Chapter 3 -- SASM

ABOUT SASM

MOTIVATION for SASM:
hiding the details of Pentium asm. lang. (on purpose!)
SASM code will look more like HLL code -- to make student's transition easier.
Introducing one more level of abstraction in order to postpone discussion of several topics.

HLL SASM assembly machine code

each HLL statement maps into 1 or MORE SASM instructions
each SASM instruction maps into 1 or MORE Pentium instructions

SASM -- the language

A subset of the functionality of most high level languages --
no records/structures
no formal arrays
no procedures/functions

What is required by a programming language?
declarations
arithmetic operations
conditional execution (if then else)
looping control structures
communication w/user. . .(write statement)

About SASM:
-- one instruction, declaration per line
-- comments are anything on a line following `;'
(comments may not span lines)
-- given the Intel architecture and its history, there are an enormous number of RESERVED WORDS. Consult APPENDIX A always!

DECLARATIONS

- they give information about how much memory space is needed
- they assign a name to the memory space

SASM has 3 basic types: integer, float (real), character
can build other types out of these,
for example, boolean is really an integer with only 2 defined values.

```
  Pascal:
       var        variablename:  type;

  C or C++:
       type     variablename;

  SASM:
       variablename  type value


     type is       dd      if integer
                   db      if character
                   dd      if floating point

     value is required -- it gives the variable an initial value
                       -- to explicitly leave value undefined, use
                          the '?' character

  examples:
       bool_flag   dd  0

       counter     dd  0

       variable3   dd   ?

       constant_e  dd   2.71828

       uservalue   db  ?

       letter_a    db 'a'

       string1     db 'This is a string.', 0
         ; null terminated string example, VERY USEFUL!

       string2     db 'Another string', 0ah, 0
         ; that 0ah is the newline character, AND this string is
         ; null terminated.



remember:
  -- one declaration per line.



DIRECTIVES
----------
  a way to give information to the assembler.

  - some directives start with `.' (period)

    examples:

         dd          # tells the assember to allocate 32 bits
         db          # tells the assember to allocate 8 bits

         .data       # identifies the start of the declaration section
                     #   there can be more than 1 .data section in
                     #   a program
```

```
        .code       # identifies where instructions are
                    #   there can be more than 1 .code section in
                    #   a program


        .stack      # You get this set of memory, called a stack.
                    #   Don't worry about it for now, just use it.

        .model      # Gives the assembler information about how to
                    #   place stuff in memory, and how to call stuff
                    #   outside the program (like library calls)
```

ARITHMETIC instructions

SASM	Pascal	C or C++	NOTES
move x, y	x := y;	x = y;	x and y are ints or floats
moveb x, y	x := y;	x = y;	x and y are chars
movezx x, y	NO EQUIV	NO EQUIV	x is int, y is char (SIZE)
movesx x, y	x := y;	x = y;	x is int, y is char (SIZE)
ineg x	x := -x;	x = -x;	
iadd x, y	x := x + y;	x = x + y;	integer addition
isub x, y	x := x - y;	x = x - y;	integer subtraction
imult x, y	x := x * y;	x = x * y;	integer multiplication
idivi x, y	x := x div y;	x = x / y;	integer division (quotient)
irem x, y	x := x mod y;	x = x % y;	integer division (remainder)
fpadd x, y	x := x + y;	x = x + y;	floating point addition
fpsub x, y	x := x - y;	x = x - y;	floating point subtraction
fpmul x, y	x := x * y;	x = x * y;	floating point multiplication
fpdiv x, y	x := x / y;	x = x / y;	floating point division

NOTES: 2. cannot increase the number of operands.
3. y can be an IMMEDIATE for all except the floating point instructions

examples:

```
move  count, 0

imult   product, multiplier

iadd   sum, 1
```

NOTE: there are other instructions that implement boolean functions, but we don't cover them yet.

The move instructions must be carefully chosen to match the type of the data being moved. The operation and difference between movezx and movesx will be covered after we talk about representations.

CONDITIONAL EXECUTION
---

sometimes an instruction (or a set of instructions) should be executed, and sometimes it (they) shouldn't.

HLL -- simplest form is a go-to. (Always discouraged.)

Pascal if-then-else (a conditional go-to!)

```
if (condition) then
    statement
else
    statement;
```

C if-then-else

```
if (condition)
    statement;
else
    statement;
```

SASM 'ifs' and 'gotos'
(a better name is CONTROL INSTRUCTIONS)

SASM	effect of instruction
br label	goto label;
blz label	if SF=1 then goto label;
bgz label	if SF=0 and ZF=0 then goto label;
blez label	if SF=1 or ZF=1 then goto label;
bgez label	if SF=0 or ZF=1 then goto label;
bez label	if ZF=1 then goto label;
bnz label	if ZF=0 then goto label;
compare x, y	result of x-y sets condition codes
compareb x, y	result of x-y sets condition codes

This is different than many other modern machines. There are two CONDITION CODES that we must think about.

condition code	contents
zero flag (ZF)	ZF=1 if result is 0
sign flag (SF)	SF=1 if result is negative SF=0 if result is zero or positive

These condition codes get changed (set) according to the result of certain instructions (iadd, isub, ineg, and some logical instructions). The condition codes are used by the control instructions.

To explicitly set the condition codes, use a compare instruction.

EXAMPLE:

Pascal if-then-else:

```
if (count < 0) then
begin
    count := count + 1;
end;
```

C equivalent:

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

SASM equiv to if-then-else:

```
            compare count, 0
            blz     ifstuff
            br      end_if
    ifstuff:  iadd    count, 1
    end_if:      # next program instruction goes here
```

-- OR --

```
            compare count, 0
            bgez    end_if
            iadd    count, 1
    end_if:      # next program instruction goes here
```

WHICH ONE OF THESE IS BETTER?

NOTE: Be careful not to use RESERVED WORDS for your variable names or label names.
(some reserved words: end endif if else elseif for while repeat)

Structured loops can be built out of IF's and GOTO's
(test and branch)

EXAMPLES:
---

while loop example

Pascal:

```
while ( count > 0 ) do
begin
    a := a mod count;
    count := count - 1;
end;
```

BAD STYLE Pascal:

```
while: if  (count <= 0) then goto endwhile;
       a := a mod count;
       count := count - 1;
       goto while;
```

```
       endwhile:
```

C or C++:

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

SASM:

```
      while_loop:
             compare count, 0
             blez    end_while
             irem    a, count
             isub    count, 1
             br      while_loop
      end_while:    # next program instruction goes here
```

while loop example (compound conditional)

Pascal:

```
          while (count < limit) and (c = d) do
          begin
             /* loop's code goes here */
          end;
```

C or C++:

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

SASM:

```
       while_loop:
                compare count, limit
                bgez    end_while
                compare c, d
                bnz     end_while

                # loop's code goes here

                br while_loop
       end_while:
```

for loop example

Pascal:

```
          for i:= 3 to 8 do
          begin
              a := a + i;
          end;
```

C:

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

SASM:

```
                move i, 3
        for_loop:
                compare  i, 8
                bgz      end_for
                iadd     a, i
                iadd     i, 1
                br for_loop
        end_for:
```

COMMUNICATION WITH THE USER (I/O operations)

SASM	effect of instruction
get_ch x	read character from input, place into x
put_ch x	send character in x to output
put_i x	send integer in x to output
put_fp x	send floating point value in x to output
put_str x	send (NULL TERMINATED) string at x to output

SASM doesn't have any oddities about
testing for eoln or eof. The newline character (0ah, or '\n' in C)
is just another character to be read or written.

NOTE: There are times when you will want to 'get' something that
isn't a character (like an integer or floating point value input
by user). In SASM, you can't, since the instruction doesn't exist.
At the end of Chapter 5, you will know enough about data representation
to be able to read an integer (or floating point value) character
by character and translate it to an integer.

It is done this way because input from a keyboard are only characters.
Output to a simple display (which we are assuming) are only characters.
The C library (that we utilize) gives easy implementation of
output for other types, so you get that benefit in this language.

EXAMPLES:

```
   ; this is a code FRAGMENT, not a whole program
   .data
   msg1    db 'The integer is ', 0
```

```
int1    dd 285
newline db 0ah
msg2    db 'The second string.', 0ah, 0

.code
  put_str  msg1
  put_i    int1
  put_ch   newline
  put_str  msg2
```

```
prints:

The integer is 285
The second string.
```