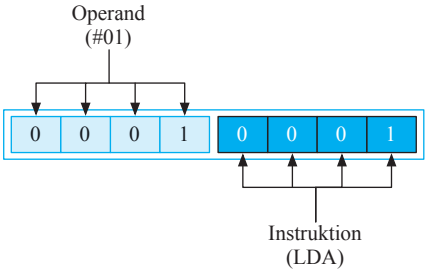


Abbildung 3.8: Das Instruktionsformat unseres Modellprozessors. Die oberen 4 Bit codieren den Operanden, die unteren 4 Bit den auszuführenden Befehl.

3.2 Ein einfacher Modellprozessor

Nachdem wir uns einen Überblick über den prinzipiellen Aufbau eines Mikroprozessors verschafft haben, werden wir in diesem Abschnitt unsere Kenntnisse in die Praxis umsetzen und einen voll funktionstüchtigen Mikroprozessor konstruieren. Obwohl sich die Komplexität und Rechenleistung des Modellprozessors in keiner Weise mit modernen Prozessoren vergleichen lässt, basiert er auf den gleichen Grundprinzipien und eignet sich dadurch bestens für einen praxisorientierten Einstieg in die Mikroprozessortechnik. Um die Komplexität des Modellprozessors gering zu halten, nehmen wir eine Reihe von Vereinfachungen vor (vgl. [6]):

- Die Registerbreite beträgt 4 Bit. Die Beschränkung ist lediglich der einfacheren Verdrahtung der beschalteten Komponenten geschuldet. Durch eine entsprechende Mehrfachauslegung der Register sowie der Ein- und Ausgangsleitungen lässt sich die Bitbreite ohne konzeptionelle Änderungen nahezu beliebig erweitern. Moderne Prozessoren der höheren Leistungskategorien arbeiten heute in der Regel mit einer Breite von 64 Bit, im Bereich eingebetteter Systeme sind jedoch auch 4- und 8-Bit-Prozessoren vertreten.

- Die Bitbreite des Adressbusses entspricht der Registerbreite. Das Zusammenfallen beider Größen erleichtert die Adressierung des Hauptspeichers erheblich, da die Speicheradresse nicht aus mehreren Datenwörtern zusammengesetzt werden muss. Diese Vereinfachung ist keineswegs praxisfern, schließlich können moderne 64-Bit-Prozessoren den gesamten Hauptspeicher ebenfalls mit einem einzigen Datenwort adressieren. 32-Bit-Prozessoren sind hier im Nachteil und benötigen eine komplexere Adressierungslogik.

- Jeder Befehl wird zusammen mit seinen Operanden in einem einzigen Datenwort codiert. Wie in Abbildung 3.8 dargestellt, besitzt jedes Datenwort eine Breite von 8 Bit, von denen die ersten 4 den Operanden und die letzten 4 den Maschinenbefehl codieren. Diese Vereinfachung bietet zwei wesentliche Vorteile. Zum einen werden Befehl und Operand in einem Schwung in die CPU geladen. Zum anderen gelangen wir, abgesehen von Sprungbefehlen, durch die Erhöhung des Instruktionszählers um eins stets zum nächsten Befehl. Trotzdem ist die Vereinfachung im Vergleich zu realen Prozessoren untypisch, da wir den möglichen Befehlssatz der CPU drastisch einschränken. Insbesondere lassen sich nur noch solche Befehle definieren, die mit maximal einem Operanden auskommen.

Des Weiteren bricht unser Modellprozessor an dieser Stelle mit den Grundsätzen der Von-Neumann-Architektur, da wir den Hauptspeicher durch unser Instruktionsformat effektiv in einen *Datenspeicher* und einen *Programmspeicher* separieren. Interpretieren wir die unteren und die oberen 4 Bit des Datenbusses als eigenständigen Transportweg, so verfügt unser Modellrechner über getrennte Daten- und Code-Speicher, die über separate Busse an die CPU angebunden sind. Unser Modellrechner folgt damit streng genommen dem Funktionsprinzip der Harvard-Architektur.

- Unser Modellrechner führt jeden Befehl in genau einem Taktzyklus aus. Hierzu legen wir die CPU-Komponenten zweiflankengesteuert aus, so dass wir innerhalb des Prozessors zwei Phasen unterscheiden können. Wie in Abbildung 3.9 dargestellt, wird der aktuell auszuführende Befehl in der ersten Hälfte eines Taktzyklus in die CPU geladen (*Fetch*). Anschließend wird der Befehl in der zweiten Takthälfte decodiert, ausgeführt und das Ergebnis zurückgeschrieben. Dass sich alle drei Phasen in unserem Prozessor nahezu zeitgleich ausführen lassen, verdanken wir dem vergleichsweise einfachen Befehlssatz. So entfällt zum Beispiel das Nachladen jeglicher Operanden, da unsere Modell-CPU ausschließlich Befehle unterstützt, die genau einen, mit dem Befehl codierten Parameter verarbeiten. Der genaue zeitliche Ablauf innerhalb der einzelnen Taktphasen ist in Abbildung 3.10 grafisch zusammengefasst. Das Laden des auszuführenden Befehls wird während der negativen Taktphase vorbereitet und synchron zur positiven Taktflanke vollzogen. Während der positiven Taktphase wird der Befehl decodiert und ausgeführt. Am Ende der positiven Taktphase steht das berechnete Ergebnis bereit und wird synchron zur negativen Taktflanke zurückgeschrieben.

- Das Rechenwerk unseres Modellrechners wird durch ein einziges Akkumulatorregister gebildet und verfügt über keine weiteren Universalregister. Damit beschränken sich die arithmetischen Fähigkeiten unserer Modell-CPU auf die einfache Addition und Subtraktion zweier Binärzahlen. Wird das Akkumulatorregister durch eine komplexe arithmetisch-logische Einheit ersetzt, so lassen sich die arithmetischen Fähigkeiten des Modellprozessors erweitern, ohne dessen grundlegende Struktur zu ändern.

- Neben dem RAM-Speicher existieren keine weiteren externen Komponenten – insbesondere entfallen sämtliche ROM- und I/O-Bausteine. Jede Speicherstelle des RAMs enthält ein einzelnes Instruktionswort, das dem weiter oben eingeführten Maschinenformat entspricht. Da der Adressbus mit 4 Bit ausgelegt ist, können wir nur magere 16 Speicherstellen adressieren, die für alle der später dis-

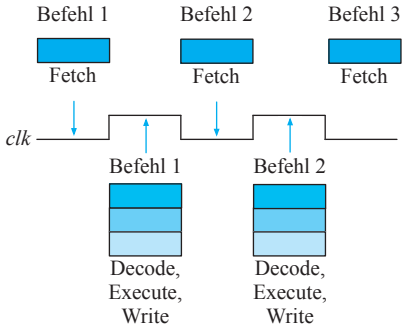


Abbildung 3.9: In jedem Taktzyklus führt die Modell-CPU genau ein Befehl aus.

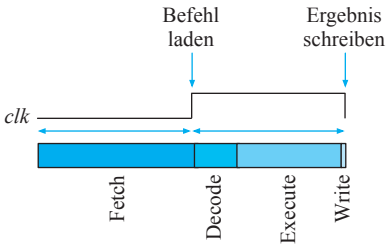


Abbildung 3.10: Zeitlicher Ablauf der Befehlsausführung

Nr	Befehl	Codierung				Beschreibung
0	`NOP`	0	0	0	0	Wartezyklus (*No Operation*)
			Lade- und Speicherbefehle			
1	`LDA #n`	0	0	0	1	Lädt den Akkumulator mit dem Wert *n*
2	`LDA (n)`	0	0	1	0	Lädt den Akkumulator mit dem Inhalt der Speicherstelle *n*
3	`STA n`	0	0	1	1	Überträgt den Akkumulatorinhalt in die Speicherstelle *n*
			Arithmetikbefehle			
4	`ADD #n`	0	1	0	0	Erhöht den Akkumulatorinhalt um den Wert *n*
5	`ADD (n)`	0	1	0	1	Erhöht den Akkumulatorinhalt um den Inhalt der Speicherstelle *n*
6	`SUB #n`	0	1	1	0	Erniedrigt den Akkumulatorinhalt um den Wert *n*
7	`SUB (n)`	0	1	1	1	Erniedrigt den Akkumulatorinhalt um den Inhalt der Speicherstelle *n*
			Sprungbefehle			
8	`JMP n`	1	0	0	0	Lädt den Instruktionszähler mit dem Wert *n*
9	`BRZ #n`	1	0	0	1	Addiert *n* auf den Instruktionszähler, falls das Zero-Bit gesetzt ist
10	`BRC #n`	1	0	1	0	Addiert *n* auf den Instruktionszähler, falls das Carry-Bit gesetzt ist
12	`BRN #n`	1	0	1	1	Addiert *n* auf den Instruktionszähler, falls das Negations-Bit gesetzt ist

Tabelle 3.1: Der vollständige Befehlssatz des Modellprozessors

kutierten Beispielprogramme jedoch vollkommen ausreichend sein werden.

Tabelle 3.1 fasst den Befehlssatz unseres Modellprozessors zusammen. Sehen wir von dem `NOP`-Befehl ab, der ausschließlich den Instruktionszähler weiterschaltet und keine sichtbare Aktion auslöst, so können wir die restlichen 11 Befehle in drei Gruppen einteilen:

- Lade- und Speicherbefehle

  Die Befehle dieser Gruppe dienen zum Laden des Akkumulators sowie dem Datenaustausch zwischen Akkumulator und Speicher. In der Form `LDA #n` lädt die CPU den Akkumulator mit dem konstanten Wert *n*, in der Form `LDA (n)` wird der in Adresse *n* gespeicherte Datenwert vom Hauptspeicher in den Akkumulator übertragen. In entsprechender Weise schreibt der Befehl `STA n` den Akkumulatorinhalt in die spezifizierte Speicherstelle zurück.

- Arithmetikbefehle

  Die Befehle dieser Gruppe dienen zur Durchführung arithmetischer Operationen. Unser Modellprozessor beschränkt sich mit den beiden Befehlen `ADD` und `SUB` auf die Addition und Subtraktion zweier

Zahlen in Zweierkomplementdarstellung. In der Form `ADD #n` bzw. `SUB #n` wird der Akkumulatorinhalt um den konstanten Wert *n* erhöht bzw. erniedrigt. In der Form `ADD (n)` bzw. `SUB (n)` wird der Akkumulator stattdessen um den Inhalt der Speicherstelle *n* erhöht bzw. erniedrigt.

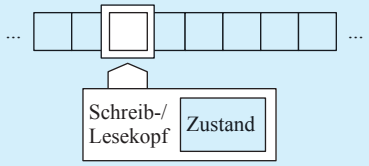
- Sprungbefehle

  Die Befehle dieser Gruppe dienen zur Manipulation des Instruktionszählers. Der Befehl `JMP n` überschreibt den Zähler mit *n* und implementiert auf diese Weise einen unbedingten Sprung. Zur Ausführung bedingter Sprünge stehen die Befehle `BRZ #n`, `BRC #n` und `BRN #n` zur Verfügung. So verzweigt der Befehl `BRZ` nur dann, wenn der Akkumulator den Wert 0 enthält – in diesem Fall wechselt das *Zero-Bit Z* auf 1. Im Gegensatz zum direkten Sprungbefehl, der die *absolute* Zieladresse als Argument entgegennimmt, führen die bedingten Sprungbefehle *relative* Sprünge aus. Folgerichtig wird der Instruktionszähler nicht überschrieben, sondern um *n* erhöht.

Obwohl der Befehlssatz im Vergleich zu modernen Prozessoren drastisch eingeschränkt ist, ändert dies nichts an der Universalität des Berechnungsmodells, das unserem Prozessor zu Grunde liegt. Nehmen wir an, dass die Größe des zur Verfügung stehenden Hauptspeichers unbegrenzt ist, so lässt sich jede berechenbare Funktion mit Hilfe unserer Modell-CPU implementieren. Insbesondere lässt sich damit jedes Assembler-Programm eines beliebigen Prozessors auf ein funktional äquivalentes Programm für unseren Modellprozessor reduzieren. Dass eine solche Reduktion ungeachtet der Komplexität des Befehlssatzes stets funktionieren muss, ist ein Ergebnis der Berechenbarkeitstheorie – eines der bedeutendsten Teilgebiete der theoretischen Informatik. Über die Länge und Effizienz des entstehenden Programms macht die Berechenbarkeitstheorie jedoch keine Aussage. In der Tat werden bereits unsere Beispielprogramme zeigen, dass sich die Umsetzung vieler Standardoperationen mitunter als reichlich umständlich erweist.

Abbildung 3.11 veranschaulicht die Architektur unseres Modellprozessors in Form eines Blockschaltbilds. In der Mitte der Darstellung ist mit dem Akkumulatorregister das Rechenwerk und damit der operative Kern der CPU abgebildet. Da der Prozessor neben dem Akkumulator keine weiteren Universalregister besitzt, müssen sämtliche arithmetischen Operationen hier ausgeführt werden. Über den aktuellen Zustand des Akkumulatorregisters gibt das Statusregister Aufschluss. Unser Modellprozessor beschränkt sich auf die drei Statusbits *C*, *Z*, *N*, die genau dann auf eins gesetzt werden, wenn das Addierwerk einen Überlauf verursacht (*Carry-Bit C*) oder der Akkumulator den Wert Null

1936 schlug der englische Mathematiker Alan Turing (1912 – 1954) ein universelles Automatenmodell vor, mit dem er den Grundstein der modernen Berechenbarkeitstheorie legte. Die sogenannte *Turing-Maschine* ist eine gedanklich konstruierte Steuereinheit, die ähnlich einem Tonband einzelne Zeichen lesen und schreiben kann. Hierzu steht neben einem Schreib-/Lesekopf ein – per Definition – unendlich langes Band zur Verfügung, das in einzelne Felder aufgeteilt ist. Jedes Feld kann mit genau einem Zeichen beschrieben werden.



Der Schreib-/Lesekopf wird über einen endlichen Automaten angesteuert, der die auszuführende Aktion aus dem internen Zustand sowie dem Inhalt des adressierten Speicherfelds bestimmt. Der Funktionsumfang der Turing-Maschine ist dabei äußerst simpel: Neben der Veränderung des aktuellen Speicherfelds sowie der schrittweisen Bewegung des Lesekopfes nach links oder rechts beherrscht die Maschine keine weiteren Operationen.
Niemand würde je in Erwägung ziehen, eine Turing-Maschine real zu bauen – alleine das unendliche Band stellt die Ingenieurkunst vor ein technisch unüberwindbares Problem. Nichtsdestotrotz ist die Turing-Maschine von unschätzbarem Wert, da sich auf ihr jeder nur erdenkliche Algorithmus implementieren lässt. Mit anderen Worten: Das Berechnungsmodell der Turing-Maschine ist *universell*. Dieses Ergebnis ist Inhalt der *Church'schen These*, die der amerikanische Logiker Alonzo Church im Jahre 1936 formulierte und die heute eine allgemein anerkannte Erkenntnis der Berechenbarkeitstheorie ist.

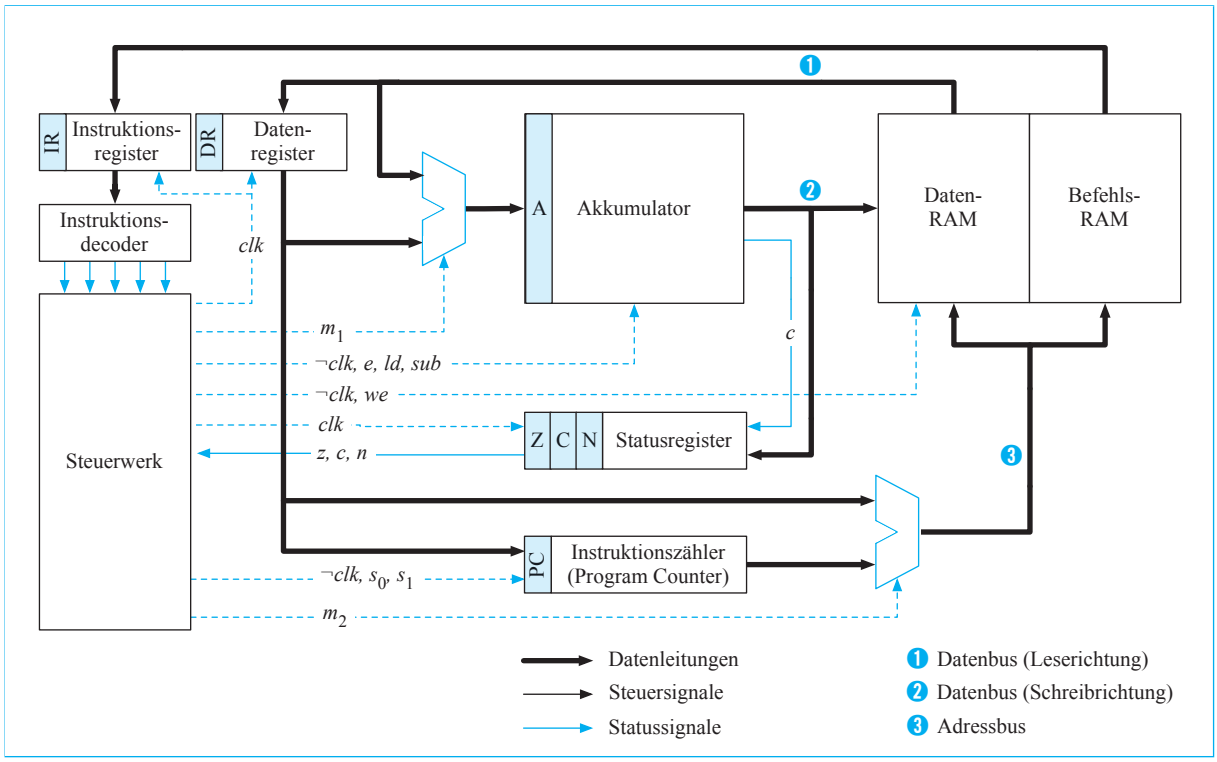


Abbildung 3.11: Schematischer Aufbau unseres Modellrechners

(*Zero-Bit Z*) bzw. einen negativen Wert enthält (*Negative-Bit N*).

Neben dem Akkumulator ist der Hauptspeicher abgebildet, der entsprechend unserem Instruktionsformat zweigeteilt ist. Während die linken vier Bits den Befehlsoperanden repräsentieren, codieren die rechten vier Bits den auszuführenden Befehl. Der RAM-Speicher ist asynchron ausgelegt, so dass zu jedem Zeitpunkt der Inhalt der über den Adressbus ausgewählten Speicherzelle auf dem Datenbus anliegt. Um die Implementierung des Datenbusses in unserem Modellprozessor so einfach wie möglich zu halten, ist er nicht bidirektional konzipiert. Stattdessen werden beide Transportrichtungen durch zwei separate Busse implementiert, die jeder für sich unidirektional arbeiten.

In jedem Takt führt die Modell-CPU genau einen Maschinenbefehl aus. Zu Beginn eines Taktzyklus muss der Adressbus so beschaltet sein, dass die nächste auszuführende Maschineninstruktion auf dem Datenbus liegt. Mit der positiven Taktflanke liest die CPU das Maschinenwort

ein, indem die oberen vier Bits in das *Datenregister DR* und die unteren vier Bits in das *Instruktionsregister IR* übernommen werden.

Das Instruktionsregister ist direkt mit dem Instruktionsdecoder verbunden, der das ankommende Bitmuster analysiert und aufschlüsselt. Die Ausgänge des Instruktionsdecoders werden in das Steuerwerk geführt und dort in entsprechende Steuersignale für die Befehlsausführung übersetzt. Zur Ausführung des decodierten Befehls muss die CPU die folgenden Steuerungsaufgaben erledigen:

- Ansteuerung des Akkumulators

  In Abhängigkeit des auszuführenden Befehls müssen die Steuerleitungen des Akkumulators so gesetzt werden, dass der Registerinhalt erhalten, überschrieben oder akkumuliert wird. Der Betriebsmodus des Akkumulators wird vom Steuerwerk über die beiden Signalleitungen e und ld eingestellt. Über die dritte Signalleitung sub legt das Steuerwerk fest, ob das Additionswerk die Summe oder die Differenz der angelegten Operanden berechnet.

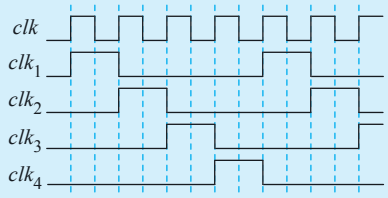
- Ansteuerung des Instruktionszählers

  Neben der Ansteuerung des Rechenwerks muss das Steuerwerk in jedem Takt den Inhalt des Instruktionszählers in Abhängigkeit des auszuführenden Befehls verändern. Das jeweilige Verhalten des Zählers bestimmt das Steuerwerk über die beiden Signalleitungen s_0 und s_1. Eine rein sequenzielle Befehlsausführung wird erreicht, indem der Zähler sukzessive um eins erhöht wird. Direkte und indirekte Sprünge lassen sich ebenfalls einfach abbilden und entsprechen dem Laden bzw. dem Akkumulieren des Instruktionszählers. Das Beschreiben des Registers erfolgt in unserem Modellrechner synchron zur negativen Taktflanke und damit zeitgleich mit dem Beginn der Ausführungsphase. Auf diese Weise ist sichergestellt, dass der nächste auszuführende Befehl zu Beginn des nächsten Taktzyklus bereits auf dem Datenbus liegt.

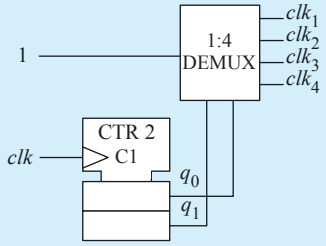
- Durchschalten der korrekten Transportwege

  Der Datenfluss innerhalb der CPU wird über zwei Multiplexer gesteuert. Der erste Multiplexer ist dem Dateneingang des Akkumulators vorgeschaltet. Ist das Steuersignal m_1 gleich 0, so werden die Dateneingänge des Akkumulators direkt mit dem Datenbus verbunden. Wird das Steuersignal m_1 stattdessen auf 1 gesetzt, so bezieht der Akkumulator die Daten direkt aus dem Datenregister. Der zweite Multiplexer steuert die Beschaltung des Adressbusses. Ist das Steuersignal m_2 gleich 0, so wird der Inhalt des Datenregisters auf den

Im direkten Vergleich mit unserem einfach strukturierten Modellprozessor, der jeden Befehl in genau zwei Ausführungsphasen abarbeitet, sind die Steuerwerke realer Prozessoren deutlich komplexer ausgelegt. Insbesondere Prozessoren mit mächtigen Befehlssätzen erfordern die Zerlegung eines Befehls in eine ganze Reihe sequenziell zu durchlaufender Phasen. Zu diesem Zweck enthalten Steuerwerke oft einen sogenannten *Mehrphasentaktgeber* (*sequencer*), der das Taktsignal clk als Eingabe entgegennimmt und auf n Ausgangsleitungen $clk_1, \ldots, clk_n$ verteilt:



Der Mehrphasentaktgeber kann durch die Zusammenschaltung eines Binärzählers und eines Demultiplexers auf einfache Weise implementiert werden:



Die Steuerleitungen des Demultiplexers sind direkt mit den Ausgängen des Zählers verbunden, so dass der auf 1 liegende Eingang nacheinander auf jeden der Ausgänge clk_1 bis clk_n durchgeschaltet wird. Legen wir am Eingang des Demultiplexers den konstanten Wert 1 an, so entsteht an den Ausgängen des Demultiplexers auf einen Schlag der oben abgebildete Signalverlauf.

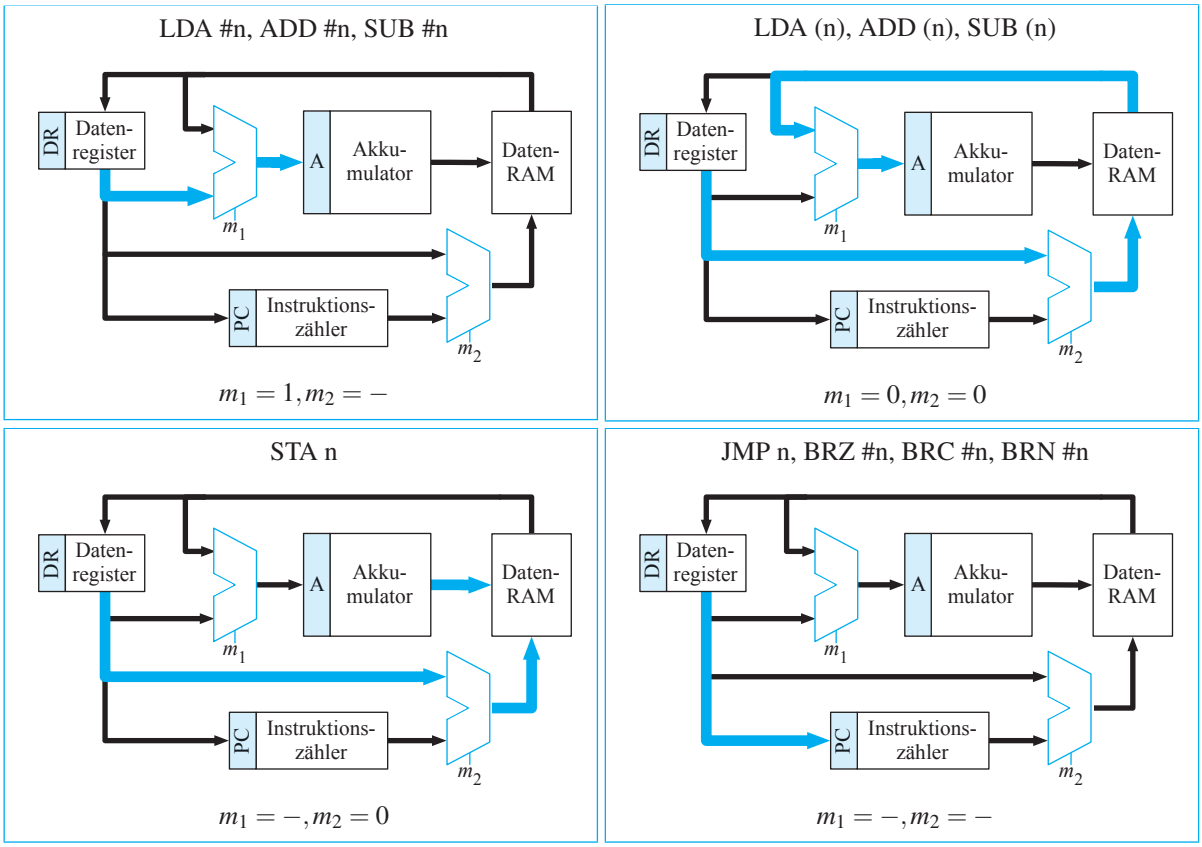


Abbildung 3.12: Datenfluss innerhalb des Modellrechners

Adressbus gelegt. Wird m_2 auf 1 gesetzt, so wird der Speicher über den Instruktionszähler *PC* adressiert.

Abbildung 3.12 zeigt, wie die Transportwege der CPU während der Ausführungsphase beschaltet werden müssen, um die weiter oben eingeführten Lade-, Speicher- und Sprungbefehle in ihren verschiedenen Adressierungsvarianten auszuführen. Von der `NOP`-Operation abgesehen, können wir jeden der 11 verbleibenden Befehle in eines der folgenden vier Datenflussszenarien einordnen:

- `LDA #n`, `ADD #n` oder `SUB #n`

  In allen drei Fällen wird der Akkumulator mit dem Inhalt des Datenregisters gespeist ($m_1 = 1$). Der Inhalt des Datenbusses spielt in

diesem Fall keine Rolle, da er vollständig vom Rechenwerk abgekoppelt ist. Folgerichtig ist auch der Inhalt des Adressbusses irrelevant und m_2 kann mit einem beliebigen Wert beschaltet werden.

- `LDA (n)`, `ADD (n)` oder `SUB (n)`

  Im Falle der indirekten Adressierung wird n nicht selbst in den Akkumulator geleitet, sondern zur Adressierung des Speichers verwendet. Hierzu legen wir den Inhalt des Datenregisters auf den Adressbus ($m_2 = 0$) und leiten den Inhalt des Datenbusses direkt in den Akkumulator weiter ($m_1 = 0$).

- `STA n`

  Zur Übertragung des Akkumulatorinhalts in den Speicher wird zunächst der Inhalt des Datenregisters auf den Adressbus gelegt ($m_2 = 0$) und unmittelbar darauf das Schreibsignal *we* aktiviert. Die Beschaltung von m_1 spielt in diesem Fall keine Rolle, da der Inhalt des Akkumulatorregisters während der Befehlsausführung nicht verändert wird.

- `JMP n`, `BRZ #n`, `BRC #n`, `BRN #n`

  Zur Ausführung eines Sprungbefehls wird lediglich der Stand des Instruktionszählers verändert – das Rechenwerk bleibt in diesem Fall vollständig inaktiv. Zur Durchführung eines direkten Sprungs wird der Befehlszähler mit dem Inhalt des Datenregisters überschrieben, zur Durchführung eines indirekten Sprungs wird der Wert auf den aktuellen Stand addiert. Da sowohl das Rechenwerk als auch der Speicher bei der Ausführung eines Sprungbefehls außen vor gelassen werden, spielt die Beschaltung von m_1 wie auch von m_2 keine Rolle.

Um ein praktisches Verständnis für den Ablauf eines Maschinenprogramms zu erhalten, wollen wir unser bisher erworbenes Wissen anhand eines einfachen Beispielprogramms konkretisieren. Hierzu betrachten wir das in Abbildung 3.13 dargestellte Assembler-Programm zur Multiplikation zweier Zahlen. Werden Multiplikand und Multiplikator vor der Ausführung des Programms in die Speicherzellen 13 und 14 geschrieben, so enthält die Speicherzelle 15 nach der Ausführung das Produkt $(13) \times (14)$. Da unser Modellprozessor über kein Multiplikationswerk verfügt und damit auch keinen Multiplikationsbefehl kennt, wird die Multiplikation durch die wiederholte Ausführung der Addition berechnet. Hierzu implementiert das Programm eine Schleife, die in jeder Iteration den Multiplikator (Speicherzelle 13) um eins erniedrigt und den Inhalt der initial mit null beschriebenen Speicherzelle 15 um den

```
// Beispielprogramm
// zur Berechnung der
// Multiplikation
//
// Eingabe:
// (13) = Multiplikator
// (14) = Multiplikand
//
// Ausgabe:
// (15) = (13) * (14)

0: init:  LDA #0
1: begin: STA (15)
2:        LDA (13)
3:        BRZ #6   // end:
4:        SUB #1
5:        STA (13)
6:        LDA (15)
7:        ADD (14)
8:        JMP 1   // begin:
9: end:   END
```

Abbildung 3.13: Ein Maschinenprogramm zur Ausführung der Multiplikation in der Sprache unserer Modell-CPU

Wert des Multiplikanden erhöht. Die Schleife wird beendet, sobald der Inhalt von Speicherzelle 13 auf null heruntergezählt wurde. In diesem Fall enthält die Speicherzelle 15 bereits das gesuchte Produkt und das Programm wird beendet.

Als letzten Befehl enthält das abgebildete Programm den *Makro-Befehl* `END`. Hinter einem Makro verbirgt sich kein Befehl im eigentlichen Sinne. Stattdessen handelt es sich um eine rein syntaktische Abkürzung, d. h., jedes Makro steht stellvertretend für ein oder mehrere Maschinenbefehle und dient lediglich zur Schreiberleichterung. In unserem Beispielprogramm ist das Makro

```
<n> : END
```

als

```
<n> : JMP <n>
```

definiert. Der Sinn und Zweck dieses Befehls ist es, die CPU am Ende eines Programms in eine Endlosschleife zu versetzen und damit die sequenzielle Ausführung der CPU effektiv zu stoppen.

Tabelle 3.2 demonstriert, wie das Beispielprogramm Schritt für Schritt in unserer Modell-CPU abgearbeitet wird. Für jede Phase der Befehlsausführung enthält die Tabelle die aktuelle Belegung der internen Prozessor-Register, des Daten- und Adressbusses sowie die Inhalte der relevanten Speicherzellen. In dem abgebildeten Beispiel sind die Speicherzellen 13 und 14 initial mit den Werten 2 und 3 befüllt. Nach 2 Schleifeniterationen steht der Ergebniswert $2 \times 3 = 6$ in Speicherzelle 15 und das Programm terminiert.

Mit unserem erworbenen Wissen über die interne Befehlsausführung unseres Modellprozessors sind wir in der Lage, uns der Detailimplementierung der verbleibenden Komponenten zuzuwenden. Abbildung 3.14 stellt die Hardware-Umsetzung des Instruktionsdecoders dar. Das zweistufig ausgelegte Schaltnetz enthält vier Eingangsleitungen $i_3, \ldots, i_0$, die direkt mit dem Datenbus der Modell-CPU verbunden sind. Das Bit-Muster der anliegenden Maschineninstruktion wird mit Hilfe der kombinatorischen Logik analysiert und im Sinne einer One-Hot-Codierung eine der Ausgangsleitungen aktiviert. Wie das Strukturbild zeigt, verfügt der Decoder für die Befehle `LDA`, `ADD` und `SUB` jeweils nur über eine Ausgangsleitung. Um die beiden zur Verfügung stehenden Adressierungsarten dieser Befehle zu unterscheiden, wird im Falle der indirekten Adressierung die zusätzlich vorhandene Ausgangsleitung *ind* auf 1 gesetzt.

Zyklus	*clk*	PC	Adressbus	Datenbus	(IR)	(DR)	(A)	(13)	(14)	(15)
01	↑	00	01	LDA #0	LDA	00	??	02	03	??
	↓	01	??	??	LDA	00	00	02	03	??
02	↑	01	02	STA (15)	STA	15	00	02	03	??
	↓	02	15	0	STA	15	00	02	03	00
03	↑	02	03	LDA (13)	LDA	13	00	02	03	00
	↓	03	13	02	LDA	13	02	02	03	00
04	↑	03	04	BRZ 06	BRZ	06	02	02	03	00
	↓	04	??	??	BRZ	06	02	02	03	00
05	↑	04	05	SUB #1	SUB	01	02	02	03	00
	↓	05	??	??	SUB	01	01	02	03	00
06	↑	05	06	STA (13)	STA	13	01	02	03	00
	↓	06	13	01	STA	13	01	01	03	00
07	↑	06	07	LDA (15)	LDA	15	01	01	03	00
	↓	07	15	00	LDA	15	00	01	03	00
08	↑	07	08	ADD (14)	ADD	14	00	01	03	00
	↓	08	14	03	ADD	14	03	01	03	00
09	↑	08	09	JMP 01	JMP	01	03	01	03	00
	↓	01	??	??	JMP	01	03	01	03	00
10	↑	01	02	STA (15)	STA	15	03	01	03	00
	↓	02	15	03	STA	15	03	01	03	03
11	↑	02	03	LDA (13)	LDA	13	03	01	03	03
	↓	03	13	01	LDA	13	01	01	03	03
12	↑	03	04	BRZ 06	BRZ	06	01	01	03	03
	↓	04	??	??	BRZ	06	01	01	03	03
13	↑	04	05	SUB #1	SUB	01	01	01	03	03
	↓	05	??	??	SUB	01	00	01	03	03
14	↑	05	06	STA (13)	STA	13	00	01	03	03
	↓	06	13	00	STA	13	00	00	03	03
15	↑	06	07	LDA (15)	LDA	15	00	00	03	03
	↓	07	15	03	LDA	15	03	00	03	03
16	↑	07	08	ADD (14)	ADD	14	03	00	03	03
	↓	08	14	03	ADD	14	06	00	03	03
17	↑	08	09	JMP 01	JMP	01	06	00	03	03
	↓	01	??	??	JMP	01	06	00	03	03
18	↑	01	02	STA (15)	STA	15	06	00	03	03
	↓	02	15	06	STA	15	00	00	03	06
19	↑	02	03	LDA (13)	LDA	13	00	00	03	06
	↓	03	13	00	LDA	13	00	00	03	06
20	↑	03	04	BRZ 06	BRZ	06	00	00	03	06
	↓	09	??	??	BRZ	06	00	00	03	06
21	↑	09	04	END	JMP	09	00	00	03	06
	↓	09	??	??	JMP	09	00	00	03	06

Tabelle 3.2: Ablaufprotokoll für die Berechnung von 2×3

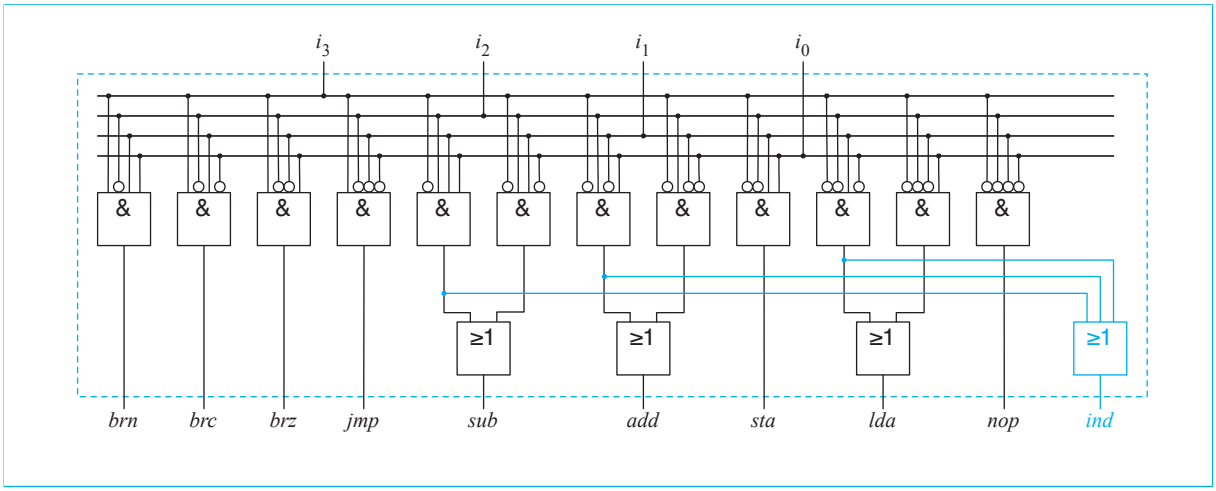


Abbildung 3.14: Implementierung des Instruktionsdecoders als zweistufiges Schaltnetz

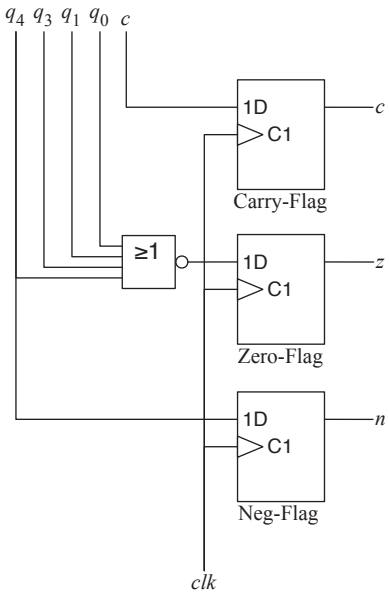


Abbildung 3.15: Das Statusregister unseres Modellrechners

Als Nächstes wenden wir uns der Konstruktion des Statusregisters zu. Wie die Implementierung in Abbildung 3.15 zeigt, hält das Register für jedes Statusbit ein D-Flipflop vor, dessen Wert sich aus dem aktuellen Inhalt des Akkumulatorregisters berechnet. Das Statusregister wird über das Steuerwerk getaktet und ist negativ flankengesteuert ausgelegt. Damit ist sichergestellt, dass die Statusbits stets in der Mitte einer Taktperiode beschrieben werden. In anderen Worten: Die Statusbits werden zeitgleich mit dem Beginn der Decodierungsphase gesetzt und geben damit über das Ergebnis der letzten durchgeführten arithmetischen Operation Auskunft. Ausgewertet werden die Statusbits im Steuerwerk zur Durchführung bedingter Sprünge. So bewirken die Befehle `BRC`, `BRZ` und `BRN` nur dann einen Sprung, wenn das entsprechende Statusbit den Wert 1 besitzt. Insgesamt kommt dem Statusregister damit die Rolle eines Bindeglieds zwischen Rechen- und Steuerwerk zu. Ohne dieses Register wäre kein Informationsfluss zwischen beiden Komponenten mehr vorhanden und damit keine komplexe Kontrollflusssteuerung mehr möglich.

Keinerlei Probleme bereitet uns der Aufbau des Akkumulatorregisters sowie des Instruktionszählers. Für beide Komponenten haben wir bereits in Kapitel 2 eine Implementierung kennen gelernt, die wir für den Einsatz in unserer Modell-CPU nahezu vollständig übernehmen können. Einzig das Akkumulatorregister erfährt eine marginale Änderung. Damit ein im Addierer etwaig entstehender Übertrag auch noch im nächsten Takt vom Steuerwerk ausgewertet werden kann, speichern wir

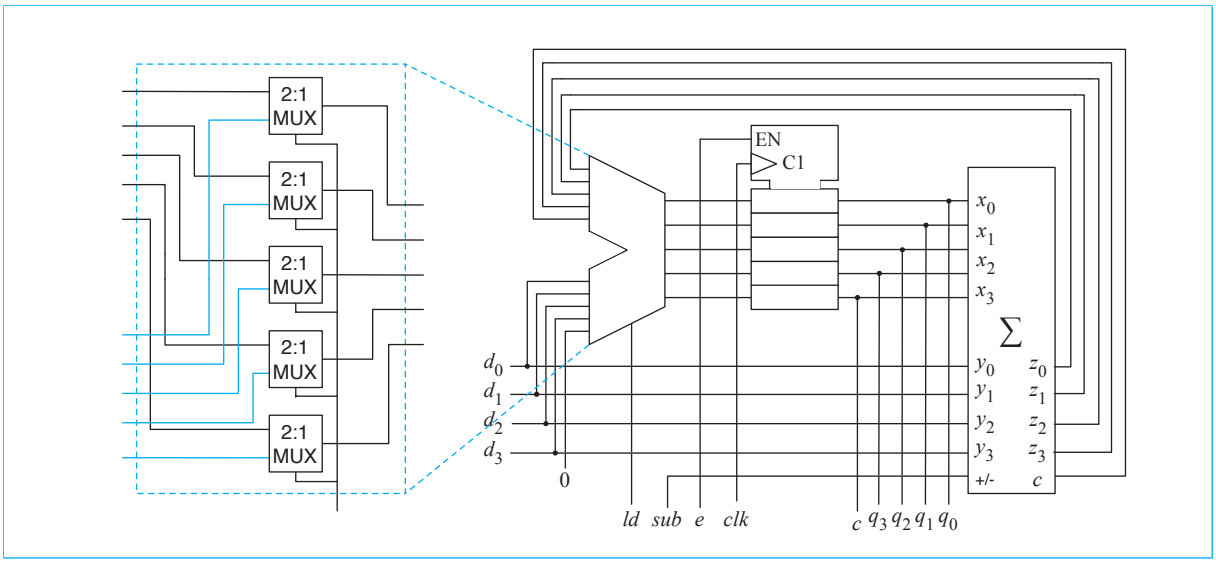


Abbildung 3.16: Das Akkumulatorregister unseres Modellrechners

dessen Wert zusammen mit den Datenbits im Auffangregister zwischen. Das modifizierte Strukturbild des Akkumulators ist in Abbildung 3.16 dargestellt. Die Beschaltung der Steuervariablen zur Einstellung der jeweils benötigten Betriebsmodi können wir ohne Umschweife aus Tabelle 2.1 sowie Abbildung 2.23 entnehmen.

Der Hauptspeicher wird in Form eines SRAM-Speichers implementiert. Um zu vermeiden, dass im Falle des Beschreibens aufgrund unterschiedlicher Signallaufzeiten die falsche Speicherzelle adressiert wird, verwenden wir einen Speicher, der im Lesemodus ($we = 0$) asynchron, im Schreibmodus ($we = 1$) dagegen synchron arbeitet. Eine Speicherzelle wird genau dann beschrieben, wenn das Write-Enable-Signal we während der negativen Taktflanke gleich 1 ist. Im asynchronen Lesebetrieb bleibt der Wert des Taktsignals dagegen ohne Bedeutung.

Auf den in Kapitel 2 geleisteten Vorarbeiten aufbauend können wir für jeden Befehl die Beschaltung der Steuerleitungen für Akkumulator, Rechenwerk, RAM sowie die beiden Multiplexer ableiten. Das Ergebnis ist in Tabelle 3.3 zusammengefasst und wir erhalten für die Ausgangssignale des Steuerwerks die folgenden booleschen Gleichungen:

	Statusvariablen			Akkumulator			PC		RAM	Multiplexer	
Befehl	z	c	n	e	ld	sub	s_1	s_0	we	m_1	m_2
	Ladephase (Fetch): $clk = 0$										
–	–	–	–	0	–	–	1	0	0	–	1
	Ausführungsphase (Decode + Execute + Write): $clk = 1$										
NOP	–	–	–	0	–	–	1	0	0	–	–
LDA	–	–	–	1	1	–	1	0	0	$\neg ind$	0
STA	–	–	–	0	–	–	1	0	1	–	0
ADD	–	–	–	1	0	0	1	0	0	$\neg ind$	0
SUB	–	–	–	1	0	1	1	0	0	$\neg ind$	0
JMP	–	–	–	0	–	–	0	1	0	–	–
BRZ	0	–	–	0	–	–	1	0	0	–	–
BRZ	1	–	–	0	–	–	0	0	0	–	–
BRC	–	0	–	0	–	–	1	0	0	–	–
BRC	–	1	–	0	–	–	0	0	0	–	–
BRN	–	–	0	0	–	–	1	0	0	–	–
BRN	–	–	1	0	–	–	0	0	0	–	–

Tabelle 3.3: Beschaltung von Akkumulator, Instruktionszähler, RAM und Multiplexer durch das Steuerwerk

- Akkumulator

$$\begin{aligned} e &= lda \vee add \vee sub \\ ld &= lda \\ sub &= sub \end{aligned}$$

- Instruktionszähler (PC)

$$\begin{aligned} s_1 &= \overline{(brz \wedge z) \vee (brc \wedge c) \vee (brn \wedge n) \vee jmp} \\ s_0 &= jmp \end{aligned}$$

- RAM

$$we = sta$$

- Multiplexer

$$\begin{aligned} m_1 &= \overline{ind} \\ m_2 &= \overline{clk} \end{aligned}$$

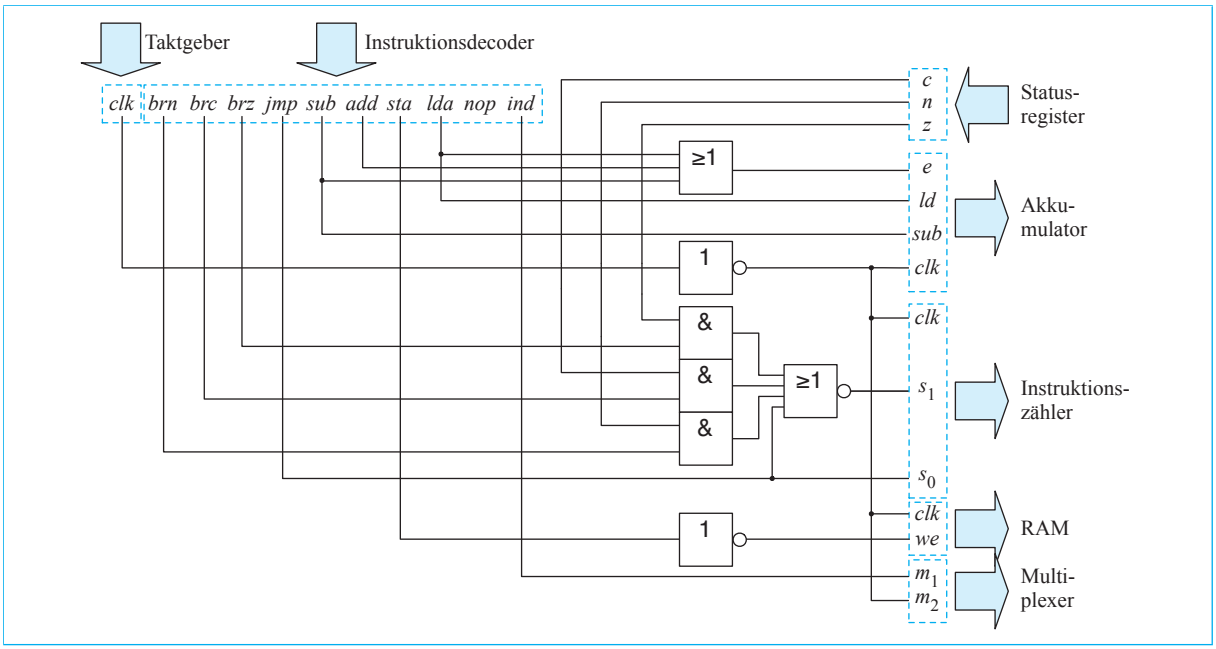


Abbildung 3.17: Vollständig implementiertes Steuerwerk unserer Modell-CPU

Die Beschaltung der Takteingänge der verschiedenen Komponenten ergibt sich direkt aus der Aufteilung eines Taktzyklus in zwei separate Ausführungsphasen. Auf den Speicher wird im Schreibmodus während der negativen Taktflanke zugegriffen. Deshalb wird das Taktsignal *clk* vom Steuerwerk invertiert an den Hauptspeicher weitergereicht. Das Akkumulatorregister und der Instruktionszähler werden am Ende der Ausführungsphase aktiviert. Folgerichtig versorgt das Steuerwerk beide Komponenten ebenfalls mit dem invertierten Taktsignal.

Insgesamt erhalten wir die in Abbildung 3.17 dargestellte Implementierung des Steuerwerks. Wie das Strukturbild zeigt, fällt die Implementierung erstaunlich einfach aus – eine Handvoll Logikgatter reicht aus, um die Kontrollzentrale unserer CPU vollständig zu realisieren. Der Grund hierfür liegt zum einen in dem vergleichsweise primitiven Befehlssatz unseres Mikroprozessors, der neben den atomaren Kontroll- und Datenflussoperationen keinerlei weiter gehende Operationen unterstützt. Zum anderen ist ein Teil der Kontrolllogik in den Befehlsdecoder ausgelagert. Durch die Aufschlüsselung des Instruktionscodes auf für jeden Befehl separate Signalleitungen wird die Implementierung des Steuerwerks deutlich vereinfacht.

3.3 Übungsaufgaben

Aufgabe 3.1

Webcode 0663

Gegeben sei das folgende Assembler-Programm in der Sprache unseres Modellrechners:

```
0000   low: LDA (13)
0001        ADD (15)
0010        BRC ub:
0011        STA 15
0100        LDA #0
0101        JMP high:
0110    ub: STA 15
0111        LDA #1
1000  high: ADD (12)
1001        ADD (14)
1010        STA 14
1011        END
```

a) Welche Funktion erfüllt das Programm? Spielen Sie hierzu den Programmablauf für eine Reihe von Eingabewerten durch.

b) Wie ließe sich unser Modellrechner verbessern, damit er Berechnungen dieser Art deutlich vereinfacht? Werfen Sie hierzu bei Bedarf einen Blick in die Instruktionssätze moderner Mikroprozessoren.

Aufgabe 3.2

Webcode 0121

Gegeben sei das folgende Assembler-Programm in der Sprache unseres Modellrechners:

```
0000        LDA #0
0001        STA 13
0010   sub: LDA (14)
0011        SUB (15)
0100        BRN exit:
0101        STA 14
0110        LDA (13)
0111        ADD #1
1000        STA 13
1001        JMP sub:
1010  exit: END
```

Was berechnet dieses Programm? Spielen Sie hierzu erneut den Programmablauf für eine Reihe von Eingabewerten durch.

Aufgabe 3.3

Webcode 0870

Das folgende C-Programm implementiert den *Euklidischen Algorithmus* zur Berechnung des größten gemeinsamen Teilers (ggT) zweier positiver ganzer Zahlen.

```
int ggt(int x, int y) {
    int r;
    do {
        r = x % y;
        x = y;
        y = r;
    } while (y != 0);
    return x;
}
```

a) Implementieren Sie die Modulo-Operation (%) in der Sprache unseres Modellrechners.

b) Übersetzen Sie die C-Implementierung in ein entsprechendes Assembler-Programm.

c) Entwickeln Sie ein Assembler-Programm zur Berechnung des kleinsten gemeinsamen Vielfachen (kgV) zweier positiver ganzer Zahlen. Greifen Sie hierzu auf die Ergebnisse der vorhergehenden Teilaufgaben zurück.

Aufgabe 3.4

Webcode 0422

Gegeben sei das folgende Programmgerüst in der Sprache unseres Modellrechners:

```
0000        LDA #0011
0001        STA 1111
0010        JMP enter:
0011        ...
...         ...
0110        LDA #1001
0111        STA 1111
1000        JMP enter:
1001        ...
...         ...
1100 enter: ...
...         ...
1111  exit: JMP 0000
```

a) Welches bekannte Programmierkonzept wird in diesem Programm umgesetzt?

b) Wie könnte der Modellrechner erweitert werden, um Programme dieser Art besser zu unterstützen?

Aufgabe 3.5

Webcode 0987

Auf Seite 85 ist die Implementierung eines Mehrphasentaktgebers abgebildet. Das gewünschte Verhalten wird durch die Zusammenschaltung eines Binärzählers und eines Demultiplexers erreicht.

a) Welcher Ihnen bekannte Zahlencode wird durch den Mehrphasentaktgeber erzeugt?

b) Reimplementieren Sie den Taktgeber mit Hilfe eines Schieberegisters.

Aufgabe 3.6

Webcode 0558

Betrachten Sie erneut die in Abbildung 3.14 dargestellte Implementierung des Instruktionsdecoders unseres Modellrechners. Statt den Decoder als zweistufiges Schaltnetz zu implementieren, können die Befehle auch mit Hilfe eines ROM-Speichers decodiert werden. Programmieren Sie hierzu den folgenden ROM-Baustein, indem Sie die entsprechenden Verbindungen in die ODER-Matrix eintragen: