GSTAL

The Georgetown Stack Assembly Language

Bryan Crawley

Introduction

The Georgetown Stack Assembly Language (GSTAL) is derived from STAL, a stack assembly language designed by Gerald Wildenberg of St. John Fisher College, Rochester, NY. [1]

The GSTAL virtual machine is a *zero-address machine*. The machine instructions do not include memory addresses. They retrieve their operands from a central stack, and they push their results onto the same stack. The machine's memory architecture comprises code memory, data memory, and three special-purpose registers.

Code Memory

Each code-memory location holds one GSTAL instruction. The first instruction of the current GSTAL program resides at address zero, with addresses increasing consecutively for subsequent instructions. The size of code memory is limited only by the size of the process in which the GSTAL interpreter runs. That is, there is no inherent upper bound on code addresses.

Data Memory

Data memory is an array of 4-byte words, addressable by word and not by byte. The first word resides at address zero, with addresses increasing consecutively for subsequent words. Each memory word can contain either an integer or a floating-point number. GSTAL instructions treat data memory as a stack. The top-of-stack pointer is in the special register called `tos`. (See Special Registers below.) Each instruction fetches its operands by popping them from the stack. The result of the operation, if any, is pushed back onto the stack. When a GSTAL program begins running, the stack is initially empty.

Some of the GSTAL instructions violate the stack abstraction by addressing data memory words other than the top of the stack. These instructions make it possible to store and retrieve variables. See the following pages for complete descriptions of the GSTAL instructions.

The size of data memory is limited only by the size of the process in which the GSTAL interpreter runs. That is, there is no inherent upper bound on data addresses. However, any address reference less than zero or greater than `tos` (see Special Registers below) is erroneous and results in a run-time error.

Special Registers

The GSTAL virtual machine has three special registers called `tos`, `pc`, and `act`. The registers cannot be addressed directly. Rather, their values are altered as side effects of the various GSTAL instructions. The operational semantics on the following pages describe how the registers are manipulated by each instruction. A description of each register follows.

`tos` The address in data memory of the current top entry of the stack. Any address reference greater than `tos` or less than zero is invalid and will result in an execution error. When the stack is empty, `tos` is undefined.

`pc` The program counter. This is the address in code memory of the current GSTAL instruction. The initial value is zero.

`act` The base address in data memory of the current activation frame. This register is relevant only to subroutine calls, returns, and parameter passing.

Input and Output

All GSTAL input comes from the standard input. All output goes to the standard output. The input instructions can read integers and floating-point numbers. The output instructions can write integers, floating-point numbers in exponential form, and individual characters.

Comments and Blank Lines

You can append a comment to the right-hand side of any GSTAL instruction. A comment consists of a semicolon (;) followed by any text extending to the right-hand end of the line. GSTAL does not permit blank lines or lines that contain only a comment with no instruction. Every line of a GSTAL program must contain a GSTAL instruction.

The GSTAL Interpreter

The GSTAL interpreter runs any valid GSTAL program. Before it executes the GSTAL code, it scans the entire program to verify the syntax. If it finds any syntax errors, it reports the errors and aborts the run. If it finds no syntax errors then it runs the program. The GSTAL program terminates under any of these three conditions:

- A `HLT` instruction is executed.
- The physically last statement of the program is executed, and it is neither a `JMP`, `JPF`, nor `RET` that transfers control to another place in the program. In other words, the program halts if it "falls through the bottom" without executing a `HLT` instruction.
- An execution error occurs in the GSTAL code. The interpreter reports all execution errors with appropriate error messages.

Use this syntax to run a GSTAL program at the command-line prompt:

```
gstal <filename>
```

where `<filename>` is the name of a text file that contains a GSTAL program. For example, if you have a GSTAL program in a file called `proj1.g`, then do this:

```
gstal proj1.g
```

Interpreter Options

The interpreter includes two options that are helpful in debugging GSTAL programs. The `-d` option runs the program and produces a stack dump if an execution error occurs. The stack dump is written to a text file called `stackdump`. For example:

```
gstal -d proj1.g
```

The `-l` (lowercase "L") option does not run the program, but instead writes a numbered listing of the program to the standard output. This helps you identify line numbers which may be the targets of `JMP`, `JPF`, or `CAL` instructions. For example:

```
gstal -l proj1.g
```

If you want to save the numbered listing in a file, then redirect the standard output to a text file of your choosing. For example, to write the numbered listing to a file called `proj1.listing`, do this:

```
gstal -l proj1.g > proj1.listing
```

References

[1] 1990. Wildenberg, Gerald. *Using a Stack Assembler Language in a Compiler Course*, *SIGCSE Bulletin* **22**, *No. 4*: p. 43 (December).

Integer Arithmetic

Op Code	Description	Semantics	Argument

ADI	Addition	b = pop();
		a = pop();
		push(a+b);	

SBI	Subtraction	b = pop();
		a = pop();
		push(a-b);	

MLI	Multiplication	b = pop();
		a = pop();
		push(a*b);	

DVI	Division	b = pop();
		a = pop();
		push(a/b);	

NGI	Negation	a = pop(); push(-a);	

Floating-Point Arithmetic

Op Code	Description	Semantics	Argument

ADF	Addition	y = pop();
		x = pop();
		push(x+y);	

SBF	Subtraction	y = pop();
		x = pop();
		push(x-y);	

MLF	Multiplication	y = pop();
		x = pop();
		push(x*y);	

DVF	Division	y = pop();
		x = pop();
		push(x/y);	

NGF	Negation	x = pop(); push(-x);	

Integer Relational Operations

Op Code	Description	Semantics	Argument

EQI	Equal To	b = pop();
		a = pop();
		push(a==b);	

NEI	Not Equal To	b = pop();
		a = pop();
		push(a!=b);	

LTI	Less Than	b = pop();
		a = pop();
		push(a<b);	

LEI	Less Than Or Equal To	b = pop();
		a = pop();
		push(a<=b);	

GTI	Greater Than	b = pop();
		a = pop();
		push(a>b);	

GEI	Greater Than Or Equal To	b = pop();
		a = pop();
		push(a>=b);	

Floating-Point Relational Operations

Op Code	Description	Semantics	Argument

EQF	Equal To	y = pop();
		x = pop();
		push(x==y);	

NEF	Not Equal To	y = pop();
		x = pop();
		push(x!=y);	

LTF	Less Than	y = pop();
		x = pop();
		push(x<y);	

LEF	Less Than Or Equal To	y = pop();
		x = pop();
		push(x<=y);	

GTF	Greater Than	y = pop();
		x = pop();
		push(x>y);	

GEF	Greater Than Or Equal To	y = pop();
		x = pop();
		push(x>=y);	

Data Type Conversion

Op Code	Description	Semantics	Argument
FTI	Floating-Point to Integer	x = pop(); push((int) x);	
ITF	Integer to Floating-Point	a = pop(); push((float) a);	

Input and Output

Op Code	Description	Semantics	Argument
PTI	Print Integer	a = pop(); printf("%d",a);	
PTF	Print Floating-Point	x = pop(); printf("%e",x);	
PTC	Print Character	a = pop(); printf("%c",a);	
PTL	Print Newline Character	printf("");	
INI	Input Integer	scanf("%d", &a); push(a);	
INF	Input Floating-Point	scanf("%f", &x); push(x);	

Stack Manipulation

Op Code	Description	Semantics	Argument
LLI <arg>	Load Literal Integer	push(arg);	<arg> is an integer.
LLF <arg>	Load Literal Floating-Point	push(arg);	<arg> is a floating-point number.
ISP <arg>	Increment Stack Pointer	tos = tos + arg;	<arg> is a non-negative integer.
DSP <arg>	Decrement Stack Pointer	tos = tos - arg;	<arg> is a non-negative integer.

STO	Store	b = pop();
		a = pop();
		datamem[a] = b;	

STM	Store Memory	b = pop();
		a = pop();
		datamem[b] = a;
		push(b);	

LOD	Load	a = pop(); push(datamem[a]);	

Flow Control

Op Code	Description	Semantics	Argument
LAA <arg>	Load Absolute Address	push(arg);	<arg> is a non-negative integer.
LRA <arg>	Load Relative Address	push(act+arg);	<arg> is a non-negative integer.
JMP <arg>	Unconditional Jump	pc = arg;	<arg> is a non-negative integer.

JPF <arg>	Jump If False	a = pop();
		if (a==0)
		  pc = arg;
			<arg> is a non-negative integer;

PAR <arg>	Load Parameter Address	push(act-arg);	<arg> is a non-negative integer.

CAL <arg>	Call Subroutine	push(act);
		act = tos;
		push(pc);
		pc = arg;
			<arg> is a non-negative integer.

RET	Return From Subroutine	pc = datamem[act+1] + 1;
		tos = act-1;
		act = datamem[act];	

NOP	No Operation	<do nothing>	
HLT	Halt	<execution terminates>	