**Problem Definition:**

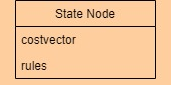
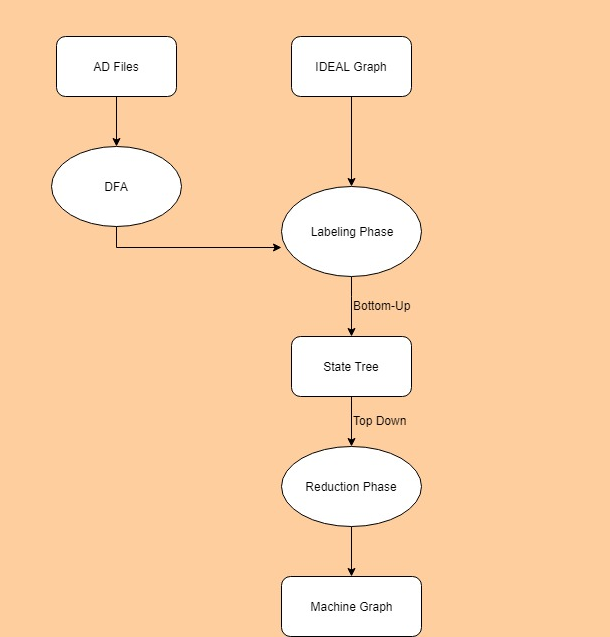
Currently there are multiple instruction patterns for various vector operations which meagerly differ in vector lengths of input/output operands i.e. they have same selector predication logic, matching pattern, data flow attribution (effect clause) over operands and same number of operands. Different clauses of instruction pattern are compiled by ADLC to generate code catering to different stages in compilation [Fig 1]. 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.

**Instruction selection:**

Selection stage uses dynamic programming-based BURS (bottom-up-rewriting system) style algorithm for instruction selection. It comprises of two phases, a labelling phase and a reduction phase.

During labelling phase, a bottom up traversal is performed over ideal expression tree to construct a state tree by consulting a DFA (deterministic finite automata generate by ADL compile) where each state node captures a list of valid transitions from current state to its parent state (determined by def operand type of its AD instruction pattern). Valid transitions from current state are selected based on the state of its child nodes. A state node also captures the cost vector corresponding to the valid transitions and associated productions. At the end of labelling phase, we have state tree which contains all the valid productions/match rules which cover the ideal expression tree.

Reduction phase gets state tree as its input and it perform a top down traversal over it. Traversal begins by selecting the minimum cost production/match rule from the root of the state tree. It then generates a machine node corresponding to the chosen match rule and creates its output operand. From here onwards, it does a recursive downward traversal over state tree and based on the input operands type needed for newly created machine node it picks the minimum cost production/match rule from the child state nodes.

At the end of the reduction phase we get a minimum cost machine node tree corresponding to given ideal expression tree. Following flow diagram describes the selection procedure.

**Fig 1: Instruction Selection Flow Graph**

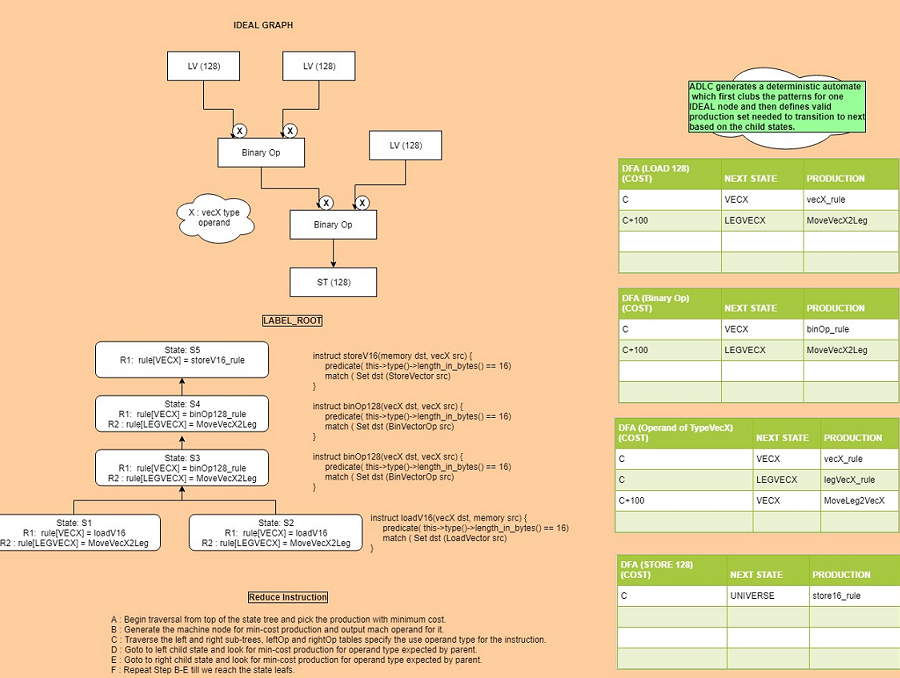
**ADLC generated DFA:**

Architecture definition language compiler (ADLC) reads various instruction patterns and operand definitions defined in the .ad files and generates DFA which guides the labelling phase of selection. It first groups the instruction patterns based on the ideal nodes present in their match clauses and then emits a state creation routine corresponding to each ideal node capturing logic to select valid transition based on the input child state/input operands for each instruction pattern in the group.

There is a direct relation b/w size of the DFA and different instruction patterns of an operation, with multiple patterns for a given ideal node whose operands meagerly differ in vector length, the control logic to select valid transitions will have larger nesting depth to cover a case for each input operand of different vector length.

In addition to the selection patterns for different ideal nodes (operations) there are chain instruction patterns which translates into a glue node which are used to connect defining and user machine nodes having operands of same type but different register class. These chain rules facilitate the selection process in absences of a direct transition from child state to parent state i.e. from def state to user state.

Thus, for each instruction pattern, in addition to emitting transition to its defining operand extra chain transitions from its output operand to equivalent legacy register class operand are emitted.



**Fig 2: Labelling and reduction phases of instruction selection.**

**Vector Operands and Register classes:**

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. These ideal types are propagated from Ideal node to its corresponding machine node.

A register class is a collection of registers each of which has 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.

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 at the startup. 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.

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.

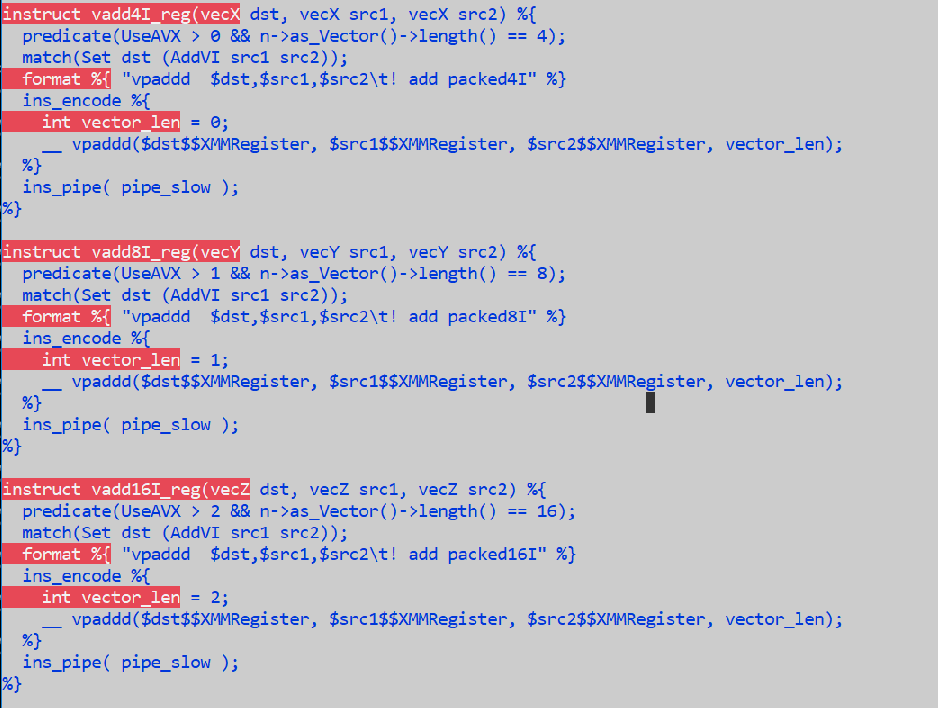
Note: AVX and AVX2 uses VEX encoding to address 16 vector registers (lower register bank) of 128/256 bits i.e. vector registers are encoded using 4 bits VEX.VVVV ranging from 0-15. AVX512 added additional 16 registers (higher register bank) and uses a new encoding EVEX which set aside an additional bit in the header apart from 4 bits of VEX.VVVV to access 32 vector registers from both high and lower register bank ranging from 0-31. Maximum vector length supported on AVX512 target is 512 bits.

**Generic operands:**

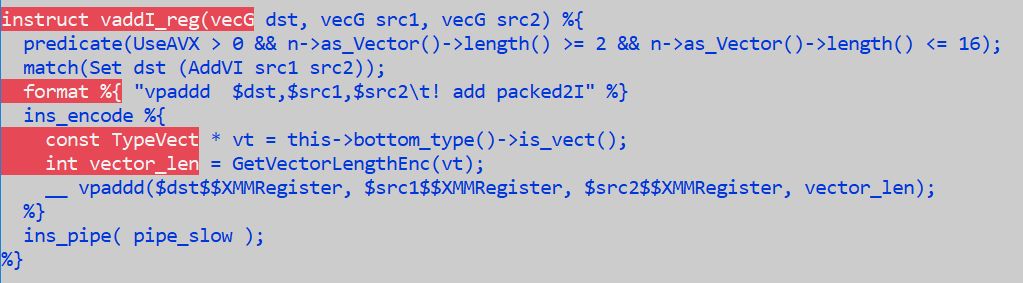
To facilitate the collapsing of multiple instruction patterns differing meagerly in vector length of their operands two new generic operands have been created vecG and legVecG. Register class associated with the generic operands corresponds to the maximal size register set supported for the target. Consider instruction patterns for vector ADD operation [Fig 3] for different vector lengths and its collapsed instruction pattern [Fig 4] based on the generic operand.

Post-selection machine graph comprises of machine nodes having generic operands. New register class convertor nodes (MoveVecG2LegVecG and MoveLegVecG2VecG) are selected in order to connect a non-legacy def to a legacy use and vice-versa.

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.

3

**Fig 3: Vector add operation for different vector lengths.**



**Fig 4: Vector add operation instruction pattern using generic operand.**

**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 also has machine operands (def and uses). Operand index mapping between the machine operand the graph edges has used to establish the data flow between different machine nodes.

Changes spans primarily in three areas

1. ADLC:
   1. 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.
   2. Muti-match rule extension for operand clause, to emit transitions to VECG and LEGVECG states for each of concrete vector operand (vecS, vecD, vecX, vecY, vecZ).
   3. Emit appropriate vector type instead of register type for machine nodes whose bottom\_type is based on the type of constant operand.
   4. Few integration changes for introduction of vecG operand to routines processing operands to generate code.
2. AD file patterns:
   1. New operand definitions for vecG and legVecG.
   2. Register class converting chain instructions MoveVecG2LegVecG and MoveLegVecG2VecG.
   3. New generic register class which captures maximal widest register set supported over target.
   4. 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:
   1. Remove generic operands from machine graph.
      1. Collects all the machine nodes which have any generic operand.
      2. Replace the generic output operand with concrete vector length operands by looking at the type of machine node.
      3. Perform an iterative forward data flow propagation from def operand to its use operand until no-more generic operand exists in the graph.
      4. Lazy operand resolution is performed for generic operands which have both TEMP and DEF data flow attribute attached to it.
   2. Fixup stage to insert appropriate register class conversion chain nodes between non-legacy def and legacy use.
      1. For shared vector nodes like LoadVector, node is shared b/w operations, in such a case for second use value is read from the vector register, state node created for this use during labelling phase is an operand state node (DFA::\_sub\_Op\_Vec[XYZ]()) which does not have additional transition to legacy operands (MoveVec[XYZDS]2Leg].

**PoC Results:**

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**

Machine generated alternative text:
Command Prompt - bash 
DPROMPT>wc -1 * 
Select Command Prompt bash 
DPROMPT>wc -1 * 
599 
499 
57977 
113e6 
26156 
7ese 
seei5 
14e1 
176 
5325 
22844 
183358 
adG10ba1 s _ x86. hpp 
ad_x86_c10ne. cpp 
ad_x86. cpp 
ad_x86_expand. cpp 
ad_x86_format . cpp 
ad_x86_gen. cpp 
ad_x86. hpp 
x86_misc . cpp 
ad 
ad_x86_peeph01e. cpp 
pel i ne. cpp 
dfa x86. cpp 
total 
599 
ses 
4965e 
Ile28 
17992 
5282 
39e38 
lese 
176 
4272 
18726 
148318 
adG10ba1 s _ x86. hpp 
ad_x86_c10ne. cpp 
ad_x86. cpp 
ad_x86_expand. cpp 
ad_x86_format . cpp 
ad_x86_gen. cpp 
ad_x86. hpp 
x86_misc . cpp 
ad 
ad_x86_peeph01e. cpp 
pel i ne. cpp 
dfa x86. cpp 
total 
DPROMPT>1s -s 
24796 
DPROMPT> 
. ./ / .. / . ./jdk/lib/server/libjvm 
. ./jdk/lib/server/libjvm. so 
. so 
DPROMPT>1s -s 
2376e 
DPROMPT> 
• • •/ • . / . ./jdk/lib/server/libjvm. so 
./jdk/lib/server/libjvm. so 

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