Look-Ahead SLP: Auto-vectorization in the presence of commutative operations

Vasileios Porpodas[1], Rodrigo C. O. Rocha[2] and Luís F. W. Góes[3]

Intel Santa Clara, USA[1]
University of Edinburgh, UK[2]
PUC Minas, Brazil[3]

CGO 2018

Legal Disclaimer & Optimization Notice

SLP Straight-Line Code Vectorizer

- Superword Level Parallelism [Larsen PLDI'00]

SLP Straight-Line Code Vectorizer

- Superword Level Parallelism [Larsen PLDI'00]
- GCC and LLVM implementations are based on Bottom-Up SLP

SLP Straight-Line Code Vectorizer

- Superword Level Parallelism [Larsen PLDI'00]
- GCC and LLVM implementations are based on Bottom-Up SLP

A[i+0] = B[i+0]
A[i+1] = B[i+1]

SLP Straight-Line Code Vectorizer

- Superword Level Parallelism [Larsen PLDI'00]
- GCC and LLVM implementations are based on Bottom-Up SLP

A[i+0] = B[i+0]
A[i+1] = B[i+1]

SLP →

A[i+1:i+0] = B[i+1:i+0]

SLP Straight-Line Code Vectorizer

- Superword Level Parallelism [Larsen PLDI'00]
- GCC and LLVM implementations are based on Bottom-Up SLP

A[i+0] = B[i+0]
A[i+1] = B[i+1]

SLP →

A[i+1:i+0] = B[i+1:i+0]

- SLP and loop-vectorizer complement each other:

SLP Straight-Line Code Vectorizer

- Superword Level Parallelism [Larsen PLDI'00]
- GCC and LLVM implementations are based on Bottom-Up SLP

A[i+0] = B[i+0]
A[i+1] = B[i+1]

SLP

A[i+1:i+0] = B[i+1:i+0]

- SLP and loop-vectorizer complement each other:
  - Unroll loop and vectorize with SLP
  - Even if loop-vectorizer fails, SLP could partly succeed

SLP Straight-Line Code Vectorizer

- Superword Level Parallelism [Larsen PLDI'00]
- GCC and LLVM implementations are based on Bottom-Up SLP

A[i+0] = B[i+0]
A[i+1] = B[i+1]

SLP

A[i+1:i+0] = B[i+1:i+0]

- SLP and loop-vectorizer complement each other:
  - Unroll loop and vectorize with SLP
  - Even if loop-vectorizer fails, SLP could partly succeed
- It is missing features present in the Loop vectorizer (e.g., Interleaved Loads, Predication)

SLP Straight-Line Code Vectorizer

- Superword Level Parallelism [Larsen PLDI'00]
- GCC and LLVM implementations are based on Bottom-Up SLP

```
A[i+0] = B[i+0]
A[i+1] = B[i+1]
```


```
A[i+1:i+0] = B[i+1:i+0]
```

- SLP and loop-vectorizer complement each other:
  - Unroll loop and vectorize with SLP
  - Even if loop-vectorizer fails, SLP could partly succeed
- It is missing features present in the Loop vectorizer (e.g., Interleaved Loads, Predication)
  - Usually run SLP after the Loop Vectorizer

Why commutative operations matter ?

- Operands can be reordered
- It can change the shape of the DAG
- Affects SLP ability to vectorize

Why commutative operations matter ?

- Operands can be reordered
- It can change the shape of the DAG
- Affects SLP ability to vectorize
- SLP is effective when the immediate operands differ

Why commutative operations matter ?

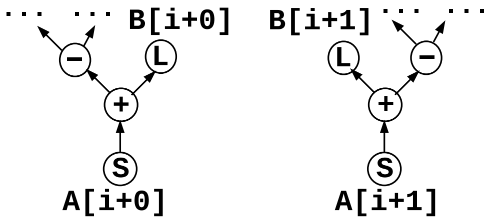
- Operands can be reordered
- It can change the shape of the DAG
- Affects SLP ability to vectorize
- SLP is effective when the immediate operands differ

```
sub1 = ... - ...;
sub2 = ... - ...;
A[i+0] = sub1   + B[i+0];
A[i+1] = B[i+1] + sub2;
```

Why commutative operations matter ?

- Operands can be reordered
- It can change the shape of the DAG
- Affects SLP ability to vectorize
- SLP is effective when the immediate operands differ

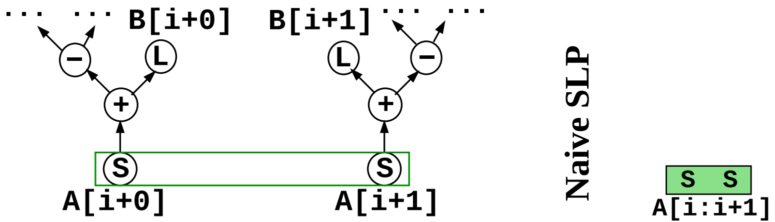
```
sub1 = ... - ...;
sub2 = ... - ...;
A[i+0] = sub1   + B[i+0];
A[i+1] = B[i+1] + sub2;
```



Why commutative operations matter ?

- Operands can be reordered
- It can change the shape of the DAG
- Affects SLP ability to vectorize
- SLP is effective when the immediate operands differ

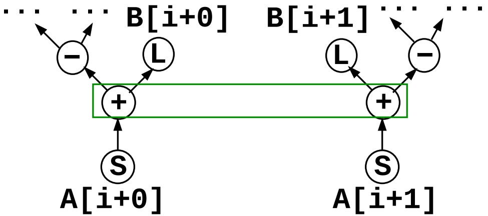
```
sub1 = ... - ...;
sub2 = ... - ...;
A[i+0] = sub1   + B[i+0];
A[i+1] = B[i+1] + sub2;
```



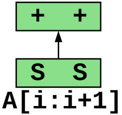
Why commutative operations matter ?

- Operands can be reordered
- It can change the shape of the DAG
- Affects SLP ability to vectorize
- SLP is effective when the immediate operands differ

```
sub1 = ... - ...;
sub2 = ... - ...;
A[i+0] = sub1   + B[i+0];
A[i+1] = B[i+1] + sub2;
```



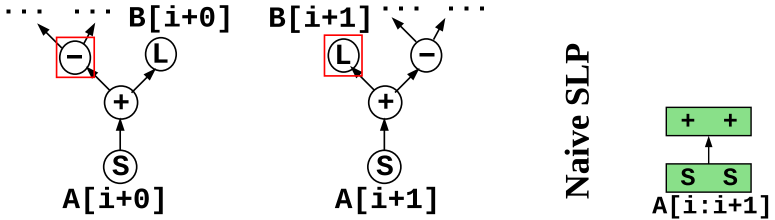
Naive SLP



Why commutative operations matter ?

- Operands can be reordered
- It can change the shape of the DAG
- Affects SLP ability to vectorize
- SLP is effective when the immediate operands differ

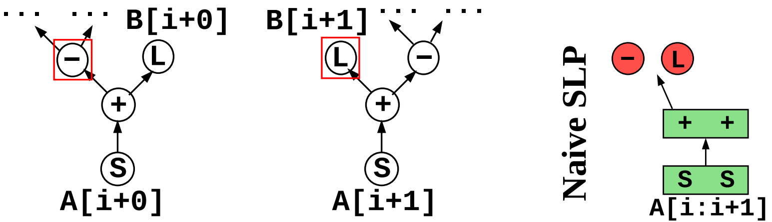
```
sub1 = ... - ...;
sub2 = ... - ...;
A[i+0] = sub1   + B[i+0];
A[i+1] = B[i+1] + sub2;
```



Why commutative operations matter ?

- Operands can be reordered
- It can change the shape of the DAG
- Affects SLP ability to vectorize
- SLP is effective when the immediate operands differ

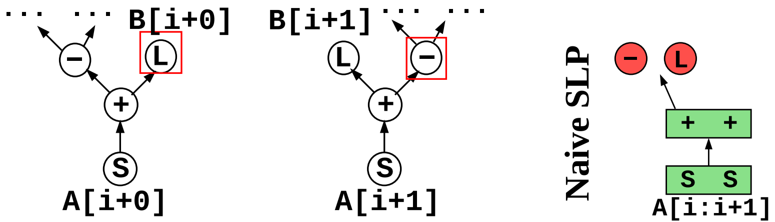
```
sub1 = ... − ...;
sub2 = ... − ...;
A[i+0] = sub1   + B[i+0];
A[i+1] = B[i+1] + sub2;
```



Why commutative operations matter ?

- Operands can be reordered
- It can change the shape of the DAG
- Affects SLP ability to vectorize
- SLP is effective when the immediate operands differ

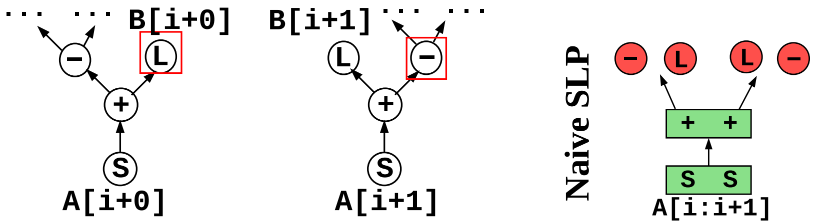
```
sub1 = ... - ...;
sub2 = ... - ...;
A[i+0] = sub1   + B[i+0];
A[i+1] = B[i+1] + sub2;
```



Why commutative operations matter ?

- Operands can be reordered
- It can change the shape of the DAG
- Affects SLP ability to vectorize
- SLP is effective when the immediate operands differ

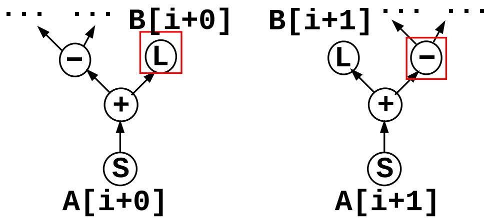
```
sub1 = ... - ...;
sub2 = ... - ...;
A[i+0] = sub1   + B[i+0];
A[i+1] = B[i+1] + sub2;
```



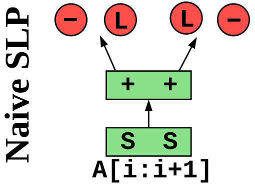
Why commutative operations matter ?

- Operands can be reordered
- It can change the shape of the DAG
- Affects SLP ability to vectorize
- SLP is effective when the immediate operands differ

```
sub1 = ... - ...;
sub2 = ... - ...;
A[i+0] = sub1   + B[i+0];
A[i+1] = B[i+1] + sub2;
```



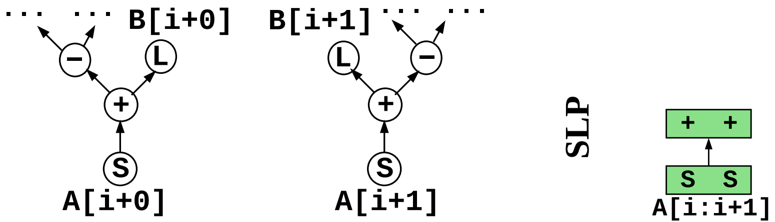
Vectorization would STOP without reordering!



Why commutative operations matter ?

- Operands can be reordered
- It can change the shape of the DAG
- Affects SLP ability to vectorize
- SLP is effective when the immediate operands differ

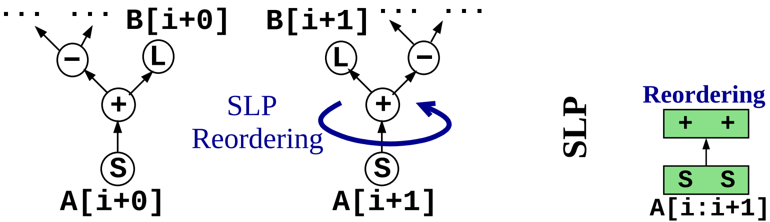
```
sub1 = ... - ...;
sub2 = ... - ...;
A[i+0] = sub1   + B[i+0];
A[i+1] = B[i+1] + sub2;
```



Why commutative operations matter ?

- Operands can be reordered
- It can change the shape of the DAG
- Affects SLP ability to vectorize
- SLP is effective when the immediate operands differ

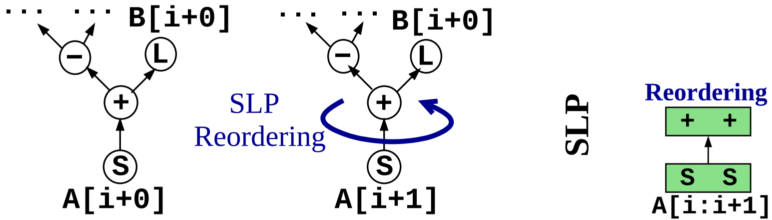
```
sub1 = ... - ...;
sub2 = ... - ...;
A[i+0] = sub1   + B[i+0];
A[i+1] = B[i+1] + sub2;
```



Why commutative operations matter ?

- Operands can be reordered
- It can change the shape of the DAG
- Affects SLP ability to vectorize
- SLP is effective when the immediate operands differ

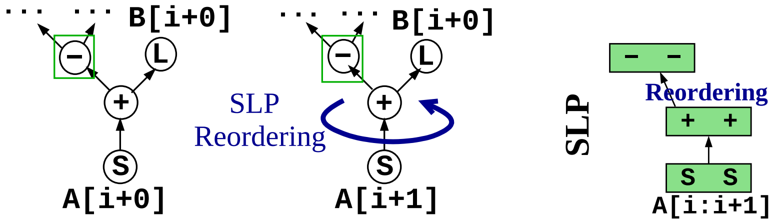
```
sub1 = ... - ...;
sub2 = ... - ...;
A[i+0] = sub1   + B[i+0];
A[i+1] = sub2   + B[i+1];
```



Why commutative operations matter ?

- Operands can be reordered
- It can change the shape of the DAG
- Affects SLP ability to vectorize
- SLP is effective when the immediate operands differ

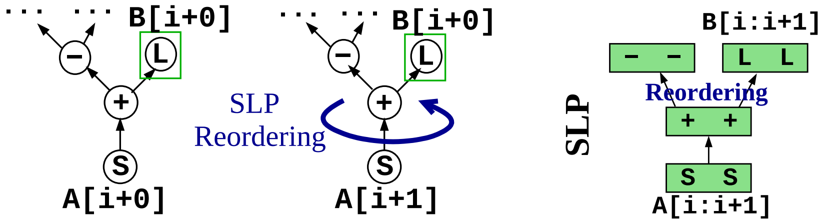
```
sub1 = ... - ...;
sub2 = ... - ...;
A[i+0] = sub1   + B[i+0];
A[i+1] = sub2   + B[i+1];
```



Why commutative operations matter ?

- Operands can be reordered
- It can change the shape of the DAG
- Affects SLP ability to vectorize
- SLP is effective when the immediate operands differ

```
sub1 = ... - ...;
sub2 = ... - ...;
A[i+0] = sub1   + B[i+0];
A[i+1] = sub2   + B[i+1];
```

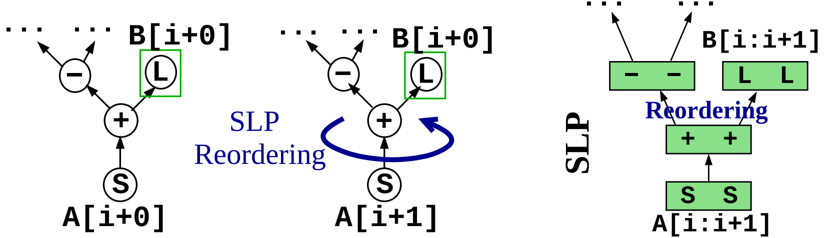


Why commutative operations matter ?

- Operands can be reordered
- It can change the shape of the DAG
- Affects SLP ability to vectorize
- SLP is effective when the immediate operands differ

```
sub1 = ... - ...;
sub2 = ... - ...;
A[i+0] = sub1   + B[i+0];
A[i+1] = sub2   + B[i+1];
```

Vectorization continues !



SLP not effective in more complex cases

- SLP reordering not effective for:

SLP not effective in more complex cases

- SLP reordering not effective for:
  1. Load address mismatch further up the graph

SLP not effective in more complex cases

- SLP reordering not effective for:
  1. Load address mismatch further up the graph
  2. Opcode mismatch further up the graph

SLP not effective in more complex cases

- SLP reordering not effective for:
  1. Load address mismatch further up the graph
  2. Opcode mismatch further up the graph
  3. Reodering across chains of commutative operations

SLP not effective in more complex cases

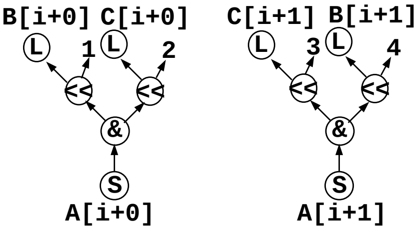
- SLP reordering not effective for:
  1. Load address mismatch further up the graph
  2. Opcode mismatch further up the graph
  3. Reodering across chains of commutative operations
- Look-Ahead SLP (LSLP) provides a solution to all three.

1/3 Load address mismatch

```
long A[],B[],C[];
A[i+0]=(B[i+0]<<1)&(C[i+0]<<2);
A[i+1]=(C[i+1]<<3)&(B[i+1]<<4);
```

1/3 Load address mismatch

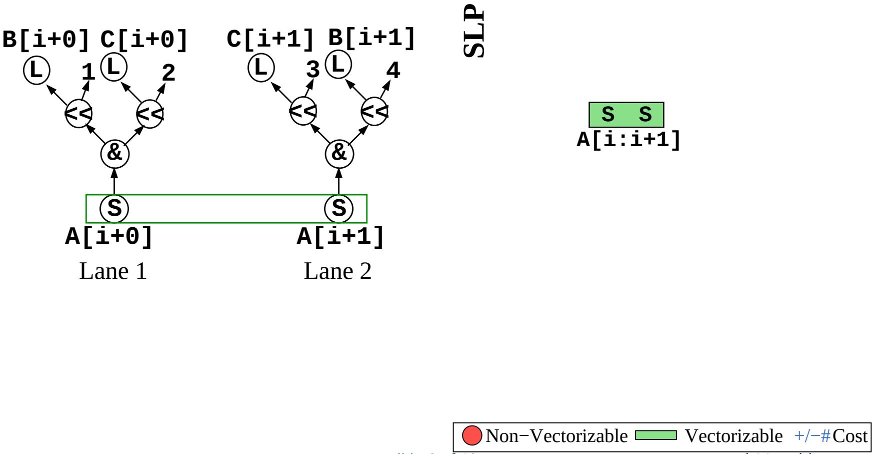
```
long A[],B[],C[];
A[i+0]=(B[i+0]<<1)&(C[i+0]<<2);
A[i+1]=(C[i+1]<<3)&(B[i+1]<<4);
```



Lane 1 Lane 2

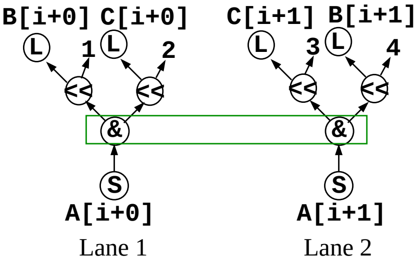
1/3 Load address mismatch

```
long A[],B[],C[];
A[i+0]=(B[i+0]<<1)&(C[i+0]<<2);
A[i+1]=(C[i+1]<<3)&(B[i+1]<<4);
```

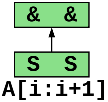


1/3 Load address mismatch

```
long A[],B[],C[];
A[i+0]=(B[i+0]<<1)&(C[i+0]<<2);
A[i+1]=(C[i+1]<<3)&(B[i+1]<<4);
```



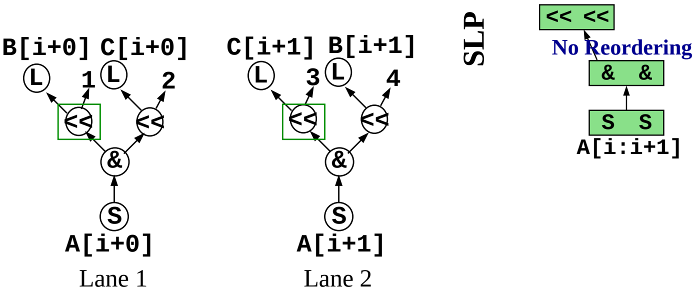
SLP



Non−Vectorizable Vectorizable +/−#Cost

1/3 Load address mismatch

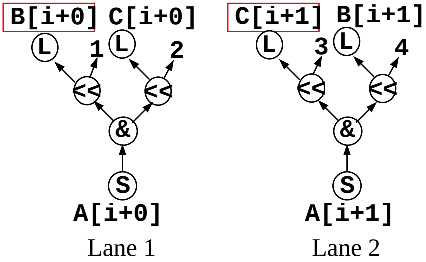
```
long A[],B[],C[];
A[i+0]=(B[i+0]<<1)&(C[i+0]<<2);
A[i+1]=(C[i+1]<<3)&(B[i+1]<<4);
```

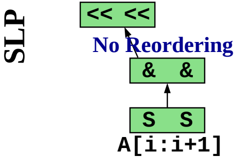


Non−Vectorizable Vectorizable +/−#Cost

1/3 Load address mismatch

```
long A[],B[],C[];
A[i+0]=(B[i+0]<<1)&(C[i+0]<<2);
A[i+1]=(C[i+1]<<3)&(B[i+1]<<4);
```

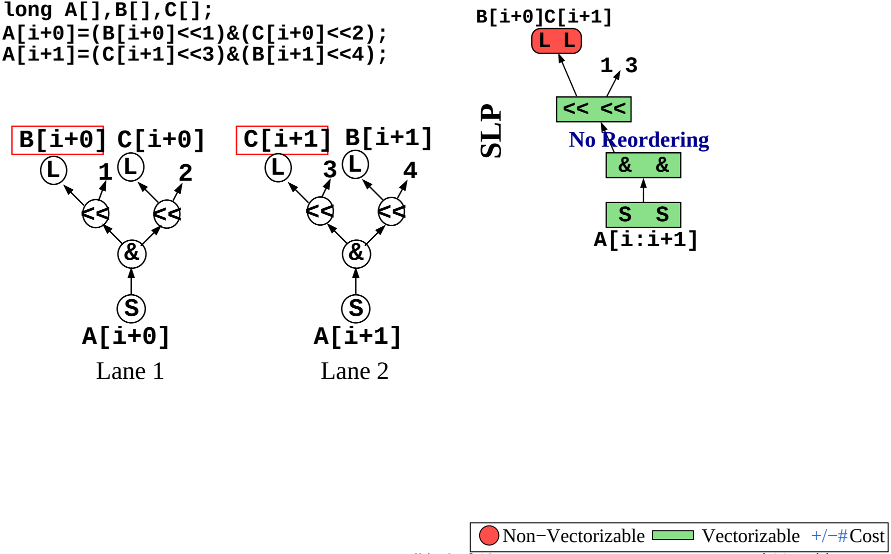




Non−Vectorizable Vectorizable +/−#Cost

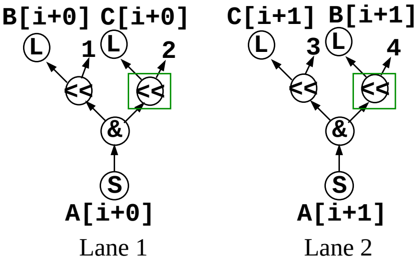
1/3 Load address mismatch

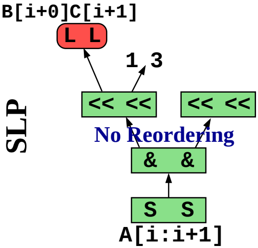
```
long A[],B[],C[];
A[i+0]=(B[i+0]<<1)&(C[i+0]<<2);
A[i+1]=(C[i+1]<<3)&(B[i+1]<<4);
```



1/3 Load address mismatch

```
long A[],B[],C[];
A[i+0]=(B[i+0]<<1)&(C[i+0]<<2);
A[i+1]=(C[i+1]<<3)&(B[i+1]<<4);
```

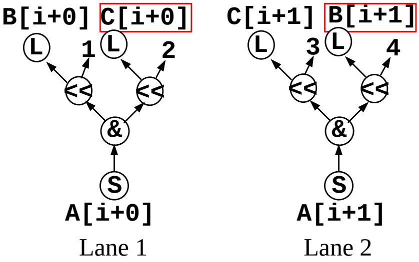


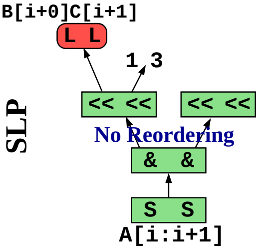


Non−Vectorizable Vectorizable +/−#Cost

1/3 Load address mismatch

```
long A[],B[],C[];
A[i+0]=(B[i+0]<<1)&(C[i+0]<<2);
A[i+1]=(C[i+1]<<3)&(B[i+1]<<4);
```

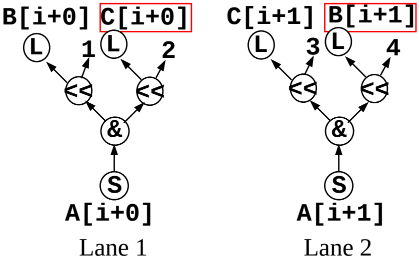


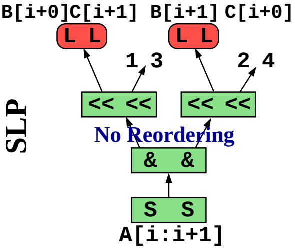


Non−Vectorizable Vectorizable +/−#Cost

1/3 Load address mismatch

```
long A[],B[],C[];
A[i+0]=(B[i+0]<<1)&(C[i+0]<<2);
A[i+1]=(C[i+1]<<3)&(B[i+1]<<4);
```

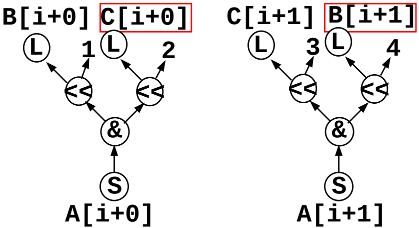




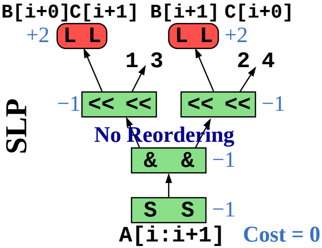
Non−Vectorizable Vectorizable +/−#Cost

1/3 Load address mismatch

```
long A[],B[],C[];
A[i+0]=(B[i+0]<<1)&(C[i+0]<<2);
A[i+1]=(C[i+1]<<3)&(B[i+1]<<4);
```



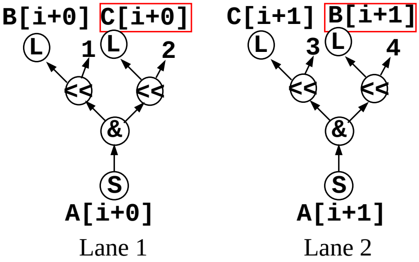
Lane 1 Lane 2

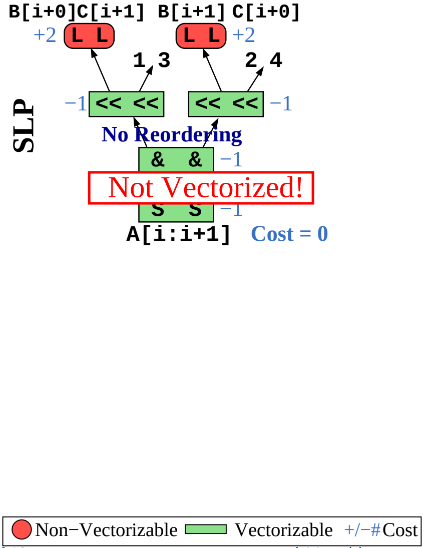


Non−Vectorizable Vectorizable +/−#Cost

1/3 Load address mismatch

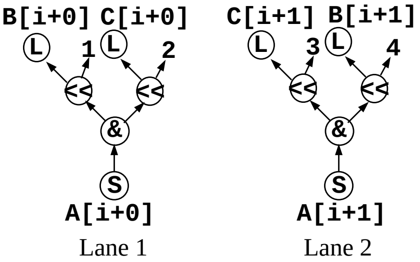
```
long A[],B[],C[];
A[i+0]=(B[i+0]<<1)&(C[i+0]<<2);
A[i+1]=(C[i+1]<<3)&(B[i+1]<<4);
```

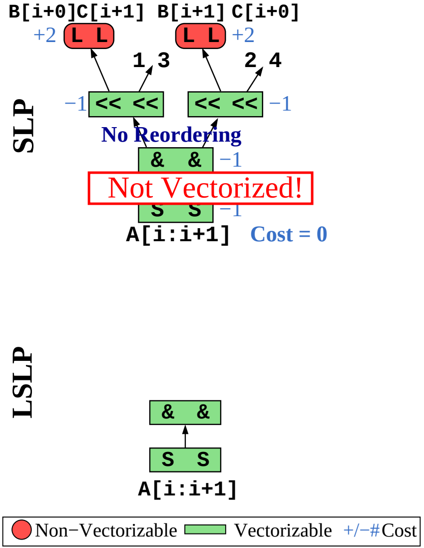




1/3 Load address mismatch

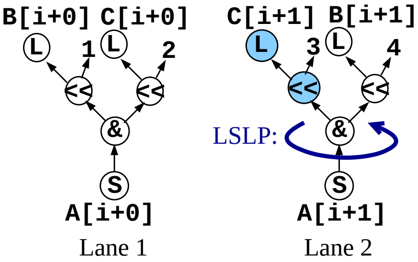
```
long A[],B[],C[];
A[i+0]=(B[i+0]<<1)&(C[i+0]<<2);
A[i+1]=(C[i+1]<<3)&(B[i+1]<<4);
```

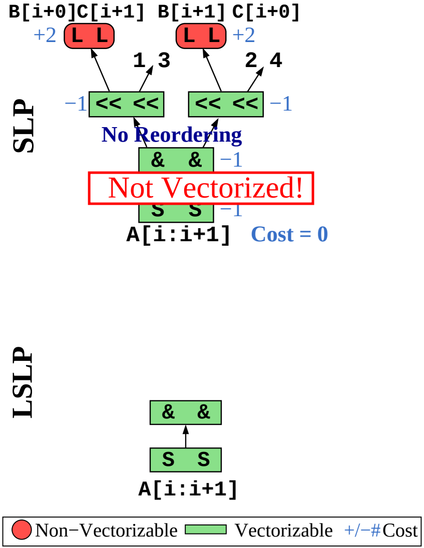




1/3 Load address mismatch

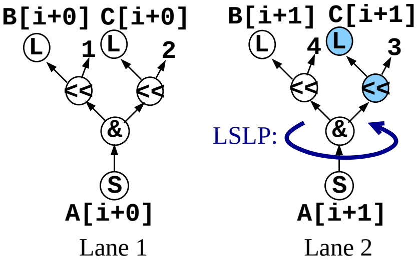
```
long A[],B[],C[];
A[i+0]=(B[i+0]<<1)&(C[i+0]<<2);
A[i+1]=(C[i+1]<<3)&(B[i+1]<<4);
```

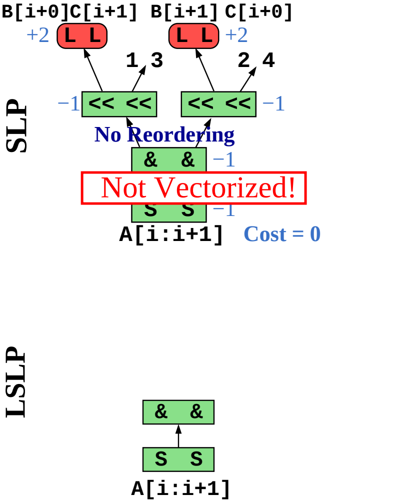




1/3 Load address mismatch

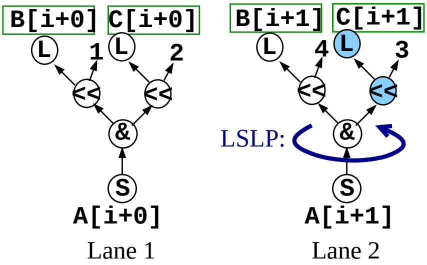
```
long A[],B[],C[];
A[i+0]=(B[i+0]<<1)&(C[i+0]<<2);
A[i+1]=(C[i+1]<<3)&(B[i+1]<<4);
```

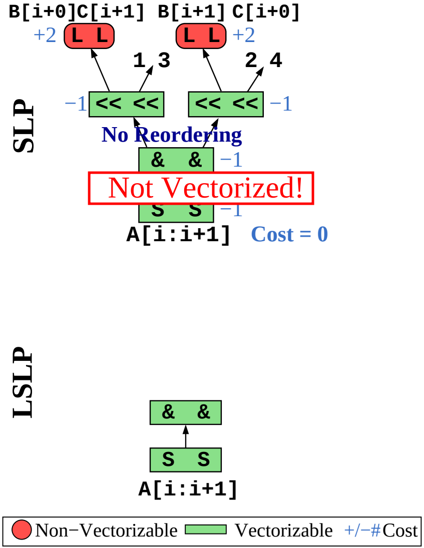




1/3 Load address mismatch

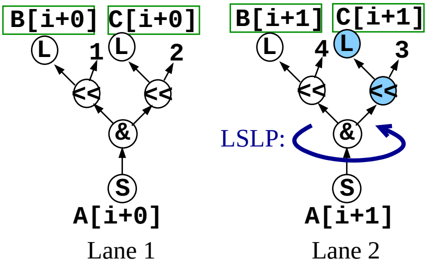
```
long A[],B[],C[];
A[i+0]=(B[i+0]<<1)&(C[i+0]<<2);
A[i+1]=(C[i+1]<<3)&(B[i+1]<<4);
```

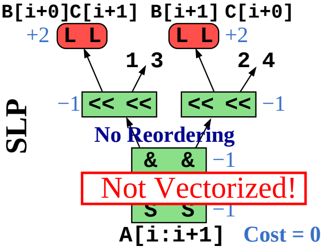


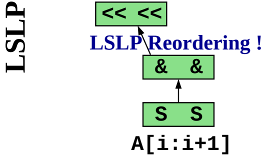


1/3 Load address mismatch

```
long A[],B[],C[];
A[i+0]=(B[i+0]<<1)&(C[i+0]<<2);
A[i+1]=(C[i+1]<<3)&(B[i+1]<<4);
```



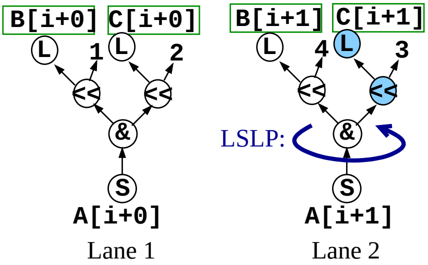


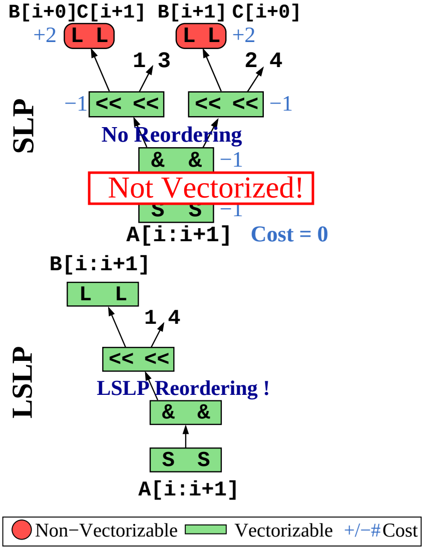


Non−Vectorizable Vectorizable +/−#Cost

1/3 Load address mismatch

```
long A[],B[],C[];
A[i+0]=(B[i+0]<<1)&(C[i+0]<<2);
A[i+1]=(C[i+1]<<3)&(B[i+1]<<4);
```

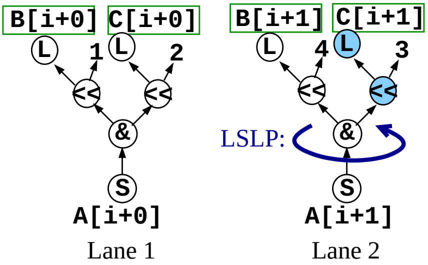


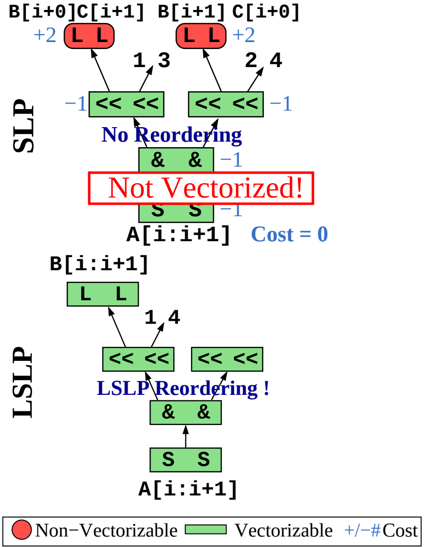


 http://vporpo.me

1/3 Load address mismatch

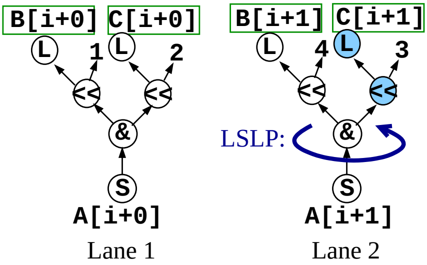
```
long A[],B[],C[];
A[i+0]=(B[i+0]<<1)&(C[i+0]<<2);
A[i+1]=(C[i+1]<<3)&(B[i+1]<<4);
```

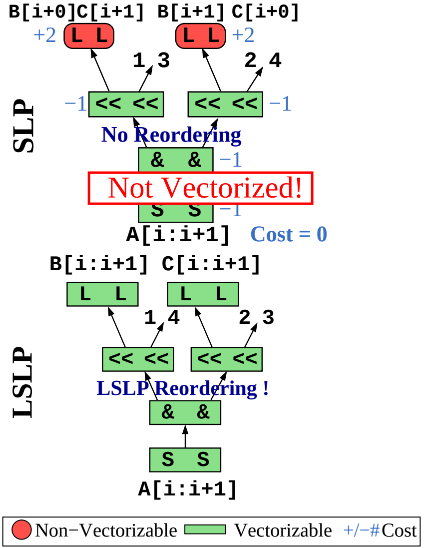




1/3 Load address mismatch

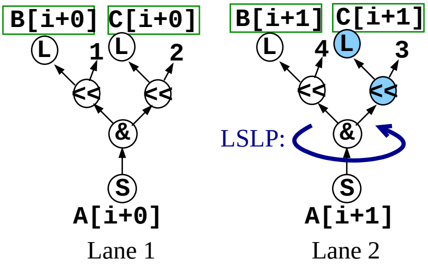
```
long A[],B[],C[];
A[i+0]=(B[i+0]<<1)&(C[i+0]<<2);
A[i+1]=(C[i+1]<<3)&(B[i+1]<<4);
```

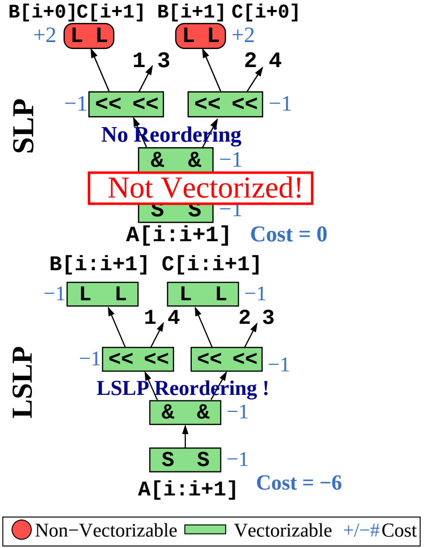




1/3 Load address mismatch

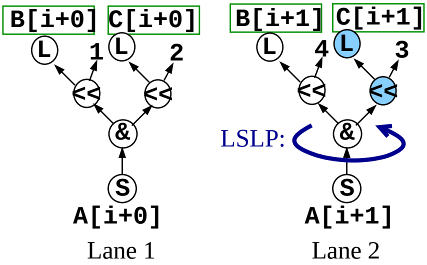
```
long A[],B[],C[];
A[i+0]=(B[i+0]<<1)&(C[i+0]<<2);
A[i+1]=(C[i+1]<<3)&(B[i+1]<<4);
```

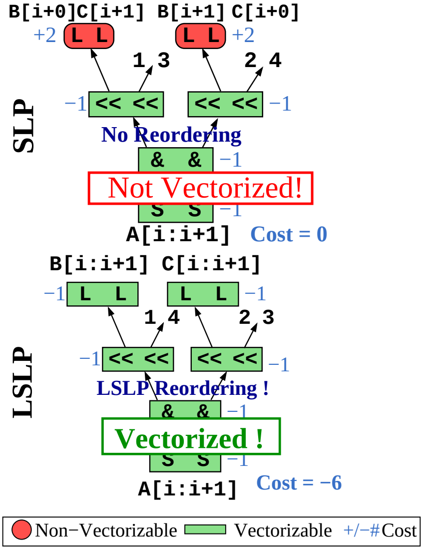




1/3 Load address mismatch

```
long A[],B[],C[];
A[i+0]=(B[i+0]<<1)&(C[i+0]<<2);
A[i+1]=(C[i+1]<<3)&(B[i+1]<<4);
```



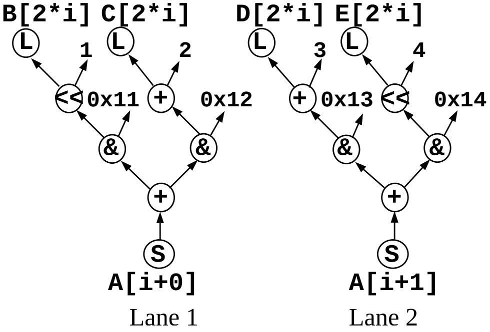


2/3 Opcodes mismatch

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```

2/3 Opcodes mismatch

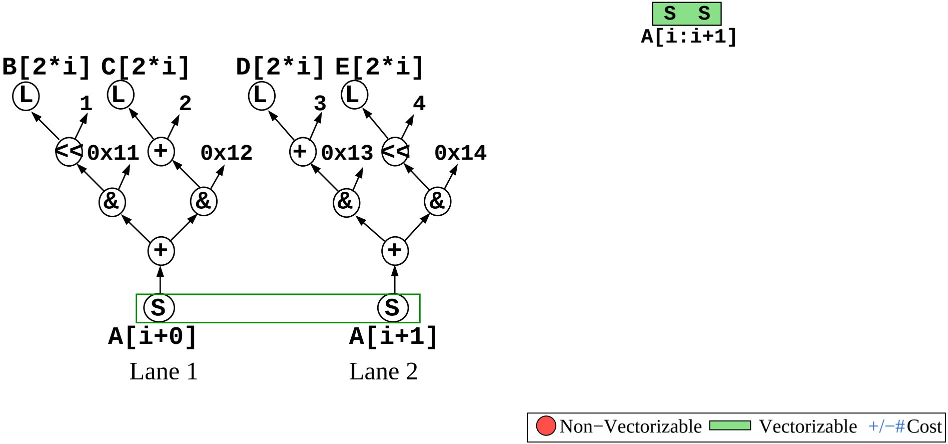
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```



2/3 Opcodes mismatch

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```

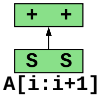
SLP

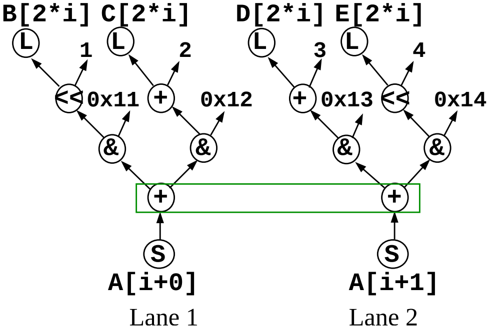


2/3 Opcodes mismatch

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```

SLP

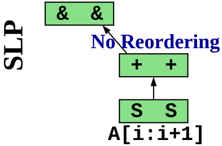


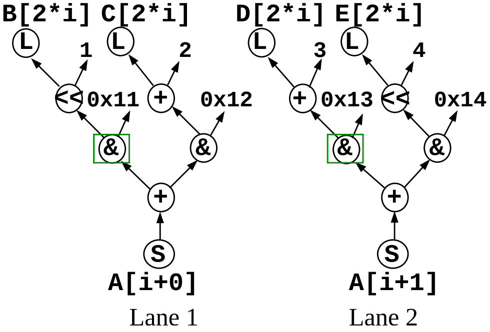


Non−Vectorizable Vectorizable +/−#Cost

2/3 Opcodes mismatch

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```



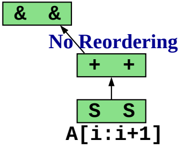


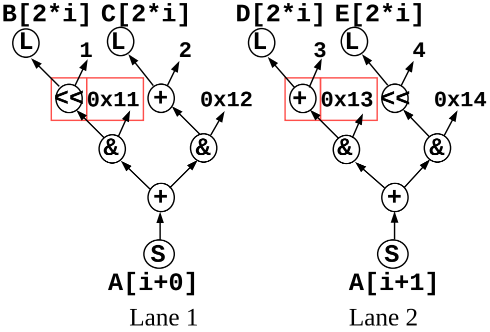
Non−Vectorizable Vectorizable +/−#Cost

2/3 Opcodes mismatch

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```

SLP

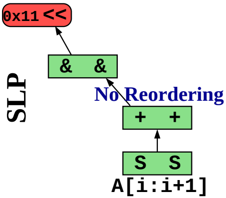


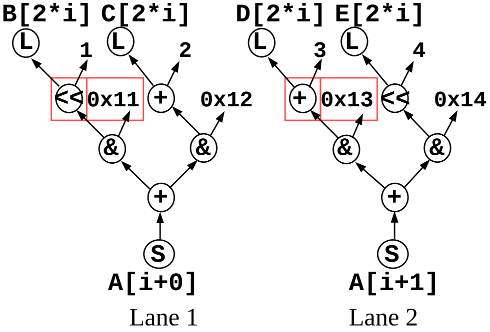


Non−Vectorizable Vectorizable +/−#Cost

2/3 Opcodes mismatch

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```

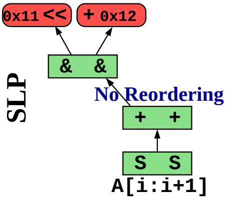


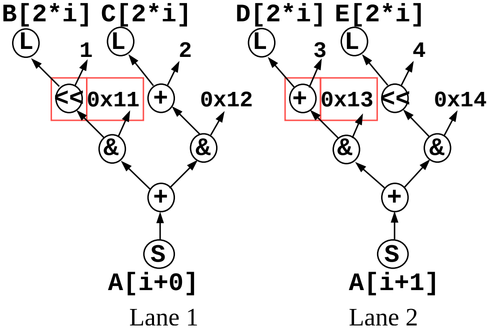


Non−Vectorizable Vectorizable +/−#Cost

2/3 Opcodes mismatch

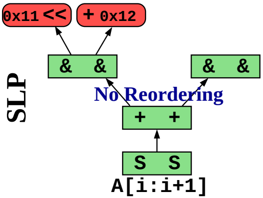
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```

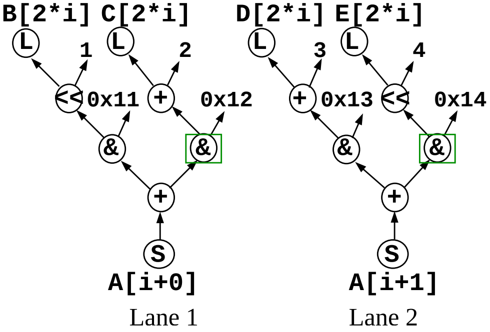




2/3 Opcodes mismatch

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```

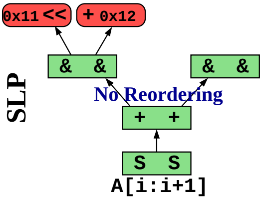


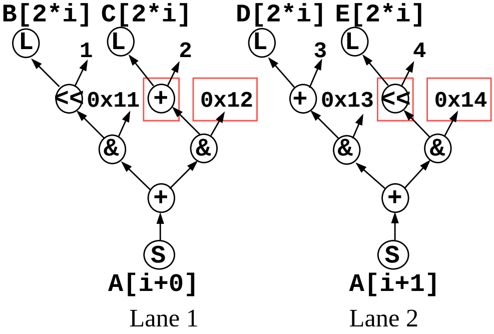


Non−Vectorizable Vectorizable +/−#Cost

2/3 Opcodes mismatch

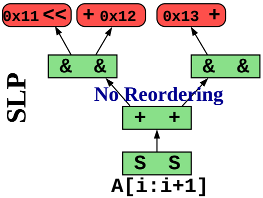
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```

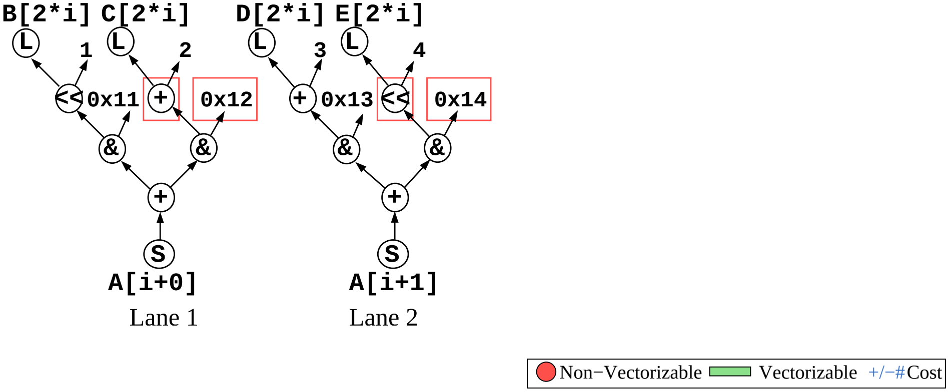




2/3 Opcodes mismatch

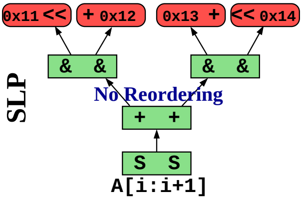
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```

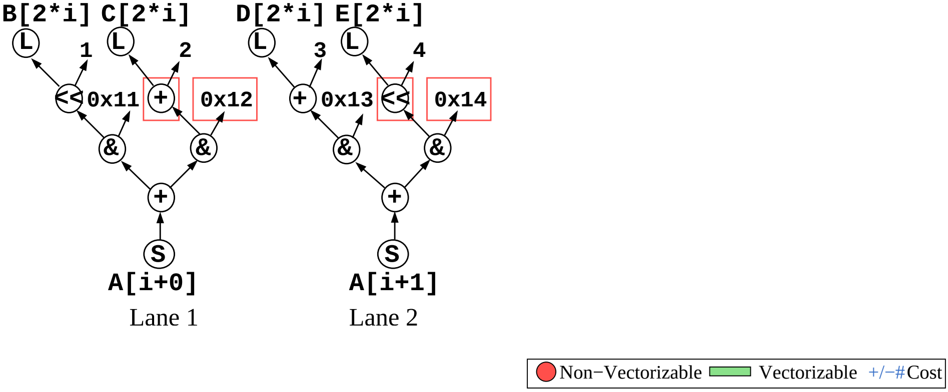




2/3 Opcodes mismatch

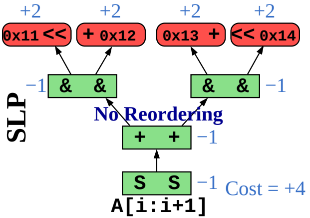
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```

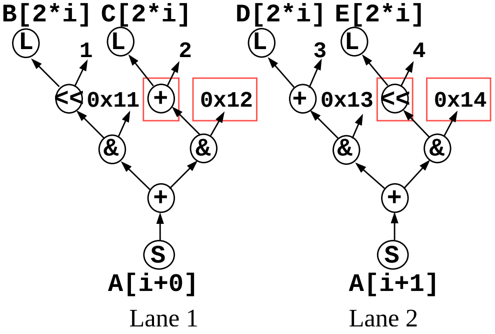




2/3 Opcodes mismatch

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```

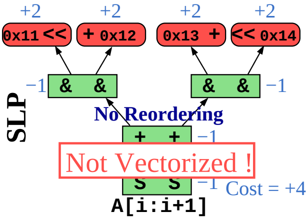


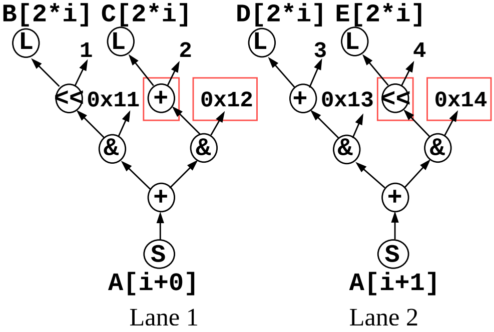


Non−Vectorizable Vectorizable +/−#Cost

2/3 Opcodes mismatch

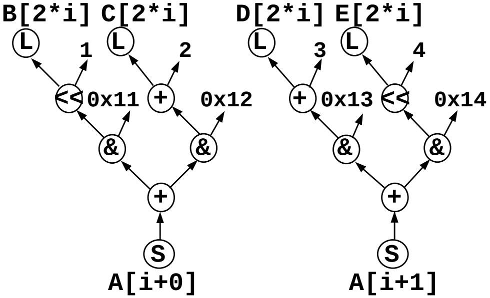
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```



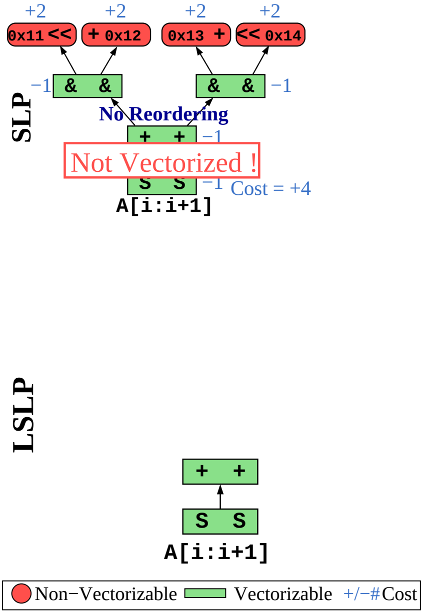


2/3 Opcodes mismatch

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```

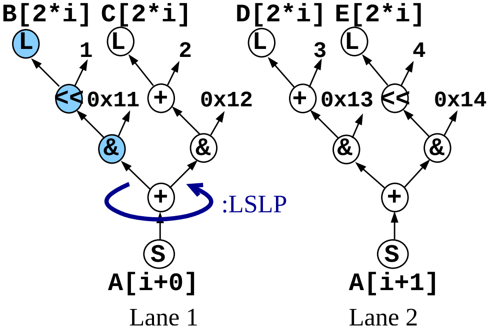


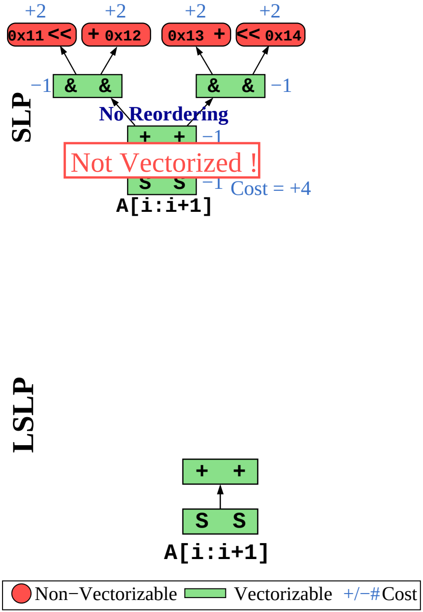
Lane 1 Lane 2



2/3 Opcodes mismatch

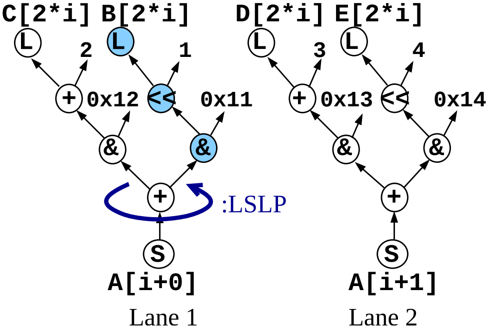
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```

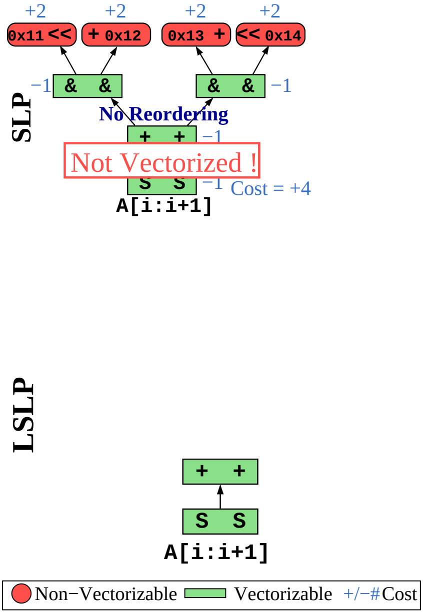




2/3 Opcodes mismatch

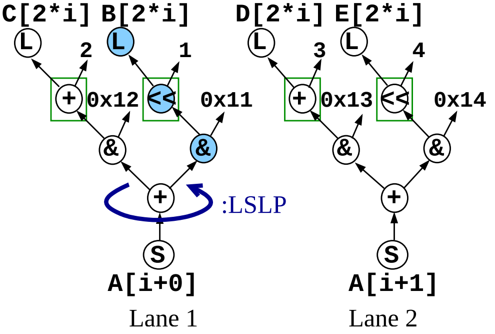
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```

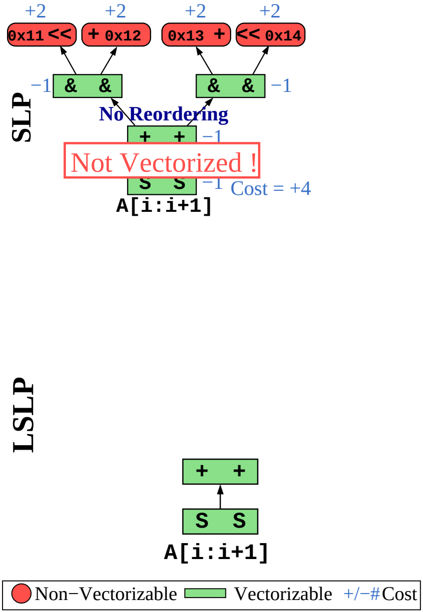




2/3 Opcodes mismatch

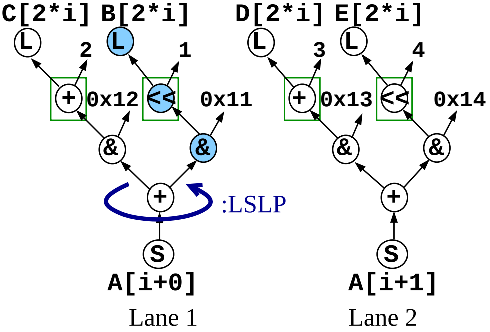
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```

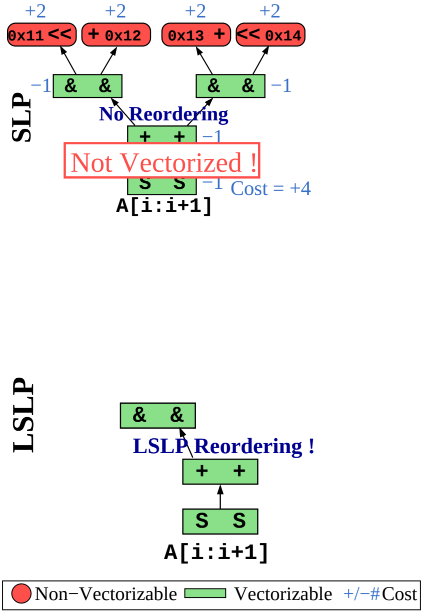




2/3 Opcodes mismatch

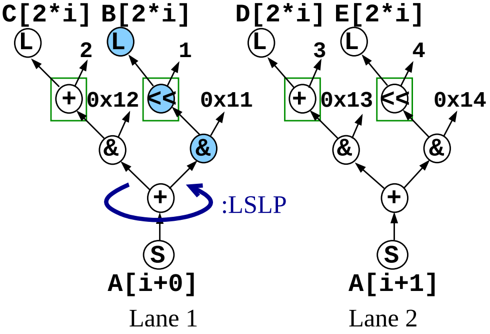
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```

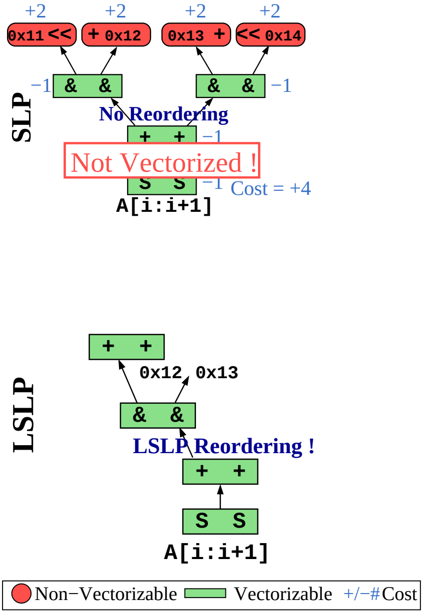




2/3 Opcodes mismatch

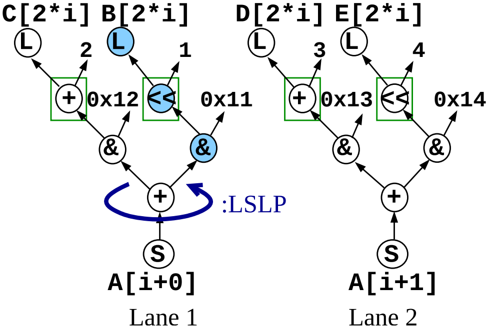
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```

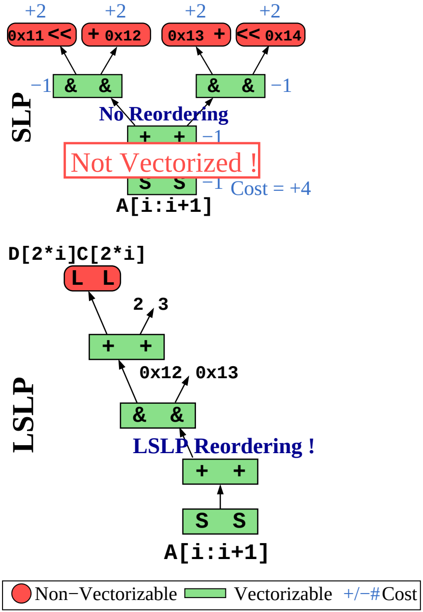




2/3 Opcodes mismatch

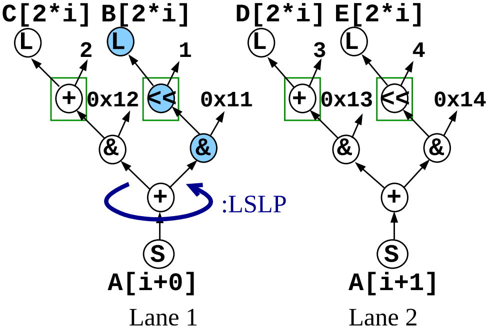
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```

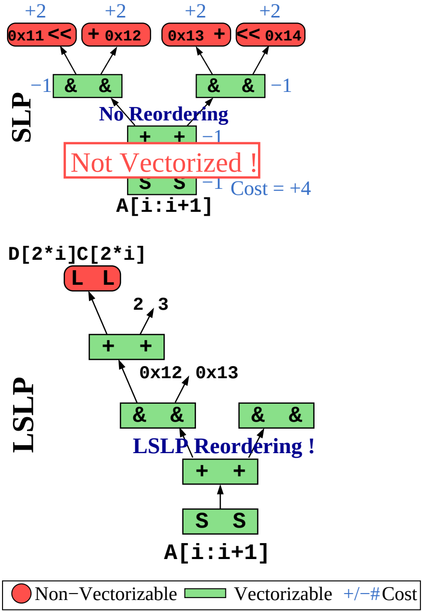




2/3 Opcodes mismatch

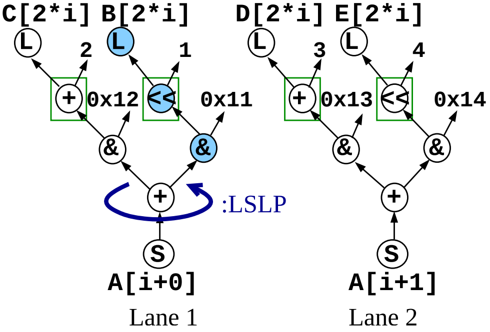
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```

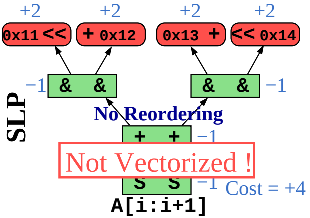


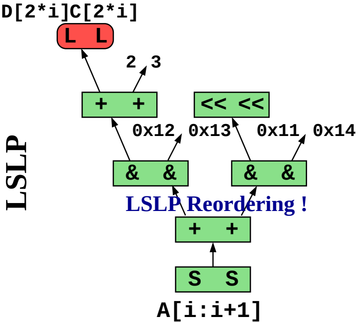


2/3 Opcodes mismatch

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```



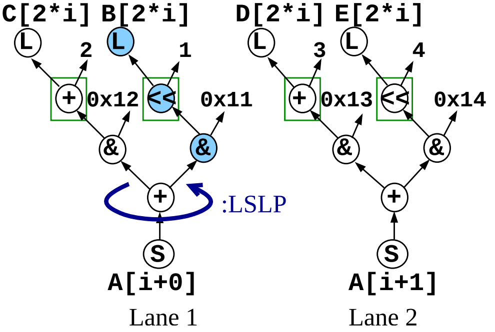


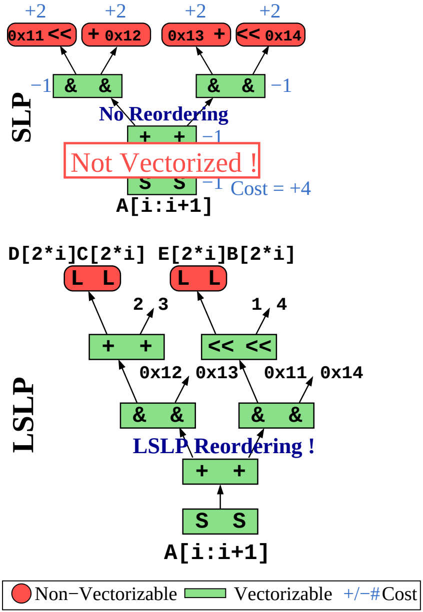


Non−Vectorizable Vectorizable +/−#Cost

2/3 Opcodes mismatch

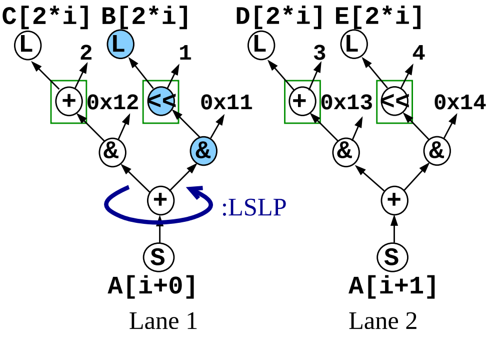
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```

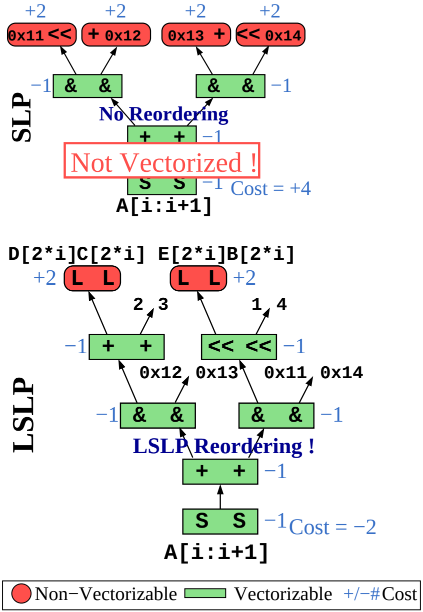




2/3 Opcodes mismatch

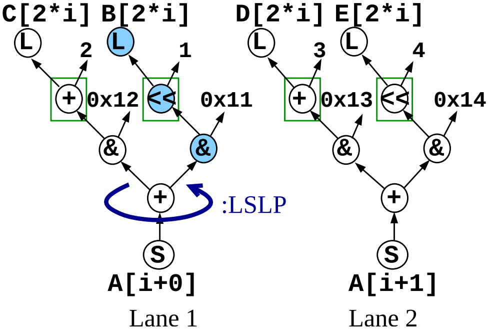
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```

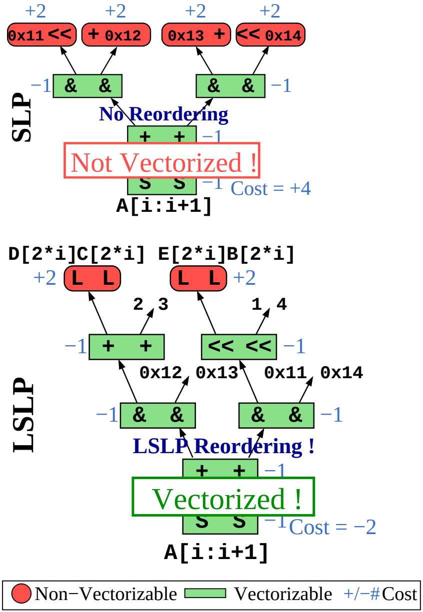




2/3 Opcodes mismatch

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=((B[2*i]<<1)&0x11)+((C[2*i]+ 2)&0x12);
A[i+1]=((D[2*i]+ 3)&0x13)+((E[2*i]<<4)&0x14);
```



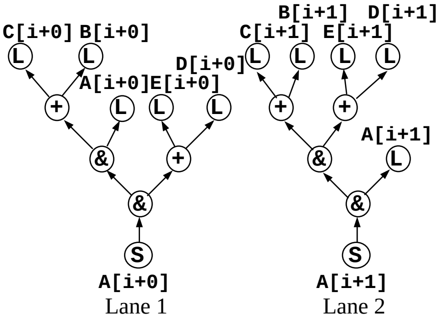


3/3 Chains of Commutative Operations

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```

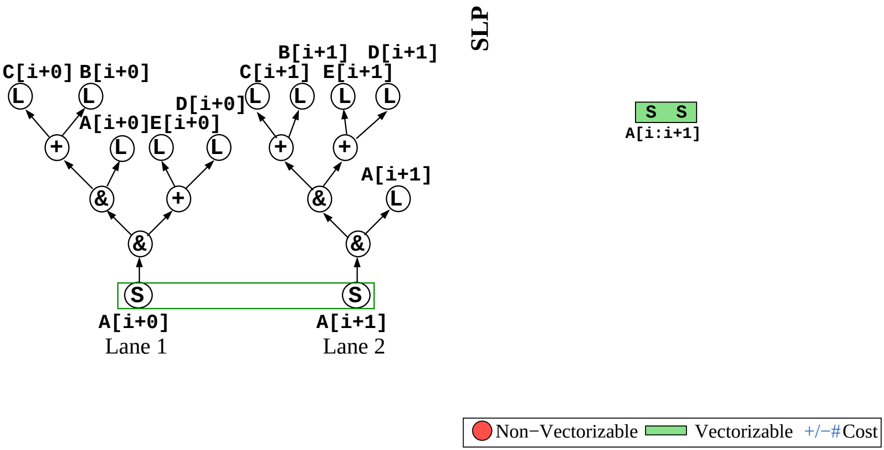
3/3 Chains of Commutative Operations

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```



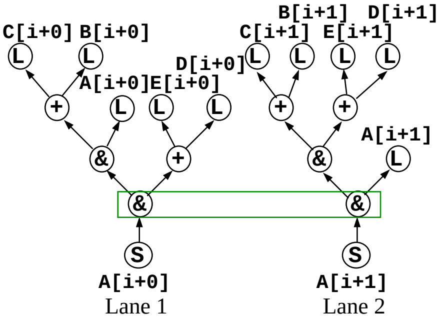
3/3 Chains of Commutative Operations

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```

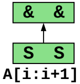


3/3 Chains of Commutative Operations

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```



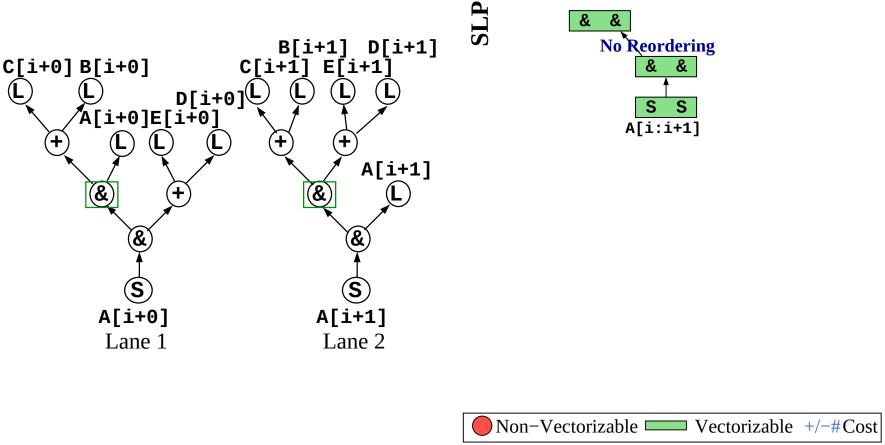
SLP



Non−Vectorizable Vectorizable +/−#Cost

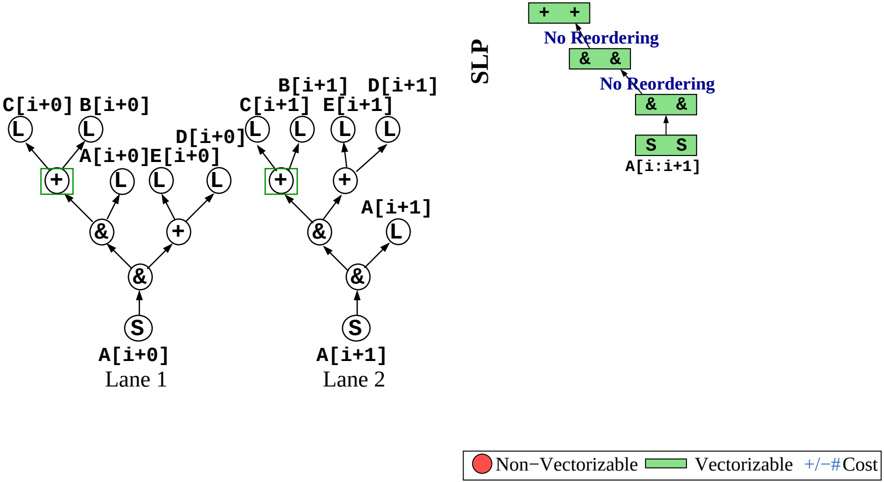
3/3 Chains of Commutative Operations

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```



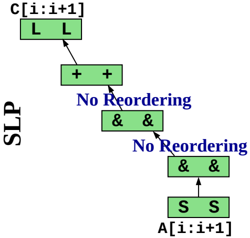
3/3 Chains of Commutative Operations

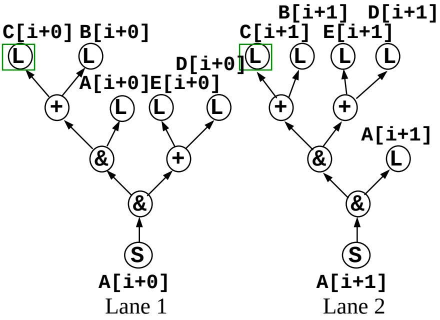
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```



3/3 Chains of Commutative Operations

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```

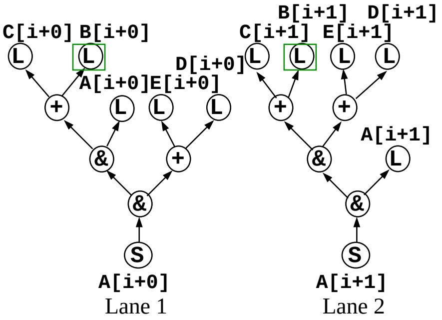


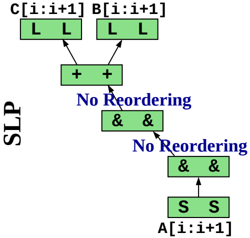


Non−Vectorizable Vectorizable +/−#Cost

3/3 Chains of Commutative Operations

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```

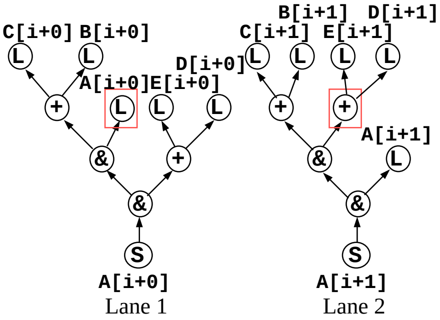


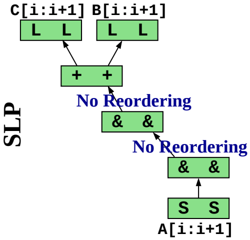


Non−Vectorizable Vectorizable +/−#Cost

3/3 Chains of Commutative Operations

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```

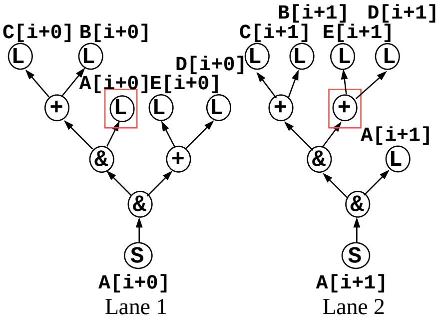


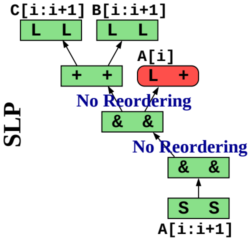


Non−Vectorizable Vectorizable +/−#Cost

3/3 Chains of Commutative Operations

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```

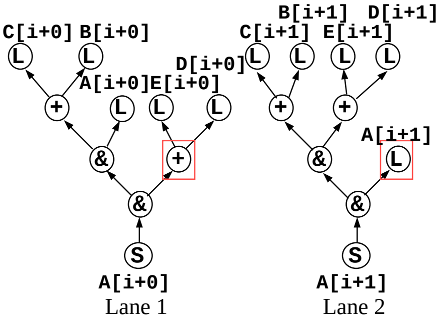


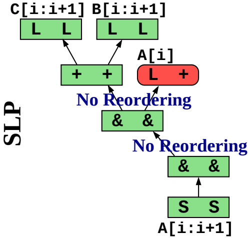


Non−Vectorizable Vectorizable +/−#Cost

3/3 Chains of Commutative Operations

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```

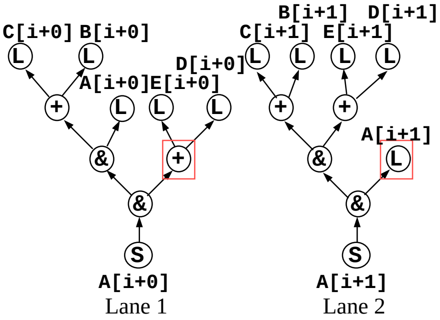


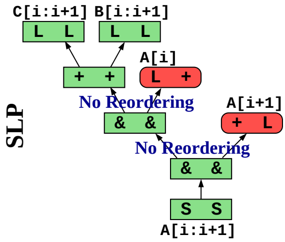


Non−Vectorizable Vectorizable +/−#Cost

3/3 Chains of Commutative Operations

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```

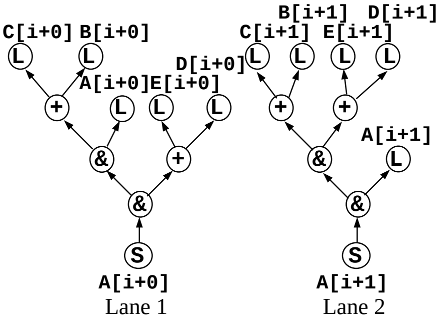


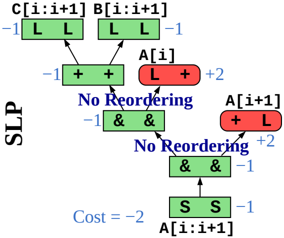


Non−Vectorizable Vectorizable +/−#Cost

3/3 Chains of Commutative Operations

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```

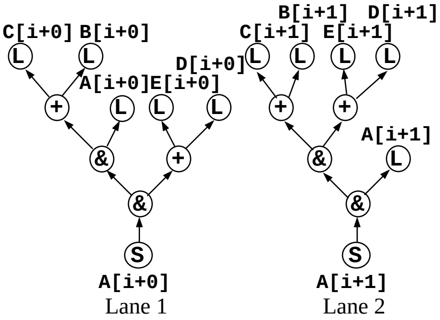


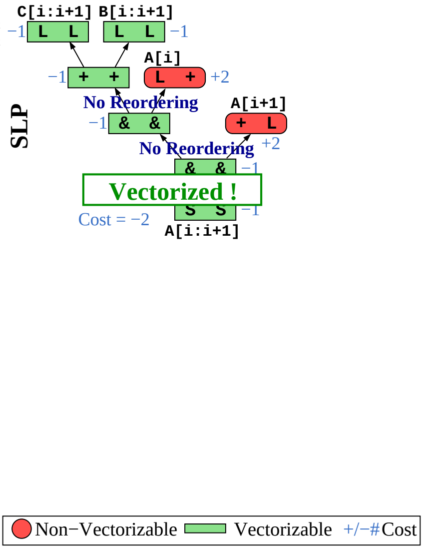


Non−Vectorizable Vectorizable +/−# Cost

3/3 Chains of Commutative Operations

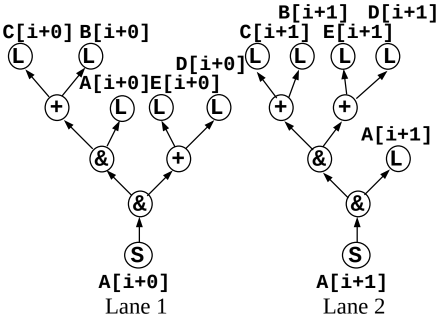
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```

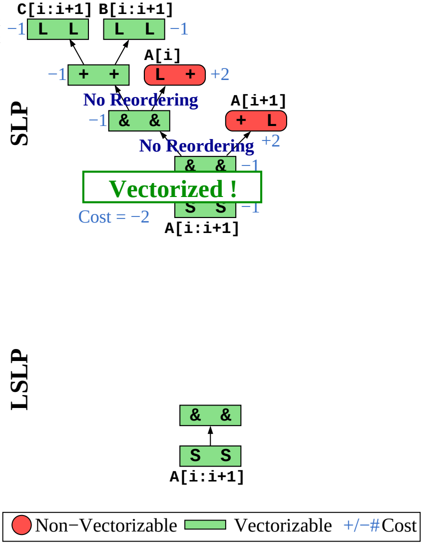




3/3 Chains of Commutative Operations

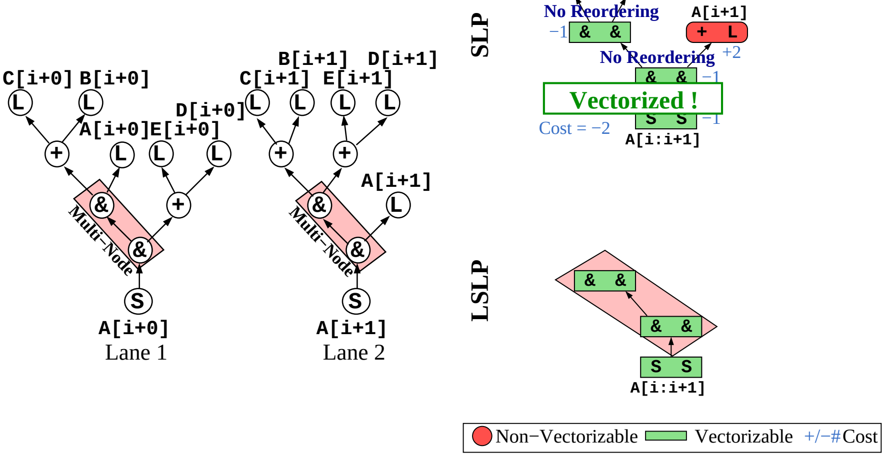
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```





3/3 Chains of Commutative Operations

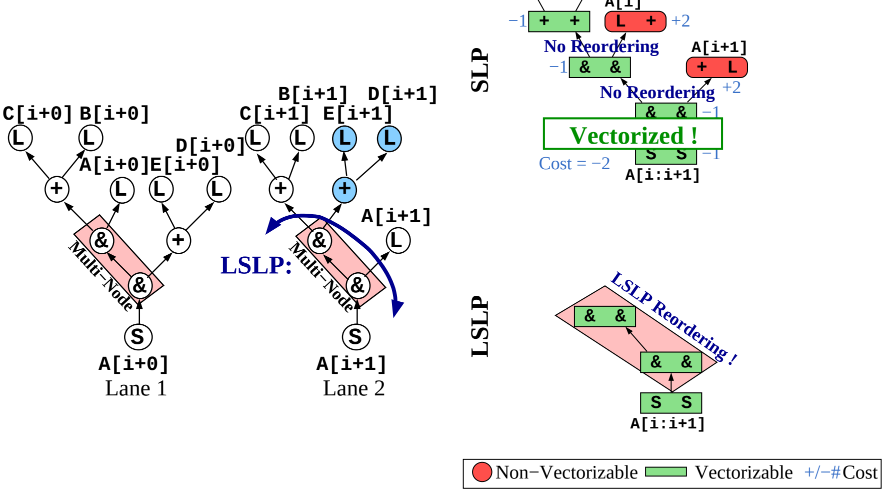
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```



 http://vporpo.me

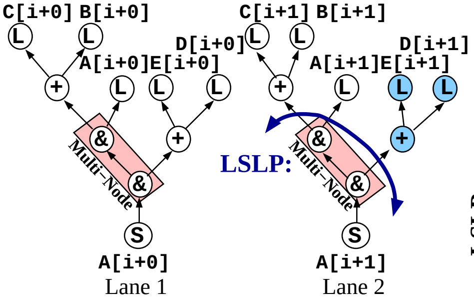
3/3 Chains of Commutative Operations

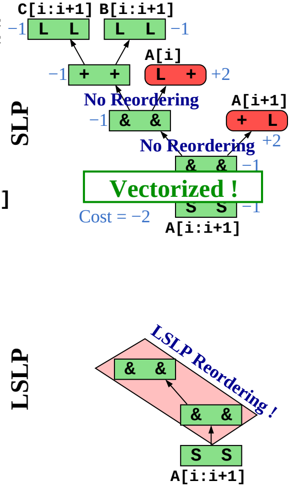
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```



3/3 Chains of Commutative Operations

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```

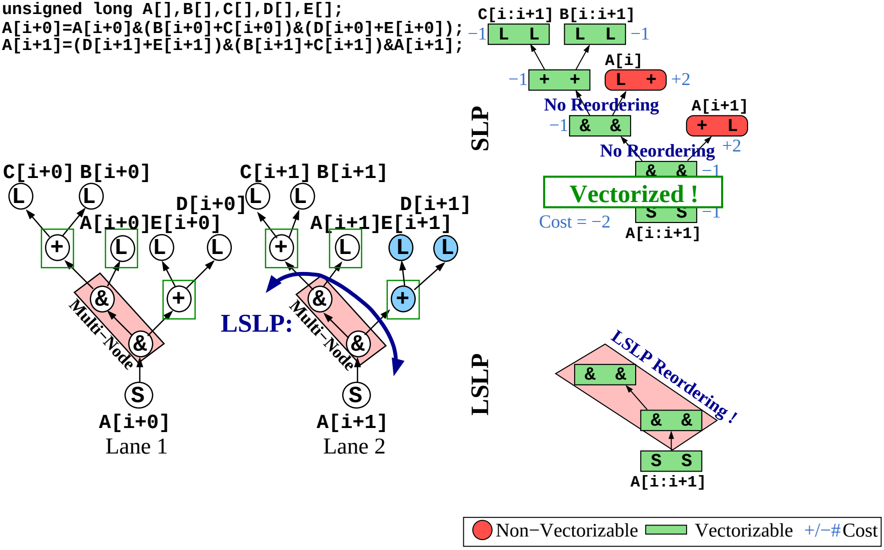




Non−Vectorizable Vectorizable +/−#Cost

3/3 Chains of Commutative Operations

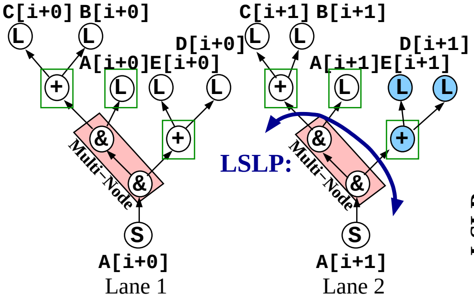
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```

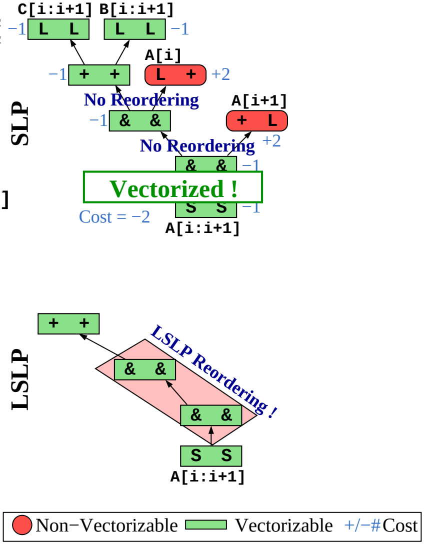


 http://vporpo.me

3/3 Chains of Commutative Operations

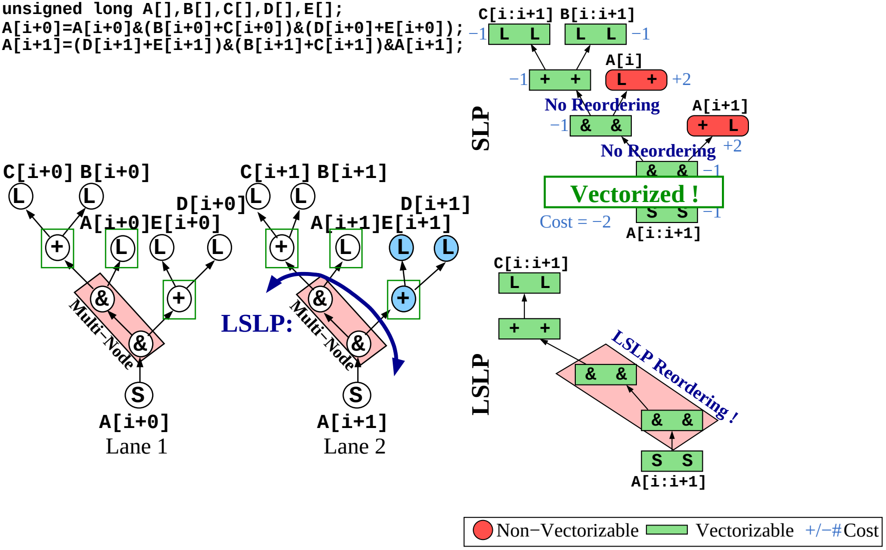
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```





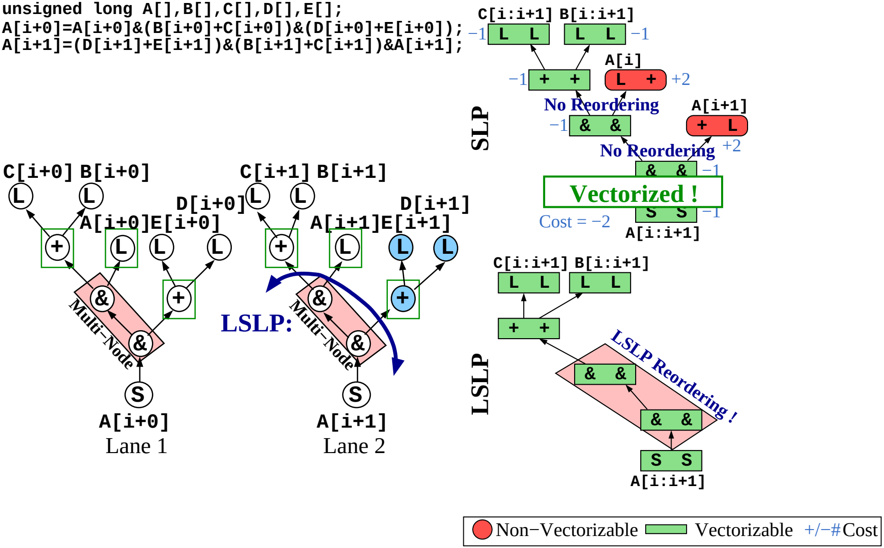
3/3 Chains of Commutative Operations

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```



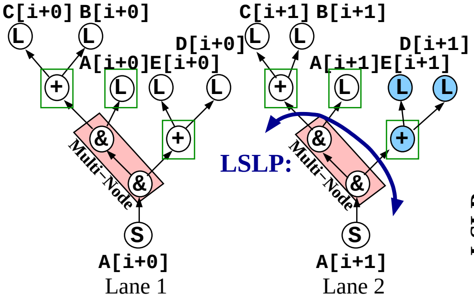
3/3 Chains of Commutative Operations

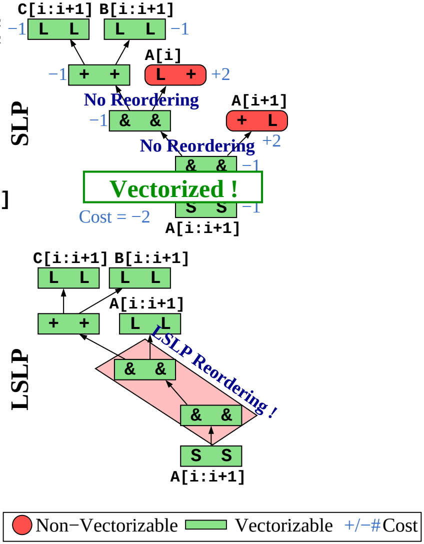
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```



3/3 Chains of Commutative Operations

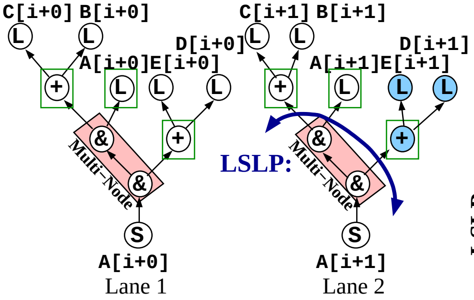
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```

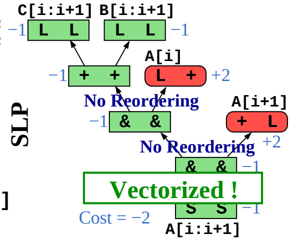


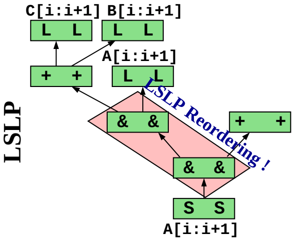


3/3 Chains of Commutative Operations

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```



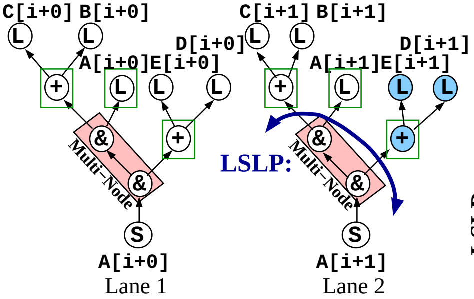


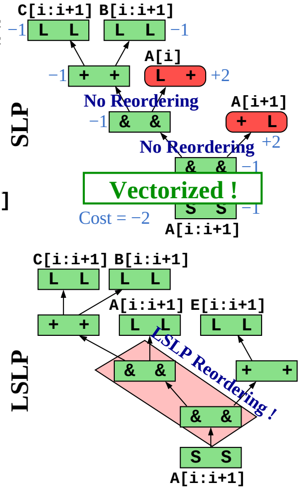


Non−Vectorizable Vectorizable +/−#Cost

3/3 Chains of Commutative Operations

```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```

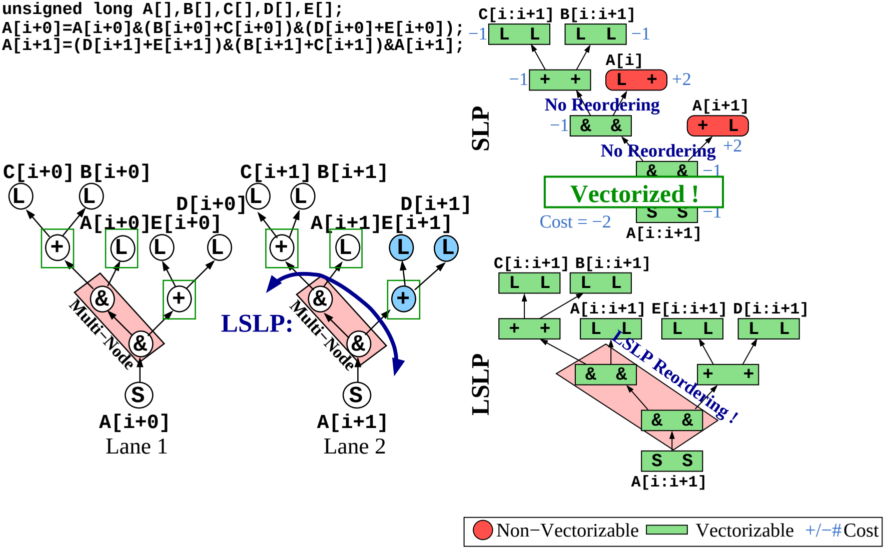




Non−Vectorizable Vectorizable +/−#Cost

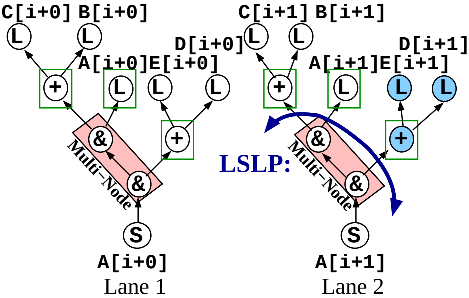
3/3 Chains of Commutative Operations

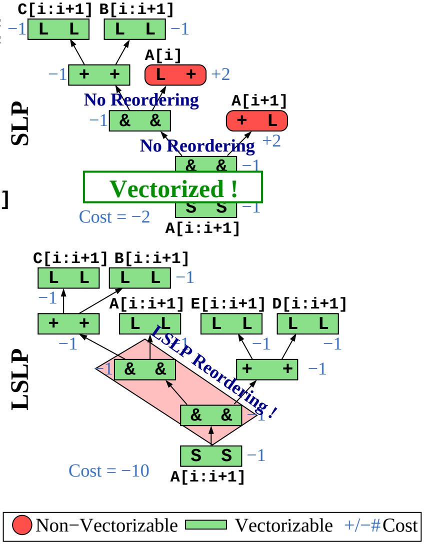
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```



3/3 Chains of Commutative Operations

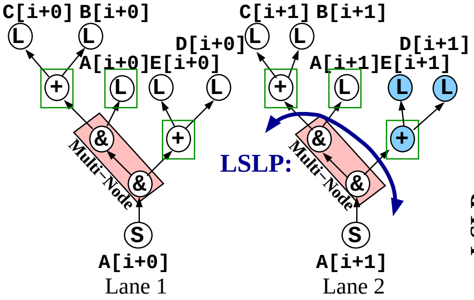
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```

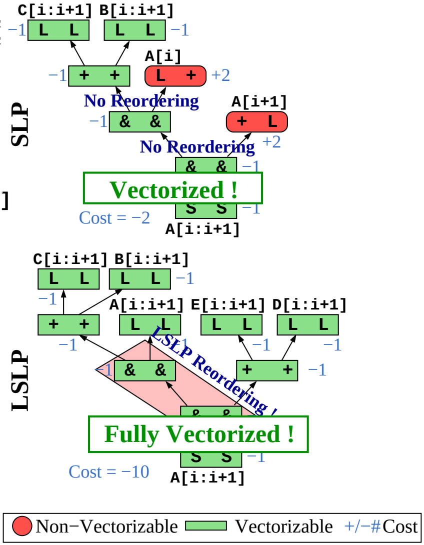




3/3 Chains of Commutative Operations

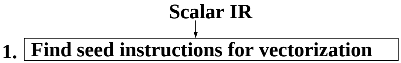
```
unsigned long A[],B[],C[],D[],E[];
A[i+0]=A[i+0]&(B[i+0]+C[i+0])&(D[i+0]+E[i+0]);
A[i+1]=(D[i+1]+E[i+1])&(B[i+1]+C[i+1])&A[i+1];
```





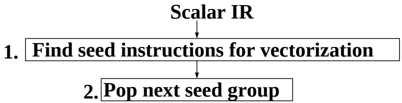
(L)SLP Algorithm

- Seed instructions are usually:
  1. Consecutive Stores
  2. Reductions



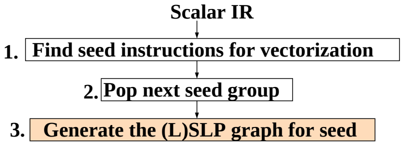
(L)SLP Algorithm

- Seed instructions are usually:
  1. Consecutive Stores
  2. Reductions



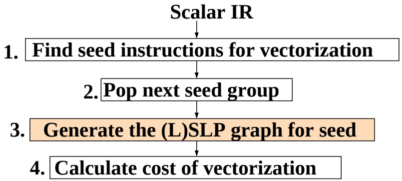
(L)SLP Algorithm

- Seed instructions are usually:
  1. Consecutive Stores
  2. Reductions
- Graph contains groups of vectorizable instructions



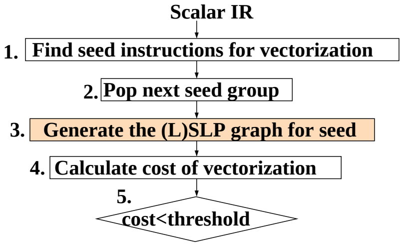
(L)SLP Algorithm

- Seed instructions are usually:
  1. Consecutive Stores
  2. Reductions
- Graph contains groups of vectorizable instructions
- Cost: weighted instr. count



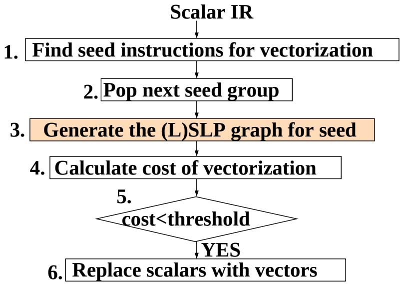
(L)SLP Algorithm

- Seed instructions are usually:
  1. Consecutive Stores
  2. Reductions
- Graph contains groups of vectorizable instructions
- Cost: weighted instr. count
- Check overall profitability



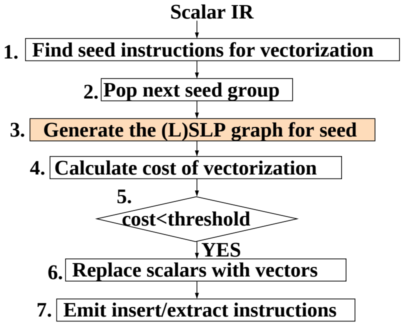
(L)SLP Algorithm

- Seed instructions are usually:
  1. Consecutive Stores
  2. Reductions
- Graph contains groups of vectorizable instructions
- Cost: weighted instr. count
- Check overall profitability
- Generate vector code



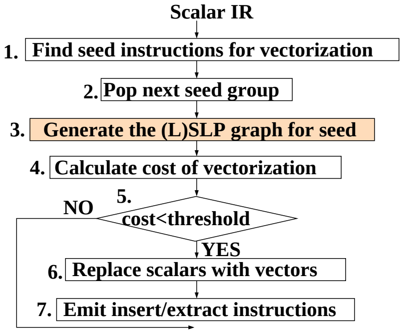
(L)SLP Algorithm

- Seed instructions are usually:
  1. Consecutive Stores
  2. Reductions
- Graph contains groups of vectorizable instructions
- Cost: weighted instr. count
- Check overall profitability
- Generate vector code



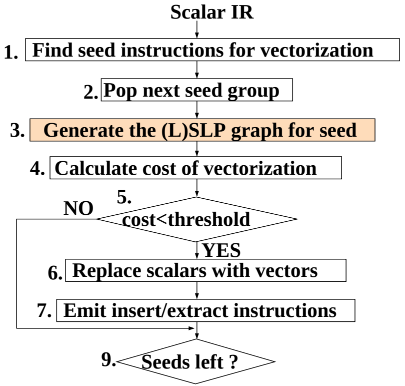
(L)SLP Algorithm

- Seed instructions are usually:
  1. Consecutive Stores
  2. Reductions
- Graph contains groups of vectorizable instructions
- Cost: weighted instr. count
- Check overall profitability
- Generate vector code



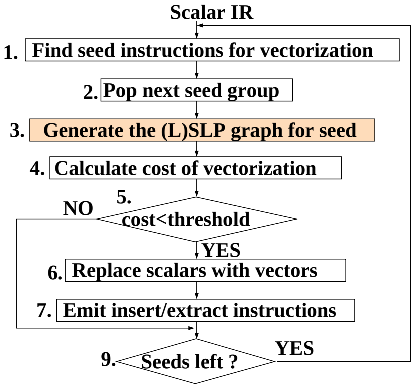
(L)SLP Algorithm

- Seed instructions are usually:
  1. Consecutive Stores
  2. Reductions
- Graph contains groups of vectorizable instructions
- Cost: weighted instr. count
- Check overall profitability
- Generate vector code



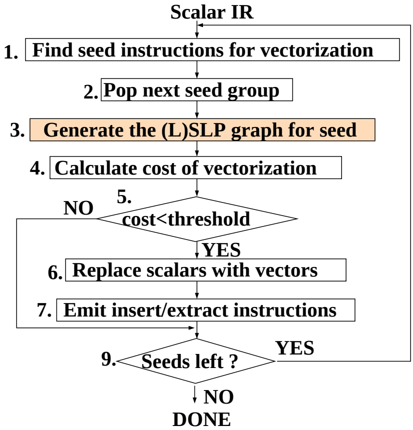
(L)SLP Algorithm

- Seed instructions are usually:
  1. Consecutive Stores
  2. Reductions
- Graph contains groups of vectorizable instructions
- Cost: weighted instr. count
- Check overall profitability
- Generate vector code
- Repeat



(L)SLP Algorithm

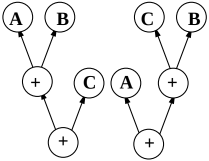
- Seed instructions are usually:
  1. Consecutive Stores
  2. Reductions
- Graph contains groups of vectorizable instructions
- Cost: weighted instr. count
- Check overall profitability
- Generate vector code
- Repeat



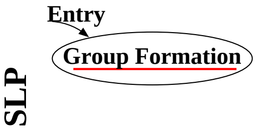
(L)SLP Graph Formation

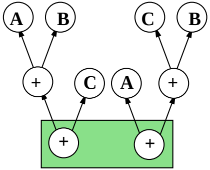
Entry

SLP

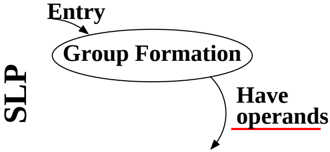


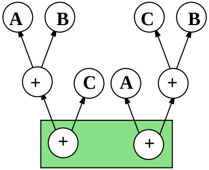
(L)SLP Graph Formation



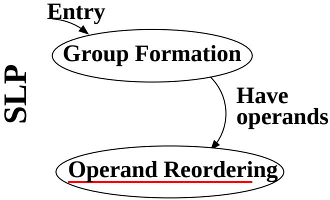


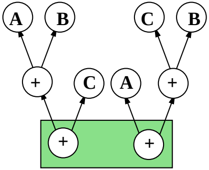
(L)SLP Graph Formation



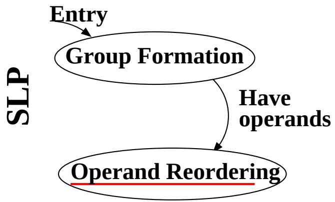


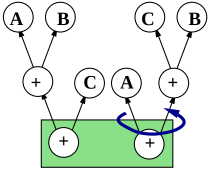
(L)SLP Graph Formation



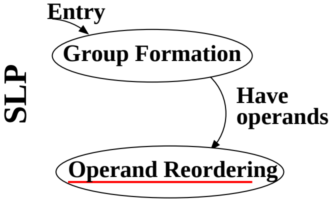


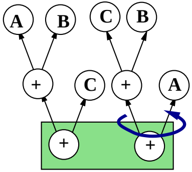
(L)SLP Graph Formation



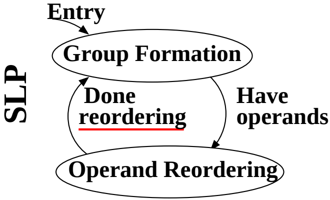


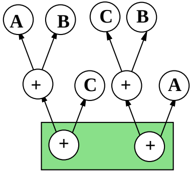
(L)SLP Graph Formation



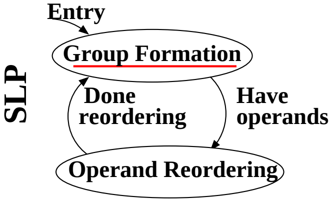


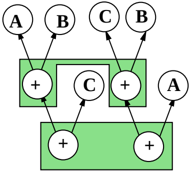
(L)SLP Graph Formation



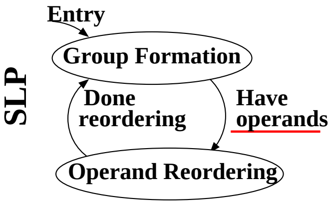


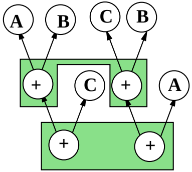
(L)SLP Graph Formation



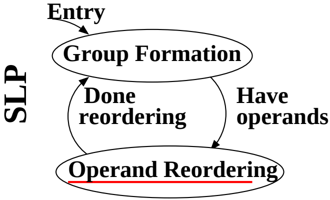


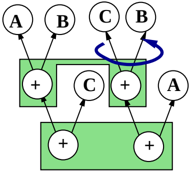
(L)SLP Graph Formation



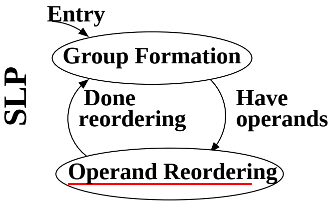


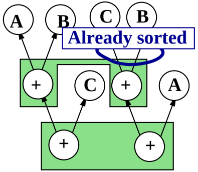
(L)SLP Graph Formation



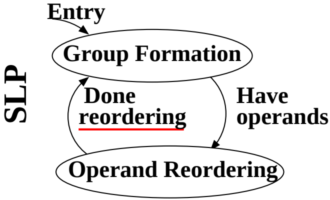


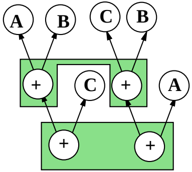
(L)SLP Graph Formation



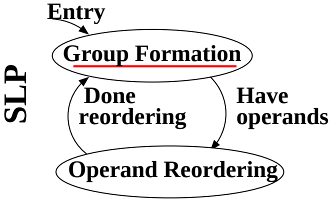


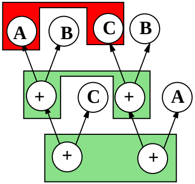
(L)SLP Graph Formation



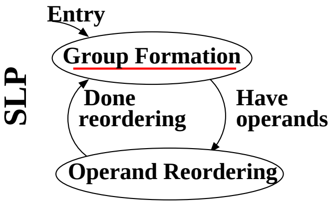


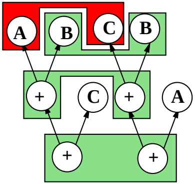
(L)SLP Graph Formation



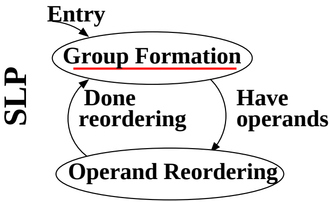


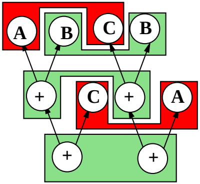
(L)SLP Graph Formation





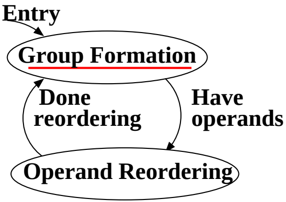
(L)SLP Graph Formation

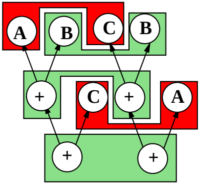




(L)SLP Graph Formation

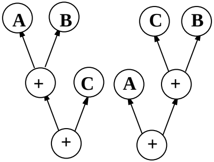
SLP





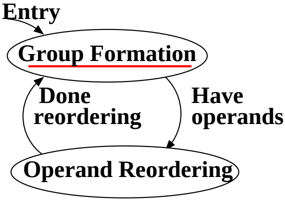
- LSLP: Extended DAG Formation with Multi-Nodes

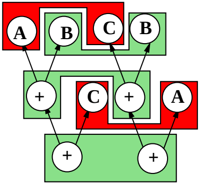
LSLP



(L)SLP Graph Formation

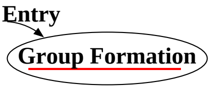
SLP

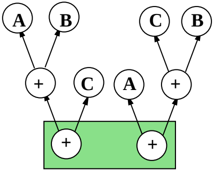




- LSLP: Extended DAG Formation with Multi-Nodes

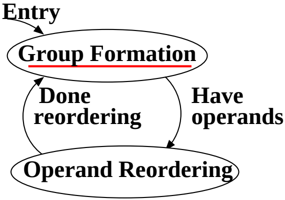
LSLP

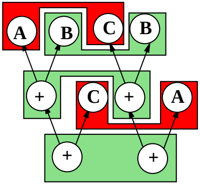




(L)SLP Graph Formation

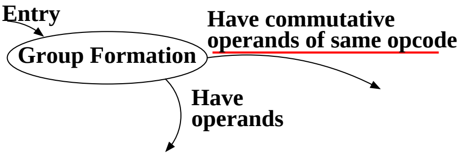
SLP

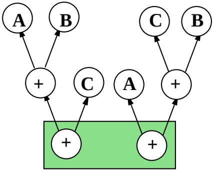




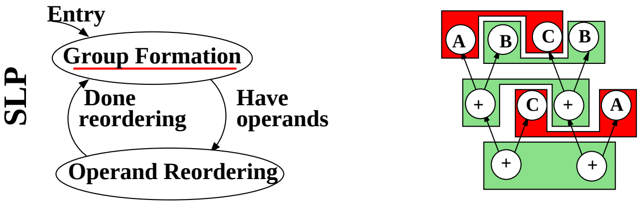
- LSLP: Extended DAG Formation with Multi-Nodes

LSLP

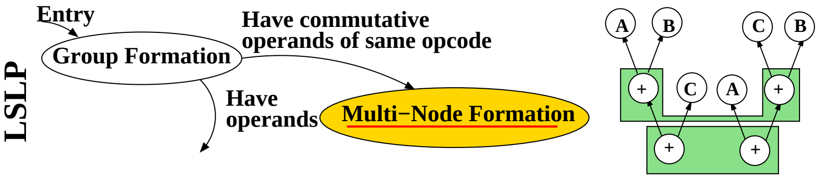




(L)SLP Graph Formation

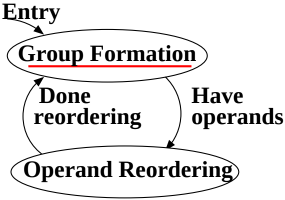


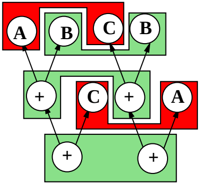
- LSLP: Extended DAG Formation with Multi-Nodes



(L)SLP Graph Formation

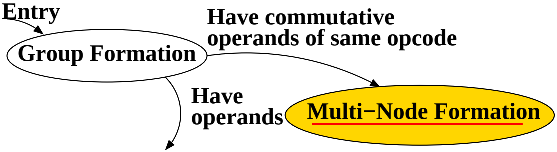
SLP

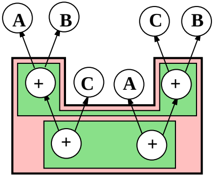




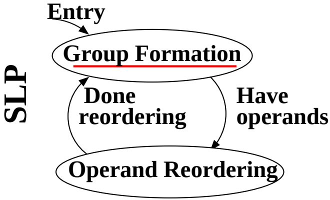
- LSLP: Extended DAG Formation with Multi-Nodes

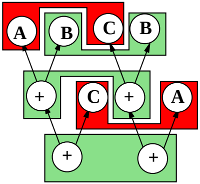
LSLP



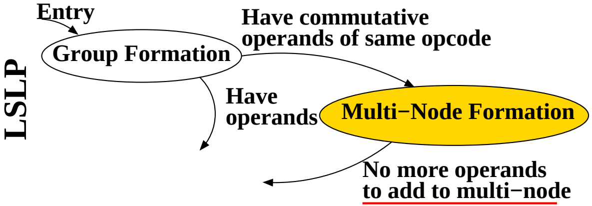


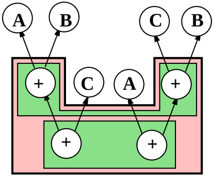
(L)SLP Graph Formation



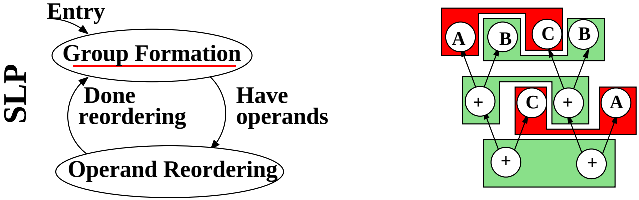


- LSLP: Extended DAG Formation with Multi-Nodes

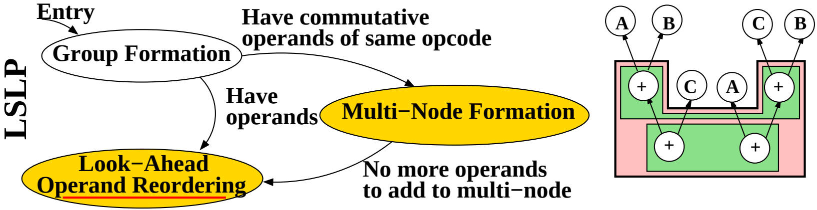




(L)SLP Graph Formation

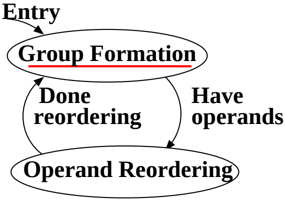


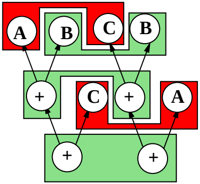
- LSLP: Extended DAG Formation with Multi-Nodes



(L)SLP Graph Formation

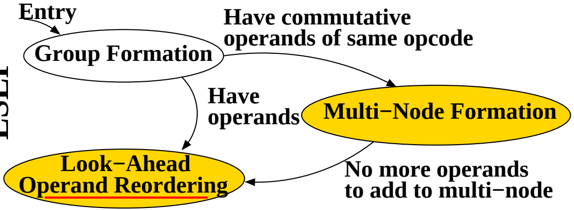
SLP

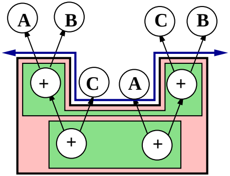




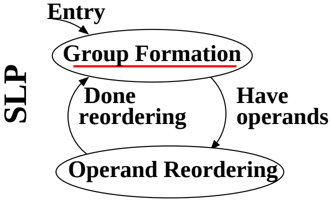
- LSLP: Extended DAG Formation with Multi-Nodes

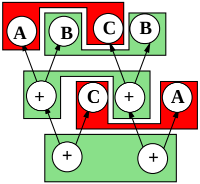
LSLP



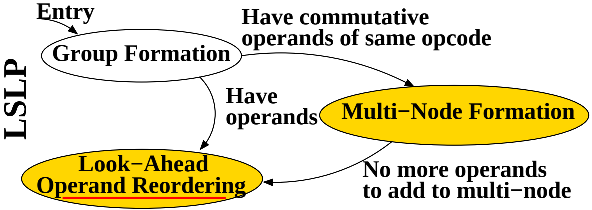


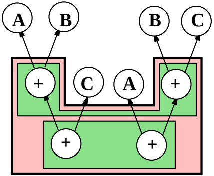
(L)SLP Graph Formation



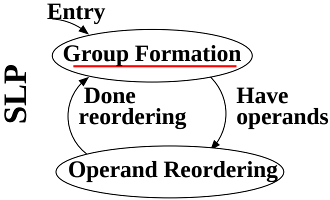


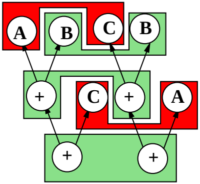
- LSLP: Extended DAG Formation with Multi-Nodes



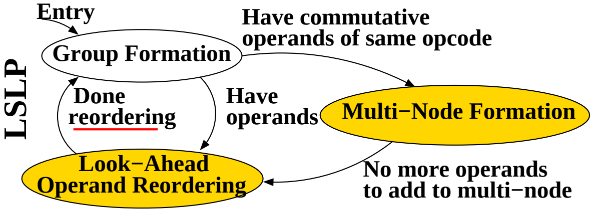


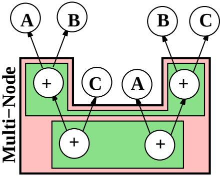
(L)SLP Graph Formation





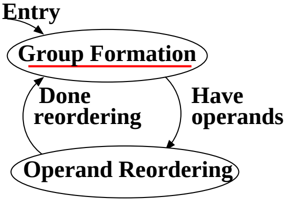
- LSLP: Extended DAG Formation with Multi-Nodes

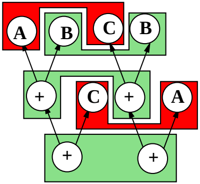




(L)SLP Graph Formation

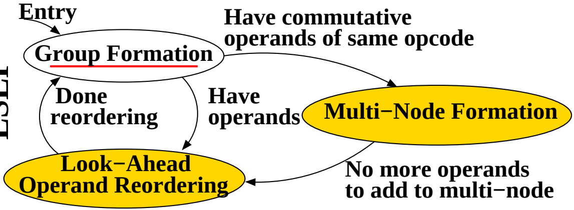
SLP

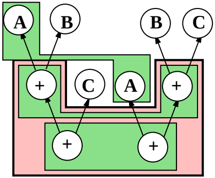




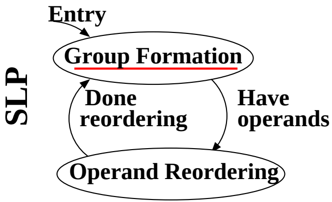
- LSLP: Extended DAG Formation with Multi-Nodes

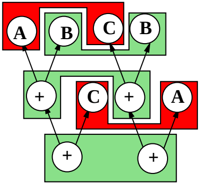
LSLP



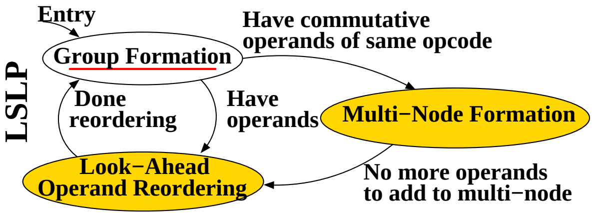


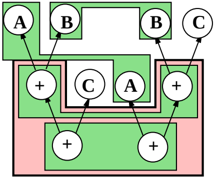
(L)SLP Graph Formation



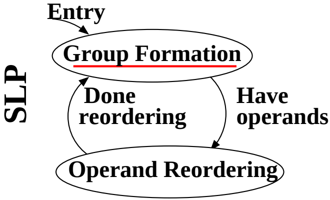


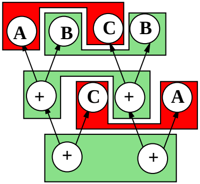
- LSLP: Extended DAG Formation with Multi-Nodes



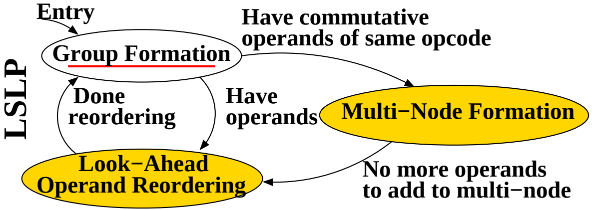


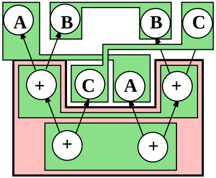
(L)SLP Graph Formation



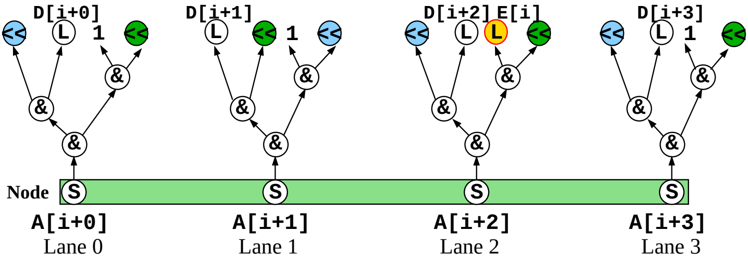


- LSLP: Extended DAG Formation with Multi-Nodes

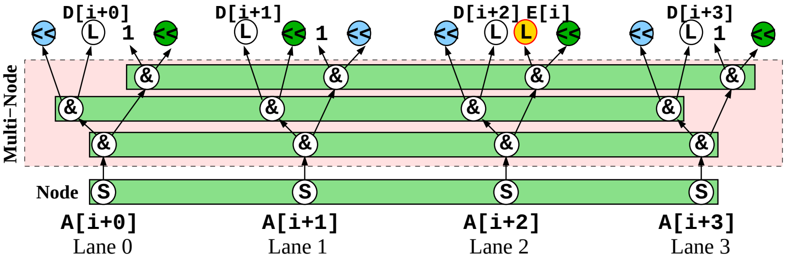




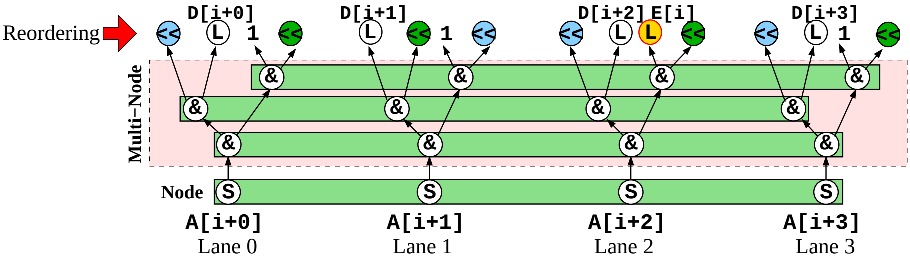
Operand Reordering with Look-Ahead



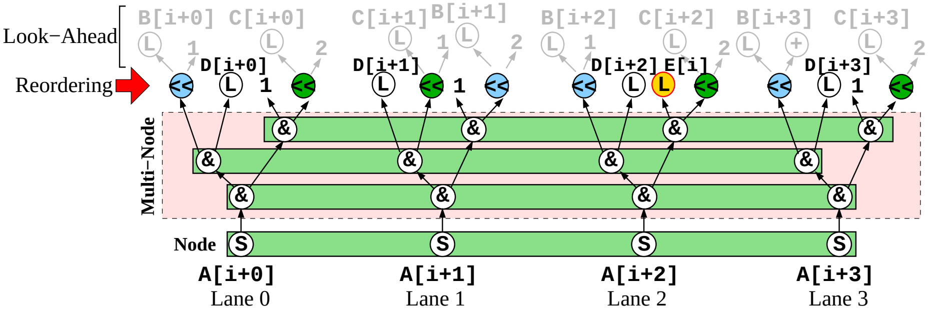
Operand Reordering with Look-Ahead



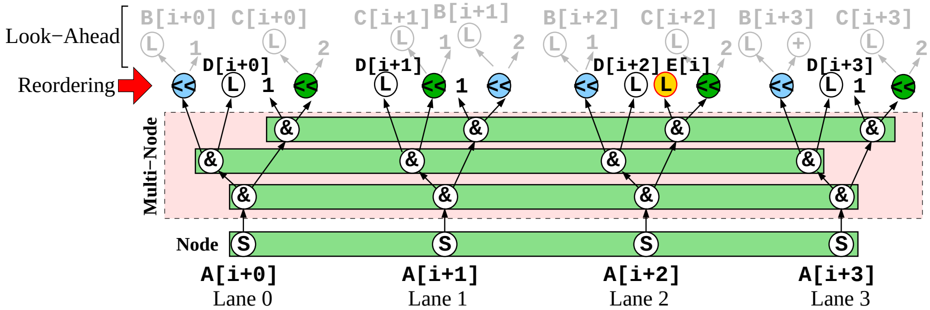
Operand Reordering with Look-Ahead



Operand Reordering with Look-Ahead

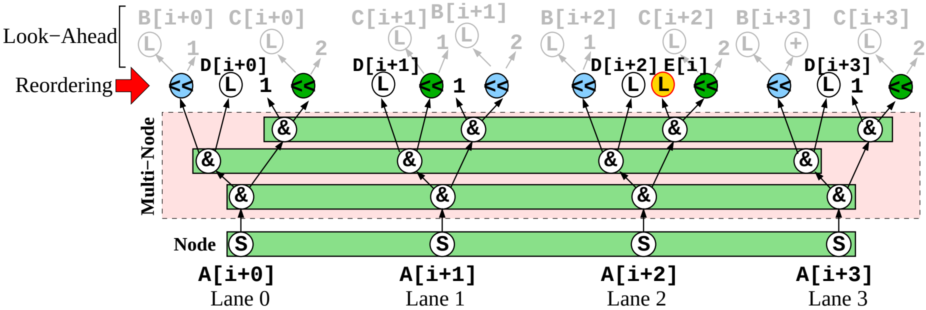


Operand Reordering with Look-Ahead



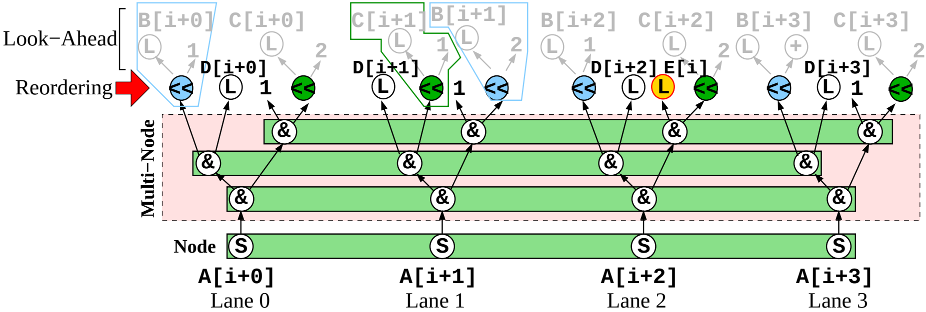
		Lane 0	Lane 1		Lane 2		Lane 3	
		final_order						
Operand Slots	0	<<						
	1	L						
	2	1						
	3	<<						

Operand Reordering with Look-Ahead



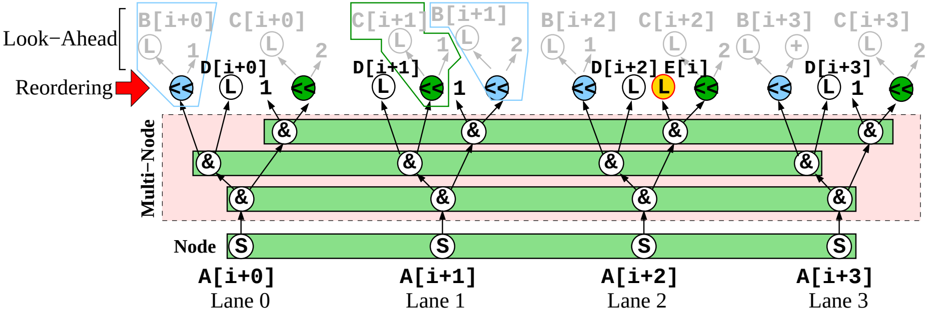
		Lane 0	Lane 1		Lane 2		Lane 3	
		final_order						
Operand Slots	0	<<						
	1	L						
	2	1						
	3	<<						

Operand Reordering with Look-Ahead



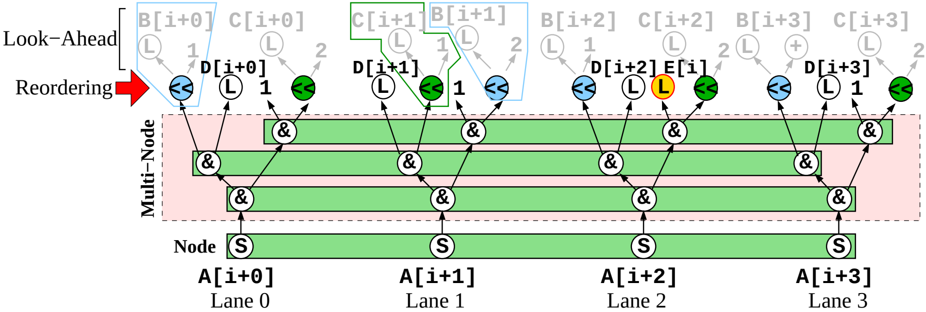
Operand Slots	Lane 0	Lane 1		Lane 2		Lane 3	
	final_order	Look−Ahead score		Look−Ahead score		Look−Ahead score	
0	<<	<< <<					
1	L						
2	1						
3	<<						

Operand Reordering with Look-Ahead



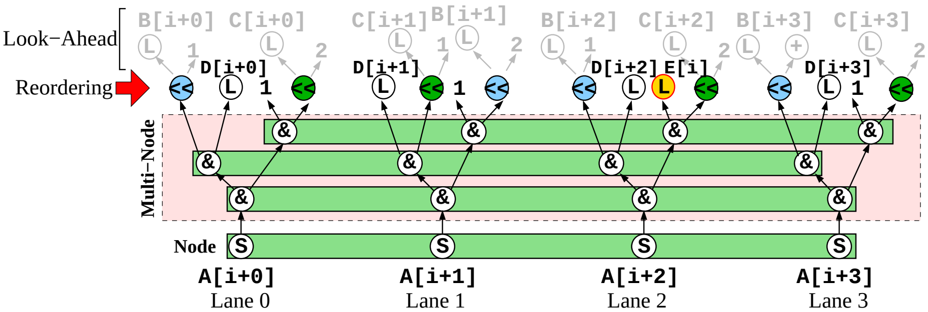
		Lane 0	Lane 1		Lane 2		Lane 3	
		final_order	Look−Ahead score		Look−Ahead score		Look−Ahead score	
Operand Slots	0	<<	<<:1 <<:2					
	1	L						
	2	1						
	3	<<						

Operand Reordering with Look-Ahead



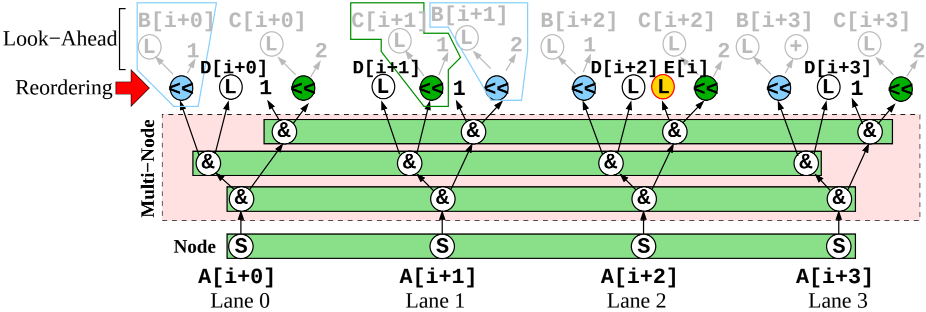
		Lane 0	Lane 1		Lane 2		Lane 3	
		final_order	Look−Ahead score		Look−Ahead score		Look−Ahead score	
Operand Slots	0	<<	<<:1 <<:2					
	1	L						
	2	1						
	3	<<						

Operand Reordering with Look-Ahead



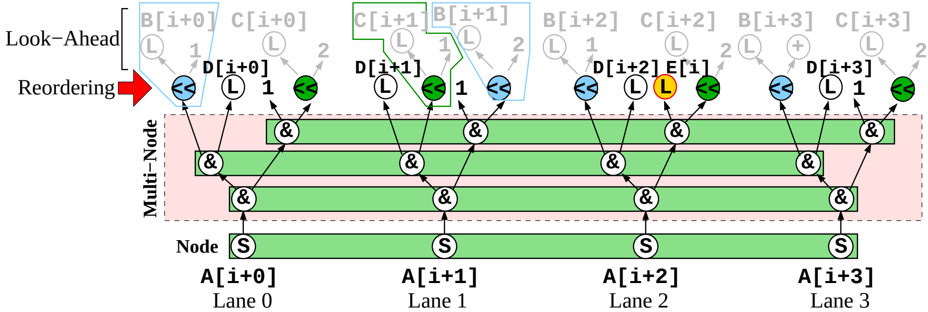
		Lane 0	Lane 1		Lane 2		Lane 3	
		final_order	Look−Ahead score		Look−Ahead score		Look−Ahead score	
Operand Slots	0	<<	<< :1	<< :2				
	1	L	L :N/A					
	2	1	1 :N/A					
	3	<<						

Operand Reordering with Look-Ahead



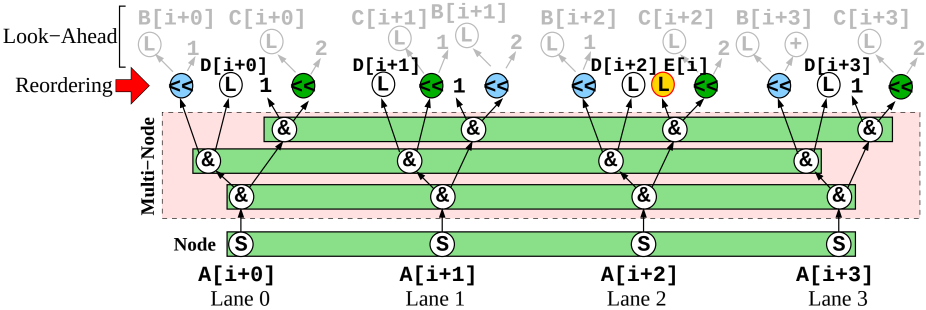
	Lane 0	Lane 1		Lane 2		Lane 3	
Operand Slots	final_order	Look−Ahead score		Look−Ahead score		Look−Ahead score	
0	<<	<<:1 <<:2					
1	L	L :N/A					
2	1	1 :N/A					
3	<<	<<:2 <<:1					

Operand Reordering with Look-Ahead



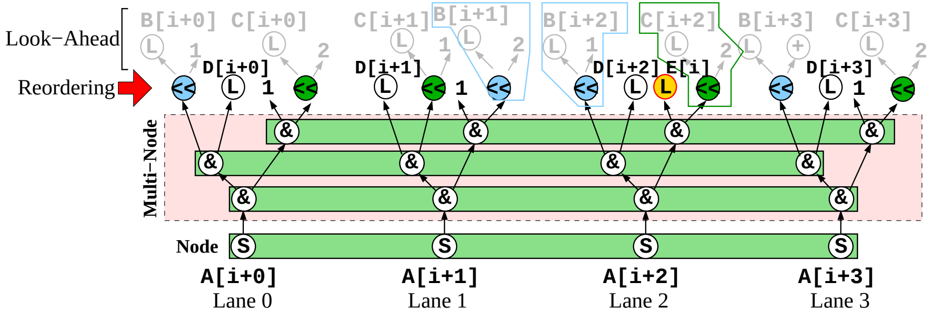
Operand Slots	Lane 0 final_order	Lane 1 Look−Ahead score	Lane 1	Lane 2 Look−Ahead score	Lane 2	Lane 3 Look−Ahead score	Lane 3
0	<<	<<:1 <<:2					
1	L	L :N/A					
2	1	1 :N/A					
3	<<	<<:2 <<:1					

Operand Reordering with Look-Ahead



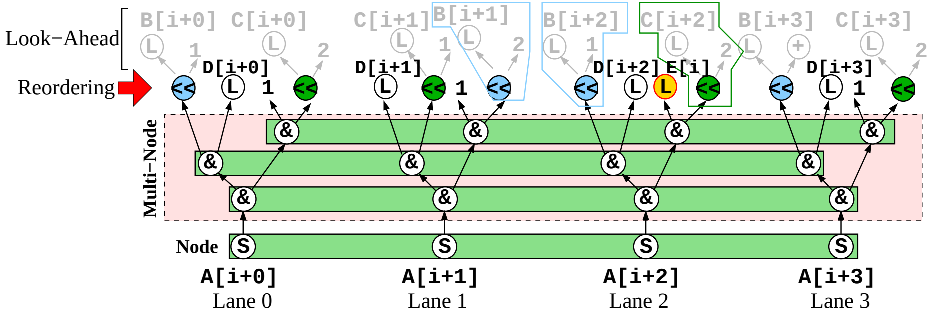
Operand Slots		Lane 0	Lane 1		Lane 2		Lane 3	
		final_order	Look−Ahead score	final_order	Look−Ahead score	final_order	Look−Ahead score	final_order
	0	<<	<<:1 <<:2	<<				
	1	L	L :N/A	L				
	2	1	1 :N/A	1				
	3	<<	<<:2 <<:1	<<				

Operand Reordering with Look-Ahead



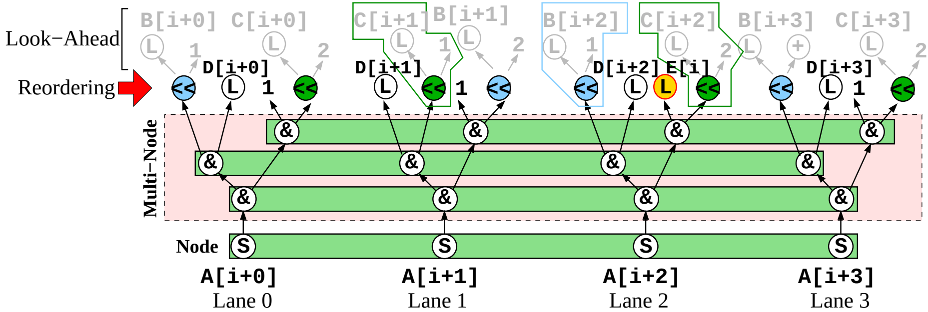
Operand Slots	Lane 0 final_order	Lane 1 Look−Ahead score	Lane 1 final_order	Lane 2 Look−Ahead score	Lane 2 final_order	Lane 3 Look−Ahead score	Lane 3 final_order
0	<<	<<:1 <<:2	<<	<<:2 <<:1			
1	L	L :N/A	L				
2	1	1 :N/A	1				
3	<<	<<:2 <<:1	<<				

Operand Reordering with Look-Ahead



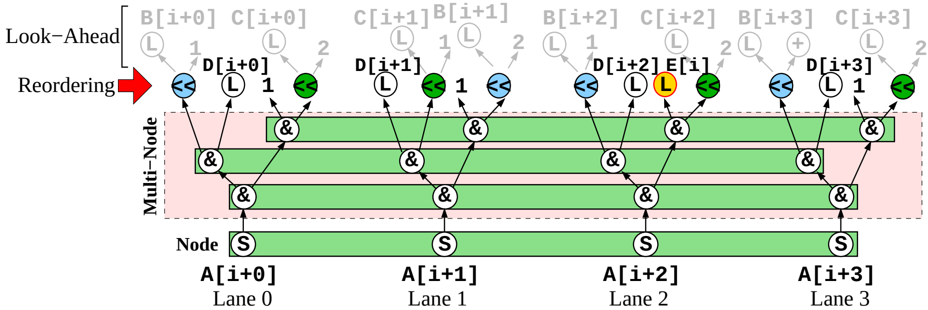
Operand Slots		Lane 0	Lane 1		Lane 2		Lane 3	
		final_order	Look−Ahead score	final_order	Look−Ahead score	final_order	Look−Ahead score	final_order
	0	<<	<<:1 <<:2	<<	<<:2 <<:1			
	1	L	L :N/A	L				
	2	1	1 :N/A	1				
	3	<<	<<:2 <<:1	<<				

Operand Reordering with Look-Ahead



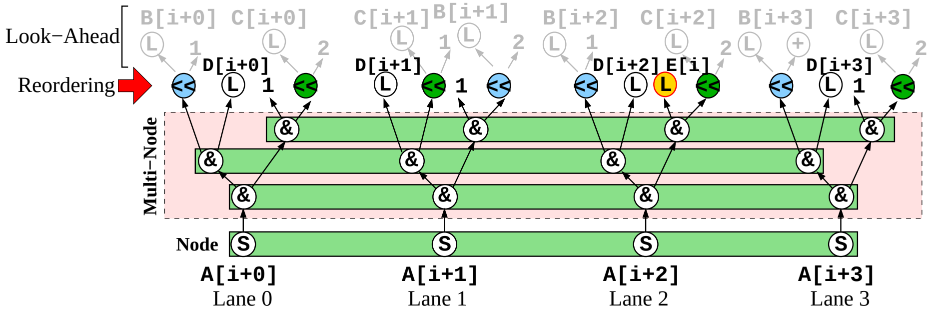
Operand Slots		Lane 0	Lane 1		Lane 2		Lane 3	
		final_order	Look−Ahead score	final_order	Look−Ahead score	final_order	Look−Ahead score	final_order
	0	<<	<<:1 <<:2	<<	<<:2 <<:1			
	1	L	L :N/A	L	L:N/A			
	2	1	1 :N/A	1	L:N/A			
	3	<<	<<:2 <<:1	<<	<<:1 <<:2			

Operand Reordering with Look-Ahead



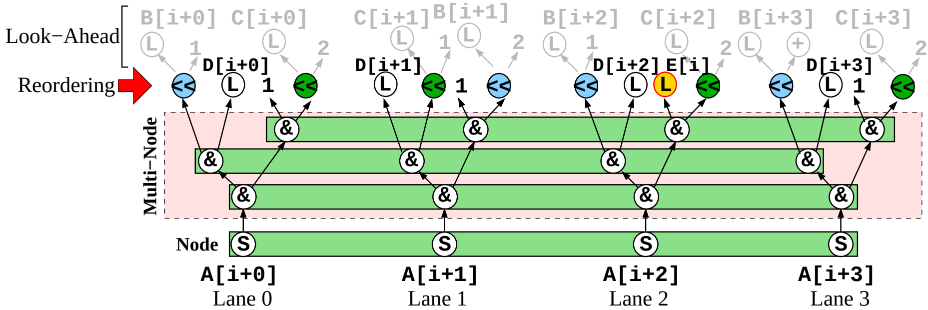
Operand Slots	Lane 0	Lane 1		Lane 2		Lane 3	
	final_order	Look−Ahead score	final_order	Look−Ahead score	final_order	Look−Ahead score	final_order
0	<<	<<:1 <<:2	<<	<<:2 <<:1	<<		
1	L	L :N/A	L	L:N/A	L		
2	1	1 :N/A	1	L:N/A	L		
3	<<	<<:2 <<:1	<<	<<:1 <<:2	<<		

Operand Reordering with Look-Ahead



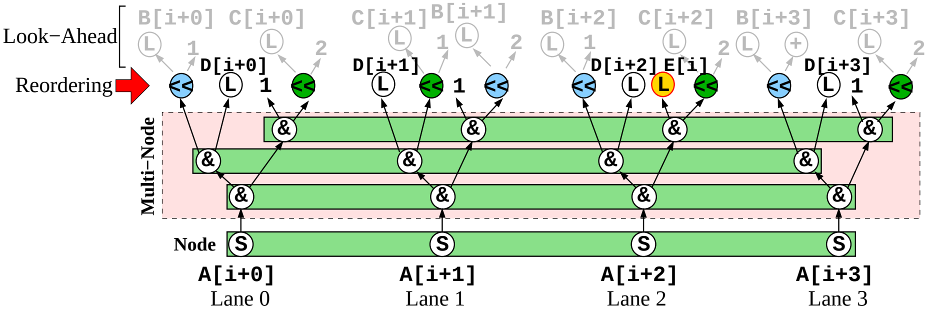
Operand Slots	Lane 0	Lane 1		Lane 2		Lane 3	
	final_order	Look−Ahead score	final_order	Look−Ahead score	final_order	Look−Ahead score	final_order
0	<<	<<:1 <<:2	<<	<<:2 <<:1	<<		
1	L	L :N/A	L	L:N/A	L		
2	1	1 :N/A	1	L:N/A	FAILED		
3	<<	<<:2 <<:1	<<	<<:1 <<:2	<<		

Operand Reordering with Look-Ahead



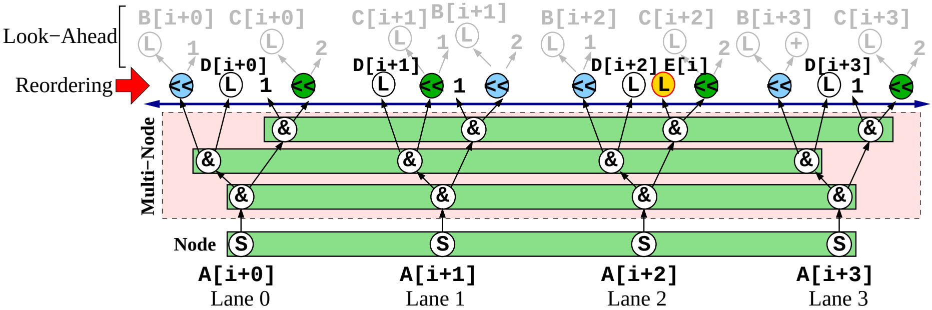
Operand Slots	Lane 0	Lane 1		Lane 2		Lane 3	
	final_order	Look−Ahead score	final_order	Look−Ahead score	final_order	Look−Ahead score	final_order
0	<<	<<:1 <<:2	<<	<<:2 <<:1	<<	<<:1 <<:1	
1	L	L :N/A	L	L:N/A	L	L:N/A	
2	1	1 :N/A	1	L:N/A	FAILED	1 :N/A	
3	<<	<<:2 <<:1	<<	<<:1 <<:2	<<	<<:0 <<:2	

Operand Reordering with Look-Ahead



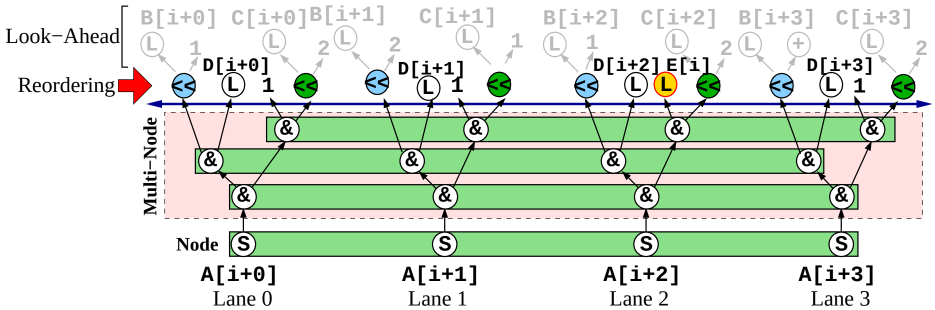
Operand Slots	Lane 0 final_order	Lane 1 Look−Ahead score	Lane 1 final_order	Lane 2 Look−Ahead score	Lane 2 final_order	Lane 3 Look−Ahead score	Lane 3 final_order
0	<<	<<:1 <<:2	<<	<<:2 <<:1	<<	<<:1 <<:1	<<
1	L	L :N/A	L	L:N/A	L	L:N/A	L
2	1	1 :N/A	1	L:N/A	FAILED	1 :N/A	1
3	<<	<<:2 <<:1	<<	<<:1 <<:2	<<	<<:0 <<:2	<<

Operand Reordering with Look-Ahead



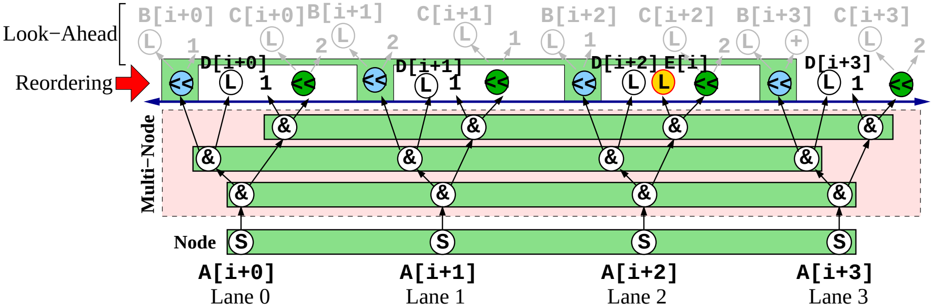
Operand Slots	Lane 0	Lane 1		Lane 2		Lane 3	
	final_order	Look−Ahead score	final_order	Look−Ahead score	final_order	Look−Ahead score	final_order
0	<<	<<:1 <<:2	<<	<<:2 <<:1	<<	<<:1 <<:1	<<
1	L	L :N/A	L	L:N/A	L	L:N/A	L
2	1	1 :N/A	1	L:N/A	FAILED	1 :N/A	1
3	<<	<<:2 <<:1	<<	<<:1 <<:2	<<	<<:0 <<:2	<<

Operand Reordering with Look-Ahead



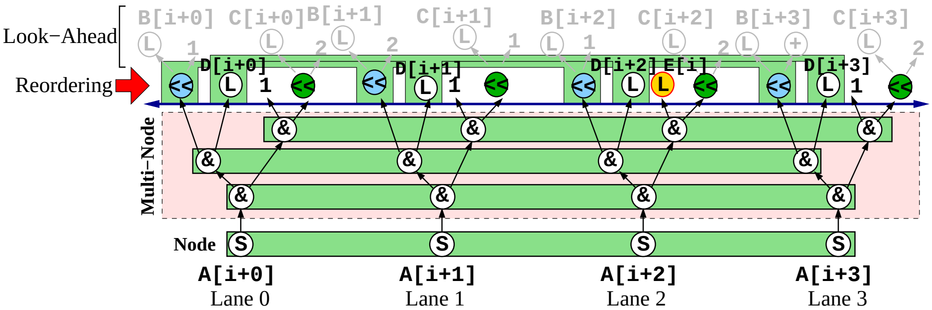
		Lane 0	Lane 1		Lane 2		Lane 3	
Operand Slots		final_order	Look−Ahead score	final_order	Look−Ahead score	final_order	Look−Ahead score	final_order
	0	<<	<<:1 <<:2	<<	<<:2 <<:1	<<	<<:1 <<:1	<<
	1	L	L :N/A	L	L:N/A	L	L:N/A	L
	2	1	1 :N/A	1	L:N/A	FAILED	1 :N/A	1
	3	<<	<<:2 <<:1	<<	<<:1 <<:2	<<	<<:0 <<:2	<<

Operand Reordering with Look-Ahead



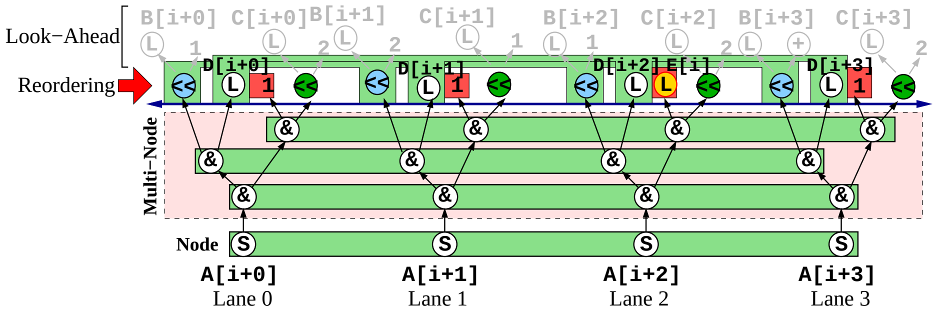
Operand Slots	Lane 0 final_order	Lane 1 Look−Ahead score	Lane 1 final_order	Lane 2 Look−Ahead score	Lane 2 final_order	Lane 3 Look−Ahead score	Lane 3 final_order
0	<<	<<:1 <<:2	<<	<<:2 <<:1	<<	<<:1 <<:1	<<
1	L	L :N/A	L	L:N/A	L	L:N/A	L
2	1	1 :N/A	1	L:N/A	FAILED	1 :N/A	1
3	<<	<<:2 <<:1	<<	<<:1 <<:2	<<	<<:0 <<:2	<<

Operand Reordering with Look-Ahead



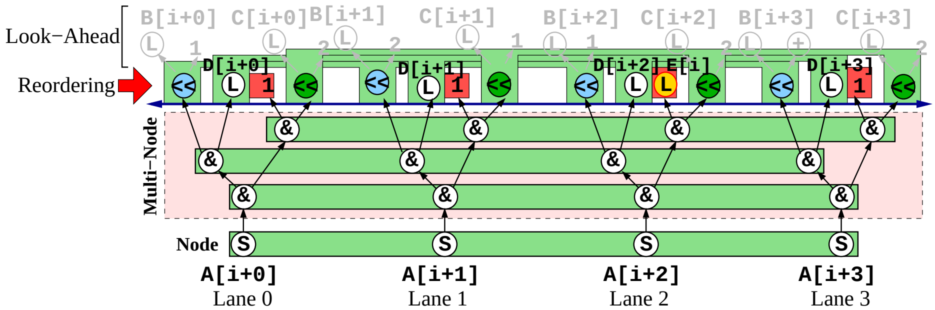
		Lane 0	Lane 1		Lane 2		Lane 3	
		final_order	Look−Ahead score	final_order	Look−Ahead score	final_order	Look−Ahead score	final_order
Operand Slots	0	<<	<<:1 <<:2	<<	<<:2 <<:1	<<	<<:1 <<:1	<<
	1	L	L :N/A	L	L:N/A	L	L:N/A	L
	2	1	1 :N/A	1	L:N/A	FAILED	1 :N/A	1
	3	<<	<<:2 <<:1	<<	<<:1 <<:2	<<	<<:0 <<:2	<<

Operand Reordering with Look-Ahead



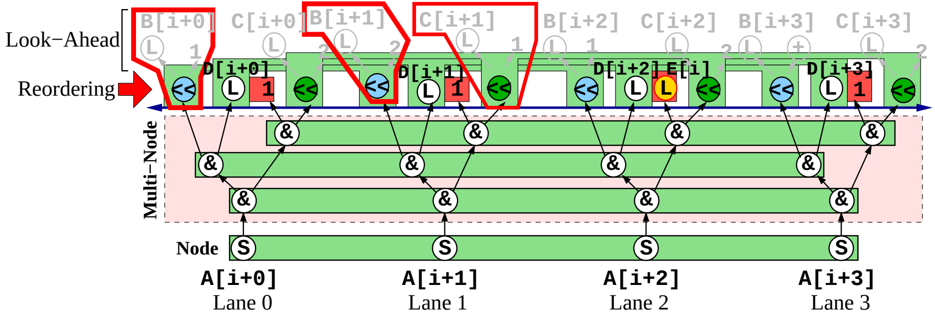
Operand Slots	Lane 0 final_order	Lane 1 Look−Ahead score	Lane 1 final_order	Lane 2 Look−Ahead score	Lane 2 final_order	Lane 3 Look−Ahead score	Lane 3 final_order
0	<<	<<:1 <<:2	<<	<<:2 <<:1	<<	<<:1 <<:1	<<
1	L	L :N/A	L	L:N/A	L	L:N/A	L
2	1	1 :N/A	1	L:N/A	FAILED	1 :N/A	1
3	<<	<<:2 <<:1	<<	<<:1 <<:2	<<	<<:0 <<:2	<<

Operand Reordering with Look-Ahead



Operand Slots	Lane 0	Lane 1		Lane 2		Lane 3	
	final_order	Look−Ahead score	final_order	Look−Ahead score	final_order	Look−Ahead score	final_order
0	<<	<<:1 <<:2	<<	<<:2 <<:1	<<	<<:1 <<:1	<<
1	L	L :N/A	L	L:N/A	L	L:N/A	L
2	1	1 :N/A	1	L:N/A	FAILED	1 :N/A	1
3	<<	<<:2 <<:1	<<	<<:1 <<:2	<<	<<:0 <<:2	<<

Operand Reordering with Look-Ahead

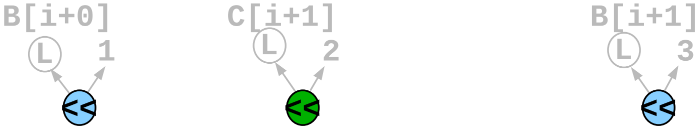


Operand Slots	Lane 0	Lane 1		Lane 2		Lane 3	
	final_order	Look−Ahead score	final_order	Look−Ahead score	final_order	Look−Ahead score	final_order
0	<<	<<:1 <<:2	<<	<<:2 <<:1	<<	<<:1 <<:1	<<
1	L	L :N/A	L	L:N/A	L	L:N/A	L
2	1	1 :N/A	1	L:N/A	FAILED	1 :N/A	1
3	<<	<<:2 <<:1	<<	<<:1 <<:2	<<	<<:0 <<:2	<<

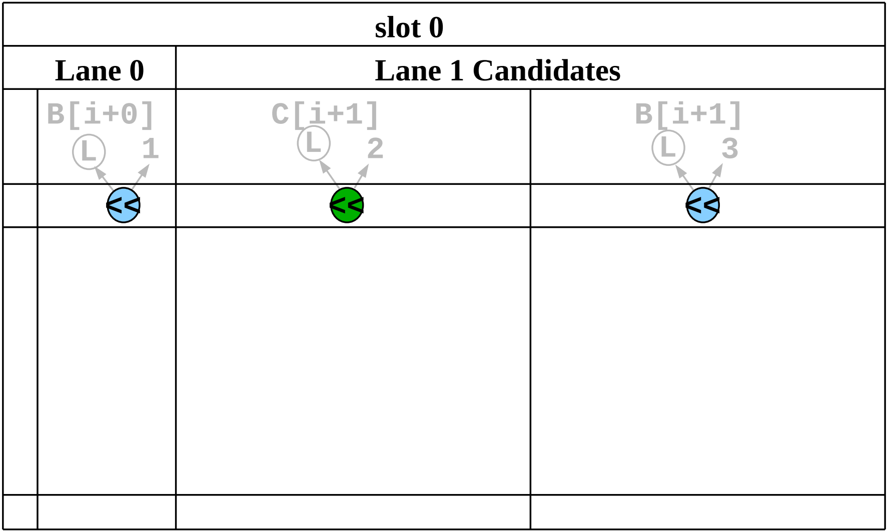
Score Calculation with Look-Ahead

slot 0

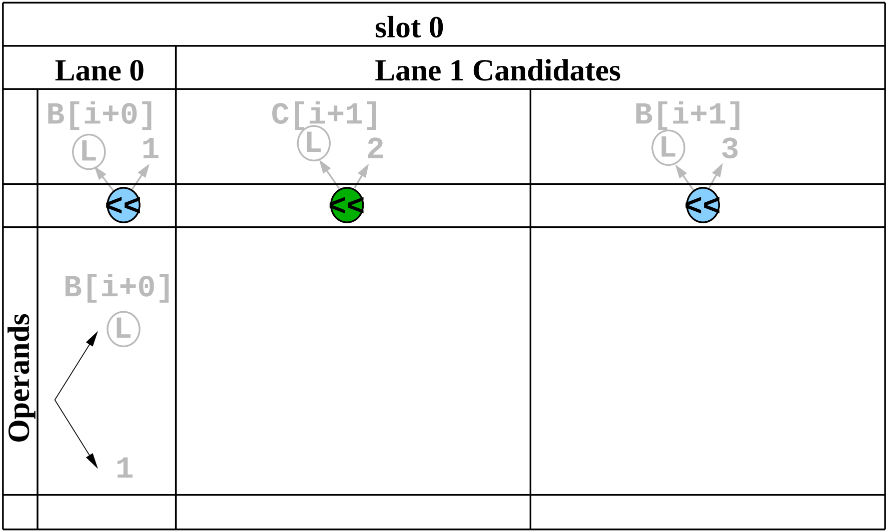
Lane 0 | **Lane 1 Candidates**



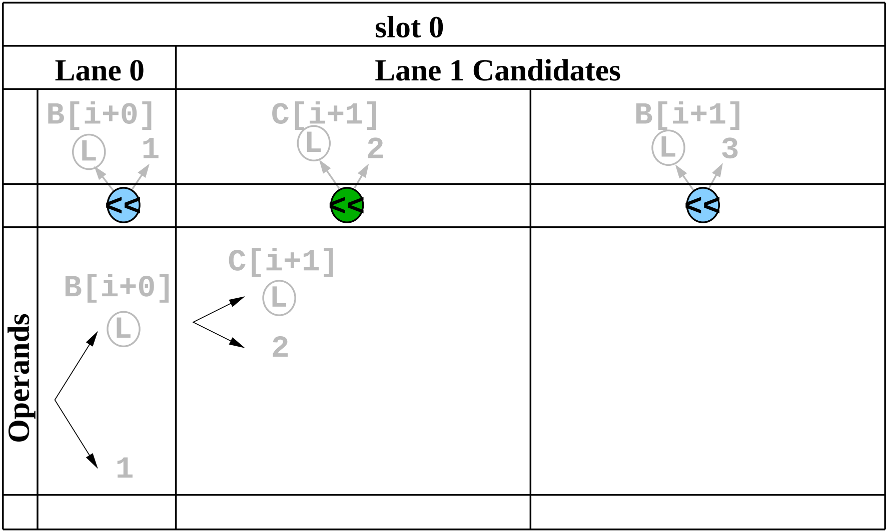
Score Calculation with Look-Ahead



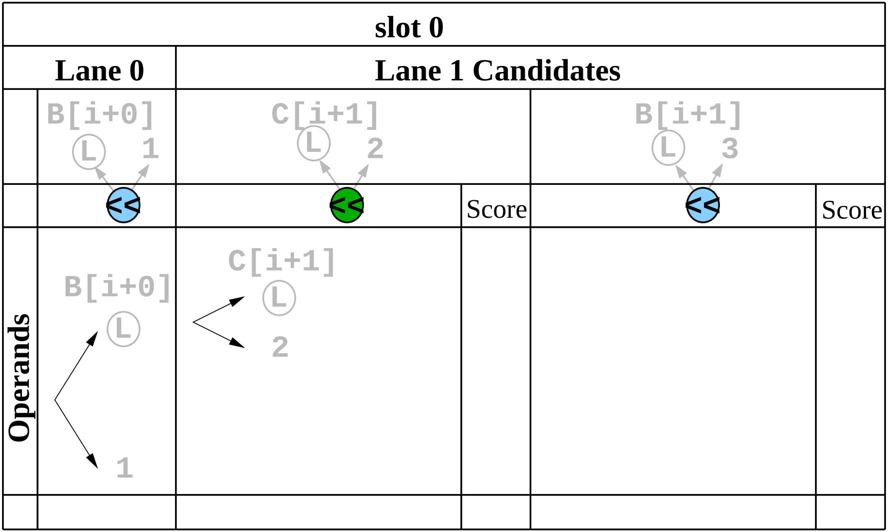
Score Calculation with Look-Ahead



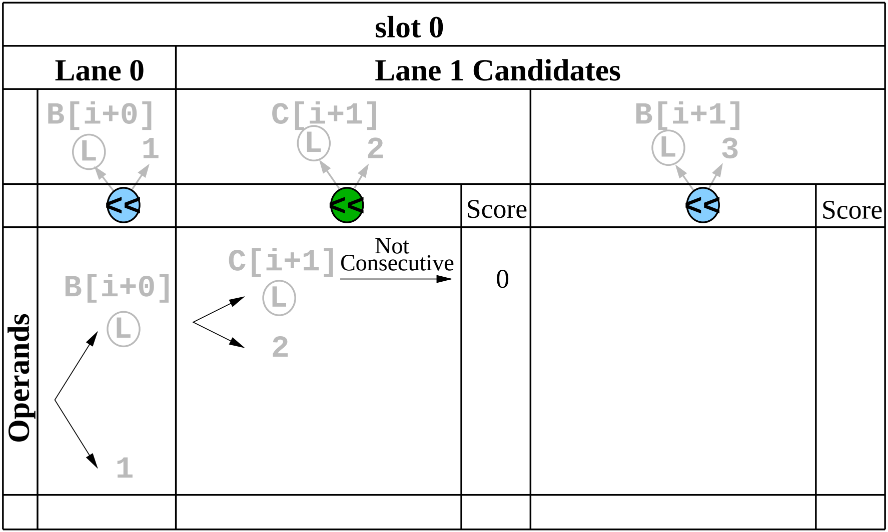
Score Calculation with Look-Ahead



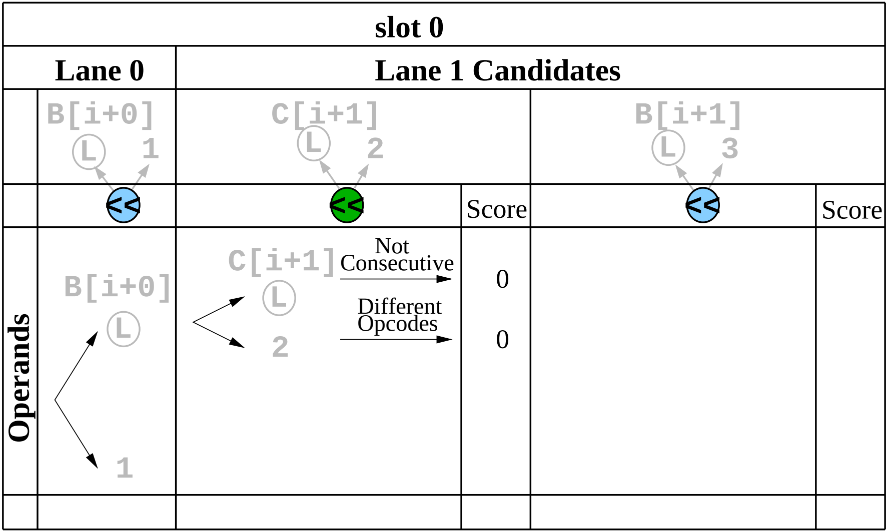
Score Calculation with Look-Ahead



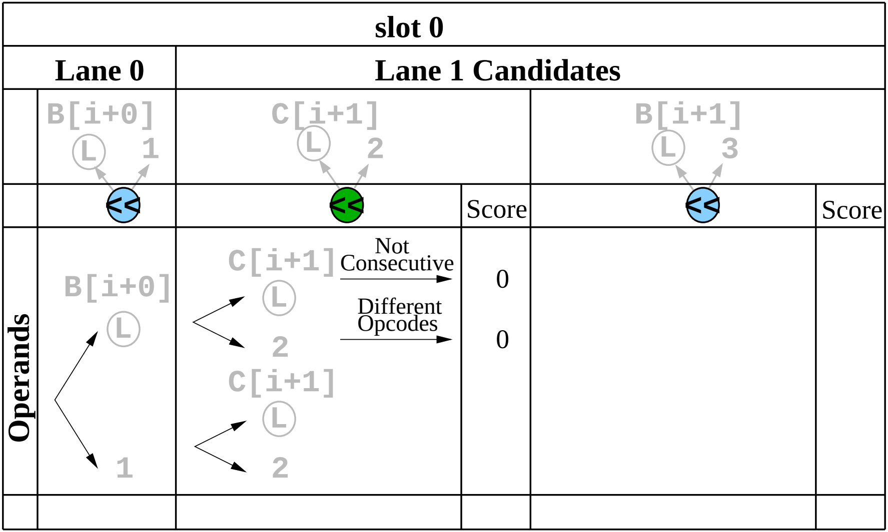
Score Calculation with Look-Ahead



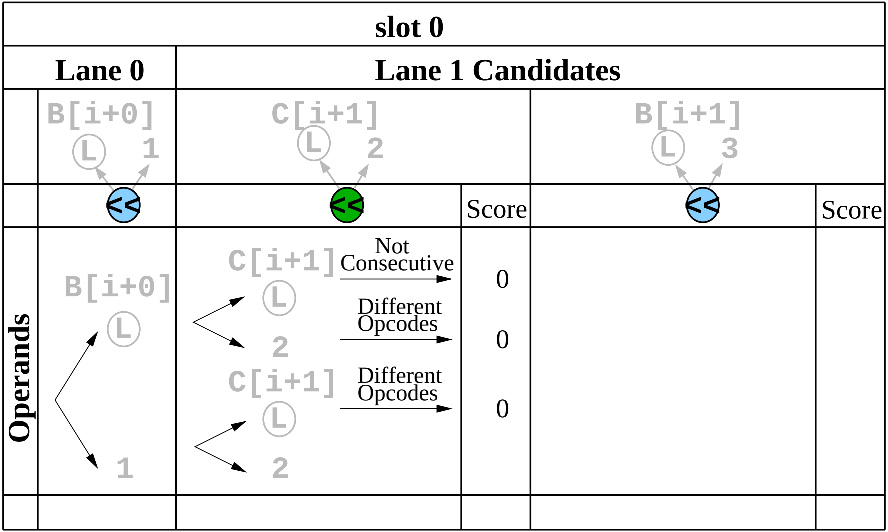
Score Calculation with Look-Ahead



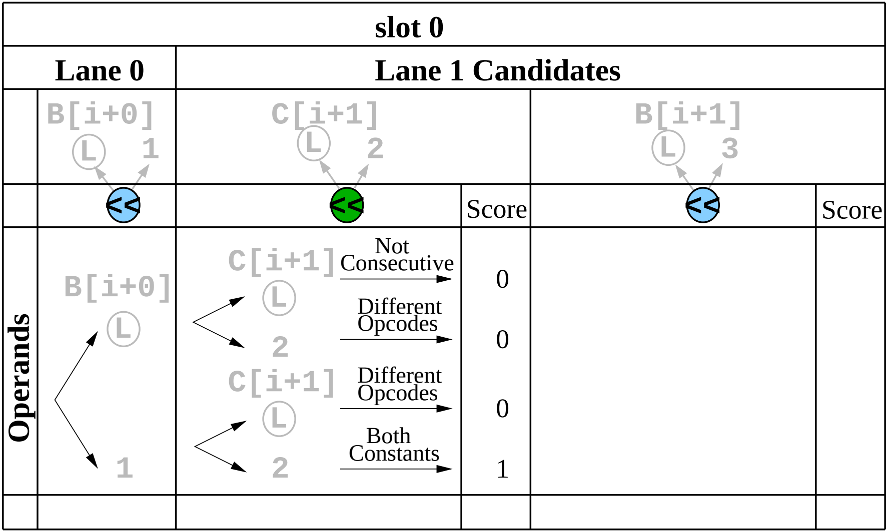
Score Calculation with Look-Ahead



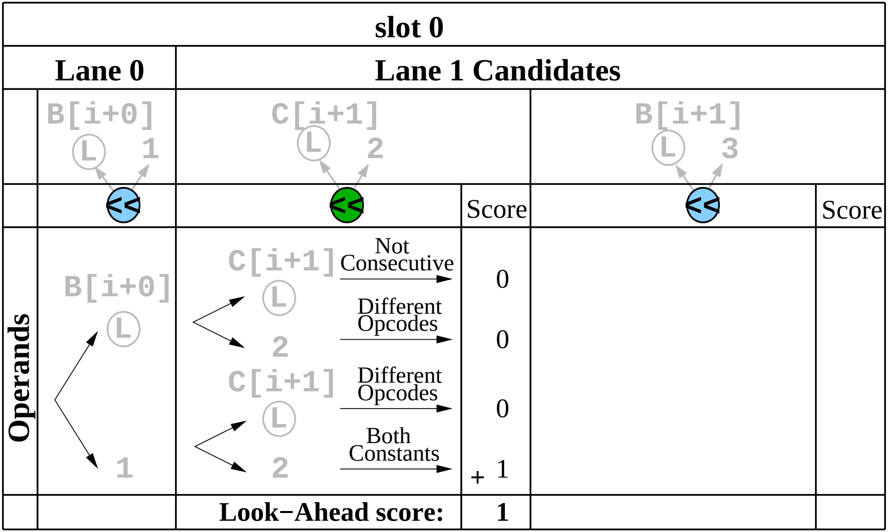
Score Calculation with Look-Ahead



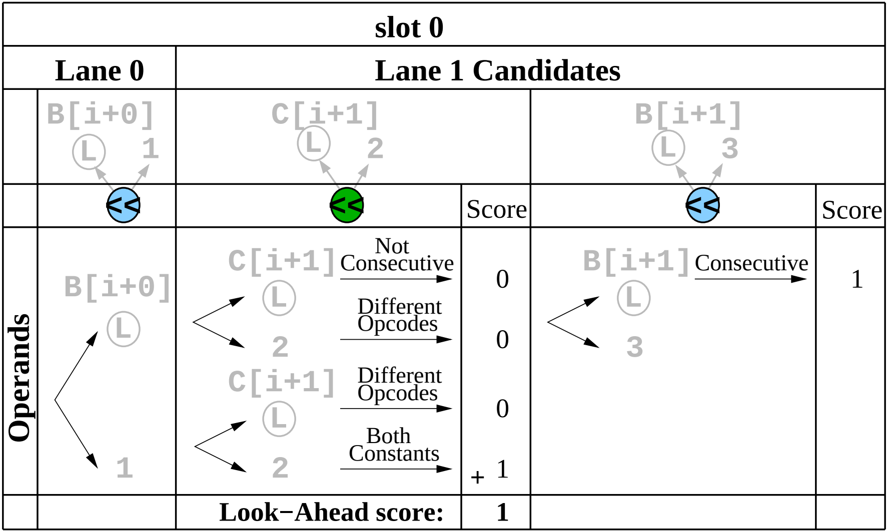
Score Calculation with Look-Ahead



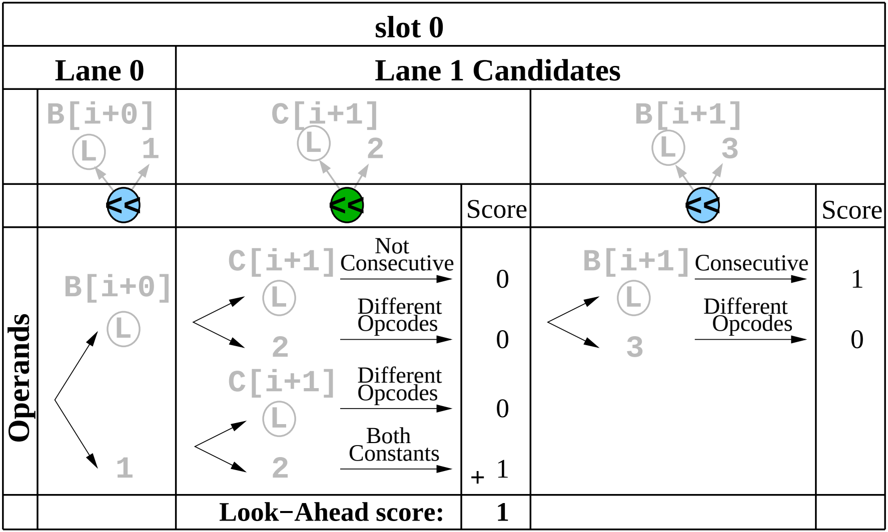
Score Calculation with Look-Ahead



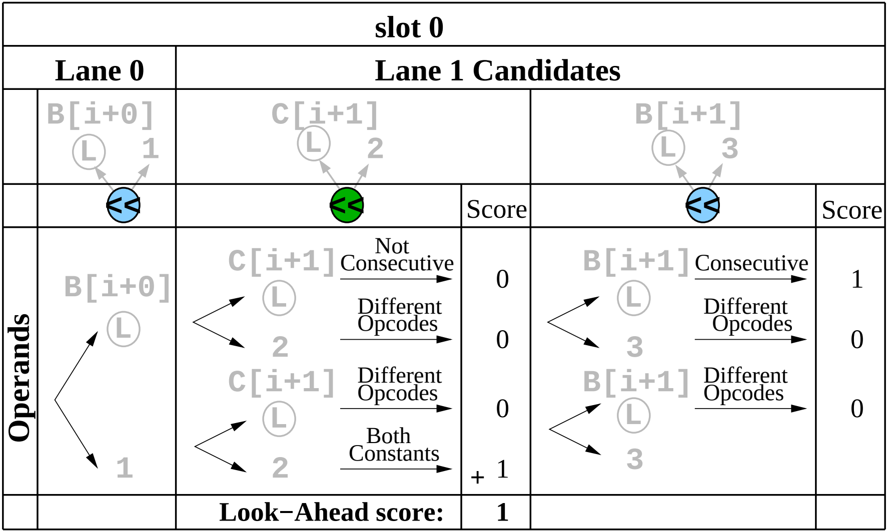
Score Calculation with Look-Ahead



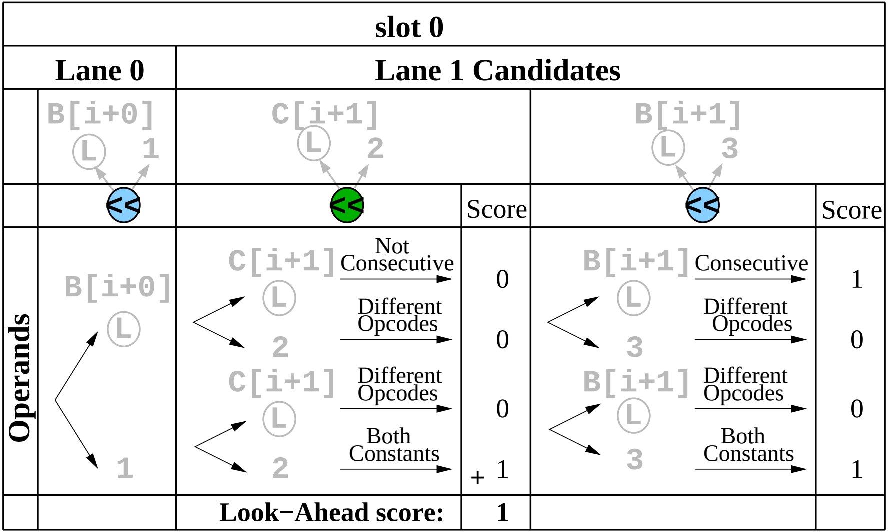
Score Calculation with Look-Ahead



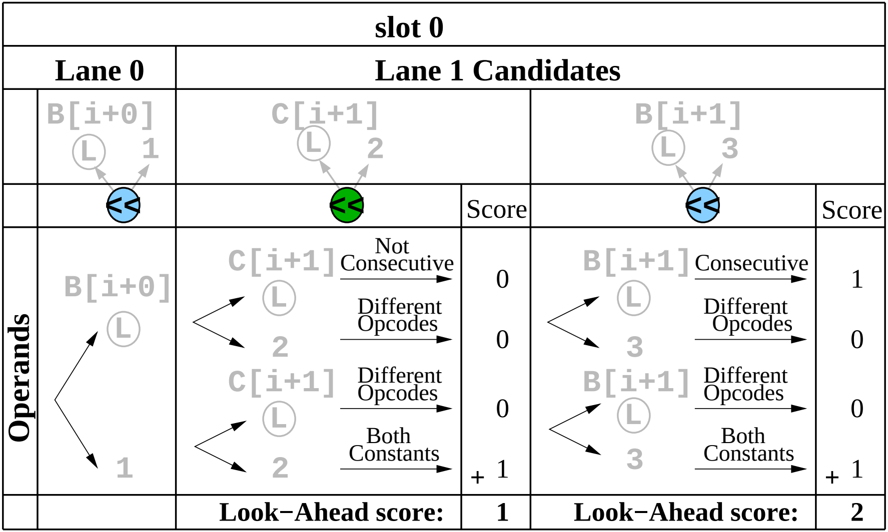
Score Calculation with Look-Ahead



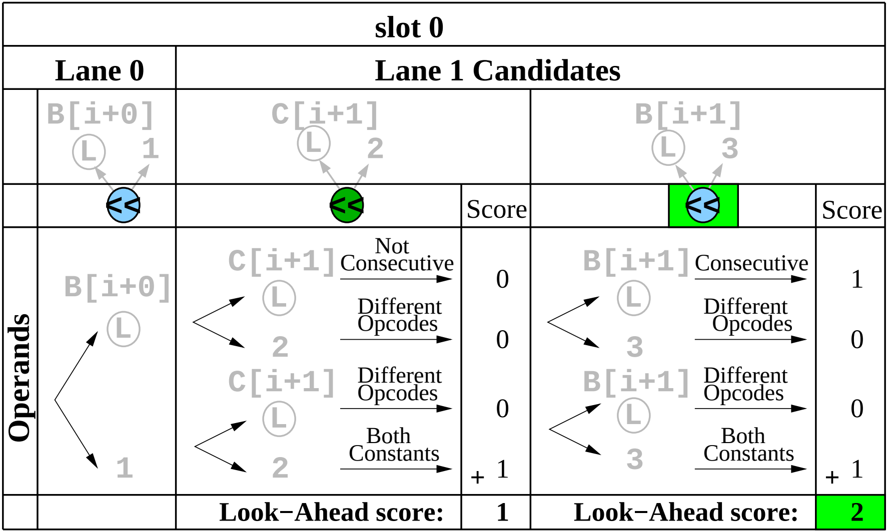
Score Calculation with Look-Ahead



Score Calculation with Look-Ahead



Score Calculation with Look-Ahead



Experimental Setup

- Implemented LSLP in LLVM 4.0

Experimental Setup

- Implemented LSLP in LLVM 4.0
- Target: Intel Core i5-6440HQ Skylake CPU

Experimental Setup

- Implemented LSLP in LLVM 4.0
- Target: Intel Core i5-6440HQ Skylake CPU
- Compiler flags: -O3 -ffast-math -mavx2 -march=skylake -mtune=skylake

Experimental Setup

- Implemented LSLP in LLVM 4.0
- Target: Intel Core i5-6440HQ Skylake CPU
- Compiler flags: -O3 -ffast-math -mavx2 -march=skylake -mtune=skylake
- Kernels from SPEC CPU2006
- We evaluated the following cases:

Experimental Setup

- Implemented LSLP in LLVM 4.0
- Target: Intel Core i5-6440HQ Skylake CPU
- Compiler flags: -O3 -ffast-math -mavx2 -march=skylake -mtune=skylake
- Kernels from SPEC CPU2006
- We evaluated the following cases:
  1. All loop, SLP and LSLP vectorizers disabled (O3)

Experimental Setup

- Implemented LSLP in LLVM 4.0
- Target: Intel Core i5-6440HQ Skylake CPU
- Compiler flags: -O3 -ffast-math -mavx2 -march=skylake -mtune=skylake
- Kernels from SPEC CPU2006
- We evaluated the following cases:
  1. All loop, SLP and LSLP vectorizers disabled (O3)
  2. O3 + SLP enabled but No Reordering (SLP-NR)

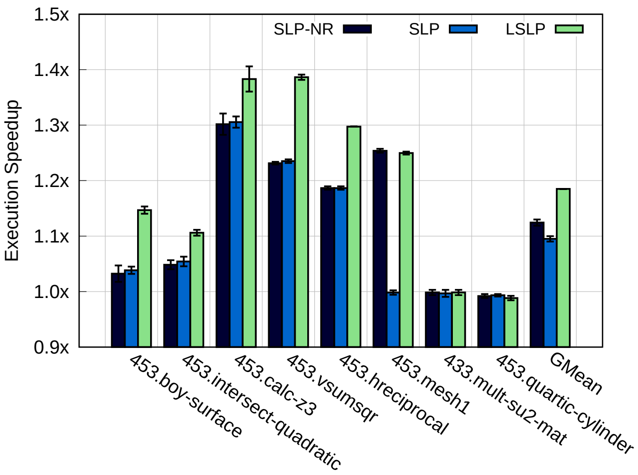
Experimental Setup

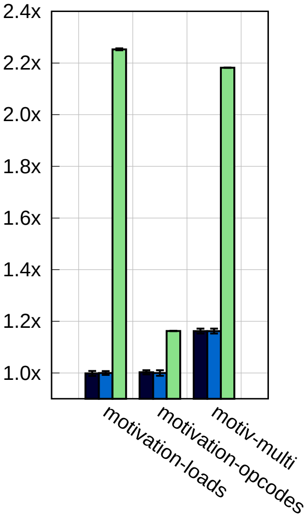
- Implemented LSLP in LLVM 4.0
- Target: Intel Core i5-6440HQ Skylake CPU
- Compiler flags: -O3 -ffast-math -mavx2 -march=skylake -mtune=skylake
- Kernels from SPEC CPU2006
- We evaluated the following cases:
  1. All loop, SLP and LSLP vectorizers disabled (O3)
  2. O3 + SLP enabled but No Reordering (SLP-NR)
  3. O3 + SLP enabled (SLP)

Experimental Setup

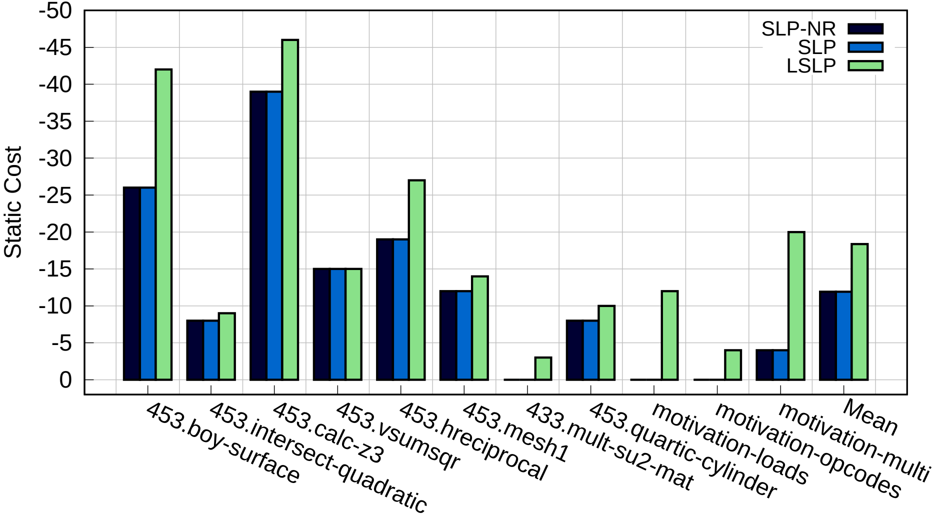
- Implemented LSLP in LLVM 4.0
- Target: Intel Core i5-6440HQ Skylake CPU
- Compiler flags: -O3 -ffast-math -mavx2 -march=skylake -mtune=skylake
- Kernels from SPEC CPU2006
- We evaluated the following cases:
  1. All loop, SLP and LSLP vectorizers disabled (O3)
  2. O3 + SLP enabled but No Reordering (SLP-NR)
  3. O3 + SLP enabled (SLP)
  4. O3 + LSLP enabled (LSLP)

Performance (normalized to O3)



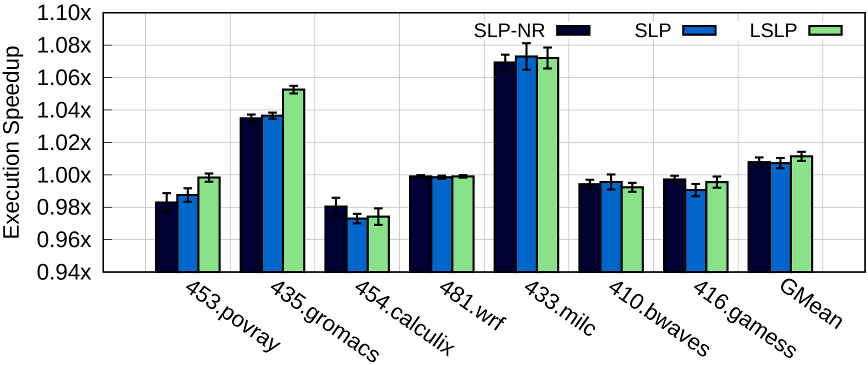


Static Cost (the higher the better)



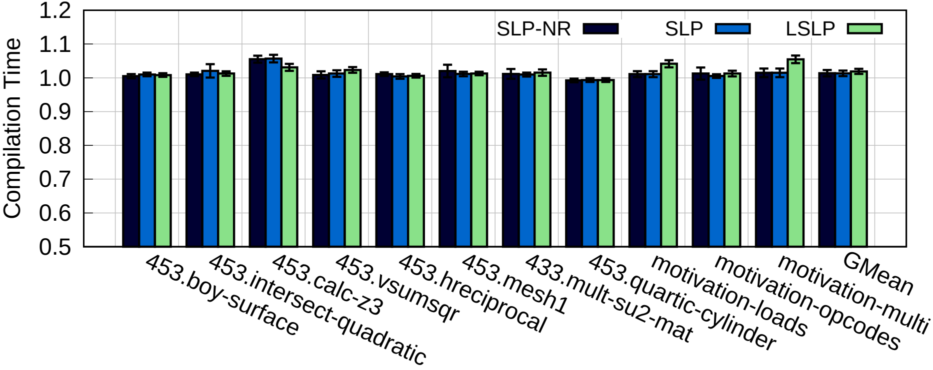
Performance (Full Benchmarks)

- About 1% speedup in 453.povray and 435.gromacs



Total Compilation Time

- No significant difference in compilation time



Conclusion

- LSLP introduces an effective scheme for dealing with commutative operations. It is based on:

Conclusion

- LSLP introduces an effective scheme for dealing with commutative operations. It is based on:
  1. Smarter operand reordering with Look-Ahead score
  2. Forming Multi-Nodes of commutative operations and reordering across them

Conclusion

- LSLP introduces an effective scheme for dealing with commutative operations. It is based on:
  1. Smarter operand reordering with Look-Ahead score
  2. Forming Multi-Nodes of commutative operations and reordering across them
- Better at identifying isomorphism

Conclusion

- LSLP introduces an effective scheme for dealing with commutative operations. It is based on:
  1. Smarter operand reordering with Look-Ahead score
  2. Forming Multi-Nodes of commutative operations and reordering across them
- Better at identifying isomorphism
- Implemented in LLVM and evaluated on a real machine

Conclusion

- LSLP introduces an effective scheme for dealing with commutative operations. It is based on:
    1. Smarter operand reordering with Look-Ahead score
    2. Forming Multi-Nodes of commutative operations and reordering across them
- Better at identifying isomorphism
- Implemented in LLVM and evaluated on a real machine
- Improves performance and coverage

Conclusion

- LSLP introduces an effective scheme for dealing with commutative operations. It is based on:
  1. Smarter operand reordering with Look-Ahead score
  2. Forming Multi-Nodes of commutative operations and reordering across them
- Better at identifying isomorphism
- Implemented in LLVM and evaluated on a real machine
- Improves performance and coverage
- Similar compilation time