EmitAdditionCode()

```
void EmitAdditionCode(string operand1,string operand2) //add operand1 to operand2
{
  if type of either operand is not integer
    process error: illegal type
  if A Register holds a temp not operand1 nor operand2 then
    deassign it
    emit code to store that temp into memory
    change the allocate entry for the temp in the symbol table to yes
  if A register holds a non-temp not operand1 nor operand2 then deassign it
  if neither operand is in A register then
    emit code to load operand2 into A register
  emit code to perform register-memory addition
  deassign all temporaries involved in the addition and free those names for reuse
  A Register = next available temporary name and change type of its symbol table entry to integer
  push the name of the result onto operandStk
}
```

EmitDivisionCode()

Division is slightly more complex because the RAMM architecture requires a double register in order to perform this operation, the A-Q pair. *Operand2* is loaded into the A register. The contents of this register is then divided by *operand1*, leaving the quotient in the A register. At this point in the compiler, you should presume that no program ever attempts to divide by zero, which is, of course, illegal and is trapped by the hardware.

```
void EmitDivisionCode(string operand1,string operand2) //divide operand2 by operand1
{
  if type of either operand is not integer
    process error: illegal type
  if A Register holds a temp not operand2 then
     deassign it
     emit code to store that temp into memory
     change the allocate entry for it in the symbol table to yes
  if A register holds a non-temp not operand2 then deassign it
  if operand2 is not in A register
    emit instruction to do a register-memory load of operand2 into the A register;
  emit code to perform a register-memory division
  deassign all temporaries involved and free those names for reuse;
  A Register = next available temporary name and change type of its symbol table entry to integer
  push name of result onto operandStk;
}
```

EmitMultiplicationCode()

Multiply also uses the A-Q pair. The result is an 8-digit number with the higher order digits in the Q register.

```
void EmitMultiplicationCode(string operand1,string operand2) //multiply operand2 by operand1
{
  if type of either operand is not integer
    process error: illegal type
  if A Register holds a temp not operand1 nor operand2 then
     deassign it
     emit code to store that temp into memory
     change the allocate entry for it in the symbol table to yes
  if A register holds a non-temp not operand2 nor operand1 then deassign it
  if neither operand is in A register then
    emit code to load operand2 into the A register;
  emit code to perform a register-memory multiplication with A Register holding the result;
  deassign all temporaries involved and free those names for reuse;
  A Register = next available temporary name and change type of its symbol table entry to integer
  push name of result onto operandStk;
}
```

EmitAndCode()

Logical operation `and` is performed between *boolean* operands. Recall that the internal representation of `false` is integer `0` (`0000`) and that the internal representation of `true` is integer `1` (`0001`). There are no bit-wise operators in the RAMM computer, so use the same code as for multiply.

```
void EmitAndCode(string operand1,string operand2) //"and" operand1 to operand2
{
  if type of either operand is not boolean
    process error: illegal type
  if A Register holds a temp not operand1 nor operand2 then
     deassign it
     emit code to store that temp into memory
     change the allocate entry for it in the symbol table to yes
  if A register holds a non-temp not operand2 nor operand1 then deassign it
  if neither operand is in A register then
    emit code to load operand2 into the A register;
  emit code to perform a register-memory multiplication with A Register holding the result;
  deassign all temporaries involved and free those names for reuse;
  A Register = next available temporary name and change type of its symbol table entry to boolean
  push name of result onto operandStk;
}
```

EmitOrCode()

Logical operation `or` is performed between *boolean* operands. Recall that the internal representation of `false` is integer `0` (`0000`) and that the internal representation of `true` is integer `1` (`0001`). There are no bit-wise operators in the RAMM computer, so start by using the same code as for addition.

Labels which stage1 generates will have the form *`'Ln'`*, where *n* is a non-negative integer. Initially, *n* is `-1` and is incremented by one with each call for a new label, so that the first label is `'L0'`. Branch instructions are issued at this stage to skip over code that is to be conditionally executed.

```
void EmitOrCode(string operand1,string operand2) //"or" operand1 to operand2
{
  if type of either operand is not boolean
    process error: illegal type
  if A Register holds a temp not operand1 nor operand2 then
     deassign it
     emit code to store that temp into memory
     change the allocate entry for it in the symbol table to yes
  if A register holds a non-temp not operand2 nor operand1 then deassign it
  if neither operand is in A register then
    emit code to load operand2 into the A register;
  emit code to perform a register-memory addition with A Register holding the result;
  emit code to perform an AZJ to the next available Ln +1
  emit code to label the next instruction with that label and do a register-memory load TRUE
  insert TRUE in symbol table with value 1 and external name true
  deassign all temporaries involved and free those names for reuse;
  A Register = next available temporary name and change type of its symbol table entry to boolean
  push name of result onto operandStk;
}
```

EmitEqualsCode()

The relational operation '=' is performed between any two operands of the same type.

```
void EmitEqualsCode(string operand1,string operand2) //test whether operand2 equals operand1
{
  if types of operands are not the same
    process error: incompatible types
  if A Register holds a temp not operand1 nor operand2 then
     deassign it
     emit code to store that temp into memory
     change the allocate entry for it in the symbol table to yes
  if A register holds a non-temp not operand2 nor operand1 then deassign it
  if neither operand is in A register then
    emit code to load operand2 into the A register;
  emit code to perform a register-memory subtraction with A Register holding the result;
  emit code to perform an AZJ to the next available Ln
  emit code to do a register-memory load FALS
  insert FALS in symbol table with value 0 and external name false
  emit code to perform a UNJ to the acquired label Ln +1
  emit code to label the next instruction with the acquired label Ln
          and do a register-memory load TRUE
  insert TRUE in symbol table with value 1 and external name true
  deassign all temporaries involved and free those names for reuse;
  A Register = next available temporary name and change type of its symbol table entry to boolean
  push name of result onto operandStk;
}
```

EmitAssignCode()

```
void EmitAssignCode(string operand1,string operand2) //assign the value of operand1 to operand2
{
  if types of operands are not the same
    process error: incompatible types
  if storage mode of operand2 is not VARIABLE
    process error:  symbol on left-hand side of assignment must have a storage mode of VARIABLE
  if operand1 = operand2 return;
  if operand1 is not in A register then
    emit code to load operand1 into the A register;
  emit code to store the contents of that register into the memory location pointed to by
    operand2
  deassign operand1;
  if operand1 is a temp then free its name for reuse;
  //operand2 can never be a temporary since it is to the left of ':='
}
```

FreeTemp(), GetTemp()

All code emitting procedures just given refer to "books" which must be "adjusted" to reflect changing uses of temporary locations. When a new temporary is needed, one must be created. When an old temporary is no longer needed, it must be discarded. There are two utility routines for this called `GetTemp()` and `FreeTemp()`, respectively. `GetTemp()` returns the name of the next temporary, and if necessary, will force allocation of a full-word to correspond to that name. `FreeTemp()` releases the name of the last temporary created by a call to `GetTemp()`; however, the storage (if any) allocated by the earlier call to `GetTemp()` remains allocated after the call to `FreeTemp()`. Once storage for a temporary has been allocated, it may not be de-allocated. Note that both `FreeTemp()` and `GetTemp()` refer to global quantities, `currentTempNo` and `maxTempNo`, which should be declared in the definition of the parser routine, and initialized to `−1` to reflect the fact that initially no temporaries are allocated (`T0` will be the first temporary name used).

```
void FreeTemp()
{
  currentTempNo--;
  if (currentTempNo < -1)
      process error: compiler error, currentTempNo should be ≥ -1;
}

string GetTemp()
{
  string temp;
  currentTempNo++;
  temp =  "T" + currentTempNo;
  if (currentTempNo > maxTempNo)
      Insert(temp, UNKNOWN, VARIABLE, "", NO, 1);
      maxTempNo++;
  return temp;
}
```

AssignRegister()

Recall that all temporary quantities are useful or "*alive*" only from the time they are created until their first use as an operand in a call to `Code`. Consequently, when each of the code emitting subroutines of `Code` has finished its references to temporary quantities, it deassigns them so that:

1) The register (if any) occupied by that temporary is deassigned
2) The names given to these temporary quantities may be reused by new temporary quantities

So, for example, in the latter part of `EmitAdditionCode()`, there is an instruction that deassigns all temporary operands.

For certain operations, the result of the operation is placed in a register, so that whatever quantity, if any, formerly resided in this register no longer does. Hence, it is necessary to first deassign the old values so that the old association is removed from the "books" and second to assign the name of the result to the new register (usually A). Because this result register will normally have a temporary name, it is important to first "release" the temporary names of the operands of the call to `Code` prior to assigning this register a temporary name in order to minimize the number of temporaries actually needed. The lines at the end of `EmitAdditionCode()` modify the book entries for a register and its assigned operand as desired.

In stage1, no main memory allocation is made when a temporary quantity is named. The temporary normally "lives" in a register and is removed to main memory only when necessary. The only temporary in Q to be removed to main memory is from the `mod` operation.