Kurz-Doku für den generischen Assembler *genasm*

Kurzbeschreibung:

genasm ist ein Programm, welches eine Datei mit Assembler-Quelltext in eine Textdatei übersetzt, die den Maschinencode in Form binärer Speicherwörter enthält, die als Zeichenkette abgespeichert werden. Der Assemblerdialekt sowie der Aufbau eines Speicherwortes werden in einer Konfigurationsdatei definiert und sind damit jederzeit an Veränderungen in der Architektur des Prozessors anpassbar. Diese Textdatei kann genutzt werden, um sie als Eingabedatei für die PROM-Bausteine im ISE SchematicEditor zu verwenden (z.B. mem16x8). Der Parser von *genasm* ist entsprechend generisch aufgebaut, so dass sowohl (eingeschränkt) die textuelle Kodierung eines Assembler-Quelltextes als auch die konkreten Befehlsformate des Befehlswortes im Speicher übersetzt werden können.

Einschränkungen der Konfigurierbarkeit des Assembler-Quelltextes:

Assemblerbefehle müssen grundsätzlich immer mit dem Befehlsnamen beginnen und können dann von einer belieben Anzahl von Operanden gefolgt werden. Trennsymbol zwischen Befehlsnamen und erstem Operand ist grundsätzlich ein Leerzeichen. Trennsymbole zwischen den Operanden können konfiguriert werden (default: Leerzeichen), auf Buchstaben, Zahlen, das Zeichen ‚#', sowie dem Zeichen für Kommentare sollte allerdings verzichtet werden. Beispiel:

```
ADD r0, #0x3fff, r3
```

Das Zeichen, welches einen Kommentar einleitet, kann definiert werden. Voreingestellt ist das Zeichen ‚;'. Ein Kommentar gilt immer ab dem Kommentarzeichen bis zum Ende der Zeile.

Einschränkung der Operanden:

Zurzeit können nur Operanden vom Typ „Register", sowie Konstanten verwendet werden. Register-Operanden müssen mit einen ‚r' oder ‚R' beginnen, gefolgt von der Nummer des Register. Die Nummer des Registers muss dezimal angegeben sein (siehe Beispiel oben). Konstanten müssen immer mit dem Zeichen ‚#' eingeleitet werden, gefolgt von dem konstanten Wert in dezimaler Form, binärer Form (vorangestellte Sequenz „0b") oder in hexadzimaler Form (vorangestellte Sequenz „0x"). Konstanten dürfen auch negativ sein (vorangestelltes Zeichen ‚-'). Beispiele:

```
#-13    #-0xafff    #-0b00110101    #42    #0x2a    #0b11010101
```

Es ist darauf zu achten, dass die Konstanten und Registernummern in ihrem Zahlwert nicht größer sind, als den Wertebereich, den das Befehlsformat aufnehmen kann; *genasm* führt hier keine Konsistenzprüfungen durch und schneidet ggf. einfach die höherwertigen Bits ab.

Allgemeine Einschränkungen des Assemblertextes:

Es werden zurzeit keine Label unterstützt. Damit sind Sprungbefehle nicht zu verwenden. Es werden auch keine weiteren Assemblerdirektiven, wie etwa .ORG o.ä. unterstützt.

Beschreibung der Konfigurationsdatei:

Die Konfigurationsdatei ist eine XML-Datei, in der die einzelnen Befehlsformate des Instruktionswortes sowie der konkrete Assemblerdialekt festgelegt werden. Die oberste Ebene der Konfiguration wird mit dem Tag <config> eingeleitet. <config> hat ein optionales Attribut, nämlich „casesensitiv“, welches auf die Werte „true“ oder „false“ eingestellt werden kann (default: false). Dieses Attribut definiert, ob *genasm* beim Übersetzen sensitiv auf Groß- und Kleinschreibung reagieren soll. Beispiel:

```
<config casesensitive="true">
  …
</config>
```

Als erste Unterebene unter <config> muss die Definition des Instruktionswortes erfolgen, so wie es später nach der Übersetzung im Speicher abgelegt werden soll. Dies erfolgt über das Tag <instruction_word> und besitzt ein optionales Attribut „width“. „width“ gibt die Breite des Instruktionswortes in Bits an (default: 8 Bit). Es wird zurzeit nur ein Instruktionswort fixer Länge unterstützt. Beispiel:

```
<config casesensitive="true">
  <instruction_word width="16">
   …
  </instruction_word>
  …
</config>
```

Innerhalb der Definition des Instruktionswortes werden nun die unterschiedlichen Instruktionsformate definiert. Hiervon kann es mehrere geben, die über einen Namen unterschieden werden. Dieser Name muss eindeutig sein und muss für jedes Instruktionsformat über das einzige Attribut „name“ übergeben werden. Beispiel:

```
<config casesensitive="true">
  <instruction_word width="16">
    <instruction_format name="RegConstRegFormat" >
        …
    </instruction_format>
    <instruction_format name="RegRegRegFormat" >
        …
    </instruction_format>
      …
  </instruction_word>
  …
</config>
```

Bei der Definition der einzelnen Befehle werden später diese Instruktionsformate ausgewählt, um beispielsweise einen ADD im einen oder anderen Format im Instruktionswort festlegen zu können, beispielsweise „ADD r0, r1, r2“ und „ADD r0, #0xff, r2“. Damit können die gleichen Bits im Instruktionswort mehrfach belegt werden, abhängig von der konkreten Instruktion (siehe unten).

Das Instruktionsformat bestimmt eine mögliche Interpretation des Instruktionswortes, indem die einzelnen Bits des Instruktionswortes den Opcodes und Operanden zugeordnet werden. Opcodes sind dabei eine spezifische, vom jeweiligen Befehlstyp mit einem konstanten Wert zu belegende, Bitsequenz. Davon darf es in jedem Instruktionsformat beliebig viele geben. Opcodes werden über eindeutige Namen angesprochen und erhalten hier zunächst keine Werte, sondern nur Bitbreiten. Beispiel:

```
<config casesensitive="true">
  <instruction_word width="16">
    <instruction_format name="RegConstRegFormat" >
      <opcode name="ArithInstr" width="3"/>
      <opcode name="CoOp" width="1"/>
        …
    </instruction_format>
    <instruction_format name="RegRegRegFormat" >
        …
    </instruction_format>
      …
  </instruction_word>
  …
</config>
```

Die Reihenfolge der Opcodes (und Operanden, siehe unten) legt fest, an welcher Stelle im Instruktionswort diese Bitsequenz steht. Dabei entscheidet die Reihenfolge in der Konfigurationsdatei und es wird beim höchstwertigsten Bit (also links) angefangen. Für das obige Beispiel folgt also das folgende Instruktionsformat im 16-Bit Instruktionswort:

Instruktionwort[15 .. 0], Instruktionsformat "RegConstRegFormat"															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ArithInstr			CoOp												

Die jeweiligen Werte der Opcode-Sequenzen werden später bei der Definition der Assemblerbefehle gesetzt.

Ein weiteres Feld sind die Operanden, die ebenfalls im Instruktionsformat definiert werden. Operanden sind Bitsequenzen, wie Opcodes auch, allerdings werden hier später die Operanden des Assemblerbefehls selbst (beispielsweise Registernummern oder Konstanten eingetragen (siehe unten). Operanden haben neben einem Namen und einer Bitbreite auch noch einen Typ. Hier werden zurzeit nur die Typen Register („register“) und Konstanten („const“) unterstützt. Für Operanden gilt ebenfalls die Reihenfolgeregel und Operanden und Opcodes können beliebig gemischt werden. Die Gesamtlänge des Instruktionswortes ist dabei jedoch zu beachten (*genasm* überprüft dies und generiert im Fehlerfall eine entsprechende Fehlermeldung). Z.B. soll der Befehl „ADD rx, #const, rz“ in das Instruktionswort, wobei die Registernummern 2 Bit breit, die Konstante 8 Bit breit sein sollen:

```
<config casesensitive="true">
  <instruction_word width="16">
    <instruction_format name="RegConstRegFormat" >
      <opcode name="ArithInstr" width="3"/>
      <operand name="Op1" type="register" width="2"/>
      <opcode name="CoOp" width="1"/>
      <operand name="Op2Const" type="const" width="8"/>
      <operand name="Result" type="register" width="2"/>
        …
    </instruction_format>
    <instruction_format name="RegRegRegFormat" >
        …
    </instruction_format>
      …
  </instruction_word>
  …
</config>
```

Es entsteht die folgende Interpretation des Instruktionswortes:

Instruktionwort[15 .. 0], Instruktionsformat "RegConstRegFormat"

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ArithInstr			Op1		CoOp	Op2Const								Result	

Eine dritte Möglichkeit neben Opcodes und Operanden ist es, eine bestimmte Bitsequenz nicht zu nutzen. Dies wird durch das Tag <unused> erreicht, welches nur über ein Breite-Attribut verfügt. Beispielsweise soll der folgende Befehl eine andere Interpretation des Instruktionswortes erlauben, allerdings sollen „Op1" und „result", sowie die Opcodes an der gleichen Stelle stehen. Hierzu wird ein neues Instruktionsformat für den Befehl „ADD rx, ry, rz" definiert:

```
<config casesensitive="true">
  <instruction_word width="16">
    <instruction_format name="RegConstRegFormat" >
      <opcode name="ArithInstr" width="3"/>
      <operand name="Op1" type="register" width="2"/>
      <opcode name="CoOp" width="1"/>
      <operand name="Op2Const" type="const" width="8"/>
      <operand name="result" type="register" width="2"/>
    </instruction_format>
    <instruction_format name="RegRegRegFormat" >
      <opcode name="ArithInstr" width="3"/>
      <operand name="Op1" type="register" width="2"/>
      <opcode name="CoOp" width="1"/>
      <operand name="Op2" type="register" width="2"/>
      <unused width="6"/>
      <operand name="result" type="register" width="2"/>
    </instruction_format>
  </instruction_word>
  …
</config>
```

Damit wird die folgende, neben „RegConstRegFormat“ weitere, Interpretation des Instruktionswortes geschaffen:

Instruktionwort[15 .. 0], Instruktionsformat “RegRegRegFormat“															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ArithInstr			Op1		CoOp	Op2		*unused*						Result	

Dies schließt die Definition der Interpretationen des Instruktionswortes ab und nun müssen die einzelnen Befehle definiert werden. Dies erfolgt als weiteres Unterfeld von <config> und nennt sich Instruktionsdefinition <instruction_def>, welches keine weiteren Attribute besitzt. Unter <instruction_def> befinden sich nun alle Befehle in allen Formaten definiert, und zwar so, wie der Compiler sie aus dem Quelltext extrahieren soll und wie welche der Bitsequenzen in welchen Instruktionsformaten zu setzen sind:

```
<config casesensitive=“true“>
  <instruction_word width=“16“>
    <instruction_format name=“RegConstRegFormat“ >
          …
    </instruction_format>
      …
  </instruction_word>
  <instruction_def>
      …
  </instruction_def>
  …
</config>
```

Die Definition einzelner Befehle des Prozessors erfolgt über das Tag <instruction>. Dieses Tag besitzt zwingend ein Attribut „name“, unter welchem der Name des Assemblerbefehls, wie er im Assembler Quelltext auftaucht, anzugeben ist, sowie ein erforderliches Attritut „instructionformat“, mit welchem das für diesen Befehl zu verwendende Instruktionsformat aus obiger Definition zu verwenden ist. Für die beiden Befehle „ADD rx, #const, rz“ und „ADD rx, ry, rz“ folgt also:

```
<config casesensitive=“true“>
  <instruction_word width=“16“>
    <instruction_format name=“RegConstRegFormat“ >
          …
    </instruction_format>
      …
  </instruction_word>
  <instruction_def>
    <instruction name=”ADD” instructionformat=”RegConstRegFormat”>
      …
    </instruction>
    <instruction name=”ADD” instructionformat=”RegRegRegFormat”>
      …
    </instruction>
  </instruction_def>
  …
</config>
```

Jeder Befehl taucht also so häufig auf, wie er verschiedene Instruktionsformate benutzt. Groß- und Kleinschreibung der Befehlsnamen ist nur dann wichtig, wenn bei <config> die entsprechende Sensitivität eingestellt wurde. Nun muss dem *genasm* noch mitgeteilt werden, mit welchen Werten die Opcode-Sequenzen des Instruktionsformates für diesen konkreten Befehl gefüllt werden müssen. Dies geschieht über das Tag <opcode> unterhalb von <instruction>. <opcode> darf mehrfach auftreten und muss den Namen des Opcode-Feldes im gewählten Instruktionsformat angeben. Opcode-Sequenzen, die nicht in <instruction> über <opcode> mit einem konkreten Wert belegt werden, werden auf '0' gesetzt. Für unser Beispiel soll der Befehl „ADD" durch die Bitkombination „101" im ArithInstr-Feld repräsentiert werden. Das Opcode-Feld CoOp dient der Identifizierung einer Konstanten bei Operand2 und wird in diesem Fall auf '1' gesetzt, sonst auf '0':

```
<config casesensitive="true">
  <instruction_word width="16">
    <instruction_format name="RegConstRegFormat" >
          …
    </instruction_format>
      …
  </instruction_word>
  <instruction_def>
    <instruction name="ADD" instructionformat="RegConstRegFormat">
      <opcode name="ArithInstr" value="101"/>
      <opcode name="CoOp" value="1"/>
       …
    </instruction>
    <instruction name="ADD" instructionformat="RegRegRegFormat">
      <opcode name="ArithInstr" value="101"/>
      <opcode name="CoOp" value="0"/>
       …
    </instruction>
  </instruction_def>
  …
</config>
```

Die Reihenfolge, in denen die Opcode-Zuweisungen auftauchen ist beliebig. Zu achten ist auf die Länge des Value-Feldes, da dies die Breite der Opcode-Definition aus dem Instruktionsformat nicht überschreiten darf (*genasm* testet dies nicht explizit, sondern schneidet die unteren Bits ggf. ab).

Der Compiler muss nun die Operanden des Befehls (bspw. „ADD rx, #const, rz") lesen und zuordnen. Dazu muss ihm pro Befehl und Instruktionsformat mitgeteilt werden, wie die Zeichenkette nach dem Befehlsnamen inkl. trennender Leerzeichen (also hier: „rx, #const, rz") zu interpretieren ist. Hierzu ist im Tag <instruction> in der Reihenfolge der Operanden im Quelltext die Interpretation anzugeben, inklusive der Trennzeichen zwischen den Operanden (hier das Zeichen ','). Die Definition der Operanden einer Instruktion muss dabei in XML innerhalb des Tags <operand_format> erfolgen, von dem pro Instruktion nur eines erlaubt ist. Die Interpretation umfasst dabei die Identifizierung des Operanden des Instruktionsformates (siehe oben). *genasm* setzt dann an der entsprechend im Instruktionsformat definierten Stelle im Instruktionswort die in eine Bitsequenz umgewandelte Zahl ein. Hierzu dienen die Tags <operand> und <delimiter>. <operand> muss mit dem Namen des Operand-Feldes aus der Definition des Instruktionsformates ausgestattet werden. <delimiter> zwischen den Operanden benennt das Trennzeichen (nur ein einzelnes Zeichen), welches *genasm* lesen soll. Ist kein <delimiter>-Tag angegeben, geht *genasm* von einem Leerzeichen als Trennzeichen

aus. Für das Beispiel oben können nun die auftretenden Assembler-Quelltext-Befehle eindeutig definiert werden (wiederum die Befehle „ADD rx, #const, rz“ und „ADD rx, ry, rz“):

```
<config casesensitive="true">
  <instruction_word width="16">
    <instruction_format name="RegConstRegFormat" >
          …
    </instruction_format>
      …
  </instruction_word>
  <instruction_def>
    <instruction name="ADD" instructionformat="RegConstRegFormat">
      <opcode name="ArithInstr" value="101"/>
      <opcode name="CoOp" value="1"/>
      <operand_format>
        <operand name="Op1"/>
        <delimiter value=","/>
        <operand name="Op2Const"/>
        <delimiter value=","/>
        <operand name="Result"/>
      </operand_format>
    </instruction>
    <instruction name="ADD" instructionformat="RegRegRegFormat">
      <opcode name="ArithInstr" value="101"/>
      <opcode name="CoOp" value="0"/>
      <operand_format>
        <operand name="Op1"/>
        <delimiter value=","/>
        <operand name="Op2"/>
        <delimiter value=","/>
        <operand name="Result"/>
      </operand_format>
    </instruction>
  </instruction_def>
  …
</config>
```

Damit ist die Konfiguration für *genasm* vollständig. Als letztes Element kann unter <config> noch das Zeichen für Kommentare definiert werden. Dies erfolgt durch das Tag <comment> mit dem Attribut „value“. Kommentare gelten immer ab dem Kommentarzeichen bis zum Zeilenende.

genasm kann nun einen Assembler-Quelltext übersetzen in eine Liste von binären Zeichenketten, die das jeweilige Instruktionswort enthalten, mithin also den Maschinencode erzeugen. Für das Beispiel (siehe die vollständige Konfigurationsdatei unten anhängend) und das folgende Programm:

```
ADD r0,#0xff , r3  ; hier wird RegConstRegFormat benutzt
; ADD r0, r0, r0    ; diese Zeile ist ein Kommentar und wird nicht übersetzt
ADD r2, r3, r1      ; hier wird das RegRegRegFormat benutzt
```

ergibt sich die folgende Binärdarstellung (Textfile):

```
1010011111111111
1011001100000001
```

Hier nochmals die vollständige Konfiguration für das Beispiel:

```
<config casesensitive="true">
  <instruction_word width="16">
    <instruction_format name="RegConstRegFormat" >
      <opcode name="ArithInstr" width="3"/>
      <operand name="Op1" type="register" width="2"/>
      <opcode name="CoOp" width="1"/>
      <operand name="Op2Const" type="const" width="8"/>
      <operand name="result" type="register" width="2"/>
    </instruction_format>
    <instruction_format name="RegRegRegFormat" >
      <opcode name="ArithInstr" width="3"/>
      <operand name="Op1" type="register" width="2"/>
      <opcode name="CoOp" width="1"/>
      <operand name="Op2" type="register" width="2"/>
      <unused width="6"/>
      <operand name="result" type="register" width="2"/>
    </instruction_format>
  </instruction_word>
  <instruction_def>
    <instruction name="ADD" instructionformat="RegConstRegFormat">
      <opcode name="ArithInstr" value="101"/>
      <opcode name="CoOp" value="1"/>
      <operand_format>
        <operand name="Op1"/>
        <delimiter value=","/>
        <operand name="Op2Const"/>
        <delimiter value=","/>
        <operand name="Result"/>
      </operand_format>
    </instruction>
    <instruction name="ADD" instructionformat="RegRegRegFormat">
      <opcode name="ArithInstr" value="101"/>
      <opcode name="CoOp" value="0"/>
      <operand_format>
        <operand name="Op1"/>
        <delimiter value=","/>
        <operand name="Op2"/>
        <delimiter value=","/>
        <operand name="Result"/>
      </operand_format>
    </instruction>
  </instruction_def>
  <comment value=";"/>
</config>
```

Aufruf von *genasm*:

genasm wird von der Kommandozeile (cmd) aus mit folgenden Parametern gestartet:

`c:\`*`Path`*`\> genasm `*`config_file  assembler_file  binary_file`

wobei:

- **config_file**: Die Datei mit der XML-Beschreibung des Instruktionswortes und der Befehle
- **assembler_file**: Die Assembler-Quelltext-Datei
- **binary_file**: Die Textdatei, in der genasm die übersetzten Speicherworte speichert