Praktikum Rechnerstrukturen

Bogen 1

Mikroprozessorsysteme

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

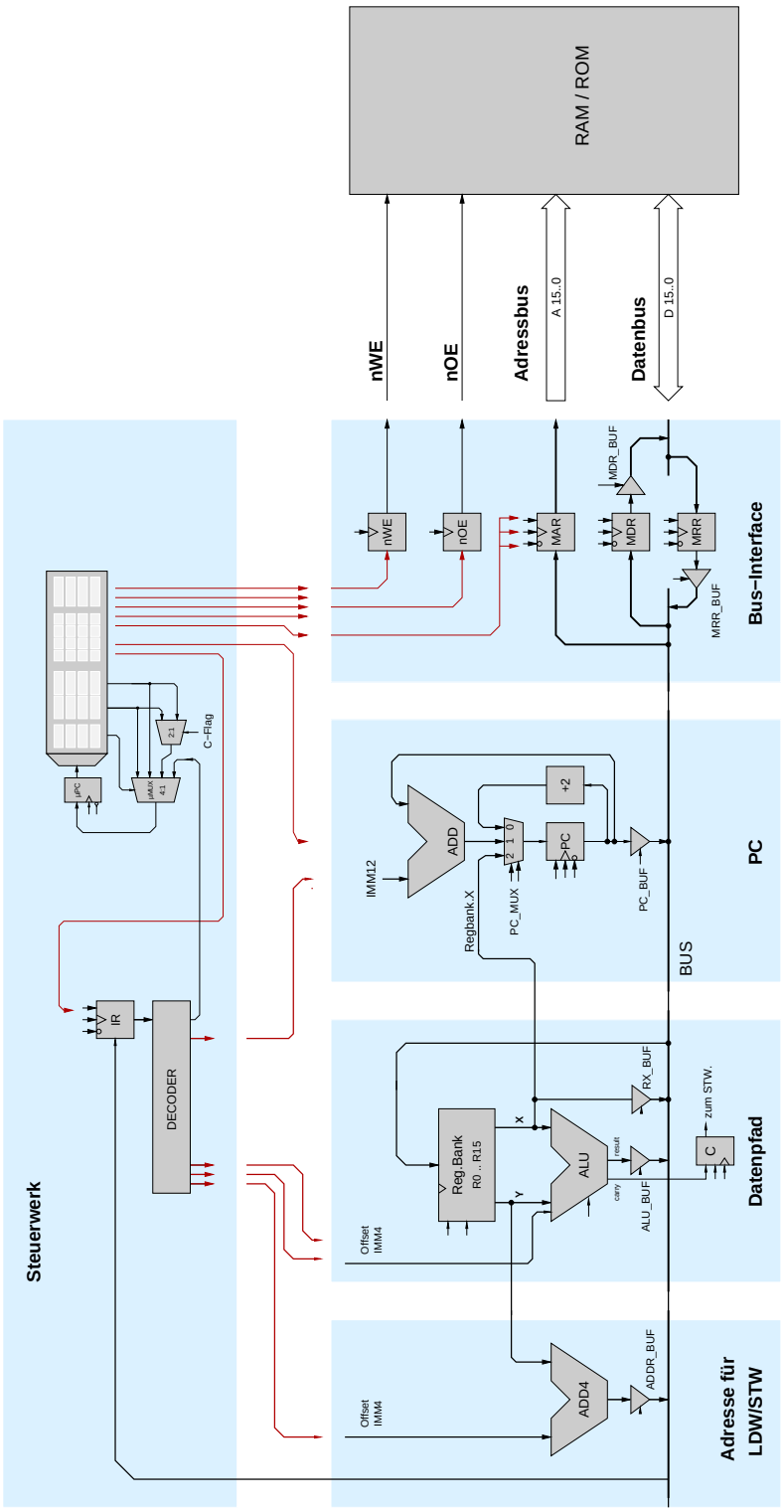


Abbildung 1: Blockschaltbild des D·CORE Prozessors

1 Einführung und Motivation

Ziel des Praktikums *Rechnerstrukturen* 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: *Assemblerebene*, *Ebene der Maschinensprache, Register-Transfer-Ebene*. Die Ebenen oberhalb der Assemblerebene sind dem P-Zyklus vorbehalten, die unteren Ebenen von Schaltwerkebene bis hinunter zur Physik kennen Sie bereits aus der Vorlesung.

Um die Hierarchie der einzelnen Abstraktionsebenen deutlich zu machen, werden alle Aufgaben in einem *bottom-up* Vorgehen aufeinander aufbauen. Am ersten Praktikumstag werden dazu die Komponenten eines einfachen Mikroprozessors erklärt (Register-Transfer-Ebene) und daraus dann am zweiten Tag auf dem Prozessor schrittweise die Befehle, die er verarbeiten können soll, implementiert. (Befehlsarchitektur). Damit können dann erste Assemblerprogramme geschrieben werden, deren Komplexität sich langsam erhöht.

Wegen der gegenüber einem echten Hardwareaufbau besseren Debug-Möglichkeiten werden die folgenden Versuche zunächst mit dem HADES-Simulator durchgeführt, der von **Norman Hendrich** in seiner Zeit am Fachbereich Informatik entworfen und in JAVA implementiert worden ist. HADESist public-domain-Software und steht bei Interesse unter

tams-www.informatik.uni-hamburg.de/applets/hades/webdemos/index.html

zusammen mit der dazugehörigen Dokumentation zum Download bereit.

Ein wirklich tiefes Verständnis des Simulators ist aber nicht unbedingt erforderlich, weil die Teilschaltungen fast alle bereits fertig aufgebaut sind und lediglich noch vervollständigt werden müssen. Die späteren Aufgaben zur Assemblerprogrammierung werden dann mit einem üblichen Assembler/Debugger durchgeführt.

Aufgabe 1.1: 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/lectures/2006ws/praktikum/rechprak/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.

Eine Quick-Reference Karte (hades-quick-reference.pdf) mit der Kurzbedienungsanleitung finden Sie ebenfalls auf dem Webserver.

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.

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 M·CORE-Architektur von Motorola, die für Anwendungen in *embedded systems* mit hoher Performance bei minimalem Stromverbrauch entwickelt wurde (etwa Mobiltelefone). Diese wurde 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

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 eine längere Beschreibung jedes einzelnen Befehls.

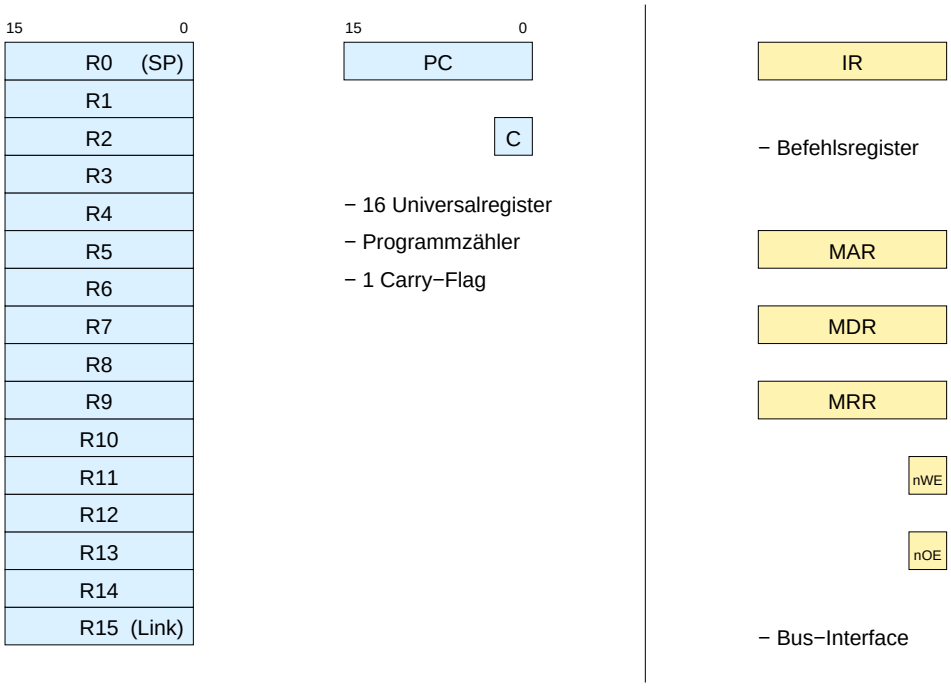


Abbildung 2: Die Architektur (Programmiermodell) des D·CORE Prozessors: Programmzähler `PC`, 16 Universalregister und 1 Carry-Flag (links). Das Befehlsregister `IR` und die zusätzlichen Register des Businterface (`MAR`, `MDR`, `MRR`, `nOE`, `nWE`) sind dagegen nicht für Programme sichtbar (rechts).

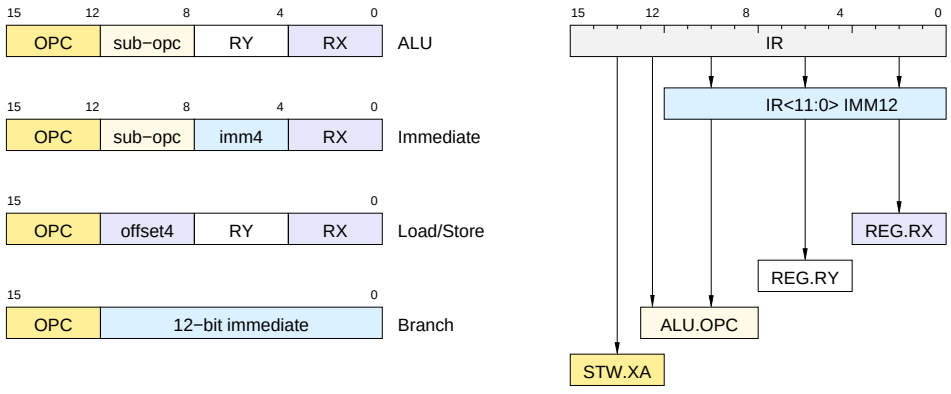


Abbildung 3: Befehlsformate (links) und Dekodierung der einzelnen Felder (rechts)

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

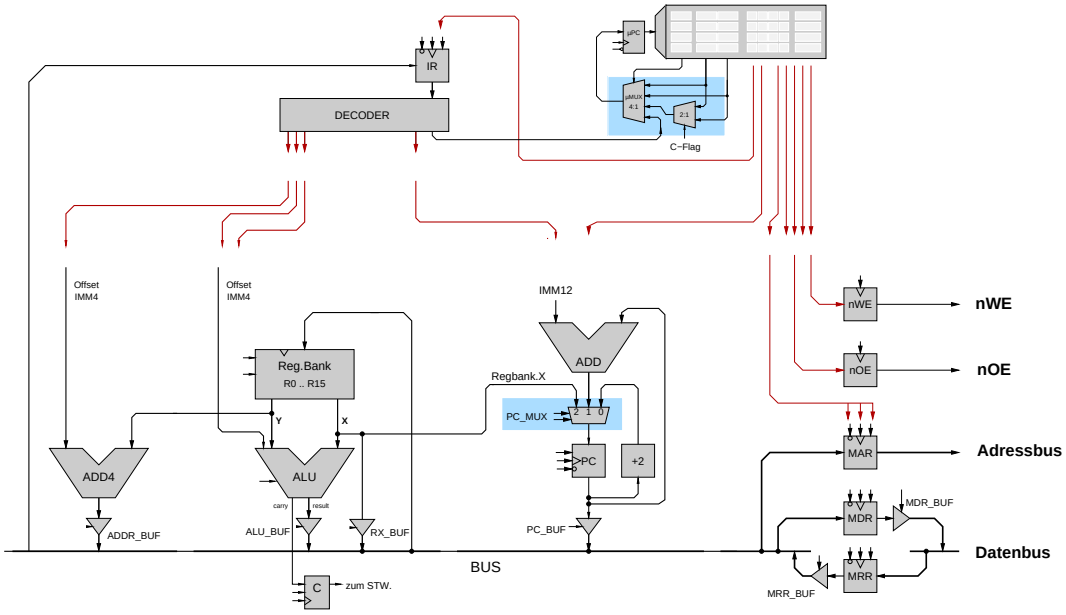
Tabelle 1: Befehlssatz des D·CORE

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 Komponenten des D·CORE-Prozessors

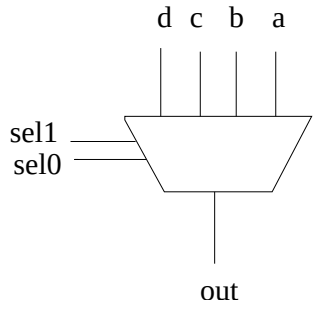
Wir wollen jetzt nacheinander die Komponenten aus dem Schaltbild des D·CORE-Prozessors von Abbildung 1 betrachten.

3.1 Multiplexer



Multiplexer finden sich an den farbig unterlegten Stellen im Schaltbild. Sie dienen dazu, um in Abhängigkeit von Steuersignalen einen ihrer Eingänge auf den Ausgang durchzuschalten.

Ein 4-zu-1-Multiplexer wie in folgenden Bild



hat also folgende Funktion

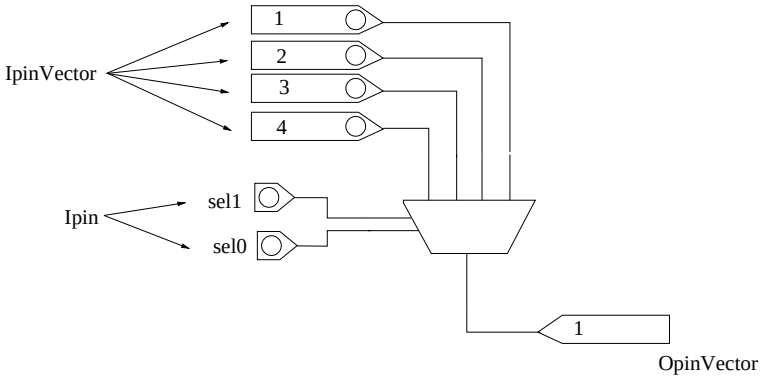
```
case  sel
  ‘‘00’’:  out:= a;
  ‘‘01’’:  out:= b;
  ‘‘10’’:  out:= c;
  ‘‘11’’:  out:= d;
end case;
```

Frage Wie viele Bit brauchen Sie für das Steuersignal mindestens, wenn Sie eine Auswahl unter a) 8, b) 32 und c) 7 Eingängen treffen wollen?

Wie viele Eingänge können Sie höchstens multiplexen, wenn Ihr Steuersignal 8 Bit breit ist?

Aufgabe 3.2: Multiplexer

Starten Sie den Hades-Simulator und laden Sie das Design `multiplexer.hds` (Menue **File** und **Open**). Sie sollten ungefähr folgendes Bild sehen:



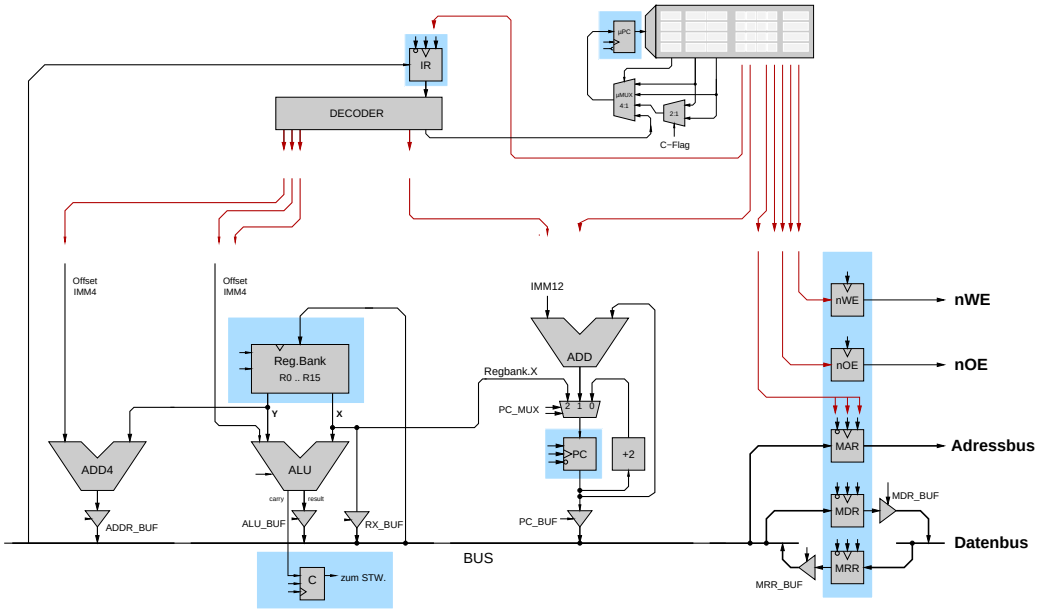
Ein sog. *Ipin* dient dabei als Eingang für ein Signal, das ein Bit breit ist. Die beiden *Ipins* sollten dabei zu Anfang die Farbe Blau haben als Zeichen, dass ihnen noch kein Wert zugewiesen worden ist (weder eine logische **0** noch eine logische **1**). Ändern lässt sich dies, indem man auf den *Ipin* klickt. Eine graue Farbe entspricht einer logischen **0**, rot einer **1**.

Über den *Ipins* befinden sich vier *IpinVectors* zur Eingabe von Signalen, die mehr als ein Bit breit sind (im konkreten Fall 16 Bit). Wenn man mit der linken Maustaste auf einen *Ipinvector* klickt, erhöht sich sein Wert um Eins, wenn man gleichzeitig die Shift-Taste gedrückt hält, erniedrigt er sich um Eins. Eine andere Möglichkeit, den Wert zu ändern, besteht darin, mit der rechten Maustaste auf den *IpinVector* zu klicken. Es öffnet sich dann ein Menue, aus dem man den Eintrag **Edit** auswählen kann, über den sich einige Parameter einstellen lassen, inbesondere auch der Ausgabewert (Output-Value).

Als Ausgabe dient ein Element des Typs *OpinVector*.

Spielen Sie ein wenig mit der Schaltung herum und machen Sie sich dabei noch einmal die Funktion eines Multiplexers klar.

3.2 Flipflops und Register



Flipflops und Register finden sich an den farbig unterlegten Stellen im Schaltbild. Wie aus der Vorlesung bekannt dient ein *Flipflop* dazu, einen ein Bit breiten logischen Wert (**0** oder **1**) zu speichern.

Aufgabe 3.3: Flipflops und Register in Hades

Laden Sie das Design `Register.hds`. Ganz oben findet sich ein D-Flipflop (von Data-Flipflop), mit dem Sie diese Funktion noch einmal nachvollziehen können. Es gibt zwei Eingänge *Data* und *Clock* (Takt) und zwei Ausgänge *Q* und *nQ*, die immer invers zueinander sind. Bei einem **0** → **1** Übergang auf dem Takteingang (Vorderflanke) wird der Wert von *Data* gespeichert und auf dem Ausgang *Q* ausgegeben.

Flipflops in dieser einfachen Form sind im Schaltbild die zum Speichern der Signale *nWE* und *nOE*.

Darunter befindet sich ein komplizierteres D-Flipflop, wie es im Schaltbild beim Flipflop *C* und in den Registern *µPC*, *PC* und *IR* verwendet wird. Wie man sieht gibt es noch zwei weitere Eingänge *Enable* und *nReset*.
Wenn am Eingang *nReset* eine **0** anliegt, liefert der Ausgang *Q* unabhängig von den anderen Eingängen eine **0**. Damit kann das Flipflop in einen definierten Anfangszustand gebracht werden.

Der Eingang *Enable* dient dazu, das Einspeichern eines neuen Werts bei einer Vorderflanke auf dem Takteingang zu unterbinden. Nur bei einer **1** am Enable-Eingang wird der Wert, der am D-Eingangs liegt, mit der nächsten Vorderflanke des Taktes wirklich übernommen.

a) Vollziehen Sie dieses Verhalten bitte nach, indem Sie etwas mit dem Hades-Modell experimentieren.

Ein *Register* ist eine Anzahl von Flipflops, die alle den gleichen Takt, Enable und Reset, aber unterschiedliche Daten-Eingänge haben.

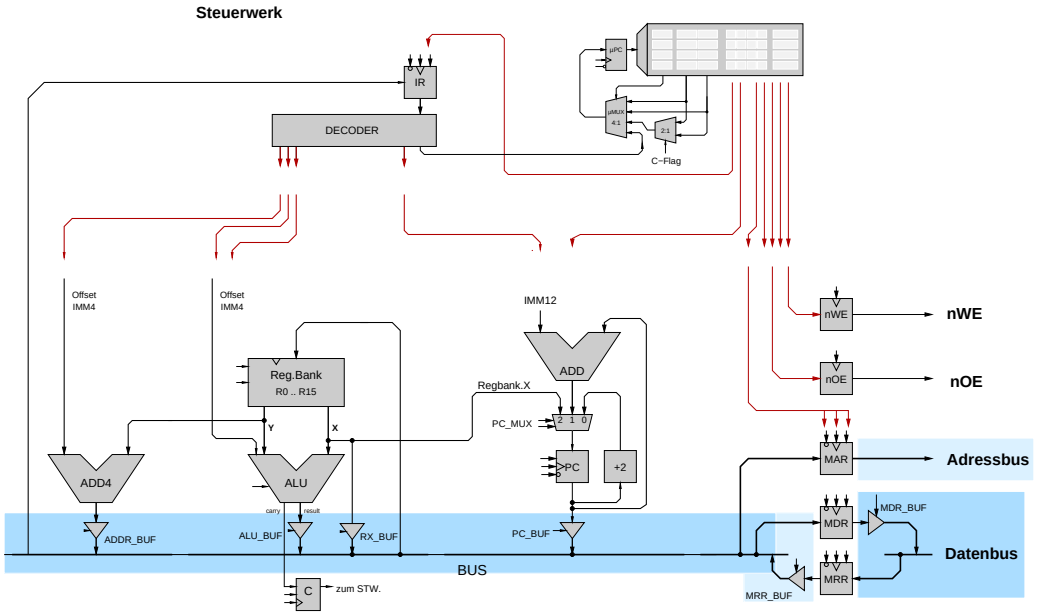
Im Beispiel-Design befindet sich ein Register, das einen 16-Bit breiten Wert aufnehmen kann, weshalb am Dateneingang ein IpinVector und am Ausgang ein OpinVector angeschlossen ist.

b) Vollziehen Sie auch hier das Verhalten bitte nach, indem Sie etwas mit dem Hades-Modell experimentieren.

Unter dem Register befindet sich noch eine *Registerbank*, wie sie auch in unserem Prozessordesign verwendet wird, d.h. eine Ansammlung von (hier 16) Registern, die alle denselben Takt *(Clock)* haben.
Man beachte, dass es kein *Reset*-Signal gibt, mit dem sich die Register auf einen definierten Anfangswert setzen lassen.
Weiterhin ist hier das *Enable*-Signal low-aktiv, d.h. für eine **0** wird bei der nächsten Vorderflanke auf der Takt-Leitung etwas in die Registerbank geschrieben und zwar in das Register, das mit dem Eingang *SelX* ausgewählt wurde.
Die Registerbank hat zwei Ausgänge, auf die sich über die Eingänge *SelX* und *SelY* Werte aus der Registerbank legen lassen.

c) Vollziehen Sie bitte auch dieses Verhalten nach, indem Sie mit dem Hades-Modell experimentieren.Den Inhalt der Registerbank können Sie sich ansehen, mit der rechten Maustaste darauf klicken und **Edit** wählen.

3.3 Busse und Tristate-Treiber



Busse und sog. Tristate-Treiber finden sich an den farbig unterlegten Stellen im Schaltbild.

Ein Bus ist dabei einfach ein Leitungsbündel, in dem innerhalb der Schaltung an verschiedenen Stellen sowohl lesend als auch schreibend auf dasselbe Signal zugegriffen wird.

Wie man sieht, dient z.B. der Bus mit Namen **BUS** als Eingang für die Register IR, MAR, MDR und die Registerbank, wobei dann natürlich über die entsprechenden Enable-Signale gesagt werden muss, ob der Wert wirklich übernommen werden soll.

Dieser lesende Zugriff auf den Bus ist elektrisch im Normalfall völlig unkritisch. Anders sieht es aus, wenn man von verschiedenen Stellen auf dieselbe Leitung schreiben möchte. Weil jeder Ausgang entweder eine **0** oder eine **1** liefert, kann es zu Kurzschlüssen zwischen einer niedrigen Spannung und einer hohen Spannung kommen, die im schlimmsten Fall die Schaltung zerstören, auf jeden Fall aber meist zu falschen Ergebnissen führen. Man muss also irgendwie eine Möglichkeit finden, um sicherzustellen, dass immer nur genau ein Ausgang auf dieselbe Leitung schreibt.

Eine Möglichkeit wäre es, alle Ausgänge erst einmal auf einen Multiplexer zu führen und dessen Ausgang dann in Abhängigkeit von Steuersignalen auf den Bus zu schalten. Aus einer Reihe von Gründen ist diese Lösung aber nicht besonders attraktiv, weil man erst einmal alle Ausgänge zu diesem zentralen Multiplexer führen muss. Das Einstecken einer Karte in einen PC wäre dann also damit verbunden, dass man erst einmal ein mehr oder weniger breites Kabel stecken muss, dass die Verbindung zwischen Karte und Multiplexer herstellt, wobei man auch noch genau darauf achten muss, dass man den richtigen Eingang wählt.

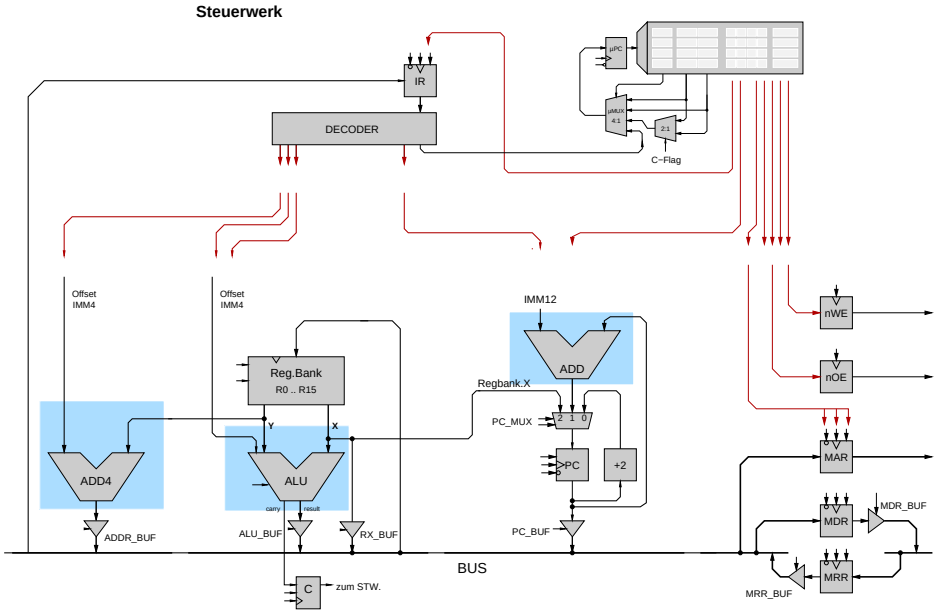
Eine bessere Lösung des Problems stellen sog. **Tristate-Treiber** dar, die einen direkten Anschluss des Ausgangs an den Bus ermöglichen. Im Schaltplan sind dies die Elemente mit den Namen ADDR_BUF, ALU_BUF, RX_BUF, PC_BUF, MRR_BUF und MDR_BUF. Einen Tristate-Treiber kann man sich dabei als einen Schalter vorstellen, der in Abhängigkeit von einem Steuersignal eine leitende Verbindung von seinem Daten-Eingang zu seinem Ausgang herstellt oder auch nicht. Im zweiten Fall liefert der Ausgang weder eine **0** noch eine **1**, sondern man sagt, er sei als dritter möglicher Zustand hochohmig (daher der Name Tristate). Es ist klar, dass dieses Konzept nur funktioniert, wenn

sichergestellt ist, dass nur höchstens einer der Tristate-Treiber einen Wert auf den Bus schaltet, weil es sonst doch wieder zu Kurzschlüssen kommen kann.

Aufgabe 3.4: 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, der seinen Eingangswert um Eins erhöht.

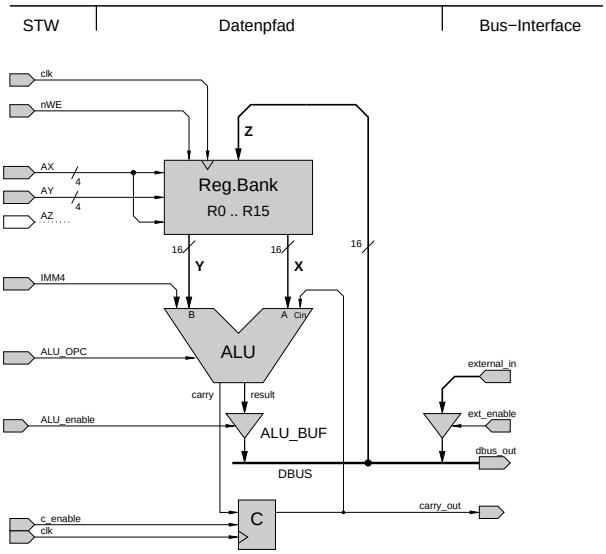
Versuchen Sie jetzt, durch geeignete interaktive Ansteuerung der Steuerleitungen einige Werte aus dem Eingabe-Schalter `I` in das Register `X` und aus `J` nach `Y` zu übertragen. Beachten Sie, dass Sie den Bus durch Ansteuerung der Tristate-Treiber auch ganz abschalten (Z) oder kurzschließen können (X).

3.4 ALU, Addierer



Zwei Addierer und eine sog. ALU finden sich an den farbig unterlegten Stellen im Schaltbild.

Die Funktion eines Addierers sollte klar sein (es werden einfach die an den beiden Eingängen liegenden Werte addiert).
Eine ALU (**A**rithmetical-**L**ogical-**U**nit) kann viel mehr, nämlich in Abhängigkeit von einigen Steuersignalen eine Vielzahl arithmetischer, logischer und Schiebe-Operationen ausführen.



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 später vom Steuerwerk geliefert.

Was die ALU, die wir für unseren Prozessor verwenden, kann, zeigt die Tabelle aus Abbildung 4. Dabei werden die fünf Bits 12 bis 8 des Befehlsworts (siehe Seite 3 und 4) direkt als ALU-Opcode verwendet.

Die in Abbildung 4 neben der Tabelle gezeigte Teilschaltung 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 Vorderflanke 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 (siehe Seite 2) 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.

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.

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. Für die möglichen Operationen, die die ALU ausführen kann, sei noch einmal auf Abbildung 4 verwiesen.

Öffnen Sie durch Klicken auf die Registerbank mit der rechten Maustaste den Editor, um die Registerinhalte direkt anzuzeigen (Verkleinern Sie eventuell 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` (was tut die ALU für den Opcode 1 0100?), 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 externen Eingang, dieser dient nur als Platzhalter für den Rest des Prozessors, etwa zum Laden der Register aus dem Speicher. Probieren Sie Ihr „Programm“ schrittweise am Simulator aus.

Dokumentieren Sie, welche Operationen dazu nacheinander ausgeführt werden müssen (Registeradressen `AX` und `AY`, Alu-Operation, Takte, etc.), da das Programm später noch einmal benötigt wird. Erweitern Sie Ihr Programm anschließend so weit, 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

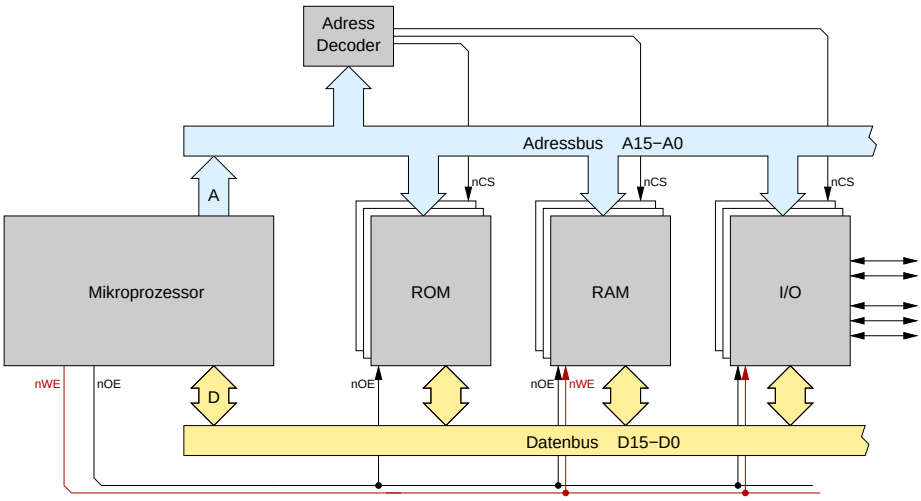


Abbildung 5: Gesamtsystem mit Prozessor, RAM, ROM und I/O Komponenten. Der Bus besteht aus Adress- und Datenbus sowie den Steuerleitungen nWE und nOE.

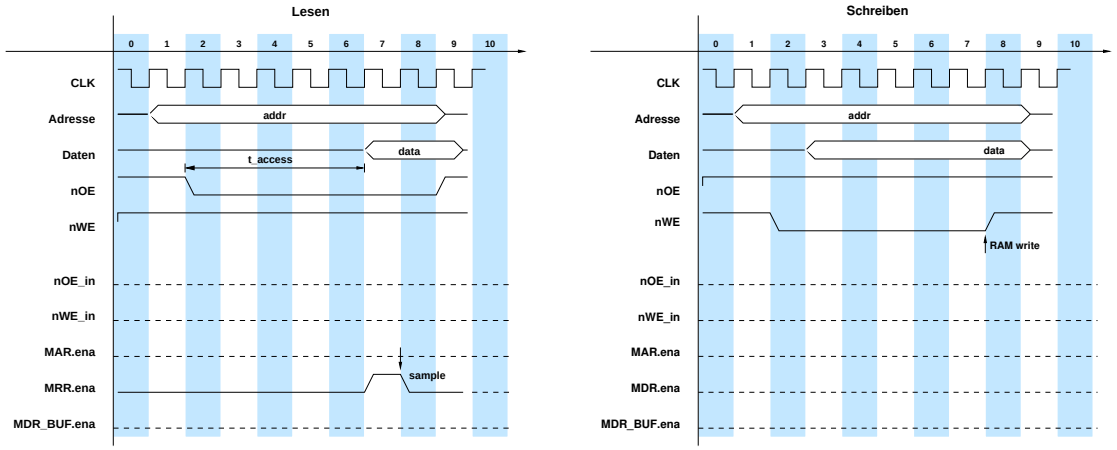


Abbildung 6: Zeitabläufe beim Speicherzugriff (Lesen und Schreiben). Der Lesezugriff wird über die `nOE` Leitung gesteuert, der Schreibzugriff über `nWE`. Nach Aktivieren des Steuersignals sind Wartezyklen notwendig, um die Zugriffszeit des Speichers einzuhalten.

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 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` (**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. D.h.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. 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 (hinter den Flipflops!) 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.

In Vorlesung und den Übungen haben Sie bereits Verfahren kennengelernt, um einfache Automaten als Schaltwerke mit Flipflops und (zweistufiger) Logik für die δ- und λ-Schaltnetze zu realisieren, z.B. 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, 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, wie es sich erweitert um das Register IR auch in unserem Prozessor finden lässt. 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

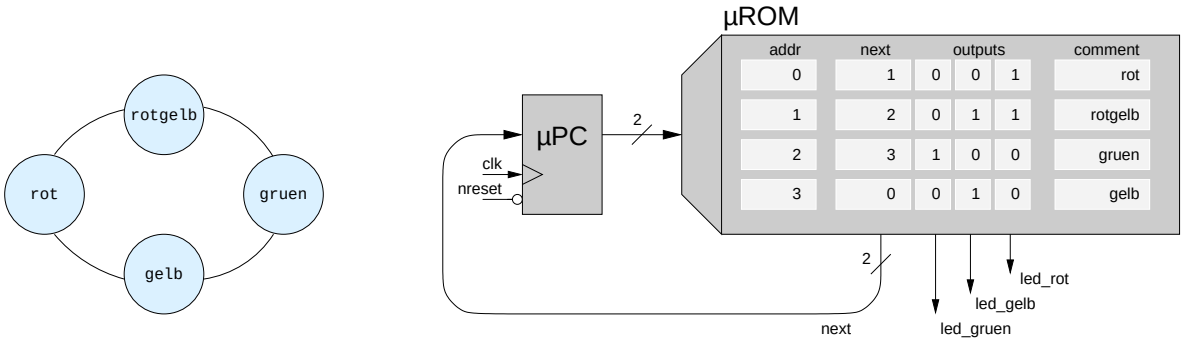


Abbildung 7: Prinzip der Mikroprogrammierung: einfacher Automat (links) und Realisierung mit linearem Mikroprogramm

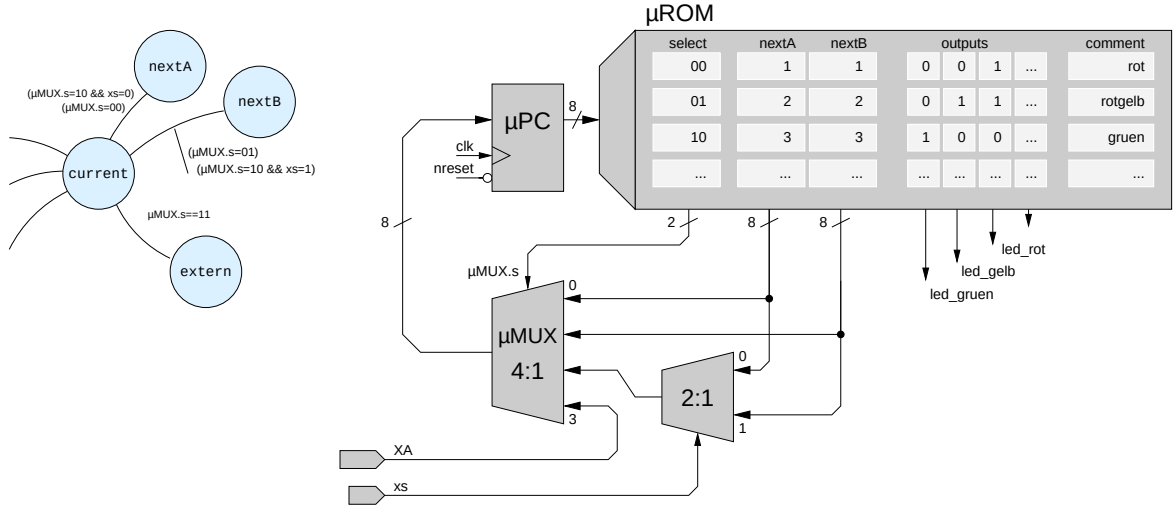


Abbildung 8: Mikroprogrammiertes Steuerwerk mit zwei externen Eingängen `xs` und `XA` und vierfacher Auswahl des Folgezustands (siehe Text).

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:

addr	nextA	nextB	MUX 4:1	LED f e d c b a 9 8 7 6 5 4 3 2 1 0	label
00	01	00	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					