert, der im MDR steht, in den Speicher geschrieben, wobei man natürlich nicht vergessen darf, ihn über den entsprechenden Tristate-Treiber auf den Bus zu legen. Sonst würden Werte vom „floatenden“ Speicherbus gelesen und in den Speicher geschrieben. Natürlich werden alle Speicherzugriffe durch die zusätzlichen Register um einen Takt langsamer. Um zum Beispiel einen Wert in den Speicher zu schreiben, muss zunächst der Wert 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 zumindest im RAM alle Speicherstellen einen undefinierten Wert **XXXX** anzeigen. Sie können die Speicher aber per Menü über Edit -> Initialize initialisieren, in eine Datei abspeichern oder aus einer Datei laden.

**Aufgabe 2.6** Speicheransteuerung

Schreiben Sie folgende Mikroprogramme:

- (a) Der Wert `0x0ff0` wird auf die erste Speicherstelle im RAM geschrieben.  
 Adresse und Daten lassen sich dabei über das Feld *ADR/DAT* im Mikroprogramm angeben und durch Aktivieren des Tristate-Treibers *ADRBUF* auf den Bus legen, an den das *MAR* und das *MDR* angeschlossen sind.
- (b) Es wird ein Wert von von der Byte-Adresse `0x0100` aus dem ROM gelesen und dieser Wert dann auf die Adresse `0x8004` ins RAM geschrieben.  
 An der entsprechenden Wort-Adresse sollte bereits etwas im ROM stehen. Prüfen Sie dies bitte.

addr	nextA	nextB	4:1	ADR/DAT	ADRBUF	MAR	MDR	MDRBUF	MRR	MRRBUF	nWE	nOE
00												
01												
02												
03												
04												
05												
06												
07												
08												
09												
0a												
0b												

Sofern Sie die Farbzuoordnung nicht modifiziert haben, zeigt der Editor die zuletzt gelesene und geschriebene Adresse in grün bzw. magenta hervorgehoben an.

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

### 3 Implementierung des Prozessors

Nachdem wir die Komponenten unseres Prozessors einzeln betrachtet haben, ist es an der Zeit, den kompletten D-CORE-Prozessor (siehe Abb. 1) zu untersuchen. Dabei ist die Hardwarestruktur mit allen Registern, der ALU und dem Steuerwerk bereits fertig vorgegeben — es fehlt jedoch das Mikroprogramm.

Ziel der nächsten Aufgaben ist es, schrittweise ein Mikroprogramm zu erstellen, um einen funktionsfähigen Prozessor auf der gegebenen Hardwarestruktur zu erhalten, der die Befehle der Tabelle 1 aus Bogen 1 abarbeiten kann.

Damit der Prozessor überhaupt einen Befehl ausführen kann, muss in einem ersten Schritt der zur Ausführung anstehende Befehl in den Prozessor geladen werden (Befehl-Holen). Anschließend muss der Befehl in seine Bestandteile zerlegt werden (Befehl-Decodieren), so dass er vom Prozessor verarbeitet werden kann (Befehl-Ausführen). Die Abfolge von Befehl-Holen (fetch), Befehl-Decodieren (decode) und Befehl-Ausführen (execute) definiert den Befehlszyklus.

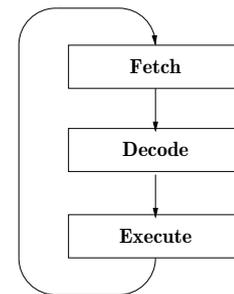


Abb. 8: Befehlszyklus des D-CORE Prozessors

#### 3.1 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.

Öffnen Sie das Design **processor.hds** mit der vollständigen Logik des D-CORE-Prozessors inklusive Datenpfad, Speicherinterface, Befehlsdecoder und Programmzähler. Verwenden Sie gegebenenfalls die Zoom-Funktion des Hades-Editors, um die gesamte Schaltung sehen zu können. Vergleichen Sie das Hades Schaltbild mit der Abbildung 1 dieses Aufgabenblattes.

Der obere Teil des Schaltplans enthält das Steuerwerk mit dem Befehlsregister IR, dem Mikroprogrammzähler  $\mu$ PC 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 2.7 Mikroprogramm für Befehl Holen

In der ersten Phase des Befehlszyklus des von-Neumann Rechners (siehe Abb. 8) wird ein Befehl aus dem Speicher (ROM oder RAM) in das Befehlsregister IR übertragen. 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 9 illustriert.

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

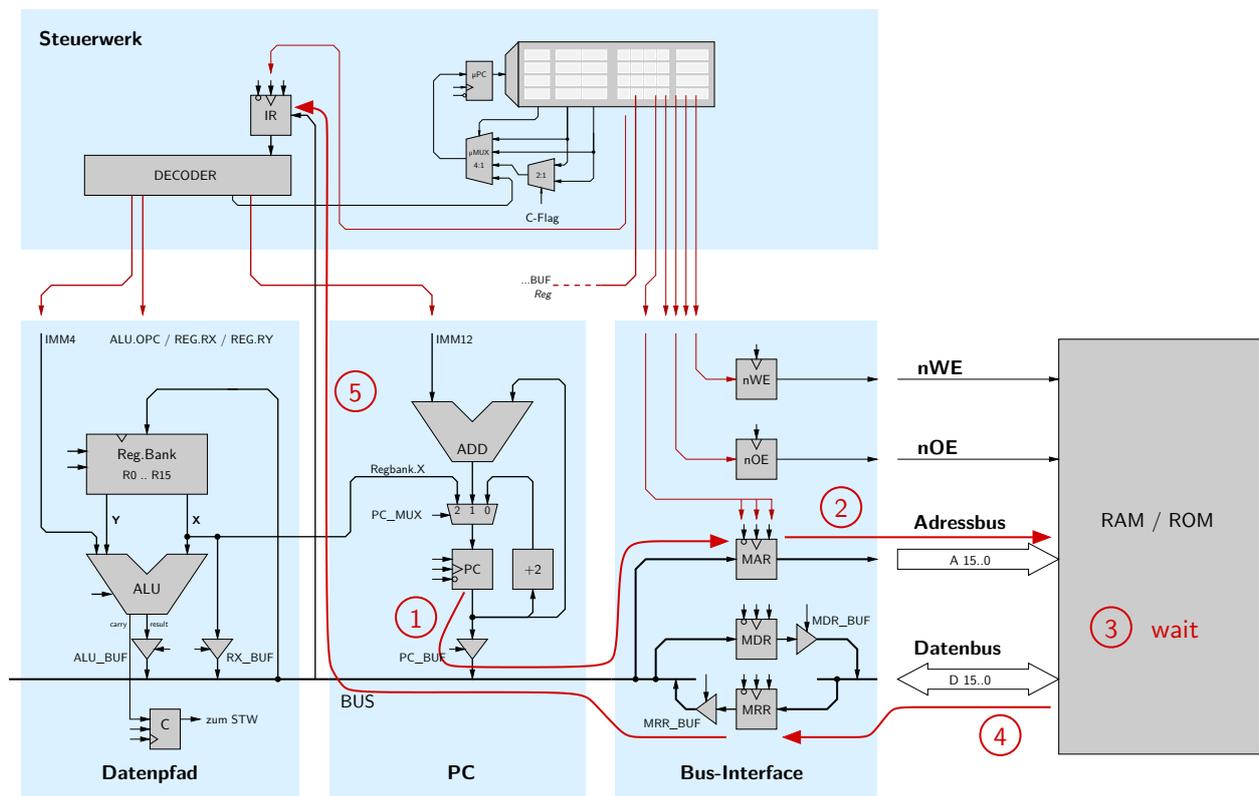


Abb. 9: Speicherinterface und Komponenten für die Befehl-Holen Phase

Ihre Aufgabe ist jetzt die Erstellung eines Mikroprogramms für die *Befehl Holen*-Phase des von-Neumann Rechners, das den Befehl, dessen Adresse 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  $\mu$ ROM beginnen.

Wie schon in Abbildung 9 angedeutet ist, sind folgende Schritte nötig:

1. Der Wert des PCs muss in das MAR gebracht werden.
2. Dem Speicher muss mitgeteilt werden, dass man lesend auf ihn zugreifen möchte. Dabei ist zu beachten, dass der Speicher (mindestens) drei Waitstates braucht.
3. Der Wert, der nach den Waitstates aus dem Speicher kommt, muss in das MRR geschrieben werden.
4. 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 Mikroprogramm Speicher ein.

Speichern Sie das Mikroprogramm, z.B. als Datei `fetch.mic`. 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	$\mu$ PCmux.s1	$\mu$ PCmux.s0	RXBUF AX=15	IR	ADDRBUF C	ALUBUF	REGS.nWE PCBUF PC	PCMUX.s1	PCMUX.s0	MIRBUF MIR	MDR	MDRBUF MAR	nWE	nOE	name	
00	00	00	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	fetch_1
01	00	00	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	fetch_2
02	00	00	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	fetch_3
03	00	00	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	fetch_4
04	00	00	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	fetch_5
05	00	00	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	fetch_6
06	00	00	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	fetch_7
07	00	00	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	fetch_8
08	00	00	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	
09	00	00	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	
0a	00	00	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	
0b	00	00	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	
0c	00	00	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	
0d	00	00	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	decode
0e	00	00	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	
0f	00	00	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	

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

### 3.2 Befehlsdecodierung

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 codiert 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 Mikroprogrammschritten 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, das 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  $\mu$ PC auf den entsprechenden Wert gesetzt wird. Im Steuerwerk aus Abbildung 3 dient dazu der externe Eingang *xa*, über den ein externer Wert direkt in den  $\mu$ PC geladen werden kann. In der Decode-Phase muss also der Multiplexer  $\mu$ MUX so angesteuert werden, dass der externe Eingang *xa* in den  $\mu$ PC geladen wird.

Das Steuerwerk des D-CORE ist entsprechend aufgebaut: die obersten vier Bits des Befehlsregisters IR –der Opcode– 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

dem Befehlscode  $(1100 \langle **** \rangle \langle **** \rangle \langle xxxx \rangle) = (C**x)_{16}$  an der Mikroprogrammadresse  $(1100\ 0000) = (C0)_{16}$  das Mikroprogramm für den HALT-Befehl mit Opcode  $(1111 \langle **** \rangle \langle **** \rangle \langle **** \rangle) = (F***)_{16}$  an der Adresse  $(1111\ 0000) = (F0)_{16}$  usw.

**Aufgabe 2.8** Decode-Phase

Welche Werte werden jeweils in der Decode-Phase in den  $\mu PC$  geschrieben, wenn folgende Werte im IR stehen?

IR	$\mu PC$
0x7000	
0xF000	
0x2000	
0x2123	
0x2104	
0x2e00	

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 (Byteadressen) inkrementiert werden, um auf das nächste Speicherwort zu zeigen.

**Aufgabe 2.9** Decode

Erweitern Sie Ihren Microcode aus Aufgabe 2.7 um die Befehlsdecodierung ( $\mu PC := XA$ ) 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 14 und speichern Sie das Mikroprogramm (z.B. als fetch-decode.mic).

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

### 3.3 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 der Tabelle 1 aus Bogen 1 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 Mikroprogramme zur Ansteuerung der verschiedenen Enable-Leitungen für Register und Tri-State-Treiber.

Wählen sie dabei folgendes Vorgehen: Schauen Sie sich zuerst die in Tabelle 1 aus Bogen 1 in Spalte „Bedeutung“ definierte Wirkung des Befehles an. Dann überlegen Sie, wie sich dieser Befehl auf der vorgegebenen Hardware (**processor.hds**) realisieren lässt; oder konkret, welche Steuersignale müssen aktiviert werden, damit die Wirkung des Befehles erzielt wird.

#### Aufgabe 2.10 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:  $\mu PC.nextA = \mu PC$ .

Tragen Sie den entsprechenden Mikroprogrammschritt in der untenstehenden exemplarischen Mikroprogrammzeile ein (die vorgegebenen ausgegrauten Einträge stellen die Standardwerte bei Inaktivität dar, die Sie für den jeweiligen Befehl entsprechend korrigieren müssen).

addr	nextA	nextB	$\mu PC_{mux.s1}$	$\mu PC_{mux.s0}$	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	0	0	0	1	1	

Vervollständigen Sie den Mikrocode des Prozessors und testen Sie den Halt-Befehl.

#### Aufgabe 2.11 ALU-Befehle

Erweitern Sie Ihr Mikroprogramm um die Schritte zum Ausführen aller ALU-Befehle (mit Opcode 0010). Dies ist genauso einfach wie die Realisierung des HALT-Befehls, da die Auswahl der eigentlichen ALU-Operation direkt in der ALU selbst vorgenommen wird. Das Mikroprogramm muss daher nur die Steuersignale für das Write-Enable der Registerbank, das Enable des Carry-Registers und für den Tristate-Puffer am Ausgang der ALU erzeugen. Machen Sie sich diese Zusammenhänge anhand des Befehlssatzes (Tabelle 1) und Blockschaltbild des D-CORE Prozessors (Abb. 1) aus Bogen 1 klar. Tragen Sie den notwendigen Mikroprogrammschritt hier ein:

addr	nextA	nextB	$\mu PC_{mux.s1}$	$\mu PC_{mux.s0}$	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	0	0	0	1	1	

Die ALU ist so entworfen, dass der Carry-Ausgang nur für Befehle gesetzt wird, die diesen Wert verändern. Für alle anderen Befehle reicht 0 die ALU den Eingangswert von CarryIn zum Carry-

Ausgang durch. Ihr Mikroprogramm für die ALU-Befehle sollte daher das C-Register aktivieren.

**Aufgabe 2.12** 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:

addr	nextA	nextB	IPCmux-s1	IPCmux-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	0	1	1	

Speichern Sie Ihr Mikroprogramm. 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 bearbeitet	abzeichnen lassen
--------------------	-------------------

**Aufgabe 2.13** Register-Initialisierung

Verwenden Sie ihre Notizen aus Bogen 1, Aufgabe 1.9, um ein Programm zu schreiben, das die ersten fünf Register initialisiert, danach ein 1900er Datum in die Register R11 (Tag), R12 (Monat) und R13 (Jahr) ablegt und schließlich mit HALT stoppt. Ihr erstes vollständiges Programm für den D-CORE! Bezüglich der Befehlskodierung sei auf die Tabelle 1 aus Bogen 1 hingewiesen. 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 bearbeitet	abzeichnen lassen
--------------------	-------------------

**Aufgabe 2.14**      Vergleichsoperationen

Schreiben Sie ein kleines Programm, um alle vier Vergleichsoperationen zu demonstrieren. Setzen Sie etwa  $R0 = 0$ ,  $R1 = 1$ ,  $R2 = -1$  und vergleichen Sie diese Werte so miteinander, dass das C-Register abwechselnd gesetzt und rückgesetzt wird.

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

**Aufgabe 2.15**      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 sogenannte *Pseudoinstruktionen* im Assembler für einen Rechner zusätzlich zur Verfügung gestellt, obwohl sie bereits vom Befehlssatz abgedeckt werden. Der Programmierer kann dann einfachere 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			

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