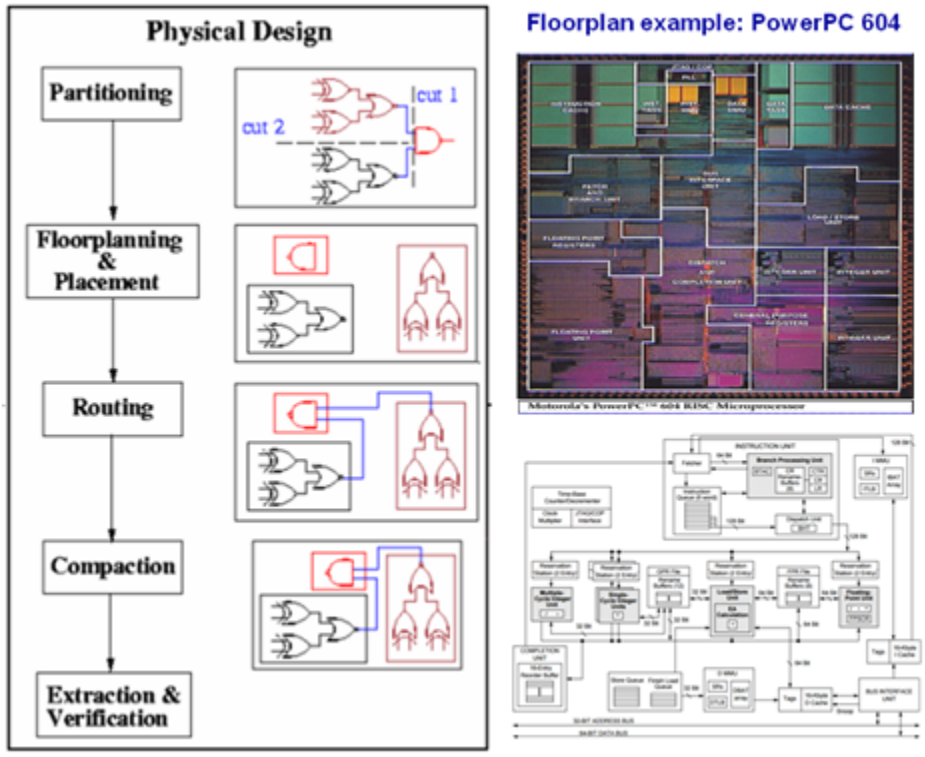
FA22 MP11 - Floorplan

Due Thursday, November 17th, 10:00 PM CT

Please work on EWS machine for this MP.

Physical design is an important step in the design automation flow of VLSI systems. It refers to all synthesis steps that convert a circuit representation (in terms of gates and transistors) into a geometric representation (see Figure 1). Floorplanning is one major step in physical design. It assembles partitioned circuit modules into a rectangular chip to optimize a predefined cost metric such as area; it is particularly crucial because the resulting floorplan affects all the subsequent steps in physical design, such as cell placement, wire routing, and so on. In this MP, you are asked to implement a floorplanner that layouts a set of rectangular modules and optimizes the packing area.



presentation.pptx**Figure 1**: Physical design flow of VLSI systems.

Floorplan Model

The floorplan model we consider for this MP is *slicing*. A slicing floorplan can be obtained by repetitively cutting the floorplan horizontally or vertically, whereas a non-slicing floorplan cannot. Figure 2 shows an example of a slicing floorplan. The floorplan here is a slicing floorplan since each module can be sliced out through a sequence of vertical cut or horizontal cut. In this example, the 1st vertical cut slices out the module 5, and the 2nd horizontal cut slices out the module 4. Subsequently, the 3rd horizontal cut slices out the module 1 and the final vertical cut slices out the module 2 and module 3. On the other hand, we cannot apply this cutting process to slice out any modules from the floorplan.

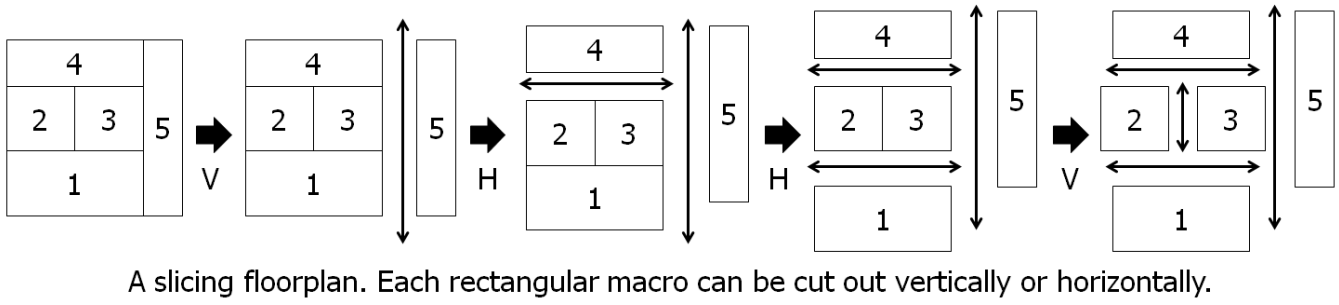


Figure 2: A slicing floorplan example.

Slicing Tree Floorplan Representation

On the basis of the property of a slicing floorplan, a tree data structure can be used to represent a slicing floorplan. A slicing tree is a binary tree with modules at leaves and cutlines at internal nodes (non-leaf nodes). There are two cutline types, *H* and *V*. The cutline *H* divides the floorplan horizontally and the cutline V divides the floorplan vertically. An example is shown in Figure 3. The tree root, *V*, represents the vertical cutline that divides the floorplan into the left sub-floorplan packed by modules 1 and 2 and the right sub-floorplan packed by modules 3, 4, and 5. The left child of the root is an internal node with horizontal cutline, which horizontally divides the left sub-floorplan into the bottom sub-floorplan (module 1) and the top sub-floorplan (module 2). Similarly, the right child of the root represents a horizontal cut that divides the sub-floorplan into the bottom (module 3) and the top (modules 4 and 5) sub-floorplans, after which the top sub-floorplan is further divided into the bottom (module 4) and the top (module 5) sub-floorplans.

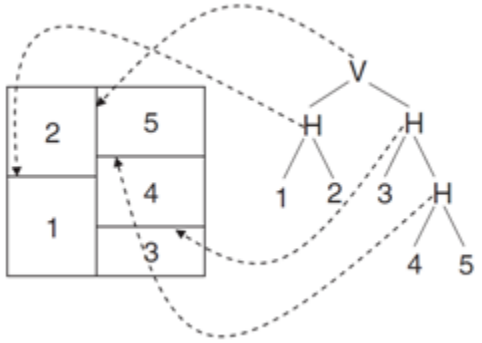


Figure 3: An example of a slicing floorplan and its slicing tree representation. The corresponding postfix expression is *12H345HHV*.

The slicing tree representation is advantageous in its 1D postfix expression $E = \{e_1, e_2, e_3, e_4, ..., e_{2n-1}\}$ where n is the number of modules and e_i is an expression unit which belongs to {*1*, *2*, *3*, *4*, ..., *n*, *V*, *H*}. Here, each number denotes a module index and *H* (*V*) represents a horizontal (vertical) cutline. The postfix expression is a postfix order of the tree nodes and it be obtained via a postfix traversal on the slicing tree. For example, the postfix expression of the slicing tree in Figure 3 is $E = \{12H345HHV\}$. With this elegant property, design automation tools can work on the 1D postfix expression instead of the explicit 2D layout so as to ease the optimization process. The transformation of a postfix expression *E* to its corresponding floorplan can be achieved via a bottom-up approach that recursively combines the sub-floorplans on the basis of *E*. The two cutlines are viewed as two binary operators. If *a* and *b* are two modules or two sub-floorplans, the expression *abH* implies to place *a* below *b*, and *abV* implies to place *a* to the left of *b*. The resulting rectangle area can be determined during the packing process, as illustrated in Figure 4. So by manipulating the slicing trees, we can generate multiple floorplan representations. This technique is very useful for optimization where the chip size should be as small as possible.

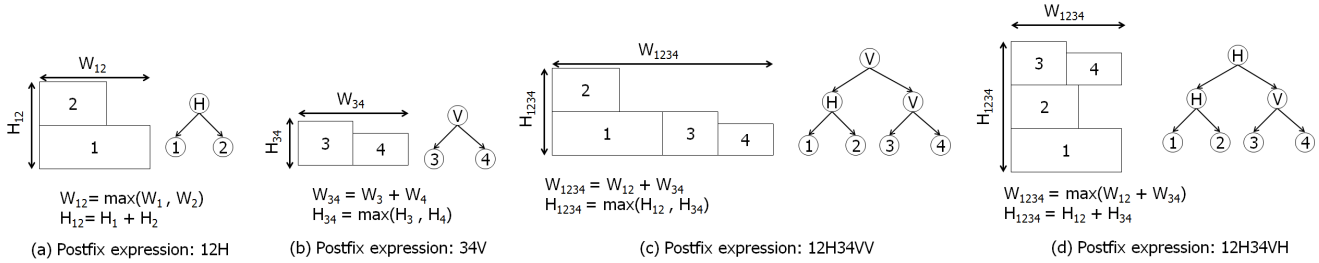


Figure 4: Transformation of a 1D postfix expression into the corresponding 2D floorplan. The resulting rectangle area can be determined in a recursive manner.

Coding Blocks

In this MP, we have defined a set of structs and you should follow them to design your floorplanner. Definitions for these structs can be found in *floorplan.h*. You can define your own structs here if necessary for your implementation (not recommended though). The **cutline_t** is an enum identifier and it defines the type of a cutline, whose value could be either vertical (*V*) or horizontal (*H*), or undefined (*UNDEFINED_CUTLINE*). The **module_t** is a struct identifier for a module, which contains 1) an integer variable *idx* denoting the unique index of the module, 2) two integer variables *llx* and *lly* denoting the lower-left *x* and *y* coordinates of the module, 3) two integer variables *w* and *h* denoting the width and height of the module, and 4) an integer variable *resource** denotin g the compute resource of the module. The **expression_unit_t** defines the unit of the postfix expression, which could be either a module pointer or a cutline value. The **node_t** is a struct identifier that defines the node in the slicing tree. A node is either an internal node, where a cutline type must be explicitly specified or a leaf node where the value of module pointer must be assigned. Each node is associated with three node pointers, *parent*, *left*, and *ri ght. T*he topology of the slicing tree can be implicitly inferred by these node-to-node pointers. The pointer *parent* points to the parent of the node. Every node has only one parent and the parent of the tree root is assigned by *NULL*. Due to the binary property of a slicing tree, a node has two children, which are connected by the pointer *left* and the pointer *right*, respectively. A node with *NULL* values on both *left* and *right* pointers is a leaf node. An example is shown in Figure 7. For instance, the node 0x40 (memory address) is a leaf node and the node 0x10 is the tree root. The nodes 0x20 and 0x30 are internal nodes.

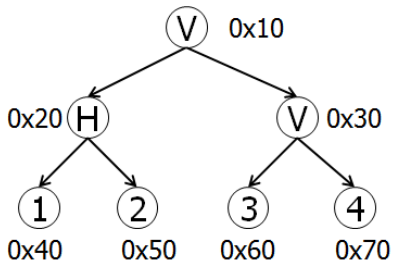
*The *resource* here in module_t is a simplified way to represent how much compute resources this module needs(i.e. ALUs, Registers, etc). A larger number for *resource* means the module needs more compute resources. This is different from the area described by width and height.

```
typedef enum CUTLINE {
  V = 0,
  H = 1,
  UNDEFINED_CUTLINE
} cutline_t;

typedef struct MODULE {
  int idx;
  int llx, lly;
  int w, h;
  int resource;
} module_t;

typedef struct EXPRESSION_UNIT {
  module_t* module;
  cutline_t cutline;
} expression_unit_t;

typedef struct NODE {
  module_t* module;
  cutline_t cutline;
  struct NODE* parent;
  struct NODE* left;
  struct NODE* right;
} node_t;
```



Node address	left	right	parent	module	cutline
0x10	0x20	0x30	NULL	NULL	V
0x20	0x40	0x50	0x10	NULL	H
0x30	0x60	0x70	0x10	NULL	V
0x40	NULL	NULL	0x20	1	UNDEFINED_CUTLINE
0x60	NULL	NULL	0x30	3	UNDEFINED_CUTLINE

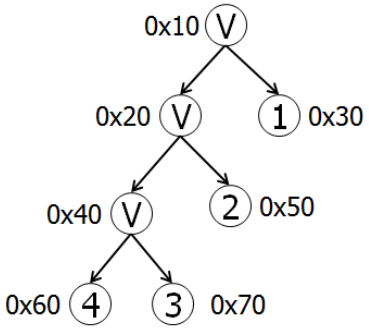
Figure 5: A slicing tree example with a table showing the corresponding data field of nodes.

In addition, we will have two **global** variables, **num_modules** and **modules**, defined in *floorplan.cpp*. As we will parse the circuit for you (the procedure for circuit parsing is **read_module**), the global variable **num_modules** records the total number of modules and the global variable **modules** is a pointer to the module array parsed from the given circuit input.

```
int num_modules;
module_t* modules;
```

The first task you have to finish after the parsing is to create an initial slicing tree by the function **init_slicing_tree**. For simplicity, we ask you to generate a slicing tree that results in a horizontally-aligned floorplan. In other words, all the modules are aligned together and placed along a horizontal line. In this case, the resulting slicing tree is left-skewed, growing down to the left. The right child of each node is always a leaf node attached by a module pointer. An example of a left-skewed slicing tree with 4 modules is shown in Figure 6. You need to implement the function **init_slicing_tree** recursively. The function **init_slicing_tree** takes a node pointer to the parent of the current recursion and an integer variable indicating the index on the module array. At each recursion step, first generate an internal node with vertical cutline and a right child node attached by a module pointer (as the property of a left-skewed slicing tree). Then create a left child node and step to the next recursion parented by the current internal node. Establish the tree links appropriately (*left*, *right*, *parent* pointers). The recursion ceases when no more modules can be attached, and should ultimately return a pointer to the tree root. We will verify your initialization by its postfix expression, which should appear like "*n, n-1, V, n-2, V, n-3, V... 2, V, 1, V*".

```
node_t* init_slicing_tree(node_t* par, int nth);
```



Node address	left	right	parent	module	cutline
0x10	0x20	0x30	NULL	NULL	V
0x20	0x40	0x50	0x10	NULL	V
0x30	NULL	NULL	0x10	1	UNDEFINED_CUTLINE
0x40	0x60	0x70	0x20	NULL	V
0x50	NULL	NULL	0x20	2	UNDEFINED_CUTLINE

Figure 6: A left-skewed slicing tree for a horizontally aligned floorplan. The right child of any internal node is a leaf node.

Based on the above primitives, the rest job of this MP is to implement a set of functions and procedures for a slicing-tree floorplanner. The first set consists of four fundamental tree queries, **get_total_resource**, **is_leaf_node**, **is_internal_node**, and **is_in_subtree**. The second set consists of two procedures, **get_expression** and **postfix_traversal**, which are essential for encoding the 2D floorplan to a 1D postfix expression. The final set is the tree modifiers for the perturbation of slicing trees, **rotate**, **recut**, **swap_module**, and **swap_topology**.

The function **get_total_resource** takes the root pointer and should return total sum of resource of modules(the variable resource in module_t) in the tree. You should traverse the tree to get this sum.

The function **is_leaf_node** takes a node pointer and should return 1 if the node pointer points to a leaf node in the slicing tree, or 0 otherwise.

The function **is_internal_node** takes a node pointer and should return 1 if the node pointer points to an internal node in the slicing tree, or return 0 otherwise.

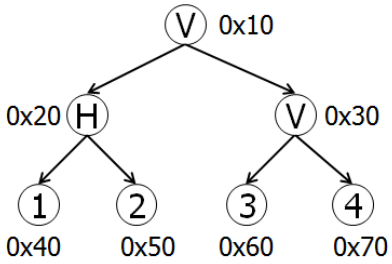
The function **is_in_subtree** takes two node pointers to nodes *a* and *b* in the slicing tree, and should return 1 if the subtree rooted at the node *b* belongs to a part of the subtree rooted at the node *a*. Notice that a subtree rooted at the node *k* is defined as the set of all downstream nodes of the node *k* (inclusive). For example, the subtree rooted at the node 0x30 (memory address) in Figure 7 consists of three nodes, 0x30, 0x60, and 0x70. Further, any subtrees rooted at one of these three nodes (i.e., 0x30, 0x60, 0x70) belong to a part of the subtree rooted at the node 0x30. On the other hand, the subtree rooted at the node 0x20 is exclusive of the subtree rooted at the node 0x30.

```
int get_total_resource(node_t* ptr);
int is_leaf_node(node_t* ptr);
int is_internal_node(node_t* ptr);
int is_in_subtree(node_t* a, node_t* b);
```

The procedure **get_expression** traverses the slicing tree in a postfix order and encodes it into a postfix expression. The postfix expression should be stored in the given array pointed by **expression**. In order to obtain the postfix expression of the slicing tree, you have to finish the recursive procedure **postfix_traversal**, which is an internal call of the procedure **get_expression**. The procedure **postfix_traversal** takes three input arguments, a node pointer pointing to a subtree from which the current recursive call takes place, an integer pointer indicating the position where the postfix expression needs to fill, and a pointer to the expression array. The algorithm of postfix traversal on a tree is given by Figure 7. The postfix traversal is performed in a recursive manner. A node is processed as long as both its two children nodes are handled. Using the postfix traversal algorithm, the derived postfix expression of the slicing tree in this example should be *12H34VV,* where the course of the postfix traversal is shown in the table.

```
void get_expression(node_t* root, int N, expression_unit_t* expression);
void postfix_traversal(node_t* ptr, int* nth, expression_unit_t* expression);
```

Postfix traversal algorithm
1. Traverse the left subtree by recursively calling the Postfix function.
2. Traverse the right subtree by recursively calling the Postfix function.
3. Process the data part of root element (or current element).

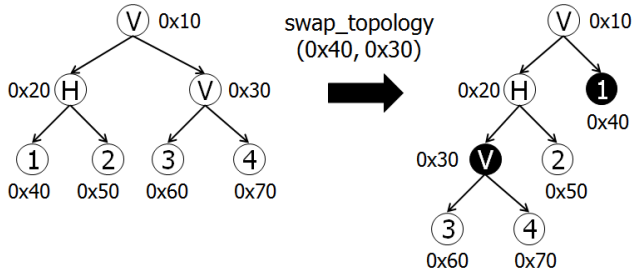


nth	0	1	2	3	4	5	6
Expression unit	1	2	H	3	4	V	V
Node address	0x40	0x50	0x20	0x60	0x70	0x30	0x10

Figure 7: An example of postfix (postorder) traversal on a slicing tree. The derived postfix expression is *12H34VV*.

The procedure **rotate** takes a pointer to an arbitrary node in the slicing tree and rotates the corresponding module (if any) by 90 degree. In other words, the width and the height of the module should be swapped. Similarly the procedure **recut** changes the cutline from either horizontal direction to vertical direction or vertical direction to horizontal direction. On the other hand, the procedure **swap_module** takes two node pointers and swaps the corresponding module pointers. Finally, the procedure **swap_topology** swaps the two subtrees rooted at two given node pointers. For the first three procedures, **rotate**, **recut**, and **swap_module**, simply change the data field of each node and **do not** modify the topology of the tree. For the procedure **swap_topology**, you have to manipulate the link between tree nodes in order to change the tree topology (i.e., *left*, *right*, *parent* pointers in the node struct should be redirected appropriately). An example of the operation **swap_topology** is demonstrated in Figure 8.

```
void rotate(node_t* ptr);
void recut(node_t* ptr);
void swap_module(node_t* a, node_t* b);
void swap_topology(node_t* a, node_t* b);
```



Node address	left	right	parent	module	cutline
0x10	0x20	**0x40**	NULL	NULL	V
0x20	**0x30**	0x50	0x10	NULL	H
0x30	0x60	0x70	**0x20**	NULL	V
0x40	NULL	NULL	**0x10**	1	UNDEFINED_CUTLINE
0x60	NULL	NULL	0x30	3	UNDEFINED_CUTLINE

Figure 8: An example of the operation **swap_topology** on two nodes 0x40 (memory address) and 0x30. The data field on corresponding nodes should be modified accordingly.

The coding sections you have to finish have been marked by "TODO" in *floorplan.c*.

Benchmark Description

A circuit benchmark example (*circuit1.txt*) is shown in the following block. The first line is always a positive number *N* (>2) indicating the total number of modules. After that, *N* lines are followed with each line consisting of 1) an integer number indicating the unique index of a module, and 2) the width, height, and resource of this module. All data field are integers (i.e., no floating points). Although we have provided a parser for you and you don't have to worry about the I/O, reading the format will help you understand the problem. More details about the parsing process can be referred to the procedure **read_module**.

```
4
0 280 296 5
1 333 188 7
2 523 192 3
3 549 296 1
```

Building and Testing

In this MP, we provide you a makefile to help automatically build and manage your project. Under the directory of your source codes, type the command "make floorplanner" to compile and build your source code into an executable binary. You will find a binary named "floorplanner" in the same folder, if no error messages occur. Similarly, the command "make clean" will remove all compiled files and temporary objects.

```
~> make floorplanner
```

You can start testing your code after you successfully compile and build the source codes. Simply type "make *circuit_name*", where *circuit_name* is replaced with *circuit1*, *circuit2*, *circuit3*, *circuit4*, and *circuit5*. You will find results on the screen. By default, the library has been installed on EWS machine and you should be able to compile your MP without error.

```
~> make circuit1

**************************** VERIFICATION ****************************
Circuit: 4 golden_modules, slicing tree size = 4 leaves and 3 internals
(1) Function  'init_slicing_tree': correct!   +25
(2) Function  'is_leaf'     : correct!   +5
(3) Function  'is_internal'   : correct!   +5
(4) Function  'is_in_subtree'  : correct!   +10
(5) Procedure 'rotate'     : correct!   +5
(6) Procedure 'recut'     : correct!   +5
(7) Procedure 'swap_module'   : correct!   +5
(8) Procedure 'swap_topology'  : correct!   +10
(9) Procedure 'get_expression'  : correct!   +15
(10) Procedure 'get_total_resource' : correct!   +5
Your final score for this MP     : 90
************************** END VERIFICATION **************************
```

Presentation Slide

The presentation slide for this MP can be found here: **presentation.pptx** or presentation.pdf Please carefully read the presentation before you start.

Grading Rubric

Functionality (90%)

1. is_leaf_node function: 5%
2. is_internal_node: 5%
3. is_in_subtree: 10%
4. rotate: 5%
5. recut: 5%
6. swap_module: 5%
7. swap_topology: 10%
8. get_expression: 15%
9. init_slicing_tree: 25%
10. get_total_resource: 5%

Style and comments (10%)

If your code doesn't compile, you will receive zero.