Generic Operand Support

Problem Definition:

Currently there are multiple instruction patterns for various vector operations which meagerly differ in vector lengths of input/output operands i.e. they have same selector predication logic, matching pattern, data flow attribution (effect clause) over operands and same number of operands.

This multiplicity in patterns for same operation meagerly differing in vector operands translates to generation of lots of extra functional and conditional logic which effectivity increases the libjvm.so size.

Collapsing such multiple patterns to one pattern should not only help in size reduction of generated object files but also help in better maintenance and cleanup of AD files.

Vector Operand:

- Currently there are various vector operands like *vecS, vecD, vecX, vecY, vecZ* and each operand also has its corresponding legacy variant i.e. *legVecS, legVecD, legVecX, legVecY and legVecZ.*
- A non-legacy and its corresponding legacy operand have same ideal type (*Type::[VECX , VECY, VECZ]* i.e. type and vector length) but differ in register classes associated with them.
- There are also chain instruction patters which connects legacy DEF to non-legacy USE and vice versa e.g MoveLeg2VecS , MoveLeg2VecD, MoveLeg2VecX, MoveLeg2VecY, MoveLeg2VecZ etc..

Register Classes:

- A register class is a collection of registers, each of which has the same vector length.
- These register classes get translated to register masks which are associated with operands.
- Register mask is bit vector where each bit corresponds to a 32-bit double word.
- ***Register mask associated with the operands are used by register allocator***
  - ***to limit the allocation set for a def operand,***
  - **to compute the spill slot size in case a definition needs to be spilled to memory on account of greater number of intersecting live ranges than available allocatable register set.**
- A legacy register class encapsulates registers from lower register bank where as a non-legacy register class encapsulates registers from both higher and lower register bank.
- Dynamic register classes associated with vector operands select b/w the non-legacy OR legacy register class based on the feature set of the target during VM initialization.
- There are various instruction patterns which uses mixed feature instructions i.e. some of the assembly instructions in the sequence are strictly supported over lower feature target, in such cases usage of a legacy register class operand becomes necessary.

Generic Operands:

- To facilitate the collapsing of multiple instruction patterns differing meagerly in vector length of their operands two new generic operands have been created ***vecG*** and ***legVecG***.
- Register class associated with the generic operands corresponds to the set of maximum size register set supported for the target.
- New register class convertor nodes/ chain instructions (***MoveVecG2LegVecG*** and ***MoveLegVecG2VecG***) are selected in order to connect a non-legacy def to a legacy use and vice-versa.
- To minimize the effect of new generic operands over downstream passes a new post selection stage is added which replaces the generic operands with their concrete vector length and register mask operand.

Pattern collapsing using generic operands

```
instruct vadd4I_reg(vecX dst, vecX src1, vecX src2) %{
  predicate(UseAVX > 0 && n->as_Vector()->length() == 4);
  match(Set dst (AddVI src1 src2));
  format %{ "vpaddd  $dst,$src1,$src2\t! add packed4I" %}
  ins_encode %{
    int vector_len = 0;
    __ vpaddd($dst$$XMMRegister, $src1$$XMMRegister, $src2$$XMMRegister, vector_len);
  %}
  ins_pipe( pipe_slow );
%}

instruct vadd8I_reg(vecY dst, vecY src1, vecY src2) %{
  predicate(UseAVX > 1 && n->as_Vector()->length() == 8);
  match(Set dst (AddVI src1 src2));
  format %{ "vpaddd  $dst,$src1,$src2\t! add packed8I" %}
  ins_encode %{
    int vector_len = 1;
    __ vpaddd($dst$$XMMRegister, $src1$$XMMRegister, $src2$$XMMRegister, vector_len);
  %}
  ins_pipe( pipe_slow );
%}

instruct vadd16I_reg(vecZ dst, vecZ src1, vecZ src2) %{
  predicate(UseAVX > 2 && n->as_Vector()->length() == 16);
  match(Set dst (AddVI src1 src2));
  format %{ "vpaddd  $dst,$src1,$src2\t! add packed16I" %}
  ins_encode %{
    int vector_len = 2;
    __ vpaddd($dst$$XMMRegister, $src1$$XMMRegister, $src2$$XMMRegister, vector_len);
  %}
  ins_pipe( pipe_slow );
%}
```

```
instruct vaddI_reg(vecG dst, vecG src1, vecG src2) %{
  predicate(UseAVX > 0 && n->as_Vector()->length() >= 2 && n->as_Vector()->length() <= 16);
  match(Set dst (AddVI src1 src2));
  format %{ "vpaddd  $dst,$src1,$src2\t! add packed2I" %}
  ins_encode %{
    const TypeVect * vt = this->bottom_type()->is_vect();
    int vector_len = GetVectorLengthEnc(vt);
    __ vpaddd($dst$$XMMRegister, $src1$$XMMRegister, $src2$$XMMRegister, vector_len);
  %}
  ins_pipe( pipe_slow );
%}
```

DFA for LoadVector– New..

```
void  State::_sub_Op_LoadVector(const Node *n){
    if( STATE__VALID_CHILD(_kids[0], MEMORY) ) {
      unsigned int c = _kids[0]->_cost[MEMORY];
        DFA_PRODUCTION__SET_VALID(_LOADVECTOR_MEMORY_, _LoadVector_memory__rule, c)
    }
    if( STATE__VALID_CHILD(_kids[0], MEMORY) &&
        (
#line 3062 "/home/intel/sandboxes/jdk-release2/jdk/src/hotspot/cpu/x86/x86.ad"
n->as_LoadVector()->memory_size() >= 4
#line 16812 "dfa_x86.cpp"
) ) {
      unsigned int c = _kids[0]->_cost[MEMORY]+125;
        DFA_PRODUCTION__SET_VALID(VECG, loadV_rule, c)
        DFA_PRODUCTION__SET_VALID(LEGVECG, MoveVecG2LegVecG_rule, c+100)
    }
}
void  State::_sub_Op_StoreVector(const Node *n){
    if( STATE__VALID_CHILD(_kids[0], MEMORY) && STATE__VALID_CHILD(_kids[1], VECG) &&
        (
#line 3095 "/home/intel/sandboxes/jdk-release2/jdk/src/hotspot/cpu/x86/x86.ad"
n->as_StoreVector()->memory_size() >= 4
#line 16824 "dfa_x86.cpp"
) ) {
      unsigned int c = _kids[0]->_cost[MEMORY]+_kids[1]->_cost[VECG]+145;
        DFA_PRODUCTION__SET_VALID(UNIVERSE, storeV_rule, c)
    }
}
```

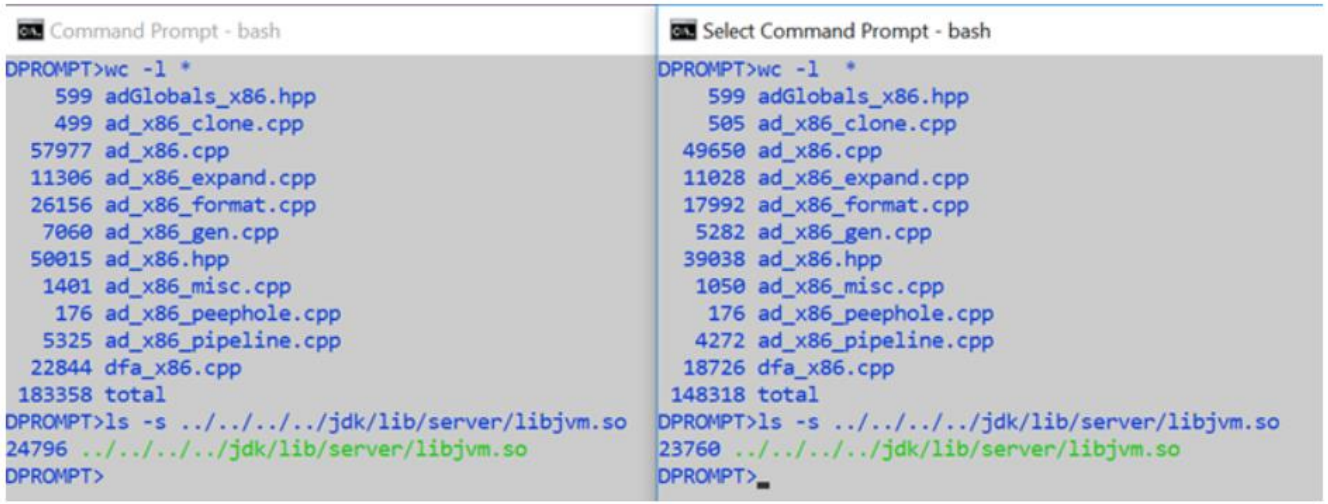
Post-selection stage

- In order to keep the downstream passes transparent to generic operands creation during selection, a new **post selection pass has been introduced.**
- This pass is currently invoked only for X86 target and its primarily responsible for replacement of generic operand with their corresponding concreter vector length operands using the ideal type associated with machine nodes.
- Pass gets kicked in only when machine graph has a vector node.

Results (1+MB size reduction in libjvm.so)

Using generic operands majority of vector instruction patterns were collapsed to one pattern

Number of vector instruction patterns in mainline (vec[XYZSD] + legVec[ZXYSD] : **510**
Reduced vector instruction patterns (vecG + legVecG) : **222**



There is a reduction of around 35K LOC generated by ADLC which translates to 1 MB (approx) size reduction of libjvm.so.

Conditions under which two instruction patterns cannot be merged:

- Match rules are different.
- Different number of instruction parameters.
- Predicate checks based on ideal nodes differ other than vector length check.
- Temp operands have different operand types associated with them.

Implementation Details

- In ideal graph the data flow information between the nodes is captured in the form of input and output edges originating from a node. In machine graph each machine node has machine operands (def and uses*). **Operand index mapping between the machine operand and graph edges has been used to establish the data flow between different machine nodes***.

Changes spans primarily in three areas

1. ADLC:
   - Earlier generated code used to emit a static output register mask for vector machine nodes, this is changed to pull the register mask from output operand since post selection stage replaces the originally created generic operands with concrete vector length operand.
   - Muti-match rule extension for operand, to emit transitions to VECG and LEGVECG states for each of concrete vector operand (vecS, vecD, vecX, vecY, vecZ).
   - Emit appropriate vector type instead of register type for machine nodes whose bottom_type is based on the type of constant operand.
   - Few integration changes for introduction of vecG operand to routines processing operand clauses to generate code.

2) AD file patterns:

- New operand definitions for vecG and legVecG.
- Register class converting chain instructions MoveVecG2LegVecG and MoveLegVecG2VecG.
- New generic register class which captures maximal size register set supported over target.
- Collapsing multiple instruction patterns for a given vector operation differing in vector length of operands to one pattern based on generic operand.

3) Post-selection stage:

- Remove generic operands from machine graph.
  i. Collect machine nodes having generic operand.
  ii. Replace the generic output operand with concrete vector length operands by looking at the type of machine node.
  iii. Perform an iterative forward data flow propagation from def operand to its use operand until no-more generic operand exists in the graph.
  iv. Replace generic operand chain nodes (MoveVecG2LegVecG and MoveLegVecG2VecG) with their concrete operand counter parts.
  v. Lazy operand resolution is performed for generic operands which have both TEMP and DEF data flow attribute attached to it.
- Optimize away register class convertor chain nodes from legacy to non-legacy operand.

Testing

- Patch has been regressed through JTREG and JMH suite.
- Compilation time statistics are being measured for SPECjvm2008 and SPECjbb.

SPECjvm startup benchmark startup.compress (Platform : SKX)

```
Accumulated compiler times
----------------------------------------------------------
  Total compilation time   :  35.076 s
    Standard compilation   :  31.563 s, Average : 0.006 s
    Bailed out compilation :   0.000 s, Average : -nan s
    On stack replacement   :   3.512 s, Average : 0.018 s
    Invalidated            :   0.000 s, Average : -nan s

    C2 Compile Time:       34.940 s
       Parse:                 3.188 s
       Optimize:             10.079 s
         Escape Analysis:       0.211 s
         GVN 1:                 0.641 s
         Incremental Inline:    0.000 s
         Renumber Live:         0.050 s
         IdealLoop:             7.313 s
         IdealLoop Verify:      0.000 s
         Cond Const Prop:       0.134 s
         GVN 2:                 0.115 s
         Macro Expand:          0.286 s
         Barrier Expand:        0.000 s
         Graph Reshape:         0.233 s
         Other:                 1.096 s
       Matcher:                   1.718 s
       Post Selection Cleanup:    0.031 s
       Scheduler:                 2.550 s
       Regalloc:             14.922 s
       Block Ordering:        0.178 s
       Peephole:              0.018 s
       Code Emission:         1.744 s
       Code Installation:     0.417 s
       Other:                 0.125 s

  Total compiled methods    :     5272 methods
    Standard compilation    :     5075 methods
    On stack replacement    :      197 methods
  Total compiled bytecodes  :  1476034 bytes
    Standard compilation    :  1347736 bytes
    On stack replacement    :   128298 bytes
  Average compilation speed :    42081 bytes/s
```

Next steps..

- Code Review: http://cr.openjdk.java.net/~jbhateja/genericVectorOperands/webrev.00/

- Vector API branch has around 1200+ vector instruction patterns which can be collapsed using generic operands.