MakeCode: A Microcode Assembler

by

J. W. Peterson

Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

Last Revision: 10 December 1986

1. Introduction

MakeCode is an Rlisp program for creating microcode PROMS. It lets you give symbolic names to both ROM outputs and ROM locations, allows you to specify the polarity of ROM output signals, and maintains the literal field in the ROM.

To use MakeCode, you'll need to know how to create and edit a file on either the "ug" system or the DEC-20. If you're somewhat fuzzy on how to do this, ask the TA's.

Note:

All examples presented in this text are strictly examples of how to use MakeCode, and are *not* intended as examples of the actual project design.

2. Some things about Rlisp

Don't panic, you *do not* have to know anything about programming in Rlisp in order to use MakeCode. But MakeCode does take advantage of the Rlisp parser, so it helps to know how Rlisp reads what is fed into it.

2.1. Numbers

In addition to base 10, Rlisp lets you specify numbers in other bases as well, such as hex or binary. If you want to specify a number in a base other than decimal, you use the form:

*base***#***number*

where *base* is the base you want to express *number* in. For example:

```
2#11111110
16#FE
254
```

all specify the same number. Case is insignificant in hex digits, but upper case is usually more readable when mixed with digits.

2.2. Comments and white space

Anytime MakeCode (and Rlisp, for that matter) reads a percent sign (%), 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 to let others (e.g., the TA grading your lab book) know what's going on. MakeCode also ignores extra tabs, spaces and blank lines, so if an input line is two long to fit on one line you can place a newline anywhere a space would be legal, and tabs or spaces to indent it. Example (don't worry about the stuff in parenthesis yet):

```
% This is a comment
%
(Next (BusEna1 BusEna2)) % This is also a comment
(Next (Signal2 ReadReg   % This is split on multiple lines.
      StoreMem))
```

2.3. Identifiers

You will want to give things like signal names and some ROM locations symbolic names. In MakeCode, identifiers can be any length and are unique to the last character. In addition, the underscore can be used to space things apart. So:

```
Bus_Enable_2 LoadRegister Halt
```

are all valid names. Case is insignificant, so "LoadRegister" would be read the same as "loadregister" or "LOADREGISTER". While it is possible, it probably isn't a good idea to start names with a digit or use funny symbols like -,*,@,$,~ in symbol names. It's also best to avoid names already defined in MakeCode. Since symbol names can be of any length, try to use reasonably long

ones, since Very_Long_Name is much more understandable than VSN.

2.4. Parenthesis

As you saw in the example above, MakeCode input lines are surrounded by parenthesis. The general form of a MakeCode input line is:

(*directive arg1 arg2* **(** *signal1 signal2 ...* **))**

where *directive* is a predefined MakeCode name. Some of the MakeCode directives take one or two *arg*uments, and some take an optional list of *signals*. The signal list is surrounded with its own set of parenthesis, and an empty set of signals must be indicated with (). This will make more sense once we get into the specifics. If you have trouble matching up parenthesis, find out how to use "Lisp Mode" in DEC-20 Emacs, and the editor will help you out.

3. Defining the PROM outputs

3.1. Individual outputs

MakeCode assumes you are going to use at least two, and up to four 2716 PROMS in your machine. It also assumes the ten bit LIT (Literal) field is going to appear in ROM1 and the lower two bits ROM2. Note the numbering in the URCPU handout is backwards for this, it should look like:

```
 3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
|                                           |                 |
|       User defined signals                |     LIT field   |
```

While you can use a smaller LIT field, it must be in ROM1 and you will be responsible for making sure the values in the LIT field never exceed the number of bits you want to use.

The directive for defining pins is called DefPin, and its syntax is:

(DefPin *PinName* **(Rom***N* **.** *Value* **))**

The *PinName* is the symbolic signalname you want to assign to that rom pin (output). *N* refers to which rom you want this signal to be defined in, i.e. Rom1, Rom2, Rom3 or Rom4. *Value* is the numerical value assigned to that pin, always a power of two. Note the period between the Rom keyword and the numeric value, it must be there and it must be surrounded by spaces.[1] Some examples:

```
(DefPin Dev_Ctrl_0 (Rom2 . 2#100000))  % This is the sixth bit of Rom 2
(DefPin Dev_Ctrl_1 (Rom2 . 2#1000000)) % This is bit seventh bit of Rom 2

(DefPin Carry_Ena (Rom4 . 2#10))       % second bit of Rom 4
(DefPin Carry_Lit (Rom4 . 2#100))
```

Note the use of binary notation. It's best to have an individual definition for each ROM output you're going to use; you'll see how to lump them together with macros later on.

The signal names defined with DefPin are used in the signal lists mentioned earlier. If a signal name appears in the list, it means that signal is to be asserted at the currently specified ROM location. Signals not appearing won't be asserted. Normally, "asserted" means a one value, so when a signal appears in a signal list the corresponding ROM output is a one, otherwise it's a zero. There are however, cases where you want "asserted" to mean a zero value, and the signal to be a one otherwise. MakeCode has a directive called Invert that allows signal definitions to be inverted. Its

---

[1] You can use something other than a power of two (it doesn't check), but that will probably result in more confusion than it's worth. Using binary (2#xxx) notation helps keep you from screwing up pin values.

syntax is:

(Invert *PinName***)**

Where *PinName* is the name of the signal, previously defined with a DefPin, you want inverted. For example, suppose the "Dev_Ctrl" lines from the previous example were going to drive the enable lines of 74C244 drivers. We would probably want to invert the meanings so the enable line would go to zero if the signal was asserted. This would be declared with:

```
(DefPin Dev_Ctrl_0 (Rom2 . 2#100000))  % This is the sixth bit of Rom 2
(Invert Dev_Ctrl_0)                    % Invert the signal
(DefPin Dev_Ctrl_1 (Rom2 . 2#1000000)) % This is bit seventh bit of Rom 2
(Invert Dev_Ctrl_1)
```

3.2. Macros

Often you want to give a single name to a combination of signals, e.g. Add_Acc_To_Bus might be a combination of signals to the ALU, accumulator and buss control.[2]. A combination of signals can be lumped together with macros. Macros are defined with DefMacro, who's syntax is:

(DefMacro *MacroName* **(***Signal1 Signal2 ...***))**

The *MacroName* is the name assigned to the macro, and the signal list *(Signal1 Signal2 ...)* is a list of signals defined with DefPin, above.

Again using the "Dev_Ctrl" example, suppose these lines went to a decoder (e.g. 74C42) instead of directly to the bus drivers, and they were decoded so that:

00 Enabled the accumulator
01 Enabled the CPU
10 Enabled the "system" bus
11 Enabled the condition codes

First note we would probably not invert these. If we wanted to use separate macros for each of these functions, they would be defined with:

```
%
% Macros assume Dev_Ctrl_0 is the least significant bit
%
(DefMacro Acc_Enable ())        % Note use of null macro, for consistency.
(DefMacro CPU_Enable (Dev_Ctrl_0))  % This is essentially using two names
(DefMacro BUS_Enable (Dev_Ctrl_1))  % for one signal, for clarity in the code.
(DefMacro Cond_Enable (Dev_Ctrl_0 Dev_Ctrl_1))
```

Note signal names used in a DefMacro must appear earlier in the source file.

4. Defining the PROMs' contents

Once the outputs of the PROMs are defined with DefPins and DefMacros, another set of directives is used to define the contents. The first of these is Location, with the syntax:

(Location *Symbol RomAddress* **(***Signal1 Signal2 ...***))**

RomAddress is the 11 bit address you want to set the location counter to. *Symbol* is a symbolic name you assign to this ROM location. **(***Signal1 Signal2 ...***)** is a list of signals (or macros) you want asserted at that ROM location. For example:

```
(Location Start 16#20 (Carry_Ena Carry_Lit Cond_Enable))
```

would turn on (or if Invert was used, off) the ROM outputs associated with Carry_Ena, Carry_Lit and Cond_Enable at the ROM address of 20 (hex). In addition, the label "Start" would have the value of 20 (hex), so this location can be referanced by name in other places in the code, perhaps as a

[2]Until (and if) nested Macros are implemented, they probably won't get quite this fancy...

branch destination.

MakeCode maintains a location counter, much like the "." value in the PDP-11 assembler you used in 322. The Location directive sets "." to the value you specify. To place microcode in consecutive locations, the directive Next is used:

(Next (*Signal1 Signal2 ...***))**

where **(***Signal1 Signal2 ...***)** is a signal/macro list, as above. Next stores the pin values associated with all of the signals in the signal list in the PROM, at address specified by the Location counter. It then increments the location counter. For example, if the code:

```
(Next (Acc_Enable B_Reg_Load))
(Next (Add_B_Reg))
```

followed the Location in the previous example, the outputs associated with both Acc_Enable and B_Reg_Load would be placed in Rom location 21 (hex), and the outputs associated with Add_B_Reg would be placed in location 22 (hex).

Often you want to assign a symbolic name to a ROM location with without caring where its exact address is, or need a symbolic name for a number that doesn't reference a ROM address. This is done with the Label directive:

(Label *SymbolName* **LOC)**

or

(Label *SymbolName value***)**

The first case assigns the current value of the location counter to *SymbolName*. The second case assigns the numeric *value* to *SymbolName*. For example, the code:

```
(Label InitValue 0)
(Location Start 16#20 (Carry_Ena Carry_Lit Cond_Enable))
(Next (Acc_Enable B_Reg_Load))
(Label Add_B LOC)       % this could be a separate entry point to add B
(Next (Add_B_Reg))
```

results in the symbol *InitValue* having a value of 0, and *Add_B* a value of 22 (hex).

4.1. Assigning values to the LIT field

For microbranches and the like, a directive called SetLit exists. Its syntax is:

(SetLit *SymbolName* **(***Signal1 Signal2 ...***))**

or

(SetLit *Value* **(***Signal1 Signal2 ...***))**

In the first form, *SymbolName* is a symbol defined with Label or Location. In the second form, *Value* is a numeric constant. In both cases, **(***Signal1 Signal2 ...***)** is a list of signals or macros. For example, if Add_B is defined as above, then

```
(SetLit Add_B (Branch_Enable B_Reg))
```

asserts the signals associated with Branch_Enable and B_Reg at the current location, and also sets the low order two bits of rom2 to 2#00 and the contents of Rom1 to 16#22.[3]

The line:

```
(SetLit 0 (Alu_Init A_Reg_Init B_Reg_Init))
```

would place a zero in the Lit field of the current location.

---

[3]A synonym for SetLit is "Jmp".

5. How it fits together

Here's an example MakeCode input file to show how all this fits together

```
%
% To start off you have the initial output definitions
%
(DefPin Dev_Ctrl_0 (Rom2 . 2#100000))  % This is the sixth bit of Rom 2
(DefPin Dev_Ctrl_1 (Rom2 . 2#1000000)) % This is bit seventh bit of Rom 2
%
% Declare macros for the above
%
(DefMacro Acc_Enable ())        % Note use of null macro, for consistancy.
(DefMacro CPU_Enable (Dev_Ctrl_0))  % This is essentailly using two names
(DefMacro BUS_Enable (Dev_Ctrl_1))  % for one signal, for clarity in the code.
(DefMacro Cond_Enable (Dev_Ctrl_0 Dev_Ctrl_1))  % Condition code

(DefPin Carry_Ena (Rom4 . 2#10))       % second bit of Rom 4
(Invert Carry_Ena)                     % Declare it as inverted.
(DefPin Carry_One (Rom4 . 2#100))

(DefPin Alu_0 (Rom3 . 2#1))
(DefPin Alu_1 (Rom3 . 2#10))
%
% Some macros for the ALU
%
(DefMacro ALU_Add ())
(DefMacro ALU_Subtract (Alu_0))
(DefMacro ALU_Multiply (Alu_1))
(DefMacro ALU_Clear (ALU_1 ALU_0))
%
%
%
(DefPin TReg_Enable (Rom3 . 2#100))
(Invert TReg_Enable)                    % Perhaps wired to a 74C244...
(DefPin Treg_Read (Rom3 . 2#1000))
(DefPin Treg_Write (Rom3 . 2#10000))
%
(DefPin Return (Rom4 . 2#1))            % returns from a micro routine
(DefPin Branch (Rom4 . 2#1000))         % if asserted causes LIT to be
                                        % a micro-branch address
%
% After outputs are defined, you have the actual microcode
% A Location is needed initially to set the location counter
%
(Location Start 0 (CPU_Enable ALU_Clear))
%
(Label Add_T LOC)             % This will define Add_T to have a value of 1
(Next (Bus_Enable Carry_Ena))
(Label Add_T1 LOC)            % Another entry point.
(Next (Treg_Read Acc_Enable ALU_Add))
(Next (Treg_Write))
(Next (Return))
%
(Label Add_T_with_Carry LOC)
(Next (Bus_Enable Carry_One))
(SetLit Add_T1 (Branch))      % Executes a branch..
```

Again, this is just an example of how to use MakeCode and doesn't have any relation to the actual CPU design.

6. Running MakeCode

Running MakeCode breaks down to the following steps:

a. Fire up RLISP and load MakeCode

b. Assemble the file into MakeCode's memory

c. Get any debugging info you might need

d. Dumping the proms

e. Burning the proms

MakeCode files source files should reside on the Dec20, and the extension (part of the name after the dot) should be ".ucode". Here's an example, note all lines starting with "%" are comments interactivly typed at RLISP.

```
@psl:rlisp              start Rlisp
PSL 3.1 RLisp, 10-Apr-83
[1] load MakeCode;      % Load MakeCode program, note use of semicolons
NIL
[2] % Now we're ready to assemble the file "examp.ucode"
[2]
[2] asm "examp"         % Note the quotes, and the extension is left OFF
 ROMS are now assembled in Memory
 Ready for DumpProms
NIL
[3] % we didn't have any errors, and, justlike it says, we're ready to
[3] % dump the proms.  But suppose you wanted to watch what MakeCode was
[3] % doing (e.g., to track down errors).  You do this by saying:
[3]
[3] on TraceMode;
NIL
[4] % Now when you assemble the file, you'll see the code MakeCode reads,
[4] % and the results in the ROMS (if any).
[4]
[4] asm "examp";
(DEFPIN DEV_CTRL_0 (ROM2 . 32))
(DEFPIN DEV_CTRL_1 (ROM2 . 64))
(DEFMACRO ACC_ENABLE NIL)
(DEFMACRO CPU_ENABLE (DEV_CTRL_0))
(DEFMACRO BUS_ENABLE (DEV_CTRL_1))
(DEFMACRO COND_ENABLE (DEV_CTRL_0 DEV_CTRL_1))
(DEFPIN CARRY_ENA (ROM4 . 2))
(INVERT CARRY_ENA)
(DEFPIN CARRY_ONE (ROM4 . 4))
(DEFPIN ALU_0 (ROM3 . 1))
(DEFPIN ALU_1 (ROM3 . 2))
(DEFMACRO ALU_ADD NIL)
(DEFMACRO ALU_SUBTRACT (ALU_0))
(DEFMACRO ALU_MULTIPLY (ALU_1))
(DEFMACRO ALU_CLEAR (ALU_1 ALU_0))
(DEFPIN TREG_ENABLE (ROM3 . 4))
(INVERT TREG_ENABLE)
(DEFPIN TREG_READ (ROM3 . 8))
(DEFPIN TREG_WRITE (ROM3 . 16))
(DEFPIN RETURN (ROM4 . 1))
(DEFPIN BRANCH (ROM4 . 16))
(LOCATION START 0 (CPU_ENABLE ALU_CLEAR))
2#0: Rom1: 2#0 Rom2: 2#100000 Rom3: 2#111 Rom4: 2#10     The first number is the ROM address
(LABEL ADD_T LOC)
(NEXT (BUS_ENABLE CARRY_ENA))
2#1: Rom1: 2#0 Rom2: 2#1000000 Rom3: 2#100 Rom4: 2#0     Note inverted signal
(LABEL ADD_T1 LOC)
(NEXT (TREG_READ ACC_ENABLE ALU_ADD))
```

```
2#10: Rom1: 2#0 Rom2: 2#0 Rom3: 2#1100 Rom4: 2#10
(NEXT (TREG_WRITE))
2#11: Rom1: 2#0 Rom2: 2#0 Rom3: 2#10100 Rom4: 2#10
(NEXT (RETURN))
2#100: Rom1: 2#0 Rom2: 2#0 Rom3: 2#100 Rom4: 2#11
(LABEL ADD_T_WITH_CARRY LOC)
(NEXT (BUS_ENABLE CARRY_ONE))
2#101: Rom1: 2#0 Rom2: 2#1000000 Rom3: 2#100 Rom4: 2#110
(SETLIT ADD_T1 (BRANCH))
2#110: Rom1: 2#10 Rom2: 2#0 Rom3: 2#100 Rom4: 2#1010
 ROMS are now assembled in Memory
 Ready for DumpProms
NIL
[5] % This output is also handy for debugging the state machine.  If you want
[5] % a summary of what it thinks the pin definitions are:
[5]
[5] ListPins();
BRANCH = (ROM4 . 2#1000)
RETURN = (ROM4 . 2#1)
TREG_WRITE = (ROM3 . 2#10000)
TREG_READ = (ROM3 . 2#1000)
TREG_ENABLE = (ROM3 . 2#100) --Inverted
ALU_1 = (ROM3 . 2#10)
ALU_0 = (ROM3 . 2#1)
CARRY_ONE = (ROM4 . 2#100)
CARRY_ENA = (ROM4 . 2#10) --Inverted
DEV_CTRL_1 = (ROM2 . 2#1000000)
DEV_CTRL_0 = (ROM2 . 2#100000)
NIL
[6] % You can also look at the macro definitions...
[6]
[6] ListMacros();
ALU_CLEAR = (ALU_1 ALU_0)
ALU_MULTIPLY = (ALU_1)
ALU_SUBTRACT = (ALU_0)
ALU_ADD = NIL                              "NIL" is the same as "()"
COND_ENABLE = (DEV_CTRL_0 DEV_CTRL_1)
BUS_ENABLE = (DEV_CTRL_1)
CPU_ENABLE = (DEV_CTRL_0)
ACC_ENABLE = NIL
NIL
[7] %... and the symbols defined:
[7]
[7] ListSymbols();
ADD_T_WITH_CARRY = 5
ADD_T1 = 2
ADD_T = 1
START = 0
NIL
[8] % Once you're satisfied it's assembled O.K., dump the PROMS out to their
[8] % respective output files.  Again, give the first part of the filename with
[8] % NO extension, surrounded by quotes:
[8]
[8] DumpProms "examp";
Dumping prom: examp.prom1
Dumping prom: examp.prom2                  These are the names of the output files
Dumping prom: examp.prom3
Dumping prom: examp.prom4
Done.
NIL
[9] % MakeCode has now created four files.  These are then fed to PTPGEN and
[9] % downloaded to the PROM burner like you did in 427 last term.
[9]
[9] quit();     % Exit RLISP
@
```

7. Errors

MakeCode tries to be reasonably intelligent about catching errors. It is not, however, too intelligent once it has caught the error. In general, A MakeCode error message starts with three question marks, like:

```
??? Label: START_SYSTEM Is already defined
or
??? Unknown signal: ALU_LOSD
```

Often, if the input is fouled up enough, it triggers a lisp error in addition to a MakeCode error, something like[4]:

```
***** Non-numeric argument in arithmetic
```

When this happens, the assembly aborts. If you have trouble finding where the error is, use the

```
on TraceMode;
```

feature to help track it down. The errors should usually be pretty obvious, since the syntax of the assembler is quite simple. One of the less obvious ones is leaving out a parenthesis, this causes MakeCode to die with

```
***** Unexpected EOF while reading on channel '6'
```

Also, in some cases, if a DefPin (or DefMacro) is messed up, the error may not show up until the signal it defined is actually used. If MakeCode chokes on a line that "looks" right, make sure the definitions used in that line are correct.

7.1. Bugs

If you think you have found a bug in MakeCode, first show it to the TA. If he can't figure out what's wrong, *try to narrow down the problem* and have the TA send me (jw-peterson@utah-20) a message, preferably with a *short* excerpt from the source file and a photo of what went wrong.

[4] Lisp errors always start with five stars

8. Appendix: quick reference guide

8.1. MakeCode syntax

(DefPin *PinName* **(Rom***N* **.** *Value* **))**

```
(DefPin Dev_Ctrl_0 (Rom2 . 2#100000))  % This is the sixth bit of Rom 2
(DefPin Dev_Ctrl_1 (Rom2 . 2#1000000)) % This is bit seventh bit of Rom 2

(DefPin Carry_Ena (Rom4 . 2#10))       % second bit of Rom 4
(DefPin Carry_Lit (Rom4 . 2#100))
```

(Invert *PinName***)**

```
(Invert Carry_Ena)
```

(DefMacro *MacroName* **(***Signal1 Signal2 ...***))**

```
(DefMacro Acc_Enable ())        % Note use of null macro, for consistency.
(DefMacro CPU_Enable (Dev_Ctrl_0))  % This is essentially using two names
(DefMacro BUS_Enable (Dev_Ctrl_1))  % for one signal, for clarity in the code.
(DefMacro Cond_Enable (Dev_Ctrl_0 Dev_Ctrl_1))
```

(Location *Symbol RomAddress* **(***Signal1 Signal2 ...***))**

```
(Location Start 16#20 (Carry_Ena Carry_Lit Cond_Enable))
```

(Next (*Signal1 Signal2 ...***))**

```
(Next (Acc_Enable B_Reg_Load))
(Next (Add_B_Reg))
```

(Label *SymbolName* **LOC)**
or
(Label *SymbolName value***)**

```
(Label InitValue 0)
(Label Add_B LOC)       % this could be a separate entry point to add B
```

(SetLit *SymbolName* **(***Signal1 Signal2 ...***))**
or
(SetLit *Value* **(***Signal1 Signal2 ...***))**

```
(SetLit Add_B (Branch_Enable B_Reg))
(SetLit 16#FA (Add_B Clear_Enable))
```

8.2. MakeCode commands

Starting MakeCode (after logging in to DEC-20)
PSL:RLISP
Load MakeCode;

Assembling a file:
asm "*filename*"**;**

Dumping the prom output:
DumpProms "*filename*"**;**

Setting Trace Mode (done before ASM)
on TraceMode

Listing symbolic values:

ListPins(); Pin Values

ListMacros(); Macro names and their expansions

ListSymbols(); Symbol names and values

9. Shipping files from the UG to the DEC-20

There is only one DEC-20 line available for CS-428 use. Since much of the terminal time is spent simply editing your input file, it is encouraged[5] that you use UG to do the basic editing. Most of you should still have UG accounts from CS322, and the password probably hasn't changed. If you don't have an UG account, one can probably be dug up. You have officially been told there is no way to ship files from the UG to the DEC-20. There is, however, a loophole: the mail system. Suppose you had a file called "fred.ucode" on the ug that you want to ship to the 20. Here's how you send it:

```
3 ug> Mail cs428@utah-20          give the address as shown
Subject: file transfer for fred
~r fred.ucode                     The "~r" command reads the file into the message buffer
"fred.ucode" 37/1124              The mailer tells you the number of lines/characters it read
EOF                               type a ^D (control D) to end the message
4 ug>
```

The message (with your file in it) will show up on the 20 within the next ten to twenty minutes. Once it arrives, save the message with your file in it and then use EMACS on the 20 to edit out the header and tailer the mailer program stuck in.

Unfortunately, MakeCode doesn't run on the ug, mainly because it would take 2.5 UG's (not to mention a few hundred PDP-11 address spaces) to hold RLISP. So you'll have to ship your file to the 20 to see if it assembles.

[5] by your classmates, not necessarily the management

Table of Contents