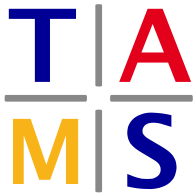
Praktikum Rechnerstrukturen

1

Mikroprozessorsysteme

Name, Vorname	
Bogen bearbeitet	abzeichnen lassen



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

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

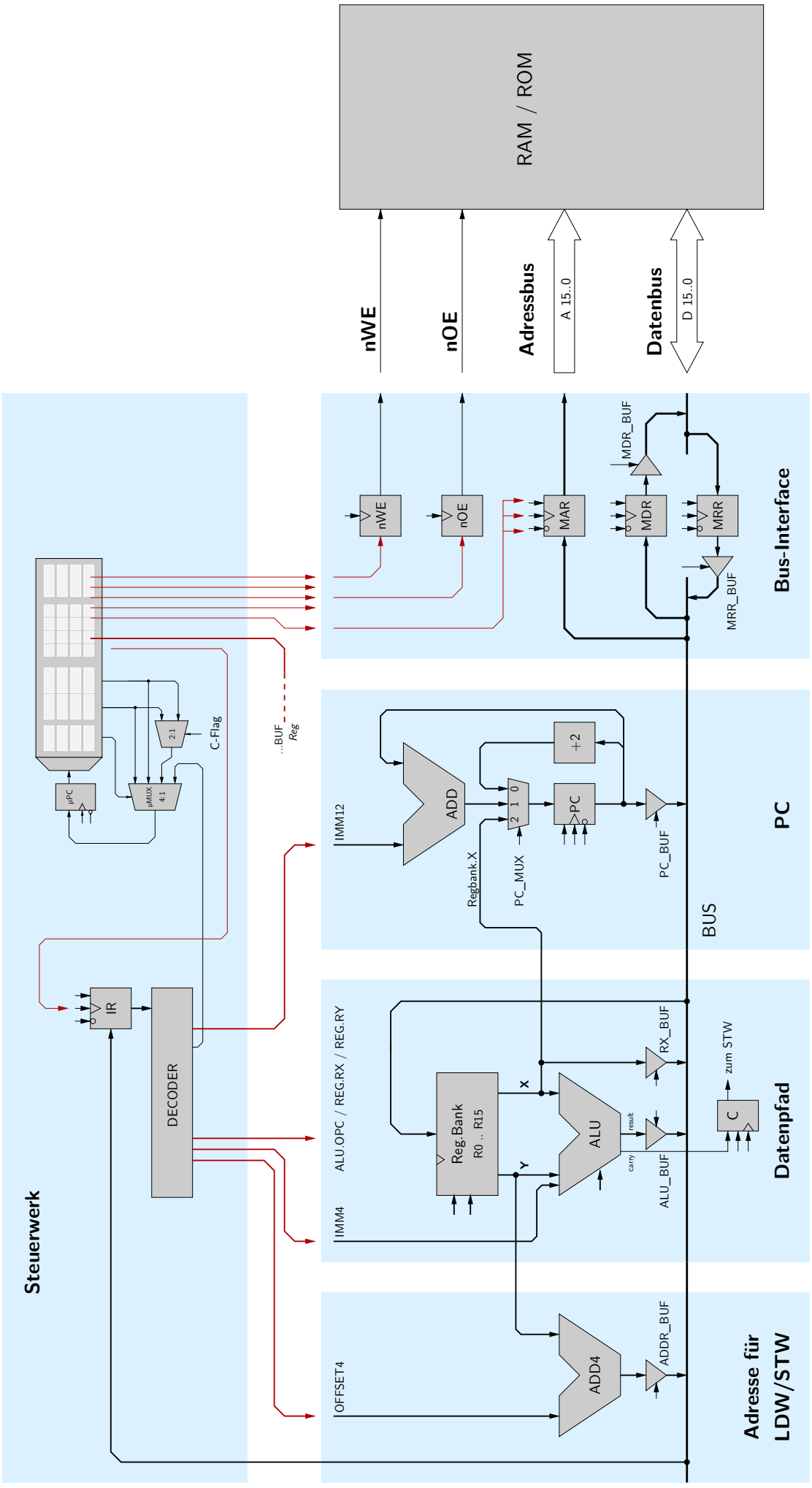


Abb. 1: Blockschaltbild des D·CORE Prozessors

1 Einführung und Motivation

Ziel des Praktikums *Rechnerstrukturen* ist die Vertiefung des Verständnis von Funktion und Struktur eines Mikroprozessorsystems. Um mit der Komplexität eines Computers umgehen zu können, hat sich die Einteilung in aufeinander aufbauende Abstraktionsebenen bewährt. In diesem Praktikum werden folgende Ebenen des Gesamtsystems an praktischen Beispielen behandelt: *Assemblerebene*, *Ebene der Maschinensprache*, *Register-Transfer Ebene*.

Um die Hierarchie der einzelnen Abstraktionsebenen deutlich zu machen, werden alle Aufgaben in einem *bottom-up* Vorgehen aufeinander aufbauen. Am ersten Praktikumstag werden Sie sich die Komponenten eines einfachen Mikroprozessors erarbeiten (auf der Register-Transfer Ebene), die dann schrittweise zum vollständigen Prozessor ergänzt werden. Im Verlaufe des zweiten Tages werden Sie beginnen auf diesem Prozessor sukzessive die Befehle, die er abarbeiten soll, zu implementieren (Befehlsarchitektur). Damit können dann auf dem quasi selbst implementierten Prozessor erste Assemblerprogramme geschrieben werden, deren Komplexität sich langsam erhöht.

Wegen der gegenüber einem echten Hardwareaufbau besseren Debug-Möglichkeiten werden die folgenden Versuche mit dem Hades-Simulator[1] durchgeführt. Hades ist public-domain Software und steht bei Interesse unter

`https://tams.informatik.uni-hamburg.de/applets/hades/webdemos`

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

Booten Sie auf Ihrem Praktikumsrechner Linux (Ubuntu) und erstellen Sie zunächst im Homeverzeichnis Ihres Praktikumaccounts ein Unterverzeichnis (Kommandozeile: `mkdir` ⟨*DirName*⟩) für die Dateien, die Sie im Laufe des Praktikums herunterladen bzw. erzeugen werden. Auf der Website des AB TAMS zum RS-Praktikum

`https://tams.informatik.uni-hamburg.de/lectures/2018ws/praktikum/rs`

stehen unter `Unterlagen` folgende Dateien zum Download bereit:

- `hades.jar` Software Archiv des Hades-Simulators
- `t3-hades.zip` Hades-Vorlagen für das Praktikum
- `T3asm.jar` Assembler und Simulator des Dcore-Prozessors (wird in Versuch 4 benötigt)

Laden Sie die Archive bzw. Dateien herunter und legen Sie sie im bereits angelegten Verzeichnis ab. Um die Zeit der Schaltplaneingabe zu sparen, finden Sie in dem Archiv `t3-hades.zip` vorbereitete Musterdateien für die nachfolgenden Versuche. Entpacken Sie die das Archiv mit unzip (Kommando: `unzip t3-hades.zip`). Es wird ein Unterverzeichnis `t3-hades` angelegt, das die benötigten Dateien und weitere Unterverzeichnisse, z. B. mit Assemblerprogrammen, enthält.

[1] ©1997-2005 by Norman Hendrich, Dept. Computer Science, Univ. of Hamburg

Eine Quick-Reference Karte (`hades-quick-reference.pdf`) mit der Kurzbedienungsanleitung finden Sie ebenfalls auf dem Webserver.
Sie starten Hades mit dem Shell-Kommando: `java -jar hades.jar`

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 *eingebetteten Systemen* 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 1-bit 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.

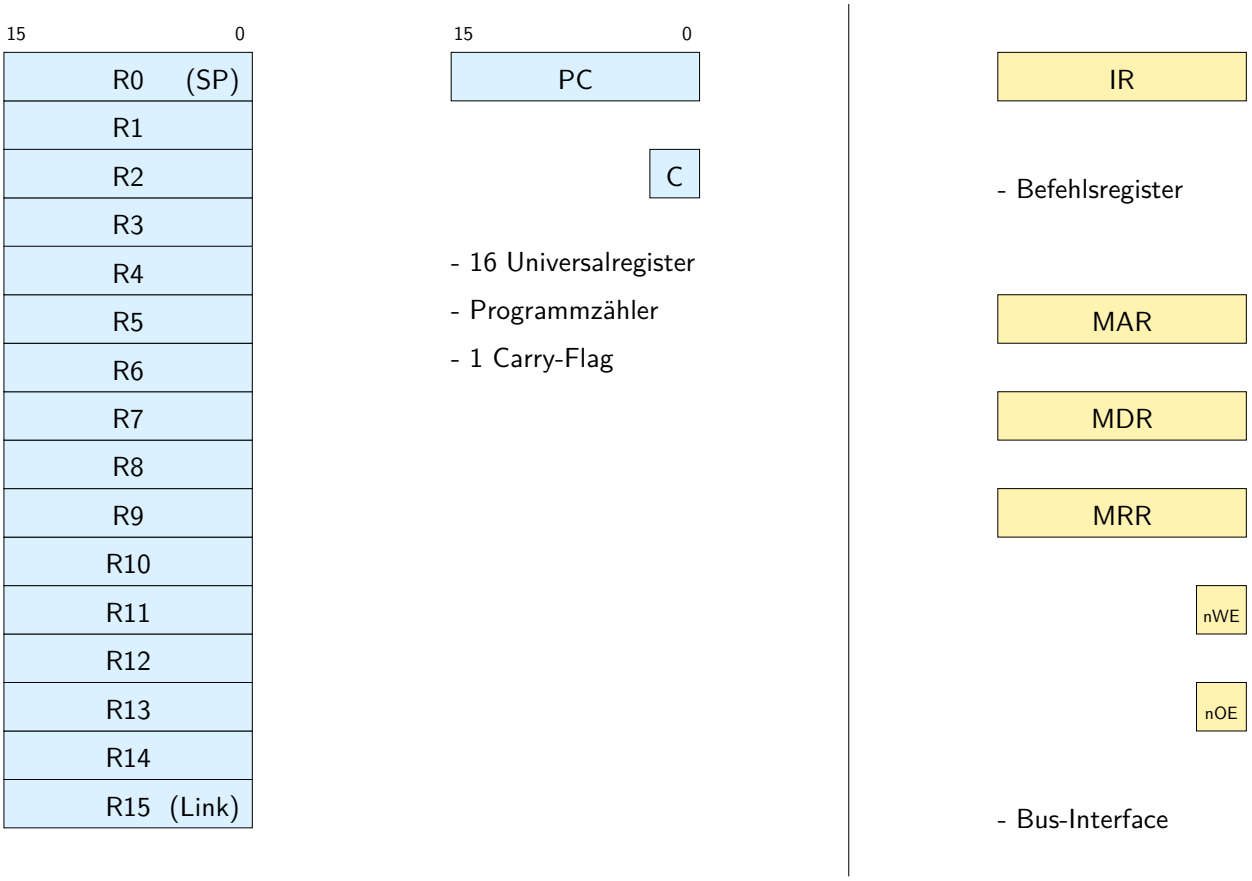


Abb. 2: Die Architektur (Programmiermodell) des D·CORE Prozessors: Programmzähler `PC`, 16 Universalregister und 1 Carry-Flag (links). Das Befehlsregister `IR` und die zusätzlichen Register des Businterface (`MAR`, `MDR`, `MRR`, `nOE`, `nWE`) sind dagegen nicht für Programme sichtbar (rechts).

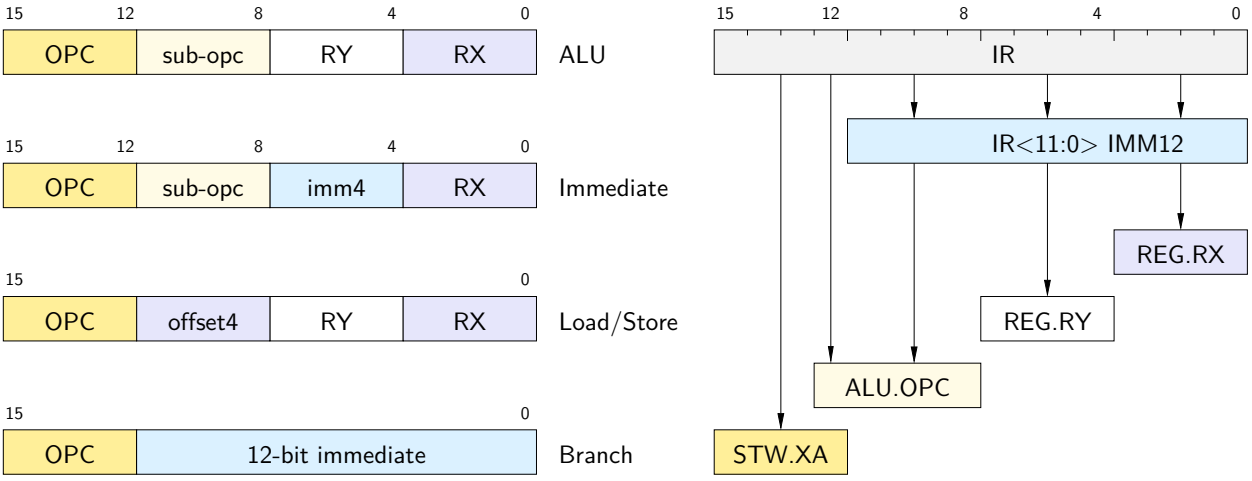


Abb. 3: Befehlsformate (links) und Decodierung der einzelnen Felder (rechts)

Mnemonic	Codierung	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])
cmpne	0011 0001 ⟨yyyy⟩ ⟨xxxx⟩	31yx	C = (R[x] != R[y])
cmpgt	0011 0010 ⟨yyyy⟩ ⟨xxxx⟩	32yx	C = (R[x] > R[y]) (signed)
cmplt	0011 0011 ⟨yyyy⟩ ⟨xxxx⟩	33yx	C = (R[x] < R[y]) (signed)
			Immediate-Operationen
movi	0011 0100 ⟨cccc⟩ ⟨xxxx⟩	34cx	R[x] = 0x000c
addi	0011 0101 ⟨cccc⟩ ⟨xxxx⟩	35cx	R[x] = R[x] + 0x000⟨c⟩
subi	0011 0110 ⟨cccc⟩ ⟨xxxx⟩	36cx	R[x] = R[x] - 0x000⟨c⟩
andi	0011 0111 ⟨cccc⟩ ⟨xxxx⟩	37cx	R[x] = R[x] AND 0x000⟨c⟩
lsli	0011 1000 ⟨cccc⟩ ⟨xxxx⟩	38cx	R[x] = R[x] << ⟨c⟩
lsri	0011 1001 ⟨cccc⟩ ⟨xxxx⟩	39cx	R[x] = R[x] >>> ⟨c⟩
bseti	0011 1010 ⟨cccc⟩ ⟨xxxx⟩	3Acx	R[x] = R[x] | (1 << ⟨c⟩) (set bit)
bclri	0011 1011 ⟨cccc⟩ ⟨xxxx⟩	3Bcx	R[x] = R[x] & !(1 << ⟨c⟩) (clear bit)
			Speicher-Operationen
ldw	0100 ⟨cccc⟩ ⟨yyyy⟩ ⟨xxxx⟩	4cyx	R[x] = MEM(R[y] + (0x000⟨c⟩<<1))
stw	0101 ⟨cccc⟩ ⟨yyyy⟩ ⟨xxxx⟩	5cyx	MEM(R[y] + (0x000⟨c⟩<<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	(C=1) ? PC = PC+2+⟨*imm12*⟩ : PC=PC+2
bf	1011 ⟨iiii⟩ ⟨iiii⟩ ⟨iiii⟩	Biii	(C=0) ? PC = PC+2+⟨*imm12*⟩ : PC=PC+2
jmp	1100 ⟨****⟩ ⟨****⟩ ⟨xxxx⟩	C**x	PC = R[x]
halt	1111 ⟨****⟩ ⟨****⟩ ⟨****⟩	F***	halt
			andere Opcodes illegal

⟨xxxx⟩, x: 4-bit Index des Quell- und Zielregisters RX ⟨xxxx⟩ – binär, x – hex
⟨yyyy⟩, y: 4-bit Index des Quellregisters RY ⟨yyyy⟩ – binär, y – hex
⟨cccc⟩, c: 4-bit Konstante IMM4 ⟨cccc⟩ – binär, c – hex
iii: 12-bit sign-extended Konstante IMM12 ⟨iiii⟩ – binär, i – hex
⟨****⟩, *: don't care

Tabelle 1: Befehlssatz des D·CORE

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 Befehlscodierung angegeben werden. Meistens gibt es deshalb eine kurze Tabelle zur Übersicht über alle Befehle und zusätzlich eine längere Beschreibung jedes einzelnen Befehls.

Eine möglichst reguläre Struktur des Befehlssatzes vereinfacht nicht nur die Struktur der Prozessor-Hardware, sondern erleichtert auch den Entwurf von Assembler und Compiler. Tabelle 1 enthält die Befehlsliste unseres D·CORE Prozessors. Es zeigt sich, dass alle Befehle der obige Liste insgesamt nur vier verschiedene *Befehlsformate* verwenden, die in Abbildung 3 dargestellt sind. In allen Befehlen werden die obersten vier Bits (15 . . . 12) für den *Opcode* verwendet; das Quell- und Zielregister `RX` wird durch die Bits (3 . . . 0) adressiert. Die einzelnen Felder lassen sich also trivial aus dem Befehlswort decodieren. (Zum Vergleich: M·CORE verwendet 14 verschiedene Befehlsformate, um die 65 536 möglichen Befehlsworte möglichst optimal ausnutzen zu können.)

3 Zahlendarstellungen

Bevor wir die einzelnen Hardware-Komponenten unseres D·CORE Processors besprechen, soll in diesem Abschnitt noch einmal darauf eingegangen werden, wie Zahlen üblicherweise codiert werden. Dieses Wissen und die Klarheit z. B. darüber, wie negative Zahlen dargestellt werden, ist für das Verständnis der Vorgänge auf der niederen Ebene unabdingbar.

Bemerkung: Auch wenn unser D·CORE Prozessor intern mit 16 Bit breiten Zahlen arbeitet, werden wir uns hier mit vier Bit breiten Zahlen begnügen. Eine Verallgemeinerung der hier gemachten Aussagen auf größere Bitbreiten sollte leicht möglich sein.

Viele höhere Programmiersprachen – aber z.B. nicht JAVA – kennen die beiden Datentypen **integer** für vorzeichenbehaftete Zahlen und **unsigned** für Zahlen ≥ 0. In vier Bit würde der Typ **unsigned** also üblicherweise repräsentiert werden durch folgende Bitkombinationen:

Zahl	binär	Zahl	binär
0	0000	8	1000
1	0001	9	1001
2	0010	10	1010
3	0011	11	1011
4	0100	12	1100
5	0101	13	1101
6	0110	14	1110
7	0111	15	1111

Für den Datentyp **integer** sind allerdings verschiedene Codierungen denkbar, von denen wir drei betrachten wollen.

3.1 Darstellung durch Vorzeichen und Betrag

Dies ist die Darstellung, an die man spontan denken würde, weil sie vom Dezimalsystem her bekannt ist. Übertragen auf das Binärsystem bedeutet dies, dass das höchstwertigste Bit als Vorzeichen interpretiert wird und die restlichen drei Bit den Betrag der Zahl darstellen. Man hätte dann also folgende Tabelle:

Zahl	binär	Zahl	binär
0	0000	(-0)	1000
1	0001	-1	1001
2	0010	-2	1010
3	0011	-3	1011
4	0100	-4	1100
5	0101	-5	1101
6	0110	-6	1110
7	0111	-7	1111

Es fällt sofort auf, dass es eine „negative Null“ gibt, d.h. eines der Bitmuster repräsentiert keine sinnvolle Zahl. Außerdem muss bei arithmetischen Operationen bekannt sein, um was für einen Datentyp es sich handelt. Z.B. sollte gelten 0001 + 1001 = 1010, wenn es sich um Zahlen vom Typ **unsigned** handelt, aber 0001 + 1001 = 0000 beim Datentyp **integer**. D.h. auf der Ebene der Hardware braucht man entweder zwei verschiedene Addierwerke oder man muss die Zahlen erst einmal konvertieren, was wertvolle Zeit kostet. Aus diesen Gründen ist die Darstellung als Vorzeichen und Betrag unüblich.

3.2 Darstellung im Einerkomplement

Negative Zahlen werden hier so dargestellt, dass sie das Einerkomplement der entsprechenden positiven Zahl sind (Invertieren aller Bits). Man erhält dann folgende Codierung.

Zahl	binär	Zahl	binär
0	0000	(-0)	1111
1	0001	-1	1110
2	0010	-2	1101
3	0011	-3	1100
4	0100	-4	1011
5	0101	-5	1010
6	0110	-6	1001
7	0111	-7	1000

Auch hier fällt wieder die Existenz einer „negativen Null“ auf. Was die Arithmetik angeht, hat sich die Situation allerdings etwas gebessert: 0001 + 1001 = 1010 liefert sowohl für den Datentyp **unsigned** als auch für den Datentyp **integer** das richtige Ergebnis. Dies ist immer der Fall, solange einer der Operanden nicht die negative Null ist, so dass man wieder einen störenden Spezialfall zu betrachten hat. Trotzdem hat es in der Frühzeit des Computerbaus wirklich Rechner gegeben, die intern mit dieser Zahlendarstellung gearbeitet haben.

3.3 Darstellung im Zweierkomplement

Negative Zahlen werden hier so dargestellt, dass sie das sog. Zweierkomplement der entsprechenden positiven Zahl sind (Invertieren aller Bits und Addition von Eins oder Subtraktion von Eins und dann Invertieren aller Bits). Man erhält dann folgende Codierung.

Zahl	binär	Zahl	binär
0	0000	-1	1111
1	0001	-2	1110
2	0010	-3	1101
3	0011	-4	1100
4	0100	-5	1011
5	0101	-6	1010
6	0110	-7	1001
7	0111	-8	1000

Dies ist die allgemein übliche Darstellung von Zahlen vom Typ **integer** in einem Rechner, die auch wir verwenden werden. Der kleine Mangel, dass es zur -8 keine entsprechende positive Zahl gibt, fällt kaum ins Gewicht. Viel wichtiger ist, dass man bei der arithmetischen Operationen auf der Ebene der Hardware nicht mehr zwischen den beiden Datentypen **unsigned** und **integer** zu unterscheiden braucht. Mathematisch lässt sich dies dadurch erklären, dass man im Restklassenring modulo 16 rechnet und einfach verschiedene Repräsentanten der Klassen wählt. Auch wenn kaum jemand auf die Idee kommen würde, möglich wäre in einer Zweierkomplement-Darstellung auch folgende Interpretation der Bitmuster:

Zahl	binär	Zahl	binär
0	0000	-1	1111
1	0001	-2	1110
2	0010	-3	1101
3	0011	-4	1100
4	0100	-5	1011
5	0101	10	1010
6	0110	9	1001
7	0111	8	1000

Auch dann würde die Arithmetik (bis auf immer mögliche Bereichsüberschreitungen) immer noch richtig funktionieren.

3.4 Zahlen im Hexadezimal-System

Wenn man mit einer größeren Bitbreite als 4 arbeitet, ist es im Dualsystem lästig, z.B. 16 Nullen und Einsen aufschreiben zu müssen. Man geht daher gerne zum Hexadezimal-System über, d.h. einem Zahlensystem zur Basis 16. Die Umrechnung von einer Binärzahl zu einer Hexadezimalzahl und umgekehrt ist dabei nicht schwierig, wie wir sehen werden.

Betrachten wie zunächst ein „exotisches“ Beispiel, das trotzdem deutlich macht, warum das angebene Verfahren funktioniert:

Die Dezimalzahl 1234567890 soll in eine Zahl zur Basis 100 konvertiert werden. Im Hunderter-System gebe es dabei die Ziffern **[00]**, **[01]**, ... **[99]**. Weil 100 eine Potenz von 10 ist, ist dies offenbar eine (fast) triviale Aufgabe. Es ist

$$(1234567890)_{10} = (\textbf{[12]}\,\textbf{[34]}\,\textbf{[56]}\,\textbf{[78]}\,\textbf{[90]})_{100}$$

Weil nun 16 eine Potenz von 2 ist, ist die Umrechnung einer Binärzahl in eine Hexadezimalzahl und umgekehrt ähnlich einfach, indem man den Bitkombinationen 0000, 0001, ...1111 die Hexadezimalziffern 0,1,...9, A, B, C, D, E, F zuordnet. Es ist also z.B.

$$(1100\ 0001\ 0110\ 1001)_2 = (C169)_{16}$$

Aufgabe 1.2 Umrechnung von Zahlen

Vervollständigen Sie folgende Tabelle

binär	hexadezimal
`1000 1100 1110 0001`	
`1010 0101 1001 0110`	
	`A106`
	`1248`

Ergänzen Sie die Tabelle (alle Angaben im **Hexadezimalsystem** und vom Typ **integer**)

A	-A (Vorzeichen/Betrag)	-A (Einerkomplement)	-A (Zweierkomplement)
`0001`			
`FFF0`			

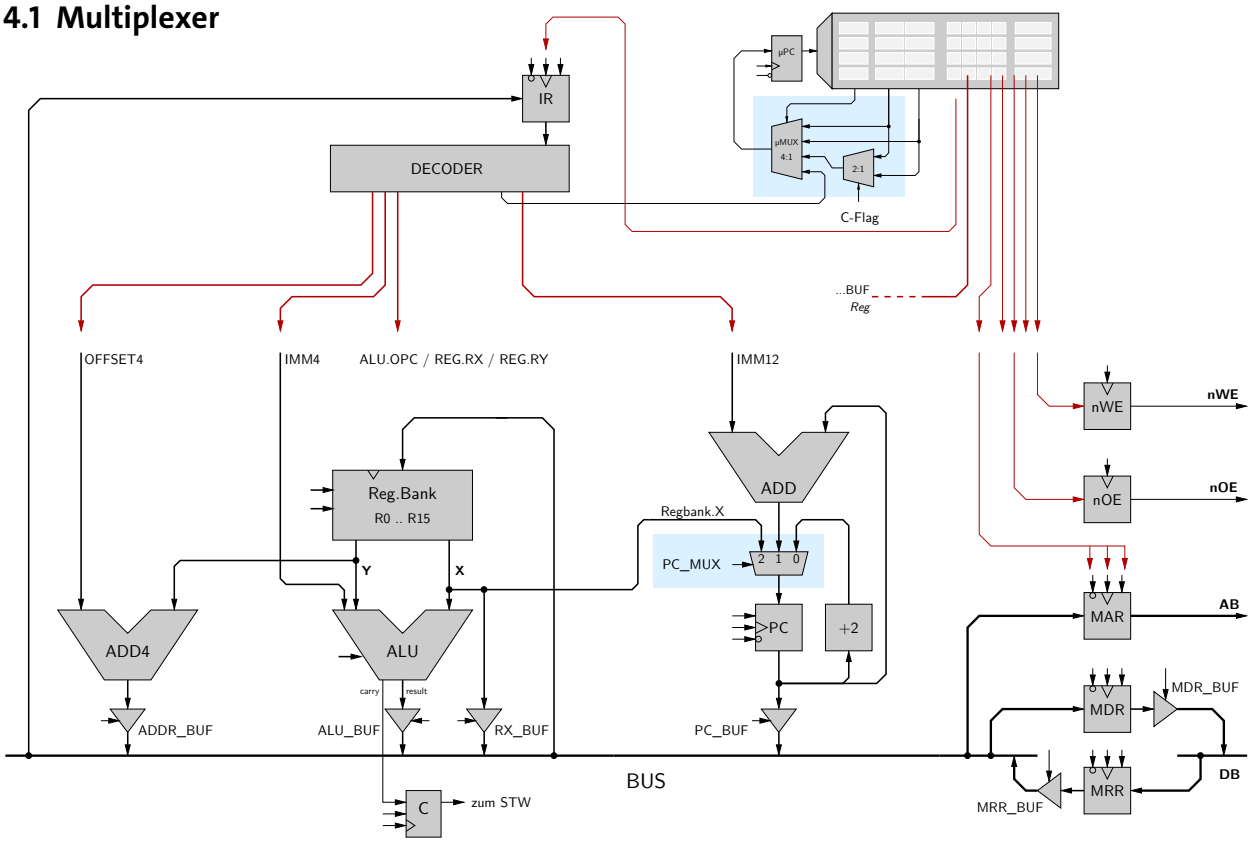
Auf Nachfrage sollten Sie erläutern können, welche der drei vorgestellten Einbettungen der negativen Zahlen üblicherweise gewählt wird und warum?

Aufgabe bearbeitet	abzeichnen lassen

4 Komponenten des D·CORE Prozessors

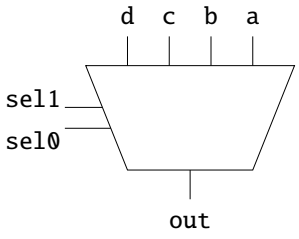
Wir wollen jetzt nacheinander die Komponenten des Schaltbildes des D·CORE Prozessors der Abbildung 1 betrachten.

4.1 Multiplexer



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

Ein 4-zu-1-Multiplexer wie im folgenden Bild realisiert also die Funktion:



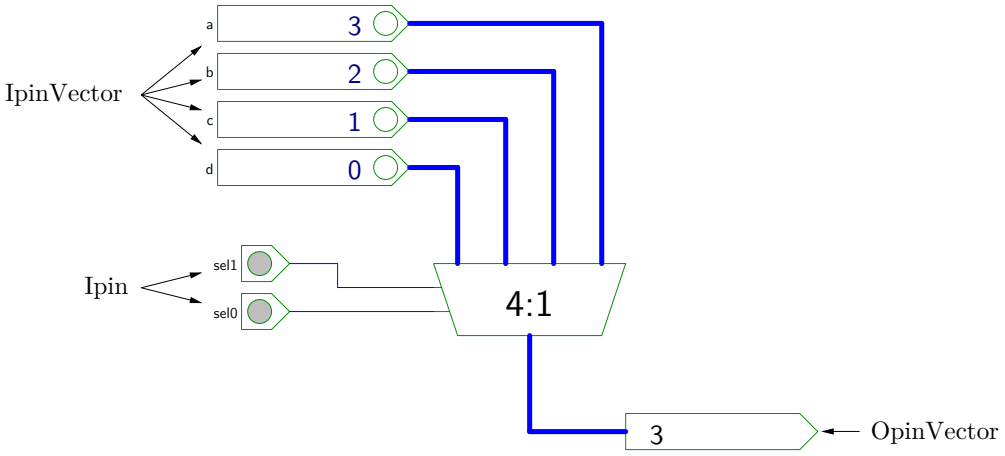

```
case sel
  "00":  out = a;
  "01":  out = b;
  "10":  out = c;
  "11":  out = d;
end case;
```

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 1.3 Multiplexer

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

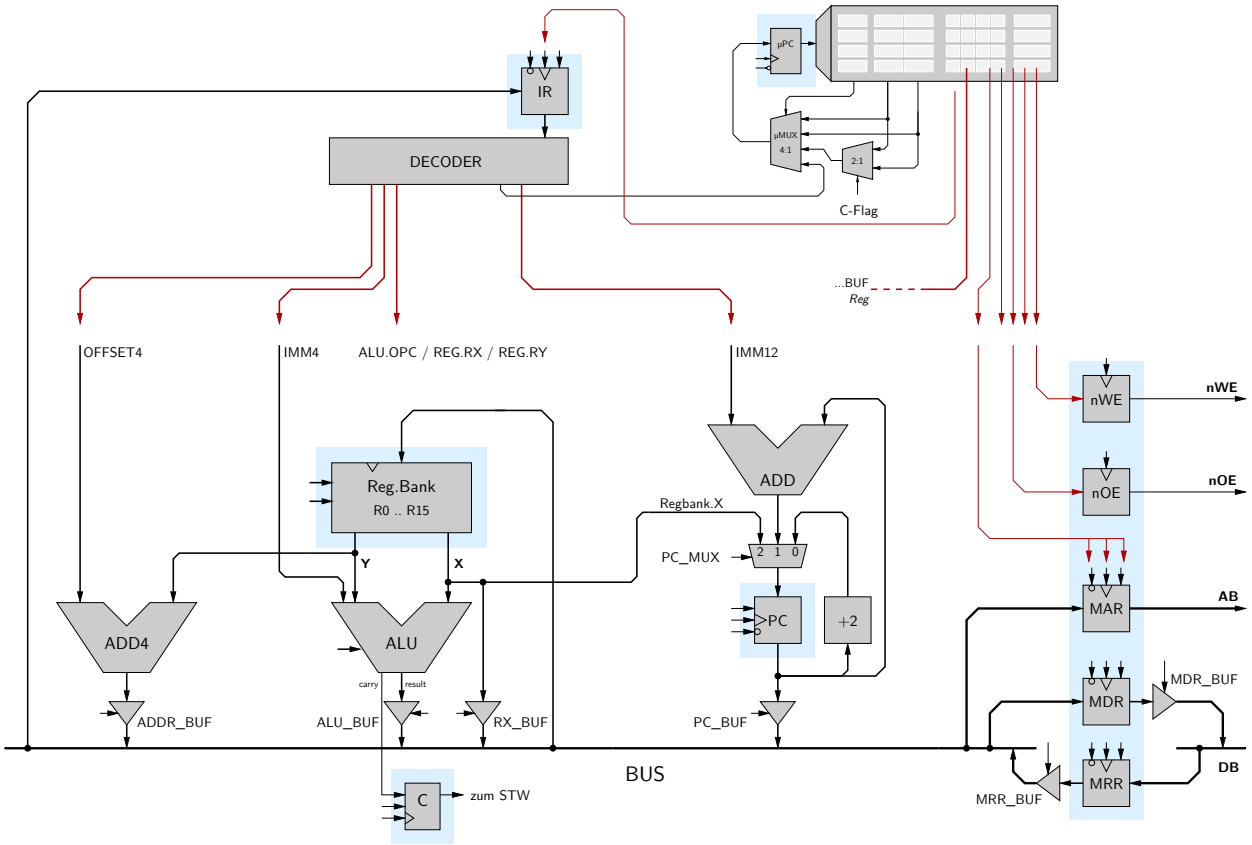


Ein sog. *Ipin* dient dabei als Eingang für ein Signal, das ein Bit breit ist. Die beiden *Ipins* sollten dabei zu Anfang die Farbe Blau haben als Zeichen, dass ihnen noch kein Wert zugewiesen worden ist (weder eine logische **0** noch eine logische **1**). Ändern lässt sich dies, indem man auf den *Ipin*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

4.2 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 1.4 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 $\mathbf{0} \rightarrow \mathbf{1}$ Übergang auf dem Takteingang (Vorderflanke) wird der Wert von *Data* gespeichert und auf dem Ausgang Q ausgegeben.

Zwei Flipflops in dieser einfachen Form sind im Schaltbild rechts zu finden und dienen zum Speichern der Signale *nWE* und *nOE*.

Das nächste Flipflop in **`register.hds`** ist ein etwas 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-Eingang anliegt, mit der nächsten Vorderflanke des Taktes wirklich übernommen.

(a) Vollziehen Sie dieses Verhalten bitte nach, indem Sie 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. Anstatt den Dateneingang mit 16 *Ipin*s zu versorgen, ist am Dateneingang ein *IpinVector* und am Ausgang entsprechend ein *OpinVector* angeschlossen.

(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, wenn Sie mit der rechten Maustaste auf das Symbol der Registerbank klicken und **Edit** wählen.

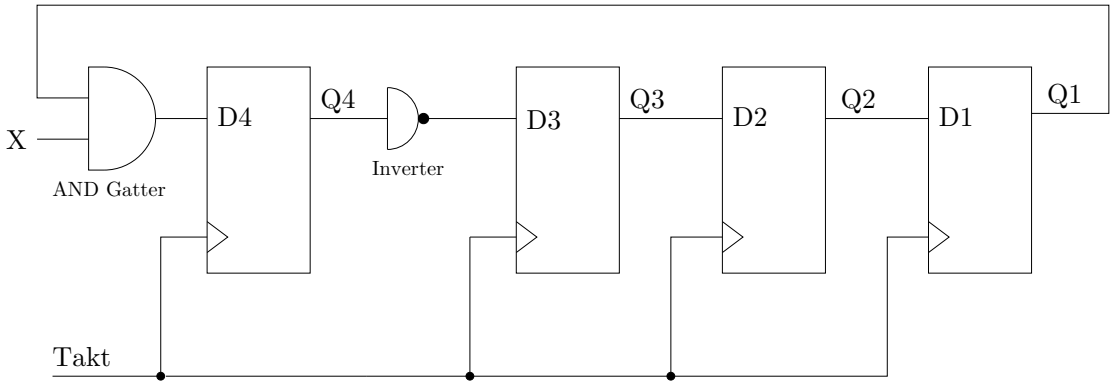
Weshalb ist für die Registerbank kein Reset-Signal vorgesehen?

Aufgabe bearbeitet	abzeichnen lassen

4.3 Exkurs – Schaltwerke

Eine Schaltung, die Flipflops enthält, wollen wir etwas ungenau ein *Schaltwerk* nennen und die Werte der Q-Ausgänge der Flipflops den *Zustand* des Schaltwerks. Schon das einfachste Beispiel eines flankengesteuerten D-Flipflops, dessen Q-Ausgang über einen Inverter auf seinen D-Eingang zurückgeführt wird, zeigt, dass die Ausgabe eines Schaltwerks vom Zustand abhängt in dem es sich gerade befindet und, anders als bei einem *Schaltnetz*, nicht nur von seinen externen Eingaben.

Wie ein Blick auf den Schalplan des D·CORE Prozessors zeigt, ist er im Grunde ein großes Schaltwerk mit einer riesigen Zahl von Zuständen. Wir wollen uns hier nicht mit dem Entwurf solcher Schaltwerke beschäftigen, aber doch wenigstens eine einfache Schaltung dieser Art betrachten, die neben dem Takt nur einen externen Eingang **X** hat.



Aufgabe 1.5 Zustände eines Schaltwerks

Bestimmen Sie die Zustände, die durchlaufen werden, wenn dieses Schaltwerk zu Beginn den Zustand 0000 hat und dann getaktet wird. Beachten Sie dabei, dass die Flipflops **gleichzeitig** schalten, wenn auf dem Takteingang eine Vorderflanke kommt!

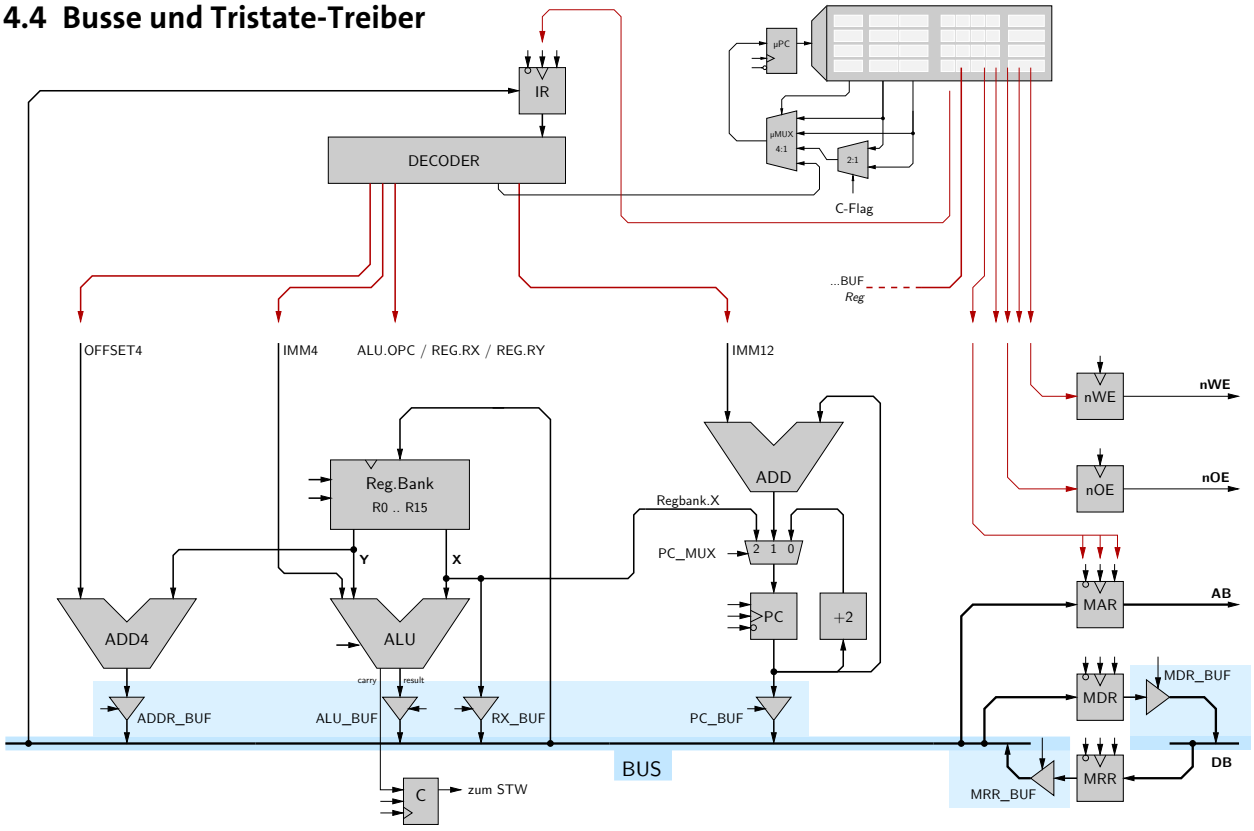
X=0

Takt	Zustand Z			
	Q4	Q3	Q2	Q1
0	0	0	0	0
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				

X=1

Takt	Zustand Z			
	Q4	Q3	Q2	Q1
0	0	0	0	0
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				

4.4 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, auf das 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 für 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 zur gegebenen Zeit auf eine Leitung schreibt.

Eine Möglichkeit wäre es, alle Ausgänge erst einmal auf einen Multiplexer zu führen und dessen Ausgang dann in Abhängigkeit von Steuersignalen auf das Leibungsbündel 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 höchstens einer der Tristate-Treiber einen Wert auf den Bus schaltet, weil es sonst doch wieder zu Kurzschlüssen kommen kann.

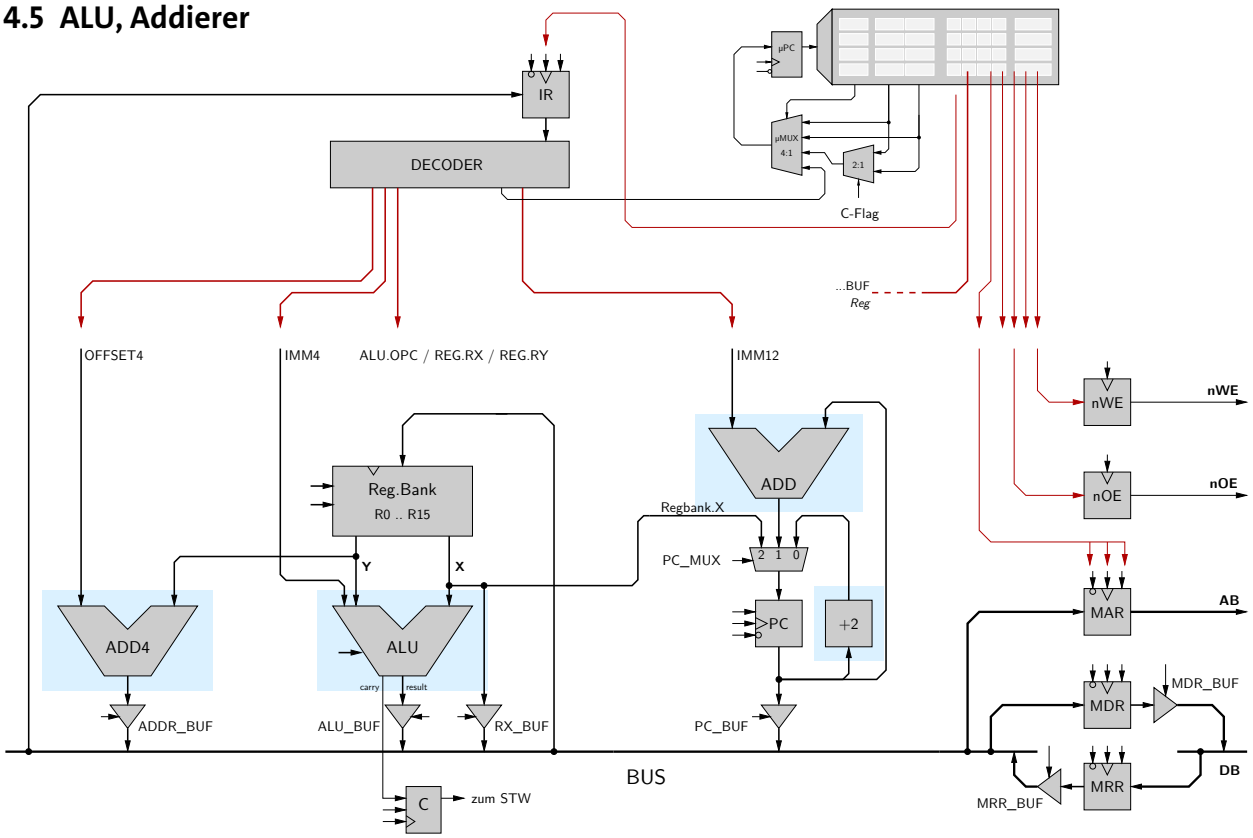
Aufgabe 1.6 RT-Komponenten in Hades

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 und einen Inkrementer, der seinen Eingangswert immer 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 entsprechende Ansteuerung der Tristate-Treiber auch ganz abschalten (Z) oder kurzschließen können (X).

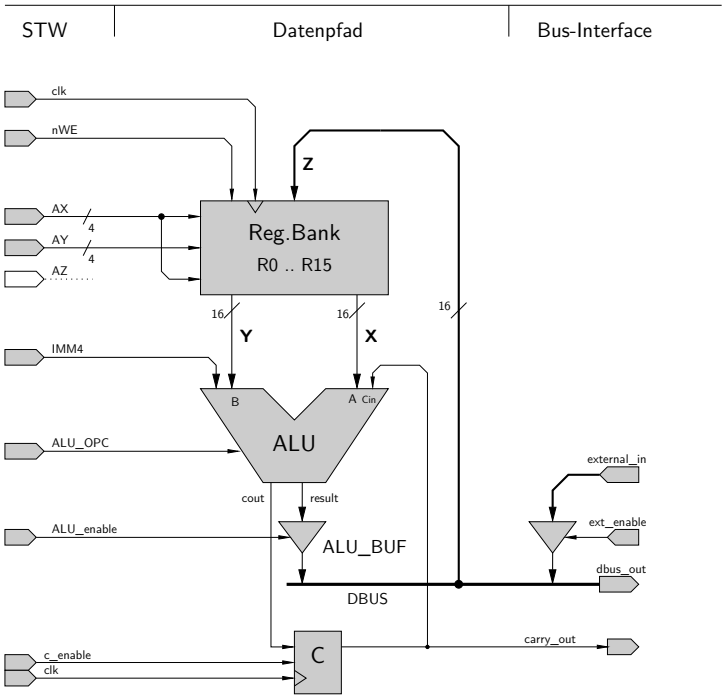
Aufgabe bearbeitet	abzeichnen lassen

4.5 ALU, Addierer



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

Die Funktion eines Addierers sollte klar sein (es werden einfach die an den beiden Eingängen liegenden Werte addiert), der Addierer neben dem Programmzähler (PC) rechnet mit einer Konstanten +2. 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	A, C = (A == B)
1 0001	A, C = (A != B)
1 0010	A, C = (A > B)
1 0011	A, 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 in A
1 1011	clear bit IMM4 in A
1 11**	undefined

Abb. 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 kann, die wir für unseren Prozessor verwenden, zeigt die Tabelle aus Abbildung 4. Dabei werden die fünf Bits (12…8) des Befehlsworts (siehe Seite 5 und 6) später direkt als ALU-Opcode verwendet.

- Man beachte dabei noch, dass „>>>“ einen **logischen** Shift bezeichnet (es werden Nullen nachgeschoben), „>>“ dagegen einen **arithmetischen** Shift (das ganz links stehende „Vorzeichen“-Bit 15 wird vervielfacht und ändert sich nicht). Beispiel: $(1000)_2 >>> 1 = (0100)_2$ aber $(1000)_2 >> 1 = (1100)_2$.
- Mit den Befehlen „set bit“ und „clear bit“ kann man genau ein Bit im Operanden A setzen bzw. löschen.
- Man beachte weiter, dass Vergleichsoperationen **vorzeichengerecht** durchgeführt werden, also z.B. gilt $(FFFF)_{16} < 0$ und $2 > (FFF0)_{16}$.
- Noch ein Hinweis: Die Bezeichnung `B.<3:0>` sagt aus, dass in B nur die Bit 3 bis 0, d.h. die vier niederwertigsten Bits betrachtet werden.

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 Universalre-

gister 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 ist (low-Pegel!), 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 2ff) 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 sowie 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 und Speicher. Wichtig ist hier die Speicherung und der Transfer von Registerinhalten, nicht aber die eigentliche Realisierung der Komponenten aus Hunderten oder Tausenden von einzelnen Logikgattern.

Aufgabe 1.7 ALU-Operationen

Tragen Sie in die Tabelle ein, was die ALU für folgende Eingaben am Ausgang liefern würde. Alle Zahlen sind hexadezimal zu verstehen und die zugrunde liegende Zahlendarstellung ist das Zweierkomplement:

ALU_OPC	A	B / imm4	Cin	Result	Cout
00	FFFF	0000	0		
01	FFFF	0002	0		
01	000A	0006	1		
02	000A	0006	1		
03	0001	0002	0		
04	1234	000F	1		
05	1234	000F	0		
06	0770	673B	0		
07	FFFF	0001	1		
08	0001	0001	1		
09	F000	0001	0		
0A	F000	FFF1	0		
0C	AFFE	000F	0		
0D	8000	000F	1		
0E	8000	000F	0		
10	FFFF	2222	1		
11	FFFF	2222	1		
12	FFFF	2222	1		
13	FFFF	2222	1		
14	0006	0004	1		
15	0006	0004	1		
16	0006	0004	1		
17	0006	0004	1		
18	0006	0004	1		
19	0006	0004	1		
1A	0006	0004	1		
1B	0006	0004	1		

Eine Web-Seite, mit der Sie ihre Ergebnisse testen können, findet sich unter der URL
https://tams.informatik.uni-hamburg.de/lectures/2018ws/praktikum/rs/downloads/tester.html
oder einfach auf der Praktikums-Website unter Aufgaben.

Aufgabe 1.8 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 können diese Werte ebenso wie der

Ausgabewert der ALU undefiniert sein. Belegen Sie zu Beginn die Eingänge der ALU (`ALU_OPC` und `ALU_IMM`) mit geeigneten Werten, so dass sie an ihrem Ausgang eine 0 liefert (siehe Abb. 4). Wählen Sie die Register-Zieladresse `0` aus, aktivieren Sie das Write-Enable der Registerbank (active low!) und sorgen Sie dafür, dass der Ausgang der ALU auf den Bus geschaltet wird. Beim der nächsten Vorderflanke des Taktes wird dann `R0` auf den Wert `0` gesetzt. Sie haben jetzt das Register 0 mit dem Wert 0 initialisiert.
Initialisieren Sie auf diese Weise noch die Register `R11` und `R12` mit dem Wert `0` und die Register `R13` bis `R15` mit dem Wert `1`. Spielen Sie anschließend ein bisschen mit den Steuerleitungen herum, um die Funktion der Registerbank und der ALU zu verstehen.

Aufgabe 1.9 Einfache ALU-Operationen

Überlegen Sie sich jetzt, wie Sie durch entsprechende Bedienung der Steuersignale nacheinander die einzelnen Register auf die folgenden Werte setzen können:

$$\texttt{R0} = 0,\ \texttt{R1} = 1,\ \texttt{R2} = 2,\ \texttt{R3} = 4,\ \texttt{R4} = 8,\ \ldots,\ \texttt{R9} = 256.$$

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 im Simulator aus.

Dokumentieren Sie, welche Operationen dazu nacheinander ausgeführt werden müssen (Registeradressen `AX` und `AY`, Alu-Operation, Takte etc.).

Schritt	ALU_OPC	AX	AY
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			

Aufgabe bearbeitet	abzeichnen lassen