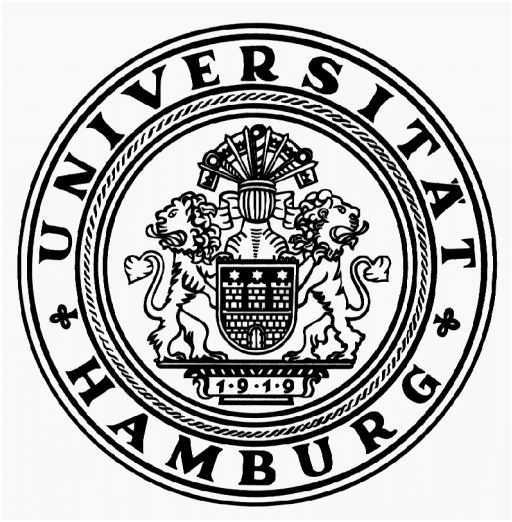
Andreas Ruge

Studienarbeit



Rekonstruktion und Visualisierung von Systemen mit Intel 4004 Prozessor

Fachbereich Informatik
Arbeitsbereich TAMS

August 2003

Inhaltsverzeichnis

1 Einleitung

Diese Arbeit behandelt die Rekonstruktion und Visualisierung der Micro Computer Set 4 (MCS 4) -Systeme von Intel, denen eine historische Bedeutung im Entwicklungsprozess der Informatik zuteil wurde.
Das Kernstück dieser Systeme ist der Intel 4004, der 1971 vorgestellte erste und in größerer Stückzahl (200 US$ pro Stück) produzierte Mikroprozessor der Welt.

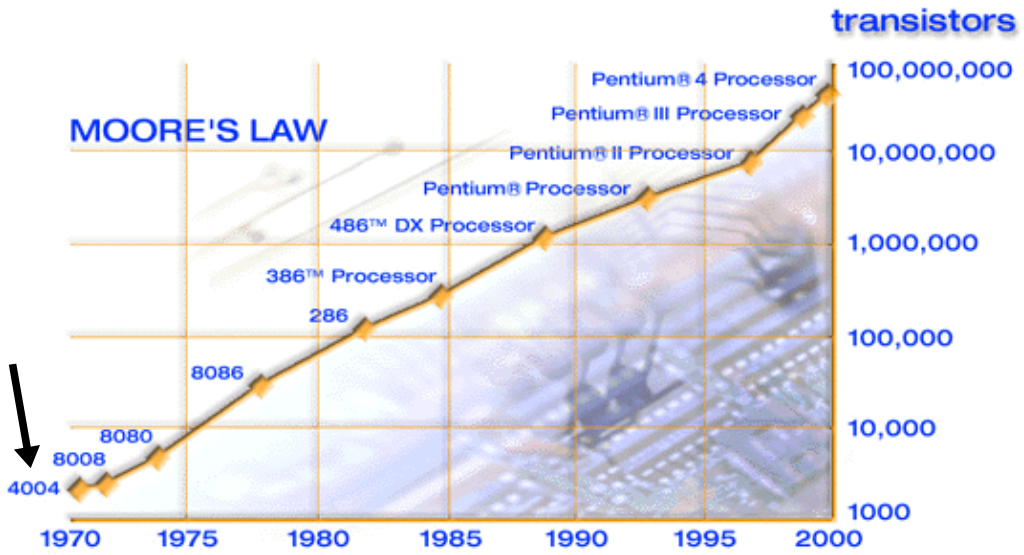


Abbildung 1.1: Das Moore'sche Gesetzt in den letzten 30 Jahren

Er bildet die Grundlage für die weiteren Prozessorfamilien des Halbleiterherstellers Intel, der vor allem durch die Prozessoren Bekanntheit erlangt hat. Aus diesem Grund widmet sich **Kapitel 2** der Geschichte um die Entstehung dieses Prozessors. Zu einem MCS 4-System gehört aber nicht nur der Prozessor, sondern auch noch andere Komponenten, deren Zusammenspiel solch ein System ausmacht. Diese Komponenten und deren Beziehungen behandelt **Kapitel 3**. Dort wird auch der Befehlssatz und dessen Ausführung erläutert. Um den Einsatz dieser Systeme beispielhaft zu demonstrieren, werden in **Kapitel 4** Anwendungsbeispiele vorgestellt. Um diese Beispiele praktisch nachzuvollziehen, wurden die beschriebenen Komponenten implementiert um sie im HADES-Simulations-Framework [Hades], das an der Universität Hamburg entwickelt wird, einsetzen zu können. Die Details der Implementierung behandelt das **5. Kapitel**. Da die

HADES-Designs auch als Java-Applets benutzt werden können, werden die Beispiele des MCS 4-Systems in Browsern oder anderen Applet-Viewern lauffähig.

Auf diese Weise soll den wissenschaftlich Interessierten eine weitere Dokumentation in die Hand gegeben werden, da die bestehenden Papiere aus den 70er Jahren stammen und daher kaum in digitalisierter Form vorliegen und nur noch schwer zu beschaffen sind. Aber die MCS 4-Systeme sollten nicht in Vergessenheit geraten.

2 Historie

Die Geschichte, die zu der Entwicklung des 4004 und der weiteren Komponenten eines MCS 4-Systems führt und somit den Grundstein für den Erfolg und die große Marktführerschaft der Firma Intel Corp. legt, liest sich wie ein modernes Märchen:

Es war einmal im April 1969, als sich eine Firma aus dem fernen Land Japan aufmachte, um einen Partner für die Herstellung von Chips zu suchen, die sie für die Produktion ihrer Rechenmaschinen benötigte. Diese Firma hieß **Busicom** und der Vertrag wurde mit der **Intel Corp.** geschlossen [Bus], die ihren Sitz in Santa Clara hatte, einer Stadt im Staate Kalifornien in Amerika. Intel hatte sich nach der Gründung durch Bob Noyce und Gordon Moore (dem Begründer des Moore'schen Gesetzes) im Jahre 1968 darauf spezialisiert, Chips zu entwickeln (hauptsächlich zur Datenspeicherung), die von mehreren Kunden sofort eingesetzt werden konnten, und nicht speziell für einzelne Kunden angefertigt wurden.

Um die Entwicklung der neuen Chips zu unterstützen, wurden von Busicom drei Ingenieure entsandt, unter ihnen **Masatoshi Shima**. Auf der Seite von Intel wurde für diese Zusammenarbeit **Marcian E. Hoff jr.** abgestellt, der die Verbindung zwischen der Entwicklungsabteilung von Intel und deren Kunden herstellt und ihnen auch beim Einsatz der Produkte zur Seite stand.

Bei der Überprüfung des Designs, das Busicom sich vorstellte, merkte Hoff schnell, dass die Komplexität – sieben verschiedene Chips und deren Packages nur für Busicoms Rechner – die Ressourcen von Intel zu stark binden würde, und bekam den Auftrag, eine Alternative zu entwickeln.

Da die Chips auch in unterschiedlichen Rechenmaschinen eingesetzt werden sollten, hatte das ursprüngliche Design die Fähigkeit, Instruktionen aus einem ROM zu lesen, zu interpretieren und so verschiedene Verarbeitungen auszulösen. Als beschreibbarer Speicher waren damals Schieberegister vorgesehen, da sie kleiner als statische RAMs (SRAMs) waren, was aber den zufälligen Zugriff auf die Daten erschwerte. Intel allerdings hatte zu der Zeit mit der Entwicklung von

dynamischen RAMs (DRAMs) begonnen, die den zufälligen Zugriff mit geringem Platzbedarf kombinierten, allerdings einen Refresh benötigten.

Diese Ideen griff Hoff, auf um einen Chip zu entwickeln, der eine arithmetische Einheit besitzen, Befehle aus einem ROM lesen, diese interpretieren und Lese- und Schreiboperationen auf DRAMs ausführen sollte. Der Refresh der DRAMs wird in der Befehlsholphase vorgenommen. Zusätzlich wurde ein Register-Array („scratch-pad“) in diesen Chip integriert. Dieser Chip wäre zudem in vielen Bereichen universell einsetzbar.

Unterstützung bei der Entwicklung bekam Hoff von **Stanley Mazor** und Shima. Der Befehlssatz wurde erweitert, um Subroutinen zu behandeln, bedingte Sprünge auszuführen und Befehle indirekt zu laden. In Hinsicht auf BCD-Zahlen wurde ein 4-bittiger bidirektionaler Datenbus vorgesehen, mit dem die Chips verbunden wurden. Das führte zu den 8 Subzyklen: 3 für das Anlegen einer 12-Bit-Adresse, 2 für das Holen der 8-bittigen Instruktionen und 3 für das Ausführen einer Instruktion.

Als dieses Design dann Busicom vorgestellt wurde, bekam das Konzept von Intel aufgrund seiner Flexibilität und Einfachheit den Zuschlag für die Realisierung. So fiel im Oktober 1969 der Startschuss für die Herstellung des ersten Mikroprozessors (CPU), der auf einem einzelnen Chip integriert wurde.

Das Design der Chips stand nun fest: Die CPU (4004) wurde durch ROMs (4001), RAMs (4002) und I/O-unterstützende Schieberegister (4003) ergänzt. Jetzt bekam Intel allerdings Schwierigkeiten, weil der Prozessor für damalige Verhältnisse recht komplex geraten war und für das Layout der Chips keine geeigneten Mitarbeiter zur Verfügung standen, weil sie entweder in anderen Projekten eingesetzt waren oder ihre Fähigkeiten nicht ausreichten. Deshalb wurde im April 1970 **Federico Faggin** eingestellt, der schon an der Entwicklung des MOS-Prozesses beteiligt war.

Faggin traf die Entscheidung, zuerst den 4001, dann 4003, 4002 und zuletzt den 4004 zu designen. Da das Projekt nun aber schon unter Verzögerung litt, arbeiteten Faggin und Shima parallel an allen 4 Chips fast rund um die Uhr und im Oktober 1970 liefen die ersten Wafer des 4001 vom Band. Die Funktionalität

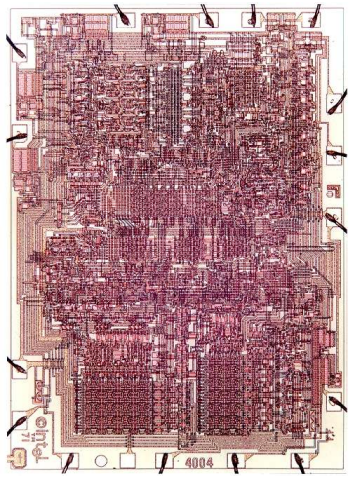


Abbildung 2.1: Das Layout des 4004

dieser Prototypen erwies sich zudem schon sofort als fehlerfrei. Shima kehrte nun nach Japan zurück, um die Software für die Systeme fertig zu stellen und die Änderungen an den Rechenmaschinen von Busicom für den Einsatz der Chips vorzubereiten. Im November desselben Jahres waren auch schon Chips des 4002 und 4003 verfügbar, wobei nur der 4002 einen kleinen Fehler aufwies, der schnell behoben werden konnte.

Die ersten Wafer des 4004 liefen dann im Dezember vom Band, neun Monate nachdem Faggin die Arbeit an dem Projekt bei Intel aufgenommen hatte. Der Prozessor bestand aus 2250 Transistoren und die Strukturbreite war 10µm im p-MOS-Prozess (Zum Vergleich: Der Intel Pentium IV hat 42 Mio. Transistoren bei 0,13µm in BICMOS-Technik). Diese erste Version des Prozessors war aber nicht lauffähig, da ein Schritt bei der Maskenerstellung ausgelassen worden war. Die korrigierte Version wurde dann im Januar 1971 hergestellt und nun lief der Prozessor mit zwei kleinen Fehlern, die zu beseitigen kein großes Problem für Faggin darstellte. Die fehlerfreie Version des 4004 wurde deshalb erst im März 1971 produziert.


Abbildung 2.2: Der erste 4004

Nun wurden die Chips der 4000-Familie nach Japan verschickt, wo Shima sie in den ersten Rechner von Busicom einsetzte. Dieser Rechner arbeitete mit einem 4004, zwei 4002, drei 4003, und vier 4001, so dass das System 1 KByte ROM und 80 Byte RAM als Speicher besaß, und es konnten 100.000 Instruktionen pro Sekunden durchgeführt werden,. Zusätzlich kam noch ein weiterer 4001 für die Wurzelberechnung zum Einsatz. Im April kam dann die Rückmeldung von Busicom, dass der Rechner fehlerfrei seinen Dienst versah, was der Beweis dafür

war, dass das komplette Design und die Zusammenarbeit aller Chips erfolgreich umgesetzt worden war. Nun wurden die finalen ROM-Programme von Shima bei Intel in Auftrag gegeben und im Juni begann die Massenproduktion.

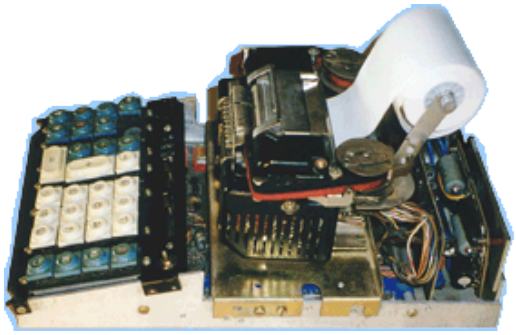


Abbildung 2.3: Der erste Prototyp des Busicom-Rechners

Der einzige Punkt, der den Erfolg der 4000er Serie auf dem Markt jetzt noch verhindert hätte, war die Klausel in dem Vertrag von Busicom und Intel, die Busicom die Exclusivrechte an diesen Chips sicherte. In Gesprächen mit Shima im Frühjahr 1971 wurde aber deutlich, dass Busicom in finanziellen Nöten steckte, weil Faggin darum gebeten wurde, die Produktionskosten der Chips zu senken um wettbewerbsfähig zu bleiben. Da zu diesem Zeitpunkt das Potential in dieser Chip-Familie schon erkennbar und ihr Einsatzzweck auch außerhalb von Rechenmaschinen möglich war, erkaufte sich die Firma Intel die Nutzungsrechte für 60.000$ im Mai 1971 zurück.

Die Chip-Familie bekam den Namen MCS 4 und wurde von nun an vermarktet. Die Erkenntnisse in der Herstellung des 4004 flossen in den i8008 ein, Intels erstem 8-Bit-Prozessor, dessen eigenständige Entwicklung zwischenzeitlich gestoppt worden war, und der als Nachfolger des 4004 im März 1972 in die Produktion ging und so die erfolgreiche Geschichte fortsetzte.

3 Die Architektur der MCS 4-Systeme

Ein MCS 4-System besteht aus einem Prozessor (4004), 1-16 ROMs (4001), und 0-16 RAMs (4002).

Alle diese Komponenten eines MCS 4-Systems sind über einen gemeinsamen Datenbus miteinander verbunden. Da es sich um ein reines 4-bit System handelt ist dieser Bus 4-bit breit und auch die Recheneinheit des Prozessors rechnet 4-bittig. Über diesen Bus werden sowohl Adressen vom Prozessor angelegt als auch Daten in bidirektionaler Richtung gesendet.

Zusätzlich sendet der Prozessor weitere Informationen über diverse Steuerleitungen an die anderen Komponenten. Das SYNC-Signal dient zur Synchronisation, und so genannte CM (Command)-Signale zeigen den angeschlossenen Chips, dass sie für die aktuelle Verarbeitung selektiert und aktiviert werden. Und zum Löschen und Zurücksetzen von Inhalten gibt es bei den Komponenten CL (clear)- und RESET-Signale.

Die Schnittstelle des Systems bilden die I/O-Ports der ROMs und die Output-Ports der RAMs, zusätzlich können an diesen Ports noch Schieberegister (4003) angeschlossen werden.

Die Funktionsweise eines solchen Systems war neuartig: Es gibt keine Chips, die eine spezielle Funktion verrichten, sondern nur einen universellen Prozessor, der in der Lage ist beliebige Programme auszuführen. Ein Programm ist eine Abfolge von Befehlen, die in binär kodierter Form vorliegen.

3.1 Die Befehlsausführung

In einem MCS 4-System wird die Befehlsausführung in Befehlszyklen durchgeführt. Der Befehlszyklus hat folgenden Aufbau:

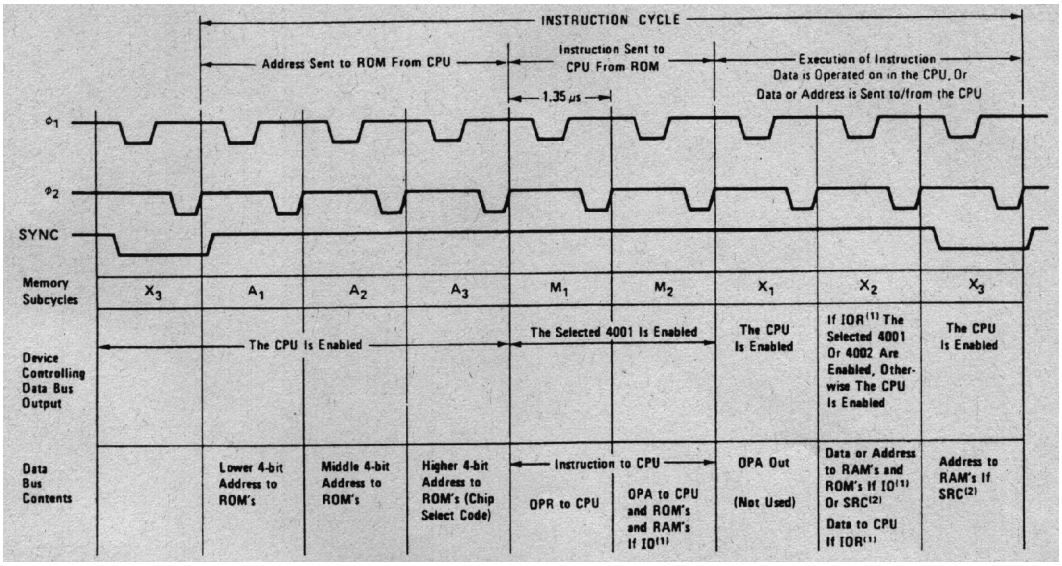


Abbildung 3.1: Der Befehlszyklus des 4004

Somit wird ein Befehlszyklus in 8 Phasen (Subzyklen) unterteilt. Diese Phasen lassen sich in drei Gruppen einordnen:

- Adressierung A_1,A_2,A_3
- Befehlsholphase M_1,M_2
- Befehlsausführung X_1,X_2,X_3

Jeder Subzyklus ist 1,35 µsec lang und dadurch hat jeder Befehlszyklus eine Gesamtlänge von 10,8 µsec.

Während der **Adressierung** wird die 12-bittige Adresse in den drei Phasen jeweils zu 4 Bit von dem Prozessor an die angeschlossenen Speicherchips gesendet. Nur die ROMs verarbeiten diese Daten. Die höheren 4 Bits, die in der Phase A_3 anliegen, geben die Chip-Nummer an, und die niederen 8 Bits (A_1 und A_2) die Adresse innerhalb des gewünschten ROMs. In A_3 wird dann das CM-ROM-Signal auf low gesetzt

In der **Befehlsholphase** werden die adressierten 8 Bit in zwei Phasen (M_1 und M_2) zu je 4 Bit vom ROM auf den Datenbus gelegt. Der Prozessor empfängt diese Daten und legt sie im Instruktionsregister ab. Diese zwei Bestandteile einer Instruktion werden OPR- und OPA-Code genannt. Wenn in M_1 der OPR-Code einen I/O- bzw. RAM-Befehl ankündigt (OPR = 0xe), dann werden in M_2 sowohl das CM-ROM- als auch die CM-RAM_x-Signale entsprechend dem Command-Line Register auf low gesetzt und deshalb wird der OPA-Code, der den genauen I/O- bzw. RAM-Befehl angibt, für die weitere Ausführung in den betroffenen Speicherchips abgelegt. In dieser Phase wird der Inhalt des Programmzählers, der die Adresse des zu holenden Befehls enthält, um 1 erhöht.

Die **Befehlsausführung** der Befehle geschieht während der Phasen X_1 – X_3. Falls ein I/O- bzw. RAM-Befehl zur Ausführung kommt, werden in X_2 entweder die 4 Bits von den betroffenen (gemäß SRC) Speicherchips gesendet und vom Prozessor gelesen (Read-Befehl) oder in umgekehrter Richtung verarbeitet (Write-Befehl). Wenn es sich um einen SRC-Befehl handelt, wird in der Phase X_2 wieder die CM-ROM- und CM-RAM_x-Signale (ebenfalls nach den Bits im Command-Line Register) gesetzt und die von SRC referenzierten 8 Bits werden in X_2 und X_3 auf den Datenbus gelegt und von allen Speicherchips ausgewertet.
In der Phase X_3 wird dann vom Prozessor ein SYNC-Signal an alle angeschlossenen Speicherchips gesendet und damit das Ende des Befehlszyklus mitgeteilt. So soll verhindert werden, dass sich die Phasen der einzelnen Komponenten nicht nach einiger Zeit überlappen und ein Versatz entsteht.

3.2 Die Komponenten

In einem MCS 4-System werden sämtliche Chips (außer 4003) mittels zwei **Taktsignalen** (φ_1 und φ_2) über Statusänderungen informiert. Die wichtigste Änderung ist eine steigende Taktflanke im Signal von φ_2 – sie bedeutet, dass ein Zykluswechsel stattfindet. Das Taktsignal φ_1 ist dazu zeitversetzt und dient zur inneren Steuerung.

An alle Komponenten eines MCS 4-Systems werden auf zwei Leitungen **Versorgungsspannungen** angelegt. V_{DD} ist die Hauptversorgung und liegt um 15V ±5% niedriger als V_{SS}.

3.2.1 4001

Der 4001 ist der ROM-Chip eines MCS 4-Systems und enthält die Programmbefehle, die abgearbeitet werden.

Der Chip findet Anschluss über folgende 16 Pins:

- $D_0 – D_3$ Der Datenbus
- $\varphi_1 + \varphi_2$ Eingang der beiden Taktsignale
- $V_{SS} + V_{DD}$ Die Versorgungsspannungen
- SYNC Das SYNC-Signal
- $I/O_0 – I/O_3$ Die I/O-Leitungen
- CM Das CM-ROM-Signal
- CL Das Clear-Signal
- RESET Das Reset-Signal

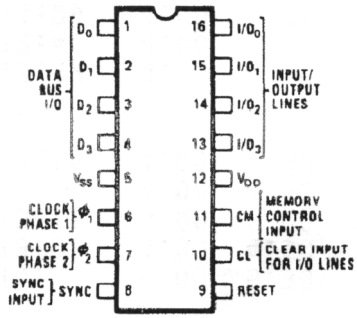


Abbildung 3.2: Pinbelegung des 4001

Der innere Aufbau des 4001 sieht folgendermaßen aus:

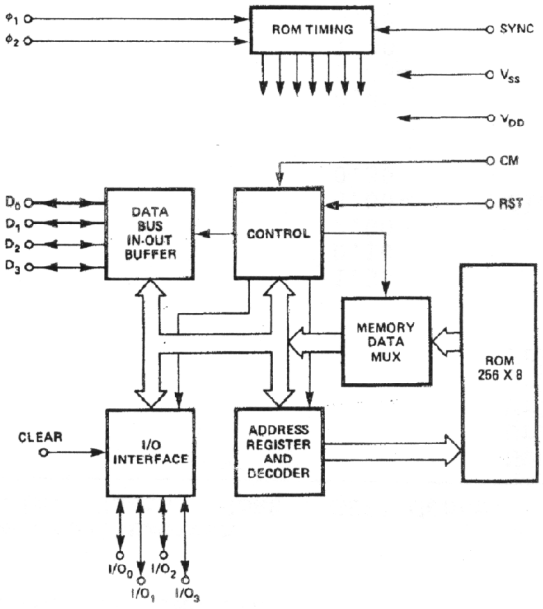


Abbildung 3.3: Schema des 4001

Das Clear-Signal löscht nur den Puffer für die I/O-Ports, der sich im I/O-Interface befindet, das Reset-Signal setzt die restlichen Register des Chips zurück, hat aber keinen Effekt auf den I/O-Puffer.

Die Hauptaufgabe des 4001 ist die Versorgung des Prozessors mit Programmbefehlen. Dazu stehen in einem MCS 4-System bis zu 16 ROMs mit einem Programmspeicher von jeweils 256 x 8 Bit zur Verfügung. Die Nummer des Chips (0-15) wird bei der Herstellung zusammen mit dem Programm hart verdrahtet. In Phase A_3 wird die gewünschte Chip-Nummer übertragen und in A_1 und A_2 die Adresse in dem betroffenen Chip. Der Befehl wird aus dem internen ROM ausgelesen und in zwei 4-bittigen Paketen wieder zurückgeschickt.

Die zweite Aufgabe besteht zur Steuerung von I/O-Operationen. Zu diesem Zweck werden die Daten vom Datenbus an die I/O-Ports weitergeleitet und umgekehrt, je nachdem welcher Befehl ausgeführt werden soll. Die I/O-Ports können entweder zur Ein- oder Ausgabe genutzt werden. Dazu wird die Konfiguration der 4 Ports bei der Bestellung des Chips mit angegeben und ebenfalls hart verdrahtet. Um einen ROM-Chip für eine I/O-Operation auszuwählen wird der **SRC-Befehl** eingesetzt. Bei der Ausführung dieses Befehls werden in X_2 und X_3 jeweils 4 Bits auf den Datenbus geschickt. Durch das Setzen von CM-ROM liest der 4001 die 4 Bits in X_2 und interpretiert sie als Chip-Nummer. Die Bits, die in X_3 ankommen, tragen nur für den 4002 Informationen und werden ignoriert. Für die I/O-Operationen gibt es zwei Befehle: **WRR** schreibt die Daten auf die Output-Ports. Diese Daten bleiben solange an den Ports anliegend bis ein erneuter WRR-Befehl zur Ausführung kommt. Da jedem Port genau ein Bit auf dem Datenbus entspricht, bleibt das Bit bei einem Input-Port ohne Wirkung. Der **RDR-Befehl** dient dazu Daten von den Input-Ports einzulesen. An der Position des Wortes, an der Ouput-Ports die Daten liefern würden, kann ebenfalls '1' oder '0' bei der Herstellung gewünscht werden (Verbindung mit V_{SS} oder V_{DD}).

Die I/O-Datentransfers auf dem Datenbus zum Prozessor finden in der Phase X_2 statt.

3.2.2 4002

Zur Speicherung von Daten während der Verarbeitung eines Programms können in ein MCS 4-System RAM-Chips integriert werden, die 4002.

Das Anschlussbild sieht ähnlich wie beim 4001 aus:

- $D_0 – D_3$ Der Datenbus
- $\varphi_1 + \varphi_2$ Eingang der beiden Taktsignale
- $V_{SS} + V_{DD}$ Die Versorgungsspannungen
- SYNC Das SYNC-Signal
- $O_0 – O_3$ Die Output-Leitungen
- CM Das CM-RAM-Signal
- P_0 Ein Identifikations-Signal
- RESET Das Reset-Signal

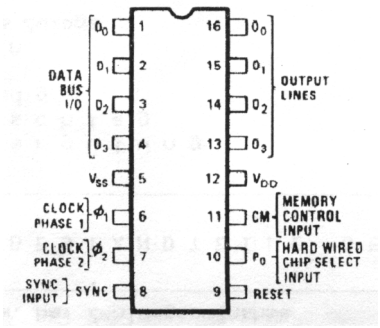


Abbildung 3.4: Die Pinbelegung des 4002

In einem MCS 4-System können ebenfalls bis zu 16 Chips angesprochen werden, allerdings werden diese Chips nicht mehr nachvollziehbar anders adressiert. Es wurden zwei Typen dieses Chips produziert: 4002-1 und 4002-2.

Und je nachdem, welcher Signalpegel am Pin P_0 anliegt, kommt es zu vier Chip-Nummern:

Chip-Nummer	***4002-Typ***	***P_0***
0	4002-1	high
1	4002-1	low
2	4002-2	high
3	4002-2	low

Tabelle 3.1: ID-Ermittlung des 4002

Es können also pro CM-RAM-Leitung vier Chips angeschlossen werden, und da der Prozessor vier CM-RAM-Leitungen besitzt, kommt es zu maximal 16 RAMs in jedem System.

Das Schema des 4002 zeigt einen etwas komplizierteren Aufbau:

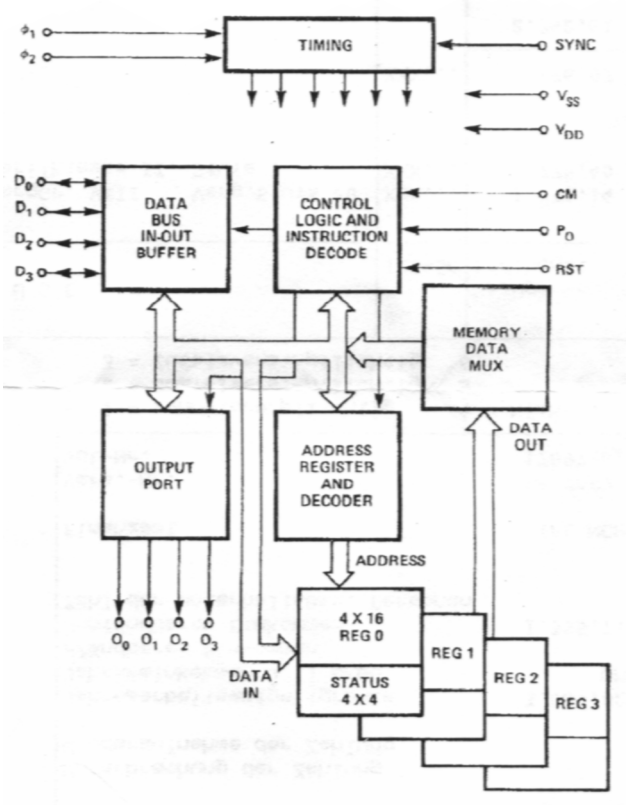


Abbildung 3.5: Schema des 4002

In einem 4002 gibt es anstatt einem einzigen inneren RAM ein Array aus 4 Registern mit 16 Adressen an denen jeweils 1 Wort (=4 Bit) steht und dazu jeweils ein Status-Register, das sich aus 4 Status-Charactern zusammensetzt, die auch jeweils 1 Wort aufnehmen. Wenn das Reset-Signal bei einem 4002 gesetzt wird, hat das Zurücksetzen der Register auch Auswirkung RAM-Array. Um das komplette RAM-Array zu löschen, muss das Reset-Signal allerdings über 32 Befehlszyklen anliegen.

Auch die RAM-Chips werden über den **SRC-Befehl** ausgewählt. Das CM-RAM-Signal gibt die Bank mit maximal vier Chips an. Die beiden höheren Bits (D_3 und D_2), die während X_2 empfangen werden geben die Chip-Nummer an und die niederen Bits wählen das (Status-)Register innerhalb dieses Chips aus. Die zweiten 4 Bits in X_3 geben die Adresse in dem Gewählten Register an und werden in einem Adress-Register gespeichert. Die Adresse wird für folgende Operationen benötigt: **RDM**, **ADM** und **SBM** sind Lese-Befehle, bei denen das Wort an der

Adresse auf den Datenbus gegeben wird und **WRM** schreibt ein Wort vom Datenbus an die abgespeicherte Adresse.
Um die Werte in den ausgewählten Status-Registern zu lesen und zu schreiben, gibt es die Befehle **RD0 – RD3** und **WR0 – WR3**, jeder Status-Character hat also sein eigenes Befehlspaar.
Die dritte Funktionsweise bildet der Befehl **WMP**, der ein Wort vom Datenbus an die Output-Ports weiterleitet.
Auch bei dem 4002 finden die Datentransfers der Lese- und Schreib-Befehle während der Phase X_2 statt.

3.2.3 4003

Der 4003 wird an den Systemschnittstellen eingesetzt und ist ein 10 Bit Schieberegister.

Er hat folgendes Aussehen:

- CP Taktsignal
- $V_{SS} + V_{DD}$ Die Versorgungsspannungen
- DATA IN Serieller Dateneingang
- $O_0 – O_9$ Die parallelen Output-Ports
- SERIAL OUT Serieller Ausgangsport
- E Enable-Signal

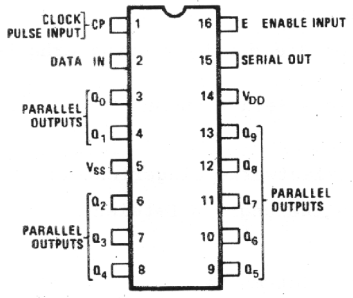


Abbildung 3.6: Pinbelegung des 4003

Es ist vorgesehen, den 4003 an die 4 Output-Ports von 4001 / 4002 anzuschließen. Ein Port wird mit CP verbunden, ein anderer mit E und ein dritter natürlich mit DATA IN.

Dass der 4003 als Schieberegister etwas simpler im Aufbau ist, verdeutlicht sein Schema:

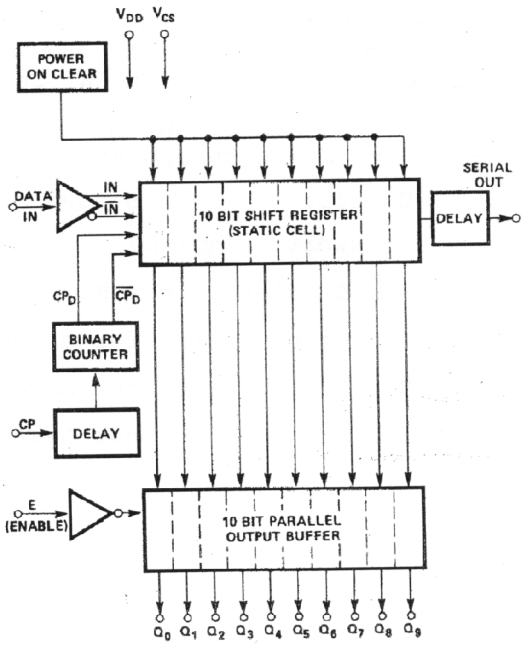


Abbildung 3.7: Schema des 4003

Wenn das Taktsignal gesetzt wird, kommt es zu einem Shift innerhalb des Chips, so dass das Bit am DATA IN-Port in das Register übernommen wird und das letzte Bit im Register wieder am SERIAL OUT-Port ausgegeben wird. Auf diese Weise lassen sich auch mehrere 4003 hintereinander schalten und erhöhen so die parallele Ausgabe auf mehr als 10 Bit. Der Pegel auf den Ports D_0 – D_9 liegt solange auf low, bis das E-Signal aktiviert wird. Dann werden die 10 Bits aus dem inneren Register gleichzeitig an den 10 Ports ausgegeben.

3.2.4 4004

Die wichtigste Komponente in einem MCS 4-System ist der 4004. Er wurde als Mikroprozessor entworfen und ist in der Lage Programme, die in den ROM-Chips abgespeichert sind, abzuarbeiten.

3.2 Die Komponenten

Auch der 4004 wird über 16 Pins angeschlossen:

- $D_0 – D_3$ Der Datenbus
- $\varphi_1 + \varphi_2$ Eingang der beiden Taktsignale
- $V_{SS} + V_{DD}$ Die Versorgungsspannungen
- SYNC Das SYNC-Signal
- $CM\text{-}RAM_x$ Die 4 CM-RAM-Signale
- CM-ROM Das CM-ROM-Signal
- TEST Ein Signal für Test-Zwecke
- RESET Das Reset-Signal

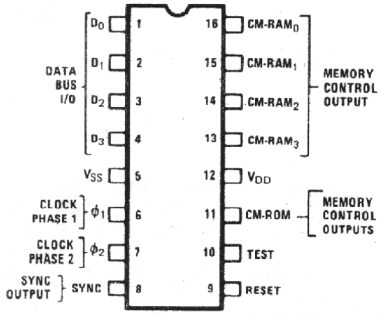


Abbildung 3.8: Pinbelegung des 4004

Da der 4004 die zentrale Recheneinheit darstellt und die restlichen Komponenten in einem MCS 4-System die Daten, die verarbeitet werden, bereitstellen oder speichern und keine weitere Operationen vornehmen, ist der Aufbau eines 4004 entsprechend komplex geraten:

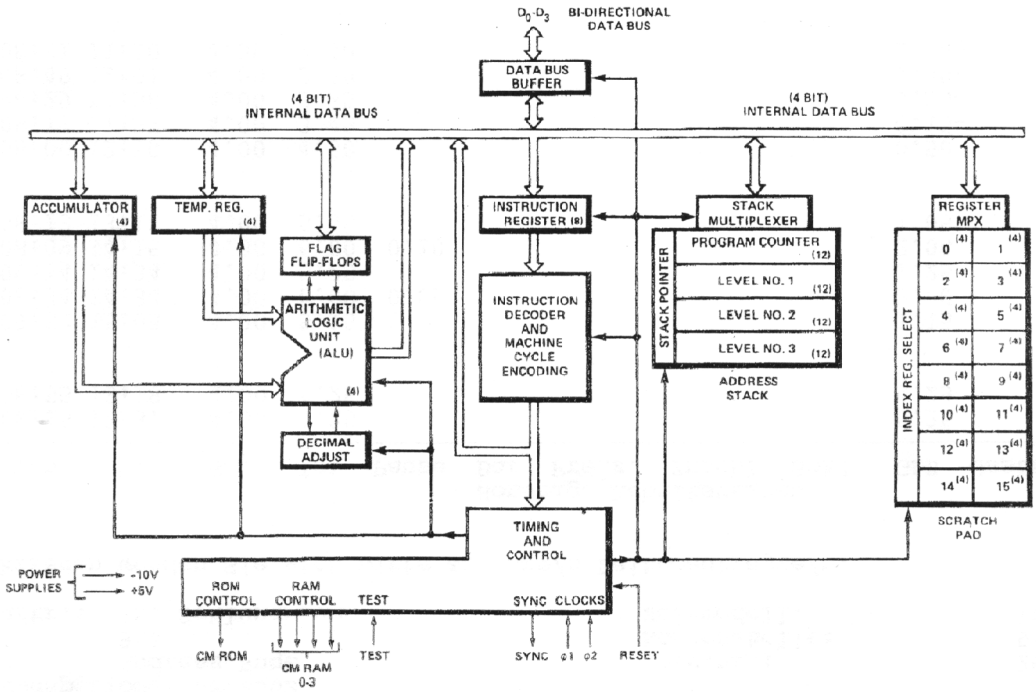


Abbildung 3.9: Schema des 4004

Der Datenverkehr innerhalb des 4004 verläuft über einen 4-bit breiten internen Datenbus, der die verschiedenen Funktionseinheiten verbindet. An der Schnitt-

stelle zum externen Bus (D_0 – D_3) ist noch ein **Data Bus Buffer** als Speicher zwischengeschaltet.

Für die Verknüpfung der Daten steht eine **ALU** (**A**rithmetic **L**ogical **U**nit) zur Verfügung, die Addition und Subtraktion (unter Berücksichtigung des Carrys) beherrscht. Die Einheit **Flag Flip-Flops** nimmt das Carry-Bit auf, das beim Überlauf gesetzt wird. Diese Operationen arbeiten rein binär, es ist aber möglich, ein **Decimal Adjust** durchzuführen. Dabei wird der Wert im Akkumulator so modifiziert, dass nur Werte von 0-9 auftreten und das Carry Flag gesetzt wird. Auf diese Art ist es auch möglich, BCD-Arithmetik durchzuführen. Die beiden Operanden werden am Eingang der ALU in einem **Accumulator** und einem **Temp.-Register** zwischengespeichert. Die Ausgabe der ALU geht direkt auf den Bus. Zusätzlich ist die ALU in der Lage, eine Shift-Operation durchzuführen, die sich auf den Akku-Inhalt und das Carry-Flag bezieht. Der Schiebevorgang kann bitweise in beide Richtungen ablaufen.

Da der Ablauf in einem MCS 4-System darauf beruht, dass der 4004 nacheinander Befehle abarbeitet, muss der Prozessor in der Lage sein, die Befehle zu erkennen und unterschiedliche Funktionen auszulösen. Dazu gehört im Inneren das Setzen von Enable-Signalen an alle Register, das Senden eines Codes an die ALU, um die erforderliche Arithmetik auszulösen, und das Steuern der Multiplexer und Treiber für den internen Bus, weil ja nur eine Einheit zur Zeit ihre Daten auf den Bus legen darf. Nach außen gehört zu den Aufgaben das Setzen der Command-Lines (CM-ROM, CM-RAM_x) und das SYNC-Signal. Das übernimmt das so genannte Steuerwerk: Die momentan aktuelle Adresse, an der der nächste Befehl abgespeichert ist, findet sich im **Program Counter**. Dieser liegt im **Adress Stack** zu dem auch drei Register gehören, die drei Level von Adressen aufnehmen können, um Unterprogrammaufrufe (JMS) und Returns (BBL) zu realisieren. In der Befehlsholphase werden die auf dem externen Bus liegenden Daten in dem **Instruction Register** abgelegt. Die Einheit **Timing and Control** ist für die Ausgaben des Steuerwerks zuständig. Die Signale müssen aber zu unterschiedlichen Zeiten in unterschiedlichen Kombinationen gesetzt werden. Die Verarbeitung hat also funktionellen Charakter und dafür sind Eingabegrößen nötig. Diese liefert der

Instruction Decoder and Machine Cycle Encoder, der die Befehle aus dem Instruktionsregister liest und auch den momentanen Subzyklus kennt und daraus die Eingabe generiert.

Als Hilfe für die Verarbeitung wurde bereits dem 4004 eine Registerbank spendiert, das so genannte **Scratch Pad**. Hier befinden sich 16 Register, die Daten zwischenzeitlich aufnehmen können.

Die Adressregister sind 12-bittig ausgeführt, alle anderen Register können nur 4 Bit aufnehmen. Das Instruktionsregister hat eine angegebene Breite von 8 Bit, besteht aber aus 2 Registern, einem OPR- und einem OPA-Register für die beiden Bestandteile eines Befehls.

Eine feinere Darstellung vom Schema des 4004 ist in [Fag96] abgebildet.

Es gibt auch noch Nebengrößen, die Einfluss auf die Steuerung nehmen: Das **TEST-Signal** kann genutzt werden, um den bedingten Sprung in einen Bereich mit Testcode zu führen und das **RESET-Signal** löscht die Inhalte aller Register und Flags. Um alle Adress- und Index-Register zu löschen, muss das Signal über 8 Befehlszyklen anliegen.

3.2.5 4008 / 4009

Zusätzlich zu den schon erwähnten Komponenten 4001 bis 4004 können im Systemdesign noch die Chips 4008 und 4009 verwendet werden. Diese beiden Chips waren im ursprünglichen Design nicht enthalten, wurden später entworfen und sind so genannte Bridges, d.h. sie bilden eine Brücke von einem MCS 4-System zu ROMs und RAMs anderer Hersteller. Die beiden Chiptypen werden wie ein 4001 behandelt. Als Steuersignale kommt bei beiden das CM-ROM- und SYNC-Signal zum Einsatz ebenso wie die Anbindung über den 4 Bit-Datenbus.

Der **4008** setzt die Adressen um. Während der Adressierung werden die 12 Bits in den drei Phasen gesammelt und in A_3 über 12 Ports ausgegeben. Die niederen 8 Bit auf 8 Adress-Ports und die höheren 4 Bit (Chip-Nummer) auf 4 Extra-Ports. Für die Ausführung des SRC-Befehls hat der 4008 ein eigenes SRC-Register, in dem die in X_2 und X_3 eingehenden 8 Bits gespeichert und bei I/O-Operationen während X_1 als Adresse wieder ausgegeben werden.

Die Befehle werden dann auf 8 Input-Ports vom **4009** in Empfang genommen und in der Befehlsholphase in zwei 4-bittige Datenpakete aufgeteilt und an den Prozessor gesendet. Außerdem ist dieser Chip zuständig für die Ausführung der I/O-Operationen (RDR / WRR), für deren Daten 4 bidirektionale I/O-Ports zur Verfügung stehen.

Als zusätzlicher Befehl wird WPM benutzt, um bei Einsatz eines RAMs neue Befehle des Programms abzuspeichern, die dann in späteren Befehlsholphasen wieder gelesen werden und zur Ausführung kommen. Da die Busbreite ja nur 4 Bits beträgt, muss dieser Befehl zweimal ausgeführt werden. Der 4008 ist dann dafür zuständig, Steuersignale zu generieren, die diesen Befehl kennzeichnen und auch, ob es sich um das niedere oder höhere Wort des Befehls handelt. Diese Worte werden dann über die I/O-Ports des 4009 an die RAMs ausgegeben.

3.3 Der Befehlssatz

Der Befehlssatz des 4004 besteht aus 1 Wort-Befehlen und 2 Wort-Befehlen. Da in der Befehlsholphase (M_1 und M_2) des Befehlszyklus ein Befehl geladen wird, erfordern die 2 Wort-Befehle auch einen zweiten Befehlszyklus. Die oberen 4 Bits eines Befehls heißen OPR und die unteren OPA. Die OPR-Bits werden in M_1 geholt und die OPA-Bits in M_2.

Die Befehle können folgende Ausprägungen und Formate haben („x“ = 0 oder 1):

- Basic Instructions als 1-Wort-Befehle

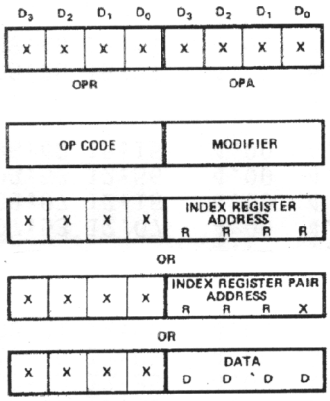


Tabelle 3.2: Befehlsformat der 1-Wort Basic Instructions

- Basic Instructions als 2-Wort-Befehle

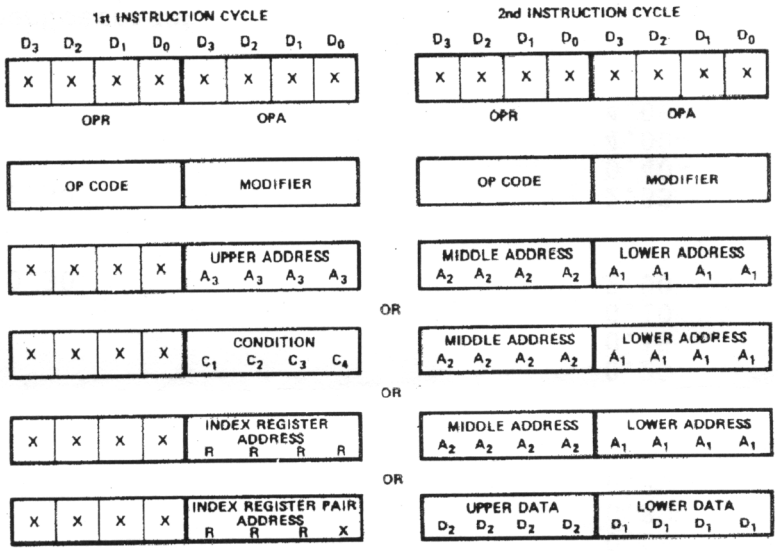


Tabelle 3.3: Befehlsformat der 2-Wort Basic Instructions

- I/O-, RAM- und Accumulator Group Instructions als 1-Wort-Befehle

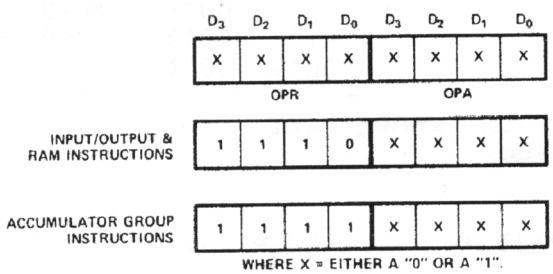


Tabelle 3.4: Befehlsformat der I/O-, RAM- und Accumulator Group Instructions

Die Basic Instructions lassen sich in vier Gruppen einordnen:

1. NOP
2. Sprungbefehle: JCN, JIN, JUN, JMS, BBL
3. Ladebefehle: FIM, FIN, LD, XCH, LDM
4. Arithmetische Befehle: INC, ISZ, ADD, SUB

Die Auflistung aller Befehle wurde [Int77] entnommen und nach Analyse der Erläuterung um RTL-Beschreibungen ergänzt. Sie sind ist in Anhang A zu finden.

MCS-4™ Instruction Set

[Those instructions preceded by an asterisk (*) are 2 word instructions that occupy 2 successive locations in ROM]

MACHINE INSTRUCTIONS

MNEMONIC	OPR $D_3 D_2 D_1 D_0$	OPA $D_3 D_2 D_1 D_0$	DESCRIPTION OF OPERATION
NOP	0 0 0 0	0 0 0 0	No operation.
*JCN	0 0 0 1 $A_2 A_2 A_2 A_2$	$C_1 C_2 C_3 C_4$ $A_1 A_1 A_1 A_1$	Jump to ROM address $A_2 A_2 A_2 A_2, A_1 A_1 A_1 A_1$ (within the same ROM that contains this JCN instruction) if condition $C_1 C_2 C_3 C_4$ [1] is true, otherwise skip (go to the next instruction in sequence).
*FIM	0 0 1 0 $D_2 D_2 D_2 D_2$	R R R 0 $D_1 D_1 D_1 D_1$	Fetch immediate (direct) from ROM Data D_2, D_1 to index register pair location RRR.[2]
SRC	0 0 1 0	R R R 1	Send register control. Send the address (contents of index register pair RRR) to ROM and RAM at X_2 and X_3 time in the Instruction Cycle.
FIN	0 0 1 1	R R R 0	Fetch indirect from ROM. Send contents of index register pair location 0 out as an address. Data fetched is placed into register pair location RRR at A_1 and A_2 time in the Instruction Cycle.
JIN	0 0 1 1	R R R 1	Jump indirect. Send contents of register pair RRR out as an address at A_1 and A_2 time in the Instruction Cycle.
*JUN	0 1 0 0 $A_2 A_2 A_2 A_2$	$A_3 A_3 A_3 A_3$ $A_1 A_1 A_1 A_1$	Jump unconditional to ROM address A_3, A_2, A_1.
*JMS	0 1 0 1 $A_2 A_2 A_2 A_2$	$A_3 A_3 A_3 A_3$ $A_1 A_1 A_1 A_1$	Jump to subroutine ROM address A_3, A_2, A_1, save old address. (Up 1 level in stack.)
INC	0 1 1 0	R R R R	Increment contents of register RRRR. [3]
*ISZ	0 1 1 1 $A_2 A_2 A_2 A_2$	R R R R $A_1 A_1 A_1 A_1$	Increment contents of register RRRR. Go to ROM address A_2, A_1 (within the same ROM that contains this ISZ instruction) if result ≠ 0, otherwise skip (go to the next instruction in sequence).
ADD	1 0 0 0	R R R R	Add contents of register RRRR to accumulator with carry.
SUB	1 0 0 1	R R R R	Subtract contents of register RRRR to accumulator with borrow.
LD	1 0 1 0	R R R R	Load contents of register RRRR to accumulator.
XCH	1 0 1 1	R R R R	Exchange contents of index register RRRR and accumulator.
BBL	1 1 0 0	D D D D	Branch back (down 1 level in stack) and load data DDDD to accumulator.
LDM	1 1 0 1	D D D D	Load data DDDD to accumulator.

INPUT/OUTPUT AND RAM INSTRUCTIONS

(The RAM's and ROM's operated on in the I/O and RAM instructions have been previously selected by the last SRC instruction executed.)

MNEMONIC	OPR $D_3 D_2 D_1 D_0$	OPA $D_3 D_2 D_1 D_0$	DESCRIPTION OF OPERATION
WRM	1 1 1 0	0 0 0 0	Write the contents of the accumulator into the previously selected RAM main memory character.
WMP	1 1 1 0	0 0 0 1	Write the contents of the accumulator into the previously selected RAM output port. (Output Lines)
WRR	1 1 1 0	0 0 1 0	Write the contents of the accumulator into the previously selected ROM output port. (I/O Lines)
WRØ [4]	1 1 1 0	0 1 0 0	Write the contents of the accumulator into the previously selected RAM status character 0.
WR1 [4]	1 1 1 0	0 1 0 1	Write the contents of the accumulator into the previously selected RAM status character 1.
WR2 [4]	1 1 1 0	0 1 1 0	Write the contents of the accumulator into the previously selected RAM status character 2.
WR3 [4]	1 1 1 0	0 1 1 1	Write the contents of the accumulator into the previously selected RAM status character 3.
SBM	1 1 1 0	1 0 0 0	Subtract the previously selected RAM main memory character from accumulator with borrow.
RDM	1 1 1 0	1 0 0 1	Read the previously selected RAM main memory character into the accumulator.
RDR	1 1 1 0	1 0 1 0	Read the contents of the previously selected ROM input port into the accumulator. (I/O Lines)
ADM	1 1 1 0	1 0 1 1	Add the previously selected RAM main memory character to accumulator with carry
RDØ [4]	1 1 1 0	1 1 0 0	Read the previously selected RAM status character 0 into accumulator.
RD1 [4]	1 1 1 0	1 1 0 1	Read the previously selected RAM status character 1 into accumulator.
RD2 [4]	1 1 1 0	1 1 1 0	Read the previously selected RAM status character 2 into accumulator.
RD3 [4]	1 1 1 0	1 1 1 1	Read the previously selected RAM status character 3 into accumulator

ACCUMULATOR GROUP INSTRUCTIONS

CLB	1 1 1 1	0 0 0 0	Clear both. (Accumulator and carry)
CLC	1 1 1 1	0 0 0 1	Clear carry.
IAC	1 1 1 1	0 0 1 0	Increment accumulator.
CMC	1 1 1 1	0 0 1 1	Complement carry.
CMA	1 1 1 1	0 1 0 0	Complement accumulator.
RAL	1 1 1 1	0 1 0 1	Rotate left. (Accumulator and carry)
RAR	1 1 1 1	0 1 1 0	Rotate right. (Accumulator and carry)
TCC	1 1 1 1	0 1 1 1	Transmit carry to accumulator and clear carry.
DAC	1 1 1 1	1 0 0 0	Decrement accumulator.
TCS	1 1 1 1	1 0 0 1	Transfer carry subtract and clear carry.
STC	1 1 1 1	1 0 1 0	Set carry.
DAA	1 1 1 1	1 0 1 1	Decimal adjust accumulator.
KBP	1 1 1 1	1 1 0 0	Keyboard process. Converts the contents of the accumulator from a one out of four code to a binary code.
DCL	1 1 1 1	1 1 0 1	Designate command line.

Tabelle 3.5: Befehlsliste des 4004

Was bei Betrachtung der Befehle und des 4004-Schemas auffällt, ist das unverständlicherweise Ausbleiben von Operationen, die Daten logisch vergleichen (AND, OR, XOR etc.), es gibt keine Bitoperationen, Sprungadressen können nur in drei Leveln im Adress-Stack gespeichert werden und es ist auch keinerlei Interrupt-Handling vorgesehen.

3.4 Architekturausprägungen

Wenn man jetzt die Beschreibung und das Verhalten der Komponenten betrachtet, lässt sich erkennen, dass bei der Konstruktion das 1946 vorgestellte Prinzip einer von-Neumann Architektur [Dud01] realisiert wurde. Und da das MCS 4-System die Grundlage für weitere Architekturentwicklungen des Halbleiterherstellers Intel bildet, hat das Prinzip bis in die heutige Zeit eine große Bedeutung.

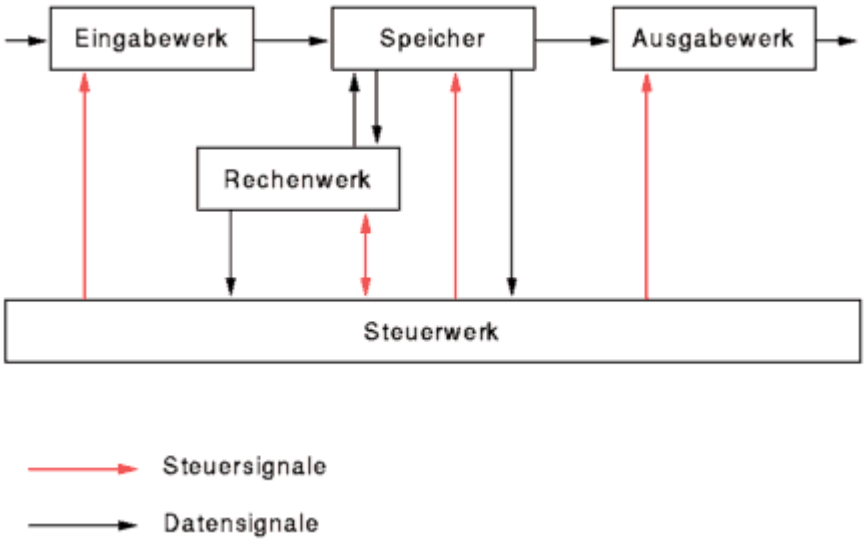


Abbildung 3.10: Architektur nach von-Neumann

- Eingabewerk: Das System erhält seine Eingaben über die als Input-Ports deklarierten I/O-Ports der ROMs.
- Ausgabewerk: Die Ausgaben finden über die Output-Ports der RAMs und über die I/O-Ports der ROMs statt, die als Ouput-Ports definiert wurden. Als zusätzliche Hilfe können die 4003-Chips mit in die Ausgabe integriert werden.

- Speicher: Als Speicher fungieren die 4001 (ROM) bzw. i4008/i4009 und 4002 (RAM). Die 4001 bilden den Programm- und Konstantenspeicher ab, die 4002 fungieren als Datenspeicher.
- Rechenwerk: Das Rechenwerk befindet sich im Mikroprozessor 4004. Dazu gehören die ALU, ihre Eingaberegister (Akkumulator und Temp.-Reg), die Flag Flip-Flops, die Decimal Adjust-Einheit und das Scratch Pad.
- Steuerwerk: Der größte Teil des Steuerwerks befindet sich im 4004. Hier gibt es die Timing and Control-Einheit, die die Steuersignale innerhalb des Prozessors und an die angeschlossenen Chips regelt. Dazu gehören noch das Instruktionsregister, der Adress-Stack und natürlich der Instruction Decoder. Die ROMs und RAMs haben allerdings auch Einheiten, die zum Steuerwerk gehören, da sie auf die I/O- und RAM-Befehle geeignet reagieren müssen.
- Datenbus: Alle Einheiten werden über einen Datenbus (unterteilt in einen externen Teil zwischen den Chips und einen internen Teil innerhalb des 4004) miteinander verbunden, der auch in einem MCS 4-System einen Flaschenhals darstellt. Zum einen müssen sowohl Adressen und Daten über denselben Bus transportiert werden und dann ist er mit 4 Bit Breite eigentlich unterdimensioniert, da die Adressen 12-bittig und die Befehle 8-bittig übertragen werden müssten. Das führt zu den Subzyklen in der Befehlsausführung.

Ein MCS 4-System besteht mindestens aus einem 4004 und einem 4001. Ein 4001 hat 256 x 8 Bit = 2.048 Bit Speicherkapazität. Maximal sind 16 ROMs und 16 RAMs von einem 4004 ansprechbar, das führt zu einem Befehlsspeicher von 16 x 2048 Bit = 32.768 Bit. Hierbei lässt sich allerdings nicht sagen, dass ein MCS 4-System maximal Programme mit 4.096 Befehlen verarbeiten kann, weil der Befehlssatz ja auch Befehle enthält, die 2 Wörter Speicher brauchen und es können mit dem FIN-Befehl auch direkt konstante Werte geladen werden. Ein 4002 hat 4 Register á 16 + 4 Adressen an denen 4 Bit gespeichert werden, das ergibt in der Summe 4 x 20 x 4 = 320 Bit pro RAM und 16 x 320 = 5120 Bit maximal in einem MCS 4-System. Dazwischen wären alle Kombinationen denkbar.

4 Anwendungsbeispiele

Um die Einsatzmöglichkeiten eines MCS 4-Systems zu demonstrieren, folgen jetzt ein paar Beispiele. Da die Programmierung direkt in Assembler erfolgt, werden nur Bruchteile komplexer Strukturen beschrieben, um die Komplexität und Übersicht nicht allzu sehr leiden zu lassen. Außerdem sollen die Beispiele leicht zu verstehen sein, ohne detaillierte Kenntnisse in Assemblerprogrammierung voraus zu setzen. Dafür sei ein Blick in [Roh01] empfohlen.
Der einzige Befehl, der in den Programmen eine Sonderstellung einnimmt, ist der JCN-Befehl. Die Condition-Bits $C_1 - C_4$ sind zum einfacheren Verständnis in eine Klartextform umgewandelt:

JCN	C	Jump on carry		JCN	NC	Not jump on carry
	Z	Jump on zero			NZ	Not jump on zero
	T	Jump on test			NT	Not jump on test

Tabelle 4.1: Parameter von JCN

4.1 Kontrollflusssteuerung

Weil Programme fast nie strikt sequentiell abgearbeitet werden können, gibt es Befehle zur Kontrollflusssteuerung. Diese Befehle sind unter dem Oberbegriff Sprungbefehle zusammengefasst. Ein Sprung kann ohne Bedingung erfolgen (JIN und JUN), nur im Falle einer Bedingung stattfinden (JCN und ISZ) oder als Aufruf einer Subroutine (JMS) mit Rücksprung (BBL) dienen.
Wie in Kapitel 3.1 beschrieben, wird der Programmzähler automatisch immer um 1 erhöht. Damit der Prozessor nicht den gesamten Speicherbereich durchläuft, ist es notwendig, am Ende des Programms einen Sprung auszuführen.

```
; jmp_inc.asm
; increase in an endless loop
 FIM R0R1, 0 ; initialize
loop:
 INC R0      ; increase Register 0
 JUN loop    ; next iteration
```

Zuerst werden die Register R0 und R1 mit 0 initialisiert und anschließend der Inhalt von Register R0 um 1 erhöht. Danach wird eine Endlosschleife am Ende ausgeführt, weil immer wieder zu Label 'done' gesprungen wird.
Die andere Möglichkeit ist ein Sprung an den Anfang des Programms. In diesem Beispiel wird der Inhalt immer wieder um 1 erhöht, weil vor den INC-Befehl gesprungen wird.

```
; jmp_test.asm
; increase on test
 FIM R0R1, 0      ; initialize
begin:
 JMS checkTest    ; check test-signal
 JCN Z, begin     ; no test, try again
 INC R0           ; increase Register 0
 LDM 1            ; 1 into accumulator
 XCH R1           ; store 1 in Register 1
 JUN begin        ; jump back
; subroutine to check test-signal
checkTest:
 JCN T, test
 LDM 0            ; 0 into accumulator
 XCH R1           ; store 0 in Register 1
 BBL 0            ; no inc
test:
 LD R1            ; load Register 1
 JCN Z, newTest   ; signal is new
 BBL 0            ; inc
newTest:
 BBL 1            ; no inc
```

Dieses Beispielprogramm erhöht den Inhalt von Register R0 nur dann, wenn das Signal TEST am Prozessor aktiviert wird. Zu diesem Zweck wird das Testsignal in einer Subroutine abgefragt. Der erste Befehl in 'begin' springt in die Subroutine 'checkTest'. Der BBL-Befehl, der einen Rücksprung bewirkt, hinterlegt einen so genannten Return-Code im Akkumulator. Hier bedeutet 0, dass die Erhöhung

nicht durchgeführt wird, und eine 1, dass erhöht werden soll. JCN T springt nur nach 'test', wenn das TEST-Signal anliegt.

In diesem Beispiel wird eine minimale State-Machine benötigt:

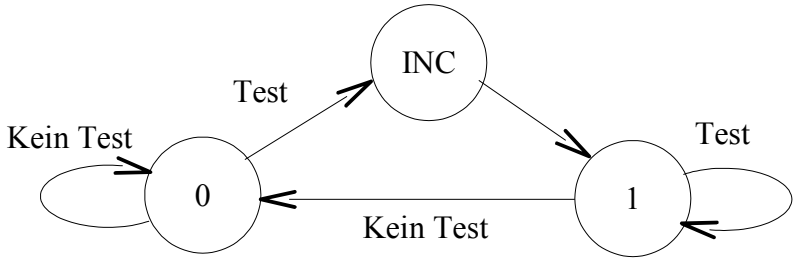


Abbildung 4.1: State-Machine zur Signalabfrage

Das ist nötig, damit nur bei Aktivierung des TEST-Signals hochgezählt wird und nicht ständig. Der momentane Zustand wird in Register R1 abgelegt und in 'test' unter LD R1 in den Akku übertragen. Nur wenn der Akku 0 enthält (JCN Z), wird nach 'newTest' gesprungen und die Subroutine mit 1 beendet. In den anderen Fällen (TEST-Signal nicht gesetzt und TEST-Signal gesetzt, aber in Zustand 1) wird mit 0 zurückgesprungen. Falls das TEST-Signal nicht anliegt, wird der Zustand 0 in R1 eingetragen. Der Return-Code wird immer im Akku abgelegt und mittels JCN Z wieder überprüft. Falls der Akku 0 ist, wird sofort an den Anfang gesprungen. Im Falle von 1 wird die Erhöhung durchgeführt und der Zustand auf 1 gesetzt. Dann wird ebenfalls wieder an den Anfang gesprungen (JUN begin).

Man sieht schon an der Länge des Programms und der Erklärung wie schnell die Komplexität von Assemblerprogrammen zunimmt. Die Labels sollen dabei helfen, nicht die Orientierung im Code zu verlieren.

4.2 Arithmetik

Um Zahlen zu verarbeiten, besitzt der 4004 eine ALU, die in der Lage ist, mittels eines Addierers Additionen und Subtraktionen auszuführen. Dabei kann die ALU nicht nur mit normalen Binärzahlen umgehen, sondern hat mit der Logik 'Decimal Adjust' auch die Fähigkeit, BCD-Zahlen zu verarbeiten. BCD-Zahlen sind **B**inary-**C**oded-**D**ecimals, d.h. in 4 Bit werden die Zahlen 0 – 9 abgespeichert, der Zahlen-

raum von 10 – 15 bleibt ungenutzt. Da solche Zahlen (0xa bis 0xf) bei der Verarbeitung aber durchaus auftreten können, konvertiert ein DAA-Befehl die Zahlen von 10 bis 15 durch nochmalige Addition von 0x6 in den Zahlenraum von 0 bis 5 und setzt das Carry-Flag. Wenn das Carry-Flag nach einer Addition schon gesetzt ist (Bsp. 8 + 9 = 17, binär 2 und Carry), dann werden auch diese Ergebnisse umgewandelt. Bei einer Subtraktion kann es auch dazu kommen, dass eine Korrektur nötig ist, dann ist das Ergebnis nämlich 0x6 bis 0xf bei gesetzten Carry-Flag. In diesem Fall wird vom Ergebnis 0x6 abgezogen und das gesetzte Carry-Flag kann dann als Borrow interpretiert werden. Das gilt aber nur für die Fälle, bei denen die Subtraktion keinen Vorzeichenwechsel auslöst. Bsp: 22 – 4 = 18, aber 2 – 4 = -2, in beiden Beispielen wird hinten 2 – 4 berechnet, DAA ist aber nur für den ersten Fall anwendbar.

4.2.1 Addition

Die einfachste Art der Arithmetik stellt eine Addition dar:

```
; add.asm
; add two 4bit numbers on the Intel 4004
 FIM R0R1, 0x78 ; initialize R0=8 & R1=7
 FIM R2R3, 0    ; clear result pair
 LD R0          ; load R0 into accumulator
 ADD R1         ; add second number
 XCH R1         ; and store in R1
done:
 JUN done       ; end
```

Zuerst werden die zu addierenden Zahlen 7 und 8 in die ersten beiden Register geladen. Danach wird das Ergebnis der Addition (0xf) dieser Zahlen in Register R1 abgelegt. Falls ein Überlauf entstehen würde, wäre danach das Carry-Flag gesetzt. Diese Addition läuft binär ab.

Für BCD-Zahlen ist eventuell eine Korrektur des Ergebnisses nötig:

```
; bcd_add.asm
; add two 4bit numbers on the Intel 4004
; bcd-wise
 FIM R0R1, 0x78 ; initialize R0=8 & R1=7
 FIM R2R3, 0    ; clear result pair
 LD R0          ; load R0 into accumulator
 ADD R1         ; add second number
 DAA            ; decimal adjust accumulator
 XCH R2         ; store new result in R2
done:
 JUN done       ; end
```

Die Addition läuft am Anfang genauso ab wie im vorigen Beispiel. Nur wird das Ergebnis noch einmal in den Akkumulator geladen und ein DAA durchgeführt, der für die nötige Korrektur sorgt. Das führt dazu, das in R2 nach dem DAA-Befehl eine 5 abgespeichert wurde, und das Carry-Flag ist auch gesetzt, um einen BCD-Überlauf anzuzeigen.

4.2.2 Zahlenformate

Was bei der Addition noch kein Problem darstellt, ist auf jeden Fall bei der Subtraktion eine Überlegung wert: In welchem Format werden Zahlen dargestellt? Im Prinzip kommen dafür vier verschiedenen Versionen in Frage (Der angegebene Zahlenbereich bezieht sich auf 4-bittige Zahlen):

1. Alle Zahlen entsprechen ihrem binäre Wert – Das führt zu einem Bereich von 0 bis 15 (nur positive Zahlen)
2. Das MSB (**M**ost **S**ignificant **B**it) zeigt positive Zahlen mit 0 und negative Zahlen mit 1 an – Der Bereich ist dann 0 bis 7 und -0 bis -7
3. Negative Zahlen werden im 2-Komplement abgespeichert (Positive Zahl invertieren und 1 addieren) – Der Zahlenbereich reicht dann von -8 bis 7
4. Das Bit, das das Vorzeichen der Zahl darstellt wird getrennt gespeichert und betrachtet – Dann reicht der Bereich von 0 bis 15 und -0 bis -15

Variante 1 ist nur eingeschränkt nutzbar, weil es schwierig ist, Programme zu entwickeln, die nur mit positiven Zahlen umgehen können. Die Möglichkeiten 2 und 3 sind extrem unpraktisch, weil der Zahlenraum zu klein ist – da passen nicht

einmal alle BCD-Zahlen hinein. Am sinnvollsten erscheint eine praktische Nutzung von Variante 4. Gerade weil das erste Einsatzgebiet eines MCS 4-Systems ein Taschenrechner war und sich deshalb eine Verarbeitung von BCD-Zahlen anbietet, kann man die Zahlen positiv abspeichern und das Vorzeichen getrennt davon festhalten (z.B. als Character-Bit im 4002). Hierbei ist dann auch Variante 1 impliziert, man kann die Betrachtung des Vorzeichenbits auch auslassen.

4.2.3 Subtraktion

Eine Subtraktion zu realisieren sieht zunächst ähnlich aus wie eine Addition:

```
; sub.asm
; subtract two 4bit numbers on the Intel 4004
 FIM R0R1, 0x35 ; initialize R0=5 & R1=3
 FIM R2R3, 0    ; clear result pair
 LD R0          ; load R0 into accumulator
 SUB R1         ; subtract second number
 XCH R1         ; and store in R1
done:
 JUN done       ; end
```

Auch bei der Subtraktion von BCD-Zahlen ist eine Korrektur durchzuführen:

```
; bcd_sub.asm
; subtract two 4bit numbers on the Intel 4004
; bcd-wise
 FIM R0R1, 0x42 ; initialize R0=2 & R1=4
 FIM R2R3, 0    ; clear result pair
 LD R0          ; load R0 into accumulator
 SUB R1         ; subtract second number
 XCH R1         ; and store in R1
 JCN C, done    ; carry not set when result<0
 LD R1          ;  load result into accumulator
 CMA            ; build complement of accu
 CLC            ; clear carry
 IAC            ; increment accu (=> 2-complement)
 XCH R1         ; right result back to R1
 STC            ; set carry to show borrow
done:
 JUN done       ; end
```

Die Korrektur ist allerdings anders realisiert: Anstatt den Befehl DAA zu benutzen wird nach der Subtraktion das Carry-Flag ausgewertet (JCN C, done) und nur im

Falle des nicht gesetzten Flags die Korrektur des Ergebnisses angestoßen. Das liegt an der internen Realisierung der Subtraktion. In [Fag96] ist beschrieben, dass die ALU nur einen Addierer besitzt, demzufolge wird die Subtraktion auf eine Addition zurückgeführt. Das geschieht, indem man nicht die zweite Zahl von der ersten abzieht sondern ihr Komplement addiert. Das hat zur Folge, dass bei einer Subtraktion von BCD-Zahlen ein Überlauf entsteht, wenn das Ergebnis negativ wird. Um dann das korrekte Ergebnis zu erhalten, braucht man das 2-Komplement des Resultats zu berechnen. Das Carry-Flag wird dann noch gesetzt um ein 'Borrow' anzuzeigen.

Diese Beispiele haben bisher nur positive Zahlen betrachtet. Für negative Zahlen sollte man im Programm das Vorzeichenbit betrachten und dann die geeignete Arithmetik wählen. So kann man z.B. die Subtraktion von zwei negativen Zahlen in eine Addition dieser Zahlen umwandeln und anschließend das Ergebnis negativ markieren oder bei Subtraktion einer negativen Zahl gleich eine Addition wählen. Außerdem sollte man immer die kleinere Zahl von der Größeren abziehen, dann kann man auch bei der Subtraktion DAA benutzen (s.o.) und muss bei Bedarf am Ende nur das Vorzeichen korrigieren.

4.3 RAM-Operationen

Während die Beispiele aus dem vorigen Abschnitt mit einem ROM auskommen, werden jetzt Programme gezeigt, die sich die Operanden aus dem RAM holen. Das ist nötig, weil die Programme ja bei Bestellung der ROM-Chips fest eingebrannt werden und das ScratchPad des 4004 für größere Berechnungen zu klein dimensioniert ist.

Das folgende Programm addiert zwei BCD-Zahlen, die in RAM Register 0 und 1 abgelegt sind und speichert das Ergebnis in Register 2:

```
; ram_add.asm
; add two BCD numbers on the Intel 4004
init:
 FIM R0R1, 0 ; ram 0, bank 0, address 0
 FIM R2R3, 1 ; ram 0, bank 1, address 0
 FIM R4R5, 2 ; ram 0, bank 2, address 0
 LDM 1
 SRC 0       ; select register 0, address 0
 WRM         ; store 1
 INC R1
 LDM 2
 SRC 0       ; select register 0, address 1
 WRM         ; store 2
 INC R1
 LDM 3
 SRC 0       ; select register 0, address 2
 WRM         ; store 3
 WR0         ; store length 3 LDM 1
 LDM 1
 SRC 1       ; select register 1, address 0
 WRM         ; store 1
 INC R3
 LDM 1
 SRC 1       ; select register 1, address 1
 WRM         ; store 1
 INC R3
 LDM 7
 SRC 1       ; select register 1, address 2
 WRM         ; store 7
 INC R3
 LDM 4
 src 1       ; select register 1, address 3
 WRM         ; store 4
 WR0         ; store length 4
```

Im 'Init'-Bereich werden zuerst die Zeiger auf die RAM-Bänke gesetzt. Die Register R0 und R1 zeigen auf den ersten Operanden in Bank 1, R2 und R3 auf den zweiten Operanden in Bank 2 und das Ergebnis wird dann in Bank 3 abgespeichert (R4 und R5).

Danach wird die Zahl 321 in Bank 1 und 4711 in Bank 2 abgelegt. Die Längenangabe der beiden Zahlen wird im Status-Character 0 erfasst (WR0).

```
begin:
 SRC 1        ; number 2
 RD0          ; read length
 XCH R6
 SRC 0        ; number 1
 RD0          ; read length
 SUB R6
 JCN NC,len2  ; number 2 >= 1
 JUN go
len2:
 SRC 1        ; take number 2
go:
 RD0          ; read length
 XCH R6       ; store length
 LDM 0        ; reset addresses
 XCH R1
 LDM 0
 XCH R3
```

Mit 'begin' wird der dynamische Teil eingeleitet. Zuerst werden die beiden Längen eingelesen und per SUB-Befehl verglichen. Es folgt eine Fallunterscheidung, die die größere Längenangabe nochmals lädt und am Anfang von 'go' im Register R6 ablegt. Das Register R6 soll somit als Schleifenindex dienen. Anschließend werden die Adresszeiger R1 und R3 wieder auf 0 zurückgesetzt

```
doAdd:
 JMS ldCarry
 SRC 0
 RDM            ; read digit of number 1
 SRC 1
 ADM            ; add digit of number 2
 DAA
 SRC 2
 WRM            ; write digit of result
 TCC            ; store carry
 XCH R7
 LD R6          ; check length
 DAC
 JCN Z, endAdd ; no digits left, ready
 XCH R6
 INC R1         ; next addresses
 INC R3
 INC R5
 JUN doAdd      ; add next digits
```

Nun folgt das Additionsmodul. Zuerst wird im Unterprogramm 'ldCarry' (s.u.) das Carry-Flag berechnet. Dann wird eine Ziffer des ersten Operanden in den Akkumulator geladen und gleich danach die entsprechende Ziffer des zweiten Operanden dazu addiert. Da dieses Programm BCD-Zahlen addiert, wird anschließend eine Korrektur des Ergebnisses veranlasst. Der nächste Schritt ist eine Ausgabe der Ergebnisziffer in das RAM. Nun wird das Carry-Flag in Register R7 gerettet. Damit ist die Addition beendet und es kommt die Überprüfung der Abbruchbedingung der Schleife. Zuerst wird R6 geladen und verringert und falls der Akkumulator eine 0 enthält, wird die Schleife verlassen (JCN Z, endAdd). Im anderen Fall werden die Adresszeiger inkrementiert und an den Anfang der Schleife gesprungen.

```
endAdd:
 JMS ldCarry
 JCN NC, len ; carry?
 INC R5
 TCC
 SRC 2
 WRM          ; store carry in next digit
len:
 LD R5
 IAC
 WR0         ; write lenght of result
done:
 JUN done    ; end
ldCarry:
 LDM 15      ; restore carry
 ADD R7
 BBL 0
```

Am Ende des Programms wird noch einmal das gespeicherte Carry-Flag betrachtet und falls es 1 sein sollte, wird das Adressregister nochmals erhöht und die 1 vor das Ergebnis geschrieben. In jedem Fall wird dieser Adresszeiger in 'len2' erhöht und als Längenangabe des Ergebnisses in das Status-Character 0 von Bank 3 geschrieben.

Dieser Programmablauf lässt sich auch auf eine Subtraktion übertragen. Es werden die gleichen Zahlen verwendet. Die erste Änderung findet man im Abschnitt 'begin': Falls die zweite Zahl größer als die erste ist, wird das Ergebnis

negativ. Deshalb werden in 'len2' die Zeiger auf die Bänke R0 und R2 ausgetauscht und das Vorzeichen des Ergebnisses in Register R8 (1 für negativ) abgespeichert. Dann zieht man die erste Zahl von der zweiten ab und markiert das Ergebnis als negativ.

```
begin:
 SRC 1         ; number 2
 RD0           ; read length
 XCH R6
 SRC 0         ; number 1
 RD0           ; read length
 SUB R6
 JCN NC, len2  ; number 2 > 1
 JCN NZ, go    ; number 1 > 2
 RD0           ; length number 1
 DAC
 XCH R6        ; store in R6
check:
 CLC
 LD R6
 XCH R1
 LD R6
 XCH R3
 SRC 0
 RDM           ; read digit number 2
 SRC 1
 SBM           ; subtract digit number 1
 JCN NC, len2  ; number 2 > 1
 JCN NZ, go    ; number 1 > 2
 LD R6         ; digits equal
 JCN Z, go     ; last digit checked
 DAC
 XCH R6
 JUN check
len2:
 LD R0         ; change numbers
 XCH R2
 XCH R0
 LDM 1         ; store negativ sign
 XCH R8
 SRC 0
```

Die Längenberechnung ist etwas komplizierter als bei der Addition, weil auch die einzelnen Ziffern nacheinander herangezogen werden müssen (z.B. 200 – 240). Diese Kontrolle erfolgt im Abschnitt 'check', der von vorne nach hinten die

Ziffern subtrahiert. Im 'go'-Bereich werden wieder die maximale Länge als Schleifenindex in R6 abgelegt und die Adresszeiger auf 0 gebracht.

```
doSub:
 JMS ldBorrow
 SRC 0
 RDM          ; read digit of number 1
 SRC 1
 SBM          ; subtract digit of number 2
 JCN C, go2   ; carry not set when result<0
 STC          ; need borrow
 DAA
go2:
 SRC 2
 WRM          ; write digit of result
 TCC          ; store borrow
 XCH R7
 RDM
 JCN NZ, noZero
 INC R9       ; zeros in result
 JUN go3
noZero:
 LDM 0
 XCH R9
go3:
```

Falls nach der Subtraktion eine positive Ergebnisziffer vorliegt (JCN C, go2), wird bei 'go2' fortgesetzt, ansonsten das Carry-Flag gesetzt und eine Korrektur durchgeführt. Der Bereich 'go2' gibt die Ziffer in Bank 3 aus, rettet das Carry-Flag (hier als borrow) und lädt dann die Ziffer nochmal, um zu prüfen ob sie 0 war. Wenn das der Fall ist, wird Register R9 erhöht. In R9 soll die Anzahl an führenden Nullen im Ergebnis festgehalten werden. Falls bei der Berechnung eine Zahl ungleich 0 ist, wird R9 wieder gelöscht. Abschnitt 'go3' überprüft die Abbruchbedingung und erhöht die Adresszeiger.

```
endSub:
 LD R8        ; write sign
 SRC 2
 WR1
 LD R5        ; write length of result
 IAC
 SUB R9
 WR0
```

Am Ende des Programms wird nur das Vorzeichenbit aus R8 in das Status-Character 1 von Bank 3 gespeichert und die Länge in Status-Character 0. Vorher wird die Anzahl an Nullen abgezogen.
Diese Beispielprogramme arbeiten nur mit positiven Zahlen (immer eine 0 in Status-Character 1) und einem möglichen negativen Ergebnis nach Subtraktion. Wenn man als Operanden negative Zahlen abspeichert, müssen weitere Fallunterscheidungen vorgenommen werden. Man kann dann sowohl die Module zur Addition und Subtraktion in einem Programm halten und dann die bessere Arithmetik auswählen. Darauf soll an dieser Stelle aber nicht weiter eingegangen werden, da dafür alle geeigneten Programmteile vorgestellt wurden. Ob damals im Rechner von Busicom wirklich nach diesem System gerechnet wurde, konnte leider nicht mehr ermittelt werden.

4.4 I/O Operationen

Es ist jetzt an der Zeit, dass ein MCS 4-System in Interaktion mit dem Benutzer tritt. Das folgende Programm ist in der Lage, Eingaben an den Input-Ports der ROMs entgegen zu nehmen und diese Eingaben zu verarbeiten. Es kommt zu diesem Zweck eine Architektur zum Einsatz, die neben dem 4004 zwei ROMs und ein RAM besitzt. Zusätzlich gibt es eine Eingabelogik, die verschiedene Tastendrücke am System geeignet kodiert. Die Logik hat 15 Eingänge und 8 Ausgänge, die an die 4 I/O-Ports der beiden 4001 angeschlossen werden, die zu diesem Zweck entsprechend konfiguriert sein müssen. Die Eingänge entsprechen den Zahlen 0 – 9, +, -, =, C und CR.

Hier die Entscheidungstabelle für die Kodierung:

	Code	Q_0	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	ACK
0	0x0	0	0	0	0	0	0	0	**1**
1	0x1	**1**	0	0	0	0	0	0	**1**
2	0x2	0	**1**	0	0	0	0	0	**1**
3	0x3	0	0	**1**	0	0	0	0	**1**
4	0x4	0	0	0	**1**	0	0	0	**1**
5	0x5	**1**	0	0	0	**1**	0	0	**1**
6	0x6	0	**1**	0	0	**1**	0	0	**1**
7	0x7	0	0	**1**	0	**1**	0	0	**1**
8	0x8	0	0	0	**1**	**1**	0	0	**1**
9	0x9	**1**	0	0	0	0	**1**	0	**1**
+	0xa	0	**1**	0	0	0	**1**	0	**1**
-	0xb	0	0	**1**	0	0	**1**	0	**1**
=	0xc	0	0	0	**1**	0	**1**	0	**1**
C	0xd	**1**	0	0	0	0	0	**1**	**1**
CR	0xe	0	**1**	0	0	0	0	**1**	**1**
keine		0	0	0	0	0	0	0	0

Tabelle 4.2: Entscheidungstabelle für die Kodierung der Eingaben

$Q_0 - Q_3$ werden mit $I/O_0 - I/O_3$ von ROM 0, $Q_4 - Q_6$ mit $I/O_0 - I/O_2$ von ROM 1 und ACK mit I/O_3 von ROM 1 verbunden.

Wie an den Tasten schon zu erkennen ist, funktioniert dieses MCS 4-System wie ein minimaler Taschenrechner. Der 4002 hat die gleiche Aufgabe wie in den angesprochenen Beispielen aus dem vorigen Kapitel. In Bank 0 wird der erste Operand, in Bank 1 der zweite und das Ergebnis in Bank 2 abgespeichert. Als einzige Erweiterung wird die durchzuführende Operation (+ oder -) im Status Character 2 von Bank 0 vermerkt. Die Addition und Subtraktion wurde auch den vorgestellten Beispielen entnommen und als Unterprogramm in ROM 1 abgelegt.

4.4 I/O Operationen

Um das Verständnis des Hauptprogramms etwas zu erleichtern, wird die Registernutzung aufgelistet:

R0R1	Statisch für ROM 0 und RAM 0, Bank 0
R2R3	Statisch für ROM 1 und RAM 0, Bank 1
R4	Temporäre Zwischenspeicher
R5	Konstante 0x8 für ACK-Filter
R6	State für Tastendrücke
R7	Temporäre Zwischenspeicher
R8	Temporäre Zwischenspeicher
R9	Flag für komplett gefüllte Bank
R14R15	Dynamisch für aktuelle Bank

Tabelle 4.3: Registernutzung

```
; input.asm
; add & sub with keyboard
init:
 FIM R0R1, 0      ; rom 0
 FIM R2R3, 1      ; rom 1
 FIM RERF, 0      ; ram 0, bank 0
 LDM 8
 XCH R5           ; store input filter
begin:
 LDM 0
 XCH R8
 SRC 1
pull:
 RDR
 JCN NZ, checkNew ; key down
 LD R6
 JCN Z, pull
 LDM 0            ; key released
 XCH R6
 JUN pull
checkNew:
 LD R6
 JCN NZ, pull     ; no new key
 LDM 1
 XCH R6
```

Der Anfang des Programms stellt eigentlich keine Neuerung dar, die statischen Register werden initialisiert und R8 gelöscht, danach wird ab 'pull' der Input an

ROM 1 abgefragt (RDR). Das ist nötig, weil der Prozessor noch nicht über eine Interrupt-Steuerung verfügt. Es wird eine gedrückte Taste bei Ungleichheit von 0 registriert, da nach der Tabelle genau dann immer ACK gesetzt ist. Die Abfrageschleife folgt dem Prinzip, das vorher bei der Reaktion auf den TEST-Eingang vorgestellt wurde.

```
execKey:
 RDR
 XCH R4
 LD R5              ; copy filter
 XCH R7
 LD R4
 SUB R7             ; remove ACK
 LD R7
 KBP
 XCH R7
loop:
 LD R7
 JCN Z, getDigit
 DAC
 CLC
 XCH R7
 LDM 4
 ADD R8
 JUN loop
```

Wenn eine Taste neu gedrückt wurde, wird als erstes der Input in Register R4 abgespeichert und dann das ACK-Signal heraus gefiltert (SUB R7). Mit KBP wird die dann noch aktive Leitung Q_4 – Q_6 auf eine binäre Zahl kodiert. Dieses Ergebnis gibt die Anzahl an, wie oft in der Schleife 'loop' immer 4 auf R7 addiert wird. Dieses Vorgehen lässt sich leicht aus der Tabelle ablesen, weil die Tasten in Klassen sortiert sind.

```
getDigit:
 SRC 0
 RDR
 KBP
 ADD R8
 SRC 7
 LD R8             ; copy digit
 XCH R7
 LDM 9
 SUB R7
 JCN NC, specKey   ; special key?
 LD R9
 JCN NZ, begin     ; bank full?
 LD R8             ; write digit
 WRM
 INC RF            ; next address
 TCC               ; store carry
 XCH R9
 JUN begin
```

In 'getDigit' wird der Eingang an ROM 0 eingelesen und nach KBP auf den Summand von der Schleife 'loop' addiert, man erhält den Keycode. Dieser Keycode wird von 0x9 abgezogen und eine negative Zahl zeigt den Druck einer speziellen Taste an (JCN NC, specKey). Ansonsten liegt 0 – 9 vor, und es wird geprüft, ob die Bank schon voll ist. Wenn dies zutrifft, wird nach 'begin' gesprungen und auf eine neue Taste gewartet. Interpretation: Jetzt wird nur noch bei einer speziellen Taste gewartet. Wenn die Bank noch nicht voll ist, wird die Zahl im RAM abgespeichert und der Adresszeiger erhöht (INC RF). Wenn bei dieser Erhöhung ein Überlauf auftritt ist die Bank voll, und deshalb wird das Carry in R9 gespeichert. Danach wird bei 'begin' auf den nächsten Tastendruck gewartet.

```
specKey:
 LD R7
 CMA
 IAC
 XCH 7
 LD RF            ; write length
 WR0
 JMS action
 JCN NZ, chBank   ; add - next bank
 JMS action
 JCN NZ, opSub    ; sub
 JMS action
 JCN NZ, calc     ; calculate?
 JMS action
 JCN NZ, clear    ; clear bank?
 JMS action
 JCN NZ, reset    ; clear all?
action:
 LD R7
 DAC
 JCN Z, found
 XCH R7
 BBL 0
found:
 BBL 1
```

Durch eine negative Zahl wurde ja eine spezielle Taste ermittelt. In 'specKey' wird diese negative Zahl komplementiert wie im einfachen Subtraktions-Beispiel. Da die Eingabe des Operanden hiermit abgeschlossen ist, wird seine Länge in das Status-Register geschrieben (WR0). Dann wird im Unterprogramm 'action' die komplementierte Zahl um 1 verringert und falls der Akku jetzt eine 0 enthält (JCN Z, found) mittels BBL eine 1 zurückgeliefert, sonst 0. Im Hauptprogramm wird immer wieder 'action' aufgerufen und im Erfolgsfall (JCN NZ) in das zu dieser Taste passende Unterprogramm verzweigt. In höheren Programmiersprache entspricht dies einem 'if' – 'else if' – 'else if'.

```
chBank:
 LD RE
 JCN NZ, begin
 LD R7            ; store operation
 WR2
 FIM RERF, 1
 JUN begin
opSub:
 LDM 2            ; new op-Code
 XCH R7
 JUN chBank       ; next bank
calc:
 LD RE
 JCN Z, begin     ; only one operand
 LDM 2
 XCH RE
 JMS clBank
 LDM 0
 XCH RF
 JMS 256
 JUN reset
```

Wenn wir die richtige Funktion ermittelt haben, steht in R7 noch eine 1, dies ist der op-code für eine Addition und für diesen Fall brauchen wir nur 'chBank' aufrufen. Bei 'chBank' wird überprüft, ob wir uns in Bank 0 befinden (LD RE) und im Erfolgsfall der Inhalt von R7 als Operator vermerkt (WR2). Danach wird auf Bank 1 umgeschaltet (FIM RERF, 1) und es kann der zweite Operand eingegeben werden. Wenn wir den zweiten Operanden schon eingeben wird gleich an den Anfang gesprungen, es wird also auf ein '=' gewartet.

'opSub' sorgt dafür, dass vorher eine 2 als op-code für Subtraktion in R7 abgelegt wird und erst dann 'chBank' ausgeführt wird.

Die Berechnung des Ergebnisses wird durch 'calc' angestoßen. Zuerst wird sichergestellt, dass der zweite Operand eingegeben wurde (LD RE), sonst wird wieder an den Anfang gesprungen. Dann stellen wir den Adresszeiger auf Bank 2 ein (LDM 2) und löschen ein altes Ergebnis mit 'clBank'. Danach wird der Adresszeiger in Bank 2 wieder auf 0 gesetzt und mit JMS 256 in die erste Zeile eines Programms in ROM 1 gesprungen. Nach der Berechnung erfolgt ein 'reset'.

```
clear:
 JMS clBank
 LDM 0
 XCH RF
 JUN begin
reset:
 LDM 0            ; bank 0
 XCH RE
 JMS clBank
 INC RE           ; bank 1
 JMS clBank
 JUN init
clBank:
 SRC 7
 RD0              ; read length
 JCN NZ doClear
 BBL 0            ; nothing to do
doClear:
 XCH R7
 LDM 0            ; reset address
 XCH RF
loop2:
 SRC 7
 LDM 0            ; clear digits
 WRM
 LD R7
 DAC
 JCN Z, endClear
 XCH R7
 INC RF
 JUN loop2
endClear
 LDM 0            ; clear length
 WR0
 WR1
 WR2
 BBL 0
```

Am Ende erfolgt das Löschen der Bänke. 'clear' löscht die momentan eingestellte Bank, 'reset' stellt nacheinander Bank 0 und Bank 1 ein und löscht so beide Operanden. 'clBank' führt das eigentliche Löschen aus. Über RD0 wird die Länge der Zahl gelesen. Wenn sie 0 ist, wird gleich zurückgesprungen, sonst wird bei 0 anfangend in einer Schleife nacheinander an jeder Adresse eine 0 eingetragen. Die Schleife hat so viele Durchläufe, wie die Länge der Zahl angegeben hat. Danach werden noch alle Status-Character gelöscht und das Unterprogramm verlassen.

```
; calc.asm
init:
 LDM 0
 XCH R7          ; clear index
 LDM 0
 XCH R8          ; clear sign
 SRC 0
 RD2
 DAC
 JCN NZ, startSub
```

Wenn man in das Unterprogramm zur Addition und Subtraktion verzweigt, wird nach Initialisierung der op-code aus Bank 0 gelesen (RD2) und in die entsprechende Routine verzweigt. Das Programm stellt eine Vermischung der Beispiele aus dem Kapitel für RAM-Operationen dar. Deshalb wurden einige Programmteile nochmals in Subroutinen gekapselt und ein paar Labels und Registernummern angepasst.

Dieses Beispiel verdeutlicht den möglichen Einsatzzweck eines MCS 4-Systems schon etwas besser, auch wenn man noch keine negativen Zahlen eingeben kann und noch keine Ausgabe am RAM erfolgt.

Für die Beispiele gibt es Hades-Designs, die in Anhang B erläutert sind, und einige davon ebenso wie eine Output-Demonstration, die RAM-Inhalte über Sieben-Segment-Decoder anzeigt, wird es auf der Homepage des HADES-Projekts geben [Hades].

5 Die Implementation

Als Grundlage für eine Visualisierung der Komponenten eines MCS 4-Systems wurde eine Einbindung in das HADES-Framework gewählt, das im Informatik-Arbeitsbereich TAMS der Universität Hamburg entwickelt wird. Das Framework wurde in Java programmiert und so sind auch die Klassen der MCS 4-Komponenten in Java implementiert worden. Die Programmiersprache Java ist nahezu Plattform-unabhängig, d.h. man braucht auf seinem Computer nur eine so genannte Java Virtual Machine (JVM) um Java-Programme auszuführen. JVMs sind in JREs (Java Runtime Environments) enthalten, die für fast alle gängigen Systeme (z.B. Linux, Windows, Solaris, Apple) verfügbar sind, und ermöglichen eine Ausführung als Stand-Alone Programme, oder sie sind als Browser Add-Ins verfügbar um Java-Applets anzuzeigen. Diese Startarten müssen bei der Programmierung berücksichtigt werden, und so ist das HADES-Framework auch entwickelt worden. Viele Beispiele werden auf der Homepage des HADES-Projektes [Hades] angeboten, so auch die hier beschriebenen. Da in diesem Kapitel Grundkenntnisse von Java nützlich sind, ist ein Blick in [Krü02] zu empfehlen, das im Internet sogar kostenlos einsehbar ist.

5.1 Das HADES-Framework

Durch das HADES-Projekt wurde ein Programm geschaffen, dass es ermöglicht, digitale Schaltungen am Bildschirm zu designen und zu simulieren. Das geht auf der untersten Ebene, indem man logische Gatter verknüpft, bis zur Verschaltung komplexer ICs. Das Kernstück von HADES bildet der Editor, in dem die Komponenten platziert und mit Signalen verbunden werden.

Es gibt unter [Tut02] ein Tutorial für eine ausführliche Einführung zur Benutzung des Editors und unter [Arc03] Archive mit den HADES-Klassen und deren Dokumentation.

Am unteren Bildschirmrand gibt es eine Leiste, mit der die Simulation gesteuert wird. Sowohl für das Entwerfen als auch für die Simulation wird das gleiche Fenster benutzt, was sehr praktisch ist. So kann man während die Simulation läuft

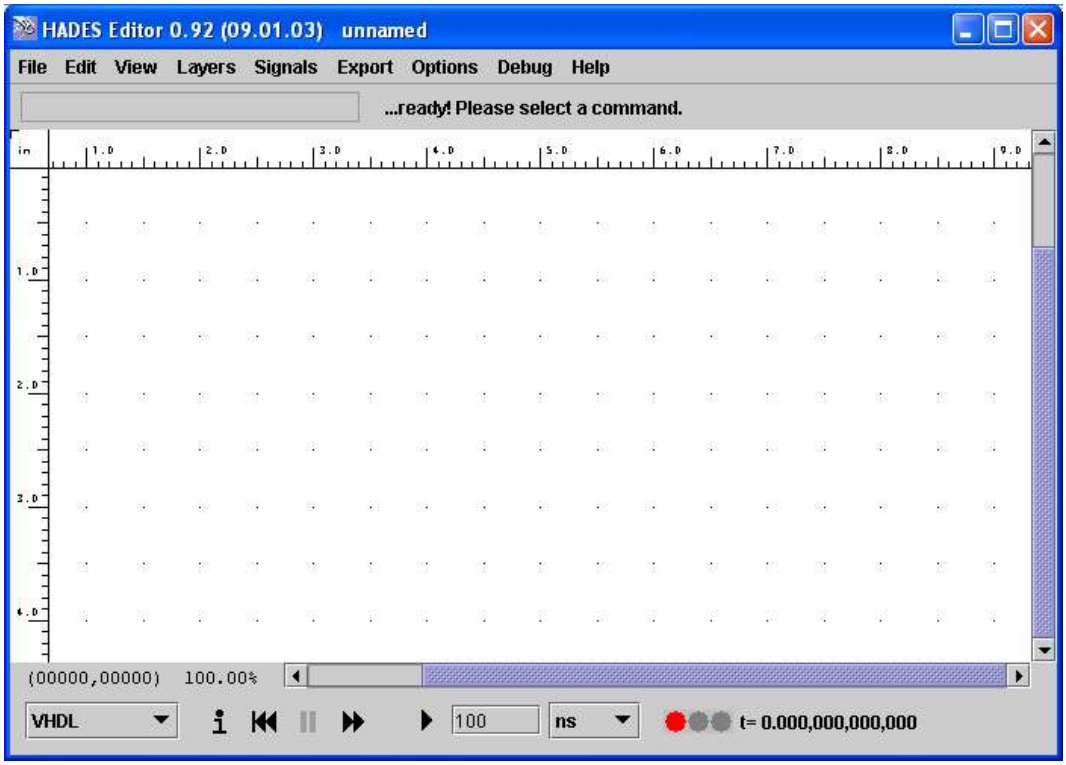


Abbildung 5.1: Die HADES-Oberfläche

oder in einer Pause das Design verändern und die Auswirkungen sofort sehen. Das wird unterstützt durch den so genannten 'glow'-Modus, in dem die Signale ihren Pegel farbig anzeigen. Außerdem können die Komponenten so entwickelt werden, dass sie Teile ihres Inhaltes (z.B. Registerbelegungen) als Text darstellen. Zusätzlich lassen sich Diagnosepunkte definieren (z.B. I/O-Pins an Komponenten), deren zeitlicher Pegel-Verlauf in einem Waveform-Viewer betrachtet werden kann.

Die so erstellten Schaltungen werden in HADES als Designs bezeichnet und als *.hds Dateien abgespeichert. Man hat die Möglichkeit, bestehende Designs als Komponente in ein neues Design zu übernehmen. So entstehen Hierarchien von Designs, in denen sich auch navigieren lässt. Exemplarisch dafür ist die Eingabelogik im Beispiel zu den I/O-Operationen. Oder man schreibt neue Java-Klassen, die das Verhalten der Schaltung im Programmcode beinhalten. Diese Form wurde für alle Komponenten des MCS 4-Systems gewählt. Für die grafische Darstellung werden in HADES Symbol-Dateien (*.sym) benötigt. Für eigene

Komponenten müssen diese selbst erstellt werden, für Subdesigns ist der Designer in der Lage, eine Symbol-Datei generisch zu erstellen.
Wenn man Chips einsetzt, die Daten speichern (RAMs und ROMs), gibt es für die Darstellung des Speicherinhalts schon vorgefertigte Klassen, die den Inhalt hexadezimal oder sogar disassembliert darstellen. Damit man (z.B. für ROMs) nicht nach jedem Laden der Designs die Daten von Hand eintippen muss, kann man *.rom-Files anlegen, die in Textform die Inhalte an jeder Adresse enthalten. Da in solchen Dateien das Kopieren, Enfügen und Löschen von Zeilen kompliziert ist, nützt der Einsatz eines Assemblers, der aus Assembler-Programmen *.rom-Dateien erzeugt. Für die ersten Durchläufe der Programme dieses Projekts wurde der Makroassembler von Alfred Arnold [Arn99] verwendet, später wurde dann ein Java-Assembler eigens für den 4004 (MCS4Assembler) entwickelt.

5.2 Die Klassenstruktur

Die Klassen, die die zu simulierende Komponenten in HADES darstellen, müssen vom Typ 'SimObject' sein, damit der Designer sie in seine Simulation aufnimmt und ihnen geeignete Nachrichten während der Simulation zukommen lassen kann.
Die wichtigste Schnittstellen für diese Nachrichten sind folgende Methoden:

- `elaborate (Object arg)` – Diese Methode wird immer bei Start einer Simulation aufgerufen und kann daher dazu benutzt werden, die Komponente in einen definierten Anfangszustand zu bringen, der bei Simulationsstart eingenommen werden soll.
- `evaluate (Object arg)` – Wird jedesmal dann aufgerufen, wenn sich an den Ports zu dieser Komponente die Pegel ändern. So kann auf die eintreffenden Signale geeignet reagiert werden. Wenn Daten an den Pins ausgegeben werden sollen, ist die Methode `getSimulator().scheduleEvent(SimEvent1164.createNewSimEvent(Simulatable target, double time, StdLogic1164 arg, Object source))` zu benutzen, die dem Simulator Signaländerungen mitteilt und der dann die weitere Steuerung übernimmt.
- `configure ()` – Wenn man mehr als nur den Namen der Komponente über den 'edit'-Befehl ändern lassen möchte, sollte man diese Methode über-

schreiben. Hier kann man ein `hades.gui.PropertySheet` mit einer Attributliste oder einer `java.beans.SimpleBeanInfo`-Klasse initialisieren. Für jedes in dieser Liste oder Klasse aufgeführten Attribute braucht man dann auch Bean-konforme 'get'- und 'set'-Methoden.

- `write(PrintWriter ps)` – Diese Methode wird aufgerufen, wenn die Komponenten beim Speichern in ein Design-File (*.hds) eingetragen werden. So kann man weitere Eigenschaften bzw. Attribute speichern und muss sie nicht bei jedem erneuten Einlesen neu konfigurieren. Diese Methoden wurden von den MCS 4-Komponenten genutzt, um die I/O-Ports der ROMs und deren *.rom-Files (der Speicherinhalt) einzutragen und die Chip-Typen der RAMs zu merken.
- `initialize(String s)` – An diese Methoden wird über den String 's' eine Zeichenkette übergeben, in der die mit 'write' abgespeicherten Attribute wieder den entsprechenden Membern bei Initialisierung der Klassen zuzuweisen. Hilfreich hierfür ist ein `java.util.StringTokenizer`, um einen String in Teile zu zerlegen.

Diese Klassen liegen alle in Unterpackages von `hades.models` und für dieses Projekt wurde das Package `hades.models.mcs4` gewählt.

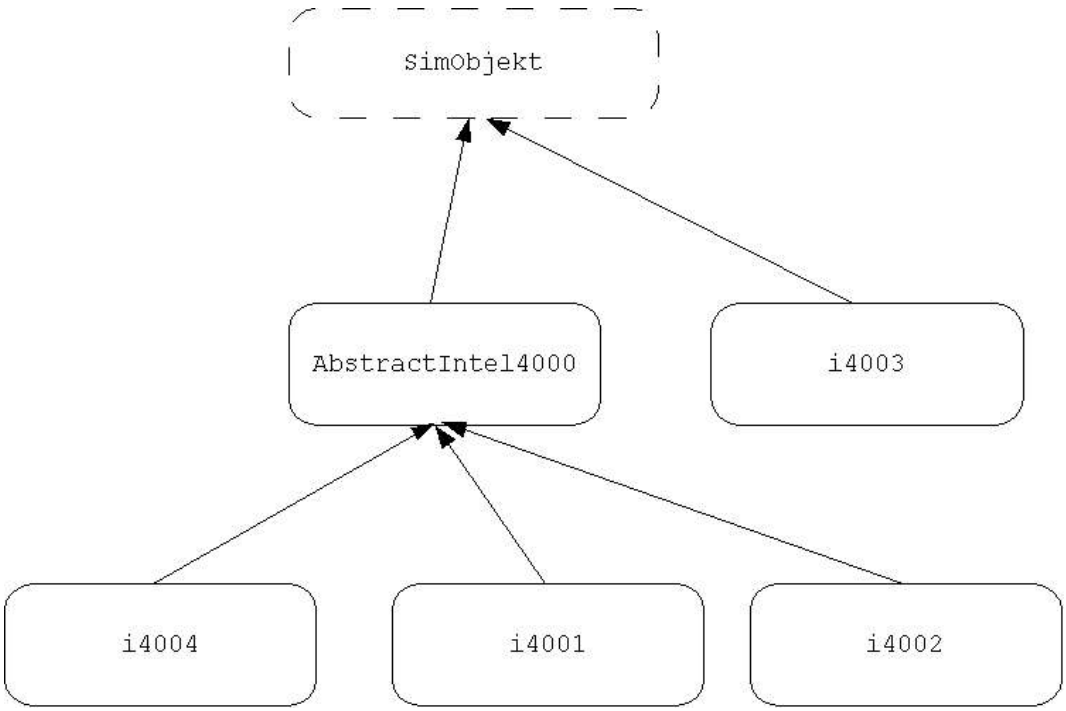


Abbildung 5.2: Verebungsdiagramm der SimObjects des MCS 4-Projekts

In diesem Vererbungsdiagramm werden die Klassen dargestellt, die sich in einem Design einbinden lassen. Die abstrakte **AbstractIntel4000-Klasse** wurde eingeführt, weil 4001, 4002 und 4004 viele Gemeinsamkeiten haben. Das betrifft die Steuerung der Subzyklen, die Schnittstelle zum Datenbus, die Stromversorgung, den Reset-Eingang und die Taktung. Diese Klasse untersucht in der **'evaluate'**-Methode die Taktsignale und ruft bei Eintritt in einen Subzyklus (phi2 ist steigend) die Methode `stateChanged(InternalState state)` an den Unterklassen auf. Hier erfolgt dann Chip-spezifisch die weitere Verarbeitung. Falls dieser Chip Daten auf den Bus gesendet hat, wird nach der 'OnHold'-Time der Bus über `releaseBus()` wieder freigegeben (Pins geben ein floating Signal). Wenn phi2 fällt, werden über `receiveData()` die Daten vom Bus gelesen. Wenn die Signale an den Bus-Pins 'floating' sind, wird mittels `hasOutputData()` abgefragt, ob Daten gesendet werden sollen und diese Daten dann nach `getOutputData()` auf den Bus gelegt. Dafür wird die Methode `sendData (StdLogicVector data)` aufgerufen. Eine **'elaborate'**-Methode besitzen alle Klassen und die Klassen i4001 / i4002 / i4004 führen einen 'super'-Aufruf aus.

Da der 4003 eine ganz andere Pinbelegung besitzt und stateless ist, ist er direkte Unterklasse von 'SimObject'.

Hier nun eine Auflistung der weiteren Hilfsklassen:

- Die Funktionseinheiten des 4004 haben eigene Klassen bekommen, sie haben alle eine 'elaborate'-Methode, die vom 4004 aufgerufen wird.

AddressStack	Der ProgramCounter und die Address-Level
AluRegion	Der gesamte ALU-Bereich mit Akkumulator, Temp.Reg, Carry-Flag, DecimalAdjust und ALU
ExecutionUnit	Das Steuerwerk des 4004 mit Instruktionsregister
ScratchPad	Die 16 internen Register

- Befehlssatz

Instruction	Implementation eines Befehls, jeder einzelne Befehl stellt eine Instanz dieser Klasse dar.
InstructionSet	Verwaltet alle Befehle und führt ein Mapping vom op-Code auf die zugehörige Befehlsklasse durch

- Assembler

MCS4Assembler	Kann direkt gestartet werden und erwartet als Parameter die zu assemblierende *.asm-Datei und erzeugt dazugehörige *.rom-, *.hex-, *.sym- und *.lbl-Dateien.

- Konfigurationsdialoge

i4001BeanInfo	Unterklasse von `java.beans.SimpleBeanInfo`. Beschreibt die Attribute des 4001 für den Property-Editor von HADES
i4001Decoder	Unterklasse von `hades.models.mcore.DcoreDecoder`. Dient zur Darstellung des Speicherinhalts als Mnemonics
i4001EditorFrame	Unterklasse von `hades.models.mcore.DcoreDisassemblerEditorFrame`. Stellt den Speicherinhalt des 4001 dar und nutzt zur Disassemblierung i4001Decoder
i4001PortTypeProperty Editor	Konstantendeklaration um die Ports des 4001 in 'In' und 'Out' zu unterscheiden
i4002BeanInfo	Unterklasse von `java.beans.SimpleBeanInfo`. Beschreibt die Attribute des 4002 für den Property-Editor von HADES
i4002ChipTypeProperty Editor	Konstantendeklaration um die verschieden Chip-Typen des 4002 in '4002-1' und '4002-2' zu unterscheiden
i4002EditorFrame	Unterklasse von `javax.swing.JDialog`. Stellt die Speicherbänke des 4002 dar
i4004EditorFrame	Unterklasse von `javax.swing.JDialog`. Stellt die internen Register des 4004 dar
i4004InternalReg	`Unterklasse von hades.models.rtlib.memory.RAM` für die internen Register des 4004.
MemoryWrapper	Diese Wrapperklasse wird vom i4002EditorFrame und i4004EditorFrame genutzt, um die RAM-Komponenten (Speicherbänke und interne Register) anzuzeigen.

Hier die sich daraus ergebenden Dialoge:

- 4001 (mit bcd_add.asm), einmal in Hex-Darstellung und einmal in der disassemblierten Darstellung

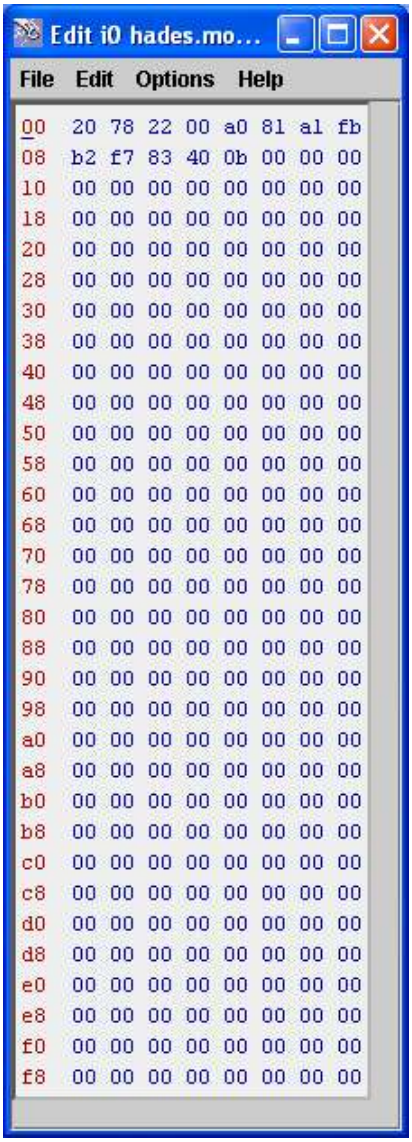


Abbildung 5.3: Hex-Darstellung des 4001

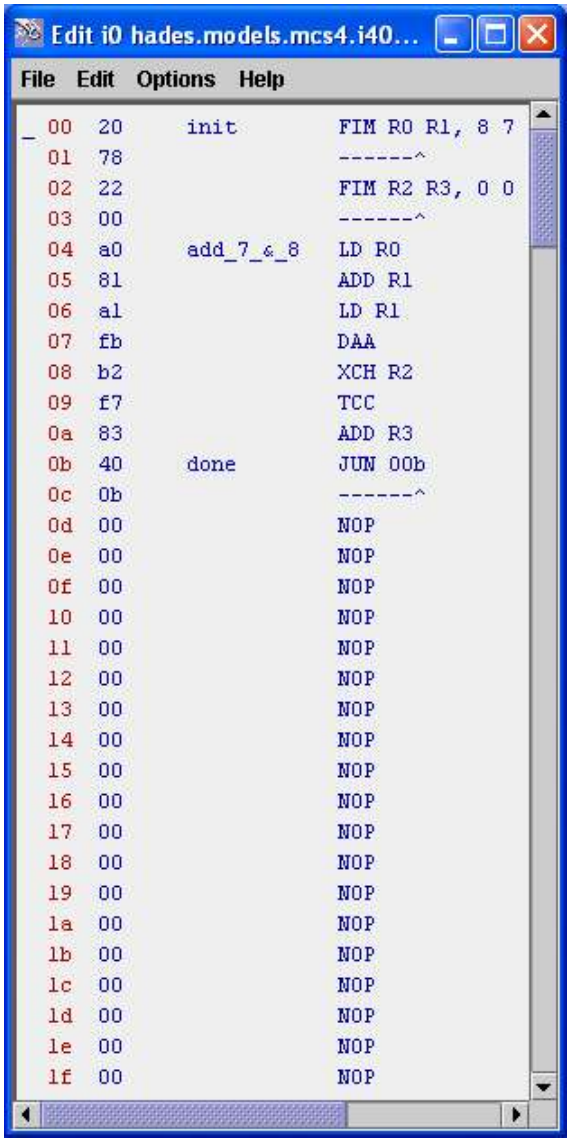


Abbildung 5.4: Disassembler-Darstellung des 4001

- 4002 und 4004

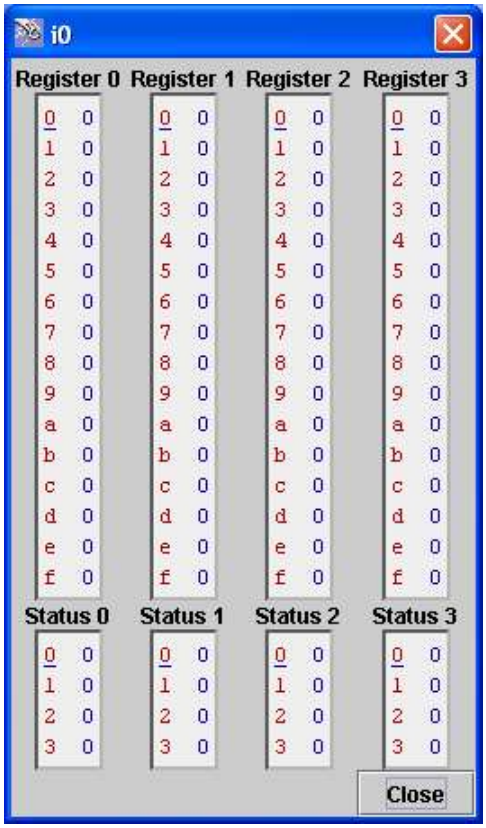


Abbildung 5.6: 4002-Darstellung

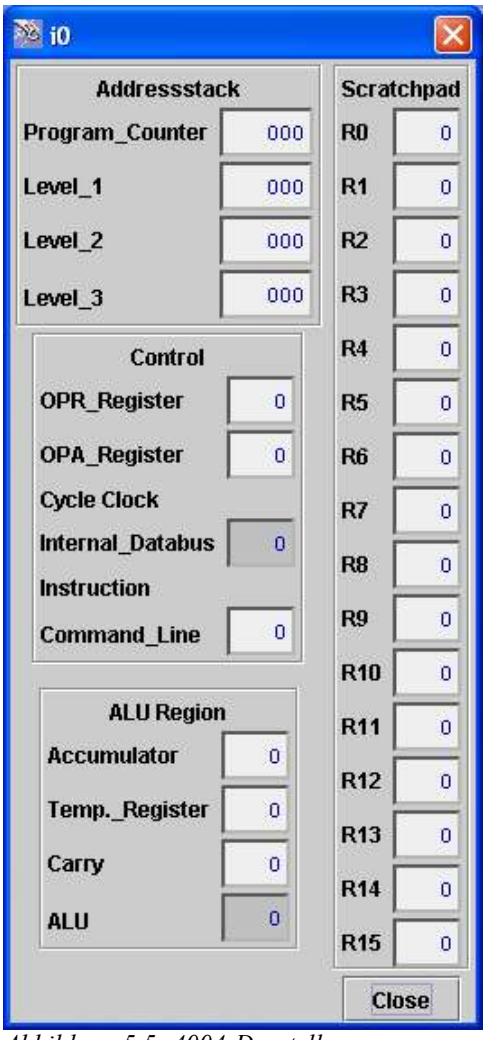


Abbildung 5.5: 4004-Darstellung

Beim 4004 ist es dadurch möglich, zu Testzwecken fast alle Inhalte 'on-the-fly' zu manipulieren. Vorsicht sei nur angeraten, wenn man das OPR- / OPA-Register verändert, da diese steuern, ob es sich um einen 2-Wort-Befehl handelt oder nicht und während aller Subzyklen der Ausführungsphase abgefragt werden. Es könnte dann kurzfristig zu einem Verhalten kommen, das nicht vorhersehbar war.

6 Literaturverzeichnis

[Arc03] HADES-Dateien. http://tech-www.informatik.uni-hamburg.de/applets/hades/archive/, Abruf am 20.08.2003.

[Arn99] Makroassembler AS. http://john.ccac.rwth-aachen.de:8000/as/index.html, Abruf am 24.08.2003.

[Bus] The agreement of Intel & Busicom. http://www.busicom-corp.com/intel1.html, Abruf am 06.08.2003.

[Dud01] Volker Klaus, Andreas Schwill: Duden Informatik. Bibliographisches Institut, 2001.

[Fag96] Faggin, Federico; Hoff Jr., Marcian E.; Mazor, Stanley; Shima, Masatoshi: The History of the 4004. In: IEEE Micro (Vol. 16, No.6). 1996, S. 10-20.

[Hades] HADES. http://tech-www.informatik.uni-hamburg.de/applets/hades/html/hades.html, Letzter Abruf am 20.08.2003.

[Int77] Intel Data Book. http://www.alikat.org/library/old/i4004-77.pdf, Abruf am 06.03.2003.

[Krü02] Handbuch der Java-Programmierung. http://www.javabuch.de, Abruf am 13.07.2003.

[Roh01] Rohde Joachim: Assembler GE-PACKT. mitp, 2001.

[Tut02] HADES-Tutorial. http://tech-www.informatik.uni-hamburg.de/applets/hades/archive/tutorial.pdf, Abruf am 20.08.2003.

7 Abbildungsverzeichnis

8 Tabellenverzeichnis

Anhang A – Befehlsreferenz

Die Tabelle enthält folgende Felder:

Beschreibung		HexCode	Mnemonic	OPR $D_3D_2D_1D_0$		OPA $D_3D_2D_1D_0$
Erläuterung						RTL
RTL A_1	RTL A_2	RTL M_1	RTL M_2	RTL X_1	RTL X_2	RTL X_3

Wenn in einer Operation ein Registerpaar RRR benutzt wird, betrifft dies die Indexregister 2*RRR und 2*RRR+1.

In der dritten Zeile werden die einzelnen Operationen und Datentransfers innerhalb des 4004 angegeben, die sich während der Implementierung des Prozessors ergeben haben. Sie geben Auskunft darüber, welche Daten während der Phase auf dem internen Datenbus liegen. Über die ursprüngliche Implementierung in der Hardware ließen sich keine detaillierten Angaben finden, deshalb lässt sich keine Aussage zu der Korrektheit der Implementation treffen. Das bedeutet, dass die Konsistenz zum historischen Prozessor in der Ausführung intern nicht auf Phasenebene sondern auf Ebene der Instruktionen beginnt. Nach Abarbeitung einer Instruktion sind die Inhalte der Register korrekt und die Daten auf dem externen Datenbus in jeder Phase ebenfalls.

Die Bezeichnung der Register in den Registertransfers ist folgende:

- A Akkumulator
- C Carry-Flag
- CL Command-Line Register
- DB Databus Buffer
- M Daten von/zu externen Speicherchips
- OPA Instruktionsregister mit OPA-Wert
- OPR Instruktionsregister mit OPR-Wert
- PC Programcounter mit folgenden Bereichen:
  H-PC: Registerbereich für High-Adress
  M-PC: Registerbereich für Middle-Adress
  L-PC: Registerbereich für Low-Adress

- R Index Register
- T Temp-Register
- S Stack-Register

Basic Instructions

No operation	00	NOP	0000	0000
Diese Anweisung löst keine Verarbeitung aus				-

Jump conditional	1* **	JCN	0001 $A_2A_2A_2A_2$	$C_1C_2C_3C_4$ $A_1A_1A_1A_1$
Wenn die Bedingung $C_1C_2C_3C_4$ wahr ist, wird zur Adresse $A_2A_2A_2A_2A_1A_1A_1A_1$ gesprungen. Die Bedingungsbits haben folgende Bedeutung: C_1 =Not, C_2 =Accumulator zero, C_3 =Carry, C_4 =Test				PC=A_2A_1

				C=0 M-PC=OPR	T=0xf L-PC=OPA	C=0

Für die Überprüfung der Bedingung wird eine condition mask angelegt. Im ersten Instruktionszyklus wird in X_1 zuerst das not-Bit, das aktuelle Carry-Flag und das Signal an dem Test-Port abgefragt und in der condition mask abgelegt, nach dem Transfer von 0xf in das Temp-Register wird noch ein A+T durchgeführt, das für A ungleich 0 das Carry-Flag setzt. Dieses wird in X_2 abgefragt und ebenfalls in die condition mask eingepflegt. Danach wird die condition mask bitweise mit dem Inhalt von OPA und-verknüpft. Im zweiten Instruktionszyklus wird die neue condition mask nach einem Auftreten von 1 (im not-Fall von 0) getestet und im Erfolgsfall der PC modifiziert.

Fetch immediate	2* **	FIM	0010 $D_2D_2D_2D_2$	RRR**0** $D_1D_1D_1D_1$
Die Datenwörter D_2 und D_1 werden in das Registerpaar RRR geladen				RRR= D_2D_1

		R(2*OPA+1)=DB	R(2*OPA)=DB			

Fetch indirect		3*	FIN	0011		RRR**0**
Der Inhalt von Registerpaar 0 wird als Adresse ausgegeben und deren Inhalt in das Registerpaar RRR geladen						RRR= (R1R0)
DB=R0	DB=R1	R(2*OPA+1)=DB	R(2*OPA)=DB			

Jump indirect		3*	JIN	0011		RRR**1**
Der Inhalt von Registerpaar RRR wird als neue Adresse $A_2A_2A_2A_2A_1A_1A_1A_1$ verwendet						PC=RRR
				L-PC= R(2*OPA)	M-PC= R(2*OPA+1)	

Jump unconditional		4* **	JUN	0100 $A_2A_2A_2A_2$		$A_3A_3A_3A_3$ $A_1A_1A_1A_1$
Die nächste Adresse ist $A_3A_3A_3A_3A_2A_2A_2A_2A_1A_1A_1A_1$						PC=$A_3A_2A_1$
				T=OPA L-PC=OPA	M-PC=OPR	H-PC=T

Jump to subroutine		5* **	JMS	0101 $A_2A_2A_2A_2$		$A_3A_3A_3A_3$ $A_1A_1A_1A_1$
Die aktuelle Adresse wird auf den Stack geschoben und die neue Adresse wird $A_3A_3A_3A_3A_2A_2A_2A_2A_1A_1A_1A_1$						S=PC PC=$A_3A_2A_1$
				S=PC,T=OPA L-PC=OPA	M-PC=OPR	H-PC=T

Increment		6*	INC	0110		RRRR
Register RRRR wird um 1 erhöht						RRRR=RRRR+1
				A=R(OPA)	T=1	R(OPA)=A+T

Increment, skip jump if zero	7* **	ISZ	0111 $A_2A_2A_2A_2$		RRRR $A_1A_1A_1A_1$
Es wird an Adresse $A_2A_2A_2A_2A_1A_1A_1A_1$ gesprungen wenn das Register RRRR nach Erhöhung um 1 ungleich 0 ist.					RRRR=RRRR+1 PC=A_2A_1
			A=R(OPA) L-PC=OPA	T=1 M-PC=OPR	R(OPA)=A+T

Das Überschreiben des PC im zweiten Instruktionszyklus wird nur durchgeführt, wenn das Carry-Flag gesetzt ist.

Add	8*	ADD	1000	RRRR
Zum Register RRRR wird der Akkumulator und Carry addiert				RRR=A+RRR
			T=R(OPA)	A=A+T

Subtract	9*	SUB	1001	RRRR
Vom Register RRRR wird der Akkumulator und Carry abgezogen				RRR=A-RRR
			T=R(OPA)	A=A-T

Load	A*	LD	1010		RRRR
Der Akkumulator wird mit dem Inhalt von Register RRRR geladen					A=RRRR
				A=R(OPA)	

Exchange	B*	XCH	1011		RRRRR
Die Inhalte von Register RRRR und dem Akkumulator werden ausgetauscht					RRR=A, A=RRR
			T=R(OPA)	R(OPA)=A	A=T

Branch back 1 level	C*	BBL	1100		DDDD
Die nächste Adresse wird vom Stack genommen und DDDD in den Akkumulator geladen					PC=S, A=D
			PC=S,A=OPA		

Load immediate	D*	LDM	1101		DDDD
Es wird DDDD in den Akkumulator geladen					A=D
			A=OPA		

I/O und RAM Instructions

Send register control	2*	SRC	0010	RRR**1**
In den Registern RRR steht der Code zur Auswahl des nächsten ROMs bzw. RAMs und wird an alle gesendet. Dieser Befehl selektiert die Chips, die die nächsten Befehle (e*) bearbeiten sollen.				M=RRR
			DB= R(2*OPA)	DB= R(2*OPA+1)

Write memory	E0	WRM	1110	0000
Der Inhalt des Akkumulators wird in den Speicherbereich des zuvor ausgewählten RAMs geschrieben				M=A
			DB=A	

Write RAM port	E1	WMP	1110	0001
Der Inhalt des Akkumulators wird an die Output-Ports des zuvor ausgewählten RAMs geschrieben				M=A
			DB=A	

Write ROM port	E2	WRR	1110	0010
Der Inhalt des Akkumulators wird an die Output-Ports des zuvor ausgewählten ROMs geschrieben				M=A
			DB=A	

Write program memory	E3	WPM	1110	0011
Der Inhalt des Akkumulators wird in das ausgewählten Wort des Programspeichers (nur bei 4008/4009) geschrieben				M=A
			DB=A	

Write status character 0	E4	WR0	1110	0100
Der Inhalt des Akkumulators wird in den Status-Character 0 des betreffenden Statusregisters des zuvor ausgewählten RAMs geschrieben				M=A
			DB=A	

Write status character 1	E5	WR1	1110	0101	
Der Inhalt des Akkumulators wird in den Status-Character 1 des betreffenden Statusregisters des zuvor ausgewählten RAMs geschrieben					M=A
				DB=A	

Write status character 2	E6	WR2	1110	0110	
Der Inhalt des Akkumulators wird in den Status-Character 2 des betreffenden Statusregisters des zuvor ausgewählten RAMs geschrieben					M=A
				DB=A	

Write status character 3	E7	WR3	1110	0111	
Der Inhalt des Akkumulators wird in den Status-Character 3 des betreffenden Statusregisters des zuvor ausgewählten RAMs geschrieben					M=A
				DB=A	

Subtract memory	E8	SBM	1110	1000	
Das Wort aus dem Speicherbereich des zuvor ausgewählten RAMs wird vom Inhalt des Akkumulators mit Carry abgezogen					A=A-M
				T=DB	A=A-T

Read memory	E9	RDM	1110	1001	
Das Wort aus dem Speicherbereich des zuvor ausgewählten RAMs wird in den Akkumulator geladen					A=M
				A=DB	

Read ROM port	EA	RDR	1110	1010	
Das Wort an den Input-Ports des zuvor ausgewählten ROMs wird in den Akkumulator geladen					A=M
				A=DB	

Add memory		EB	ADM	1110	1011	
Das Wort aus dem Speicherbereich des zuvor ausgewählten RAMs wird zu dem Inhalt des Akkumulators mit Carry addiert						A=A+M
					T=DB	A=A+T

Read status character 0		EC	RD0	1110	1100	
Der Status-Character 0 aus dem zuvor gewählten RAM und dem betreffenden Statusregister wird in den Akkumulator geladen						A=M
					A=DB	

Read status character 1		ED	RD1	1110	1101	
Der Status-Character 1 aus dem zuvor gewählten RAM und dem betreffenden Statusregister wird in den Akkumulator geladen						A=M
					A=DB	

Read status character 2		EE	RD2	1110	1110	
Der Status-Character 2 aus dem zuvor gewählten RAM und dem betreffenden Statusregister wird in den Akkumulator geladen						A=M
					A=DB	

Read status character 3		EF	RD3	1110	1111	
Der Status-Character 3 aus dem zuvor gewählten RAM und dem betreffenden Statusregister wird in den Akkumulator geladen						A=M
					A=DB	

Accumulator Group Instructions

Clear both		F0	CLB	1111	0000	
Akkumulator und Carry werden gelöscht						A=0,C=0
				A=0,C=0		

Clear carry		F1	CLC	1111	0001	
Carry wird gelöscht						C=0
				C=0		

Increment accumulator	F2	IAC	1111	0010	
Der Inhalt des Akkumulators wird um 1 erhöht					A=A+1
			C=0	T=0x1	A=A+T

Complement carry	F3	CMC	1111	0011	
Das Carry-Flag wird negiert					C=!C
			C=!C		

Complement accumulator	F4	CMA	1111	0100	
Der Inhalt des Akkumulators wird durch das Komplement ersetzt					A=!A
			T=A	A=0xf	

Rotate accumulator left	F5	RAL	1111	0101	
Der Akkumulator und das Carry werden bitweise nach links rotiert					A=rotate A
			A=rotate A		

Rotate accumulator right	F6	RAR	1111	0110	
Der Akkumulators und das Carry werden bitweise nach rechts rotiert					A=rotate A
			A=rotate A		

Transmit and clear carry	F7	TCC	1111	0111	
Der Akkumulators wird mit dem Wert des Carry geladen und das Carry gelöscht					A=C, C=0
			A=C	C=0	

Decrement accumulator	F8	DAC	1111	1000	
Der Inhalt des Akkumulators wird um 1 verringert					A=A-1
			C=0	T=0x1	A=A-T

Transfer carry subtract, clear carry	F9	TCS	1111	1001	
Das Carry wird von dem gelöschten Inhalt des Akkumulators subtrahiert und anschließend gelöscht					A=0-C, C=0
			A=0,T=0	A=A-T	C=0

Set carry		FA	STC	1111		1010
Das Carry-Flag wird gesetzt						C=1
				C=1		

Decimal adjust accumulator		FB	DAA	1111		1011
Der Inhalt des Akkumulators wird zu einer BCD-Zahl korrigiert						A=decimal adjust A
				A=decimal adjust A		

Keyboard process		FC	KBP	1111		1100
Der Inhalt des Akkumulators wird als 1-von-4-Code in einen Binärwert konvertiert: 0000 => 0000, 0001 => 0001, 0010 => 0010, 0100 => 0011, 1000 => 0100, alle anderen Werte => 1111						A= modified A
				OPA=A	A=modified OPA	

Designate command line		FD	DCL	1111		1101
Das Register zur Ansteuerung der CM-RAM$_x$-Signale wird mit dem Inhalt des Akkumulators geladen. Für jede '1' wird das entsprechende CM-RAM$_x$-Signal in M_2 auf low gesetzt.						CL=A
				CL=A		

Anhang B – Die HADES-Designs zu den Anwendungsbeispielen

Die Beispiele zur Kontrollflusssteuerung und zur Arithmetik benötigen lediglich ein minimales MCS 4-System:

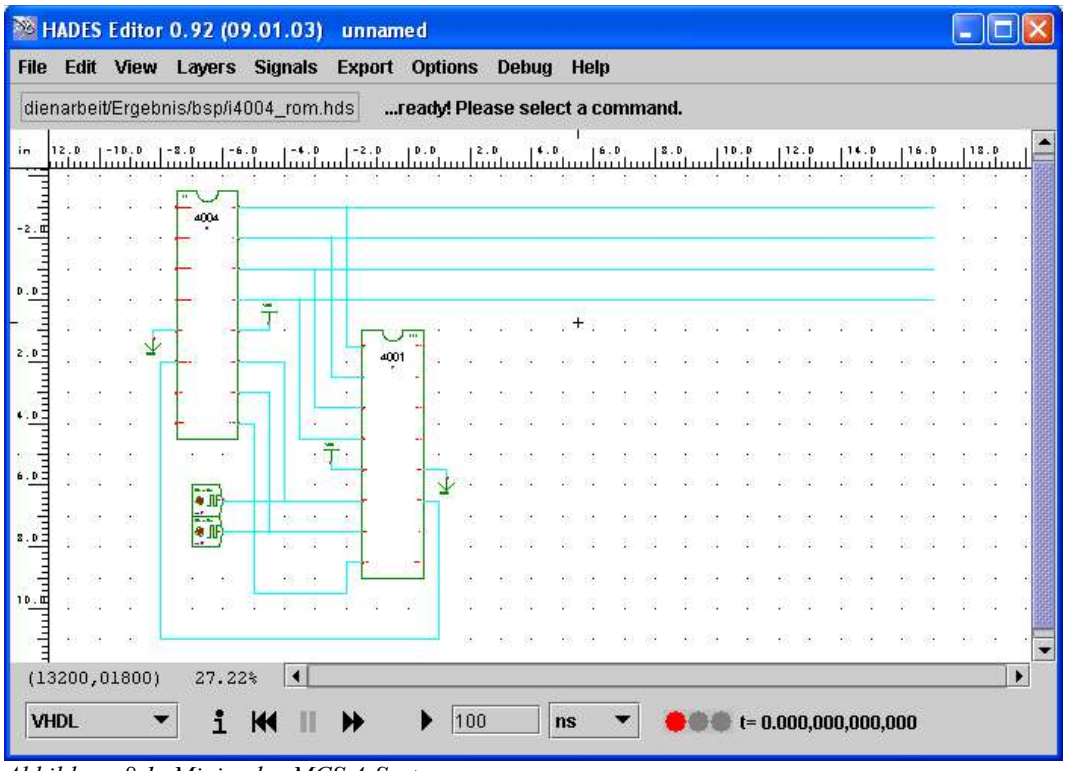


Abbildung 8.1: Minimales MCS 4-System

Oben links befindet sich der 4004. Um das Design übersichtlich zu gestalten, wurde er in der x-Achse gespiegelt.

Am oberen Rand ist der Datenbus, der die Chips verbindet und im unteren Teil sind die Steuersignale verteilt.

Für die Beispiele zur RAM-Nutzung wurde das Design um einen 4002 erweitert:

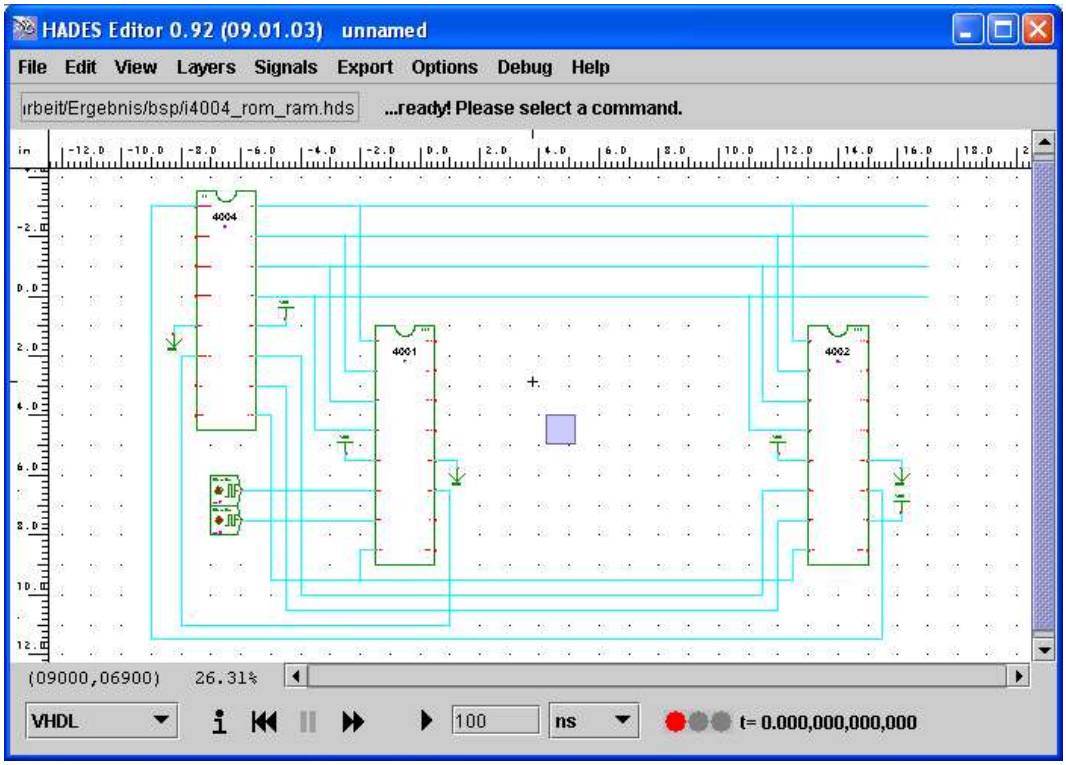


Abbildung 8.2: MCS 4-System mit RAM

Und für das komplexere Beispiel zur I/O-Steuerung benötigt man einen zweiten ROM-Chip:

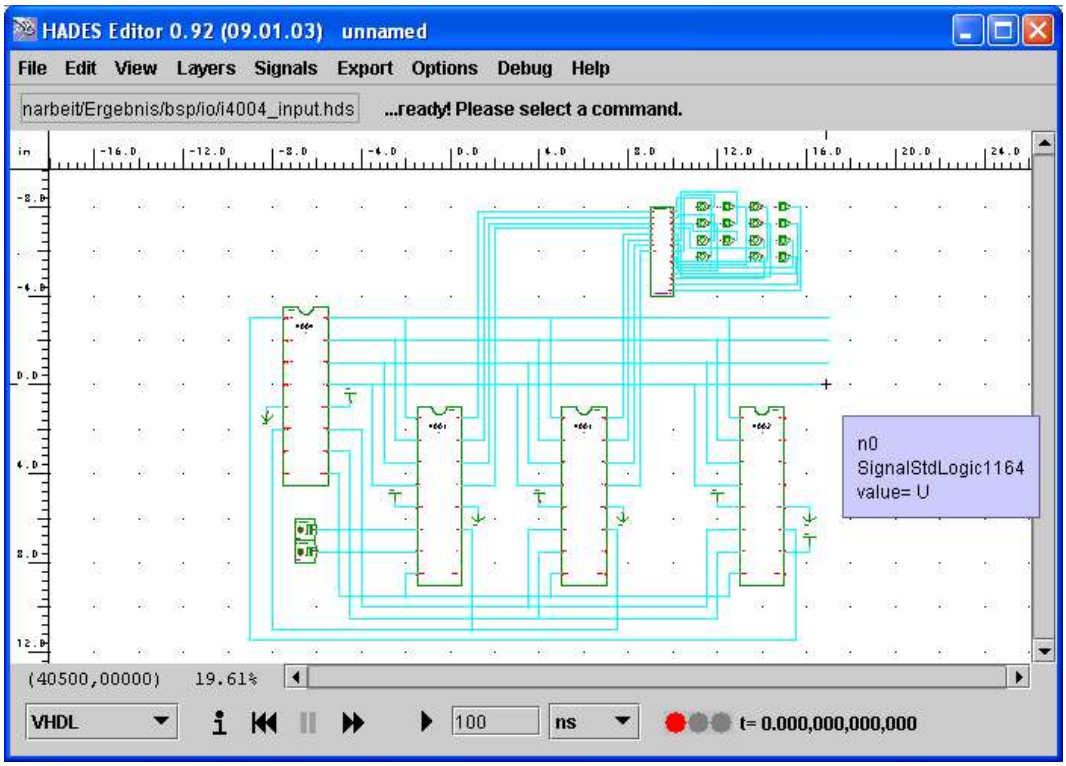


Abbildung 8.3: MCS 4-System mit Tastenfeld

Oben rechts in der Ecke wurde das Tastenfeld platziert und daneben ist der IC, der die Tastendrücke codiert und als Subdesign realisiert wurde:

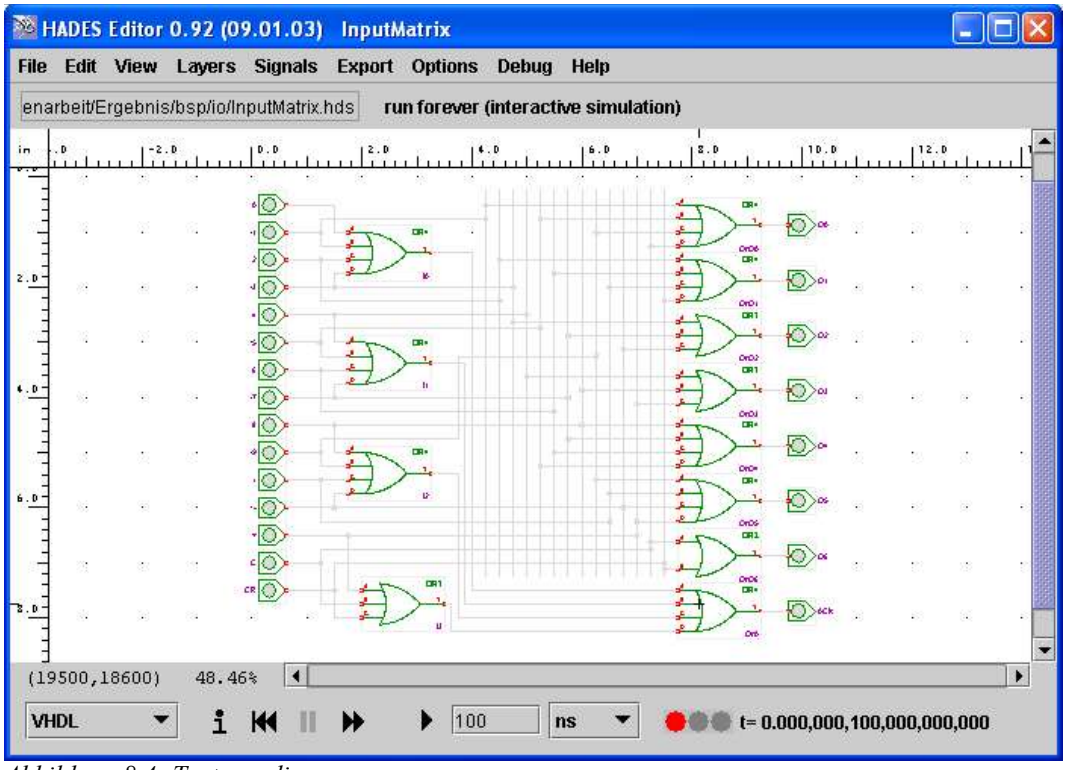


Abbildung 8.4: Tastencodierer

Auf der linken Seite sind die 15 Eingangspins, die im Hauptdesign mit den Tastern verbunden sind, und auf der rechten Seite die 8 Ausgangspins, die Signale an die I/O-Pins der ROMs senden.