A Regularity-Driven Fast Gridless Detailed Router for High Frequency Datapath Designs

Sabyasachi Das
Design Technology
Intel Corporation
Santa Clara, CA 95054.
sabyasachi.das@intel.com

Sunil P. Khatri
Department of Electrical and Computer Engg
University of Colorado,
Boulder, CO 80309.
spkhatri@colorado.edu

ABSTRACT

We present a new detailed routing methodology specifically designed for datapath layouts. In typical state-of-the-art microprocessor designs, datapaths comprise about 70% of the logic (excluding caches). Although research on datapath placement and global routing has been reported, very little research has been reported in the area of detailed routing for datapaths.

Datapaths typically comprise regular *bit-slices* which are replicated. We define a *net-cluster*, which is collection of similarly structured nets present across different bit-slices. We introduce two clustering schemes (*Footprint-driven clustering* and *Instance-driven clustering*) to extract such net-clusters. Then, we perform a *strap-based* routing on exactly one member net of each net-cluster (in a single representative bit-slice). Next, for each net, we *propagate* its route to all other nets in its net-cluster. Our algorithm is unique in that it performs the detailed routing on a *single* bit-slice, and infers the routing for all bit-slices using the notion of net clusters.

We demonstrate at least 6$\times$ speed gains for industrial 32 and 64-bit datapath designs. The regularity of the routes across the bit-slices results in more predictable timing characteristics for the resulting datapath layout.

1. INTRODUCTION

As we migrate toward ultra deep sub-micron feature sizes, designs are becoming increasingly complex with very aggressive goals. Datapaths are one of the more critical parts of the design. It is well understood that traditional design automation methodologies are not well suited for the design of high-performance datapaths. As a result, datapath blocks are usually manually designed, resulting in a significantly larger design time and cost.

To solve this problem, researchers are actively trying to develop design automation methodologies which are suitable for the design of datapath circuits. For example, several datapath placement [1], [2] and synthesis [3] techniques have been reported. In [4], the authors introduce a datapath routing methodology. Their work differs from ours in that it uses probabilistic measures of congestion to guide the routing which is performed simultaneously for all nets. Results are reported on small designs, while our goal is to tackle very large industrial datapaths. To the best of our knowledge, there has been no other research on detailed routing for datapaths.

In this paper, we propose a new detailed routing methodology that exploits the regularity of connections in a datapath circuit. In our scheme, we route all the regular nets in a similar fashion so as to ensure good quality, regular routes. This results in highly predictable timing characteristics of the resulting design and the routing process is much faster than other conventional routers.

We have organized the rest of the paper as follows: Section 2 presents general characteristics and some definitions of a datapath. In Section 3, we discuss our proposed flow. Section 4 presents the advantages of our approach. Experimental results are provided in Section 5 and conclusions are drawn in Section 6.

2. CHARACTERISTICS OF DATAPATHS

Datapaths are commonly found in microprocessors, DSP and graphics ICs. In datapaths, same logic is repeated multiple times. We define a *bit-slice* as the logic corresponding to a particular bit. In practice, *N* bit-slices are abutted to obtain the design of an *N*-bit datapath. The layout width of all bit-slices is identical, and we call this the *bit-pitch* or *pitch*. The convention we follow for this paper is that the data flows vertically and control flows horizontally. In most standard-cell based datapath styles, each bit-slice is composed of multiple instances of standard cells (or larger master-cells).

3. OUR APPROACH

Figure 1 describes our overall flow. In following sub-sections, we discuss each step in detail.

3.1 Reading the Schematic Netlist

First, we read the schematic (logic) netlist of the whole block, which consists of several instances of library cells. Currently, our tool can handle only two levels of hierarchy. In the top of hierarchy, all the connections between the instances are specified. In the lower level, logical details of the library cells are specified.

3.2 Generating the Placement

Next, we place instances of the master-cells of the datapath block in a structured manner. In this work, we used an industrial datapath placement tool to produce a regular placement.

3.3 Reading the Layout Information of Cells

In this step, we read the layout information of the library cells that make up the datapath. In this step, we obtain details about the blockages present in the datapath block.

ISPD'01, April 1-4, 2001, Sonoma, California, USA.

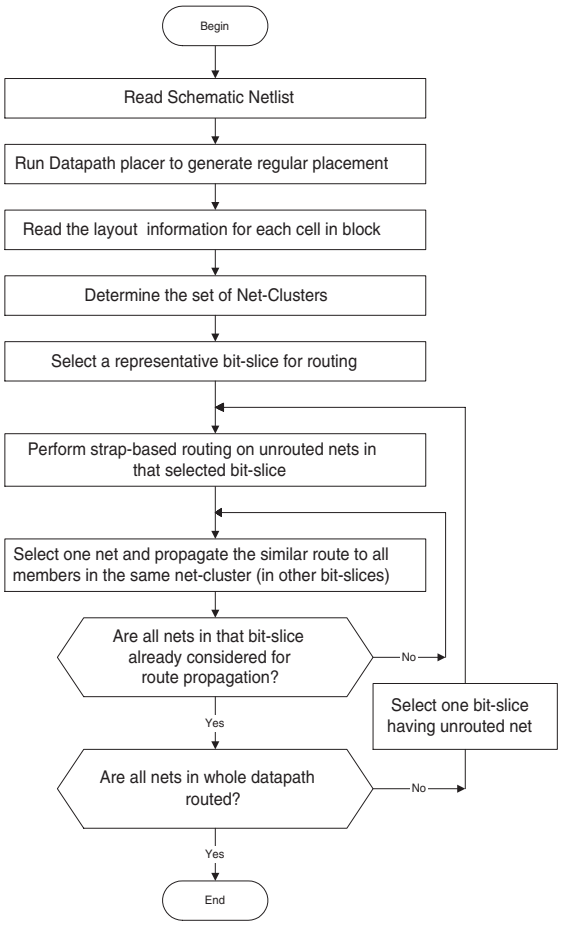

Figure 1: Overall Datapath Routing Flow

3.4 Extracting Net-Clusters

In a datapath block, several regular structures are present across multiple bit-slices. Techniques to extract regular instance structures have been proposed by Arikati et. al. [5] and Hassoun et. al [6].

In this paper, we extract regular *net structures* present in different bit-slices. We define a *net-cluster* as a collection of nets (spread over different bit-slices), in which all nets have similar connections. In particular if two nets net_1 and net_2 belong to same net-cluster, then net_1 and net_2 contain the same number of pins and for each pin p of net_1 with co-ordinates (x_p, y_p), there exists a pin q of net_2 with co-ordinates (x_q, y_q), such that

- $y_p = y_q$
- $|x_p - x_q| = k \cdot bit\_pitch \qquad (1 \leq k \leq N-1)$

To denote a net-cluster NC-1 with nets $N0, N1, N2, N3, N4$, we use the following notation: NC-1 = {N0, N1, N2, N3, N4} We have developed different algorithms to identify the net-clusters. The *footprint-driven clustering* algorithm creates net-clusters based on the names of pins, master-cells and nets in the datapath. This is supplemented by a powerful *instance-driven clustering algorithm*, which extracts clusters based on position information of the pins of nets. These techniques are described below:

3.4.1 Footprint-driven clustering (FDC)

In general, datapath designers follow a very regular naming style, in order to effectively manage and debug the datapath design. The Footprint-driven clustering exploits this naming regularity. Figure 2 shows a 4-bit datapath which follows a regular naming scheme. We define the *global footprint* of a net as a string which is created by lexicographically concatenating the names of the net pins (of the connecting instances) and names of master-cells of those instances. The *detailed footprint* of a net is defined as a string that is created by lexicographically concatenating the names of all the connecting instances and the name of the net. Footprint-driven clustering is described in Algorithm 1. Detailed comments are provided below.

Algorithm 1 : Footprint-Driven Clustering

```
NetNames = findAllNetNames(designName)
GFs = findGlobalFootPrints(NetNames)
AllGroupsOfNets = NULL
for each unique global footprint (ugf) do
   NewGroups = getGroupsOfNets(ugf)
   AllGroupsOfNets = AllGroupsOfNets + NewGroups
end for
AllNetClusters = NULL
for each Group in AllGroupsOfNets do
   NetsInGroup = findNetsInGroup(Group)
   DFs = findDetailedFootPrints(NetsInGroup)
   NewNetClusters = CreateNetCluster(DFs, NetsInGroup)
   AllNetClusters = AllNetCluster + NewNetClusters
end for
Return AllNetClusters
```

- In the first part, we calculate global footprints for all nets. We form a number of groups of nets, such that all nets in a single group have the same global footprint and no two nets belonging to different groups have same global footprint.
- In the next part, our target is to create one or more net-clusters from each group. We generate detailed footprints for each member net in a group. If the indices of the names in the detailed footprints of two nets differ by a constant k, then these two nets belong to a single net-cluster. Otherwise these two nets belong to different net-clusters.

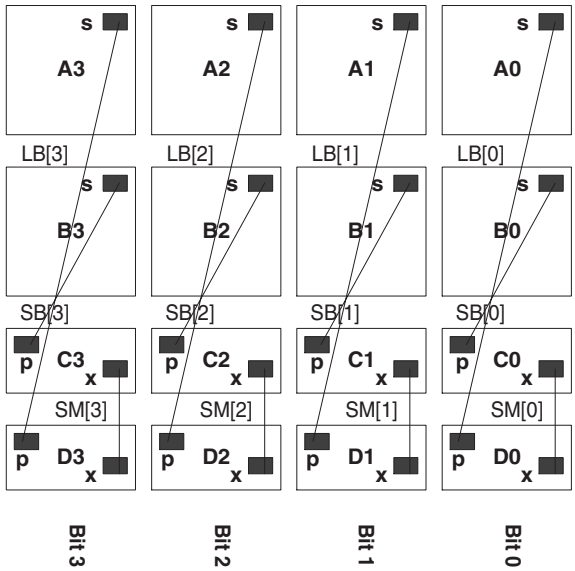

Figure 2: Footprint-Driven Clustering

We illustrate FDC by applying Algorithm 1 on the design shown in Figure 2. We assume that $A3, A2, A1, A0$ belong to same master-cell (say M_1). $B3, B2, B1, B0$ also belong to the master-cell M_1.

Let $C3, C2, C1, C0$ and $D3, D2, D1, D0$ all belong to master-cell *M_2*. The nets in bit-slice 3 are as follows. Net $LB[3]$ connects pin *s* of *A*3 to pin *p* of *D*3, net $SB[3]$ connects pin *s* of *B*3 to pin *p* of *C*3 and net $SM[3]$ connects pin *x* of *C*3 to pin *x* of *D*3.

- After running the first step of FDC algorithm on the design of Figure 2, we get two groups. *Group-1* has nets LB[3], LB[2], LB[1], LB[0], SB[3], SB[2], SB[1], SB[0]. *Group-2* has nets SM[3], SM[2], SM[1], SM[0]
- In second step, we consider one group at a time to create net-clusters. At the end of this step, we get total three net-clusters (two from *Group-1* and one from *Group-2*). These are:
  NC-1 = {LB[3], LB[2], LB[1], LB[0]};
  NC-2 = {SB[3], SB[2], SB[1], SB[0]} and
  NC-3 = {SM[3], SM[2], SM[1], SM[0]}.

3.4.2 Instance-driven Clustering (IDC)

While designing the datapath, if some nets were not named using a uniform naming scheme, then FDC algorithm would not identify their corresponding net-clusters. This problem often occurs when a logic synthesis tool is utilized to create the schematic. Synthesis tools usually assign randomly generated names for unnamed nets. To identify such nets and create the appropriate net-clusters, we apply *instance-driven clustering*. This is also a two-step technique. The IDC algorithm is as follows:

- For each un-clustered net, we first compute the global footprints to form candidate groups, just as in the FDC algorithm.
- Next we consider one group at a time and create net-clusters from that group by using the following definition: Let us assume that we have two nets *netA* and *netB* which are connected to *P* pins each. After sorting the pin co-ordinates by *Y*-coordinate value, assume that the pins of *netA* are at locations $(x_a^1, y_a^1), (x_a^2, y_a^2), \cdots, (x_a^P, y_a^P)$ and the pins of *netB* are at $(x_b^1, y_b^1), (x_b^2, y_b^2), \cdots, (x_b^P, y_b^P)$. Then two nets belong to same net-cluster if the following $2 \cdot P$ conditions are satisfied:
  1. $y_a^j = y_b^j$ (for $j = 1, 2, \cdots, P$)
  2. $|x_a^j - x_b^j| = k \cdot bit\_pitch$ (for $j = 1, 2, \cdots, P$ and $1 \leq k \leq N-1$)

The above algorithm can be illustrated by using a slightly modified version of the design shown in Figure 2. Let us assume that in Figure 2, the logic synthesis tool specified names AB, CD, EF, GH, KL, MN, RS and TV for nets LB[3], LB[2], LB[1], LB[0], SB[3], SB[2], SB[1] and SB[0] respectively.

After running IDC algorithm, we get two net-clusters. These are *NC-1* = {AB, CD, EF, GH} and *NC-2* = {KL, MN, RS, TV}.

3.4.3 Cluster merging

After running both FDC and IDC, we attempt to merge clusters. This is useful in designs where some nets in a cluster have regular names while others do not. We illustrate this algorithm with the help of a slightly modified version of Figure 2. Let us assume that net $SM[3]$ was named as *ABC* and net $SM[2]$ was named as *DEF*.

First, we invoke FDC algorithm to get following 3 net-clusters. *NC-1* = {LB[3], LB[2], LB[1], LB[0]}; *NC-2* = {SM[1], SM[0]} and *NC-3* = {SB[3], SB[2], SB[1], SB[0]}. Then, IDC identifies an additional cluster: *NC-4* = {ABC, DEF}.

Now, we apply our cluster merging technique. We only consider those clusters which have less than *N* member nets, where *N* is the number of bits in the datapath. Now, after selecting one representative net from each net-cluster, we check whether the $2 \cdot P$ conditions described in the IDC algorithm are satisfied. If they are satisfied, then we merge the two net-clusters. After each merging operation, if the number of nets in a merged cluster becomes equal to *N*, then it is not considered as a candidate for further merging. We continue this process on all candidate clusters until no further merging is possible. In our example, clusters *NC-2* and *NC-4* get merged to form a new cluster: *NC-New* = {ABC, DEF, SM[1], SM[0]}.

3.5 Selecting a Representative Bit-slice

One of the powerful features of our router is that we *explicitly route a single bit-slice* and then propagate the routes to other bit-slices. Therefore, all other nets in a net-cluster are routed *implicitly*. To use this routing approach, we need to select a representative bit-slice on which to perform explicit routing.

A net is said to have *same-bit connections* if all the pins connected to that net belong to a single bit-slice. Such a net is called a *same-bit net*. On the other hand, if all the pins of a net do not belong to the same single bit-slice, we define that connection as a *cross-bit connection*, and call the net a *cross-bit net*. If a net *X* has a cross-bit connection from bit-slice *SX* (source) to bit-slice *DX* (destination), we denote that connection as $\{X : SX, DX\}$; where $SX \neq DX$. There are two types of cross-bit connections.

- *Forward cross-bit connection of degree l*: $\{X : SX, DX\}$, if $(DX - SX) = l$ (where $l > 0$).
- *Backward cross-bit connection of degree l*: $\{X : SX, DX\}$, if $(SX - DX) = l$ (where $l > 0$).

To determine the representative bit-slice, we conceptually consider the datapath to have an infinite number of bit-slices on the left of the $(N-1)^{th}$ bit-slice, and on the right of the 0^{th} bit-slice. In this way, each of the *N* bit-slices has an identical number of nets. In such a structure, any of the *N* bit-slices can be used as the representative bit-slice for explicit routing.

3.6 Routing the Nets in the Selected Bit-slice

Our next task is to route the nets present in the representative bit-slice. As we can see in the Figure 2, the datapath cells are placed in a row-like fashion. After placement, most of the connections are found to be confined within nearby rows.

Our routing approach is a combination of pattern-based routing and maze routing. Lee. et. al. [7] originally proposed the maze routing algorithm. Pattern routing was introduced by L. S. Pugh et. al. [8] and subsequently modified by J. Soukup et. al [9] and T. Asano [10].

We call our main approach as *strap-based routing*. We define a strap as a straight segment, which can be either vertical or horizontal. We denote a strap between points (x_i, y_j) to (x_k, y_j) as: $(x_i, y_j) \rightarrow (x_k, y_j)$. Strap-based routing is a gridless routing approach. In our router, if strap-based routing fails, a maze router is invoked as a fall-back. We first discuss the routing strategy for same-bit nets and then consider cross-bit nets.

3.6.1 Routing Same-Bit Nets

Initially, we have the list of nets that need to be routed. For each net, we have the list of end-points (which will be some pins). We first sort the nets in decreasing order of the largest *Y* co-ordinate value of their pins. In case of a tie, we select the longest net first, with the intuition that longer nets are expected to be harder to route later. We also have a mechanism by which the user can assign precedence to a particular net by assigning a large weight to that net. Algorithm 2 describes our net ordering scheme.

If two end points of a net are (x_1, y_1) and (x_2, y_2), then we define a *direct route* as a path which has one the following strap patterns:

- Case 1: if $(x_1 = x_2)$ AND $(y_1 \neq y_2)$
  Only strap (vertical): $(x_1, y_1) \rightarrow (x_1, y_2)$
- Case 2: $if (x_1 \neq x_2)$ AND $(y_1 = y_2)$
  Only strap (horizontal): $(x_1, y_1) \rightarrow (x_2, y_1)$

Algorithm 2 : Net Ordering Process

```
if (special weights present) then
   WtNets = findWeightedNets(allNets)
   SortedWtNets = sortNetsByWeightDescend(WtNets)
   OtherNets = allNets - WtNets
else
   OtherNets = allNets
end if
SortedOtherNets = sortNetsByPinYCoordsDescend(OtherNets)
if (same Y for multiple nets among SortedOtherNets) then
   SameYNets = GetSameYNets(SortedOtherNets)
   SortedConflictedNets = SortByLongestLength(sameYNets)
   SortedOtherNets = ModifySortedNets(SortedConflictedNets,
   SortedOtherNets)
end if
SortedAllNets = SortedWtNets + SortedOtherNets
Return SortedAllNets
```

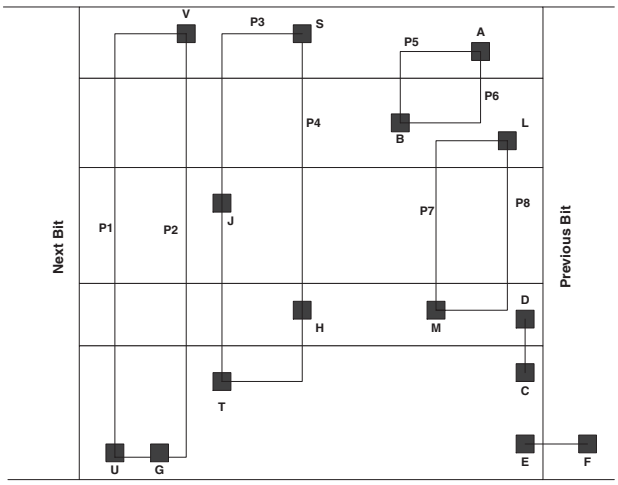

Figure 3: Routing in Single Bit-Slice

- Case 3: if $(x_1 \neq x_2)$ AND $(y_1 \neq y_2)$
  1. *Vertical-Then-Horizontal* (VTH)
     First strap (vertical): $(x_1,y_1) \rightarrow (x_1,y_2)$
     Second strap (horizontal): $(x_1,y_2) \rightarrow (x_2,y_2)$
  2. *Horizontal-Then-Vertical* (HTV)
     First strap (horizontal): $(x_1,y_1) \rightarrow (x_2,y_1)$
     Second strap (vertical): $(x_2,y_1) \rightarrow (x_2,y_2)$

We illustrate our routing algorithm using representative bit-slice of Figure 3. Case 1 of *direct route* is shown between pins *D* and *C*. Case 2 is shown between pins *E* and *F* (this case occurs only for cross-bit nets). Case 3 is shown in between pins *A* and *B*. Examples of case 3 are shown in path *P*5 (HTV) and path *P*6 (VTH).

After sorting all nets with respect to the largest *Y* co-ordinate of their pins, we note that the topmost two pins are pins *V* and *S*. We select the net associated with pin *V* as our first routing candidate, since it would have a longer vertical strap (assuming we can find a direct route for the net associated with pin *S*). We first try to find a direct route (this minimizes the via count). We attempt both VTH and HTV direct routes, and check whether either of these routes intersect with any other pin/blockage. In this example, path *P*2 (VTH direct route) intersects with pin *G*. So we choose path *P*1 as the route between pins *V* and *U*. If there was no blockage in path *P*2, then we could have taken any of those two paths as the final route.

Another scenario that often occurs is that both the VTH and HTV direct routes intersect some pin/blockage. In Figure 3, we note that the connection between pins *S* and *T* illustrates this case. Pins *J* and *H* block both the direct paths (*P*3 and *P*4 respectively). Therefore, we try to form a 3-strap path from both the direct paths. To denote the co-ordinates of any pin, we use following convention: pin *A* is located at (x_A,y_A), pin *B* is at (x_B,y_B) and so on. To tackle the problem of forming a 3-strap path from 2 direct paths (when we have only 2 pin-blockages), we do the following:

- if $(y_H < y_J)$ and there exists two legal straps $(x_S,y_S) \rightarrow (x_H,y_H)$ and $(x_J,y_J) \rightarrow (x_T,y_T)$; then we check if there exists a legal strap from $(x_H,y_H) \rightarrow (x_J,y_H)$. If it exists, we have got one 3-strap route. Otherwise, we search for the existence of a strap $(x_H,y_*) \rightarrow (x_J,y_*)$, for $y_H \leq y_* \leq y_J$. If such a strap is not found, we search for 5-strap routes.
- if $(y_H > y_J)$ and there exists two legal straps $(x_S,y_S) \rightarrow (x_J,y_S)$. and $(x_T,y_T) \rightarrow (x_H,y_T)$; then we try to find a vertical strap $(x_*,y_T) \rightarrow (x_*,y_S)$; where $x_T \leq x_* \leq x_S$. If we are successful, then we get a 3-strap route. Otherwise, we extend the bounding box of our vertical strap finder to $(x_T - width) < x_I < (x_T + 2 \cdot width)$, where $width = x_S - x_T$.

Once we are done routing the nets originating from the topmost row (i.e. the nets between *V* - *U*, *S* - *T*, and *A* - *B*, we route the nets originating from the next row from the top. So we next select the net connecting pin *L* and pin *M*. By routing nets one row at a time, we minimize the possibility of conflict with routes in lower rows. Unfortunately, we cannot guarantee that there will be no conflicts. To handle conflicts between an existing route and a new route, we sometimes need to *rip-up* existing routes.

3.6.2 Routing Cross-Bit Nets

Consider a design in which the maximum degree of forward or backward cross-connectivity is *k*. One method to route this design is to route $k+1$ consecutive bit-slices as a single unit. Such an approach would result in longer run-times and may generate irregular routes for different nets in a net-cluster. In our method, we route a *single* representative bit-slice and infer cross-bit connections while performing the route.

In our approach, we model a single net, spread over multiple bit-slices, as a combination of multiple sub-nets, each confined within a single bit-slice. As a result, only one such sub-net explicitly belongs to the representative bit-slice. Other sub-nets belong to other bit-slices. We *virtually instantiate* the sub-nets of other bit-slices into the representative bit-slice. Virtual instantiation of a sub-net implies treating the sub-net as a part of the representative bit-slice while routing. After routing, we reinstate the actual sub-net route back to its original bit-slice. This method of modeling cross-bit nets saves run-time, memory and ensures regularity of the resulting design.

Figure 4 shows 4 bit-slices of a larger design, with a forward cross-connectivity of degree 2. Let us assume that our representative bit-slice is the bit-slice *k*. *Net-1* connects pin *S* of instance *A*5 (in bit-slice *k*) to pin *X* of instance *D*7 (in bit-slice $K+2$). We assume that *Net-1* belongs to a net-cluster which has other member nets as well. In order to maintain the readability of Figure 4, only one other net (*Net-2*) of this net-cluster is shown.

Our aim is to route all cross-bit nets with just the data of the representative bit-slice (bit-slice *k*) loaded in memory. The core algorithm for finding routes is the same as that for same-bit nets, with a few modifications to handling cross-bit nets. To obtain the route for *Net-1*, we actually need to traverse through 3 bit-slices (because the degree of cross-bit connectivity is 2). Therefore, whenever we

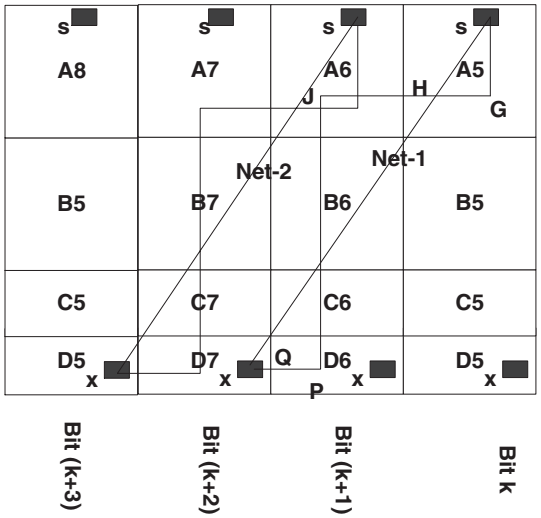

Figure 4: Routing of Cross-Bit nets

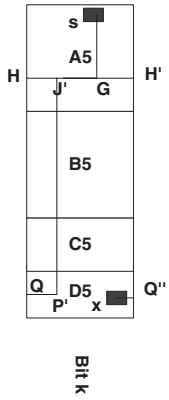

Figure 5: Representative bit-slice with virtual instantiations of sub-nets

try to extend a horizontal strap to some location in the adjacent bit, we split the strap into two straps such that each strap is confined to a single bit-slice. In the case of *Net-1*, we first obtain a strap from point S (in instance $A5$) to point G (in $A5$). Next, we attempt to create a horizontal strap from point G (in $A5$) to point J (in $A6$). In order to illustrate the mechanism by which we model this strap using a single representative bit-slice, we do following splitting:

$(x_G, y_G) \rightarrow (x_J, y_J) = (x_G, y_G) \rightarrow (x_H, y_H) + (x_H, y_H) \rightarrow (x_J, y_J)$, where H is a point on the bit-slice boundary and $y_H = y_G = y_J$.

Now we virtually instantiate the pins of other bit-slices into the representative bit-slice. Figure 5 shows the representative bit-slice for the datapath of Figure 4, with all virtual instantiations performed. First, we virtually instantiate the point J within the representative bit-slice k by incrementing the X co-ordinate of point J by a bit-pitch.

In Figure 5, we denote the virtual point as J'. Similarly, we virtually instantiate the point H as the point H' in the bit-slice k. Once these points have been instantiated, we instantiate the $(x_H, y_H) \rightarrow (x_J, y_J)$ strap of bit-slice $k+1$ within the k^{th} bit-slice as a strap $(x'_H, y'_H) \rightarrow (x'_J, y'_J)$. Depending on the location of points J and G, we may get an overlap between the straps $(x_G, y_G) \rightarrow (x_H, y_H)$ and $(x'_H, y'_H) \rightarrow (x'_J, y'_J)$. This virtual overlap is not a problem, because the overlapping straps belong to the same net. After routing, the virtual straps will be reinstated to their actual bit-slices, solving the overlap problem. Once we reach the virtual destination pin, then we simply perform the reverse mapping of all virtual locations to their original locations to get the legal straps. We take the same virtual instantiation approach for vertical straps such as $(x_J, y_J) \rightarrow (x_P, y_P)$.

If we follow the above approach, then we may get overlaps in the horizontal straps of cross-bit nets belonging to same net-cluster. To avoid this problem, we use the following approach: in any net-cluster, if the horizontal span between the source and destination pins is p, then we generate p different routes for that net-cluster.

3.7 Propagating the Routes and Completing

Having performed the routing in a representative bit-slice, we propagate the routes to other bit-slices. For this, we use the net-cluster information. Our route propagation schemes are described in Algorithm 3 (for same-bit nets) and in Algorithm 4 (for cross-bit nets).

Algorithm 3 : Route Propagation (for Same-Bit nets)

```
AllNets = getAllNets(k)
for each net (MasterNet) in AllNets do
  MasterRoute = getRouteForNet(MasterNet)
  NetCluster = getNetClusterForNet(MasterNet)
  OtherSisterNets = getSisterNets(NetCluster, MasterNet)
  for each net (SisterNet) in OtherSisterNets do
    SisterRoute = ModifyRoute(MasterRoute, SisterNet, MasterNet)
    AssignRoute(SisterNet, SisterRoute)
  end for
end for
```

Algorithm 4 : Route Propagation (for Cross-Bit nets)

```
AllCrossBitNets = getAllCrossBitNetsWithSourceInBit(k)
for each net (MasterNet) in AllCrossBitNets do
  MasterRoutes = getMasterRoutes(MasterNet, p) /* MasterRoutes is an array of p distinct routes */
  MasterRoute = MasterRoutes[0]
  NetCluster = getNetClusterForNet(MasterNet)
  OtherSisterNets = getSisterNets(NetCluster, MasterNet)
  for each net (SisterNet) in OtherSisterNets do
    NewSourceBit = getSourceBit(SisterNet)
    PositiveBitDiff = NewSourceBit-k+N /* k is the index of SisterNet */
    ModValue = (PositiveBitDiff modulus p)
    SisterRoute = ModifyRoute(MasterRoutes[Modvalue], SisterNet, MasterNet)
    AssignRoute(SisterNet, SisterRoute)
  end for
end for
```

At this stage, we check whether there are any unrouted nets present in the design. Usually, there are no nets left unrouted at this stage. If some nets have not been routed, we invoke the strap-based routing scheme to route these nets.

Circuit	# Instances	# Connections	# Bits
Industry-1	1056	5504	32
Industry-2	2368	12672	32
Industry-3	4672	29184	64
Industry-4	6208	48128	64

Table 1: Characteristics of Example Circuits

	Run-time (minutes)			Wire- length (μm)			Via-count		
Example	Ind. Router	Our Router	Ratio	Ind. Router	Our Router	Ratio	Ind. Router	Our Router	Ratio
Industry-1	70	12	0.17	47458	46790	1.41	6856	5678	17.18
Industry-2	145	26	0.18	97453	99476	-2.07	25876	22336	13.68
Industry-3	264	36	0.14	276456	282569	-2.21	39568	39452	0.28
Industry-4	394	42	0.11	589679	564568	4.25	44568	42976	3.57
AVERAGE			0.15			0.35			8.68

Table 2: Run-time, Wire-length and Via-count comparison between an Industrial Router and Our Router

4. ADVANTAGES OF OUR APPROACH

Our routing approach has several advantages over traditional routing schemes utilized in the datapath context. Some of these are:

- **Speed of Routing:**
  By exploiting datapath regularity, our routing technique is able to route an entire datapath while explicitly routing only a small subset of the nets. This approach makes it possible to route large industrial datapaths with significantly shorter run-times compared to a traditional router.

- **Easy incremental routing:**
  Before a design is taped out, several iterations of routing and timing checks are performed. In these iterations, designers often modify the design slightly. In such a scenario, the router needs to perform efficient *incremental routing*. Because of the inherent speed and the regular nature of our router, it is very much suitable for incremental routing.

- **Predictable Routes :**
  The routes obtained by our router are highly regular across bit-slices. So, the wiring parasitics for different nets in a net-cluster are very similar, resulting in a predictable design.

- **Better Debuggability and Timing:**
  If a datapath, routed using conventional routers, does not meet timing requirements, the designer usually spends a significant amount of time trying to find the badly routed nets and then re-routes those nets. However, the potential irregularity of the routed nets can cause a ripple effect, making some other nets critical. In our flow, all nets in a net-cluster have substantially similar delays. This allows better and more predictable timing characteristics of the design, and eases the debugging task.

5. EXPERIMENTAL RESULTS

We implemented our router in the C++ programming language. The code for our datapath router consists of about 7000 lines of C++. Experiments were run on a 440 MHz HP machine with 512 MB memory, running the HP-UX 10.20 operating system.

To compare our results, we used datapath blocks from state-of-the-art 32-bit or 64-bit microprocessors. Table 1 describes the characteristics of our benchmark circuits. We compared our algorithm against a commercially available router. In Table 2, we report run-time, wire-length and via-count results from both our router and industrial router.

On an average, our router is about $6\times$ faster for 32-bit datapaths, and $8\times$ faster for 64-bit designs. This is expected since we only route a single representative bit-slice in our approach. After our current prototype router is optimized for speed, we expect our run-time numbers to improve further.

We note that the average wire-length gain of our method is minimal. Finally, we notice that our technique reduces the total number of vias by about 8%. We conjecture that the our router utilizes fewer vias because of its strap-based nature.

6. CONCLUSION

In this paper, we have presented a new style of detailed routing for datapath designs, which fully utilizes the regular structures present in a datapath. In our technique, we first extract interconnection regularity within the datapath by creating "net-clusters". Next, we route a single representative bit-slice of the datapath, and from the routes thus obtained, we infer routes for the rest of the nets in its net-cluster. Experimental results demonstrate a significant improvement in run-time over a commercial router. Also, our router produces highly predictable timing results and allows easy incremental routing.

7. REFERENCES

[1] N. Buddi, M. Chrzanowska-Jeske, and C. Saxe, "Layout synthesis for datapath designs," in *Proceedings of European Design Automation Confererence*, pp. 86–90, 1995.

[2] T. Ye and G. D. Micheli, "Data path placement with regularity," in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pp. 264–270, 2000.

[3] J. King and S. M. Kang, "A timing-driven data path layout synthesis with integer programming," in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pp. 716–719, 1995.

[4] S. Raman, S. Sapatnekar, and C. Alpert, "Datapath routing based on a decongestion metric," in *Proceedings of the ACM International Symposium on Physical Design*, pp. 122–127, 2000.

[5] S. Arikati and R. Varadarajan, "A signature based approach to regularity extraction," in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pp. 542–545, 1997.

[6] S. Hassoun and C. McCreary, "Regularity extraction via clan-based structural circuit decomposition," in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pp. 414–418, 1999.

[7] C. Lee, "An algorithm for path connections and its applications," *IRE Transactions on Electronic Computing*, pp. 346–365, 1961.

[8] L. Pugh, "An improvement in printed circuit board routability using a maze-running algorithm," *Electronic Letters*, vol. 14(1):8-9, Jan 1991.

[9] J. Soukup and S. Fournier, "Pattern router," in *Proceedings of the International Symposium on Circuits and Systems*, pp. 486–489, 1979.

[10] T. Asano, "Parametric pattern router," in *Proceedings of ACM/IEEE Nineteenth Design Automation Conference*, pp. 411–417, 1982.