FG Leistungselektronik
und Steuerungen in der
Elektroenergietechnik

The SPIRIT of science
TECHNISCHE UNIVERSITÄT
ILMENAU

Mini-Projekt: Simulator des Beispielrechners aus der Vorlesung

Hendrik Fehr

15. Mai 2024

1 Ziele

Das Mini-Projekt hat folgende Ziele:

Produkte In Gruppenarbeit sollen folgende Produkte erstellt werden:

- Ein in C geschriebener Simulator für den Beispielrechner aus der Vorlesung.
- Drei Beispielprogramme für den Simulator.
- Ein Git-Repository mit der Entwicklungsgeschichte des Quelltextes.

Ausführung Jedes Mitglied der Gruppe übernimmt einen Teil der Programmieraufgabe, beispielsweise das Einlesen der Datei, das Erstellen der Beispielprogramme, den Simulator oder die Ausgabe der Ergebnisse. Die Entwicklung findet hauptsächlich im eigenen Zweig des Git-Projekts der Gruppe statt, wobei häufige und nachvollziehbar dokumentierte Commits entstehen sollen. Beachten Sie die Hinweise aus der Vorlesung zum Erstellen der Commit-Einträge. Die Zweige werden dann zu einer gemeinsamen Version zusammengeführt. Das Zusammenführen geschieht natürlich nach Bedarf, zum Beispiel wenn in einem Programmteil eine Verbesserung erzielt wurde. Ein wesentlicher Aspekt der Arbeitsweise besteht darin, die noch nicht fertigen Teile des Programms im eigenen Zweig geeignet nachzustellen. Wenn das zeilenweise Einlesen noch nicht bereitsteht kann die Testeingabe zum Beispiel aus einem Array geschehen. Wenn der Assembler noch nicht bereitsteht kann zum Testen des Simulators das weiter unten gezeigte Speicherabbild sowie zusätzliche von Hand erstellte Testfälle im Binärformat genutzt werden. Oftmals ist es sinnvoll für die jeweiligen Testfälle eine eigene `main`-Funktion zu definieren. Die zugehörige Quelltextdatei muss auch im Git-Repository geführt werden. In *Code Composer Studio* bzw. in *Eclipse* kann man unter *Resource Configurations* mittels *Exclude from Build* sicherstellen, dass nur die Quelltextdatei mit der gewünschten `main`-Funktion beim *Build* berücksichtigt wird. Andernfalls gibt es einen Fehler beim *Build*, weil mehr als eine `main`-Funktion vorhanden ist.

Prüfungsvorbereitung In einem technischen Gespräch demonstriert die Gruppe den Entwicklungsprozess anhand des Git-Projekts. Die Teilnehmer sollen z. B. Zwischenstände auschecken und die Details des jeweiligen Versionsstands erklären und dabei gestellte Fragen beantworten. Diese Vorbereitung wird nicht bewertet, dient also der gezielten Vorbereitung auf die Prüfung.

Zu Beginn des technischen Gesprächs wird das Git-Repository aus dem persönlichen Bereich eines Gruppenmitglieds auf einen Rechner geklont. Es ist dabei egal, von welchem Mitglied geklont wird, weil jeder aus der Kleingruppe eine Instanz des Repositorys im persönlichen Gitlab-Bereich pflegen soll.

2 Beschreibung des Simulators

2.1 Funktion

Das Blockschaltbild des Rechners ist in Abbildung 1 dargestellt. Die wichtigsten Komponenten sind die ALU[1], der Speicher MEM, sowie die Register AC[2], DR[3], PC[4], IR[5] und AR[6]. Der Simulator soll diesen Rechner simulieren, d. h. die Werte der Register, Spezialregister und der geänderte Speicherinhalt des Speichers MEM am Ende jeder Fetch- bzw. Execute-Phase wird ermittelt und ausgegeben. Die Implementierungsdetails der Befehle (z. B. der arithmetischen und logischen Befehle) werden vernachlässigt, aber die Ergebnisse sollen mit der tatsächlichen Hardware übereinstimmen.

Bevor der Rechner laufen kann muss der Simulator ein Assemblerprogramm einlesen und daraus den Anfangszustand des Speichers erstellen. Der Speicher wird also mit den Maschinenbefehlen und Daten initialisiert. Dann startet die Simulation des Modells, ausgehend vom Reset des Rechners, also mit der Fetch-Phase an Adresse 0 des Speichers. Signallaufzeiten und der Zustand der internen Steuerleitungen z. B. für die Multiplexer und für die ALU müssen nicht simuliert werden.

2.2 Eingabe

Das zu simulierende Programm wird aus einer Eingabedatei gelesen, wobei die Mnemonics des Instruction Sets aus der Vorlesung zum Einsatz kommen. Beispieldatei:

```
; Zeilen, die mit Semikolon beginnen sind Kommentare
; und werden ignoriert. Die Datei ist in Abschnitte unterteilt.
; Im Abschnitt .DATA werden Namen fuer Speicherstellen angegeben.
.DATA
a: 7
b: 3
c: ? ; das Fragezeichen bedeutet, dass der Anfangswert beliebig ist.
```

[1] *arithmetic logic unit* (arithmetisch-logische Einheit)
[2] *accumulator* (Akkumulator)
[3] *data register* (Datenregister)
[4] *program counter* (Programmzähler)
[5] *instruction register* (Befehlsregister)
[6] *adress register* (Adressregister)

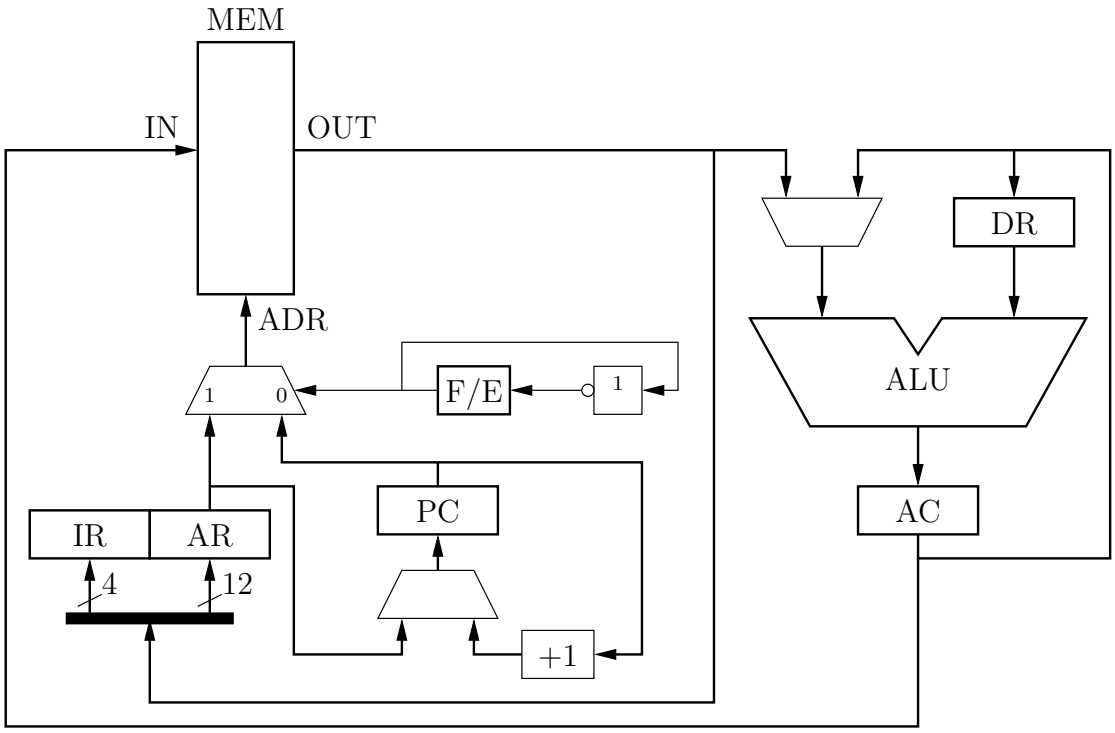


Abbildung 1: Blockschaltbild der Hauptkomponenten des Rechners. Die Steuerung des Adressmultiplexers in Abhängigkeit der Fecht- und Execute-Phasen mit dem F/E-Register ist ausführlich dargestellt. Die restlichen Steuerleitungen, z. B. für den Multiplexer vor dem PC-Register oder für die ALU sind nicht dargestellt. Dünne Signallinien stellen die Übertragung eines Bits dar. Breite Signallinien symbolisieren die parallele Übertragung mehrere Bits. Der Speicher arbeitet mit 16-Bit-Worten und 12-Bit-Adressen, wie die angegebenen Bitbreiten von IR und AR andeuten.

```
; Die Anweisungen folgen im Abschnitt .CODE
.CODE
LD a
MOV DR, AC
LD b
ADD
ST c
; Das Ende der Anweisungen wir mit .END signalisiert
.END
```

Jede Zeile im Abschnitt `.DATA` entspricht einer Speicherstelle. Die Syntax lautet `<Name>: [Anfangswert]` oder `<Name>: ?`. Beispielsweise deklariert die Zeile

```
z100: 20
```

eine Variable mit dem Namen `cnt100` und dem Anfangswert 20. Soll kein expliziter Anfangswert vergeben werden, kann die Angabe entfallen oder durch `?` ersetzt werden.[7]

Die Assemblersprache unterstützt symbolische Sprungmarken im Abschnitt `.CODE`. D. h. die Sprungziele der Anweisungen `BRA <Sprungmarke>` und `BZ <Sprungmarke>` können symbolisch angegeben werden durch sogenannte Sprungmarken. Der Sprung erfolgt zu der Instruktion, die durch Angabe der Sprungmarke, gefolgt von einem Doppelpunkt, definiert wurde. Direkt angegebene Zahlen werden als Adresse interpretiert, wobei die Adressierung von null aufsteigend an der ersten Instruktion im `.CODE`-Abschnitt beginnt. Der Simulator erhält die Instruktionen und Daten im Maschinenformat, weshalb die Speicheradressen der durch Sprungmarken gekennzeichneten Instruktionen ermittelt und in den Adressteil der Sprungbefehle eingetragen werden müssen. Die Speicheradressen von Variablen müssen ebenfalls ermittelt werden um sie in den Adressteil der Lese- und Schreibinstruktionen einzutragen. Bei den `LD`- und `ST`-Instruktionen können auch numerische Adressen angegeben werden. Im folgenden Assemblerprogramm wird der Schleifenkörper von `loop` bis `loop_ende` dreimal durchlaufen:

```
; Beispiel einer Schleife
.DATA
cnt: 3; Zählvariable 3, 2, 1 führt zum Durchlaufen, 0 zum Verlassen
u: 1
.CODE
LD u
MOV DR, AC ; in DR steht jetzt 1
loop: LD cnt
BZ loop_ende
SUB
ST cnt
BRA loop
loop_ende: ST cnt; hier weiter nach der Schleife
.END
```

[7] Die Notation mit spitzen, eckigen und runden Klammen wird oft zur Beschreibung von Befehlen genutzt, siehe zum Beispiel www.tfug.org/helpdesk/general/man.html. Spitze, eckige oder geschweifte Klammern erklären den Aufbau der Zeichenkette und werden in der Ausführung weggelassen. Spitze Klammern weisen auf zwingend erforderliche Bestandteil hin. `<Pflicht>`, eckige Klammern signalisieren optionale Bestandteile `[optional]`. Geschweifte Klammern signalisieren mehrere Optionen: `{entweder, oder}`.

Die Sprungmarken und Variablennamen sind global und müssen eindeutig sein. Mehrfache Definition desselben Namens ist ein Fehler, der zum Abbruch führt, wobei eine aussagekräftige Meldung auf das Problem hinweist.

Die Assemblersprache unterstützt die Adressbildung mit der Notation `&<Variable>` und `&<Sprungmarke>`, wie folgendes Beispielprogramm demonstriert:

```
; Hier werden der indirekte Sprung und das indirekte Laden
; demonstriert. Zuerst wird ein Sprungbefehl gebildet, der die
; Sprungmarke fkt anspringt, und zwar durch Verwendung der
; Adresse, die in adr_fkt bereitstellt wird.
; Die Instruktionen ab fkt zeigen den indirekten Zugriff auf
; a mit Hilfe der Adresse von a, die in adr_a
; bereitliegt. Am Ende liegt a in AC.
.DATA
b: 99; Beispielwert zur Erzeugung der dummy-Instruktion in Zeile 34
a: 71;
adr_a: &a ; In adr_a steht die Adresse der Speicherstelle von a.
adr_fkt: &fkt ; In adr_fkt steht die Adresse der Instruktion von fkt.
OPCODE_BRA: 40960; Opcode des Sprungbefehls mit ADR = 0
OPCODE_LD: 32768; Opcode des Ladebefehles mit ADR = 0
; Indirekter Sprung zu fkt, und indirekter Zugriff auf a.
.CODE
LD adr_fkt; erstelle Sprungbefehl zu fkt
NOT
MOV DR, AC
LD OPCODE_BRA
NOT
AND
NOT; In AC steht jetzt die Instruktion 'BRA fkt'
ST mod_bra
mod_bra: BRA mod_bra; diese Instruktion wird zu 'BRA fkt' geändert
fkt: LD adr_a; Erstelle Ladebefehl von a
NOT
MOV DR, AC
LD OPCODE_LD
NOT
AND
NOT; In AC steht jetzt die Instruktion 'LD a'
ST ind_ld
ind_ld: LD b; diese Instruktion wird zu 'LD a'
; jetz steht a in AC
.END
```

2.3 Ausgabe

Ausgegeben wird der Zustand der Register IR, AR, PC, F/E, DR und AC nach jeder Fetch- und jeder Execute-Phase. Außerdem wird die zugehörige Zeile aus dem `.CODE`-Abschnitt angegeben. Nicht initialisierter Speicher oder nicht initialisierte Register werden durch Ausgabe eines Fragezeichens signalisiert. Wenn sich der Speicher geändert hat, wird außerdem die betroffene Adresse und der neue Wert ausgegeben. Die Ausgabe erfolgt auf der Standardausgabe und auf Wunsch in eine Datei. Die Ausgabe der Simulation des

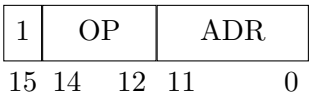
obigen Assemblerprogramms könnte wie folgt aussehen:

```
Step                F/E PC  IR  AR  DR  AC  |    MEM
0   (reset)          0   0   0   0   0   0  |
1   LD a             1   0   8   5   0   0  |
2                    0   1   8   5   0   7  |
3   MOV DR, AC       1   1   0   0   0   7  |
4                    0   2   0   0   7   7  |
5   LD b             1   2   8   6   7   7  |
6                    0   3   8   6   7   3  |
7   ADD              1   3   2   0   7   3  |
8                    0   4   2   0   7  10  |
9   ST c             1   4   9   7   7  10  |
10                   0   5   9   7   7  10  |  @7:10
```

Der Inhalt des Speichers soll ebenfalls ausgegeben werden können.

3 Opcode und Binärformat

In der Vorlesung wurden schon Wortbreite und Definition der Opcodes eingeführt. Die dort vorgeschlagene Festlegung zwischen Befehlen mit Adressteil und Befehlen ohne Adressteil zu unterscheiden wird wieder aufgegriffen. Die Instruktionen mit Adressteil sind wie folgt aufgebaut:



Und die Instruktionen ohne Adressteil sind wie folgt aufgebaut:

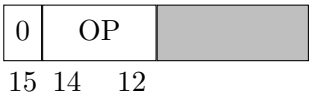


Tabelle 1 zeigt die Zuorgnung der Opcodes. Die Speicherdarstellungen des obigen Programms `Beispiel einer Schleife` sieht vor dem Start aus wie folgt:

```
Speicherdarstellung vor dem Start:
[0] = 0x8009
[1] = 0x0000
[2] = 0x8008
[3] = 0xb007
[4] = 0x3000
[5] = 0x9008
[6] = 0xa002
[7] = 0x9008
[8] = 0x0003
[9] = 0x0001
```

Nach dem Durchlauf sieht der Speicher wie folgt aus:

```
Speicherdarstellung nach dem Ende:
[0] = 0x8009
[1] = 0x0000
```

Tabelle 1: Binärformat der Instruktionen des Rechners.

Instruktion	Bit 15	OP	ADR
LD x	1	0b000	Adresse der Speicherstelle x
ST x	1	0b001	Adresse der Speicherstelle x
BRA *label*	1	0b010	Adresse des Sprungziels *label*
BZ *label*	1	0b011	Adresse des Sprungziels *label*
MOV DR, AC	0	0b000	0
MOV AC, DR	0	0b001	0
ADD	0	0b010	0
SUB	0	0b011	0
AND	0	0b100	0
NOT	0	0b101	0

```
[2] = 0x8008
[3] = 0xb007
[4] = 0x3000
[5] = 0x9008
[6] = 0xa002
[7] = 0x9008
[8] = 0x0000
[9] = 0x0001
```

Diese Initialisierung und der Endzustand eignen sich als Testfälle für die Entwicklung des Simulators, wenn der Assembler noch nicht fertig ist.

4 Kommandozeilenparameter

Der Simulator soll im ersten Pflicht-Argument den Dateinamen der Assemblerdatei erhalten: `msp_simulator <Datei>`. Optionale Schalter beginnen mit P, A, D, S oder m und werden von einer direkt folgenden Zahl abgeschlossen:

- `P<PC init>` Anfangswert des PC-Registers (default 0)
- `A<AC init>` Anfangswert des AC-Registers (default 0)
- `D<DR init>` Anfangswert des DR-Registers (default 0)
- `S<PC stop>` stoppt Simulation wenn PC der Wert `<PC stop>` hat
- `m<max step>` stoppt Simulation nach `<max step>` Schritten

Der Aufruf `msp_simulator Test.sasm A13 S123 m1000` assembliert und simuliert das Programm in der Datei `Test.sasm` ab Adresse 0 mit dem Anfangswert 13 in AC bis der Programmzähler den Wert 123 hat oder für 1000 Schritte, je nachdem was zuerst eintritt.

main

- Parser → File_reader

Array mit Befehlen

- Simulator → Argumente von Kommandoeingabe für erste Pflichtabfrage
  - Memory initialisieren: - Speicher wird mit Anzahl der Code-Zeilen initialisiert
    - Übersetzung in Hex-Code und schreiben in einzelne Adressen
      - Bei .CODE anfangen, dann .DATA und unten drunter blättern (auch von oben)
      - Code in 4 Register aufteilen / dies über struct definieren (wenn erstes Bit > 8, dann automatisch auf 1 setzen)
      - Fragezeichen initialisiert random-Wert für Variable
  - Register anlegen und initialisieren: Alle 6 Register aufrufen und auf 0 setzen

! global !

Schleife

→ Programm counter +1

- Fetch: Aufteilen der Hex-Codes in IR und AR laden
- Execute: Verarbeiten der Befehle aus IR und AR und laden in AC
  - Schleifen (loop) müssen in Register zucken, da für BZ muss AC = 0 sein
    - Ausgabematrix wird trotzdem weiter gefüllt.
    - Nur IR wird bei Schleifensprung geladen