YAMA : Yet Another MicroAssembler

Description && User's Guide

Venkatesh Akella

Dept. of Computer Science

University of Utah

Salt Lake City

Utah 84112 USA

Email:`akella@cs.utah.edu`

October 3, 1988

UUCS-88-016

Abstract

YAMA is an CommonLisp program for creating microcode PROMS. It lets you specify the microcode in a fairly high level language that provides various features found normally in sophisticated assemblers. The salient features of YAMA are

1. Provision for *record* and *enumerated* data types at micro assembly level.
2. Provision for a *flexible* and *heirarchical* microword format.
3. Provision for *horizontal* and *vertical* microprograms with the help of special syntactic constructs.
4. Provision of various directives to tailor the code according to one's taste.

It produces the "bit" pattern for the microwords. The input specifications are in a data definition language called MICRO which has a data declaration part, a directives part and a specification section. This report contains the summary of the design of the micro assembly language and the construction of the micro assembler and also serves as the user's guide.

Contents

1 Introduction

YAMA is an CommonLisp program for creating microcode PROMS. It lets you specify the microcode in a fairly high level language that provides various features found normally in sophisticated assemblers. It produces the "bit" pattern for the microwords. The input specifications are in a data definition language (MICRO) which has three distincts parts.

1. **Declaration:** Here you specify the general format of the microword in your application.
2. **Directives :** Here you instruct the assembler how to organize the microwords in the control store.
3. **Specification:** Here you describe the microinstructions to do a specific task.

The report is organized in the following manner:

We begin with a discussion on reasons for choosing CommonLisp to implement the micro assembler followed by the description of the the section on MICRO contains the abstract syntax of the micro assembly language being used. We then describe the construction of the micro assembler and how to go about using it (with a terminal session). The next section contains some more examples of the micro assembly language followed by a section on error recovery and debugging.

2 Using CommonLisp

YAMA has been implemented in CommonLisp which is available on the HP Bobcats. It is helpful to know that the MICRO is based on the CommonLisp syntax. This is used in the specification of numbers and comments in the source file and also for debugging the source programs.

2.1 Numbers

The user has the luxury of specifying numeric data in binary,hexadecimal and decimal notations supported by CommonLisp. If you want to specify a number in a base other than decimal, you use the form

#xnumber in hex notation and

#bnumber in binary notation

2.2 Comments

Anytime YAMA reads a semicolon (;) it ignores everything until the end of the line. This is used for placing comments in your source files. Since microcode is by nature very cryptic, it is a good idea to use lots of comments. Since YAMA uses the lisp "read" function for input, it ignores extra tabs, spaces and blank lines and the user is free to intersperse them to improve the readability of the source code.

2.3 Identifiers

You will soon want to give things like the control signals, control store locations some symbolic names. CommonLisp allows you to use identifiers of any length and are unique to the last character. It is advisible to use reasonably long names to improve readability and facilitate others to understand what the source code is trying to achieve.

2.4 Syntax

Users without prior exposure to Lisp may find the use of s-expression based syntax rather cumbersome. But it is very logical and not difficult to get accustomed to.

3 MICRO

In this section an attempt is made to explain the various features of the MICRO data definition language. It tells the specification writer how to write his microcode.

First we shall present the abstract syntax of the MICRO assembly language. Then we shall examine each facet of the assembly language in detail in the the subsequent subsections.

3.1 Abstract Syntax of MICRO Assembly Language

```
%% The abstract syntax of the micro-assembly language is presented

%% It is evident that it is a data definition language which allows
%% the user to declare the fields and formats of the microinstructions

%% The syntax is basically LISP like ( I understand that it is cumbersome
%% with the myriad parantheses but believe me you will get used to it)

Lang -> Decl Micro-instr*

Micro-instr -> [Label] Micro-ops* | Directives*

Directives -> "(" ORIGIN Offset")"

| "(" RESERVE num ")"


Decl -> "(" RECORD recName (FieldName num BITS
                                   Implementation Default)* ")"*

| "(" ENUM FieldName Implementation ")"

Implementation -> Expn | Decl

Expn ->   num | Label | OR Expn Expn |CONCAT Expn Expn

Micro-ops ->   "(" CREATE-REC RecName
                                  (FieldName Implementation)* ")"*
                      |  "(" CREATE-REC RecName
                                 "("FieldName CREATE-REC RecName
"(" FieldName Implementation ")"* ")" ")"

Label -> id

FieldName -> id
```

```
RecName -> id

Offset -> Label "+" num | Label "-" num | num

        Default -> num

        id -> any valid Pascal identifier

        num -> any valid number in Common Lisp (binary,hex,octal or decimal)
```

3.2 Declaration

MICRO supports three user defined types.

1. *RECORD*: Similar to a Pascal record. The abstract syntax for a record declaration is

   ```
   (RECORD (RecName (field 1) (field 2) ... (field i) .. ))
   ```

2. *ENUM*: Semantically similar to enumerated data type in Pascal. The abstract syntax of the ENUM declaration is

   ```
   (ENUM (component-1 implementation-1)
   (component-2 implementation-2)
     (component-i implementation-i)
   )
   ```

   where component-i is an identifier and implementation-i is a litera bitvector.

3. *LITERAL*: could be a label or literal bit vector.

The user is expected to first define the general format of the microword. The microword is of type record (as in Pascal or C).It has a unique name and fields. Each field itself could be of any of the three types. Hence, the overall structure is that of a tree (heirarchical).

The abstract syntax for the field is as follows:

```
(FieldName FieldSize BITS
     Field-Implementation Default
)
```

FieldName is an unique identifier.

FieldSize is an integer.

Field-Implementation could be of Enumerated Type or a Record Type.

Default is again a literal bitvector which is used if a certain application does not use the field in question.

It would be nice to illustrate the above with a concrete example.

```
(RECORD (instr (dbus 4 bits (ENUM
(acc #x2 )
(mar 3))
(#xa))
(alu 4 bits (RECORD (function
 (shift 3  bits (7))
 (logical_op 1  bits
(ENUM (not #x0)
      (exor #b1))
  (#x0))))
      (#xb))
(next 4 Bits (8))
)
)
```

The name given to the microword is *instr*. It has three fields namely *dbus*,*alu* and *next*. The *dbus* field is of size 4 and is implemented as an enumerated type whose components are *mar* and *acc* and the default implementation of the *dbus* field is **#xa** which is a literal quantity specified in the hexadecimal notation. The components of the enumerated type *mar* and *acc* are in turn specified as literal bitvectors in decimal and hexadecimal notation. The *alu* field is of size 4 and is implemented as a record called *function* which in turn consists of two subfields *shift* and *logical-op* respectively. The *shift* subfield is implemented as a field called *shift* and the *logical-op* field is implementated as a enumerated type. Each of these subfields again has a fixed size and a default option as discussed above. Finally the last field of the *instr* microword is the *next* field which has a size of 4 and whose implementation is decimal 8. Note that for fields which are not implemented as records ore enumerated types there is no necessity for a default option.

3.3 Directive

The MICRO data definition language enables the user to specify directives to the YAMA microassembler to tailor the microwords in the control store to his/her requirements. The following directives are permissible.

- ORIGIN : It enables the user to change the location counter; so that the next microword could be placed at an address of his/her choice.

  Example:-

  ```
  (i) (ORIGIN #xffff)
  (ii) (ORIGIN label +/- offset)
  ```

  In the first directive the location counter is changed to the hex address `ffff` while in the second case the location counter is shifted relative to a previously specified label.
  RESERVE : This directive enables a user to skip some locations in the control store probably for future usage.

  Example:-

- `(RESERVE nloc)`

  The above directive is very simple: it advances the location counter by *nloc* locations.

3.4 Specification

Finally, we shall see how the user can write the microcode in MICRO after having defined the format of the microword as described in the declaration subsection and specified the required directives to the assembler. Note that in the absence of any specific ORIGIN directive in the begining, forces the assembler to start from address 0.

The general format of a microinstruction in MICRO is as follows

```
(create-rec instr
(dbus acc)
(alu CREATE-REC function (shift #b1) (logical-op not))
(next #xf)
)
```

The microinstruction format has been illustrated with an example based on the declaration which was described in the previous subsection. The microcode described above instantiates a structure of the type *instr* with portion of the microword designated by *dbus* represented by the implementation of *acc* and the portion designated by *next* represented by the bit pattern "1111". The *alu* portion of the microword is itself instantiated to a structure of the type *function* and its subfields represent the appropriate bit patterns corresponding to *shift* and *logical-op* fields.

The next example illustrates the use of label and the default option.

```
(fetch (create-rec instr
(dbus acc)
(alu CREATE-REC function (shift #b1) (logical-op not))
(next #xb)
)
)

(create-rec instr
(dbus mar)
(next fetch)
)
```

Note the definition of the label "fetch" in the first microinstruction and its use in the next field of the record type *instr*. This is valid because in the declaration of the record *instr* we have specified the implementation of the field *next* to be of type literal.

Also note that in the second microinstruction we have not specified any code for the *alu* field. It was totally omitted. It is perfectly valid because in the declaration of the *instr* record we have specified a default option for the *alu* field which will come handy now. The result is the compactness of the microinstructions. Probably, the current user has nothing to do with alu.

Finally, we shall discuss an example which portrays some of the advanced featurs supported by our MICRO data definition language.

```
(create-rec instr
(dbus OR mar acc)
(next fetch)
)

(create-rec instr
(dbus CONCAT mar acc)
(next fetch)
)
```

The "OR" and "CONCAT" keywords in the microinstruction instruct the YAMA microassembler to perform bitwise logical OR and concatenate operations on its arguments. The code produced for the *dbus* portion of the microword in the first microinstruction will be "0011" while in the second microinstruction it will be "1110". One very common application for such a feature would be in a microengine for a single bus architecture wherein you would like to push the contents of a particular register into more than one destinations at the same time.

4 YAMA - How it Works?

This section is intended for those who wish to make some changes to the source code at a later day. If you wish to merely use the YAMA, you can skip this section without any loss of contiuity.

YAMA is architecturally identical to any high level language compiler. It has the usual four phases.

1. **Lexical Analysis:**

It was our desire to keep the lexical anaylsis phase as clean as possible. Sowe adopted lisp like s-expression based syntax for the data definition language though it entails in the extra burden of matching parentheses, otherwise, the scanner itself would be an enormous amount of code resulting in complications in debugging and maintanence.
The outer level (user accessible) **`assemble`** function asks for the file name containing the source code and initialises the data-structures before passing the control to the function called **`do-assembly`** which reads the input and calls the workhorse function **`generate-code-for-micro-instruction`** which does the parsing based on the keyword in the input instruction.
Lexical errors are detected and appropriate error messages are printed on the screen.

2. **Syntax Analysis:**
   The simplicity of the data definition language renders the parser to be a simple function which looks at the keyword and branches to an appropriate procedure to parse the instruction. **`proc-org`** parses the ORIGIN directive while **`proc-res`** deals with the RESERVE directive. **`proc-rec`** function parses the declaration section of the source code. Finally, if no errors are detected during the course of the specification of declarations and the directives (i.e. no static errors) the assembler proceeds with the microinstructions. During the course of the parsing every effort is made to detect violations of the MICRO grammar and appropriate error messages are printed.
3. **Semantic Analysis:** As in most compilers, there is no definite demarcation between the semantic analysis and the syntax checking phase. Both are done almost concurrently. The things we look for here are the bounds on the field sizes, define before use restriction (for example in usage of labels), type of the field usage and declaration, duplicate definitions (of field names, labels etc) and correct usage of the various identifiers with respect to the semantics of the MICRO data definition language.
4. **Code Generation:** We follow the technique which is normally called "syntax directed translation". We process one

microinstruction at a time and produce the code which in our case is the address of the microinstruction in the control store and it implementation in the form of a bit vector. First the size of the bit vector is estimated from the declaration. Then we fill up the slots corresponding to all the specified fields and then finish off by filling up the unspecified slots with defaults. Anytime an error is detected, we simply abort the code generation by flashing a relevant error message.

Most of the functions in the source code are very well documented and self-explanatory.

5 Running YAMA

It is very simple. All you need is an account on any HP Bobcat. You should have /lisp/bin in your path. Then, invoke GNU-Emacs and do Meta-x "run-hpcl" and you are ready to use the wonders of CommonLisp. Then you would probably want to load the microassembler assuming that you have source program ready with you written in MICRO data definition language. An example of the terminal session is shown below.

```
                           Common Lisp


                   Part No. 98678A   Rev. 1.01
(c) Copyright 1986, Hewlett-Packard Company.  All rights reserved.

HP-UX 5.2 / Common Lisp, 22-Feb-88

(load "yama")
"yama"


(assemble)
Please enter the source file name .......>>example

REPORT: No. of Static Errors Detected =0
ASSEMBLY  IN  PROGRESS ..........
```

```
REPORT: No. of Static Errors Detected =0
ASSEMBLY  IN  PROGRESS ..........

REPORT: No. of Static Errors Detected =0
ASSEMBLY  IN  PROGRESS ..........

REPORT: No. of Static Errors Detected =0
ASSEMBLY  IN  PROGRESS ..........

REPORT: No. of Static Errors Detected =0
ASSEMBLY  IN  PROGRESS ..........

The control store looks like this ...........
ADDRESS                         MICROCODE
 0                         #(0 0 1 0 0 0 1 0 1 1 1 1)
 1111                      #(0 0 1 0 0 0 1 0 1 0 1 1)
 10000                     #(0 0 1 1 1 0 1 1 1 1 1 1)
 11011                     #(0 0 1 1 1 0 1 1 0 0 0 0)
 11100                     #(1 1 1 0 1 0 1 1 0 1 0 0)

ASSEMBLY WAS SUCCESSFUL
NIL


(pht *rht*)
 key = (FUNCTION INSTR)   value is ((SHIFT 3 4 (7)) (LOGICAL_OP 1 7 (0)))

 key = (INSTR NIL)   value is ((DBUS 4 0 (10)) (ALU 4 4 (11)) (NEXT 4 8

NIL


(pht *fht*)
 key = (SHIFT FUNCTION)   value is 7

 key = (DBUS INSTR)  value is ((ACC 2) (MAR 3))

 key = (LOGICAL_OP FUNCTION)   value is ((NOT 0) (EXOR 1))
 key = (NEXT INSTR)  value is 8

NIL



*label-list*
((FETCH 15))


*instr-num*
9
```

6 More Examples

In this section we shall illustrate the constructs of the MICRO assembly language with some more examples especially those which occur commonly in practice such as *conditional/unconditional jumps* and *register to register moves.* We also choose a sufficiently complicated microword format to illustrate our examples.

```
%% Here are some more examples to demonstrate the adequacy ofthe above
 constructs to write meaningful microcode

1. (ORIGIN offset)

Eg : (ORIGIN 16#200)

     (ORIGIN loop + 2#101000100)

2. (RESERVE num)

Eg : (RESERVE 16#36) % reserves 36hex locations
```

3. declaration

```
(RECORD (instr ( dbus 12 BITS (ENUM  ( mar #x800)
(mdr #x400)(r1 #x040)
(acc #x200)(ro #x080)) (0))
     (sbus  4 BITS (ENUM (mar #b1000) (mdr #x1001)
    (imm #b1010) (ro #b0000)
    (r1 #b0001)(r2 #b0010)) (0))
     (memory 2 BITS (ENUM (read #b01)
(write #b10)(nop #b00))(0))
     (alu-gen 7 BITS (RECORD (alu
(function 3 BITS
(ENUM (add #b0)
      (sub #b1)
      (and #b10)
      (or #b11)
      (pass.acc #b110)) (#xf))
(shift 3 BITS
(ENUM (shl #b0)
      (shr #b1)
      (ror #b10)
      (rol #b11)
      (no.shift #b110)) (#xf))
(latch 1 BITS
(ENUM (do.latch #b1)
(no.latch #b0)) (#xf))))(0))
    ( control 8 BITS (RECORD (ctrl
(source 2 BITS
(ENUM (f.dbus #b00)
      (f.imm #b10)
      (f.mpc #b10)
             (pass.acc #b110))(#xf))
(mask 6 BITS
(ENUM (zero #b100000)
      (pos #b001000)
      (neg #b000100)
      (carry #b000011)
  (true #b111111))(#xf))))(0))
```

```
     (next 13  BITS  (#x0))))

%% The above declaration depicts the various fields that the microinstru
%% have and their possible values.

%% The micro-ops will be merely to CREATE and instance of the RECORD "in
%% with the fields properly instantiated to the desired values


;;Eg: Unconditional Branch to a label

(CREATE-REC instr (dbus ro) (sbus ro)(memory nop)
   (alu-gen CREATE-REC alu
(function pass.acc)
 (shift no.shift)
 (latch no.latch)
 )
   (control CREATE-REC ctrl
   (source f.imm)
    (mask true)
    )
   (next label)
   )


;;Eg: Conditional Branch to a label

(CREATE-REC instr (dbus ro) (sbus ro)(memory nop)
   (alu-gen CREATE-REC alu
(function pass.acc)
 (shift no.shift)
 (latch no.latch)
 )
   (control CREATE-REC ctrl
   (source f.imm)
    (mask true)
    )
   (next label)
   )

%%  In the above example one can see that the branch condition is tested
%%  in the mask field.


;;Eg: move the contents of ACC to reg R1

(label (CREATE-REC instr (dbus r1) (sbus ro)(memory nop)
   (alu-gen CREATE-REC alu
(function pass.acc)
 (shift no.shift)
 (latch no.latch))
   (control CREATE-REC ctrl
   (source f.mpc)
   )
;; note that the mask subfield has been left unspecified here.
   (next #b0))
       )

%%  You can see the change in the dbus field value


;;Eg : AND reg R2 with ACC and store the result in R2

(CREATE-REC instr (dbus acc) (sbus r2)(memory nop)
   (alu-gen CREATE-REC alu
(function and)
 (shift no.shift)
 (latch no.latch)
 )

   (control CREATE-REC ctrl
   (source f.imm)
    (mask zero)
    )
   (next #xf)

   )
```

7 Errors

YAMA tries to be reasonably intelligent about catching errors. It detects most of the lexical, syntactic and semantic errors. Of course, the lexical analysis is mainly done by the Lisp Read function and not by the YAMA in particualr. Usually, all the static errors (those encountered during the parsing of the declaration/directive) sections of the source code are detected and the total count is expressed in the error message while the code generation is suppressed at the first sight of an semantic/run-time error (encountered during the parsing of the microinstruction).

An attempt is made to classify the type of error and indicate the place where things have gone wrong and also print the instruction number at which things got messy. But, it is not always accurate and it is advisable to look around the instruction where things went wrong.

Often, if the input is fouled up enough, it triggers a lisp error which starts with 5 exclamation marks. When this happens, the assembly aborts and it could mean mismatch in the parenthesis or improper/unintended use of s-expressions. Here it is advisable to go through the source program once again manually.

If you are familiar with CommonLisp, it is worth invoking the "backtrace" option or other debugging aids.

7.1 Debugging Aids

Sometimes, it useful to take a look at the various data structures (symbol tables) to see if right stuff has been placed at the right place. We essentially have two hash tables called the record hash table and the field hash table respectively which stores the various attributes of the records and the fields which have been discussed in the previous sections. The function **pht** with the hash table name as its arguments prints the contents of the desired hash table. In addition, we have an a-list to store the labels encountered in the source file along with their attributes.

`*label-list*` displays the labels currently in the memory while `*instr-num*` displays the current instruction number. The latter is very useful if you run into a Lisp Error (the one with !!!!!). You can make where things went wrong.

We hope these will be useful if the error messages are cryptic (or even vague).

7.2 Bugs

Though the microassembler has been tested for all its modes of behavious (means all the construct of the language), it cannot be called foolproof as the tests haven't been exhaustive or regressive. If you think you have found a bug you are advised to do a couple of things, First, make sure that your source program is syntactically correct and you haven't violated any of the rules of the MICRO grammar, then send a message to akella@humber, preferably with a short excerpt from the source file and a photo of what went wrong.

Wish You Luck !!