Selective Automation of GNU Binutils Assembler and Linker for Custom SPARC-v8 Instruction Sets

ELG7199 Final Project Report

Jonathan Parri
3750685
December 19th 2008

Supervisors: Prof. Miodrag Bolic & Prof. Voicu Groza
# Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Assembly to memory initialization file sub-toolchain.</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>Basic allocation with 2 memories.</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>SPARC unsigned Division instruction format.</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>Assembly to Machine code.</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>SPARC ld Instruction verification.</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>Linking view of ELF object file.</td>
<td>12</td>
</tr>
<tr>
<td>7</td>
<td>Example of Extracted Section Headers.</td>
<td>13</td>
</tr>
<tr>
<td>8</td>
<td>Symbol resolution From Assembler to Linker.</td>
<td>14</td>
</tr>
<tr>
<td>9</td>
<td>Linker Input and Output.</td>
<td>14</td>
</tr>
<tr>
<td>10</td>
<td>Memory Map Example.</td>
<td>15</td>
</tr>
<tr>
<td>11</td>
<td>Linker Address Remapping and Resolution Example.</td>
<td>16</td>
</tr>
<tr>
<td>12</td>
<td>Application Flow.</td>
<td>17</td>
</tr>
<tr>
<td>13</td>
<td>Application Interactions.</td>
<td>17</td>
</tr>
<tr>
<td>14</td>
<td>Application Interactions.</td>
<td>17</td>
</tr>
<tr>
<td>15</td>
<td>Application Package Diagram.</td>
<td>18</td>
</tr>
<tr>
<td>16</td>
<td>Class Diagram.</td>
<td>19</td>
</tr>
<tr>
<td>17</td>
<td>READELF Section Machine Code Extraction.</td>
<td>20</td>
</tr>
<tr>
<td>18</td>
<td>Interaction between Classes, Folders and files.</td>
<td>21</td>
</tr>
<tr>
<td>19</td>
<td>Generated Memory Map Output.</td>
<td>22</td>
</tr>
<tr>
<td>20</td>
<td>AND Instruction Machine Code Output.</td>
<td>23</td>
</tr>
<tr>
<td>21</td>
<td>Custom Assembly Mneumonic with Matching Instruction Format.</td>
<td>24</td>
</tr>
<tr>
<td>22</td>
<td>Dissassembly of Executable with Custom Instruction.</td>
<td>24</td>
</tr>
<tr>
<td>23</td>
<td>Example System Layout</td>
<td>25</td>
</tr>
<tr>
<td>24</td>
<td>Address mapping with 2 Memories.</td>
<td>25</td>
</tr>
<tr>
<td>25</td>
<td>Program startup Requesting Memory Architecture.</td>
<td>27</td>
</tr>
<tr>
<td>26</td>
<td>Entering Assembly Files.</td>
<td>27</td>
</tr>
<tr>
<td>27</td>
<td>Automatic Placement in Memory.</td>
<td>28</td>
</tr>
<tr>
<td>28</td>
<td>Memory Unit Selection.</td>
<td>28</td>
</tr>
<tr>
<td>29</td>
<td>Memory Unit 1 Address Space Usage.</td>
<td>30</td>
</tr>
<tr>
<td>30</td>
<td>Memory Unit 2 Address Space Usage.</td>
<td>30</td>
</tr>
<tr>
<td>31</td>
<td>Compiled Binutils Executables.</td>
<td>34</td>
</tr>
</tbody>
</table>
TABLE OF TABLES
Table 1: Binutils tool collection [1] ............................................................................................................................... 7
Table 2: SPARC Instruction Categories. ....................................................................................................................... 10
Table 3: Commonly Used ELF Sections [3]. .................................................................................................................. 13
Table 4: Example of Unused Opcodes. ........................................................................................................................ 23

TABLE OF CODE
Code 1: Linker Script Example. .................................................................................................................................... 15
Code 2: MEMORY Command Usage. ........................................................................................................................... 16
Code 3: Get System Memory Information................................................................................................................ 19
Code 4: Calling the Pre-Compiled Assembler. ............................................................................................................ 20
Code 5: Calling the Pre-Compiled Linker. .................................................................................................................. 20
Code 6: Writing the Default Linker Script ................................................................................................................ 21
Code 7: Writing a Section Customizable Linker Script.............................................................................................. 21
Code 8: AND Instruction Format and Opcode ........................................................................................................ 23
Code 9: Custom Multiplier Instruction Set Extension ............................................................................................ 24
Code 10: Assembly with Custom Instruction .......................................................................................................... 24
Code 11: Processor 1 C Code .................................................................................................................................... 25
Code 12: Processor 2 C Code .................................................................................................................................... 25
Code 13: Processor 1 Assembly Code ........................................................................................................................ 26
Code 14: Processor 2 Assembly Code ......................................................................................................................... 26
Code 15: Generated PROC 1 Linker Script. .............................................................................................................. 29
Code 16: Generated PROC 2 Linker Script. .............................................................................................................. 29
Code 17: Machine Code Extraction For PROC 1. ................................................................................................. 29
Code 18: Machine Code Extraction for PROC 2 ................................................................................................. 29
Code 19: Memory Unit 1 MIF File for PROC 1. ......................................................................................................... 30
Code 20: Memory UNIT 2 MIF File for PROC 2. ................................................................................................. 30
ABSTRACT

Compiler tools are often offered as standalone executables but are hidden from the developer through dense layers of abstraction. To truly tailor a tool for ones needs, the root components must be examined and clearly understood. A set of tools, GNU Binutils, is an often overlooked component despite forming the resilient backbone of the GNU Compiler Collection (GCC).

This paper looks to automate available tools through a Java interface while allowing for the customization of instructions. This customization is of particular importance to embedded system designers where application specific processors can be used. Embedded systems often have various memory architectures and must be supported as well. This customization becomes imperative when an OS employing virtual memory is not used.

A definitive separation exists between these tools and a typical end user where design consideration control can be lost. Other considerations involve the use of other compilers besides GCC that wish to use both the assembler and linker provided within. Through completion of a Java compliant frontend with Windows ready custom Binutils components a definitive solution has been constructed offering an easily deployable componentwise alternative for developers. Only assembly files and a description of the system’s memory space are required. The software approach serves as a methodology for guiding the process automation to a more object-oriented feel.
INTRODUCTION

OVERVIEW

The Selective Automation of GNU Binutils Assembler and Linker for Custom SPARC-v8 Instruction Sets encompasses a small portion of a larger project, a complete multiprocessor system generation tool chain, Software/Hardware Implementation and Research Architecture (SHIRA). A user need only supply their program and a series of constraints, whether they are hardware cost, power consumption or other. The system uses the user-defined constraints as required system characteristics. A number of techniques may be used to meet the user specifications. The system may introduce a multiprocessing system with custom topology requiring adequate code partitioning. If this is too costly custom instructions may be introduced within the processor. If constraints are specified loosely or are automatically met with the default system, no further solution searching is required. Despite various attempts a solution may never be found if constraints are overly strict.

Figure 1 illustrates a particular portion of the tool chain designed to cater to augmented SPARC\textsuperscript{1} v8 instruction sets, creating machine code, using custom linker scripts to redefine code based on available memory units in a target platform and deploying the code to Altera or Xilinx defined memory. We will be focusing on all areas outlined in red defining the scope of this document and project.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{assembly_to_memory_init_file_sub_toolchain.png}
\caption{Assembly to Memory Initialization File Sub-Toolchain.}
\end{figure}

\textsuperscript{1} Scalable Processor ARCitecture
Using the COINS compiler developed by the Tokyo Institute of Technology and modified by the Computer Architecture Group (CARG) at the University of Ottawa an assembly file can be generated from a user-created C program. The assembler must be informed of the instruction set to consider when generating an object file\(^2\). A base sparc-elf target is used but can be augmented for custom instructions and opcodes. Besides the assembly file, a description of the memory space is required as an input. Systems may exist with one or more memories, a variety of placement options or shared memory needs. Data, instruction and the stack should be permitted to be placed at the user’s discretion. This application permits the user to individually place these sections or have the application dynamically allocate them to a given memory. Consider Figure 2 where two programs are placed in a system with two separate memories. No sharing is taking place. The first memory unit is situated within 0x0000 to 0x1000 while the second unit goes from 0x1000 to 0x2000.

![Figure 2: Basic Allocation with 2 Memories.](image)

The user specifies the number of memory units and the appropriate start and end addressing scheme. With this, linker scripts are generated based on the new memory environment. The linker uses these scripts to modify the memory addressing of the object file and links the object code to other libraries and object files based on remaining symbols.

Once the linker has finished resolving dependencies and matching memory addresses specified by the user an external memory initialization generator is called where Altera (.mif) or Xilinx (.mem) memory files are generated. These files can be deployed directly to memory at system synthesis time. Note that before the memory initialization files can be generated data must be extracted from the object files in a human-readable form. Appropriate parsing then takes place on used sections to meet Altera and Xilinx standards.

\(^2\) Object code, object file and machine code are used interchangeably.
GNU Binutils

As part of the GNU operating system a set of programming tools were developed for dealing with object code in a variety of formats. This package of tools is named Binutils. Its content applications are described in Table 1.

**TABLE 1: BINUTILS TOOL COLLECTION [1].**

<table>
<thead>
<tr>
<th>Binary Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr2line</td>
<td>Converts addresses into filenames and line numbers.</td>
</tr>
<tr>
<td>ar</td>
<td>A utility for creating, modifying and extracting from archives.</td>
</tr>
<tr>
<td>as</td>
<td>GNU assembler.</td>
</tr>
<tr>
<td>c++filt</td>
<td>Filter to demangle encoded C++ symbols.</td>
</tr>
<tr>
<td>dlltool</td>
<td>Creates files for building and using DLLs.</td>
</tr>
<tr>
<td>gold</td>
<td>ELF only linker.</td>
</tr>
<tr>
<td>gprof</td>
<td>Displays profiling information.</td>
</tr>
<tr>
<td>ld</td>
<td>GNU linker.</td>
</tr>
<tr>
<td>nlmconv</td>
<td>Converts object code into an NLM.</td>
</tr>
<tr>
<td>nm</td>
<td>Lists symbols from object files.</td>
</tr>
<tr>
<td>objcopy</td>
<td>Copies and translates object files.</td>
</tr>
<tr>
<td>objdump</td>
<td>Displays information from object files.</td>
</tr>
<tr>
<td>ranlib</td>
<td>Generates an index to the contents of an archive.</td>
</tr>
<tr>
<td>readelf</td>
<td>Displays information from any ELF format object file.</td>
</tr>
<tr>
<td>size</td>
<td>Lists the section sizes of an object or archive file.</td>
</tr>
<tr>
<td>strings</td>
<td>Lists printable strings from files.</td>
</tr>
<tr>
<td>strip</td>
<td>Discards symbols</td>
</tr>
<tr>
<td>windmc</td>
<td>A Windows compatible message compiler.</td>
</tr>
<tr>
<td>windres</td>
<td>A compiler for Windows resource files.</td>
</tr>
</tbody>
</table>

For this particular project we are mainly concerned with the GNU assembler (as) and GNU linker (ld). Other tools are used but simply acquire data for the backing software system. All tools were compiled from source code for a Windows environment. At compilation time an architecture target is required. This target specifies the type of object files that the Binutils tools will work with. In our case sparc-elf is used. This means that SPARC assembly code is required from the user. This assembly code is converted into a SPARC Executable and Linkable Format (ELF) file which is a form of object code. The SPARC mnemonics are translated into appropriate SPARC opcodes which can be read by a SPARC based processor. At compilation time Binutils reads an opcode file that lists all the mnemonics, their appropriate opcodes and instruction formatting. By adding to the SPARC opcode file listing custom instructions can be added. At this point instruction augmentation can only be done at compile time. It should be noted that if custom instructions are added the base processor must be modified to support the new opcodes, instructions and formatting. See Appendix A: Compiling GNU Binutils 2.19 From Windows for directions to compile the latest GNU Binutils v2.19 from source.
LITERATURE REVIEW

AUTOMATIC LINK EDITOR GENERATION FOR EMBEDDED CPU CORES [8]

While performing design space exploration it is often imperative to explore CPU alternatives requiring a high number of code generations. Despite this it is impossible to create an entire code generation toolkit for every targeted system especially when Application Specific Instruction Set Processors (ASIP) are involved. Using the GNU Binutils package this paper describes techniques for creating retargetable link editors also known as linkers. Architecture independent libraries are used to generate dependent ones which define the target at compile time.

The creation of a linker is an intensive laborious task as it depends greatly on the target architecture. A large divide exists between desktop and embedded markets as desktop systems use only a handful of architectures while the embedded section uses a great many including the ever popular ASIPs. In the hopes of making automatic link generation a viable endeavor 3 caveats are introduced by the authors. These include the reuse of formatters for relocation, section unification and the actual implementation through modification of the GNU Binutils linker.

The reuse of formatters stems from Architecture Description Language (ADL) portions dealing with relocation. Formatters can be used in a C like syntax and associated with instruction fields. Section unification involves the unification of only key sections within an object file: .text, .data and .bss. The physical implementation involves using the ELF library contained within the GNU BFD to generate an architecture dependent library. The architecture dependent library is tasked with holding relocation directives in a table for resolution purposes. To generate the libraries, for each instruction added the ADL description is searched. From the ADL the format and expression fields are extracted. If a format is found a relocation directive is generated. If one already exists it is replaced. When the code is assembled the relocation information must be produced as described in the relocation table.

To test out the new tool chain a variety of Mibench benchmarks were used as they are designed for embedded systems. In tests only the standard MIPS, SPARC and PowerPC were targeted. Verification showed both efficiency and robustness of results after specifying the architecture in ArchC.

The described tool shows its merit if a target architecture is entirely new. This is often not the case as many targets even for embedded systems are available and described in Binutils. The traditional SPARC, ARM, PowerPC and MIPS are only a few of the popular architectures described. The main issue that this paper tries to bring to light is the use of ASIPs. An ASIP is not usually an entirely new processor with an entirely new instruction set; it is based on an existing architecture and extended for the specific application. The Selective Automation of GNU Binutils Assembler and Linker for Custom SPARC-v8 Instruction Set, this project, introduces efficient techniques for augmenting existing architectures without having to generate entirely new architectures using ADL. Default relocation directives along with custom linker scripts and moderate configuration modification can be used instead.
EXTENDING THE ARCHC LANGUAGE FOR AUTOMATIC GENERATION OF ASSEMBLERS [9]

Continuing with the theme of the first reviewed paper ADLs can be used to generate custom assemblers. A strong ADL implementation is ArchC (based on SystemC) which allows a user to specify an architecture in the C programming language.

The central exploit described in this paper is the extension of ArchC to support descriptions of assembly language syntax. As a processor’s ISA changes, ADL can be used to reflect this in the compiler toolkit. At this point ArchC lacks both assembly syntax and operand encoding support. Using the ADL description with extended constructs a compatible assembler can be generated.

The user’s ADL is divided into two portions: a structural description and an ISA description. The structural description defines concrete facets of the processor such as memory size and register banks, while the ISA description deals with instruction formatting and decoding. All of the extensions introduced in this paper are part of the ISA description. Three new constructs have been added to help describe a complete architecture: register mapping and declaration, instruction syntax with operand encoding and the creation of synthetic instructions.

Using the augmented ArchC, compiler tools can be automatically generated. A processor model written in ArchC is entered into the system with a target specific assembler being generated. The assembler is not compiled automatically. The tool creates files that need to be placed within the directory hierarchy of GNU Binutils at pre-compilation time.

Using the generated files, the Binutils’ assembler was compiled and tested for both MIPS and SPARC architectures. Both execution and ELF sectional comparisons were performed. Equivalent object files were generated.

RETARGETABLE BINARY UTILITIES [10]

This older paper introduces many key concepts and facts to consider when customizing and retargeting GNU Binutils. All major components are discussed: assembler, linker, library manager, ELF extractor and manipulator. The authors begin by introducing an abstract model of an ISA and capturing the information required to perform a retargeting. The paper serves to compliment ADL not mimic it on a syntactical basis.

Not only is information about an architecture’s structure needed but an understanding of how object files are to be manipulated. Discussed are the internals of Binutils involving the Binary File Descriptor Libraries (BFD) and opcode libraries. The generic base assembler and linker hook into all these libraries for architecture specific routines.

Using available knowledge and a proprietary tool (internal language Babel) created by these students they solve the problem of reformulating the Binutils package to a more automated process when targeting custom systems.
ASSEMBLER

An assembler accepts an assembly file and converts it into machine code readable by the processor. The assembler translates mnemonics into opcodes and resolves symbols particularly those pertaining to memory instructions. The assembler must also consider instruction scheduling to optimize processor pipelining. Assemblers can be generic where the target architecture is user specified or they can be predefined for a specific instruction set.

The SPARC v8 architecture is of interest for this specific application. SPARC in and of itself is in fact an architecture derived from Reduced Instruction Set Computer (RISC) implementations. SPARC claims high execution rates and short time-to-market development schedules while targeting optimizing compilers and easily pipelines hardware implementations [2]. SPARC offers 72 basic instruction operations defined as opcodes for the processor. All instructions are 32-bit wide. These instructions fall into 6 basic categories shown in Table 2. Note that Load/Store instructions are the only ones that can access memory.

TABLE 2: SPARC INSTRUCTION CATEGORIES.

<table>
<thead>
<tr>
<th>SPARC Instruction Categories</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load/Store</td>
</tr>
<tr>
<td>Arithmetic/Logical/Shift</td>
</tr>
<tr>
<td>Control Transfer</td>
</tr>
<tr>
<td>Read/Write Control Register</td>
</tr>
<tr>
<td>Floating-Point Operate</td>
</tr>
<tr>
<td>Coprocessor Operate</td>
</tr>
</tbody>
</table>

The assembler must somehow understand the entire target system architecture. For example Figure 3 illustrates the assembly language syntax for an unsigned division. The mnemonic is analyzed and used to set up a 32-bit machine code instruction. This division instruction can be done on either an immediate or register value. The assembler must determine whether the desired source value is an immediate value or register and set bit 13 accordingly.

udiv rsl, rs2 or immediate value, rd

udiv opcode = 001110

FIGURE 3: SPARC UNSIGNED DIVISION INSTRUCTION FORMAT.
GNU ASSEMBLER

The GNU assembler also known as Gas can operate with many architectures. Gas has been compiled and targeted for a sparc-elf object file. At compile time a SPARC opcode file is read specifying the underlying operational requirements of the architecture. Figure 4 shows an input assembly file with the corresponding machine code extracted from an object file. It can be seen that both the instruction and data memory will overlap. This will be resolved later through a dynamic remapping according to specified memory addresses. The or operations have been equated to mov operations since the final symbol resolution has not occurred. After the code runs through the linker mov will be returned to or and appropriate values will be generated. Registers are categorized as in, local, out and global. The registers %i[0-7] are registers covering r[24-31]. The local registers span r[16-23] while the out registers are situated at r[8-15]. The global registers can be accessed by any instruction at anytime. They are located from r[0-7]. r[7-31] exist within a 24-register window.

```
.section " .text"
.align 4
.global main
main:
save %sp, -96, %sp
sethi %hi(b), %i0
or %i0, %lo(b), %i0
ld [%i0], %i1
sethi %hiI, %i0
or %i0, %loI, %i0
ld [%i0], %i0
add %i1, %i0, %i0
sethi %hi(t1), %i1
or %i1, %lo(t1), %i1
st %i0, [%i1]
.L3:
ret
restore

.section " .data"
.align 4
.global a
a:
.word 0
.align 4
.global b
b:
.word 1
.align 4
.global c
c:
.word 2
.common t1,4,4
```

For verification purposes consider the ld[%i0],%i0 instruction. The instruction says to use the address located within i[0](r[24]) and store the value at that memory location into i[1](r[25]).
Since a register value is present as providing the source address it can be seen that the immediate flag is set to 0.

```
ld [address], rd
```

![Figure 5: SPARC LD Instruction Verification.](image)

The assembler appears to be correctly providing machine code, however some unresolved symbols remain and memory addresses are incorrect. This will be addressed and corrected within the Linker section. Figure 4 shows an efficient output of the instruction and data portion of the compiled program however this information is not immediately available as it is stored as computer readable machine code. The assembler generates object files as Executable and Linkable Format (ELF) files.

### Executable and Linkable Format

ELF offers a common executable standard that is very flexible and can be applied to different architectures. An ELF file holds the code and data along with information about the system needed to run the contents.

The ELF header begins all ELF files. Described within the ELF header is the type of object file and compiled architecture along with sizes of all other headers. An entire ELF file extraction can be found in Appendix B: Full ELF File Extraction Content Example.

ELF files can be viewed in two respects, a linking or execution view. Figure 6 illustrates the linking view as it specifies sections and not segments. The Program Header is optional from the linking perspective. Section information can define symbol tables, executable code and dynamic linking data. The Section Header is used to describe and point to section data while the Program Header similarly describes segments. All of the instruction code, initial data and other pertinent runtime code can be found in the section and segment data. For the purposes of our analysis the Program Header table will be ignored therefore only sections will be discussed.

A variety of sections can appear in an ELF. For the application at hand we consider only the .text, .data and .bss sections. Within the .text section lies the executable code of the compiled program. Initialized data is typically stored in .data but .data1 can be used as well. Uninitialized data is placed within .bss. Other sections do exist but have not been implemented in the application. Of interest is the unused .strtab section which is used to associate names to symbol table entries. When the data from the ELF file is to be analyzed all symbols (post linking) have been already resolved obfuscating this section. Dynamic linking is not being considered at this point. Architecture and assembler specific section names are also common. Other special sections can be easily added to the application by increasing the breadth of the section extraction. A summary of common section names is shown in Table 3.
TABLE 3: COMMONLY USED ELF SECTIONS [3].

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.bss</td>
<td>Uninitialized data</td>
</tr>
<tr>
<td>.comment</td>
<td>Version control information</td>
</tr>
<tr>
<td>.data</td>
<td>Initialized data</td>
</tr>
<tr>
<td>.debug</td>
<td>Symbolic debug information</td>
</tr>
<tr>
<td>.dynamic</td>
<td>Information about dynamic linking</td>
</tr>
<tr>
<td>.dynstr</td>
<td>Required strings for dynamic linking</td>
</tr>
<tr>
<td>.dynsym</td>
<td>Dynamic linking symbol table</td>
</tr>
<tr>
<td>.fini</td>
<td>Process termination code</td>
</tr>
<tr>
<td>.got</td>
<td>Global offset table</td>
</tr>
<tr>
<td>.hash</td>
<td>Hash of symbol table</td>
</tr>
<tr>
<td>.init</td>
<td>Process initialization code</td>
</tr>
<tr>
<td>.interp</td>
<td>Path name of program interpreter</td>
</tr>
<tr>
<td>.line</td>
<td>Line numbers</td>
</tr>
<tr>
<td>.plt</td>
<td>Procedure linkage table</td>
</tr>
<tr>
<td>.relnam</td>
<td>Relocation information</td>
</tr>
<tr>
<td>.rodata</td>
<td>Read-only data</td>
</tr>
<tr>
<td>.shstrtab</td>
<td>Section names</td>
</tr>
<tr>
<td>.strtab</td>
<td>Names associated with symbol table entries</td>
</tr>
<tr>
<td>.symtab</td>
<td>Symbol table</td>
</tr>
<tr>
<td>.text</td>
<td>Executable instruction code</td>
</tr>
</tbody>
</table>

Despite all the benefits and common use of ELF files, Altera’s memory initialization solution (*elf2mif*) is unsupported and ineffective. Xilinx however does offer direct deployment to memory units. Altera memory requires that all pertinent code and data be extracted from the ELF and converted into a memory initialization file. At this time `.text`, `.data` and `.bss` data is extracted. Figure 7 shows the section headers for the code from Figure 4 post linking. Memory addresses have been reassigned. Note the flags denoting executable (or loadable) and allocatable. Allocatable demands memory be set aside.

FIGURE 7: EXAMPLE OF EXTRACTED SECTION HEADERS.
LINKER

Typically programs are not presented as a single block of assembly. With one static assembly file it becomes wasteful to retranslate, compile and assemble. Often called library routines do not change therefore do not require constant recompiling. Individual procedures should be compiled and assembled separately. A linker allows independently assembled object files to come together to form a single homogeneous executable program. Symbols have been previously alluded to and play a key role when compiling pieces of a large program. Instruction and data code are placed symbolically in memory. All undefined symbols are resolved from information within the symbol table and maintained relocation data. The symbol table contains labels and their corresponding addresses. These symbol references typically occur within branch, jump and data addresses [5]. The linker essentially fixes addresses defined within the code. It was noted that in Figure 4 a resolution was missing for the or instruction. The transition from assembly to final resolution from the linker is shown in Figure 8. The symbol b is used in the assembly language. The GNU assembler does not resolve it and leaves a temporary fainéant instruction.

After the linker pass, the b label reference is resolved to a runtime address, 0x38. For this particular example the label was in fact located in the same object file.

This is often not the case as references can be made to other object files or libraries.

Both the assembler and linker produce object files. The difference lies in the fact that the linker produces executable object code since all references have been resolved.

The final task of the linker is to rearrange object file contents to the defined program address space. The relocating of code is often done, moving one base address to another, since many assemblers assume a base address of 0x0000 for all sections if not explicitly set with assembler directives such as ORG. Linker scripts define how to link files and libraries along with the placement of different sections (.text, .data) based on the available address space.
GNU LINKER

Another important tool provided by Binutils is the linker named GNU ld. The name actually stands for loader which is often used as a synonym for linker in the UNIX world. Due to the interchange of names it is often difficult to separate the compile and run-time understanding. Typically the linker deals with compile-time concepts while the loader with run-time. For the purposes of this project we consider the loader as the actual mechanism for tailoring our program to the specific memory, Altera or Xilinx.

As previously discussed, linkers usually accept linker command language files, more commonly known as linker scripts. These scripts describe how to map sections (.text, .data) to an output file. The GNU ld uses scripts written in AT&T’s Link Editor Command Language. GNU ld alone is called from the command prompt with an extensive array of options and flexibility. Despite the intense variety of features, very few are typically used [6]. Object files and libraries are specified directly at the command line. Custom linker scripts can be called as well, otherwise default scripts that have been compiled into the linker are used. Most of the syntax available within the AT&T Link Editor Command Language can actually be input as arguments.

A basic linker script example is shown in Code 1. The SECTIONS command is used to specify the output address of the defined sections. The ‘.’ is used as a location counter. The .text : { *(.text) } line indicates that the output .text portion should consist of all the input .text sections defined using a wildcard, ‘*’. The next line resets the location counter to 0x1000. The .data section is then placed here. Once the location counter has been changed to 0x2000 the .bss section is then placed followed automatically by .rodata since the location counter automatically updates to 0x1000+size(.bss).

**CODE 1: LINKER SCRIPT EXAMPLE.**

```
SECTIONS
{
   . = 0x0000;
   .text : { *(.text) }
   . = 0x1000;
   .data : { *(.data) }
   . = 0x2000;
   .bss : { *(.bss) }
   .rodata : { *(.rodata) }
}
```

Using the above script with a memory that spans address 0x0000 to 0xFFFF (64KB) is a system where the .bss section requires 0xA0 addresses and .rodata 0x40 Figure 10 illustrates the positioning in memory. Both .text and .data may not in fact use all the memory allocated to them through the location counter assertions. It is noted that we are discussing memory address ranges instead of traditional sizes. 32-bit or 4 byte instructions and data are being considered. Byte addressing means that each line in memory contains 4 possible locations : lineAddress = [lineAddress] by 4 bytes.
Besides the use of SECTIONS many other common commands can be used. Program entry points can be specified using available symbols, formatting options can be set along with Endianness. Of particular importance are the PROVIDE statements. These statements allow a symbol to be replaced with an expression only if the symbol is referenced and not defined by any object file or library specified in the link. It is imperative for setting the beginning of the stack PROVIDE (_stack = 0x4000);. Another feature offered involves the MEMORY command which can define all of the available memory in the system. At this point the application does not automatically generate memory portions of the script. This is automatically done address-wise with information from the Java backed front-end supplied by the user. In Code 2 two memory units are specified. The first region is a ROM that has a size of 512KB and starts at address 0x0000. The second region, a RAM, starts at address 0x80000000 and continues for 4MB. The ROM is specified as read-only and executable while the RAM is not read-only and not executable hence the ‘!’ syntax.

**CODE 2: MEMORY COMMAND USAGE.**

<table>
<thead>
<tr>
<th>MEMORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROM (rx) : ORIGIN = 0, LENGTH = 512K</td>
</tr>
<tr>
<td>RAM (!rx) : org = 0x80000000, l = 4M</td>
</tr>
</tbody>
</table>

Returning to the assembly code presented in Figure 4 and the assembled object file it was noted that both the addresses for the data and instructions overlapped and symbols remained unresolved. Considering an available memory unit with a size of 1KB³ and addresses beginning at 0x0000 with sections being placed one after another and the entry point at 0x0000, a final executable can be created. The disassembly and new address mappings are shown in Figure 11.

³ 1 KB=1024 bytes
SOFTWARE DESIGN & METHODOLOGY

OVERVIEW

To automate the required GNU Binutils tools, the tools were compiled separately and called as processes from a front-end Java application. Using the application, the user specifies their memory architecture as the number of memory units and start/end addresses of each unit. The user may then select which assembly files to use and how to map the resultant executable addresses whether automatically or manually. Using the information provided by the user linker scripts are generated according to the specified memory architecture. These scripts are fed to the linker along with object files and libraries. The resulting object file or files are ELF files designated for a particular memory unit. Data and instruction code sections are extracted from the ELF and converted to a format deployable to Xilinx or Altera memory. The flow of the application is shown in Figure 12.

Once the system has finished generating and processing all information, the user is presented with a visualization of all memory units and position of sections.

At this moment the software solution can only be run on Windows based hosts as the tools have been compiled with a Windows host in mind. Java is convenient in the sense that it is OS independent. Even though the front-end is not OS dependent the compiled tools are. Other host dependent compilations can be performed and called based on the system environment. This process can be easily automated using predefined executables.

The tools called and provided in the application hierarchy are GNU ld, GNU gas, GNU readelf and GNU size. They all target the sparc-elf architecture. For information on adding custom instructions to the available instruction set see the Error! Reference source not found. section. Tools are called directly from the Java application but data is returned through a Data Manipulation Environment where files can be analyzed and created. It serves as a temporary sandbox for the application.

FIGURE 12: APPLICATION FLOW.

FIGURE 13: APPLICATION INTERACTIONS.
**TOP-LEVEL DESIGN**

In a traditional software design approach, code has been partitioned into packages keeping common tasks together. A package with the stereotype <<nonJava>> has been introduced to summarize external Binutil tool usage. Binutils’ executables are found here. Both illustrator and streamManager are helper packages that aid the application in processing and presenting information. The illustrator package contains all the Java code for generating visual representations of memory spaces post-linking. The streamManager is required to capture and store data generated by Binutil tools in the Windows environment as stdio redirection involving Java and executables is not supported. Both helper packages are called directly by the gen package where user interactions are required and tools are called directly. The actual application logic resides here. A package overview is available in Figure 15.

![Figure 15: Application Package Diagram.](image)

**GEN PACKAGE**

As previously stated, application logic is found within this package. Defined classes support the automated calling of an assembler, linker and loader. To support the calling of tools, user entered information is processed and the tool chain is customized to support the user provided system specifications. Whether it be a custom memory architecture or multiple object files and libraries the system adapts and adjusts its parameters accordingly. Such is in the case of custom linker script generation based on entered memory specifications. All customization and temporary files are stored in the Data Manipulation Environment so they can be easily discovered by the application.

An option presented to the user early on deals with whether the system should dynamically (automatically) or manually place sections into memory. This causes specific classes to be called and modified accordingly in response.

Figure 16 shows an overview of the classes and relationships covering the internal operations of the gen package. For further details on the classes outlined in red see [7] where they have been taken from. The generation of an executable uses the GenerateMain class as a metaphoric representation. The actions involved in creating an executable, ergo the parts of the generation are the assembling...
(Assembler class), the linking (Linker class) and the Altera/Xilinx memory initialization modules (MemInit abstract class).

The Linker class creates linker scripts through a LinkerScript class. If the user does not want the system to automatically allocate memory a CustomLinker, a generalization of Linker is used. Other classes are present but are not imperative for understanding the conceptual model of the system. See Appendix C: Source Code for all code and classes.

The GenerateMain class accepts the system memory architecture from the user along with a series of assembly files. The user is asked for the number of memories and the start and ending addresses for each.

**CODE 3: GET SYSTEM MEMORY INFORMATION.**

```java
for(int i=0; i<numberOfMem; i++)
{
    System.out.println("Memory Unit "+(i+1));
    System.out.print("\tStarting Address: 0x");
    startAddress[i] = getInput(br);
    System.out.print("\tEnding Address: 0x");
    endAddress[i] = getInput(br);
}
```

Next the user is able to list the number of assembly files and their relative location. The assembler executable can now be called. The application loops through all the assembly files and calls the Assembler class sending the file names as parameters.

Once the application has returned from assembling, primary ELF files can be found in the Data Manipulation Environment. The Linker class is then called translating the memory architecture of the system. Here the final linking takes place and all pertinent sections are extracted from the final executables and stored as plaintext for the next phase. Arrays are the most used data structure in the application as they can be initialized to suit any variable quantities the user requests.
The final step in processing a user request involves the parsing of plaintext files generated by the Linker, which house the data and instruction machine code. The target filename and list of generated executables are sent to the AsmToMif and AsmToMem classes as described in [7].

Once the application has run its course, a visual application window is opened showing the memory map of each memory unit. It shows how the sections of each executable have been stored according to addresses.

The Assembler class is basic as it does a direct call by executing a Windows process. The pre-compiled GNU gas is called with appropriate parameters. The resultant object file is temporarily placed in a known location. The call is shown in Code 4 where fileName is the name of the assembly file to be assembled and shortName is the output object file. Note that the assembler has been pre-compiled to target sparc-elf however the sub-architecture is being specified as version 8.

CODE 4: CALLING THE PRE-COMPILED ASSEMBLER.

```
Process p = r.exec("Binaries\jp-sparc-elf-as.exe -xarch=v8"+fileName +" -o "+"temp\"+shortName);
```

The Linker class must determine how the user wants their machine code placed; once again the system can make the decision alone. Dependent on the wish either a general or custom linker script will be created through the LinkerScript class. Once the scripts have been generated GNU ld is called and will use the newly created scripts.

CODE 5: CALLING THE PRE-COMPILED LINKER.

```
Process p = s.exec("Binaries\jp-sparc-elf-ld.exe -o "+"elf\final_"+shortName[1] +" temp\"+shortName[1] +" -T "+"linker-scripts\"+scriptNames[1]);
```

Once the object files have been linked and resolved, the executables need to be turned into a user-readable format. GNU readelf is used to extract the fully resolved and assigned instructions in hexadecimal. Section additions can be made in future application versions. A separate method is used which interacts with the streamManager package to capture the results of the extraction and store them into plaintext. Figure 17 shows the equivalent syntax to the lengthy Java process call that is repeated for each section done directly via a command prompt call. Addresses precede lines of 4 instructions or data.

![FIGURE 17: READELF SECTION MACHINE CODE EXTRACTION.](image-url)
The LinkerScript class creates a physical linker script following AT&T’s syntax. If the user wishes to allow the system to dynamically assign memory, .text is placed at the starting memory address with .data and .bss right after in an available memory. The script is written and stored as a file with a unique name to be called by the linker.

**CODE 6: WRITING THE DEFAULT LINKER SCRIPT.**

```java
scriptFile.println("SECTIONS\n{\n. = 0x"+Long.toHexString(d)+";\n.text : { *(.text) }\n.data : { *(.data) }\n.bss : { *(.bss) }\n}\n);"
scriptFile.println("PROVIDE (_stack = \n0x"+Long.toHexString(stackAddress)+")\n";"
```

If the user wishes to assign their own addresses they are presented with an outline of the executable with all the size constraints through GNU size. Knowing the required minimum they user may assign their own addresses which may separate instructions from data or the stack within the same memory or different memories.

**CODE 7: WRITING A SECTION CUSTOMIZABLE LINKER SCRIPT.**

```java
scriptFile.println("SECTIONS\n{\n. = 0x"+Long.toHexString(textAddress)+";\n.text : { *(.text) }\n.data : { *(.data) }\n.bss : { *(.bss) }\n}\n);"
scriptFile.println("PROVIDE (_stack = \n0x"+Long.toHexString(stackAddress)+")\n";"
```

So far a brief overview of calls and program structure have been explored. It is important to consider the creation and modification of files from this package. Figure 18 shows the file structure and the class interactions highlighting the read and write actions.

**FIGURE 18: INTERACTION BETWEEN CLASSES, FOLDERS AND FILES.**
STREAM MANAGER PACKAGE

In a Windows environment executables called from Java cannot have their outputs easily captured through redirection. (It is possible if system variables are used to make calls, which is not the case.) This package maintains the streams for capturing GNU Binutil tool output and writing it to plaintext for analysis. It uses a PrintWriter, InputStreamReader and BufferedReader. This class, streamOutput is called whenever a capture is required. Another helper class is situated within this package. It is used to delete temporary files after they have been read by the needed process.

ILLUSTRATOR PACKAGE

The illustrator package is used after the memory files have been generated. Classes within this package create a visual pane showing the position of code sections within the address space separated by individual memory units. All of the addresses are normalized to a standard visualizable range. The java AWT library is used to physically draw the map. A generated memory map is shown in Figure 19.

![FIGURE 19: GENERATED MEMORY MAP OUTPUT.](image-url)
CUSTOM INSTRUCTION EXTENSION

So far the use of custom instructions by augmenting an available instruction set has been mentioned but not discussed. The matter is actually a simple affair pending that the target hardware actually supports the instructions/opcodes. At this point instruction sets can only be augmented prior to GNU Binutils compilation and require manual interaction. Future iterations of this application will allow for customization at run-time.

At compile-time Binutils looks in an opcode directory for a listing of opcodes and instruction formats based on the specified target architecture. To augment the SPARC instruction set, sparc-opc.c must be modified. An example of the AND instruction is shown in Code 8.

CODE 8: AND INSTRUCTION FORMAT AND OPCODE.

```c
( "and",   F3(2, 0x01, 0), F3(~2, ~0x01, ~0)|ASI(~0),  "1,2,d", 0, v6 ),
( "and",   F3(2, 0x01, 1), F3(~2, ~0x01, ~1),  "1,i,d", 0, v6 ),
( "and",   F3(2, 0x01, 1), F3(~2, ~0x01, ~1),  "i,1,d", 0, v6 ),
```

The mnemonic is clearly presented as a string in each line. The F3(x,y,z) and its negation define a format 3 instruction. All functions and variables used are defined in sparc.h. The first two bits of the instruction are set to “10” while the opcode is “00 0001” as confirmed in the SPARC Architecture Manual. The next integer within F3 defines whether the instruction is to use an immediate or a register value (0=register 1=immediate).

Looking at the line using solely register values a “1,2,d” string is defined. 1 corresponds to source register 1, rs1, while the 2 equates to source register 2, rs2. The d represents the destination register rd. The v6 is the initial SPARC version where this instruction is available. These 3 lines convert assembly and rs1,rs2,rd or and rs1,imm,rd to outputs defined in Figure 20.

<table>
<thead>
<tr>
<th>10</th>
<th>rd</th>
<th>000001</th>
<th>rs1</th>
<th>0</th>
<th>00000000</th>
<th>rs2</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>rd</td>
<td>000001</td>
<td>rs1</td>
<td>1</td>
<td>simm13</td>
<td></td>
</tr>
</tbody>
</table>

FIGURE 20: AND INSTRUCTION MACHINE CODE OUTPUT.

To add custom format 3 instructions unused opcodes must be used. A variety of unused codes exist within the specified instruction set. Examples of unused opcodes are available in Table 4.

TABLE 4: EXAMPLE OF UNUSED OPCODES.

<table>
<thead>
<tr>
<th>Bits 31 and 30</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>001000</td>
</tr>
<tr>
<td>10</td>
<td>001001</td>
</tr>
<tr>
<td>10</td>
<td>001101</td>
</tr>
<tr>
<td>10</td>
<td>011001</td>
</tr>
<tr>
<td>11</td>
<td>011100</td>
</tr>
<tr>
<td>11</td>
<td>011011</td>
</tr>
</tbody>
</table>
The `sparc-opc.c` file creates an array structure with all the instruction possibilities. Consider a situation where a processor, say a SPARC based LEON-3 has a custom multiplier unit called the *Super Multiplier* that is triggered by a specific opcode. This multiplier quickly performs a signed multiplication of two registers. Suppose the new bit31+bit30+opcode used to call this hardware extension is 10001001. The new instruction as machine code would be defined as Figure 21.

```
supermul rs1,rs2,rd
```

![FIGURE 21: CUSTOM ASSEMBLY MNEMONIC WITH MATCHING INSTRUCTION FORMAT.](image)

Knowing that this instruction follows a format 3 structure an additional line can be added to the `sparc-opc.c` file. The custom multiplier instruction is shown in Code 9. The new mnemonic has been added along with opcode and required registers. The architecture version has been set to 8 despite the fact that this is not a SPARC supported instruction. The version does not matter as long as the new instruction specification is included at compile time.

**CODE 9: CUSTOM MULTIPLIER INSTRUCTION SET EXTENSION.**

```c
{ "supermul",F3(2, 0x09, 0), F3(~2, ~0x09, ~0),"1,2,d", 0, v8 },
```

Once the instruction has been added GNU Binutils must be recompiled. After compilation the new assembly instruction will be understood by all tools. The custom instruction has been used in the assembly program shown in Code 10 with the resultant disassembly in Figure 22.

**CODE 10: ASSEMBLY WITH CUSTOM INSTRUCTION.**

```c
.section ".text"
.align 4
.global main
main:
supermul %g0,%g1,%g2
```

![FIGURE 22: DISSASSEMBLY OF EXECUTABLE WITH CUSTOM INSTRUCTION.](image)
RESULTS AS A CASE STUDY

Consider an embedded multiprocessing system as shown in Figure 23. Two processors share a bus with two memories attached. These processors are soft-core SPARC based LEON-3 processors provided by Aeroflex Gaisler. Processor 1 is a standard configuration while the second has the previously discussed Super Multiplier hardware implemented. GNU Binutils has already been recompiled with this new custom multiplier instruction.

A user has two C files that they wish to run on this system, one program for each processor. The custom instruction is called with inline assembly. Each processor will use its own memory, but is aware of the other unit; PROC 1 will use Memory 1 and vice-versa.

```
CODE 11: PROCESSOR 1 C CODE.
int a = 0;
int b = 1;
int c = 2;
int t1;

int main(int argc, char *argv[])
{
    t1 = b+c;
}

CODE 12: PROCESSOR 2 C CODE.
int d = 3;
int e = 4;
int t1;

int main(int argc, char *argv[])
{
    t1 = d+e;
    __asm__ __volatile__ ("supermul %0,%1,%2");
}
```

Running these two programs (Code 11 and Code 12) through a compiler will yield two assembly files. These files become inputs to the application along with the memory addressing of the system.

Since the memories form a shared address space, start and end addresses must be assigned based on size. The first memory covers addresses 0 to 255 while the second 256 to 512. Each processor will know the entry point of their program which will be placed at the first address in the corresponding memory.

![Diagram of example system layout](image)

**FIGURE 23: EXAMPLE SYSTEM LAYOUT.**

![Diagram of address mapping with 2 memories](image)

**FIGURE 24: ADDRESS MAPPING WITH 2 MEMORIES.**
Both Code 13 and Code 14 show the assembly code derived from the C program after compilation. Each of these will be assembled, linked and placed in an available memory. The user is required to know the name and location of these files. The inline assembly code as been imposed after the addition.

CODE 13: PROCESSOR 1 ASSEMBLY CODE.

```
.section "text"
.align 4
.global main

main:
    save %sp,-96,%sp
    sethi %hi(b),%i0
    or %i0,%lo(b),%i0
    ld [%i0],%i1
    sethi %hi(c),%i0
    or %i0,%lo(c),%i0
    ld [%i0],%i0
    add %i1,%i0,%i0
    sethi %hi(t1),%i1
    or %i1,%lo(t1),%il
    st %i0,[%i1]

.L3:
    ret
    restore

.section "data"
.align 4
.global a
a:
    .word 0
    .align 4
    .global b
b:
    .word 1
    .align 4
    .global c
c:
    .word 2
    .common t1,4,4
```

CODE 14: PROCESSOR 2 ASSEMBLY CODE.

```
.section "text"
.align 4
.global main

main:
    save %sp,-96,%sp
    sethi %hi(d),%i0
    or %i0,%lo(d),%i0
    ld [%i0],%i1
    sethi %hi(e),%i0
    or %i0,%lo(e),%i0
    ld [%i0],%i0
    add %i1,%i1,%i1
    supermul %i0,%i1,%g1
    sethi %hi(t1),%i1
    or %i1,%lo(t1),%il
    st %g1,[%i1]

.L3:
    ret
    restore

.section "data"
.align 4
.global d
d:
    .word 3
    .align 4
    .global e
e:
    .word 4
    .common t1,4,4
```

When the application begins the user is asked about how many memory units are available in the system and the addressing scheme to use. The scheme for this example was shown in Figure 24. Addressing is not done automatically and must be provided by the user or calling system. The initial command line view is shown in Figure 25. Memory unit 1 is entered as address 0x00 to 0xFF along with memory unit 2 encompassing addresses 0x100 to 0x1FF.
Once the user has specified the number of memory units and the addressing scheme they must enter the relative path names of the assembly files after specifying the number of input files. Two assembly files are available so the paths to `proc1.s` and `proc2.s` are entered.

After each assembly file name is accepted the assembler is called in the backend to create individual object files in the ELF format. These files are stored locally in a temp directory.
Once all the assembly files have been successfully converted into object files the user must decide whether the system should automatically place the sections contained in the ELF file or manually give addresses based on entered memories.

For this example dynamic allocation is preferred. The user must decide which memory unit to use for each future executable. \texttt{proc1.out} will be assigned to memory unit 1 and \texttt{proc2.out} will go to memory unit 2. The automatic allocation tells the system to place the instructions at the base of addresses of the corresponding memory followed directly by the data.
Any allocation, whether custom or automated as in this case requires that linker scripts be generated by the system. These scripts correspond to the previous memory selections. The generated scripts are shown in Code 15 and Code 16. They map the .text sections to the beginning of the memory unit. The .data and .bss are automatically placed after the .text.

**CODE 15: GENERATED PROC 1 LINKER SCRIPT.**

```c
/* Automatic linker script
generation of proc1.out */

SECTIONS
{
   . = 0x0;
   .text : { *(.text) }
   .data : { *(.data) }
   .bss : { *(.bss) }
}
PROVIDE (_stack = 0xff);
```

**CODE 16: GENERATED PROC 2 LINKER SCRIPT.**

```c
/* Automatic linker script
generation of proc2.out */

SECTIONS
{
   . = 0x100;
   .text : { *(.text) }
   .data : { *(.data) }
   .bss : { *(.bss) }
}
PROVIDE (_stack = 0x1ff);
```

Using the generated scripts the application automatically calls the linker which resolves any remaining symbols and applies the memory scheme to the instructions and data. The final executable object files are created and all the pertinent information is extracted and turned into plaintext which is stored for final conversion. The two sectional extractions with addresses and contents are shown in Code 17 and Code 18.

**CODE 17: MACHINE CODE EXTRACTION FOR PROC 1.**

```
Hex dump of section '.text':
  0x00000000 9de3bfa0 31000000 b0162038 f2060000 ....1..... 8....
  0x00000010 31000000 b016203c f0060000 b0064018 1..... <......@.
  0x00000020 33000000 b2166040 f0264000 81c7e008 3.....`@.&@.....
  0x00000030 81e80000                            ....

Hex dump of section '.data':
  0x00000034 00000000 00000001 00000002          ............

Section '.bss' has no data to dump.
```

**CODE 18: MACHINE CODE EXTRACTION FOR PROC 2.**

```
Hex dump of section '.text':
  0x00000100 9de3bfa0 31000000 b0162138 f2060000 ....1.....!8....
  0x00000110 31000000 b016213c f0060000 b0064018 1.....!<......@.
  0x00000120 824e0019 33000000 b2166140 f0264000 81c7e008 3.....`@.&@.
  0x00000130 81e80000                            ....

Hex dump of section '.data':
  0x00000138 00000003 00000004                   ...........

Section '.bss' has no data to dump.
```
This plaintext data can be parsed and turned into Altera Memory initialization files to be deployed directly to a memory instantiated on an FPGA. The conversion is done through the classes provided by [7]. The two generated memory initialization files are illustrated in Code 19 and Code 20 respectively.

CODE 19: MEMORY UNIT 1 MIF FILE FOR PROC 1.

WIDTH=32;
DEPTH=255;

ADDRESS_RADIX=HEX;
DATA_RADIX=HEX;

CONTENT BEGIN
  0 : 9de3bfa0;
  4 : 31000000;
  8 : b0162038;
 c : f2060000;
10 : 31000000;
14 : b016203c;
18 : f0060000;
1c : b0064018;
20 : 81c7e008;
24 : b2166040;
28 : f0264000;
2c : 81c7e008;
30 : 81e80000;
34 : 00000000;
38 : 00000001;
3c : 00000002;
  [040..fe] : 00000000;
END;

CODE 20: MEMORY UNIT 2 MIF FILE FOR PROC 2.

WIDTH=32;
DEPTH=255;

ADDRESS_RADIX=HEX;
DATA_RADIX=HEX;

CONTENT BEGIN
  100 : 9de3bfa0;
  104 : 31000000;
  108 : b0162138;
  10c : f2060000;
  110 : 31000000;
  114 : b016213c;
  118 : f0060000;
  11c : b0064018;
  120 : 824e0019;
  124 : 33000000;
  128 : b2166140;
  12c : c2264000;
  130 : 81c7e008;
  134 : 81e80000;
  138 : 00000003;
  13c : 00000004;
  [140..fe] : 00000000;
END;

Once the memory initialization files have been generated the system shows the user a memory map denoting section placement with respect to the container executable object file.

FIGURE 29: MEMORY UNIT 1 ADDRESS SPACE USAGE.

FIGURE 30: MEMORY UNIT 2 ADDRESS SPACE USAGE.
FUTURE WORK
As GNU slowly changes its code base to C++ as seen with GOLD the new linker, it is hoped that eventually Java code ports become available. As previously discussed, at the present time the Binutils’ tools must be precompiled. As this is the case the program itself does not support multiple operating systems. To work around this a set of binaries can be precompiled and selected based on a Java query of the system environment. Suggested builds include Windows, Debian Linux and Solaris.

The addition of custom instructions has been touched on but the issue of constant recompile leaves much to be desired as it falls outside of the tool chain. Further analysis of Binutils’ source may yield a simple solution to accepting new instructions post-compile via arguments or even doing the entire architecture specification separately. The problem still remains not with only the assembler requiring the architecture information, but all the tools as well. It may prove beneficial to rewrite a simpler but more focused assembler and linker based on GNU looking solely at the SPARC architecture and augmentation possibilities.

Despite these shortcomings modifications will progress overtime as this small tool is integrated into much larger projects allowing for user control over how source code is mapped to a system. The review of ArchC has also opened the possibility of linking the simulator into this tool chain for verification of results.

CONCLUSION
Through Binutils’ customization, a successful linker and assembler emerged. These two integral tools were automated by a front-end Java application allowing more user control over how assembly code will be handled by the system. Based on a user’s needs, their target system, custom scripts can be generated to map to any system’s memory layout. Binutils is not bound to GCC and there now exists an effective application to harness its power without bombarding oneself at the terminal writing one configuration file after another.

The possibility of custom instruction addition opens many venues with respect to streamlining existing tools from the compile to the run-time realm.

The application has reached its initial stages with the possibility of advances defined above. As for now it has met its goals of successfully assembling and linking SPARC assembly source and bringing them to a hardware-aware and ready stage. Results shown within this document have been shown to be correct and concise. This application should excel with the more demanding programs it will encounter as part of the SHIRA project.
REFERENCES


*All diagrams are original.
APPENDIX A: COMPILING GNU BINUTILS 2.19 FROM WINDOWS

Minimalist GNU for Windows (MinGW) is used for compilation as Cygwin compiled executables require extra libraries when running. MinGW guarantees standalone interoperability. Both of these programs implement a POSIX to Win32 system call interface while making a variety of GNU tools available in Windows. Compilation has been tested on Windows XP and Windows Vista 32/64-bit.

1) Download MinGW and the MinGW Bourne shell entitled (Minimal SYStem) MSSYS. Windows executables are available from SOURCEFORGE at http://sourceforge.net/project/showfiles.php?group_id=2435.

2) Since v2.16 binutils has made extra calls to other packages that will now be required for a full compilation. Both the POSIX regular expression library and makeinfo are now required. makeinfo is noted as optional but compilation will fail if not included in the path. Only makeinfo requires compilation.
   a. A port of the regex library has been made for MinGW. Download both mingw-libgnurx-2.5.1-bin.tar.gz and mingw-libgnurx-2.5.1-dev.tar.gz from http://sourceforge.net/project/showfiles.php?group_id=2435&package_id=73286&release_id=140957.
   b. Place libgnurx-0.dll in the MinGW bin directory, <Installation Path>/MinGW/bin/.
   c. Place /include/regex.h into MinGW include directory, <Installation Path>/MinGW/include/.
   d. Move /libgnurx.dll.a and libregex.a to the MinGW lib folder, <Installation Path>/MinGW/lib/.

3) makeinfo is not supported by MinGW by default so a work around will be implemented. makeinfo is part of the GNU texinfo package.
   b. Untar or unzip the source code in a local directory.
   c. It is good practice to not compile in the same directory as the base source good. Create a new directory off of the source directory using MSYS, mkdir build
   d. Switch to build directory, cd build
   e. Run the configure script, ./texinfo-4.13/configure
   f. The makefile must be manually linked to the previously inserted regex libraries. Open install-info\Makefile from the build folder with any text editor.
   g. Find the line LIBS = and equate it to -lregex, LIBS=-lregex
   h. Return to the build directory if you have left and run make, make
   i. The compilation of texinfo WILL FAIL due to a variety of missing components but not until after makeinfo has been compiled.
   j. Move to the build/makeinfo folder, cd makeinfo
   k. Complete the compile and installation of makeinfo by typing make then make install in the makeinfo directory, make followed by make install

4) Binutils v2.19 can now be compiled without error.
   b. Untar or unzip the source code in a local directory.
   c. Create a new build directory buildBIN for compilation, mkdir buildBIN
   d. Change to build directory, cd buildBIN
   e. Run configure script targeting the SPARC architecture, ../binutils-2.19/configure - target=sparc-elf
   f. Build the source while in the buildBin directory, make
g. Versions 2.17 and above require a `make install` to finalize the executables for the Windows environment. Without running `make install` the executables will still run in the Bourne shell but cannot be called from command prompt. `make install`  

h. After the previous make all the executables have been added to the `local/bin` directory. All compiled executables have been annotated with `sparc-elf`.

\(<Installation\ Path>\msys\1.0\local\bin\)

![Compiled Binutils Executables](image)

**FIGURE 31: COMPILED BINUTILS EXECUTABLES.**
**APPENDIX B: FULL ELF FILE EXTRACTION CONTENT EXAMPLE**

Below shows an extraction of all the ELF information from the final\_simplest0.out ELF file post linking. GNU readelf was used to create the dump.

**ELF Header:**
```
Magic:    7f 45 4c 46 01 02 01 00 00 00 00 00 00 00 00 00
Class:    ELF32
Data:     2's complement, big endian
Version:  1 (current)
OS/ABI:   UNIX - System V
ABI Version:  0
Type:     EXEC (Executable file)
Machine:  Sparc
Version:  0x1
Entry point address: 0x0
Start of program headers:  52 (bytes into file)
Start of section headers:  65708 (bytes into file)
Flags:    0x0
Size of this header:  52 (bytes)
Size of program headers:  32 (bytes)
Number of program headers:  1
Size of section headers:  40 (bytes)
Number of section headers:  8
Section header string table index: 5
```

**Section Headers:**
```
[Nr] Name          Type         Addr        Off      Size  ES Flg Lk Inf Al
[ 0]                NULL         00000000  000000  000000  00      0   0  0
[ 1] .text          PROGBITS     00000000  010000  000034  00  AX  0   0  4
[ 2] .data          PROGBITS     00000000  010034  00000c  00  WA  0   0  4
[ 3] .bss           NOBITS       00000034  010040  000004  00  WA  0   0  4
[ 4] .comment       PROGBITS     00000000  010040  000035  00      0   0  1
[ 5] .shstrtab      STRTAB       00000000  010075  000035  00      0   0  1
[ 6] .symtab        SYMTAB       00000000  0101ec  0000a0  10      7   5  4
[ 7] .strtab        STRTAB       00000000  01028c  00000f  00      0   0  1
```

Key to Flags:
- W (write), A (alloc), X (execute), M (merge), S (strings)
- I (info), L (link order), G (group), x (unknown)
- O (extra OS processing required) o (OS specific), p (processor specific)

There are no section groups in this file.

**Program Headers:**
```
Type     Offset      VirtAddr      PhysAddr      FileSize   MemSiz  Flg Align
LOAD     0x010000   0x00000000  0x00000000  0x00040   0x00044  RWE 0x10000
```
Section to Segment mapping:
Segment Sections...
  00  .text .data .bss

There is no dynamic section in this file.

There are no relocations in this file.

There are no unwind sections in this file.

Symbol table '.symtab' contains 10 entries:

<table>
<thead>
<tr>
<th>Num</th>
<th>Value</th>
<th>Size</th>
<th>Type</th>
<th>Bind</th>
<th>Vis</th>
<th>Ndx</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000000</td>
<td>0</td>
<td>NOTYPE</td>
<td>LOCAL</td>
<td>DEFAULT</td>
<td>UND</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>00000000</td>
<td>0</td>
<td>SECTION</td>
<td>LOCAL</td>
<td>DEFAULT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>00000034</td>
<td>0</td>
<td>SECTION</td>
<td>LOCAL</td>
<td>DEFAULT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>00000040</td>
<td>0</td>
<td>SECTION</td>
<td>LOCAL</td>
<td>DEFAULT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>00000000</td>
<td>0</td>
<td>SECTION</td>
<td>LOCAL</td>
<td>DEFAULT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>00000038</td>
<td>0</td>
<td>NOTYPE</td>
<td>GLOBAL</td>
<td>DEFAULT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>000003c</td>
<td>0</td>
<td>NOTYPE</td>
<td>GLOBAL</td>
<td>DEFAULT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0000040</td>
<td>4</td>
<td>OBJECT</td>
<td>GLOBAL</td>
<td>DEFAULT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>00000000</td>
<td>0</td>
<td>NOTYPE</td>
<td>GLOBAL</td>
<td>DEFAULT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>00000034</td>
<td>0</td>
<td>NOTYPE</td>
<td>GLOBAL</td>
<td>DEFAULT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
APPENDIX C: SOURCE CODE

/**
 * @author Jonathan Parri Created: December 2008 Editors: Michael Montcalm
 * GenerateMain.java is the main class for calling the assembler, generating linker scripts, linking elf files and creating memory initialization files for Xilinx and Altera. Multiple memories can be used with a known address mapping scheme.
 */

package gen;

import illustrator.MemoryMapCreator;
import java.io.BufferedReader;
import java.io.FilenameFilter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.File;

public class GenerateMain {

    public static void main(String[] args) {

        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        long[] startAddress;
        long[] endAddress;
        String[] fileNames, finalFileList, elfFileList;

        System.out.println("gas and ld SPARC V8 toolchain 1.0 \n" + "This program is has absolutely no warranty \n" + "Jonathan Parri 2008 \n");

        System.out.print("Number of memory units: ");
        int numberOfMem = 1;
        numberOfMem = (int) getInput(br);

        // Allocate array memory for addresses
        startAddress = new long[numberOfMem];
        endAddress = new long[numberOfMem];

        // Get memory addresses, that will give memory sizes
        for (int i = 0; i < numberOfMem; i++) {
            System.out.println("Memory Unit " + (i + 1));
            System.out.print("\tStarting Address: 0x");
            startAddress[i] = getInput(br);
            System.out.print("\tEnding Address: 0x");
            endAddress[i] = getInput(br);
        }

        Assembler assemble = new Assembler();
    }
}
// Get assembly files
int numberOfFiles = 1;
System.out.print("Number of .s files to assemble: ");
numberOfFiles = (int) getInput(br);

// Allocate array memory for addresses
fileNames = new String[numberOfFiles];

for (int i = 0; i < numberOfFiles; i++) {
    System.out.print("File " + (i + 1) + ": ");
    try {
        fileNames[i] = br.readLine();
    } catch (IOException e) {
        e.printStackTrace();
    }
    assemble.run(fileNames[i]);
}

Linker linker = new Linker();
linker.run(fileNames, startAddress, endAddress, br);

// Turn machine code into a .mif file for Altera FPGAs
File direc = new File("codeDump");
FilenameFilter filter = new CodeFilter();
finalFileList = direc.list(filter); // Filter out garbage
MifLoader load = new MifLoader();
for (int i = 0; i < finalFileList.length; i++) {
    System.out.println("Processing: "+ finalFileList[i]);
    load.readMachineCode("codeDump\" + finalFileList[i],
            "memInit\mem" + (i + 1) + ".mif", (int)
            endAddress[i] - (int) startAddress[i]);
}

// Draw the memory maps...Requires further automation
File direc2 = new File("elf");
elfFileList = direc2.list(filter);
System.out.println("First File : " + elfFileList[0]);
MemoryMapCreator mapShow = new MemoryMapCreator();
mapShow = new MemoryMapCreator();
for (int i = 0; i < elfFileList.length; i++) {
    System.out.println("Publishing: "+ elfFileList[i]);
    mapShow.start((int) startAddress[i], (int) startAddress[i],
            (int) endAddress[i], "elf/" + elfFileList[i]);
}

private static long getInput(BufferedReader br) {
    long temp = 0;
    try {
        temp = Long.parseLong(br.readLine(), 16);
    } catch (IOException e) {
        e.printStackTrace();
    }
    return temp;
}
package gen;

public class Assembler {

    public void run(String fileName) {
        Runtime r = Runtime.getRuntime();

        // Parse file name
        String shortName;
        int index = fileName.indexOf(".");
        shortName = fileName.substring(0, index).concat(".out");

        // Create elf file using custom GNU gas
        try {
            Process p = r.exec("Binaries\jp-sparc-elf-as.exe -xarch=v8 "
                    + fileName + " -o " + "temp\" + shortName);
            System.out.println(System.currentTimeMillis() + " - " +
            shortName
                    + " assembled!");
        } catch (Exception e) {
            System.out.println(System.currentTimeMillis() + "-Assembler, jp-sparc-elf-as.exe not found. Check Binaries directory."");
            System.exit(1);
        }
    }
}
/**
 * @author Jonathan Parri Created: December 2008
 * Linker.java calls the GNU Binutils ld to create final executable object files. Linker scripts are used after calling the creator class and are based on memory architecture. Once executables have been created the machine code of vital sections are extracted and stored as plain text.
 */

package gen;

import java.io.BufferedReader;
import java.io.FileOutputStream;
import java.io.IOException;
import streamManager.Delete;
import streamManager.streamOutput;

public class Linker {

Runtime s = Runtime.getRuntime();
Delete deleteFile = new Delete(); // Remove pre-existing file
LinkerScript linkerScript = new LinkerScript();
FileOutputStream fos;

public void run(String[] fileNames, long[] startAddress, long[] endAddress, BufferedReader br) {
    int iterations;
    int index;

    String[] scriptNames;
    String[] shortName;

    iterations = fileNames.length;
    shortName = new String[iterations];
    scriptNames = new String[iterations];

    for (int i = 0; i < iterations; i++) {
        index = fileNames[i].indexOf(".");
        shortName[i] = fileNames[i].substring(0, index).concat(".out");
    }

    char userResponse;
    int memSelect;
    CustomLinker customLinker = new CustomLinker();

    for (int i = 0; i < iterations; i++) {
        System.out.print("Allocate memory dynamically for " + shortName[0] + " (y/n): ");
    }
userResponse = getYN(br);
if (userResponse == 'y') {
    System.out.println("Memory unit to use for " +
    shortName[i] + " ?");
    System.out.println("Available memory units: ");
    for (int j = 0; j < iterations; j++) {
        System.out.println("[" + (j + 1) + "] : "
            + "Memory Unit 0x"
            + Long.toHexString(startAddress[j])
            + " -> 0x"
            + Long.toHexString(endAddress[j]));
    }
    System.out.print("Memory selection: ");
    memSelect = getInt(br);
    try {
        scriptNames[i] = linkerScript.createScript(
            startAddress[memSelect - 1],
            endAddress[memSelect - 1],
            shortName[memSelect - 1], true, 0,
            0, 0, 0);
    } catch (IOException e) {
        System.out.println(System.currentTimeMillis()
            + "-Could not create script.");
    }
    System.out.println(System.currentTimeMillis()
        + "-Linker script generated.");
} else {
    // CREATE CUSTOM BSS, DATA AND TEXT ADDRESSES!
    System.out.println("Memory units: ");
    for (int j = 0; j < iterations; j++) {
        System.out.println("[" + (j + 1) + "] : "
            + "Memory Unit 0x"
            + Long.toHexString(startAddress[j])
            + " -> 0x"
            + Long.toHexString(endAddress[j]));
    }
    scriptNames[i] = customLinker.customLinker("temp\"
        + shortName[i], br, "custom_script" + i);
}

// Link those scripts
for (int i = 0; i < iterations; i++) {
    try {
        Process p = s.exec("Binaries\jp-sparc-elf-ld.exe -o "
            + "elf\final_" + shortName[i] + "temp\"
            + shortName[i] + " -T " + "linker-
            scripts\"
            + scriptNames[i]);
System.out.println(System.currentTimeMillis() + "- Linked with "+ scriptNames[i] + ".");
}

} catch (Exception e) {
    System.out.println(System.currentTimeMillis() + "-Linker, jp-sparc-elf-ld.exe not found. Check Binaries directory.");
    System.exit(1);
}

// Dump machine code
for (int i = 0; i < iterations; i++) {
    deleteFile.delete("codeDump\memBlock" + (i + 1) + ".code");
    generate("Binaries\jp-sparc-elf-readelf.exe ",
            "codeDump\memBlock" + (i + 1) + ".code", s, "-x 1 ",
            "elf\final_" + shortName[i], true);
    generate("Binaries\jp-sparc-elf-readelf.exe ",
            "codeDump\memBlock" + (i + 1) + ".code", s, "-x 2 ",
            "elf\final_" + shortName[i], true);
    generate("Binaries\jp-sparc-elf-readelf.exe ",
            "codeDump\memBlock" + (i + 1) + ".code", s, "-x 3 ",
            "elf\final_" + shortName[i], true);
}

System.out.flush();
System.out.println(System.currentTimeMillis() + "- Machine code extracted.");

}

void generate(String executable, String fileName0, Runtime r0,
String command, String elfFile, boolean append) {
    try {
        fos = new FileOutputStream(fileName0, append);
        Process q = r0.exec(executable + command + " " + elfFile);
        q.waitFor(); // Force to wait until process is done,
    } catch (Exception e) {
        System.out.println(System.currentTimeMillis() + "-Linker, jp-sparc-elf-ld.exe not found. Check Binaries directory.");
        System.exit(1);
    }
    // Error?
    streamOutput errorGobbler = new streamOutput(q.getErrorStream(),
            "ERROR");

    // Output?
    streamOutput outputGobbler = new streamOutput(q.getInputStream(),
            "OUTPUT", fos);

    // Get that data
errorGobbler.start();
outputGobbler.start();

fos.flush();
}
catch (Throwable t) {
    System.out.println(System.currentTimeMillis() + "-Can't find read-elf utility.");
}

private static char getYN(BufferedReader br) {
    char response = 'y';
    try {
        response = br.readLine().charAt(0);
    } catch (IOException e) {
        e.printStackTrace();
    }

    if (response != 'y' && response != 'Y' && response != 'N' && response != 'n') {
        System.out.println(System.currentTimeMillis() + "-Incorrect value...Try again.");
        response = getYN(br);
    } else {
        response = Character.toLowerCase(response);
    }

    return response;
}

private static int getInt(BufferedReader br) {
    int temp = 1;
    try {
        temp = Integer.parseInt(br.readLine());
    } catch (IOException e) {
        e.printStackTrace();
    }

    return temp;
}

protected static long getLong(BufferedReader br) {
    long temp = 1;
    try {
        temp = Long.parseLong(br.readLine());
    } catch (IOException e) {
        e.printStackTrace();
    }

    return temp;
}
package gen;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class CustomLinker extends Linker {

    public String customLinker(String fileName, BufferedReader br, String name) {

        long textAddress, dataAddress, bssAddress, stackAddress;
        String holdName = null;
        System.out.println("Current section sizes in ELF:");

        // create temp file of sizes
        generate("Binaries\jp-sparc-elf-size.exe -B -x", "temp\temp.tmp", s, " ", fileName, false);

        try {
            FileReader fr = new FileReader("temp\temp.tmp");
            BufferedReader inFile = new BufferedReader(fr);
            System.out.println(inFile.ready());
            while (inFile.ready()) {
                System.out.println(inFile.readLine());
            }
        }
        catch (FileNotFoundException e) {
            System.out.println(System.currentTimeMillis() + "- Can't find temp.tmp file.");
        }
        catch (IOException e) {
            System.out.println(System.currentTimeMillis() + "- Can't read temp.tmp file.");
        }

        System.out.print("Start of code address: ");
        textAddress = getLong(br);
        System.out.print("Start of data address: ");
        dataAddress = getLong(br);
        System.out.print("Start of bss address: ");
        bssAddress = getLong(br);
        System.out.print("Stack address: ");
        stackAddress = getLong(br);
    }
}
try {
    holdName = linkerScript.createScript(0, 0, name, false,
                                       textAddress, dataAddress, bssAddress,
                                       stackAddress);
} catch (IOException e) {
    e.printStackTrace();
}

// Remove the temp file
// deleteFile.delete("temp\temp.tmp");

return holdName;
package gen;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

public class LinkerScript {

    public String createScript(long d, long e, String name, boolean isDefault,
                                long textAddress, long dataAddress, long bssAddress,
                                long stackAddress) throws IOException {

        String scriptName;
        if (isDefault == true) {
            int index;
            String shorterName;
            index = name.indexOf(".");
            shorterName = name.substring(0, index);
            scriptName = shorterName + "Link" + ".t";
        } else {
            scriptName = name + "Link" + ".t";
        }

        FileWriter fw = new FileWriter("linker-scripts\" + scriptName);
        BufferedWriter bw = new BufferedWriter(fw);
        PrintWriter scriptFile = new PrintWriter(bw);

        scriptFile.println("/* Automatic linker script generation of " +
                        name + " */
                        );

        if (isDefault == true) {
            // Start code then data from lowest memory address.
            scriptFile.println("SECTIONS\n{\n.text : { *(.text) }
.data : { *(.data) }
.bss : { *(.bss) }
}\n.text : { *(.text) }
\n.data :
\n.bss :
\n.text :
PROVIDE (_stack = 0x" + Long.toHexString(e) + ");
            // Set stack to end of memory.
            scriptFile.println("PROVIDE (_stack = 0x" +
                              Long.toHexString(e) + ");
        } else {
            // Use user entered values, hopefully they choose correct ones....
scriptFile.println("SECTIONS
{n. = 0x" + Long.toHexString(textAddress) + ";\n.text : { *(.text) }\n" + ".
= 0x"
+ Long.toHexString(dataAddress) + ";\n.data : { *(.data) }\n" + ".
= 0x"
+ Long.toHexString(bssAddress) + ";\n.bss : { *(.bss) }\n";)
.scriptFile.println("PROVIDE (_stack = 0x"
+ Long.toHexString(stackAddress) + ");\n");
}
scriptFile.close();
return scriptName;
}
/**
 * @author Jonathan Parri Created: December 2008
 * CodeFilter.java implements a file name filter.
 * This is used to search for files based on extensions.
 * It is easier to do this than use a series of datastructures.
 */

package gen;

import java.io.FilenameFilter;
import java.io.File;

//Returns .code and .out files if found
public class CodeFilter implements FilenameFilter {
    public boolean accept(File dir, String name) {
        return (name.endsWith(".code") || name.endsWith(".out"));
    }
}
/**
 * @author Jonathan Parri Created: December 2008
 * Delete.java deletes temporary files without
 * throwing an error if not present.
 */

package streamManager;

import java.io.File;

public class Delete {
    public void delete (String fileName) {
        File f = new File(fileName);
        boolean success = f.delete();
        if (success)
            System.out.println(System.currentTimeMillis()+"- Previous dump overwritten.");
    }
}
/**
 * streamOutput.java captures executable output as redirection
 * is not supported. Results are stored in a text file.
 */

package streamManager;

import java.io.*;

public class streamOutput extends Thread {
    InputStream is;
    String type;
    OutputStream os;

    public streamOutput(InputStream is, String type) {
        this(is, type, null);
    }

    public streamOutput(InputStream is, String type, OutputStream redirect) {
        this.is = is;
        this.type = type;
        this.os = redirect;
    }

    public void run() {
        try {
            PrintWriter pw = null;
            if (os != null)
                pw = new PrintWriter(os);

            InputStreamReader isr = new InputStreamReader(is);
            BufferedReader br = new BufferedReader(isr);
            String line = null;
            while ( (line = br.readLine()) != null )
            {
                if (pw != null)
                    pw.println(line);
                System.out.println(type +">" + line);
                //System.out.println(System.currentTimeMillis() + "- " + line);
            }
            if (pw != null)
                pw.flush();
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }
}
package illustrator;

import java.io.FileOutputStream;

public class GoodWinRedirect {
    public void start(String fileName) {
        try {
            FileOutputStream fos = new FileOutputStream("temp\sizes.tmp");
            Runtime rt = Runtime.getRuntime();
            Process proc = rt.exec("Binaries\jp-sparc-elf-size.exe -B " + fileName);
            // any error message?
            OutputRedirect errorGobbler = new OutputRedirect(proc.getErrorStream(), "ERROR");
            // any output?
            OutputRedirect outputGobbler = new OutputRedirect(proc.getInputStream(), "OUTPUT", fos);

            // kick them off
            errorGobbler.start();
            outputGobbler.start();

            // any error???
            proc.waitFor();
            //System.out.println("ExitValue: " + exitVal);
            fos.flush();
            fos.close();
        } catch (Throwable t) {
            t.printStackTrace();
        }
    }
}
package illustrator;

import java.io.*;

class OutputRedirect extends Thread {
    InputStream is;
    String type;
    OutputStream os;

    OutputRedirect(InputStream is, String type) {
        this(is, type, null);
    }

    OutputRedirect(InputStream is, String type, OutputStream redirect) {
        this.is = is;
        this.type = type;
        this.os = redirect;
    }

    public void run() {
        try {
            PrintWriter pw = null;
            if (os != null)
                pw = new PrintWriter(os);
            InputStreamReader isr = new InputStreamReader(is);
            BufferedReader br = new BufferedReader(isr);
            String line = null;
            while ((line = br.readLine()) != null) {
                if (pw != null)
                    pw.println(line);
                // System.out.println(type + "">" + line);
            }
            if (pw != null)
                pw.flush();
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }
}
/**
 * @author Jonathan Parri Created: December 2008
 * MemoryMapCreator.java uses passed parameters
 * to setup the data required to show the address
 * space map.
 */
package illustrator;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.StringTokenizer;
public class MemoryMapCreator {
    public static int startAddress;
    public static int endAddress;
    public static String[][] references;
    public static String filename;

    public void start(int memBegin, int Address0, int Address1, String fileName) {
        int startMem = memBegin;
        startAddress = Address0;
        endAddress = Address1;
        filename = fileName;

        GoodWinRedirect forceRedirectFromConsole = new GoodWinRedirect();
        forceRedirectFromConsole.start(fileName);
        String holdResults = null;
        String str = null;
        try {
            BufferedReader in = new BufferedReader(new FileReader("temp\sizes.tmp"));
            while ((str = in.readLine()) != null) {
                holdResults = holdResults + str;
            }
            in.close();
        } catch (IOException e) {

        }
        StringTokenizer tokenizer;
        tokenizer = new StringTokenizer(holdResults);
        String[] lesTokens = new String[tokenizer.countTokens()];
        int i = 0;
        while (tokenizer.hasMoreTokens() == true) {
            lesTokens[i] = tokenizer.nextToken();
            i++;
        }

        int textSize = Integer.parseInt(lesTokens[7]);
        int dataSize = Integer.parseInt(lesTokens[8]);
        int bssSize = Integer.parseInt(lesTokens[9]);
// Static assignment..must automate
references = new String[3][3];
references[0][0] = ".text";
references[0][1] = startMem + "";
references[0][2] = (startMem + textSize) + "";
references[1][0] = ".data";
references[1][1] = (startMem + textSize) + "";
references[1][2] = (startMem + textSize + dataSize) + "";
references[2][0] = ".bss";
references[2][1] = (startMem + textSize + dataSize) + "";
references[2][2] = (startMem + textSize + dataSize + bssSize) + "";

// Print the graphs
ShowMem printCharts;
printCharts = new ShowMem();
printCharts.run();
/**
 * @author Jonathan Parri  Created: December 2008
 * ShowMem.java uses static attributes from MemoryMapCreator to draw the address space map on the screen using Java's awt.
 */

package illustrator;

import java.awt.Color;
import java.awt.Container;
import java.awt.Graphics;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class ShowMem extends JPanel {
    private static final long serialVersionUID = -5938180588827323037L;

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.setColor(Color.black);
        g.drawRect(10, 10, 80, 300);
        g.drawString("0x" + Integer.toHexString(MemoryMapCreator.startAddress), 95, 15);
        g.drawString("0x" + Integer.toHexString(MemoryMapCreator.endAddress - 3), 95, 15 + 300);
        int tempStart, tempStop;
        // loop for all sections
        tempStart = Integer.parseInt(MemoryMapCreator.references[0][1]);
        tempStop = Integer.parseInt(MemoryMapCreator.references[0][2]);
        g.drawRect(10, scale(300, 10, MemoryMapCreator.endAddress, MemoryMapCreator.startAddress, tempStart), 80,
                  scale(300, 10, MemoryMapCreator.startAddress, tempStop));
        // -scale(300, 10, MemoryMapCreator.endAddress, MemoryMapCreator.startAddress, tempStart)
        g.drawString("0x" + Integer.toHexString(Integer.parseInt(MemoryMapCreator.references[0][1])), 95,
                          scale(300, 10, MemoryMapCreator.endAddress, MemoryMapCreator.startAddress, tempStart) + 5);
        g.drawString("0x"
+ Integer.toHexString(Integer.parseInt(MemoryMapCreator.references[0][2]), 95, scale(300, 10, MemoryMapCreator.endAddress, MemoryMapCreator.startAddress, tempStop) + 15);  
g.drawString(MemoryMapCreator.references[0][0], 15, ((scale(300, 10, MemoryMapCreator.endAddress, MemoryMapCreator.startAddress, tempStop)) - (scale(300, 10, MemoryMapCreator.endAddress, MemoryMapCreator.startAddress, tempStart))) / 2 + (scale(300, 10, MemoryMapCreator.endAddress, MemoryMapCreator.startAddress, tempStart) + 15));

for (int i = 1; i < MemoryMapCreator.references.length; i++) {
    tempStart = Integer.parseInt(MemoryMapCreator.references[i][1]);
    tempStop = Integer.parseInt(MemoryMapCreator.references[i][2]);
    g.drawRect(10, scale(300, 10, MemoryMapCreator.endAddress, MemoryMapCreator.startAddress, tempStart) + 10, 80, scale(300, 10, MemoryMapCreator.endAddress, MemoryMapCreator.startAddress, tempStart));
    g.drawString("0x" + Integer.toHexString(Integer.parseInt(MemoryMapCreator.references[i][1])), 95, scale(300, 10, MemoryMapCreator.endAddress, MemoryMapCreator.startAddress, tempStart) + 15);
    g.drawString("0x" + Integer.toHexString(Integer.parseInt(MemoryMapCreator.references[i][2])), 95, scale(300, 10, MemoryMapCreator.endAddress, MemoryMapCreator.startAddress, tempStop) + 15);
    g.drawString(MemoryMapCreator.references[i][0], 15, ((scale(300, 10, MemoryMapCreator.endAddress, MemoryMapCreator.startAddress, tempStop)) - (scale(300, 10, MemoryMapCreator.endAddress, MemoryMapCreator.startAddress, tempStart))) / 2 + (scale(300, 10, MemoryMapCreator.endAddress, MemoryMapCreator.startAddress, tempStart) + 15));
}
g.setColor(Color.red);
g.setFont(new java.awt.Font("Arial", 3, 14));
g.drawString(MemoryMapCreator.filename, 80, 350);

public void run() {
    JFrame frame = new JFrame();
    frame.setTitle("Memory");
    frame.setSize(250, 400);
    frame.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    });
    Container contentPane = frame.getContentPane();
    contentPane.add(new ShowMem());
    frame.show();
}

// perform range scaling normalization
private int scale(int rmax, int rmin, int dmax, int dmin, int x) {
    double y;
    y = x * ((float) (rmax - rmin) / (float) (dmax - dmin))
        + ((float) (rmin * dmax - rmax * dmin) / (float) (dmax - dmin));
    return (int) y;
}