BBN Technical Memorandum No. 1220

Introduction to Spanner: Assembly Language for the Smart Packets Project

Job No. 11822000

September 27, 1999

Prepared for:
Defense Advanced Research Projects Agency
Computer Systems and Technology Office
3701 North Fairfax Drive
Arlington, VA 22203-1714

Prepared by:
GTE
10 Moulton Street
Cambridge, MA 02138
Introduction to Spanner: Assembly Language for the Smart Packets Project

Beverly I. Schwartz

Spanner is a CISC assembly language with high level types created at BBN Corporation for the Smart Packets project. Spanner is designed with compact representation as a goal; a Spanner program must fit into a single unfragmented packet along with header and authentication information. Because of the prominence of ethernet in today’s LAN’s, it is expected that Spanner programs will have to fit in about 1000 bytes.

Spanner was created to be an assembly language for Sprocket, a high-level language resembling C, created by BBN for Smart Packets. See sprocket.txt for more information on Sprocket. Although Spanner was created to support Sprocket, meaningful Spanner programs can be written without too much pain. A Spanner programmer must have an understanding of the Spanner virtual machine before writing Spanner programs.

This document will give a brief description of the virtual machine, enough for writing Spanner programs. It will describe Spanner syntax. It will not enumerate all of the Spanner operations. These will be discussed in the document primitive.txt.

The data file data/primchart definitively enumerates all Spanner operations, the arguments the operations take, and how the operations manipulate the stack and condition code. Parts of the assembler and virtual machine code are generated from this data file - if the data file is changed, the implementation of the operations will also change. In other words, data/primchart is the final word on what Spanner really is.

The virtual machine has a stack-based CISC architecture. The virtual machine has a stack which is manipulated by most Spanner operations, variable sets to store variables, a condition code for reporting the results of operations and frames for managing subroutines.

The stack can contain value/type pairs, or references to variables or array/list elements. A Spanner instruction can get its arguments from the stack and can put its result on the stack. When an instruction gets an argument from the stack, it may remove the item from the stack or it can leave it there. If an instruction produces a result, it may or may not push the result on the stack, depending on what the default action is for the instruction and whether or not the default was overridden. data/primchart indicates, for each Spanner operation, the default action for stack manipulation and what gets pushed on the stack if something does get pushed. data/primchart also indicates where Spanner operation arguments can come from; not all can use stack entries.

When arguments are resolved in an instruction, the _rightmost_ stack indicator corresponds to the top of the stack, the next one from the right corresponds to the next entry down, etc.

When a subroutine is called, a frame marker is put on the stack. While the subroutine is active (it
is active until a return call is made), stack access cannot cross the frame marker. If a stack access
does cross the frame marker, an exception is generated and the program is terminated.

Most Spanner operations affect the condition code, data/primchart indicates how operations
determine whether to set or clear the condition code. The only operations which use the
condition code are the conditional branch and jump instructions, which determine whether to
branch/jump based on whether the condition code is set or cleared.

Spanner uses variables. Variables are stored in variable sets. When executing a subroutine,
variable declarations are put in a local variable set for the subroutine; if no subroutine is being
executed then variable declarations are put in the global variable set. If a variable of the same
name is found in the variable set, the virtual machine checks if it is the same type. If it is, the
variable is reinitialized; if not, an exception is generated and the program terminates.

When variables are used, the virtual machine looks in one or two places to find the variable. If a
subroutine is being executed, it will first look in the local variable set; if not found there it will
then look in the global variable set. If no subroutine is being executed, it only looks in the global
variable set. If the variable is not found, an exception is generated and the program is
terminated.

Spanner programs have a syntax similar to many assembly languages; a Spanner statement is on
a single line and only one statement per line. Each line starts with an operation and is followed
by arguments. A statement can be preceded by an address label. The address label is
immediately followed by a colon.

Spanner programs can contain comments; anything that follows a semi-colon until the end of the
line is considered a comment. Comments can be put anywhere within a Spanner program.

Spanner has two assembler directives, .IGNORE and .LOOK. These directives must be placed at
the first column of a line. Anything after an .IGNORE directive is ignored until a .LOOK is
found. A .LOOK indicates that Spanner code follows. If neither directive is used, .LOOK is the
default mode.

All Spanner operations are enumerated in data/primchart. (Also see primitive.txt.) Operations
can have modifiers which take the form of a dash (-) followed by the modifier name.

Each operation has a default as to whether or not the results of the operation should be pushed
on the stack. This default can be overridden with a modifier which indicates to push (+pu) or not
to push (+np). For some operations, push/no-push is irrelevant and so using a +pu or +np
modifier will generate an assembler error. In data/primchart, "NA" appears in the p/np column
for these operations.

Operations decl, cast, and sbr must take a type modifier. A type modifier is a dash followed by
the type name. For decl, the type modifier indicates the type being declared. For cast, the type
modifier indicates the result type of the cast. For sbr, the modifier indicates the return type of the
subroutine. Spanner types are: u8, u16, u32, u64, u128, s8, s16, s32, s64, s128, f32, f64, bool, str;
addr, pkt, spid, and mib. sbr can also be modified with +void, indicating the subroutine does not
return any value.

decl can also be modified by -arr, -lst, +prm, -ref, -1by or -2by. -arr indicates the variable is an
array of the indicated type. -lst indicates the variable is a list of the indicated type. If both -arr
and -lst are used, then the variable is an array of lists of the indicated type.

+prm indicates the declaration is a parameter to a subroutine. -ref indicates the declaration is a
parameter to a subroutine passed by reference. If both +prm and -ref are used, it is the same as if
-ref appears alone. If only +prm is used, then the parameter is passed by value to the subroutine.

-1by and -2by modifiers are directives to the assembler rather than a language feature. Variable
names can be encoded in one byte or two bytes; if there are fewer than 191 variables, they will all
be encoded in one byte. If a Spanner program contains more than 191 variables, the programmer

 can indicate which variable names should take priority for one byte encoding, and which ones

 are used rarely and can afford two byte encoding. See the document encoding.txt for more

 information on encoding variable names.

Operations are followed by arguments. Arguments can come in one of four forms: a variable, a

 literal, an address label or from the stack. A stack argument can indicate whether or not the stack

 entry should be removed from the stack. All variables are preceded by a "\%". All literals are

 preceded by a "#". All address labels are preceded by a "@". "@" indicates using a stack entry and

 removing the entry. "&" indicates using a stack entry, but the entry should be left on the stack.

Address labels and variable names must fit the regular expression \[a-zA-Z_][a-zA-Z09_\-]*\]. In

 English, this means labels and names must begin with an alphabetic character or underscore, and

 can be followed by zero or more alphanumeric characters, dots, or underscores.

Literals are preceded by a #. The following table shows how to represent different types of

 literals. Packets and nub variables cannot be represented by a literal.

<table>
<thead>
<tr>
<th>Type</th>
<th>How to represent it</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero</td>
<td>0</td>
</tr>
<tr>
<td>decimal integer</td>
<td>-?[1-9][0-9]*</td>
</tr>
<tr>
<td>octal integer</td>
<td>0[0-7]+</td>
</tr>
<tr>
<td>hexadecimal integer</td>
<td>0x[0-9a-fA-F]+</td>
</tr>
<tr>
<td>floating point number</td>
<td>[+-]?[1-9][0-9]*</td>
</tr>
<tr>
<td>boolean</td>
<td>T, TRUE, True, true, 1</td>
</tr>
<tr>
<td>string</td>
<td>Any sequence of characters enclosed in double quotes. A double quote or a backslash inside the string must be preceded by a backslash. Special characters, enumerated below, can be embedded in a string. If a backslash is followed by anything other than what’s listed below, the backslash is just dropped.</td>
</tr>
</tbody>
</table>

    | \a | \b | \f | \n | \r | \t | \0 |
Type       How to represent it
---------- ----------------------
address    Both IPv4 and IPv6 addresses are recognized. IPv4 addresses come in the form num.num.num.num where num can be any value from 0 through 255, inclusive. Any value for num can be preceded by a zero, and it will still be interpreted as a decimal number, not an octal number. IPv6 addresses can be expressed as defined in section 3.1.1, pages 40-41 of "IPv6: The New Internet Protocol" by Christian Huitema.

spid       < address, non-negative integer, non-negative integer >

array      A comma separated list of the members of the array, delimited by [ ]. For multi-dimensional arrays, each dimension must be delimited by [ ]. Arrays can have a maximum of three dimensions. If dimensions differ in the array literal, the largest one is taken for the dimension. Where values are missing in the literal, the array member will be initialized to zero (or equivalent).

list       A comma separated list of members of the list, delimited by ( ).

A standalone virtual machine is supplied as part of this distribution. See the README file contained within the software distribution for instructions on how to run it. Command line arguments can be provided to the standalone machine to tell it to print out each instruction as it executes, the state of the stack with each instruction, as well as other information about the state of the virtual machine and its components.

The rest of this document contains a sample Spanner program. This program gets all of the addresses and MTU associated with each interface on a host. This program encodes to 54 bytes; it makes use of stack manipulation to optimize the compactness of the program.

There are many comments which describe why the program is structured the way it is and how it works.
; declare an array of addresses - this will be used later when asking for
; all of the addresses corresponding to each interface
; decl pushes a reference to the variable by default; in this declaration,
; the default is overridden with the -np modifier
decl addr arr -np %addresses

; declare a packet; this will be filled with the information we get from
; the interfaces
; since we will soon use this variable, do have a reference to this variable
; pushed on the stack
decl pkt %pkt

; get the number of interfaces on this host; the result is pushed on the stack
niface

; put the result of the previous command (number of interfaces) into the
; packet being sent back to the host
; The & refers to the number of interfaces (the topmost value on the stack),
; and will leave this value on the stack.
; The @ refers to the reference to the variable %pkt, the next thing down
; on the stack. The reference to %pkt will be popped off the stack.
papp @ &

; declare a counter, initialize it to 1
; this counter will be used to query each interface
; we use type u16 because niface returns a u16 (see data/primchart)
; we could use a different type since the virtual machine will do automatic
; type promotion. However, if we use a u8 and the host has more than 255
; interfaces, the program will not work correctly. if we use a larger
; integer type like u32, the encoding will be larger. So the ideal type
; to use is the one that matches the return from niface.
decl u16 %index #1

; put an address label before this statement since this is where the program
; will loop back to when processing the next interface
; the first & corresponds to the value returned from niface
; the second & corresponds to a reference to the variable %index
; it (the less than operation) compares the number of interfaces to the
; value of the interface counter (%index)
; if the number of interfaces is less than the value of %index, then the
; condition code is set, otherwise it is cleared
$loop: 1t & &

; if the condition code is set (in other words, if we have looked at all of
; the interfaces), branch to the address label $done
bret $done

; put the interface number into the packet (represented by the variable %pkt)
; the top stack entry is still a reference to the variable %index
papp %pkt &

; get the number of addresses which correspond to this interface (a reference
; to %index is still at the top of the stack)
; push the result of this operation on the stack (default for naddr is to
; push the result)
naddr &

; see if we have any addresses (the top of the stack is the number of
; addresses)
; the condition code will be set if the number of addresses is not
; equal to 0
ne0 &

; if the condition code is set (means we do have some addresses), jump to
; the code which processes the addresses
    brt $do_addresses

; since we don’t have any addresses, pop the number of addresses off the stack
    pop

; and now jump over all of the instructions that process addresses
    bru $skip_addresses

; make %addresses a one-dimensional array
; the length of the array is the result of the call to naddr (this value
; is still the top of the stack)
; the result of the naddr call will be popped
; a reference to %addresses is pushed on the stack as a result of the sdim
; operation (see data/primchart)
$do_addresses:
    sdim %addresses @

; get the addresses for the interface
; the first argument is the interface (which is the second item on the stack)
; the second argument is a reference to the array of addresses to be filled
; (which is on the top of the stack)
; both arguments will stay on the stack, and gaddr will not put anything
; on the stack after execution (see data/primchart)
    gaddr & &

; put the addresses into the packet which will go back to the source (%pkt)
; the second argument is the reference to the addresses, which will be
; removed from the top of the stack
; this operation does not put anything on the stack
    papp %pkt @

; get the mtu for the interface
; the argument is the interface number - the reference to %index is now on
; the top of the stack
; the mtu is pushed on the stack
$skip_addresses:
    mtu &

; put the mtu in the packet (%pkt)
; the second argument is the result of mtu, and it is popped off the top
; of the stack
; this instruction does not push anything on the stack
    papp %pkt @

; increment the interface counter (%index)
; ainc normally pushes a reference to the counter on the stack; override the
; default and don’t push the reference (since it is already on the stack)
    ainc-&np &

; branch back to $loop and process the next interface
    bru $loop

; if we’re here, it means we have processed all of the interfaces
; send the packet (%pkt) back to the originating host
$done:
    send %pkt

; continue delivery - send this program to the next hop
    cont