# Projekt 1: Mikroprogrammierung

ERA-Praktikum

Stephan Rabanser
Fakultät für Informatik
Technische Universität München
rabanser@cs.tum.edu

14. Juli 2013

# Inhaltsverzeichnis



# Abbildungsverzeichnis



# Wichtige Dokumente

Folgende Dokumente halfen uns bei der Realisierung unseres Projektes. Sie können auch für Sie zum Verständnis oder beim Ausbau dieses Projektes nützlich sein:

- Dokumentation zur Zielarchitektur
- Beschreibung der mikroprogrammierbaren Maschine
- MI-Merkblatt
- Hinweise zur Durchführung eines Mikroprogrammierprojekts
- ERA-Leitbuch

# 1 Einführung

Dies ist die Dokumentation für das Mikroprogrammierprojekt (Projekt 1) im ERA Praktikum des Sommersemesters 2013. Es wurde sowohl eine allgemeine Anwenderdokumentation, als auch eine technische Entwicklerdokumentation erstellt. Um ein generelles Verständnis über das Projekt zu erhalten und den Einstieg zu erleichtern, empfiehlt es sich (auch für Entwickler) die Anwenderdokumentation anzusehen.

# 2 Anwenderdokumentation

## 2.1 Übersicht

Das von uns umgesetzte Mikroprogrammier-Projekt ist Teil eines Gesamtprojekts, welches die Realisierung der gewünschten Zielarchitektur zum Ziel hat. Diese wurde bereits in der zum Projektbeginn mitgelieferten Dokumentation zur Zielarchitektur genau beschrieben.

Diese Anwenderdokumentation beschreibt kurz die Befehle, welche zum implementieren waren, sodass ein Entwickler weiß, wie diese Befehle anzuwenden sind und wozu sie dienen. Auch wird darauf eingegangen, was das Testprogramm macht und wie es bedient wird. Für detailiertere technische Informationen zum Projekt wird auf die Entwicklerdokumentation verwiesen.

## 2.2 Befehle

Die folgenden fünf Befehle wurden mit den angegebenen Adressierungsarten realisiert:

- **move imm,RB**: Der Befehl kopiert den immidiate-Wert *imm* in das Register *RB*. Es handelt sich dabei um einen Datentransport-Befehl.
- **inc RB**: Der Befehl inkrementiert den Wert des Registers *RB* um 1. Dabei wird das Maschinenstatusregister nicht verändert. Es handelt sich dabei um einen arithmetischen Befehl.
- **bscanf [RA],RB**: Der Befehl sucht in der von *RA* adressierten Speicherzelle nach dem höchstwertig gesetzten Bit und schreibt dessen Bitposition in *RB*. Falls in der gesamten Zelle kein 1-Bit vorhanden ist, so wird -1 abgespeichert. Es handelt sich dabei um einen Bitfeld-Befehl.
- **cmp imm,RB**: Der Befehlt subtrahiert den immidiate-Wert *imm* von *RB* und setzt die Statusflags wie folgt:
  - C = 1 $\Leftrightarrow$ $RB < imm$ (als vorzeichenlose Zahlen)
  - Z = 1 $\Leftrightarrow$ $RB = imm$
  - N = Bit 15 von $RB - imm$
  - OVR = 1 $\Leftrightarrow$ Zulässiger Wertebereich überschritten

Das eigentliche Ergebnis der Operation wird dabei nicht abgespeichert. Es handelt sich dabei um einen Vergleichsbefehl.

- **jumplt imm**: Der Befehl springt an den angegebenen immediate-Wert *imm*, falls das N-Flag (negativ) oder das OVR-Flag (overflow) gesetzt wurde. In der gegebenen Aufgabenstellung lautete der Befehl "jmplt", wir haben ihn dann allerdings in jumplt umbenannt. Es handelt sich dabei um einen Sprungbefehl.

Beide genannten Register (RA und RB) sind 16-Bit breit, selbes gilt für Konstanten (immediate).

## 2.3 Testprogramm

Um die Funktionsweise der implementierten Befehle zu demonstrieren, wurde zusätzlich ein Maschinenprogramm geschrieben, welches mit Hilfe der oben aufgelisteten Befehle in einem beliebig langen Bit-Feld nach dem ersten gesetzten Bit sucht.

Das besagte Programm befindet sich in der Datei "bitsucheBundProg.mpr" im Implementierungsordner. Sie enthält sowohl die von uns implementierten Mikrobefehle, als auch das Testprogramm im Hauptspeicher. Diese Datei kann dann zur besseren Veranschaulichung mit JMic geöffnet werden, wie in Abbildung 1 dargestellt.

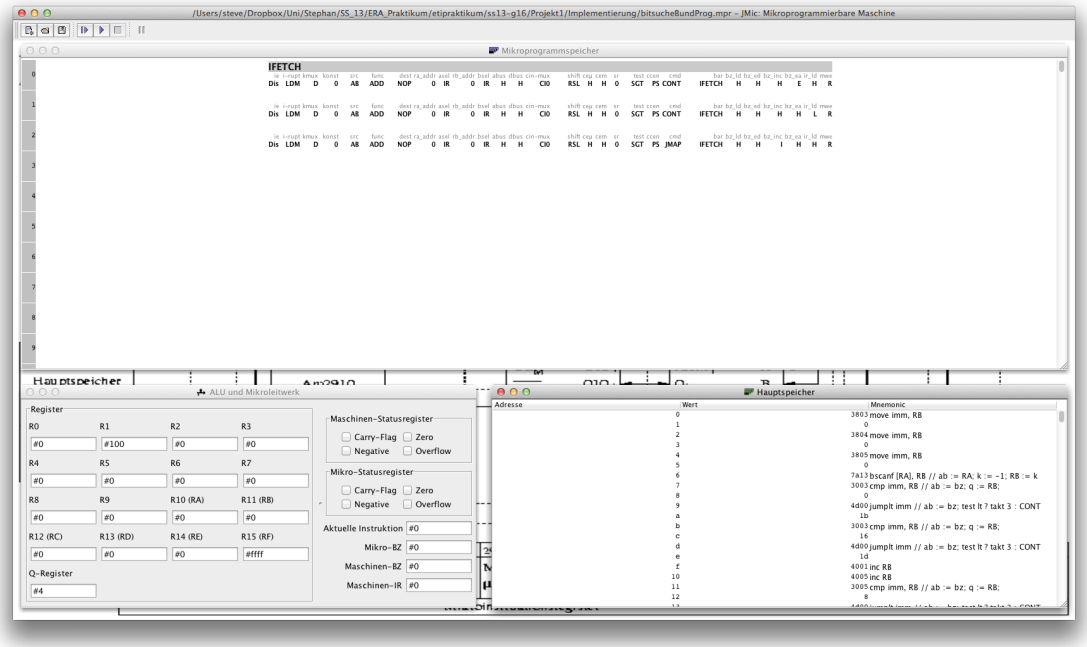


Abbildung 1: Bitsuche-Programm in JMic

Zum Schnelldurchlauf des Testprogramms muss in der Menüleiste der rechte Pfeil (ohne vorangehenden Balken) betätigt werden, der linke Pfeil (mit vorangehenden Balken) führt das

Programm Schritt für Schritt aus. Der Durchlauf kann stets mit dem Stopp-Knopf (Quadrat) gestoppt werden, oder mit dem Pause-Knopf (zwei Balken) pausiert werden.

In dem Fenster mit dem Titel "Mikroprogrammspeicher" befinden sich die von uns umgesetzten Mikroprogrammbefehle. Unter "ALU und Mikroleitwerk" sind die aktuellen Inhalte der Register, der Flags und der Zähler zu finden. Das Fenster "Hauptspeicher" gibt letztlich Auskunft über den Inhalt des Hauptspeichers.

Detailierte Informationen zu JMic lassen sich hier finden.

# 3 Entwicklerdokumentation

## 3.1 Übersicht

Diese Entwicklerdokumentation beschreibt die Implementierung der in der Anwenderdokumentation bereits aufgelisteten Befehle, sowie die Strukturierung und die genaue Funktion unseres Testprogrammes.

## 3.2 Befehle

In der Anwenderdokumentation wurden die Befehle mitsamt Adressierung und Funktion bereits erklärt. Hier wird nun die genaue Implementierung dieser Maschinenbefehle beschrieben:

- **move imm,RB**:

  1. Takt (0x380): Der Befehlszähler wird auf den Adressbus gelegt (BZ_EA auf E) und auf den Hauptspeicher wird lesend zugegriffen (MWE auf R). Somit erhalten wir den immidiate-Wert im 2. Takt, da dieser an der Speicherstelle direkt nach dem Befehl gespeichert ist. Da wir in diesem Schritt keine Rechnung durchführen müssen, wird ADD in Kombination mit NOP als Funktion angegeben. Als Quelloperandensteuerung wird AB angegeben, da wir beide übergebenen Parameter verarbeiten müssen. Wichtig ist es bei CMD CONT anzugeben, sodass die zweite Zelle unseres Befehls ausgeführt wird.
  2. Takt (0x381): Vom Datenbus wird der immidiate-Wert ohne verändert zu werden (Addition mit 0 durch ADD und RAMF) durch die ALU transportiert und in RB abgelegt. Da unsere Quelle nun der Datenbus ist, muss als Quelloperandensteuerung DZ angeben werden. Außerdem wird der Befehlszähler inkrementiert (BZ_INC auf I), da es sich bei diesem Befehl um einen 32-Bit Befehl handelt (durch immediate-Wert bedingt).

  Der Befehl wurde beginnend an der Stelle 0x380 implementiert und ist 2 Zellen lang.

- **inc RB**: Der in RB stehende Wert wird in der ALU mit 0 addiert (ADD) und das Übertragsbit CIN_MUX wird auf CI1 gesetzt. So erhalten wir die Inkrementierung des Wertes in RB um 1. Damit das Ergebnis auch abgespeichert wird, haben wir RAMF als Zielfunktion gewählt. Als Operandensteuerung wurde ZB gewählt, da wir nur einen Eingangsparameter haben. Der Befehl wurde beginnend an der Stelle 0x400 implementiert und ist 1 Zelle lang.

- **bscanf [RA],RB**:

  1. Takt (0x7a0): Zunächst wird der Inhalt von RA mittels RAMA an der ALU vorbei geschleust und auf den Adressbus gelegt (ABUS auf AB). Parallel dazu wird die Konstante -1 in RB gespeichert. KMUX muss deshalb auf K und KONST auf FFFF gesetzt werden. Da uns die Konstante über den D-Eingang erreicht, muss DZ als Quelle angegeben werden. Als Funktion wird ADD verwendet. Es muss sichergestellt werden, das zum nächsten Mikrobefehl gesprungen wird, daher muss bei CMD erneut CONT angegeben werden.
  2. Takt (0x7a1): Da wir später einen Vergleich vornehmen wollen, muss der Inhalt des Datenbuses (der Inhalt der Adresse, die in RA steht) zunächst in das Q-Register kopiert werden (von diesem Register aus sind Vergleiche möglich). Es muss sichergestellt werden, das zum nächsten Mikrobefehl gesprungen wird, daher muss bei CMD erneut CONT angegeben werden.
  3. Takt (0x7a2): Nun addieren wir den Inhalt des Q-Registers mit 0 und setzen dementsprechend die Flags des Mikrostatusregisters (SR auf MI). Dies muss gemacht werden, weil wir wissen wollen, ob der Inhalt des Q-Registers gleich 0 ist. Als Quelle wird daher ZQ angegeben und da wir eine Addition durchführen wollen wird ADD als Funktion gewählt. Das Ergebnis wollen wir nicht abspeichern, daher wird als Ziel NOP angegeben. RA_ADDR und RB_ADDR enthalten noch aus Debugginggründen den Wert F, dieser ist für die Funktionsweise des Programmes also nicht ausschlaggebend. Getestet wird, ob das Zero-Flag (Z) gesetzt wurde, daher steht Z im TEST-Feld. Es muss sichergestellt werden, das zum nächsten Mikrobefehl gesprungen wird, daher muss bei CMD erneut CONT angegeben werden.
  4. Takt (0x7a3): Der Inhalt des Q-Registers wird nun nach rechts geshiftet. Nun wird auch kontrolliert, ob das Q-Register leer ist. Falls dies der Fall ist, so wird zu IFETCH zurückgesprungen, andernfalls muss eine weitere Mikroinstruktion durchgeführt werden. Das Shiften erfolgt mit der Zielsteuerung RAMQD, als Funktion wird wieder ADD und als Quelloperandensteuerung ZQ angegeben. Hier wird nun aber RA_ADDR und RB_ADDR mit F benötigt, da der Wert erneut in das Q-Register und in den Speicher geschrieben wird. Wir überprüfen erneut das Zero-Flag (Z in TEST) und spezifizieren einen bedingungsgebundenen Sprung zu IFETCH, indem wir CCEN auf C und CMD auf CJP (conditional jump) stellen.
  5. Takt (0x7a4): Falls der 5. Takt erreicht wird, so ist der Inhalt des Q-Registers nicht 0 und wir müssen den Wert in RB inkrementieren und zurück an die Adress 0x7a2 des Mikroprogrammspeichers springen. Dies erfolgt fast gleich wie beim inc Befehl, bloß dass durch den Sprung nicht zu IFETCH, sondern zu 0x7a2 gesprungen wird.

  Der Befehl wurde beginnend an der Stelle 0x7a0 implementiert und ist 5 Zellen lang.

- **cmp imm,RB**:

  1. Takt (0x300): Der Befehlszähler wird auf den Adressbus gelegt (BZ_EA auf E) und auf den Hauptspeicher wird lesend zugegriffen (MWE auf R). Somit erhalten wir den

immidiate-Wert im 2. Takt, da dieser an der Speicherstelle direkt nach dem Befehl gespeichert ist. Der Wert in RB wird im selben Takt in das Q-Register geschrieben, um im 2. Takt den Vergleich durchführen zu können. Da wir hier nur den zweiten Parameter betrachten wird als Quelle ZB angegeben, das Kopieren des Inhaltes von RB in das Q-Register erfolgt mittels ADD QREG. Wichtig ist es hier erneut bei CMD CONT anzugeben, sodass die zweite Zelle unseres Befehls ausgeführt wird.

2. Takt (0x301): Vom Datenbus wird nun der Wert des Q-Registers abgezogen (Q-D). Dies erfolgt mittels SUBR RAMF und mit CIN_MUX auf CI1 (kein Übertrag bei Subtraktion). Das Maschinenstatusregister (enthält die für uns wichtigen Flags) wird aktualisiert, indem wir SR auf MA und TEST auf ULT (unsigned less than - vorzeichenlos kleiner gleich) setzen. Außerdem wird der Befehlszähler inkrementiert (BZ_INC auf I), da es sich bei diesem Befehl um einen 32-Bit Befehl handelt (durch immediate-Wert bedingt).

Der Befehl wurde beginnend an der Stelle 0x300 implementiert und ist 2 Zellen lang.

- **jumplt imm**: Für diesen Befehl sind nur Befehlszähler und Bedingungen relevant, daher ist nur der rechte Abschnitt einer Mikroprogrammspeicherzelle für uns wichtig. Alle drei Mikroinstruktionen haben als Quelloperandensteuerung AB, als Funktion ADD und als Ziel NOP, da nicht gerechnet wird oder Werte verschoben werden.

  1. Takt (0x4d0): Der Befehlszähler wird auf den Adressbus gelegt (BZ_EA auf E) und auf den Hauptspeicher wird lesend zugegriffen (MWE auf R). Somit erhalten wir den immidiate-Wert im 2. Takt, da dieser an der Speicherstelle direkt nach dem Befehl gespeichert ist. Nun muss die Sprungbedingung überprüft werden. Dazu setzten wir SR auf MA und TEST auf SLT. Außerdem setzten wir CCEN auf C, sodass die Bedingungsüberprüfung ausgeführt wird. Im nächsten Takt wir dann je nach Fall (N- oder OVR-Flag gesetzt oder nicht gesetzt) anders vorgegangen.
  2. Takt:
     - Fall 1 (0x4d1): Die Flags sind nicht gesetzt, wir müssen also nicht an eine andere Stelle in unserem Maschinenprogramm springen. Es muss hier lediglich der Befehlszähler inkrementiert werden (BZ_INC auf I), da es sich um einen 32-Bit Befehl handelt.
     - Fall 2 (0x4d2): Eines der Flags wurde gesetzt und somit müssen wir an eine andere Stelle springen. Diese neue Stelle ist der immediate-Wert, der am Datenbus anliegt. Wir müssen also den Befehlszähler mit dem immediate-Wert aus dem Datenbus beschreiben (BZ_LD auf L).

  Der Befehl wurde beginnend an der Stelle 0x4d0 implementiert und ist 3 Zellen lang.

Die Befehlsadressen - also die Speicherstellen im Mikroprogrammspeicher, an denen die Befehle implementiert werden sollen - lassen sich aus der bereits mitgelieferten Dokumentation der Zielarchitektur entnehmen.

Zusätzlich zu den zu implementierenden Befehlen wurde auch **IFETCH** (Intruction FETCH) in den Mikroprogrammspeicher eingefügt, sodass das Programm testbar ist. IFETCH sorgt dafür, dass neue Befehle Maschinenbefehle automatisch geholt (gefetched) werden und stellt somit einen kontinuierlichen Maschinenbefehlszyklus her. Dies funktioniert allerdings nur, wenn alle Befehle nach der Ausführung der letzten Mikroinstruktion zurück zu IFETCH springen. Dies ist bei unseren Befehlen stets der Fall. IFETCH befindet sich, wie bereits aus der ERA-Vorlesung bekannt, an Stelle 0x000 des Mikroprogrammspeichers und ist insgesamt 3 Speicherzellen lang.

## 3.3 Testprogramm

Das Maschinentestprogramm liegt in der Datei "bitsucheBundProg.mpr" zwar im Hauptspeicher vor, ist dort allerdings für einen schnellen Überblick nicht leicht zu verstehen bzw. zu lesen. Aus diesem Grund haben wir eine getrennte Datei "BitsucheTest.asm" angelegt, welche nur die Maschinenbefehle des Testprogramms enthält. Zusätzlich befindet sich im Implementierungsordner auch noch ein "Test"-Ordner, der einige Mikroprogrammtests der komplexeren Befehle enthält. Da sämtliche Befehle bereits in der Datei "bitsucheBundProg.mpr" vorhanden sind, wird auf diese Dateien hier allerdings nicht weiter eingegangen.

Um unser Testprogramm zu realisieren benötigen wir 4 Register. In R1 steht die Startadresse des zu durchsuchenden Bitfeldes, in R3 wird die höchste 1-Bit-Position im aktuell durchsuchten Byte gespeichert, in R4 soll die höchste 1-Bit-Position des gesamten Feldes stehen und R5 dient uns als Feldindex. Die Länge des Feldes wird in unserem Testprogramm als Konstante auf 8 festgelegt.

Zunächst werden R3, R4 und R5 mittels **move** mit 0 als Standardwert befüllt. Danach springt das Programm schon in den ersten Schleifendurchlauf bei *suche* ein und sucht mittels **bscanf** nach dem ersten Bit im ersten Byte unseres Feldes. Nun kann es allerdings passieren, dass das Feld leer ist, also nur aus 0en besteht. In diesem Fall erhalten wir in R3 den Wert -1. Es muss also eine Fallunterscheidung stattfinden. Dazu vergleichen wir den in R3 gespeicherten Wert durch den **cmp**-Befehl mit 0. Falls der Wert kleiner als 0 ist, gab es in dem durchsuchten Byte kein 1-Bit und es wird an die Marke *not_f* gesprungen. Falls jedoch ein 1-Bit gefunden wurde, so müssen wir ebenfalls springen, in diesem Fall allerdings zur Marke *add*.

Nun geht es darum die Gesamtposition zu setzen und genau für diesen Fall mussten wir die Fallunterscheidung durchführen. Falls wir kein 1-Bit in unserem Feld gefunden haben, so beschreiben wir R3 mit 16 (Erklärung folgt). Jetzt geht es auch in diesem Fall an der Stelle *add* weiter. Nun wollen wir den Wert von R3 auf den bereits vorhandenen Wert in R4 aufaddieren, also R4 := R4 + R3. Da wir in unserem Programm allerdings nur Maschinenbefehle verwenden dürfen, welche wir selbst umgesetzt haben, muss dies etwas umständlicher gemacht werden, als es beispielsweise mit dem Additionsbefehl **add** ginge. Wir vergleichen nun den Wert in R3 mit jedem Wert von 1-16 und ikrementieren R4 mittels **inc** stetig um 1, solange wie R3 größer als die aktuelle Zahl (1-16) ist. Jetzt wird auch klar, wieso wir im Falle eines Nicht-Findens eines 1-Bits im zuvor durchsuchten Byte R3 mit dem Wert 16 beschrieben haben; wir müssen uns um genau eine Registerbreite (16-Bit) weiterbewegen. Falls an irgendeiner Stelle die Konstante größer ist als der Wert in R3, so befindet sich das erste 1-Bit an der Stelle, welche nun in R4 steht und das Programm kann beendet werden. Falls allerdings in dem durchsuchten Byte kein 1-Bit gefunden wurde, so müssen wir R3 mit -1 als Standradwert beschreiben und mittels einer immer wahren

Bedingung (wir dürfen nur **jumplt** verwenden, nicht **jmp**) nach *add_ende* springen.

Im zuvor durchsuchten Bitfeld befindet sich also kein 1-Bit. Die Startadresse des Bitfeldes (Speicherzelle), sowie der Feldindex müssen somit inkrementiert werden. Nun muss überprüft werden, ob wir die endgültige Länge des Feldes erreicht haben, in dem wir suchen dürfen. Ist dies nicht der Fall, so können wir einen erneuten Schleifendurchlauf starten. Falls wir allerdings alle Teil-Felder durchsucht haben und das Programm nicht im *add*-Abschnitt verlassen haben, so bedeutet das, dass es in unserem Feld gar kein 1-Bit gibt. Wir speichern in R4 also den Wert -1 ab und beenden das Programm.

## 3.4 Laufzeitanalyse

### 3.4.1 Best Case

Im besten Fall befindet sich gleich im ersten Wort die Zahl 1. Das Ergebnis 0 wird sofort in R4 abgespeichert und das Programm endet. Daraus lässt sich eine Laufzeit von $\mathcal{O}(1)$ ableiten.

### 3.4.2 Average Case

Im Schnitt gehen wir davon aus, dass jedes Bit mit einer Wahrscheinlichkeit von $1/2$ gesetzt bzw. nicht gesetzt wird. Die Laufzeit liegt somit in $\sum_{k=1}^{n} \mathcal{O}(\frac{k}{2^k}) = \mathcal{O}(\sum_{k=1}^{n} \frac{k}{2^k})$.

### 3.4.3 Worst Case

Im schlechtesten Fall steht im gesamten Feld kein einziges gesetztes Bit. Dies bedeutet, dass wir eine Laufzeit von $\mathcal{O}(n)$ erhalten, da jedes Wort überprüft werden muss.