Agenda: Tuesday, June 14

Chapter 2: SAL - Simple Abstract Language

- Motivation for SAL
- Programming Language Requirements
- SAL Examples
- Procedures

Sending e-mail needed to complete P0 (due Thursday)

Motivation for learning SAL

1. SAL is a "fake" assembly language. There is no machine that can run SAL code directly.
2. We learn SAL to bridge the gap between HLLs and true assembly languages.
    a. It abstracts (hides some details) of true assembly languages.
    b. We will learn about the hidden details later.
3. This is a "Top-down" approach to learning about Computer Architecture.

HLL --> SAL --> assembly language --> machine code
HLL --> SAL --> MAL --> TAL --> MIPS RISC machine code

<u>SAL</u>

- subset of the functionality of most high level languages
- no records or structures.
- no formal arrays (but they can be implemented- see ch 7)
- one instruction, declaration per line
- comments are anything on a line following '#'

<u>MAL</u> - the MIPS assembly language

<u>TAL</u> - is a true assembly language for MIPS each SAL instruction maps to one or more MAL instructions

each MAL instruction maps to one or more TAL instructions

each TAL instruction maps to one line of MIPS RISC machine code

---

Programming Language Requirements

1. Declarations

- tell how much space in memory is needed for a variable
- assign (attach) a name or label to a the reserved space

There are three basic types

- integer (.word)
- floating point or real (.float)
- character (.byte)

```
Value is optional- it gives the variable an initial value

Other types can be created out of these three:
for example,
      boolean is really an integer with only 2 defined values.
```

C/C++/Java example

```
vartype varname [= initvalue] ;
int i = 3;
```

Sal example

```
variablename:  type value
```

Unlike many HLLs, in most assembly languages (including SAL) all declarations are grouped together and placed in a separate (data) section of the program code.

Each type of data requires a different amount of space to be reserved.

- Integers need one "word" of space or 32 bits
- Characters need one "byte" or 8 bits
- Floating point values need 32 or 64 bits

SAL declaration examples (in a section defined by the data directive)

```
         .data        #directive
x:       .word       # space for an integer
                      # integer-sized variable, defaults to
                     # an initial value of 0

y:       .word  250
int:     .word  3

letter:  .byte
ch:      .byte  'b'    # space for a character
newline: .byte  '\n'  # space for a new line character
                      # reminder: '\n' is one character
A:       .byte  'A'

e:       .float 2.71828  # space for a floating point value
```

Note:

- only be only declaration per line.
- The default initial value is always 0.
- Comments are anything on a line following ‘#’ (comments may not span lines)
- Read code linearly (top to bottom)

SAL Directives

Directives- special instructions used to give additional information to the assembler.
- they give information about how much memory space is needed
- they can label (assign a name to) the memory space

** not executed as part of the program.
The character '.' is used to indicate a directive in SAL and MAL.

<u>common directives</u>:

```
 .data     # identifies the start of the variable declaration section
            #   There can be more than 1 .data section in
            #     a program.
            #   There will be 1 global location where data
            #     from all .data sections is placed.
 .byte
 .word
 .float

 .text       # identifies where instructions are
            #   There can be more than 1 .text section in
            #   a program.

 .asciiz  "a string.\n"  # places a string into memory
                         # and null terminates the string.

 .ascii  "new string."   # places a string into memory
                         # WITHOUT null termination.
       .space          # used to reserve a specific number of bytes

        __start:        # label to start program
                        # identifies the first instr to execute
        done            # syscall to end program
```

```
The variable names are labels.  Labels (in SAL/MAL) should start
with a letter of the alphabet ('A'-'Z', 'a'-'z'), and may be
followed by other letters, digits or the underscore character ('_').

Some useful characters:
      '\n'  the newline (a line feed followed by a carriage return)
      '\t'  horizontal tab
      '\\'  the backslash character
      '\"'  the double quote mark
      '\0'  the null character  (for SAL/MAL, appended to the end of
                                 a string to identify the end of the
                                string.)
```

SAL Arithmetic Operations

- The type of the result of the operation depends on the type of variables
- cannot increase the number of operands
- y and/or z can be IMMEDIATES, but x can not

```
ARITHMETIC operations
----------------------

SAL                   C or C++ or Java
-----------------------------------------
move x, y              x = y;
add x, y, z            x = y + z;
sub x, y, z            x = y - z;
mul x, y, z            x = y * z;
div x, y, z            x = y / z;  (gives quotient)
rem x, y, z            x = y % z;  (gives remainder)
```

There are other instructions that implement boolean functions, but we don't cover them yet. (not, and, or, xor, nand, nor)

C/C++ examples	**SAL examples**
`count = 0;`	`move x, y`
`x = a + b;`	`add x, a, b`
`y = c - d;`	`sub y, c, d`
`z = e * 35;`	`mul z, e, 35`
`result = 3 / numstudents;`	`div  result, 3, numstudents`
`remainder = total % 3;`	`rem  remainder, total, 3`

SAL Conditional Operations

Conditional execution is when some condition or test is used to determine if a specific set of instructions will be executed.

In most HLLs, this is usually an "if" statement. If the condition is true some block of instructions are executed, otherwise (else) a different block of instructions (or none) are executed.

At the machine level, this is called a **conditional branch**. If the condition in the instruction is true, then we branch (jump) to a different instruction in the program.

```
In the mnemonics,
          b   stands for branch
          g              greater
          l              less
          t              than
          e              equal to
          z              zero
```

SAL 'ifs' and 'gotos'

SAL	C/C++ (kind of)
b label	goto label;
bltz x, label	if (x < 0) goto label;
bgtz x, label	if (>) goto label;
blez x, label	if (<=) goto label;
bgez x, label	if (>=) goto label;
beqz x, label	if (==) goto label;
bnez x, label	if (!=) goto label;
beq x, y, label	if (x == y) goto label;
bne x, y, label	if (!=) goto label;
blt x, y, label	if (<) goto label;
bgt x, y, label	if (>) goto label;
ble x, y, label	if (<=) goto label;
bge x, y, label	if (>=) goto label;

<u>C/C++ examples</u>

```
if (x < 0) {
   < more instructions here >
}
```

<u>SAL examples</u>

```
    bgez x, label
    < more instructions here >
label:
```

Structured loops like those used in HLLs (do-while & for), can be built out of ifs and gotos. Combine a condition test with a branch.

About Labels

A label is an identifier. It follows the same rules as those given for identifiers (variable names).

A label identifies a location (an address).

The syntax for the use of a label places the label first, and follows it with a colon.

Examples of labels that you have already used:

```
        count:       .word  0
        my_string:   .asciiz "Here is my string, ready to go!\n"
```

Each of these examples assigns a human-readable mneumonic to an address (assigned by the assembler).

The same may be done within code. These labels are necessary in the case of identifying the instruction which is the target of a branch instruction. But, we could also add unnecessary labels.

Examples of labels that might be unnecessary:

```
  label1:
             add  x, y, z
  label2:
  label3:
             sub  aa, bb, cc
             putc char8
```

In this code fragment, the address assigned for both label2 and for label3 is the same. No syntax rules are broken by having more than one label for the same thing.

It could get confusing, as well as misleading. Fortunately, our simulator disallows this double labeling of items within the .data section.

```
  .data
count1:
count2:  .word 0
str1:    .asciiz  "count1 is "
str2:    .asciiz  "count2 is "
newline: .byte    '\n'
  .text
__start:
         add   count1, count1, 1
         add   count2, count2, 1
         puts  str1
         put   count1
         put   newline
         puts  str2
         put   count2
         done
```

The simulator gives the following output for this program:

```
spim: (parser) Unknown type on line 12 of file lotsalabels.s
          add   count1, count1, 1
                                ^
spim: (parser) Type mismatch on line 12 of file lotsalabels.s
spim: (parser) Unknown type on line 12 of file lotsalabels.s
spim: (parser) Unknown type on line 12 of file lotsalabels.s
spim: (parser) Unknown type on line 18 of file lotsalabels.s
    put   count1
               ^
spim: (parser) Unknown type on line 18 of file lotsalabels.s

put   count1
               ^
count1 is
count2 is 1
```

Examples

C equivalent:

```
if (count < 0)
  count = count + 1;
```

SAL equiv to if-then-else:

```
            bltz count, ifstuff
            b endif
 ifstuff:   add count, count, 1
 endif:       # next program instruction goes here
```

-- OR --

```
              bgez count, endif
              add count, count, 1
 endif:       # next program instruction goes here
```

***** last one is best**

Examples of compound conditionals:

C/Java:

```
if ( (x < y) || (w == z) ) {
    a = a + 1;
}
```

One possible SAL equivalent:

```
           blt   x, y, increment  # no need to check second
           bne   w, z, no_increment # condition if first is True
increment: add   a, a, 1
no_increment:
```

C/Java:

```
if ( (x < y) && (w == z) ) {
    a = a + 1;
}
```

One possible SAL equivalent:

```
        bge   x, y, no_increment   # must check second
        bne   w, z, no_increment   # condition if first is True
        add   a, a, 1
    no_increment:
```

Example: while loop

C:

```
while (count > 0) {
  a = a % count;
  count --;
}
```

SAL:

```
while: blez  count, endwhile
       rem a, a, count
       sub count, count, 1
       b while
endwhile:    # next program instruction goes here
```

---

repeat loop example

(NOTE: This example shows an implementation of nonsense code.)

C:

```
/* do statement while expression is TRUE */
/*   when expression is FALSE, exit loop */
do {
    if (aa < bb)
        aa++;
    if (aa > bb)
        aa--;
} while( aa != bb);
```

SAL:

```
repeat:    bge aa, bb, secondif
           add aa, aa, 1
secondif:  ble aa, bb, until
           sub aa, aa, 1
until:     bne aa, bb, repeat
```

C:

```
while ( (count < limit) && (c==d) )
{
 /* loop's code goes here */
}
```

SAL:

```
while:   bge count, limit, endwhile
         bne c, d, endwhile

         # loop's code goes here

         b while
endwhile:
```

<u>Example: for loop</u>

C:

```
for ( i = 3; i <= 8; i++)
{
    a = a + i;
}
```

SAL:

```
         move i, 3
for:     bgt  i, 8, endfor
         add  a, a, i
         add  i, i, 1
     b for
endfor:
```

[More SAL conditional examples]

Communication with user

SAL has simple read (get) and write (put) commands for communicating with the user of a SAL program. The following table shows a close match in a HLL.

SAL	C++	Java
`put x`	`cout << x;`	`System.out.print(x);`
(x is either 1 char or an int)		
`puts msg`	`cout << msg;`	`System.out.print(msg);`
(msg is a string)		
`get x`	`cin >> x;`	`x = stdin.read();`

`get x`, where x is an integer variable. (.word)
SAL will read input from the user and interpret it as an integer. If a non-integer character is found before a valid integer, the value returned is zero. It will discard the rest of the line.

`get c`, where c is a character variable. (.byte)
SAL will read one character and place it into the variable c.
It will not discard the rest of the line.

Examples: (input)

```
    >23 abc
    > -13
    >1234fgh!

SAL Code (each using the same input above):
    get int1     # int1 <-- 23
    get int2     # int2 <-- -13
    get int3     # int3 <-- 1234
```

```
    OR: (1st line)
    get char1    # char1 <-- '2'
    get int1     # int1  <-- 3   (found therefore discard rest
                                  of the line)
    get char2    # char2 <-- ' ' (2nd line)
```

```
   OR: (1st line)
    get char1   # char1 <-- '2'  (read char by char)
    get char2   # char2 <-- '3'
    get int1    # int1  <-- 0, because the first value read
(ignoring white space)
                # is not a digit, so the type does not match.
```

** To get more than one non-character value from a single line of input, you must read input character by character, and convert to whatever form is desired. (More in Chapter 4)

A Simple Example (either calling get ch or get y)

```
     .data
x:   .word    3
msg: .asciiz "hi"
ch:  .byte
y:   .word

     .text
put x        # x can be int, char, float
puts msg
get ch       # ch is a character
get y
done
```

The SAL "get" instruction has some interesting results

Input	returned by "get y"	returned by "get ch"
23	23	2
-13	-13	-
3, hello	3	3
" "123hi	123	" " (" " are spaces)
13.2	13	1
hi	0	h

SAL Program Examples

```
# this simple program adds up 2 integers and prints their sum and
products.

   .data
 prompt1:  .asciiz "Enter an integer: "
 prompt2:  .asciiz "Enter a second integer: "
 linefeed: .byte '\n'
 msg1:     .asciiz "The sum of "
 msg2:     .asciiz " and "
 msg3:     .asciiz " is "
 msg4:     .asciiz "The product of "
 int1:     .word  0
 int2:     .word  0
 sum:      .word
 product:  .word

   .text
      # get the 2 integers from user
 __start:  puts prompt1
           get int1
           put linefeed
           puts prompt2
           get int2
           put linefeed
      # calculate the sum and products
           add sum, int1, int2
           mul product, int1, int2
      # print out the sum and products
           puts msg1
           put int1
           puts msg2
           put int2
           puts msg3
           put sum
           put linefeed

           puts msg4
           put int1
           puts msg2
           put int2
           puts msg3
           put product
           put linefeed
           done
```

```
On Screen:
Enter an Integer
      5
Enter a second Integer
      3
(sum:3, prod:15)
The sum of 5 and 3 is 8
The product of 5 and 3 is 15
```

For the students to try at home.

```
# A SAL program to print out a multiplication table

 .data
start:    .word 0  # entered by user
finish:   .word 0  # entered by user
ii:       .word    # loop induction variable
jj:       .word    # loop induction variable
product:  .word
prompt1:  .asciiz "Enter starting value: "
prompt2:  .asciiz "Enter ending value: "
newline:  .byte '\n'
x_symbol: .byte 'X'
equals:   .byte '='
space:    .byte ' '

 .text

__start:    puts prompt1                # get user input
            get  start
            puts prompt2
            get  finish

            move ii, start
for:        bgt  ii, finish, all_done   # nested for loop to print out
            move jj, start              # the table
nested:     bgt  jj, finish, next_iter
            mul  product, ii, jj

            # print one line of table
            put  ii
            put  space
            put  x_symbol
            put  space
            put  jj
            put  space
            put  equals
            put  space
            put  product
            put  newline

            add  jj, jj, 1
            b    nested
next_iter:  add ii, ii, 1
            put  newline
            b    for
all_done:   done
```

Procedures

SAL has only rudimentary methods for procedure call and return.
There is no explicit mechanism for parameter passing or function return values.
However, you will see how you can implement this functionality.
(It just won't be as convenient as you would like.)

Parts of a procedure

1. The call to the procedure
   Example: `b procname`

2. The execution of the procedure's code

3. The return from the procedure
   This is the hard part. (branch or jump)

By adding a label to the return instruction, we can branch to that instruction when the procedure is complete.

 # A BAD example

```
             b procname
  rtn1:      # more code here


procname:    # procedure code here
        .
        .
        .
             b rtn1
```

Unfortunately, this is not a procedure. It just jumps to a different place in the code and the "procedure" cannot be called from more than one location. No matter where the procedure is called from, it returns to the same location.

We need an ADDRESS to return to!

Example:

```
In main program:

      ...
      y = abs(x);     <--call--need a branch
      y = y + 1;      <--return point/address
      ...


(function:)
      int abs (x);
      int x, y;
      {
          if (x < 0) {
           y = -x;
          }
          else {
           y = x;
          }
      return(y);
      };   <-- return -- branch back


Ignore parameters, simplest SAL:

      ...
      b abs
return:   add y,y,1
      ...

abs: bgez x, nonnegative
      sub y, 0, x
      b endabs
nonnegative: move y, x
endabs:   b return            # really needs (return here)
```

Try x=1 & x= -1

---

an address- a label for a specific spot in memory.

Load Address- SAL instruction that can put the address of a label into a variable.

```
la var1, label
```

The address implied by `label` is placed into `var1`.
var1 must be declared as an integer (.word)

So, `var1` is a POINTER to the memory with the label `label`.

Notice difference between address and contents of the address.

```
       label        address       contents
        aa:           103            6
        bb:           104           'a'
        cc:           105          2001
```

The SAL instruction **`la cc, bb (take address of bb and store it in cc)`** changes the table above to be:

```
       label        address       contents
        aa:           103            6
        bb:           104           'a'
        cc:           105        **  104  **
```

For procedure call and return, save a return address before branching to the procedure.

```
            la  procname_ret, rtn1
            b procname
rtn1:       # more code here
.
.
.
procname:   # procedure code here
            .
            .
            .
            b procname_ret     ****variable
```

THIS STILL DOESN'T WORK!

It branches to label `procname_ret`.
But, `procname_ret` is a variable! We do NOT want to branch to a variable!
To solve the problem, there is a special form of the **`b`** instruction used only for procedure return.

`b (var1) #parentheses identify the special form`
This branches to the contents of var1, not to var1 itself.

So, the complete and correct SAL call/return code is:

```
                la  procname_ret, rtn1    # one call
                b   procname
      rtn1:     # more code here
    |
2nd | .
    | .
    | .
    |           la  procname_ret, rtn2    # a second call
    |           b   procname
      rtn2:     # more code here
    | .
4th | .
    | .
      procname: # procedure code here
    |           .
1st,|           .
3rd |           .
    |           b (procname_ret)          # procedure return
```

```
Prior abs() example: Change SAL procedure:

abs: bgez x, nonnegative
     sub y, 0, x
     b endabs
nonnegative: move y, x
endabs:  b returnhere     <-- change, NOT!

Must be:

endabs:  b (returnhere)   <-- Parentheses are important
```