Tutorat 4

Zustandsdiagramme, DMA

Betriebssysteme, Tutorat 4, Gruppe 6, juergmatth@gmail.com, Universität Freiburg

Einstieg

Einstieg

Fakeupdate

- https://fakeupdate.net/

Korrektur

Korrektur

Interessantes und häufige Fehler

- überschreiben der Daten, die in der b) nach links geshiftet wurden. `OR IN1 1`, **Non-Controlling** Bit `0`
- ~~nach dem **Polling erst shiften~~ → am Ende um 8 Stellen zu viel gehiftet
- `SUBI ACC 2` um `b1` `0` zu setzen
  - einfach `0` setzen geht nicht, weil die anderen **Flags** des **Statusregisters** erhalten bleiben sollen
- neuen 8Bitvektor dranfügen aus **Empfangsregister** `ADD IN1 1`
- der **EPROM** ist **READONLY** → hat keinen Stack
- andere **Flags** des **Statusregister** nicht überschreiben, nur das Bit, was geändert werden soll
- bei `JUMPc i` beschreibt das `i`, wie oft man die Speicherzelle wechselt, und zwar von der Speicherzelle, wo das `JUMPc i` steht aus (wie `<count>j` in **(Neo-)Vim**)

Vertiefungen

Vertiefungen

Datensegmentregister

- Solange im `DS` die Bits `30` und `31` mit dem gewollten **Präfix** besesetzt sind muss man sich keine Sorgen machen
- **Verändern kann man die beiden Bits durch:**
  - durch `LOADI DS 0` z.B. mit `0`en überschrieben durch **Signextension**
  - wenn man durch **Multiplikation** andere Bitwerte an Stelle `31` und `30` shiftet
  - oder wenn man den `DS` mit einem anderen **Register** oder **SRAM Inhalt** überschreibt, die `32` Bit lang sind

Vertiefungen

Signed (2er Komplement oder 1er Komplement) und Unsigned

Unsigned (oder Betrag mit Vorzeichen)

- $< x >= x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$
- $[x]_{BV} = (-1)^{x_{n-1}} \cdot (x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0)$
- **Bereich:** 0 bis $+2^n - 1$ oder $-2^{n-1} + 1$ bis $+2^{n-1} - 1$

Signed (2er Komplement)

- $[x]_2 = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$ (weil `1000` - `1` = `111`)
- **Bereich:** -2^{n-1} bis $+2^{n-1} - 1$ (die 0 muss auch iwo hin)

Vertiefungen

Signed (2er Komplement oder 1er Komplement) und Unsigned

Signed (1er Komplement)

- $[x]_1 = -x_{n-1}(2^{n-1} - 1) + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$
- **Bereich:** $-2^{n-1} + 1$ bis $+2^{n-1} - 1$ (es gibt **2** Kodierungen für die `0`)

Vergleich Kodierung von Unsigned, Signed im 1er und 2er Komplement

x	000	001	010	011	100	101	110	111
$[x]_{BV}$	0	1	2	3	0	-1	-2	-3
$[x]_2$	0	1	2	3	-4	-3	-2	-1
$[x]_1$	0	1	2	3	-3	-2	-1	0

Vertiefungen

Signed (2er Komplement oder 1er Komplement) und Unsigned

Kodierung Bedeutungen

- **Höchstwertiges** Bit ist **Sign Bit**, `1` für **negativ**, `0` für **positiv**
- `<i>` **unsigned** und `[i]` **signed**
- **Little Endian**=niedrigstwertiges Bit zuerst, **Big Endian**=höchstwertiges Bit zuerst

Vertiefungen

Signed (2er Komplement oder 1er Komplement) und Unsigned

Interessante Zahlen für 2er Komplement

- `_0`: `0000 0000 ... 0000`
- `-1`: `1111 1111 ... 1111`
- **Negativste Zahl:** `1000 0000 ... 0000`
- **Positivste Zahl:** `0111 1111 ... 1111`

Signed Negation (2er Komplement)

- $\bar{x} + 1 = -x$ (1er Komplement Negation + 1, da $x + \bar{x} = 1111 \ldots 111_2 = -1$)

Vertiefungen

Signextension

- **von 8 Bit auf 16 Bit:**
  - `+2` : 0 `000 0010 => 0000 0000` 0 `000 0010`
  - `−2` : 1 `111 1110 => 1111 1111` 1 `111 1110`
- **unsigned** wird mit `0`en extendet
- das **Sign Bit** wird nach **links** dupliziert

Vertiefungen

Addition binär und dezimal

```
  011011 (27)          17718
+ 011101 (29)        +  6524
  11111                 11 1
  ======               ======
  111000 (56)          24242
```

```
00 + 00 = 00          00 + 00 (+ 01) = 01
00 + 01 = 01          00 + 01 (+ 01) = 10
01 + 00 = 01          01 + 00 (+ 01) = 10
01 + 01 = 10          01 + 01 (+ 01) = 11
```

Vertiefungen

Subtraktion binär und dezimal (nicht empfohlen, dient Vergleich mit nächster Folie)

```
(1)
  0111000 (56)          24242
- 0011011 (27)        - 17718
   11111                 11 1
  =======               ======
  0011101 (29)           6524
```

```
10 - 00 = 10                    10 - 00 (- 01) = 01
10 - 01 = 01                    10 - 01 (- 01) = 00
11 - 00 = 11                    11 - 00 (- 01) = 10
11 - 01 = 10                    11 - 01 (- 01) = 01
```

Vertiefungen

Subtraktion binär und dezimal (funktioniert immer, egal was für Vorzeichen Zahlen haben)

```
(2)
   0111000 (56)
 + 1100101 (27) (0011011 negiert und +1)
  11
   =======
   0011101 (29)
```

- **Zweierkomplement Negation:** `11011 -> 011011 -> 100100 -> 100101`
  - `0`en hinzufügen bis **Minuend** und **Subtrahend** beide gleiche Länge haben und Platz für ihr **Vorzeichenbit** ist und dieses korrekt gesetzt ist
  - **1er Komplement Negation** und `+1` nicht vergessen für den **Subtrahenden**

Vertiefungen

Multiplikation binär und dezimal

```
1101 x 1001 (13 * 9)          1304 x 12
       1101                          48
      0000                     +    0
     0000                      + 36
    1101                       +12
   ========                    ======
    1110101 (117)               15648
```

- **Verschiebung** ist aufgrund der `0`en, die hier ausgelassen sind

Vertiefungen

Division binär

```
  1110101 / 1011 (117 : 11) = 1010 (10) Rest: 111 (7)
- 1011|||
 =====|||
    111||
-     0||
   ====||
    1110|
-   1011|
   =====|
      111
-       0
     ====
      111
```

Vertiefungen

Division dezimal

```
15658 / 12 = 1304,833...
12|||
==|||
 36||
 36||
 ==||
  05|
   0|
  ==|
   58
   48
    ...
```

Vertiefungen

Division dezimal

```
==
10|0  oder Rest: 10
 9 6
 ===
   40
   36
   ==
    40
    36
    ==
     4...
```

Vertiefungen

Division binär

- bei **binärer Division** gibt es nur **2 Zuständen** (`1` oder `0`), dementsprechend wird entweder die Zahl so übernommen (Zahl $\cdot$ `1`) oder die Zahl ist `0` (Zahl $\cdot$ `0`)

Division allgemein

- nach jeder Addition ein Zahl runterholen, bis keine mehr runtergeholt werden kann $\rightarrow$ dann Ende (bei **ganzzahliger Division**). Was unten stehen bleibt ist der **Rest**
- bei Division mit Nachkommastellen, 0en runterbringen, bis einmal **kein Rest** mehr rauskommt oder Grenze setzen bis zu der man weiter macht $\rightarrow$ dann Ende
- ist der **Dividend** trotz runtergebrachter weiter Stelle (weil einmal kein Rest übrig blieb) immernoch kleiner als der **Divisor**, so ist der **Quotient** 0, weil nur durch $\cdot 0$ rechnen kann der **Divisor** noch kleiner sein als der **Dividend**

Linux Background Knowledge

Linux Background Knowledge

Desktopenvironment aufsetzen

Quellen

Quellen

Wissenquellen

Vielen Dank für eure Aufmerksamkeit!