Praktikum Rechnerstrukturen

WiSe2021/22

2

Mikroprogrammierung
Speicheransteuerung

Name, Vorname	
Bogen bearbeitet	abzeichnen lassen

Universität Hamburg – MIN – Fachbereich Informatik
Arbeitsbereich Technische Aspekte Multimodaler Systeme

https://tams.informatik.uni-hamburg.de

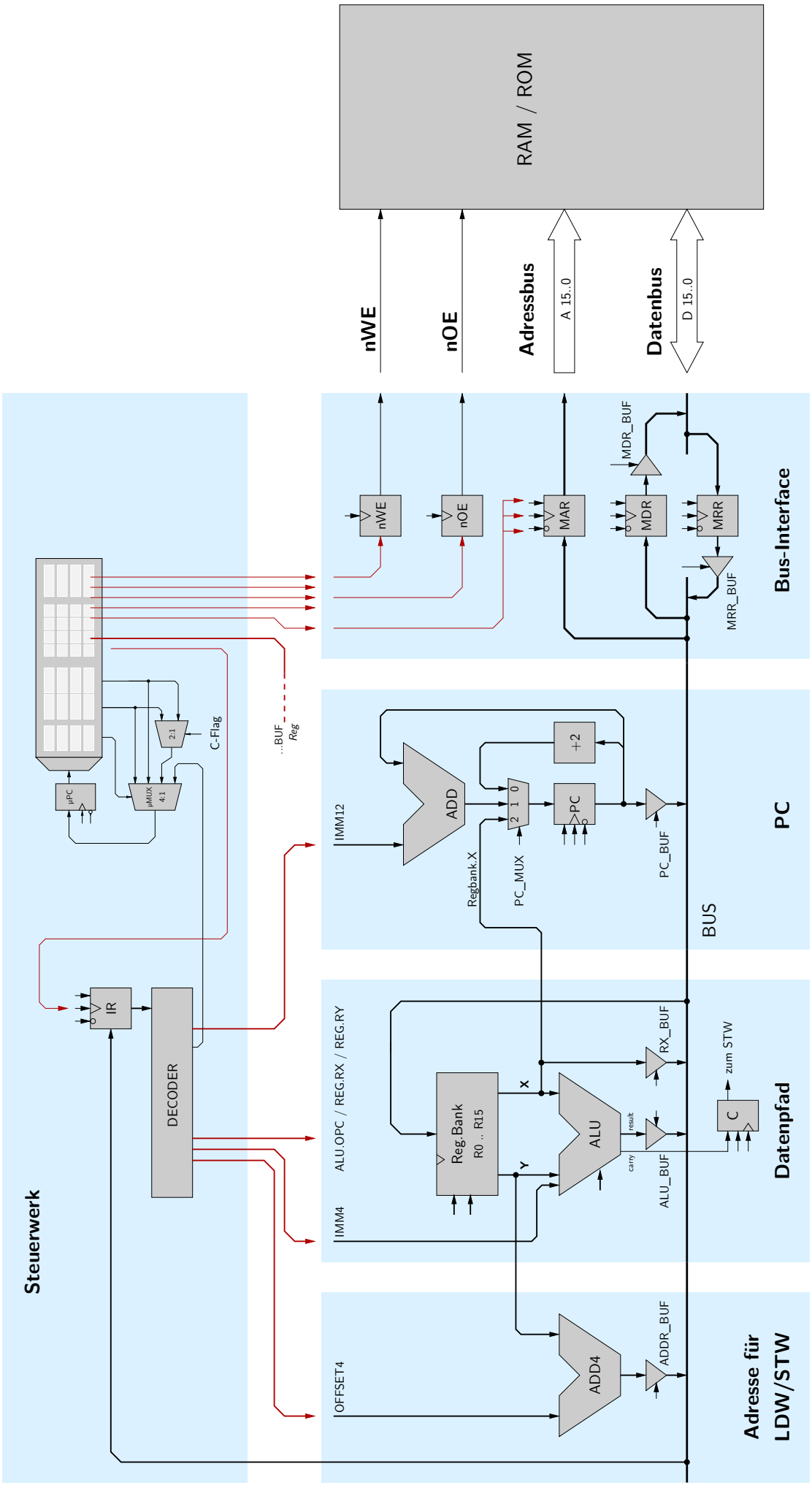


Abb. 1: Blockschaltbild des D·CORE Prozessors

5 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 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.

Abbildung 2 zeigt ein allgemeines Automatenmodell. Der aktuelle Zustand Q^t des Automaten zum Zeitpunkt t ist im Zustandsregister gespeichert. Das Ausgabeüberführungs- oder kurz λ-Schaltnetz leitet aus dem aktuellen Zustand Q^t und ggf. weiteren externen Eingaben (in der Abb. 2 grau dargestellt) die aktuellen Ausgaben des Automaten ab. Das Zustandsüberführungsschaltnetz oder kurz das δ-Schaltnetz berechnet aus dem aktuellen Zustand Q^t und normalerweise zusätzlichen externen Eingaben den Folgezustand Q^{t+1}, der durch das Clockereignis zum Zeitpunkt $t+1$ in das Zustandsregister übernommen und damit wieder zum aktuellen Zustand wird.

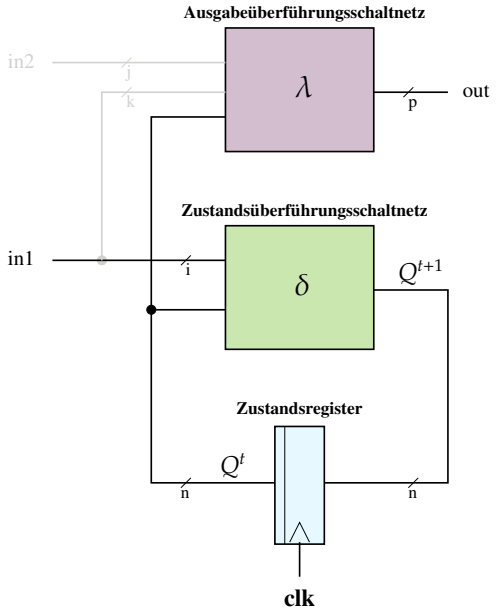


Abb. 2: allgemeines Automatenmodell
Die Kennzeichnung auf den Leitungsbündeln (diagonaler Strich mit darunter angeordnetem Buchstaben) gibt die Anzahl der Einzelleitungen des Leitungsbündels (hier auch gleichbedeutend mit Bitbreite des zu übertragenden Signals) an.

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 Berechung der Folgezustände (das δ-Schaltnetz) und das Schaltnetz zur Berechnung der Ausgaben (das λ-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 komfortabel realisiert werden können.

Das grundlegende Prinzip eines mikroprogrammierten Schaltwerks ist in Abbildung 3 zusammen mit dem zugehörigen endlichen Automaten skizziert (Kreise stellen die Zustände und Pfeile die Zustandsübergänge dar). Das entsprechende Schaltwerk (siehe Abb. 3 rechts) besteht aus drei Komponenten:

- einem n-bit Register, genannt Mikroprogrammzähler (μPC, micro program counter).

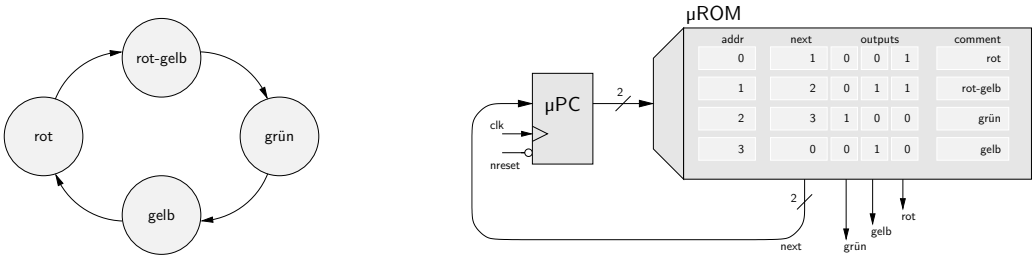


Abb. 3: Prinzip der Mikroprogrammierung: einfacher Automat (links) und Realisierung mit linearem Mikroprogramm (hier: $n = 2$, $m = 5$)

- einem ROM mit 2^n Adressen (also n Adressbits) 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, der das zugehörige Speicherwort im Mikroprogrammspeicher adressiert. Die Ausgangswerte des Mikroprogrammspeichers liefern dann neben den Signalen für die Berechnung des Folgezustandes (s. u.) auch 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 3 zeigt die einfachste davon: hier wird der μPC bei jedem Taktimpuls mit dem Wert von μ`ROM.next` geladen; dieser Automat kann also nur lineare Zustandsfolgen ohne Verzweigungen realisieren. Das letzte Feld μ`ROM.comment` dient nur der Veranschaulichung und wird in der Hardware natürlich nicht realisiert.

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

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

Machen Sie sich diese Zusammenhänge anhand Abb. 4 klar.

Aufgabe 2.1 Mikroprogrammierung

Geben Sie jeweils den Wert an, der als nächster in den μPC geladen wird. Nehmen Sie hierzu das Schaltwertk aus Abb. 4 zu Hilfe und machen Sie sich die Funktion anhand des Schaltbildes klar:

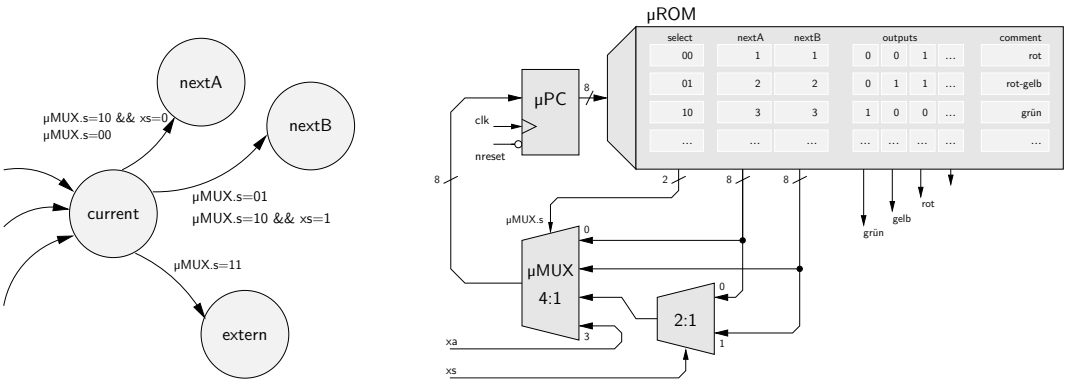


Abb. 4: Mikroprogrammiertes Steuerwerk mit zwei externen Eingängen `xs` und `xa` und vierfacher Auswahl des Folgezustands (siehe Text).

$(\mu\texttt{MUX.s})_2$	$(\texttt{nextA})_{16}$	$(\texttt{nextB})_{16}$	$(\texttt{xs})_2$	$(\texttt{xa})_{16}$	neuer $(\mu\text{PC})_{16}$
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	

Da in der Tabelle der Übersichtlichkeit halber die binäre und die hexadezimale Darstellung gemischt sind, ist hier die Basis explizit angegeben. Im weiteren Verlauf wird die Zahlendarstellung nicht mehr explizit kenntlich gemacht, wenn sich sich aus der Aufgabenstellung bzw. dem assoziierten Schaltbild eindeutig ergibt.
Wie Sie aus dem Schaltbild der Abb. 4 erkennen, wird die Multiplexersteuerung μ`MUX.s` durch das Feld μ`ROM.select` des μ`ROM` bestimmt.

Aufgabe bearbeitet	abzeichnen lassen

Aufgabe 2.2 Ampelschaltung

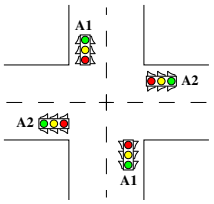
Laden Sie die Beispieldatei **`sequencer.hds`**. Diese enthält den Mikroprogrammspeicher μ`ROM` mit zugehörigem Mikroprogrammzähler μPC gemäß Abbildung 3 und einige LEDs für die Ausgabe des mikroprogrammierten Automaten. Selektieren Sie den Eintrag *Edit* aus dem Kontextmenü des Mikroprogrammspeichers, um den Mikroprogramm-Editor zu

öffnen. Dort können Sie jetzt die Speicherinhalte direkt modifizieren (Eingaben per Tastatur; 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 (Ampel1) bzw. D4, D5, D6 (Ampel2) sind bereits die richtigen Farben rot, gelb, grün zugewiesen.

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

Speichern Sie Ihr Mikroprogramm (z.B. als `ampel.mic`) und tragen Sie die Daten auch in die folgende Tabelle ein:

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

Hinweis: Nach dem Laden eines Mikroprogramms aus einer Datei, z.B. ampel.mic, müssen Sie die laufende Simulation anhalten (⏮-Button) und neu starten (⏭-Button), damit der Simulator das geänderte Mikroprogramm für die Simulation korrekt übernimmt.

Aufgabe bearbeitet	abzeichnen lassen

5.1 Automatisierte Berechnung

Jetzt haben wir auch eine Möglichkeit, die ALU-Operationen aus Bogen 1 Aufgabe 1.9 zu automatisieren. Laden Sie dazu in HADES die Datei **`datapathMR.hds`**.
`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 auch noch das Enable des Tristate-Treibers am Ausgang der ALU und das Enable des Carry-Registers mit in das Mikroprogramm aufgenommen werden können. Darauf wird aber verzichtet, weil sie für diese Aufgabe konstant 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.

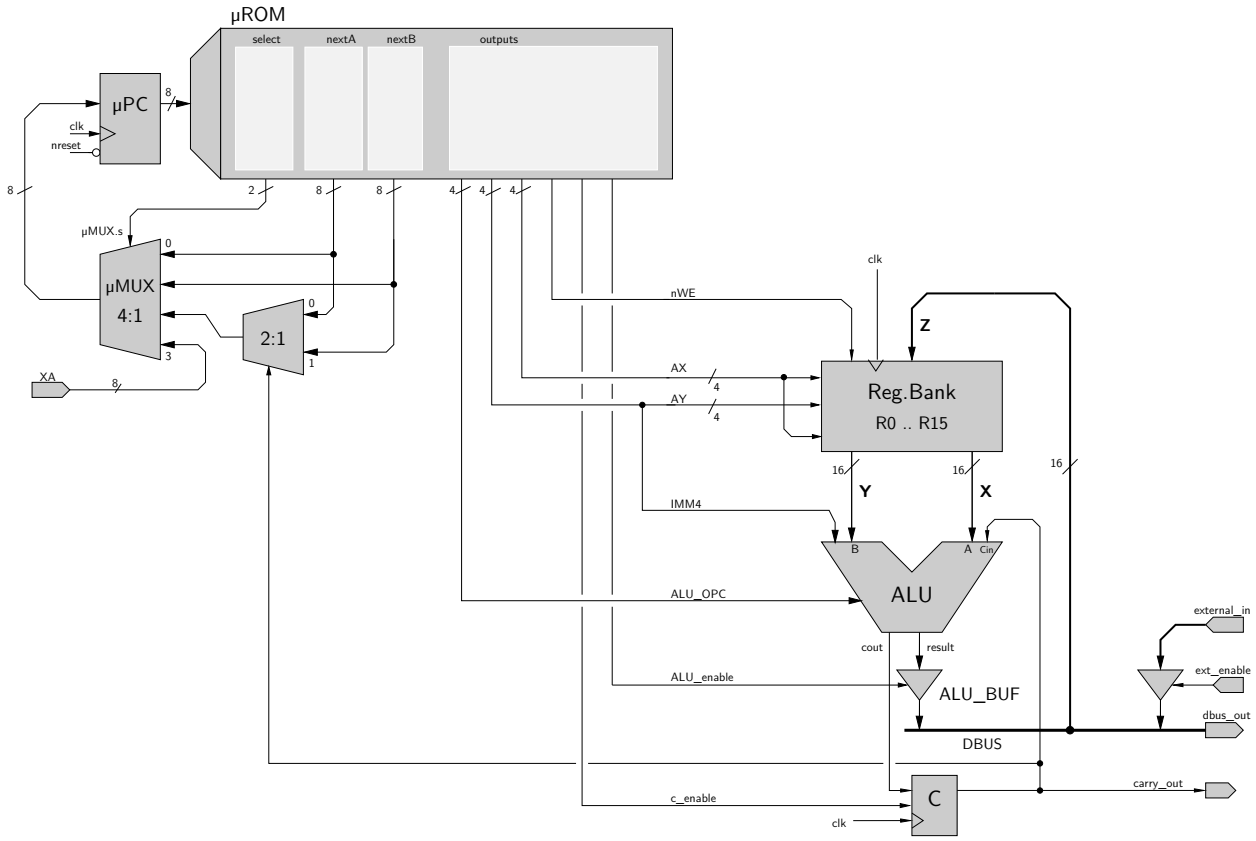


Abb. 5: Steuerwerk als Mustergenerator für Operationswerk

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 berechnet wird (Zeilen 00, 01), soll ermittelt und im Register R2 abgelegt werden. Das lässt sich zum Beispiel wie folgt (Zeilen 02–06) realisieren:

addr	nextA	nextB	μMUX.s (4:1)	ALUOP	SelY	SelX	REGs. nWE	Kommentar
00	01	00	00	14	0	0	0	R0=0 (Zahl für 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 (für 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 ($*$ – don't care) ist, weil nichts in die Registerbank geschrieben wird (REGs.nWE=1).

Aufgabe 2.3 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	μMUX.s (4:1)	ALUOP	SelY	SelX	REGs. nWE	Kommentar
00								
01								
02								
03								
04								
05								
06								
07								
08								
09								
0a								
0b								

Aufgabe bearbeitet	abzeichnen lassen

6 Speicheransteuerung

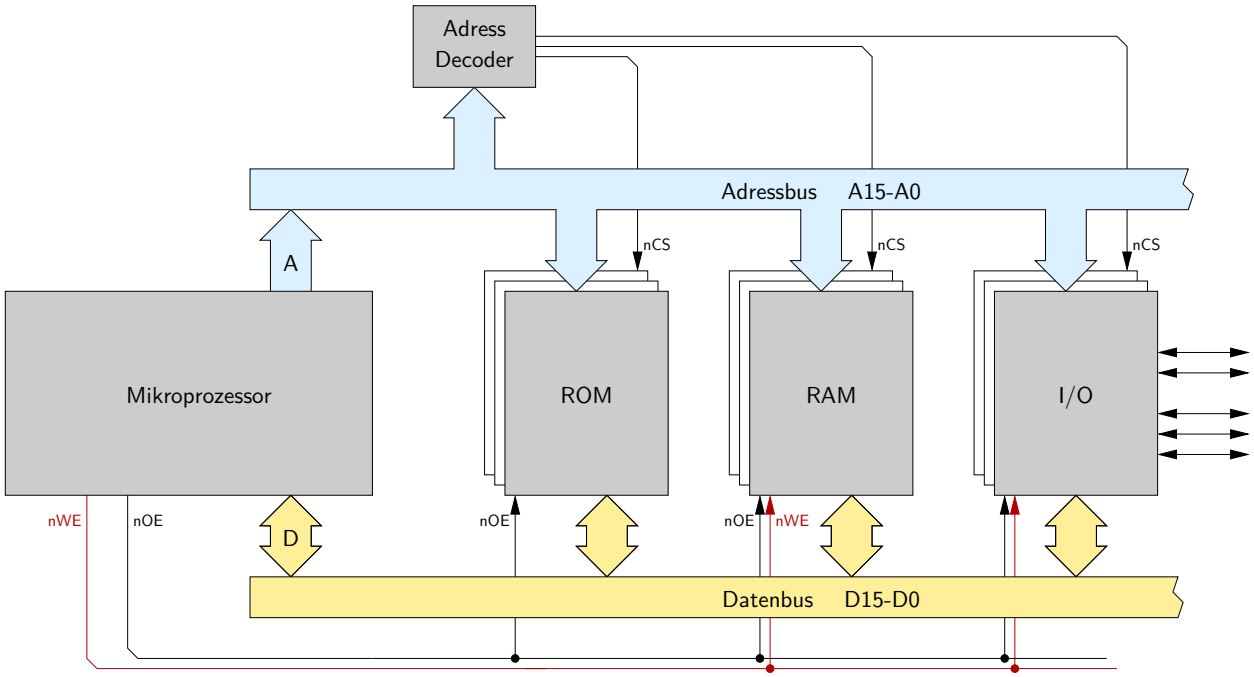


Abb. 6: Gesamtsystem mit Prozessor, RAM, ROM und I/O Komponenten. Der Bussystem zur Ansteuerung des Speichers besteht aus Adress- und Datenbus sowie den Steuerleitungen nWE und nOE.

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 6 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 mit 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

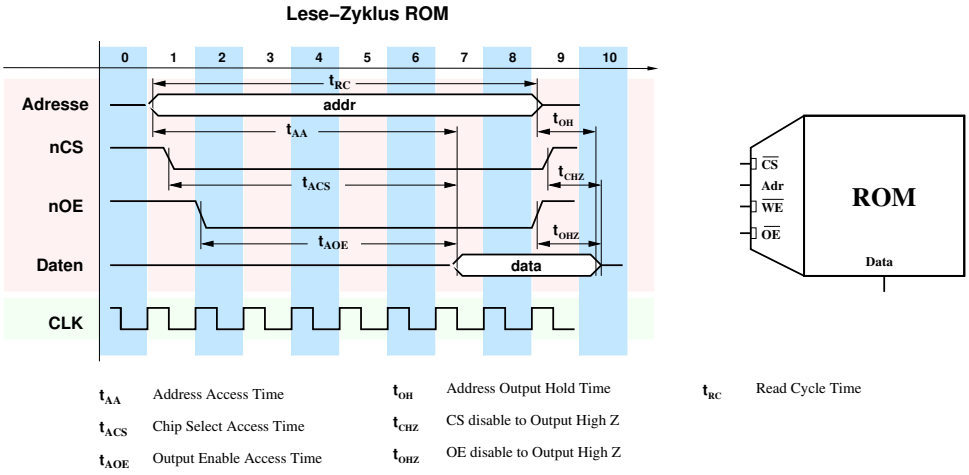


Abb. 7: Vereinfachtes Impulsdiagramm eines Lesezugriffs auf ein ROM (mit OE). Die Zugriffszeit (t_{access}) kann grob betrachtet als das Maximum aus t_{AA}, t_{ACS} und t_{AOE} angesehen werden. Die Zeitbedingungen müssen in einem getaktetem System auf ein Vielfaches der Clock-Periode umgesetzt werden. Da das *Chip-Select-Signal* (nCS) der Speicherbausteine in der vorliegenden Implementierung (siehe Abb. 8) unmittelbar aus der Adresse abgeleitet wird, folgt es mit geringer Zeitverzögerung der Adresse und erfüllt somit bei korrekten Adresssignalen auch alle Zeitbedingungen. Daher wird das *Chip-Select*-Signal bei den späteren Betrachtungen (vgl. Abb. 9) nicht weiter berücksichtigt.

I/O-Bausteinen. Statt dessen ist eine kompliziertere Ansteuerung nötig, die in Abbildung 7 (oberer pink hinterlegter Bereich des Diagramms) 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 einem 1 GHz Takt bereits 200 Wartezyklen.

Starten Sie **HADES** und laden Sie die Datei **`memoryMR.hds`**, um sich mit der Speicheransteuerung vertraut zu machen. Im Hauptfenster des Simulators sollten Sie das in Abb. 8 dargestelle Hades-Modell angezeigt bekommen.

Das Design enthält je eine RAM- und ROM-Komponente und das Mikroprogramm-ROM des Steuerwerks zur Ansteuerung der Adress-, Daten- und Steuerleitungen. Der einfach gehaltene *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` über deren Chip-Select-Signal (nCS).

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

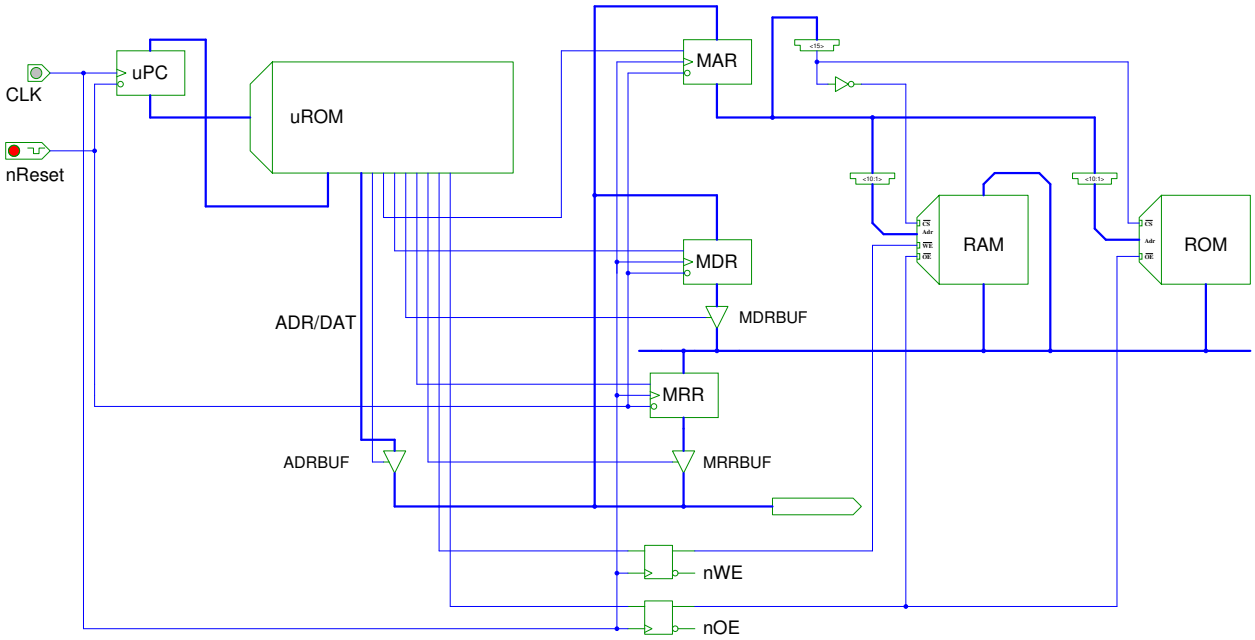


Abb. 8: Hades-Modell

dadurch realisiert, dass das unterste Bit des Adressbusses, also das Bit, das die Byteposition im Speicherwort bestimmt, nicht an die Speicherbausteine angeschlossen wird.

Typisch ist auch, dass die Speicher mehrfach in den Adressraum eingeblendet werden. Obwohl das ROM in **`memoryMR.hds`** nur 1K Worte aufweist (weil für die Aufgaben des Praktikums nie mehr Speicher benötigt wird und vor allem, um die Smulation nicht durch ein unnötig großes Speichermodell zu verlangsamen), 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 oder mehrere Takte langsamer. Um zum Beispiel einen Wert in den Speicher zu schreiben, muss zunächst eine Adresse in das Register `MAR` und ein Wert in das Register `MDR` übertragen werden. Erst danach kann der eigentliche Speicherzugriff stattfinden.

Öffnen Sie den Editor für das RAM und das ROM (Rechtsklick auf den entsprechenden Speicher-

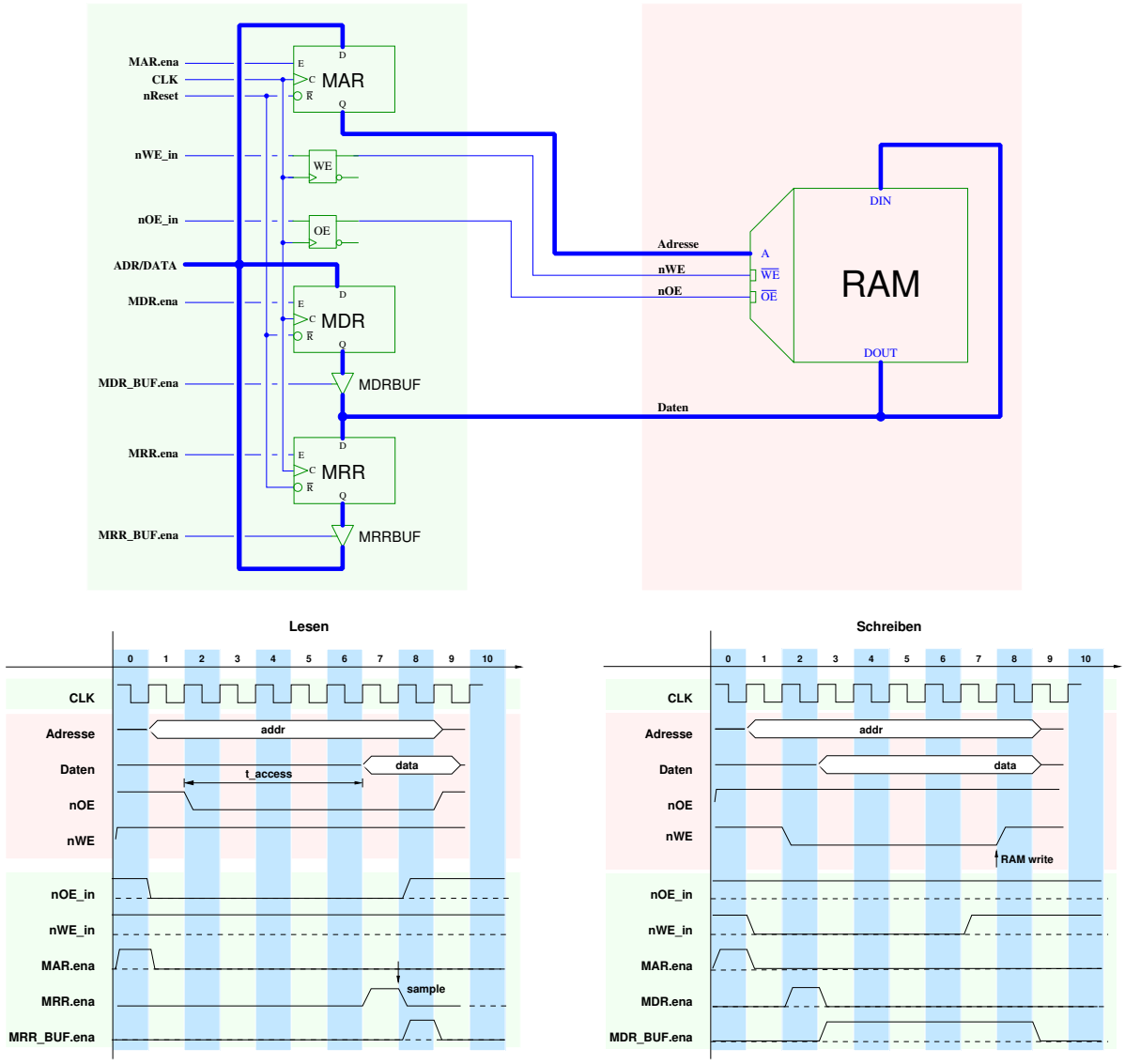


Abb. 9: Zeitabläufe beim Speicherzugriff (Lesen und Schreiben). Der Lesezugriff wird über die `nOE` Leitung gesteuert, der Schreibzugriff über `nWE`. Nach Aktivieren des Steuersignals sind Wartezyklen notwendig, um die Zugriffszeit des Speichers einzuhalten. `Adresse`, `Daten`, `nOE` und `nWE` sind Signale, die direkt am Speicherbaustein anliegen (pinker Bereich), während `nOE_in`, `nWE_in`, `MAR.ena`, `MDR.ena`, `MRR.ena`, `MDR_Buf.ena` und `MRR_Buf.ena` Signale sind, die an den entsprechenden Bauteilen (Register, Flip-Flops, Buffer, siehe Bogen 1 und vgl. auch Abb. 8) des Speicherinterfaces (grüner Bereich) anliegen.

baustein > edit), 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 2.4 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.

Hinweis: Aus Gründen der Übersichtlichkeit wurden die in Abb. 9 eingeführten Signalnamen für die Verwendung in Hades entsprechen eingekürzt, beispielsweise **`nWE_in`** zu **`nWE`**, Enable-Signale wie **`MAR.ena`** zu lediglich **`MAR`** oder **`MDR_BUF.ena`** zu **`MDRBUF`**.

addr	nextA	nextB	µPC-mux	ADR/DAT	ADR-BUF	MAR	MDR	MDR-BUF	MRR	MRR-BUF	nWE	nOE
00												
01												
02												
03												
04												
05												
06												
07												
08												
09												
0a												
0b												

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

Aufgabe bearbeitet	abzeichnen lassen

7 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.

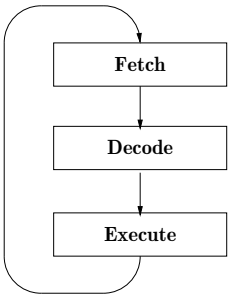


Abb. 10: Befehlszyklus des D·CORE Prozessors

7.1 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 μ`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 2.5 Mikroprogramm für Befehl Holen

In der ersten Phase des Befehlszyklus des von-Neumann Rechners (siehe Abb. 10) 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 11 illustriert.

Öffnen Sie den Editor für den Mikroprogrammspeicher. 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 und für den Programmzähler und ganz rechts liegen die Steuersignale für die Speicheransteuerung (`MAR`, `MDR`, `MRR`, `nOE`, `nWE`).

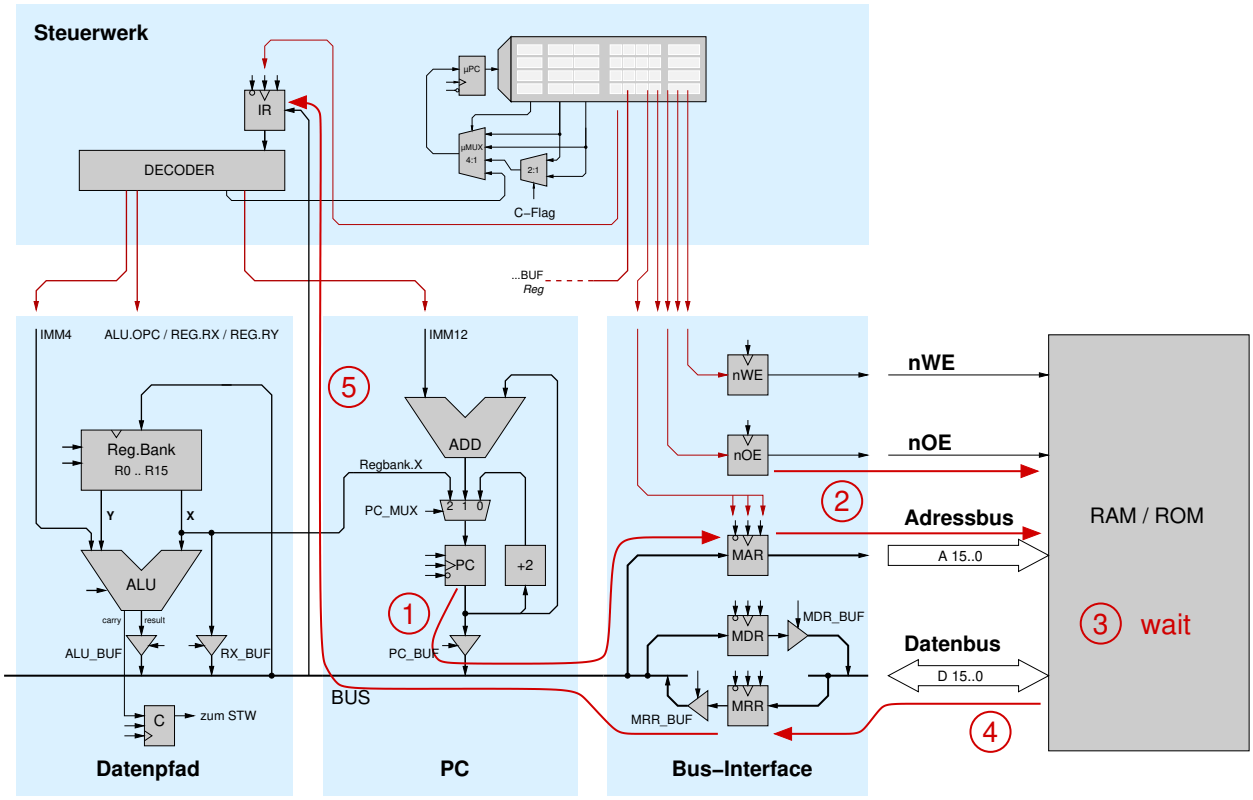


Abb. 11: Speicherinterface und Komponenten für die Befehl-Holen Phase

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

Wie schon in Abbildung 11 angedeutet, sind folgende Schritte nötig:

1. Der Wert des `PC`s muss in das `MAR` gebracht werden. Nach Übernahme in das Register liegt die Adresse am Speicher an.
2. Dem Speicher muss mitgeteilt werden, dass lesend auf ihn zugegriffen wird (Wert des Flip-flops für `nOE` entsprechend setzen).
3. Zugriffszeit des Speichers abwarten (für das Speichermodell werden (mindestens) drei Wait-states angesetzt).
4. Der Wert, der nach den Waitstates aus dem Speicher kommt, muss in das MRR geschrieben werden.
5. Der Wert muss aus dem MRR in das IR gebracht werden.

Überlegen Sie sich für jeden dieser Schritte, welche Steuersignale aktiviert werden müssen und tragen Sie es in Ihr Mikroprogramm ein.

Speichern Sie das Mikroprogramm, z.B. als Datei `fetch.mic`.
Hinweis: Nach dem Laden eines Mikroprogramms aus einer Datei, z.B. fetch.mic, müssen Sie 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. Die erste und letzte Zeile der Tabelle zeigt die Belegung der Signale bei Inaktivität.

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		annotation
–	00	00	0	0		0	0		0	0	0	0		1	0	0	0	0	0	0	0		0	0	1	1		inactive
00																												fetch_1
01																												fetch_2
02																												fetch_3
03																												fetch_4
04																												fetch_5
05																												
06																												
07																												
08																												
09																												
0a																												
0b																												
0c																												
0d																												decode
0e																												
0f																												
–	00	00	0	0		0	0		0	0	0	0		1	0	0	0	0	0	0	0		0	0	1	1		inactive

Aufgabe bearbeitet	abzeichnen lassen

7.2 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 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, das dann in der anschließenden Ausführen-Phase den Befehl wirklich

ausführt. Die Zerlegung des Befehls in die einzelnen Teile wie Opcode, ALU-Opcode, die Register-Adressen oder Offsets für die Sprungbefehle erfolgt in der als Decoder bezeichneten Hardware-Komponente, die Sie sich auch einzeln als Design **`decoder.hds`** anschauen können.

Ein Sprung im Mikroprogramm wird einfach dadurch realisiert, dass der Mikroprogrammzähler μPC auf den entsprechenden Wert gesetzt wird. Im Steuerwerk aus Abbildung 4 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 μMUX so angesteuert werden, dass der externe Eingang `xa` in den μPC 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` ⟨`****`⟩ ⟨`****`⟩ ⟨`xxxx`⟩) = $(\texttt{C**x})_{16}$ an der Mikroprogrammadresse (`1100 0000`) = $(\texttt{C0})_{16}$ das Mikroprogramm für den `HALT`-Befehl mit Opcode (`1111` ⟨`****`⟩ ⟨`****`⟩ ⟨`****`⟩) = $(\texttt{F***})_{16}$ an der Adresse (`1111 0000`) = $(\texttt{F0})_{16}$ usw.

Aufgabe 2.6 Decode-Phase und zugehöriger Microcode

Welche Werte werden jeweils in der Decode-Phase in den μPC geschrieben, wenn folgende Werte (Befehle) im `IR` stehen?

IR	μPC
0x7000	
0xF000	
0x2000	
0x2123	
0x2104	
0x2e00	

Die hier gewählte Lösung, bzw. Transformation des Opcodes in die Einsprungadresse, 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.

Erweitern Sie Ihren Microcode aus Aufgabe 2.5 um die Befehlsdecodierung (μ`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 16 und speichern Sie das Mikroprogramm (z.B. als `fetch-decode.mic`).

Aufgabe bearbeitet	abzeichnen lassen

7.3 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 Mikrorprogramme 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.

7.3.1 Der 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“.

Aufgabe 2.7 Implementierung des HALT-Befehls

Realisieren Sie den Befehl zum Beispiel durch eine Endlosschleife im Mikroprogramm: μ`PC.nextA` = μPC.
Tragen Sie den entsprechenden Mikroprogrammschritt in der untenstehenden exemplarischen Mikroprogrammzeile ein. Die zweite, ausgegraute Zeile zeigt zur Hilfestellung wieder die Werte der Signale bei Inaktivität.

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	annotation
–	00	00	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	inactive

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

7.3.2 Die ALU-Befehle

Die hier als ALU-Befehle bezeichneten Maschinenbefehle des D·CORE bezeichnen die in Tabelle 1 auf Bogen 1 unter „ALU- und Shift-Operationen und Vergleichs- und Immediate-Operationen“ aufgeführten Befehle, die die in der rechten Tabelle der Abb. 5 auf Bogen 1 dokumentierten Alu-Funktionalitäten in Maschinenbefehle umsetzen.

Aufgabe 2.8 Implementierung der ALU- und Shift-Befehle und der Immediate- und Vergleichsbefehle

ALU- und Shift-Befehle
Erweitern Sie Ihr Mikroprogramm um die Schritte zum Ausführen aller ALU- und Shift-Befehle (also die Befehle mit Opcode `0010`). Dies ist ähnlich 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, Bogen 1) **und** des Blockschaltbilds des D·CORE Prozessors (Abb. 1) klar. Tragen Sie den notwendigen 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	annotation
–	00	00	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	inactive

Die ALU ist so entworfen, dass der Carry-Ausgang nur für Befehle gesetzt wird, die diesen Wert verändern. Für alle anderen Befehle reicht die ALU den Wert von Carry-Eingang (`Cin`) zum Carry-Ausgang (`Cout`) durch (siehe Aufgabe 1, Kap.4.5). Ihr Mikroprogramm für die ALU-Befehle sollte daher das C-Register aktivieren.

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 um Befehle, die die ALU-Operationen mit Immediate-Operanden realisieren. Es sind hier also die Befehle mit dem Opcode `0011` (siehe Tabelle 1, Bogen 1). Wie bei den ALU-Rechenbefehlen reicht wiederum ein einziger Mikroprogrammschritt aus, da die Auswahl der eigentlichen ALU-Funktion intern in der ALU erfolgt. Machen Sie sich auch diesen Sachverhalt anhand des Befehlssatzes (Tabelle 1, Bogen 1) und des Blockschaltbilds des D·CORE Prozessors (Abb. 1) für die Immediate- und Vergleichsbefehle klar und implementieren Sie diese:

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	annotation
–	00	00	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	inactive

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 bearbeitet	abzeichnen lassen

Aufgabe 2.9 Erste Programme für den D·CORE

Register-Initialisierung

Verwenden Sie ihre Notizen aus Bogen 1, Aufgabe 1.9, um ein Programm zu schreiben, das die ersten fünf Register mit einem beliebigen Wert initialisiert, danach ein 1 900er 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 Befehlscodierung sei auf die Tabelle 1 aus Bogen 1 hingewiesen. Bitte dokumentieren Sie die einzelnen Maschinenbefehle Ihres Programms:

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`			

Vergleichsoperationen

Schreiben Sie ein Programm, um alle vier Vergleichsoperationen zu demonstrieren. Setzen Sie `R0 = 0, R1 = 1, R2 = -1` und vergleichen Sie diese Werte derart miteinander, dass das `C`-Register abwechselnd gesetzt und rückgesetzt wird.
Bitte dokumentieren Sie die Maschinenbefehle Ihres Programms in der folgenden Tabelle:

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 bearbeitet	abzeichnen lassen

Aufgabe 2.10 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 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			

Aufgabe bearbeitet	abzeichnen lassen