GASM MANUAL

Peter Ashenden, Ashenden Designs
1 August 2010

INTRODUCTION

Gasm is a simple assembler for the Gumnut soft-core processor described in *Digital Design: An Embedded Systems Approach*. The Gumnut is a simple 8-bit processor with separate instruction and data memories. Gasm translates a program written in assembly language into the binary encoding required by the Gumnut core. The program text (the instructions) are encoded and written to a text memory image file, and the initial values for the program data are binary encoded and written to a data memory image file. The contents of these files can be loaded into simulated memories in a simulation of the Gumnut core, or can be programmed into physical memories in a real hardware system.

Gasm is a simple multi-pass assembler. It first parses the source file for syntax errors, with very rudimentary error checking. It then performs two passes over the program. On the first pass, Gasm works out the storage requirement for each instruction and directive and defines the values of labels. On the second pass, it translates each instruction and writes the output files.

The assembled instructions and data for the program can be written to the memory image files in a number of different formats, depending on the command line options. The names for the files can be specified on the command line. Gasm can also produce a file containing a symbol table listing each label in the program and its value. This can be helpful when debugging a program running on a Gumnut simulator.

INSTALLATION

Gasm is provided as a Java command-line application in a Java archive file, Gasm.jar. It uses the ANTLR parser, also provided in the file antlr.jar. You will need a Java runtime system that can run compiled Java code, such as Sun's JRE or JDK, both of which are available from java.sun.com.

Download the Gasm.jar and antlr.jar files to a directory on your computer. Note the name of the directory. In the following steps, we refer to it as <installpath>.

Set the CLASSPATH environment variable to include the two jar files: in Windows systems

```
<installpath>/Gasm.jar;<installpath>/antlr.jar
```

or in Unix/Linux systems

```
<installpath>/Gasm.jar:<installpath>/antlr.jar
```

COMMAND LINE USAGE

You can invoke Gasm with a command line of the following form:

```
java Gasm [-a | -m | -v | -x]
          [-t textfile] [-d datafile] [-s symfile]
          [file.gsm]
```

The square brackets indicate that the enclosed part of the command line is optional. The vertical bar indicates choice between options. The command line options and argument are as follows.

- -a: Specifies that the text and data memory images are to be written in Altera Memory Initialization File (MIF) format.
- -m: Specifies that the text and data memory images are to be written in Verilog hexadecimal memory load file format. This is the default if neither -v nor -m is specified.
- -v: Specifies that the text and data memory images are to be written in the form of VHDL package code that can be compiled into a VHDL model.
- -x: Specifies that the text and data memory images are to be written in a form usable in VHDL models processed by the Xilinx XST synthesis tool.
- -t *textfile*: Specifies the file name for the text memory image, that is, the file containing the program instructions. If the option is omitted, the default file name is gasm_text.vhd (if VHDL code format is used), gasm_text.mif (if Altera MIF format is used), or gasm_text.dat (if Verilog memory format or Xilinx XST format is used).
- -d *datafile*: Specifies the file name for the data memory image, that is, the file containing the initialized data used by the program. If the option is omitted, the default file name is gasm_data.vhd (if VHDL code format is used), gasm_data.mif (if Altera MIF format is used), or

gasm_data.dat (if Verilog memory format or Xilinx XST format is used).

- -s *symfile*: Specifies the file name for the symbol table. If the option is omitted, no symbol table file is produced.
- *file.gsm*: The name of the assembly language source file. If the file name is omitted, the default is to read the source program from the standard input. If multiple input files need to be assembled together they can be concatenated and piped into Gasm.

The command line options and source file argument can be written in any order.

COMMAND LINE EXAMPLES

```
java Gasm myProg.gsm
```

This assembles the program in myProg.gsm and writes the text and data memory images in Verilog hexadecimal memory load file format to the files gasm_text.dat and gasm_data.dat, respectively. No symbol table file is produced.

```
java Gasm -v myProg.gsm
```

This assembles the program in myProg.gsm and writes the text and data memory images as VHDL source code to the files gasm_text.vhd and gasm_data.vhd, respectively. No symbol table file is produced.

```
java Gasm myProg.gsm -m -t myProg_text.dat -d myProg_data.dat
```

This assembles the program in myProg.gsm and writes the text and data memory images in Verilog hexadecimal memory load file format to the files myProg_text.dat and myProg_data.dat, respectively. No symbol table file is produced.

```
java Gasm -t p1_text.dat -d p1_data.dat -s p1.sym p1.gsm
```

This assembles the program in p1.gsm and writes the text and data memory images in Verilog hexadecimal memory load file format to the files p1_text.dat and p1_data.dat, respectively. The symbol table is written to p1.sym.

```
cat p1.gsm p2.gsm p3.gsm | java Gasm -v
```

On a Unix/Linux system, this concatenates the three source files p1.gsm, p2.gsm, and p3.gsm, and pipes them into Gasm. The assembler

writes the text and data memory images as VHDL source code to the files gasm_text.vhd and gasm_data.vhd, respectively. No symbol table file is produced.

```
type p1.gsm p2.gsm p3.gsm | java Gasm -v
```

The Windows Command Prompt version of the preceding Unix/Linux command line.

ASSEMBLY PROGRAM FORMAT

A Gumnut assembly language program consists of a sequence of lines, each of which is a Gumnut instruction or an assembler directive. An Gumnut instruction is translated into its binary representation and written to the text memory image file. A directive controls how Gasm translates the program, or specifies data for the data memory image file.

Each line of a program can contain a comment, starting with a semicolon (';') character and extending to the end of the line. Comments serve as documentation for the human reader. They are ignored by the assembler. A line can contain just a comment (without an instruction or directive).

All instructions and several of the directives can include a label. The label is a name that represents a value: a memory address or some other specified value. A labels consists of a string of letters, numeric digits, and underscore ('_') characters, with the first character required to be a letter. Examples of illegal labels are:

```
L  L1   label_1   x_  distance_limit_9_a
```

Examples of illegal labels are:

```
9ab  _L  a-z
```

The first and second do not start with a letter, and the third includes a character that is not a letter, digit, or underscore. The case of letters in labels is significant. Thus, the labels start, Start, and START are distinct.

Many instructions and directives include expressions for specifying values. The format of expressions is very simple, consisting of a primary value, optionally followed by the addition or subtraction of further primary values. The first primary value can be preceded by a plus or minus sign. The primary values can takes the form of label names, decimal or hexadecimal numbers, or character literals. Some examples of expressions involving labels and decimal numbers are:

```
x1
-3
+a - 1
b + diff + 120
```

Hexadecimal numbers are written with the prefix "0x", for example, 0x3F. The case of the digits A through F is not significant. Thus, we would write the hexadecimal value 0xFC equally as 0xFc, 0xfC or 0xfc.

Character literals are written with the character enclosed in single quote marks, and represent the Latin-1 character code of the enclosed character. Thus, for example, the character literal 'A' has the value 65, 'µ' has the value 181, and ' ' (the space character) has the value 32. Certain control characters can also be written as character literals using a '\' character as an escape. They are:

'\n'	New line (line feed)
'\r'	Carriage return
'\t'	Horizontal tab
'\b'	Backspace
'\f'	Form feed
'\"'	Double quote
'\''	Single quote
'\\'	Backslash

The two directives ascii and asciz include string literals. The format of a string literal is a sequence of characters enclosed by double quote marks, such as "Ready". The backslash escapes listed above can also be included in a string literal, for example, "Press \"OK\" to exit\n".

INSTRUCTIONS

The Gumnut instruction set is described in *Digital Design: An Embedded Systems Approach*. In the following descriptions, square brackets are used to indicate optional parts. The format of an instruction in a Gasm program is:

```
[label:] opcode [operands]
```

The label, if included, is set to the address of the instruction in the text memory image. The opcode specifies one of the Gumnut instructions. If the instruction requires operands, they are written after the opcode. The opcodes and operands for Gumnut instructions are listed in Table 7.1 of *Digital Design: An Embedded Systems Approach*. Immediate operands,

shift counts, offsets, displacements and addresses are written in the form of expressions, described in "Assembly Program Format" above.

DIRECTIVES

The directives recognized by Gasm are described in this section. In the following descriptions, square brackets are used to indicate optional parts, and curly braces are used to indicate optional repetition (zero or more times).

THE TEXT DIRECTIVE

The text directive changes assembly to the text memory image. The format of the directive is:

```
text
```

Gasm maintains a text location counter representing the next address in the text memory image to be filled. It is initialized to 0 at the start of assembly. As each instruction is translated, it is written to the text memory image at the address given by the text location counter, and the text location counter is incremented. An instruction can only appear in a Gasm program if the text memory image is selected. Assembly starts with the text memory image selected.

THE DATA DIRECTIVE

The data directive changes assembly to the data memory image. The format of the directive is:

```
data
```

Gasm maintains a data location counter representing the next address in the data memory image to be filled. It is initialized to 0 at the start of assembly. As byte, ascii, asciz and bss directives reserve space in the data memory, the data location counter is incremented by the amount of space reserved. Those directives can only appear in a Gasm program if the data memory image is selected. Thus, a data directive must precede them.

THE ORG DIRECTIVE

The org directive sets the currently selected location counter to a specified value. The format of the directive is:

```
org expression
```

The expression is evaluated and the currently selected location counter is set to the result. The value must be within the range of valid addresses for the currently selected memory: 0–4095 for text memory addresses and 0–255 for data memory addresses.

Example—The following directive sets the current location counter to the value 0x10:

```
        org  0x10
```

THE EQU DIRECTIVE

The equ directive defines a label and sets it to a specified value. The format of the directive is:

label: equ *expression*

The label is defined to have the value of the expression.

Example—The following directive defines the label input_reg to have the value 0x08:

```
input_reg:  equ  0x08
```

THE BYTE DIRECTIVE

The byte directive reserves one or more locations in the data memory image and initializes them to specified values. The format of the directive is:

[*label*:] byte *expression* {, *expression*}

The label, if present, is defined to have the value of the data location counter, which is the address of the first data memory location reserved. The number of locations reserved is the number of expressions. Each expression is evaluated, and the results are used to initialize successive locations in the data memory image. The data location counter is incremented by the number of locations.

Example—The following directive reserves one location labeled count and initializes it to 0:

```
count:  byte  0
```

Example—The following directive reserves two locations, with the first labeled CRLF, and initializes them to the character codes for a carriage return and a line feed:

```
CRLF:  byte  '\r', '\n'
```

Example—The following directive reserves eight locations, with the first labeled primes, and initializes them to the first eight prime numbers:

```
primes:  byte  2, 3, 5, 7, 11, 13, 17, 19
```

THE ASCII DIRECTIVE

The ascii directive reserves locations in the data memory image and initializes them to the character codes of the characters in a specified string literal. The format of the directive is:

```
[label:] ascii string
```

The label, if present, is defined to have the value of the data location counter, which is the address of the first data memory location reserved. The number of locations reserved is the number of characters in the string. The data location counter is incremented by the number of locations.

Example—The following directive reserves locations for a string to be displayed on an output device:

```
PIN_message:  ascii  "Enter PIN: "
```

THE ASCIZ DIRECTIVE

The asciz directive reserves locations in the data memory image and initializes them to the character codes of the characters in a specified string literal. It reserves an additional location, after those reserved for the character codes, that it initializes with the value 0 (the Latin-1 NUL character). The format of the directive is:

```
[label:] asciz string
```

The label, if present, is defined to have the value of the data location counter, which is the address of the first data memory location reserved. The number of locations reserved is one more than the number of characters in the string. The data location counter is incremented by the number of locations.

Example—The following directive reserves locations for a string with a further location: reserved for a terminating 0 byte:

```
manufacturer_name:  asciz  "Widgets Inc"
```

THE BSS DIRECTIVE

The bss ("block starting with symbol") directive reserves a specified number of locations in the data memory image without initializing them. The format of the directive is:

label: bss *expression*

The label is defined to have the value of the data location counter, which is the address of the first data memory location reserved. The number of locations reserved given by the value of the expression. The data location counter is incremented by the number of locations.

Example—The following directive reserves a single location for use by the program:

```
result:  bss  1
```

Example—The following directive reserves 4 locations for use by the program:

```
PIN_buf:  bss  4
```

LOADING THE OUTPUT FILES

The memory image files can be written in various formats, depending on how they are to be used by simulation or synthesis tools.

VHDL SOURCE CODE FORMAT

If the -v command line option is used, the memory images files are written in the form of VHDL source code. This is useful for synthesis tools that do not provide a method of loading memories with data from files.

The code for the text memory image defines a VHDL package as follows:

```
library ieee; use ieee.std_logic_1164.all;
use work.gumnut_defs.all;
package gasm_text is
  constant program : IMem_array := (
    0 => "1111000000000010000",
    1 => "1111110001000000000",
    ...
    4094 => "0000000000000000000",
    4095 => "0000000000000000000" );
end package gasm_text;
```

A model can initialize the storage for instruction memory using the value of the constant defined in the package, for example:

```
constant IMem : IMem_array := work.gasm_text.program;
```

The code for the data memory image similarly defines a VHDL package:

```
library ieee; use ieee.std_logic_1164.all;
use work.gumnut_defs.all;
package gasm_data is
  constant data : DMem_array := (
    0 => "00000000",
    1 => "00000000",
    ...
    254 => "00000000",
    255 => "00000000" );
end package gasm_data;
```

Likewise, a model can initialize the storage for the data memory using the value of the constant in this package:

```
signal DMem : DMem_array := work.gasm_data.data;
```

VERILOG MEMORY FILE FORMAT

If the -m command line option is used, the memory images files are written in Verilog hexadecimal memory load-data format. Verilog simulation and synthesis tools can read such files using the $readmemh system task. For example, the instruction and data memories in a Verilog model can be defined and initialized as follows:

```
reg [17:0] IMem [0:4095];
reg  [7:0] DMem [0:255];
...

initial $readmemh("gasm_text.dat", IMem);
initial $readmemh("gasm_data.dat", DMem);
```

XILINX XST READ FORMAT

If the -x command line option is used, the memory images files are written in such a way that simple VHDL code to read the files can be recognized by the Xilinx XST synthesis tool.

The text memory image file consists of one line per location, from address 0 through address 4095. Each line contains the 18-bit content of the

location expressed as string of 0 and 1 digits. The following code is recognized by XST as initialization of the instruction memory:

```
library ieee;
use ieee.std_logic_1164.all, ieee.numeric_std.all;
use std.textio.all, ieee.std_logic_textio.all;
...

  impure function load_IMem (IMem_file_name : in string)
                            return IMem_array is
    file IMem_file : text is in IMem_file_name;
    variable L : line;
    variable instr : std_logic_vector(instruction'range);
    variable IMem : IMem_array;
  begin
    for i in IMem_array'range loop
      readline(IMem_file, L);
      read(L, instr);
      IMem(i) := unsigned(instr);
    end loop;
    return IMem;
  end function;

  constant IMem : IMem_array := load_IMem("gasm_text.dat");
```

The data memory image file format is similar, consisting of one line per address from 0 to 255. Each line contains 8-bit binary data. The following code, similar to that for the instruction memory, is recognized by XST as initialization of the data memory:

```
library ieee;
use ieee.std_logic_1164.all, ieee.numeric_std.all;
use std.textio.all, ieee.std_logic_textio.all;
...

  impure function load_DMem (DMem_file_name : in string)
                            return DMem_array is
    ...
  begin
    ...
  end function;

  signal DMem : DMem_array := load_DMem("gasm_data.dat");
```