CAD for VLSI

References

- Algorithm for VLSI Design Automation
  - Sabih H Gerez
- High - Level Synthesis Introduction to Chip & System Design
  - Daniel D Gajski, Nikil D Dutt, Allien C-H Wu, Steve Y-L Lin
- Logic synthesis & Verification Algorithms
  - Gary D Hachtel, Fabio Somenzi
- Graph Theory with application to Engg & Computer Science
  - Narsingh Deo

Control and Data Flow graphs - CDFG

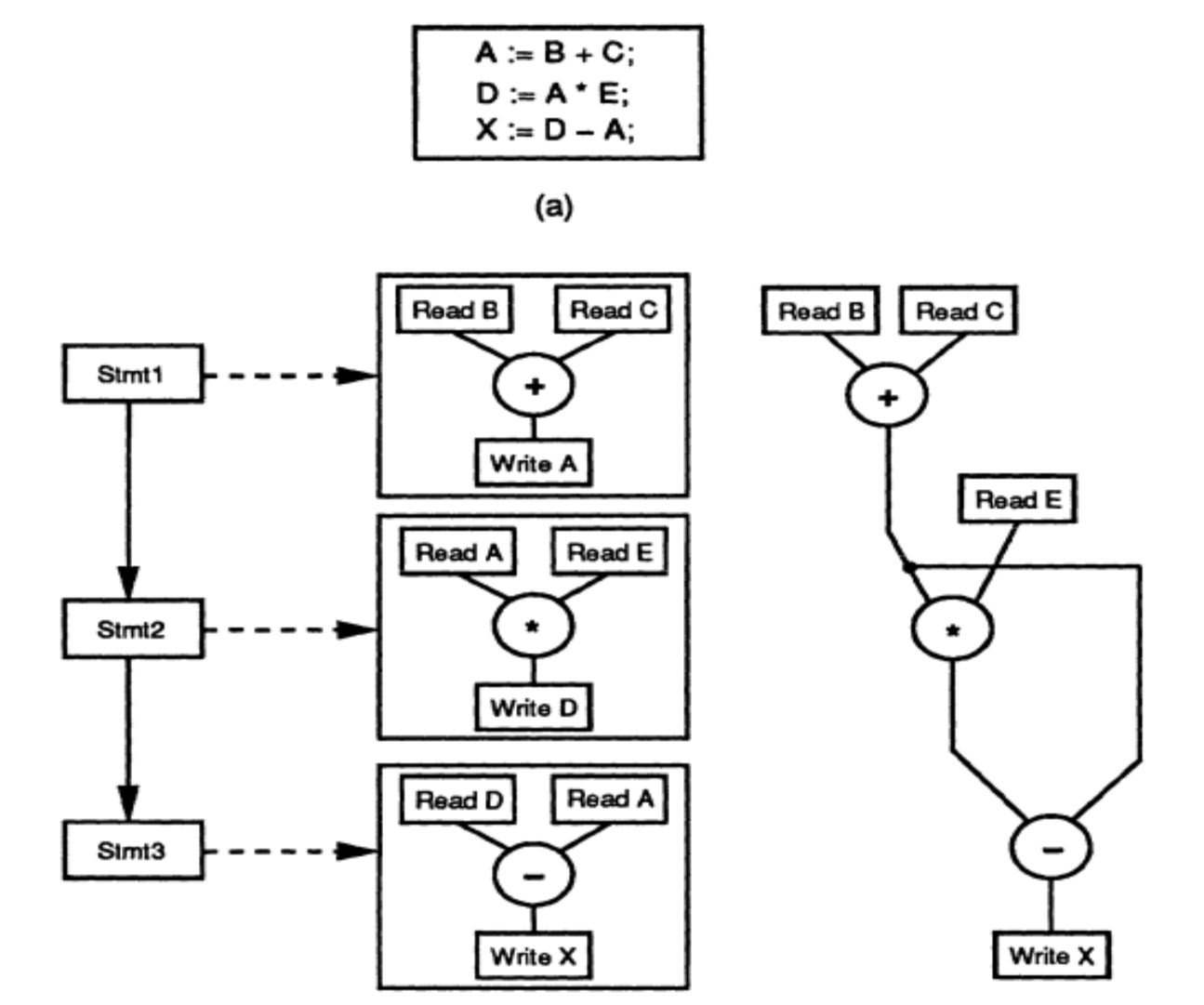
- First step in synthesis is compilation of input behavior in into a intermediate graph representation

- We perform high level synthesis tasks such as
  - Scheduling
  - Unit Selection
  - Functional, Storage and Interconnection Binding
  - Control Generation

Data Flow Graph (5.3)

- Captures operational activity
- Data flow graph generation steps :
- Generation of parse trees
- Interpret the execution ordering
  - Sequential
  - Concurrent
- Dataflow analysis
- Fusing of parse trees

Data Flow Graph - Example



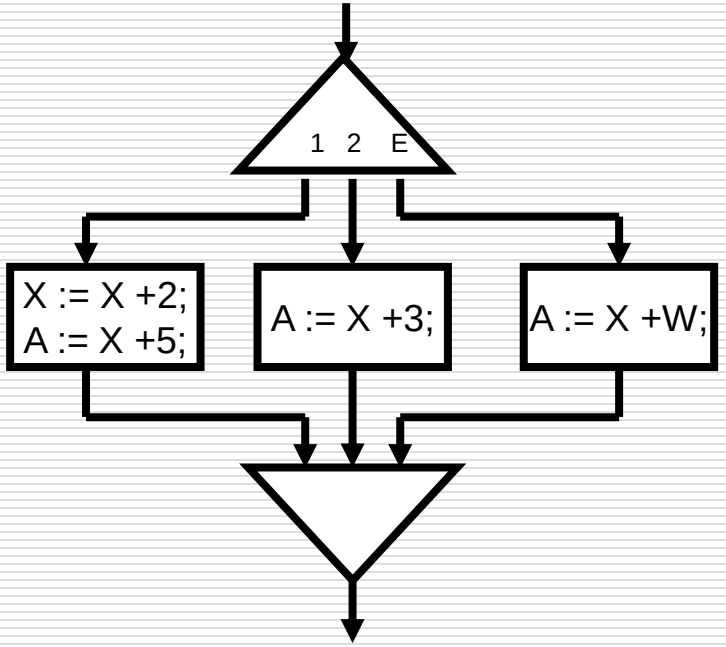
Control flow graph (5.4)

- Consists of
  - Conditional branches
  - Conditional join
  - Loops
  - Statement assignment blocks

- Each of these are represented by symbols – Triangles, Inverted triangles, rectangles

Control flow Representation

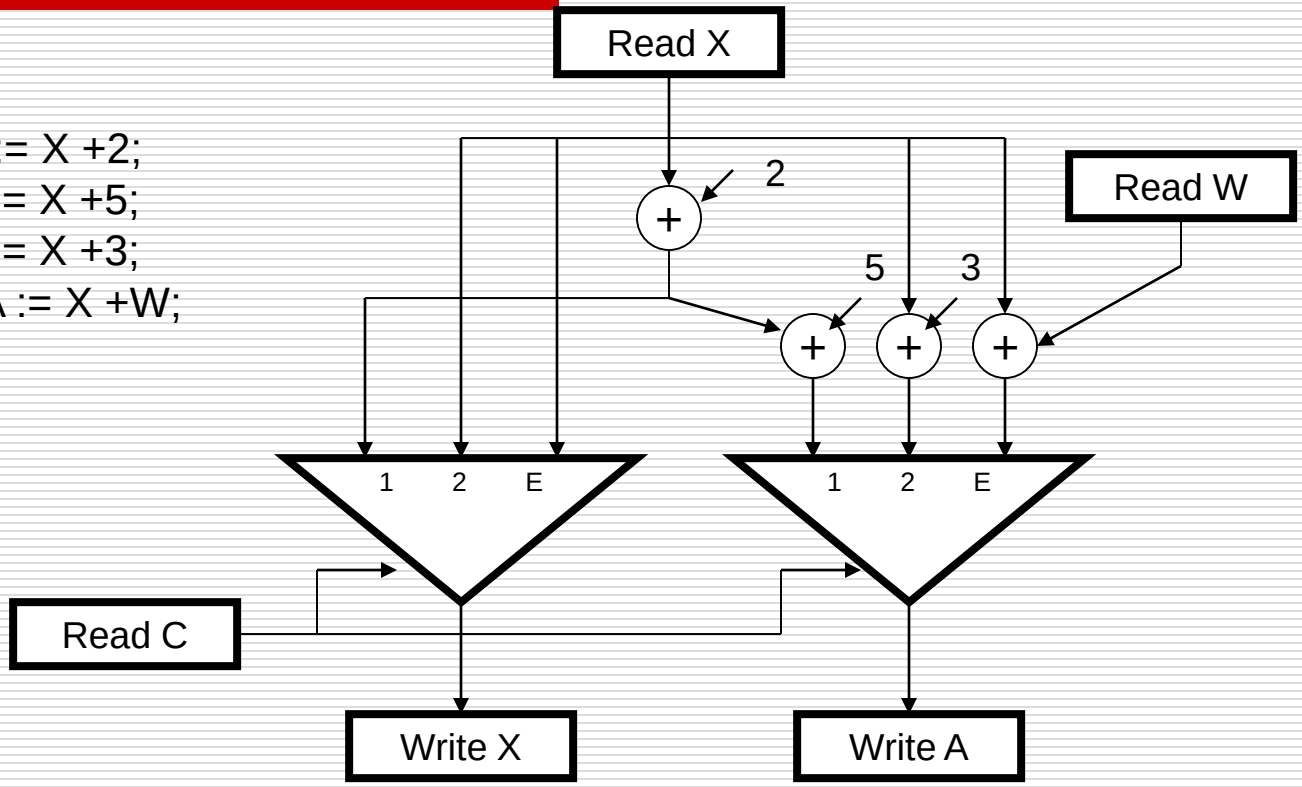
```
Case c is
      When 1 =>      X := X +2;
                     A := X +5;
      When 2 =>      A := X +3;
      When others => A := X +W;
End case;
```



Control Flow Representation

Data flow Representation

```
Case c is
      When 1 =>      X := X +2;
                     A := X +5;
      When 2 =>      A := X +3;
      When others => A := X +W;
End case;
```



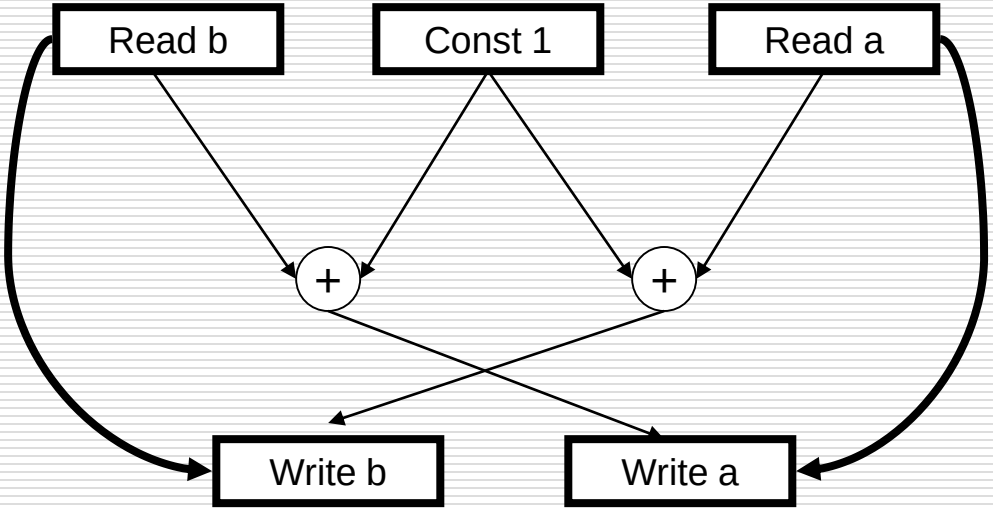
Data Flow Representation

Sequencing & Timing

Signal statement

```
b <= a + 1;
a <= b + 1;
```

Statements execute concurrently



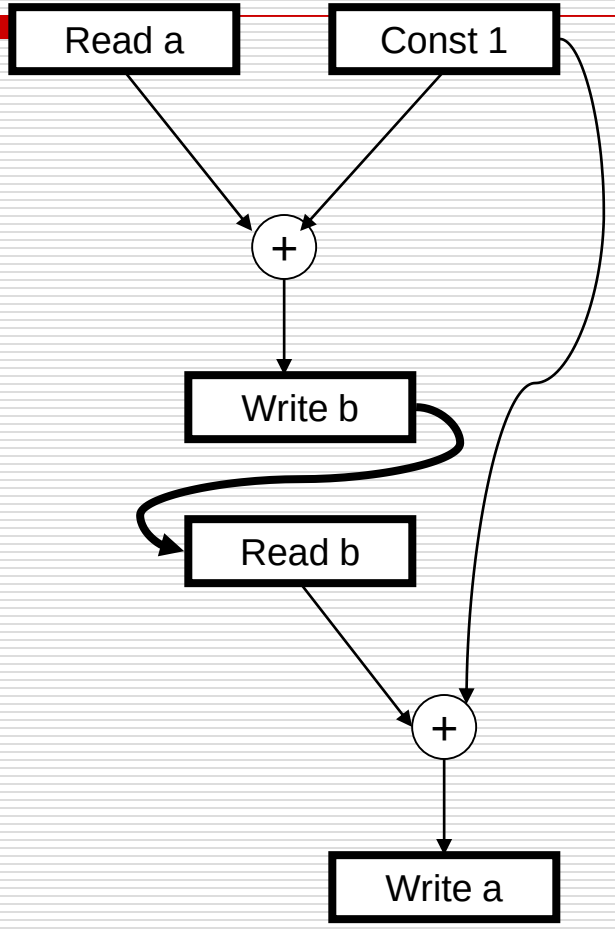
Note:

- We must ensure read operation precedes the write operation
- We must ensure that variable values are read before a new value is defined
- Conversely, a read access of a variable should not be executed before its value is defined.

Sequencing & Timing

Variable statement

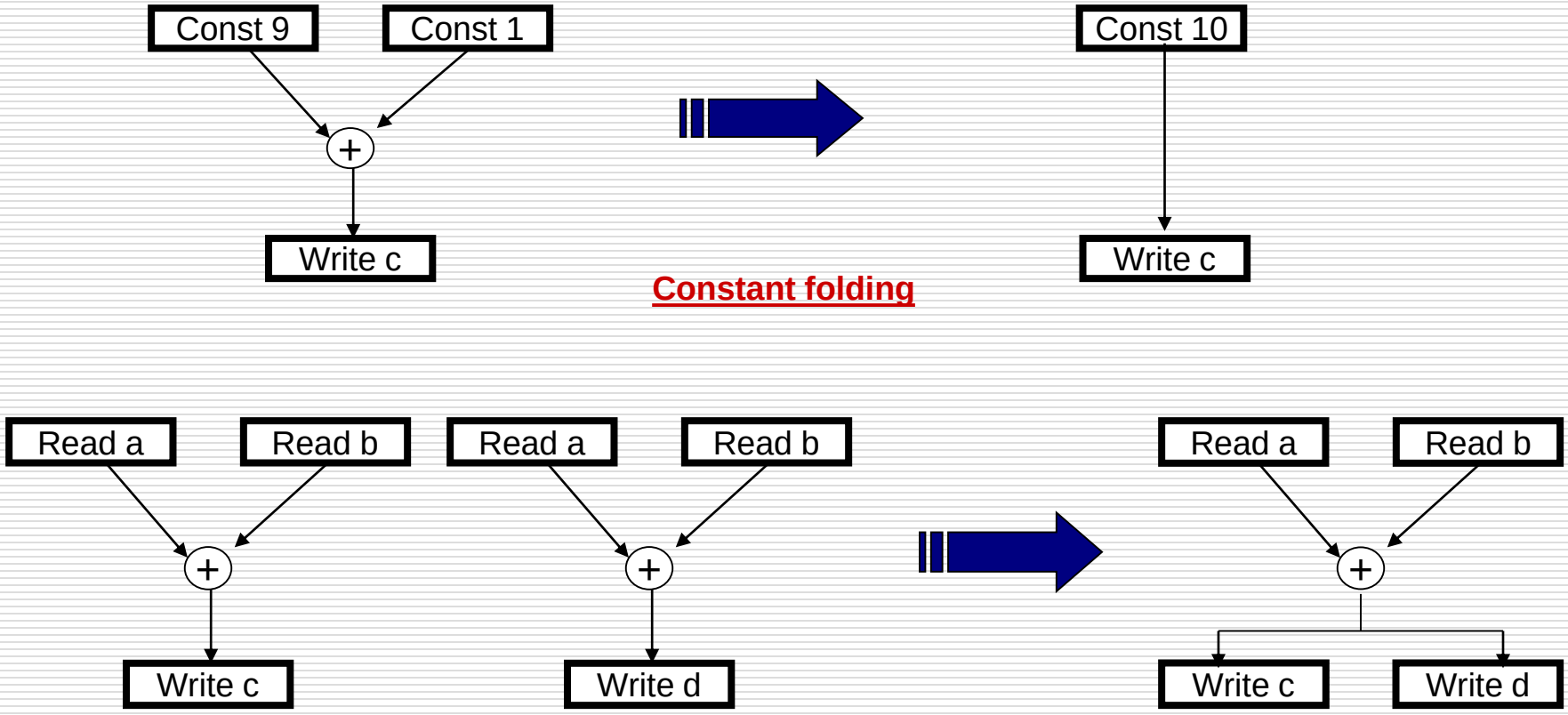
```
b := a + 1;
a := b + 1;
```



Transformations (5.7)

- Initial flow graph representation closely matches input description – language syntax and constructs
- These may not be relevant or useful to synthesis
- The original flow graph is transformed into a form, which is more amenable for high level synthesis

Compiler Transformation

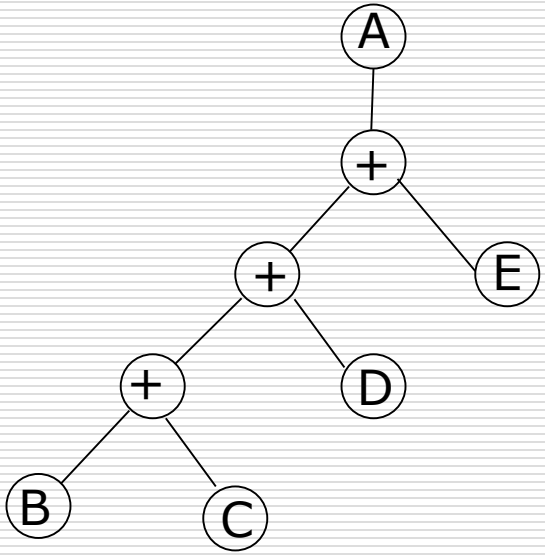


Flow Graph Transformation

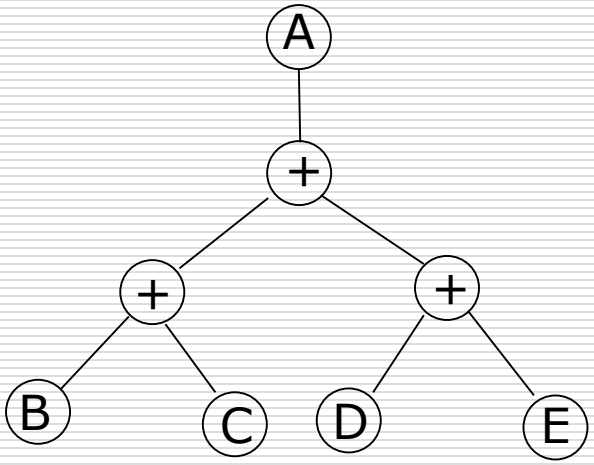
- Flow graph level transformations are done to improve parallelism in the design
- Tree height reduction
- CF - DF Transformation / Flattening of hierarchical CDFG graph into flat DFG

Flow Graph Transformation

- Tree height reduction
  - Tree height reduction uses the commutative and distributive properties of operators to decrease the height of a long expression tree
  - Exposes the potential parallelism within a complicated dataflow graph
  - Eg: A :=B + C + D + E



After tree height reduction
A := (B+C) + (D+E)

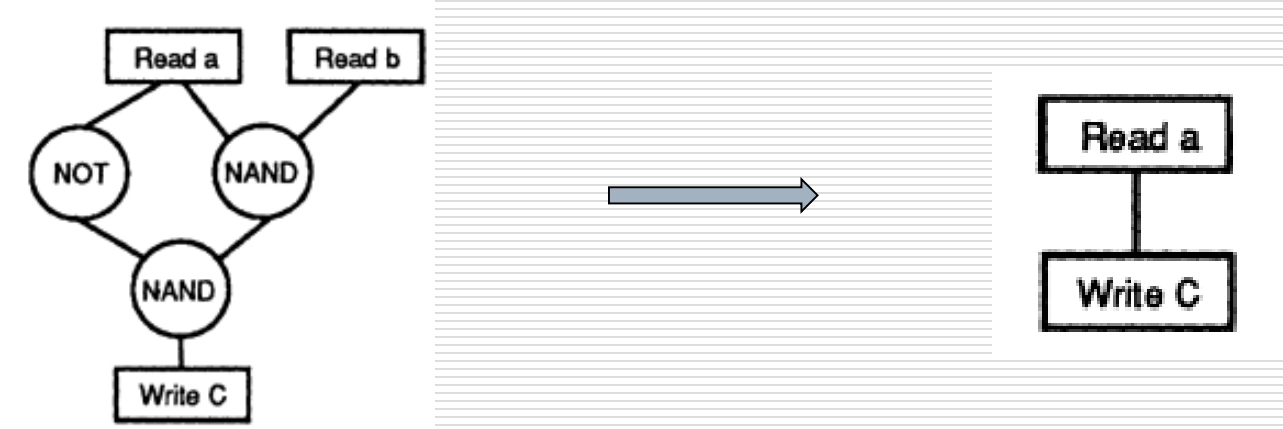


Flow Graph Transformation

- CF - DF Transformation / Flattening of hierarchical CDFG graph into flat DFG
  - Useful when Boolean variables are tested
  - Eliminates redundant states
  - Data flow representation exploits parallelism better
  - Loops in the control flow can be unrolled into DFG which exposes parallelism

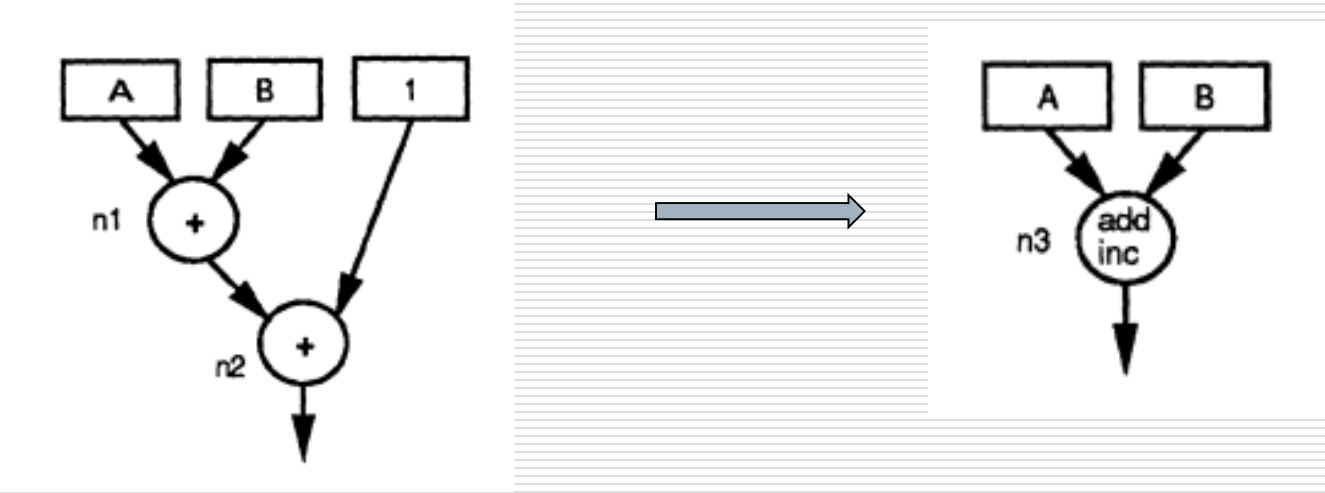
Hardware - Specific Transformations

- These make use of properties of hardware to optimize the intermediate representation
- At the logic level, we can apply local Boolean optimization techniques to locally optimize parts of a flow graph



Hardware - Specific Transformations

- At the RT level, we can use pattern matching to detect and replace portions of the flow graph with simpler flow-graph segments corresponding to functional units available in a RT library



Partitioning (Ch. 6 - Gajski)

Contents

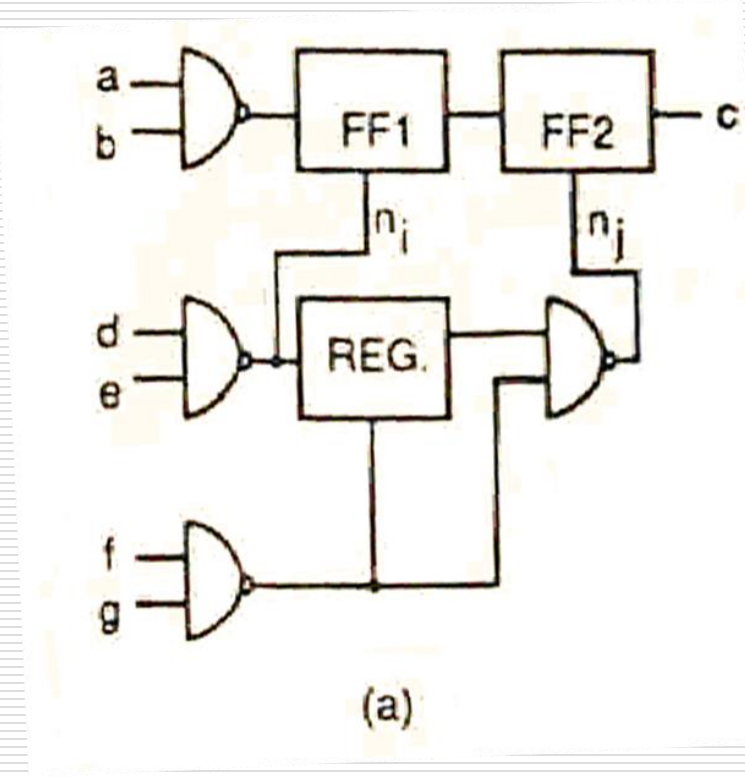
- Introduction

- Basic partitioning methods
  - → Problem Formulation
  - → Random Selection
  - → Cluster Growth
  - → Hierarchical Clustering
  - → Min-Cut Partitioning
  - → Simulated Annealing

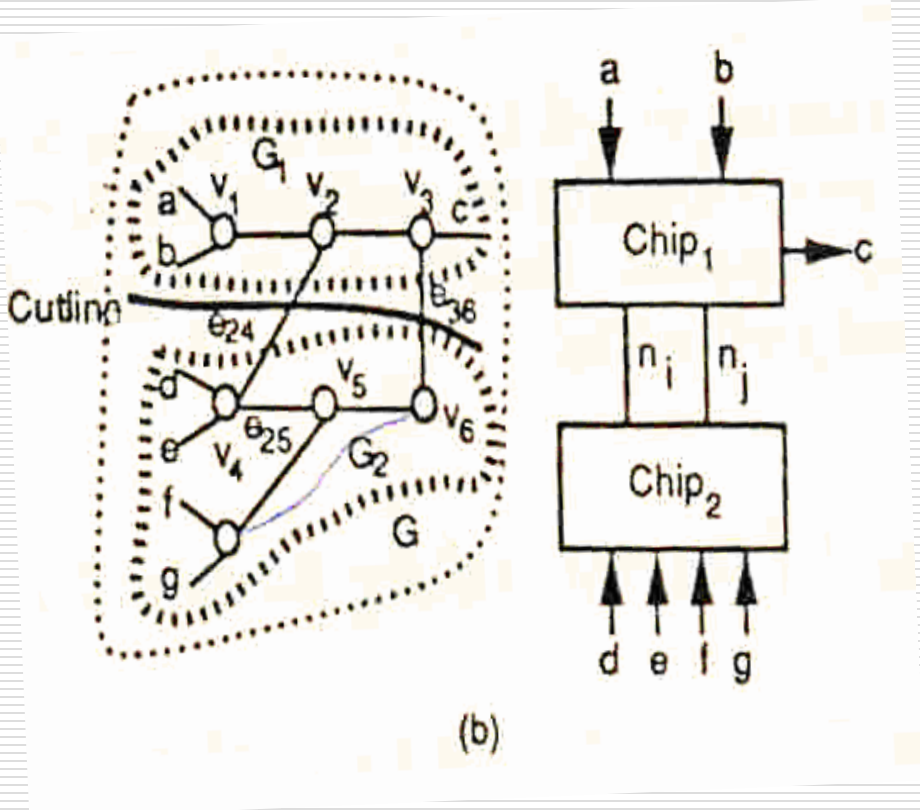
Introduction

- Partitioning is used in HLS for scheduling, allocation, unit selection, and chip & system partitioning.
- It is used to cluster variables and operations into groups so that each group is mapped into storage elements, functional unit or an interconnection unit of a real design.
- The result is used for unit selection before scheduling and binding, or it can be used for allocation.
- It is used to decompose a large behavioral description into several smaller ones, that can be solved efficiently.

Problem Formulation



(a) An example



(a) Graph & physical representation

Partitioning Techniques

- Two basic partitioning techniques

Constructive method

- Partition the graph by starting with one or more seeds of nodes & adding the nodes to the seeds one at a time

Iterative - improvement method

- Starts with initial partition, and then successively improves the result by moving objects between partitions.

Partitioning algorithms

Constructive

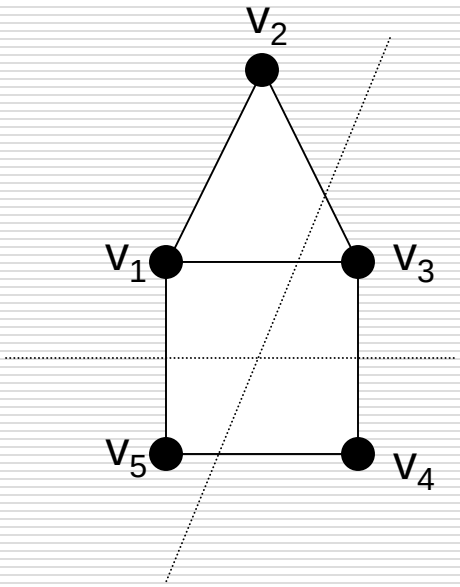
- Random selection
- Cluster growth
- Hierarchical clustering

Iterative

- Min - cut
- Simulated annealing

Random Selection

- Randomly selects the nodes one at a time and places them into cluster of fixed size until the proper size is reached.
- The procedure is repeated until all the nodes have been placed.
- Algorithm is fast, often produces the poor results



Cluster Growth

- This algorithm starts with a non partitioned set of objects
- It places objects into a given number of clusters according to some closeness measure

Three tasks:

1. **Seed selection**
   - Seed node may be specified by the designer or chosen based on certain attributes
2. **Unplaced node selection**:
   - Algorithm determines the clustering order based on some closeness measure/score
   - Closeness score can be no. of connections between unplaced & placed nodes
3. **Node placement**:
   - Adds the unplaced node with the highest closeness score to a cluster
   - The process is repeated until all nodes are placed into clusters

Cluster Growth

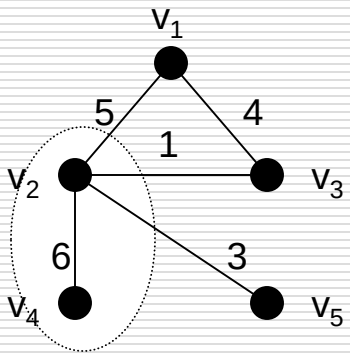
```
num_cluster = [V/m]
For I = 1 to num_cluster do
/* select a seed node */
      seed = Connected (V, V)
      Vi = {seed}
      V = V – {seed}
/* construct a cluster of m nodes */
      for j = 1 to m-1 do
            temp = Connected (V, Vi)
            Vi = Vi U {temp}
            V = V – {temp}
      end for
end for
```

- Let G =(V, E) be a graph
  V → Vertex set, E → Edge set
- Criteria :
  Size of cluster : m nodes
  Closeness measure : number of connections between vertices
- Connected (V, V_i) returns maximally connected node to V_i
- Algorithm is easy to implement, but results are not very good
- Used to generate initial partitions

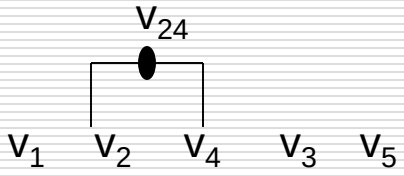
Hierarchical Clustering

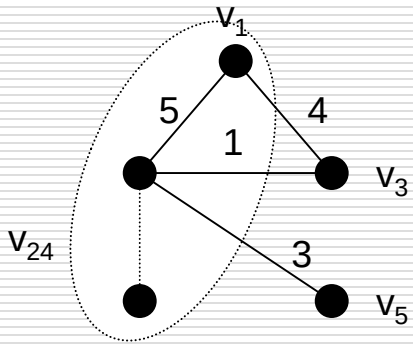
- The hierarchical-clustering algorithm considers a set of objects and groups them according to some measure of closeness
- The two closest objects are clustered first and considered to be a single object for further clustering
- The clustering process continues by grouping two individual objects, which can be objects or clusters themselves
- The algorithm stops when a single cluster is generated and a hierarchical cluster tree has formed
- Thus, any cut-line through the tree indicates set of subtrees or clusters to be cut from the tree.

Hierarchical Clustering

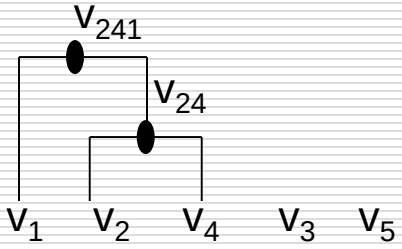


	v_1	v_2	v_3	v_4	v_5
v_1	--	--	--	--	--
v_2	5	--	--	--	--
v_3	4	1	--	--	--
v_4	0	(6)	0	--	--
v_5	0	3	0	0	--

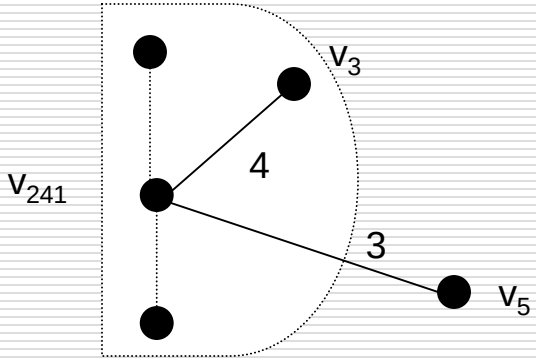




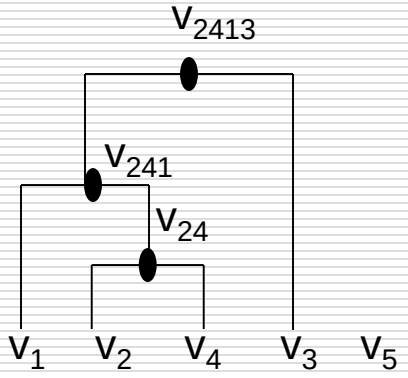
	v_1	v_{24}	v_3	v_5
v_1	--	--	--	--
v_{24}	(5)	--	--	--
v_3	4	1	--	--
v_5	0	3	0	--

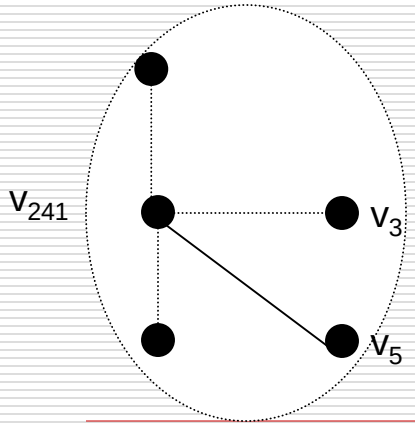


Hierarchical Clustering

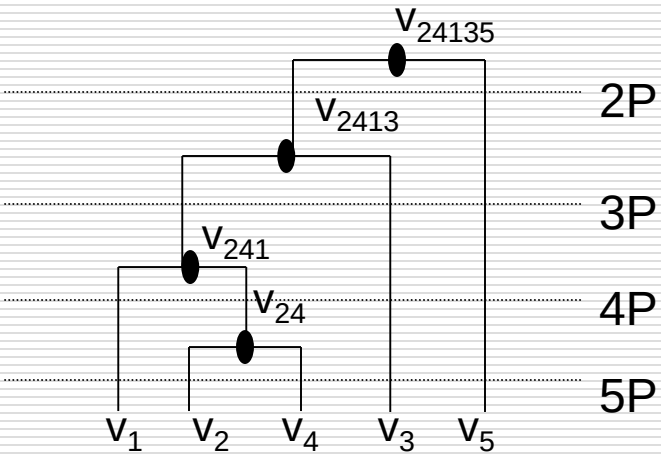


	v_{241}	v_3	v_5
v_{241}	--	--	--
v_3	(4)	--	--
v_5	3	0	--





	v_{2413}	v_5
v_{2413}	--	--
v_5	(3)	--



Hierarchical Clustering

For all $V_i \in V$ do
    $L(V_i) = 0$;
end for

/* compute closeness measure */

For all $v_i, v_j \in V$ do
    c_{ij} = Closeness (v_i, v_j)
end for

/* construct cluster tree */

For k = 1 to n-1 do
    $C_{max1, max2}$ = Maxclose (C)
    $V = V \cup \{n_k\}$
    $V = V - \{v_{max1}, v_{max2}\}$
    $Lsucc(n_k) = v_{max1}$
    $Rsucc(n_k) = v_{max2}$

/* compute closeness measure for n_k*/

$L(n_k) = \max(L(v_{max1}), L(v_{max2})) + 1$
    For all $V_m \in V$ do
        $C_{km} = \max(c_{max1,m}, c_{max2,m})$
        $c_{max1,m} = c_{max2,m} = 0$
    end for
end for

n_k _ set of non-terminal nodes

Min - Cut Partitioning

- This algorithm interchanges two groups of nodes between two partitions of equal size, to produce maximal partitioning improvement
- Widely used, produces good results using less CPU time
- Also known as the Kernighan – Lin algorithm

- The algorithm partitions a graph into two sub-graphs minimizing the connections between them

- Let us take a graph G = (V,E) with 2n vertices
- Goal is to divide the graph into two sub-graphs G1 & G2
- For each vertex(v_i) in G1, we define

Internal cost

$$IC_i = \sum C_{im}$$

External cost

$$EC_i = \sum C_{ik}$$

Min - Cut Partitioning

External & Internal cost difference

$$D_i = EC_i - IC_i$$

- Similarly IC, EC and D are calculated for all vertices(v_j) in G2

Gain of interchanging any two vertices vi and vj

$$Gain(v_i,v_j) = D_i + D_j - 2C_{ij}$$

C_{ij} ← the no of edges between v_i & v_j

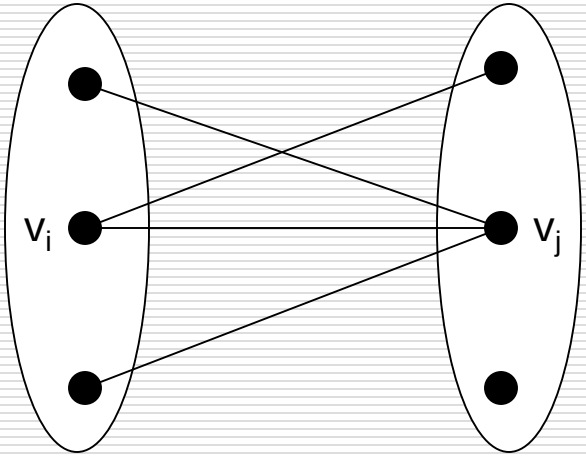
- **Note**

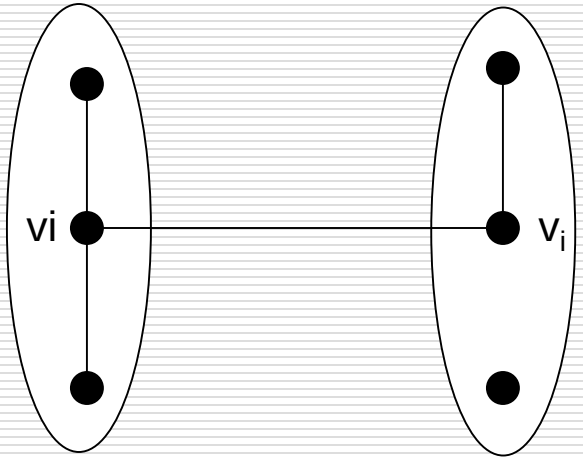
If gain is +ve, cost(no. of external edges) ↓

If gain is 0, no improvement

If gain is –Ve, cost(no. of external edges) ↑

Min - Cut Partitioning





$IC_i = 0$ $IC_j = 0$ $IC_i = 2$ $IC_j = 1$

$EC_i = 2$ $EC_j = 3$ $EC_i = 1$ $EC_j = 1$

$D_i = 2 - 0 = 2$ $D_j = 3 - 0 = 3$ $D_i = 1 - 2 = -1$ $D_j = 1 - 1 = 0$

Gain = 2 + 3 – 2(1) = 3

Gain = 0 + (- 1) – 2(1) = - 3

Simulated Annealing

T = initial_temperature
S_{old} = initial_partition
C_{old} = cost (S_{old})
While stopping criterian is not satisfied do
    while inner loop criterion is not satisfied do
        S_{new} = Generate (S_{old})
        C_{new} = cost (S_{new})
        $\Delta C = C_{new} - C_{old}$
        x = F(Δc,T)
        r = Random(0,1)
        If r < x then
            $S_{old} = S_{new}$
            $C_{old} = C_{new}$
        end if
    end while
    T = Update (T)
end while

Scheduling (Ch. 7 - Gajski)

Introduction & Assumption

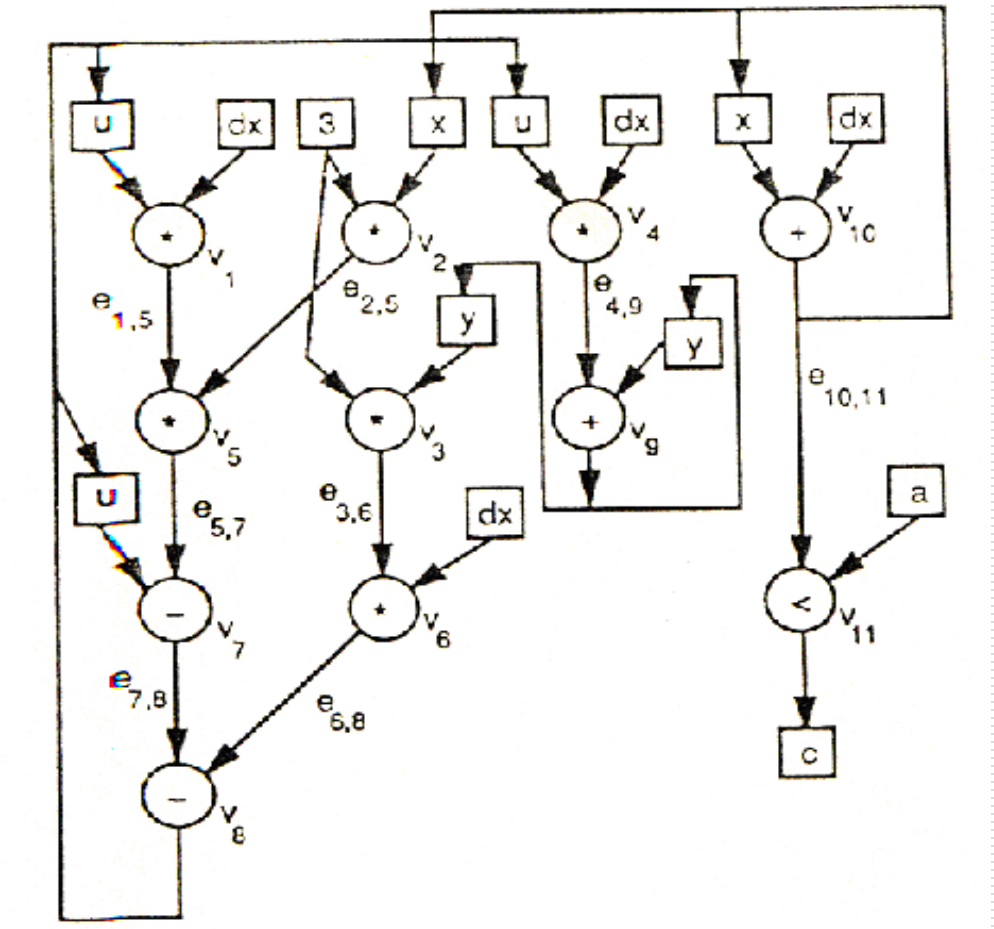
- Scheduling is nothing but partitioning of set of operations with respect to time.

- Assumptions:
  - No Multi Functional Units
  - All functional units with same execution times
  - No looping constructs in the behavioral description

- ASAP (As Soon As Possible) and ALAP (As Late As Possible) algorithms of scheduling give the earliest and latest time bounds within which an operation can be scheduled.

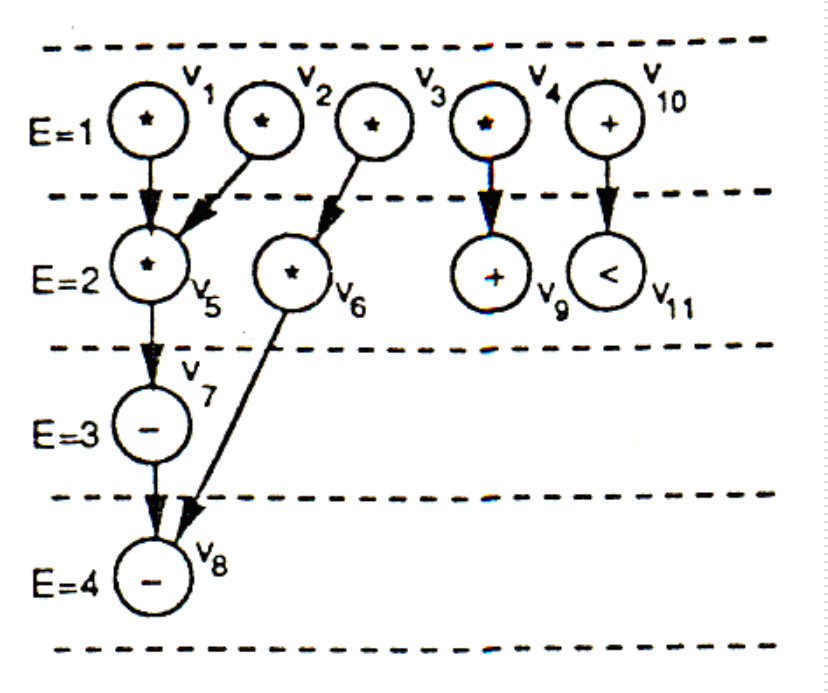
Example

```
While (x<a) do
      x1:= x + dx;
      u1:= u –
            (3 * x * u * dx) –
            (3 * y * dx);
      y1:= y + (u * dx);
      x:= x1;
      u:=u1;
      y:=y1;
end while
```



ASAP Scheduling

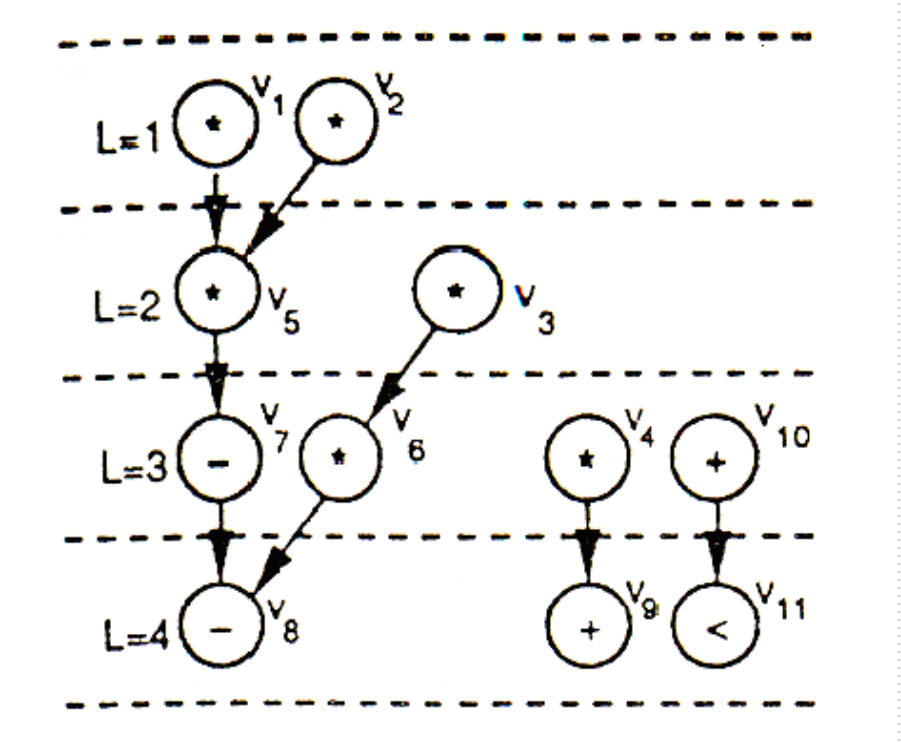
```
For each node vi ∈ V do
        If Predvi = ϕ then
                Ei = 1;
                V = V – {vi};
        Else
                Ei = 0;
        End if
End for

While V ≠ ϕ do
        For each node vi ∈ V do
        If ALL _NODES _SCHED (Predvi, E)
        then
                Ei = MAX(Predvi, E) +1;
                V = V – {vi};
        End if
        End for
End while
```



ALAP Scheduling

For each node vi $\in$V do
    If Succvi = ϕ then
        Li = T;
        V = V – {vi};
    Else
        Li = 0;
    End if
End for

While V $\neq$ ϕ do
    For each node vi $\in$ V do
    If ALL_NODES_SCHED(Succvi, L) then
        Li = MIN(Succvi, L) – 1;
        V = V – {vi};
        End if
    End for
End while



Scheduling Algorithms

Time Constrained

- Given the number of control steps, try to optimize the resources required for implementation.

Types

- Force directed scheduling method
- Iterative Rescheduling method

Resource constrained

- Given the number of resources, try to optimize the control steps into which the operations are scheduled.

Types

- List based Scheduling method
- Static List based scheduling method

Force Directed Scheduling

```
Call ASAP (V); Call ALAP (V);
While there exists oi such that Ei ≠ Li do
      Max Gain = - ∞;
/* Try scheduling all unscheduled operations to every state in its range */
      for each oi, Ei ≠ Li do
            for each j, Ei ≤ j  ≤ Li do
                        Swork = SCHEDULE_OP(Scurrent, oi, sj);
                        ADJUST_ DISTRRIBUTION (Swork, oi, sj);
                         If COST(Scurrent) – COST(Swork) > Max Gain then
                                    Max Gain = COST(Scurrent) – COST(Swork);
                                    BestOp =oi; BestStep = sj;
                        End if
            End for
      End for
Scurrent = SCHEDULE _ OP(Scurrent, BestOp, Best Step);
ADJUST _ DISTRIBUTION(Scurrent, BestOp, BestStep);
End while
```

Force Directed Scheduling

- The main goal of the algorithm is to reduce the total number of functional units needed in the implementation
- The algorithm tries to achieve this by uniformly distributing operations of same type across all available states.

Steps

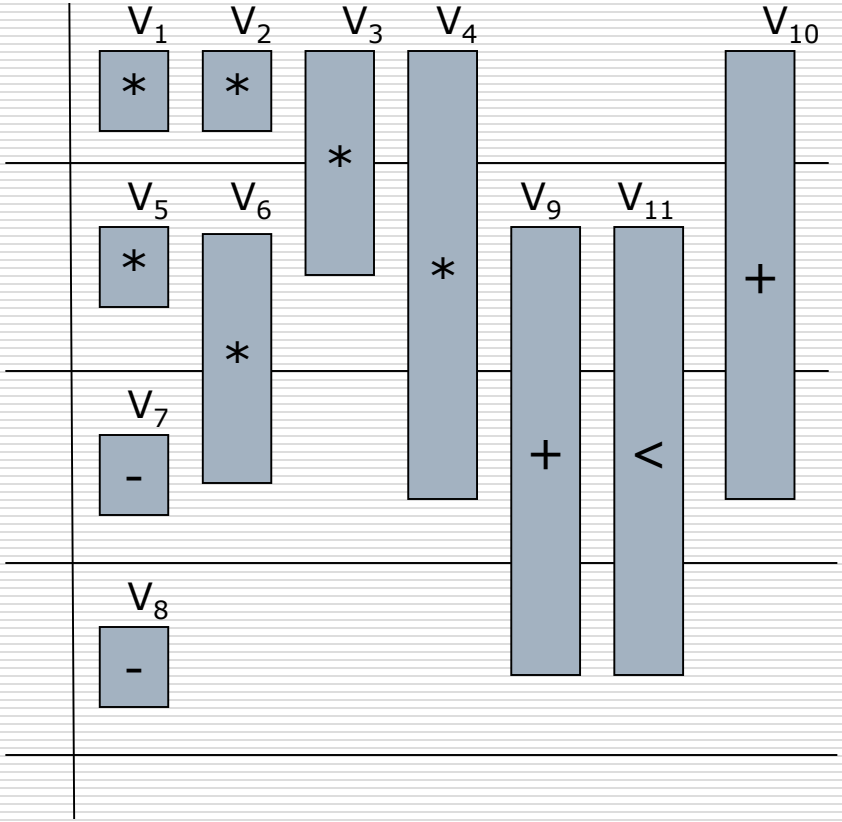
- Calculate ASAP, ALAP
- $P(O_i) = 1 / ((L_i - E_i) + 1)$, the probability that operation Oi will be scheduled in a state in the range.
- $EOC = C_{ik} * \sum P_j(O_i)$

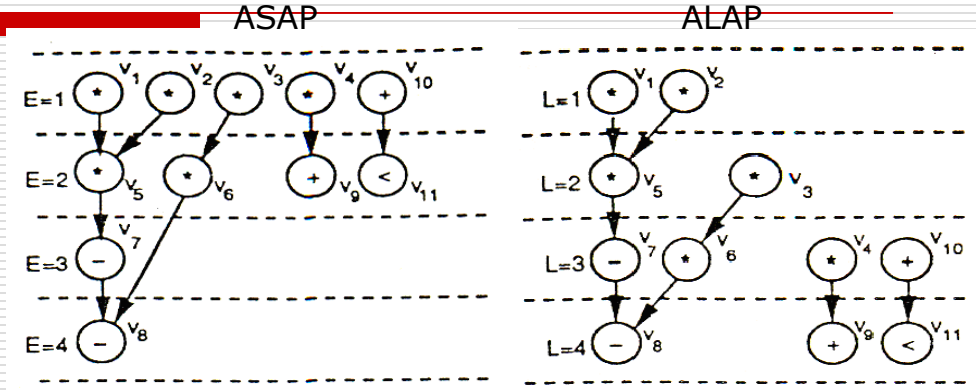
  - $P(O_i)$ – Probability of distribution
  - EOC – Expected Operator Cost
  - O – Operation
  - J – J^{th} Control Step

Force Directed Scheduling

- A set of probability distribution graphs(bar charts) are created for each operation type for each control step
- This bar graph represents EOC(expected operator cost) in each control step
- EOC= $C_{ik} * \sum P_j(O_i)$ where C_{ik} is the cost of the functional unit performing operation of type k

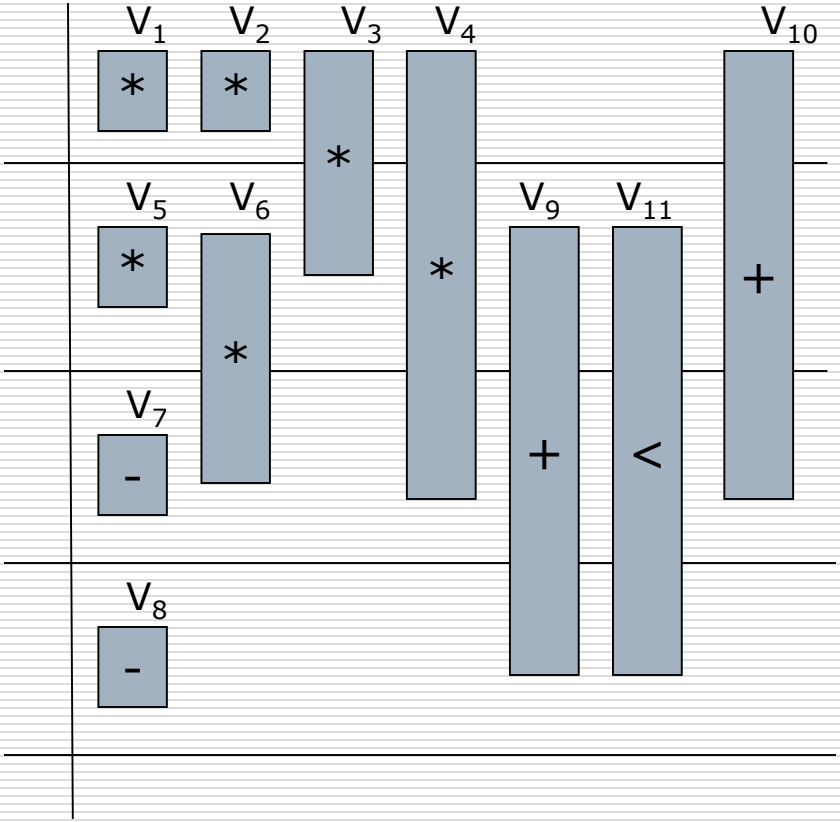
Force Directed Scheduling





$P(O_1) = 1 /((1-1) + 1) = 1$
$P(O_2) = 1 /((1-1) + 1) = 1$
$P(O_3) = 1 /((2-1) + 1) = 0.5$
$P(O_4) = 1 /((3-1) + 1) = 0.33$
$P(O_5) = 1 /((2-2) + 1) = 1$
$P(O_6) = 1 /((3-2) + 1) = 0.5$
$P(O_7) = 1 /((3-3) + 1) = 1$
$P(O_8) = 1 /((4-4) + 1) = 1$
$P(O_9) = 1 /((4-2) + 1) = 0.33$
$P(O_{10}) = 1 /((3-1) + 1) = 0.33$
$P(O_{11}) = 1 /((4-2) + 1) = 0.33$

Force Directed Scheduling



EOC Calculation

EOC for * Operation

- 1 + 1 + 0.5 + 0.33 = 2.83
- 1 + 0.5 + 0.33 + 0.5 = 2.33
- 0.5 + 0.33 = 0.83
- 0 = 0.0

Force Directed Scheduling

- Since the functional units can be shared across the control steps, the maximum of EOC over all states gives a measure of cost of implementing all operations of that type in a design

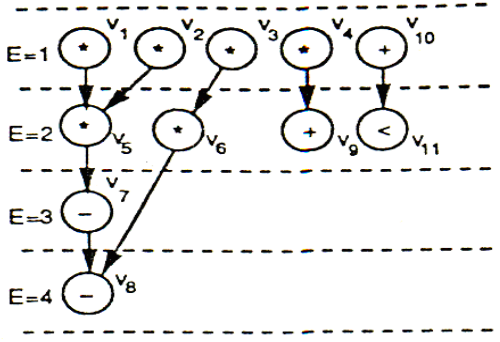
- Since the main goal of FDS is efficient sharing of functional units across all states, it attempts to balance EOC value of each operation type across all available states by appropriately moving operations between the sates.

Static - List Based Scheduling

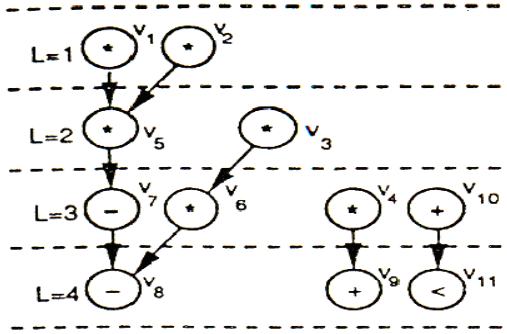
- This algorithm is resource constrained type
- A priority list is created before starting scheduling
- The priority list is created by sorting all operations using ALAP labels as the primary key and ASAP labels as the secondary key in descending order
- If both keys have the same value, an arbitrary ordering is used
- This sorted list is maintained as the priority list that determines the order in which the operations are scheduled

Static - List Based Scheduling

Priority list generation



ASAP



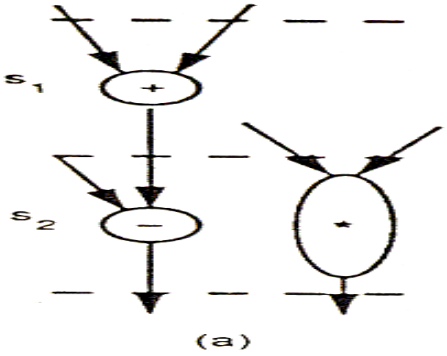
ALAP

	V_8	V_9	V_{11}	V_7	V_6	V_4	V_{10}	V_5	V_3	V_1	V_2
ALAP	4	4	4	3	3	3	3	2	2	1	1
ASAP	4	2	2	3	2	1	1	2	1	1	1
Priority	1	2	3	4	5	6	7	8	9	10	11

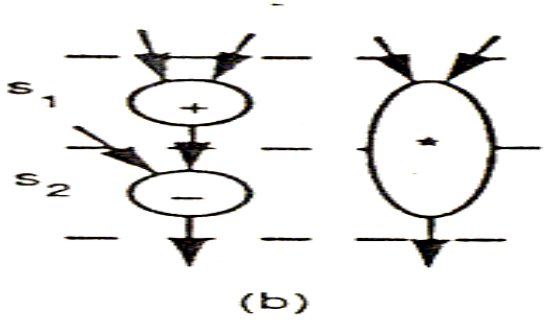
Static - List Based Scheduling

- Operations are scheduled sequentially starting with the last operation(highest priority)
- An operation is scheduled as early as possible, subject only to the availability of resources and operator dependencies
- Thus V2 is scheduled first in first control step – s1
- Assuming 2 multipliers are available V1 is also scheduled in the same control step (s1)
- V3 can't be scheduled in the same step as no more multipliers are available and hence it is scheduled into s2

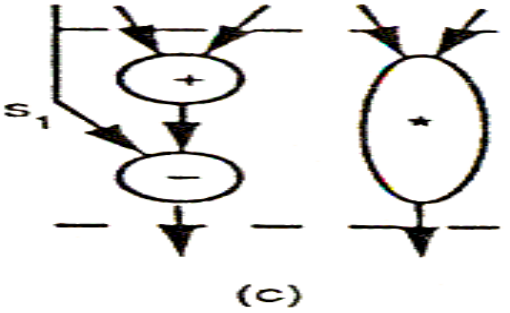
Functional Units with Varying Delays



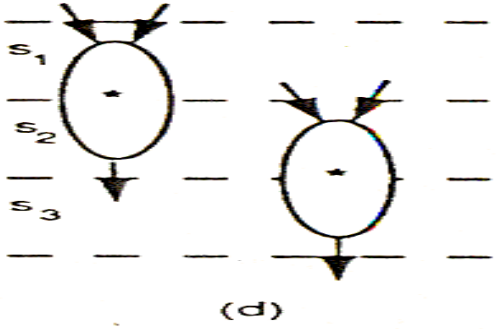
Schedule in which every operation is executed in one step



Schedule with a 2-cycle multiplier



Schedule with a chained adder & subtractor



Schedule with a 2-stage pipelined multiplier

Multi-functional units and units with multiple implementations

- Multifunctional units are area efficient compared to set of uni-functional units, they are chosen whenever available
- Libraries usually have multiple implementations of the same function
- The main goal of the algorithm is to do operation scheduling and component selection
- Efficient components are selected, so that the operations on the critical path are implemented with faster components, and that not on the critical path are implemented with slower components.
- Algorithm also ensures sharing of the functional units across various states for optimization

Realistic Design Descriptions

- A realistic scheduling algorithm must be able to deal with both of the following modeling constructs, as behavioral descriptions generally contain them
  - Conditional Constructs
  - Loops Constructs

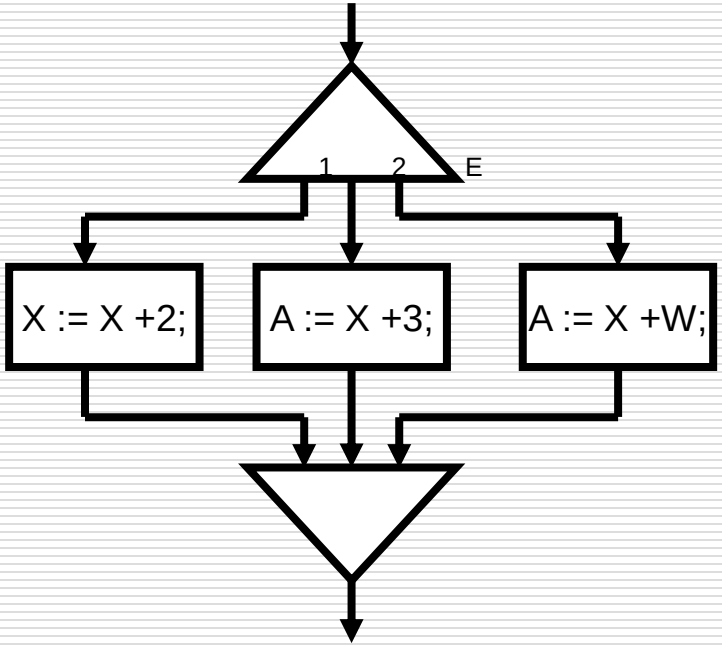
Conditional Constructs

```
case c is
      When 1 =>          X := X +2;
      When 2 =>          A := X +3;
      When others =>  A := X +W;
End case;
```

Scheduling algorithm shares the resources among mutually exclusive operations

Each of the addition operations can be scheduled in the same control step even if only one adder is available

Control Flow Representation



Loop Constructs

- Loop Scheduling
  - Sequential execution
  - Partial loop unrolling
  - Loop folding

Sequential Execution

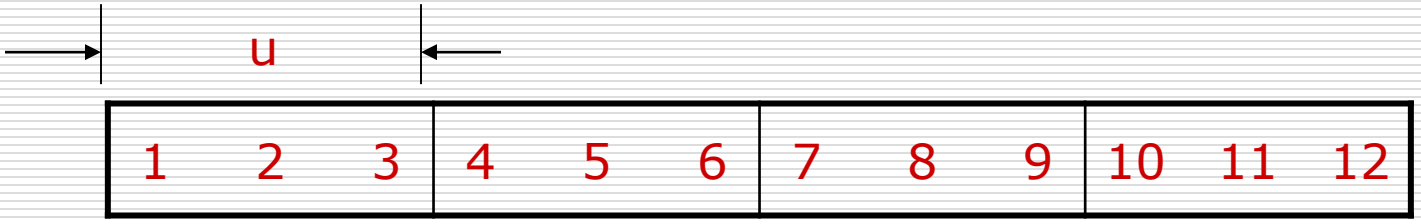
Each loop iteration takes 'b' control steps for execution

Total execution time is 12b control steps

Loop Constructs

Partial Loop Unrolling

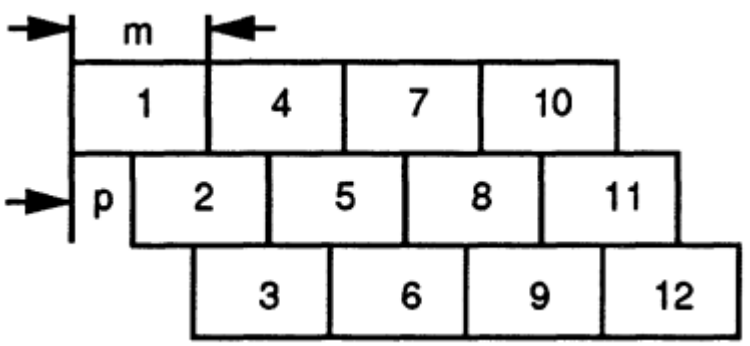
- A certain number of loop iterations are unrolled.
- This will result into a larger loop body with fewer iterations
- The larger loop body provides greater flexibility for optimizations
- If u < 3b, the total execution time is less than 12b control steps
- Can be used when loop bound is known in advance



Loop Constructs

Loop Folding

- Successive iterations of the loops are overlapped in a pipelined fashion
- A loop body takes 'm' control steps to execute
- We initiate new iterations every p control steps, where p < m, we overlap successive iterations
- The total execution time is m + (n-1) * p control steps



Allocation

Contents

- Problem definition
- Datapath architecture
- Allocation tasks
  - Unit selection
  - Functional unit binding
  - Storage binding
  - Interconnection binding
  - Interdependence & ordering
- Greedy constructive Approaches
- Decomposition approaches
  - Clique partitioning
  - Left edge algorithm

Problem Definition

- Scheduling assigns operations to control steps
  - Converts a behavioral description into set of register transfer operations that can be described by a state table
- To implement a design
  - Control unit is derived from control step sequence
  - Datapath is derived from register transfer operations assigned to each control step
- The task of realizing the datapah is called ‘datapath synthesis’ or ‘datapah allocation’ or ‘allocation’
- Allocation is performed on scheduled CDFG of a design

Problem Definition

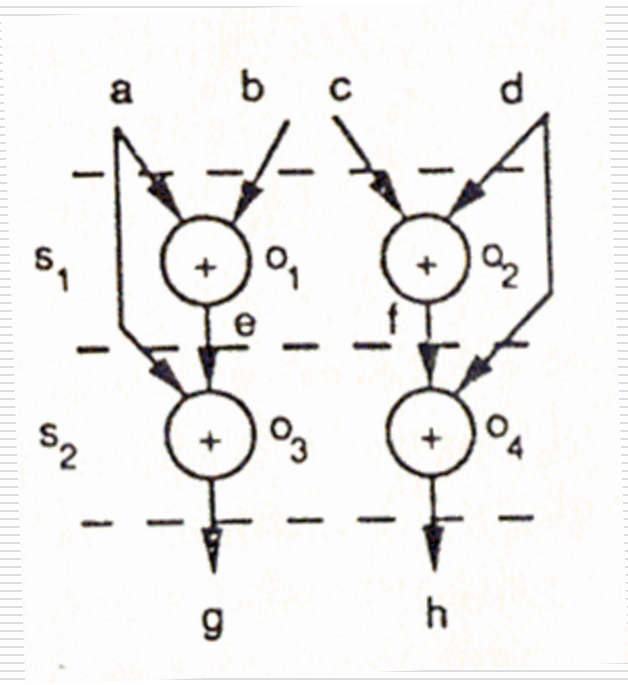
- An RTL datapah has 3 types of RT components or units
  - Functional units
  - Storage units
  - Interconnection units

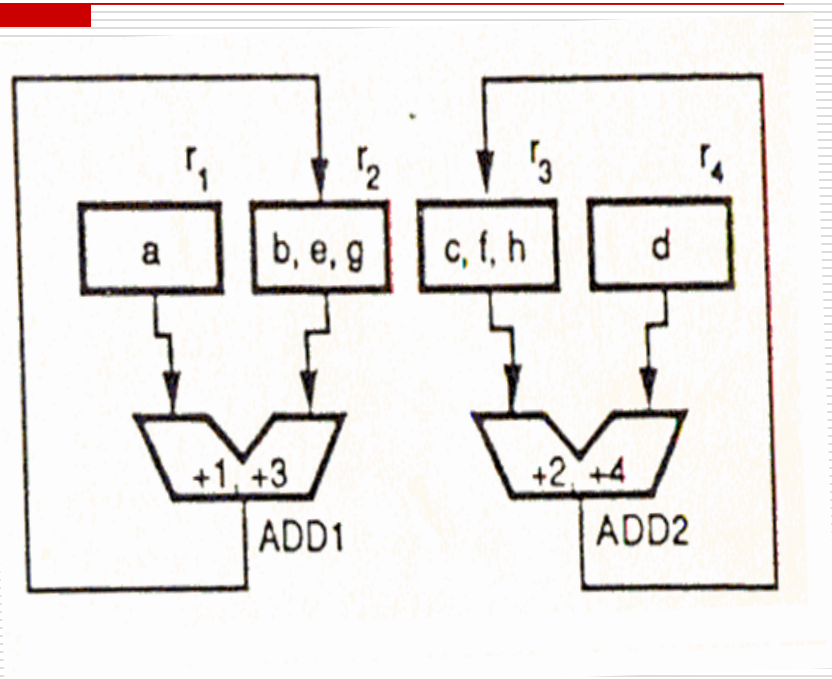
- Datapath allocation consists of 2 tasks for each type of RT units/components
  - Unit selection
  - Unit binding

- Unit selection
  - Determines the number and types of RT units/components to be used in the design

- Unit binding - involves mapping of
  - Operations to Functional units
  - Variables to Storage units
  - Data transfers to Interconnection units

Problem Definition





Two basic quality measures for datapath allocation can be

- The total size/area/cost and
- The worst case register –to-register delay/speed of the design

Allocation Tasks

- Datapath synthesis or allocation consists of following interdependent tasks,
  - Functional unit allocation
  - Storage allocation
  - Interconnection allocation

- This involves Unit selection and Unit binding of Functional, Storage and interconnection units

Allocation Tasks

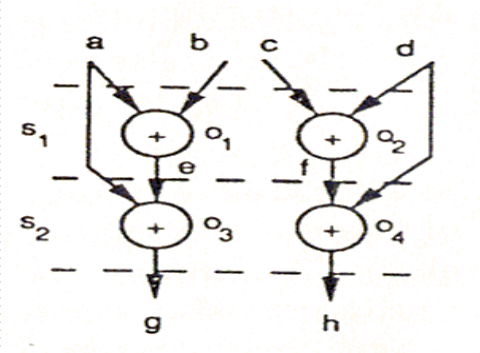
Unit selection (of Functional, Storge & interconnection units)

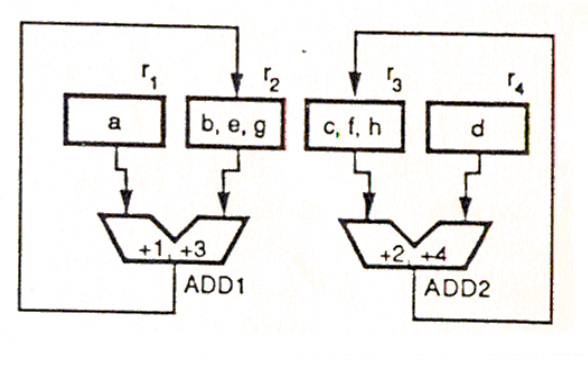
- This step selects the number and types functional, storage and interconnection units from the component library.
- RT components library contains multiple types of functional units, each with different characteristics (functionality, size, delay, power…)
- A basic requirement of the unit selection is that the number of units performing a certain type of operations must be equal to or greater than the maximum number of operations of that type scheduled in any control step.

Allocation Tasks

Functional - Unit Binding

- Behavioral description must be mapped into the set of selected functional units
- Whenever we have operations that can be mapped onto more than one functional unit, the binding algorithm needs to determine the exact mapping of the operations to functional units





Allocation Tasks

Storage Binding

- This step maps data carriers (constants, variables) in the behavioral description to storage elements in the datapath.
- Variables whose life time intervals do not overlap with each other may share the same register
- The registers can be merged into a register file with a single access port if the registers in the file are not accessed simultaneously.
- Registers can be merged into multi-port register file as long as the number of registers accessed in each control step does not exceed the number of ports.

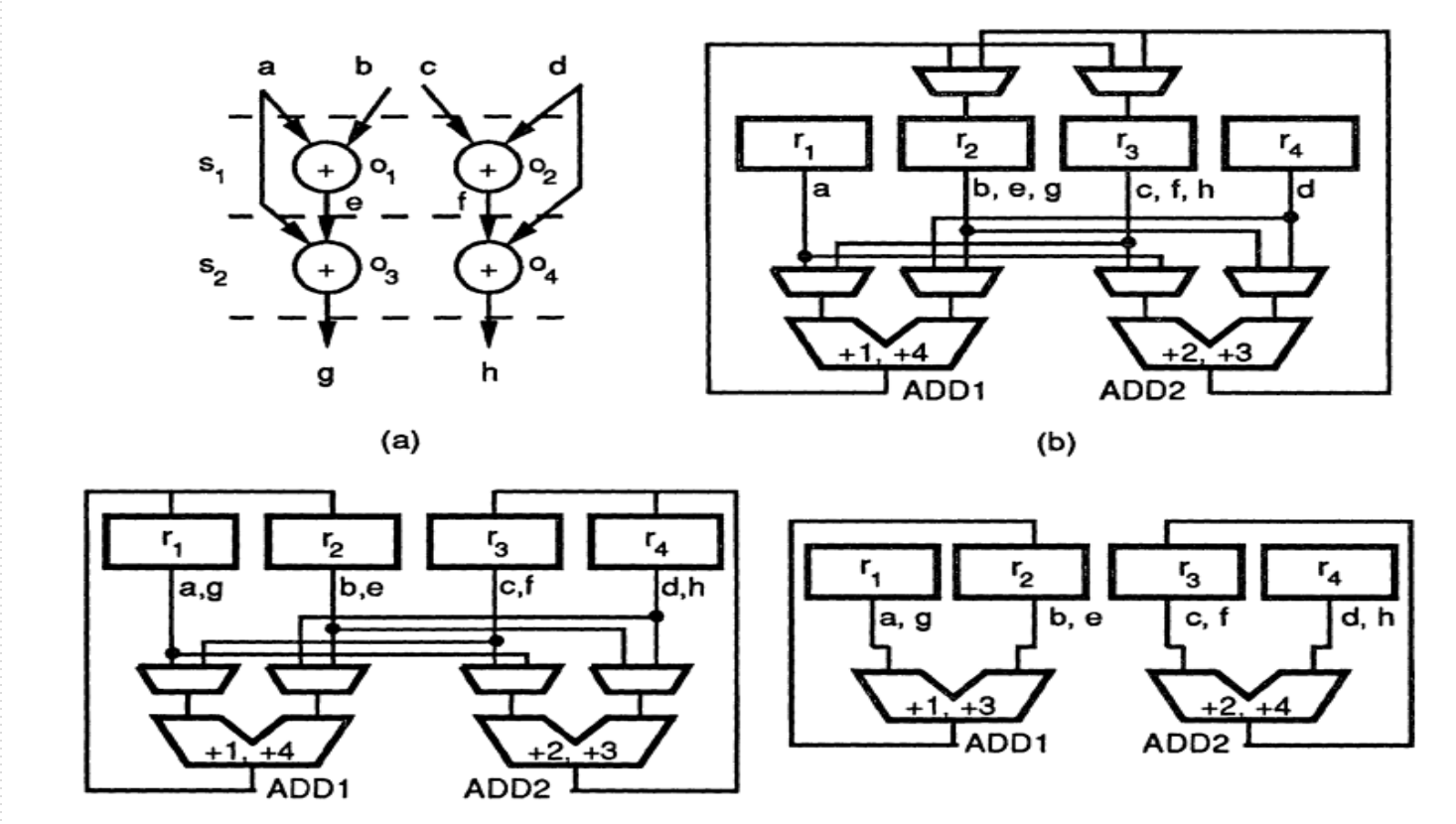
Allocation Tasks

Interconnection Binding

- Every data transfer needs an interconnection path from its source to its sink
- Two data transfers can share all or part of the interconnections path if they do not take place simultaneously.
- The main objective of interconnections binding is to maximize the sharing of interconnections units & minimize the interconnection cost.

Allocation Tasks

Interdependence & Ordering



Greedy Constructive Allocation

$DP_{current} = \phi$;
While UBE $\neq \phi$ do
    LowestCost = ∞;
    for all ube $\in$ UBE do
        DP_{work} = ADD($DP_{current}$, ube);
        C_{work} = COST(DP_{work});
        If C_{work} < LowestCost then
            LowestCost = C_{work};
            BestEntity = ube;
        end if
    end for
    D Pcurrent = ADD($DP_{current}$, BestEntity);
    UBE = UBE – BestEntity;
end while

$DP_{current}$ ← partially designed Datapath

UBE ← Unallocated behavioral entity

Decomposition Approach

- Decomposition Approach
  - Clique partitioning
  - Left edge algorithm

Clique Partitioning

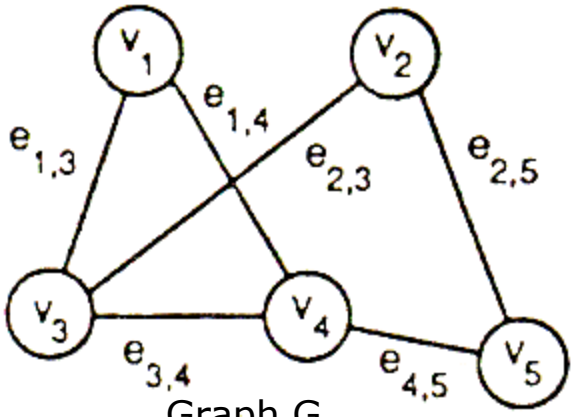
- A clique of G is a complete subgraph of G
- In a complete subgraph there exists an edge between every pair of its vertices
- Partitioning a graph into minimal no. of cliques such that each node belongs to only one clique is called clique partitioning.

Clique Partitioning

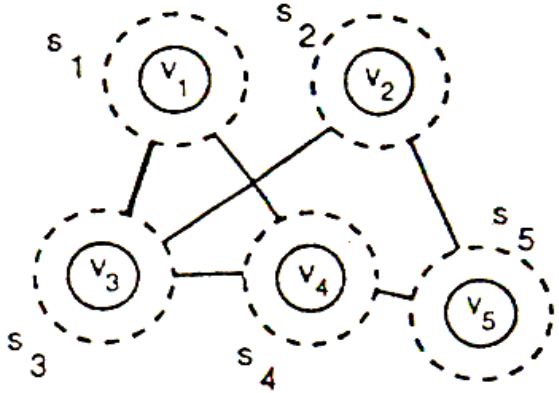
Method

- A super graph is obtained from the main graph
- Each node in the supergraph is a supernode that can contain a set of one or more vertices
- Edges in supergraph link supernodes
- A function returns set of supernodes that are common neighbors
- A supernode is a common neighbor of the two supernodes if there exist edges between the common neighbor and the two supernodes
- Initially each vertex is treated as a separate supernode
- Algorithm finds out supernodes connected by an edge and having max. no. of common neighbors

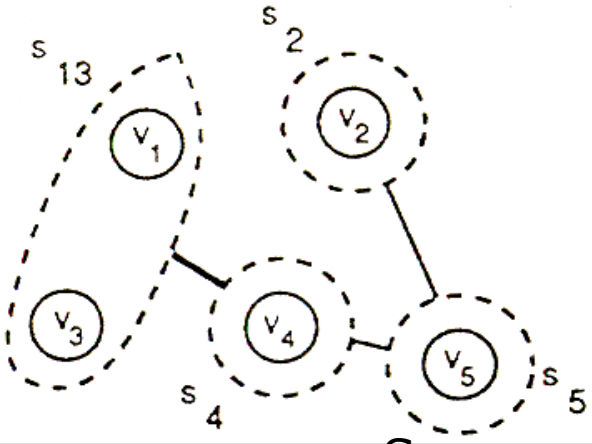
Clique Partitioning



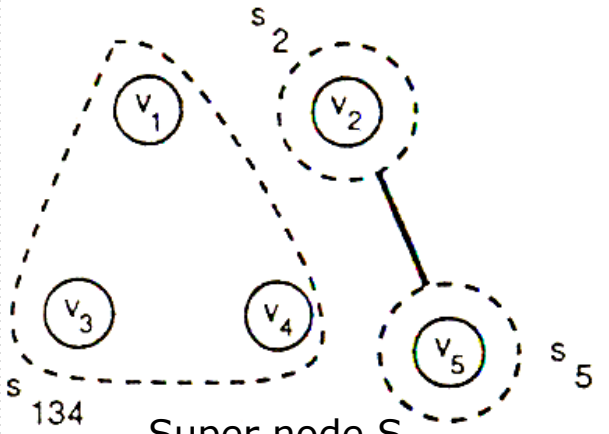
Graph G



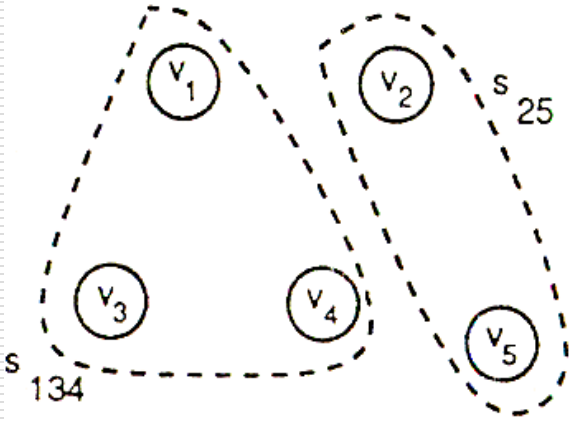
Calculating common neighbor



Super node S_{13}



Super node S_{134}



Super node S_{25}

cliques

$S_{134} = \{ v_1 , v_3, v_4\}$

$S_{25} = \{ v_2 , v_5 \}$

Clique Partitioning

```
/* create a super graph G′(S,E′) */
S = ϕ;      E′ = ϕ;
For each vi ∈ V do
      si = {vi};
      S = S U {si};
end for
For each ei,j ∈ E do
      E′ = E′U{e′i,j};
end for

While E′ ≠ ϕ do
/* Finding common neighbors */
MostCommons = -1;
for each e′i,j ∈ E′ do
      Ci,j=|COMMON_NEIGHBOUR(G′,Si,SJ)|;
      if Ci,j > MostCommons then
            MostCommons = Ci,j;
            Index1 = i; Index2 = j;
      end if
end for
```

```
CommonSet    =    COMMON_NEIGHBOUR(G′,
      SIndex1, sIndex2);

/*delete edges linking SIndex1 or SIndex2 */

E′ = DELETE _ EDGE(E′, SIndex1 );
E′ = DELETE _ EDGE(E′, SIndex2);

/*merge SIndex1 and SIndex2 into SIndex1Index2 */

SIndex1Index2 = SIndex1 U s SIndex2;
S = S – SIndex1 – SIndex2;
S = S U {SIndex1Index2};

/* add edge from  SIndex1Index2 to super nodes
      in Common Set */

for each si ∈ CommonSet do
      E′ = E′ U {e′i,Index1Index2};
end for
end while
```

Clique Partitioning - Applications

Application to allocation problems

1) Storage/register allocation

- Goal is to minimize register cost by maximizing the sharing of common registers among variables

- Derive the graph model
  - Every vertex represents a variable
  - There exists an edge between two vertices only if the variables can be stored in the same resister or if the lifetimes of the variables do not overlap

- After the clique partitioning, all the vertices/variables in a clique can be stored in a single register

Clique Partitioning - Applications

2) Functional unit allocation

Derive the graph model

- Every vertex represents an operation
- There exists an edge between two vertices only if
  - Two operations are scheduled in different control steps
  - There exists a multi functional unit which can carry out both the operations

- After the clique partitioning, all the vertices/operations in a clique can share the same functional unit

Clique Partitioning - Applications

3) Interconnection unit allocation

Derive the graph model

- Each vertex represents a connection between two units
- An edge exists between two vertices, if two corresponding connections are not used concurrently in any control step

- After partitioning, all the vertices in a clique are assigned same bus/multiplexer

Clique Partitioning

How to take care of interdependence ?

- Augment the graph edges with weights that reflect the interconnection complexity
- Higher weight if interconnection cost reduces
- Cliques with heavier edges are preferred

Floorplanning (Ch. 8 - Gerez)

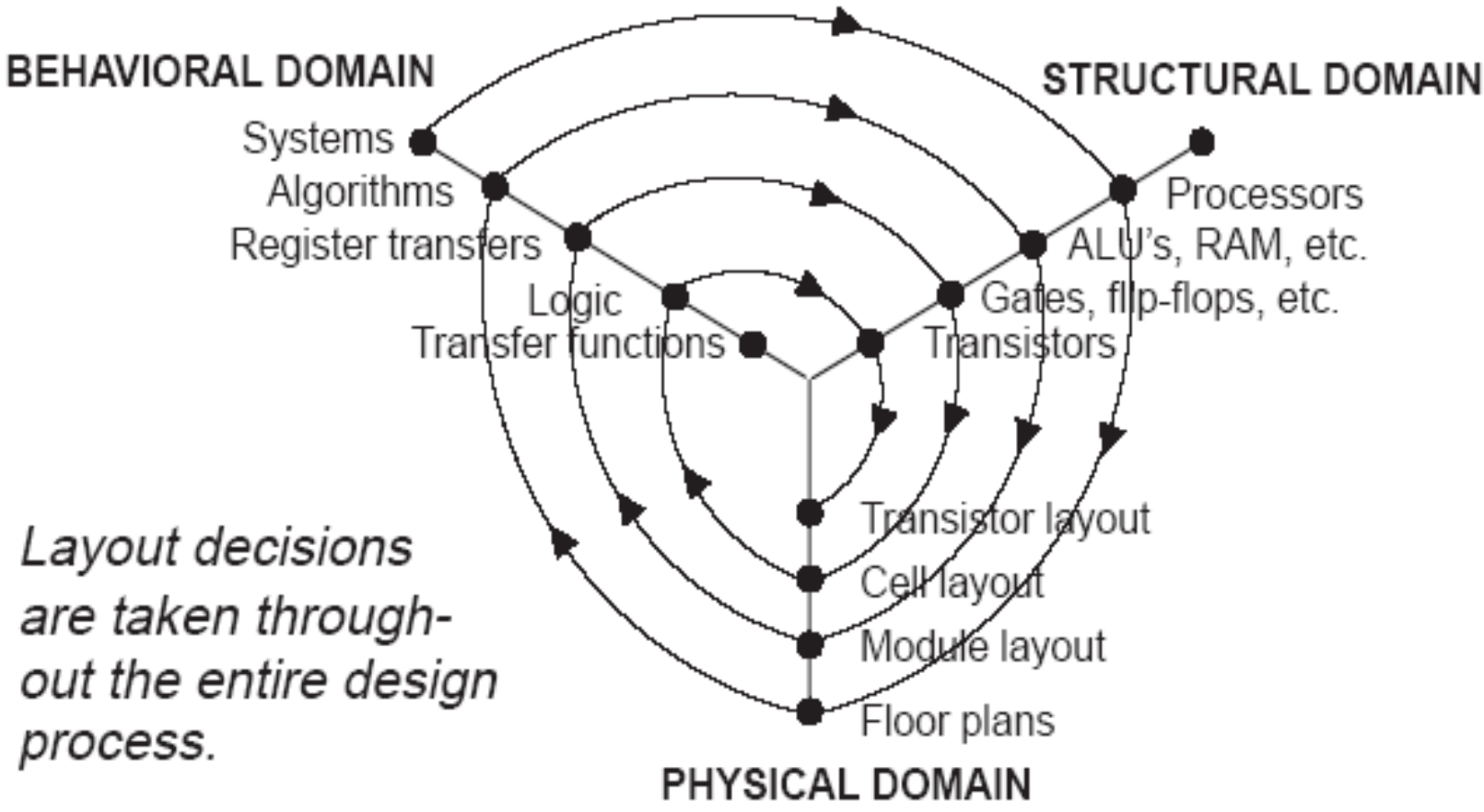
Contents

- Floorplan based design methodology
- Floorplanning concepts
  - Terminology & Representation
  - Optimization problem
- Shape function & Floorplan sizing

What is a Floorplan?

- Fixing of relative positions of sub-blocks
- Allocation of the specific region in the IC
- Estimation of the area of sub-blocks
- Estimation of interconnection pattern

Floorplan Based Design Methodology



Floorplan Based Design Methodology

- A synthesis step from behavioral to structural domain is followed by a synthesis step from structural to physical domain
- Gives early feedback on the layout consequences
- Allows for estimation of wire-lengths, and hence timing and power
- Position, shapes and terminal positions of various sub-blocks are fixed

Floorplanning Inputs

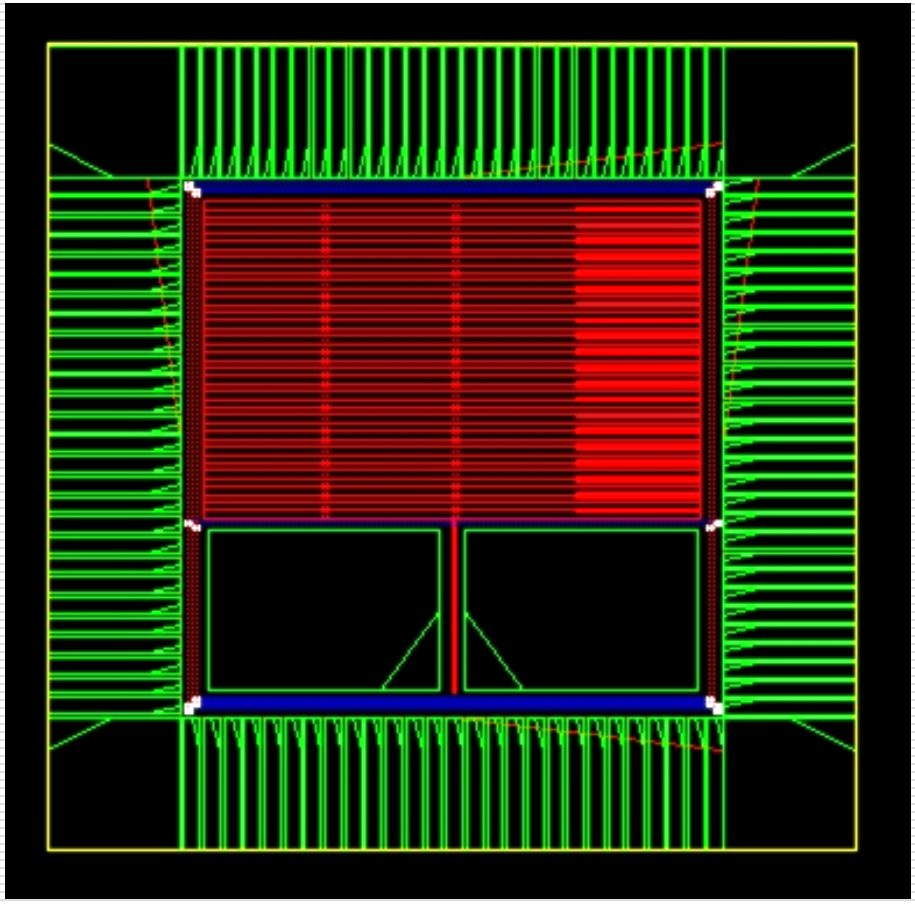
- Design netlist
- Area requirements
- Power requirements
- Timing constraints
- Physical partitioning information
- Die size vs. performance vs. schedule trade-off
- I/O placement (optional)
- Macro placement information (optional)

Floorplanning Outputs

- Die/block area
- I/Os placed
- Macros placed
- Power grid designed
- Power pre-routing
- Standard cell placement areas

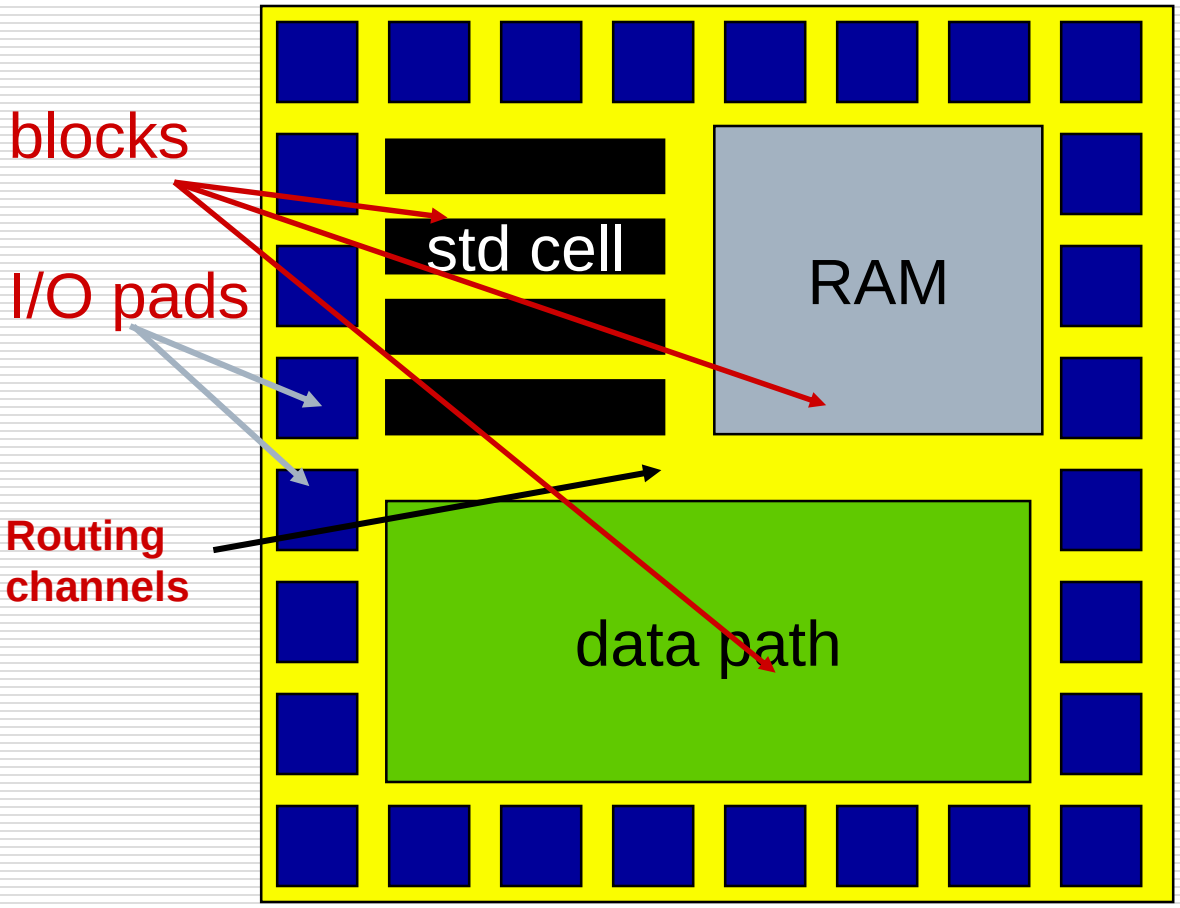
→ **Design ready for standard cell placement**

Floorplanning Output

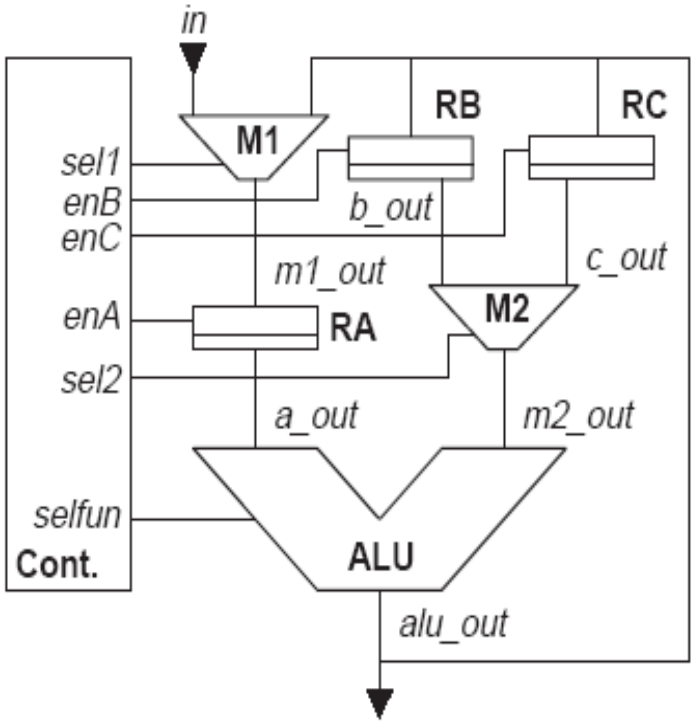


Sample Floorplan

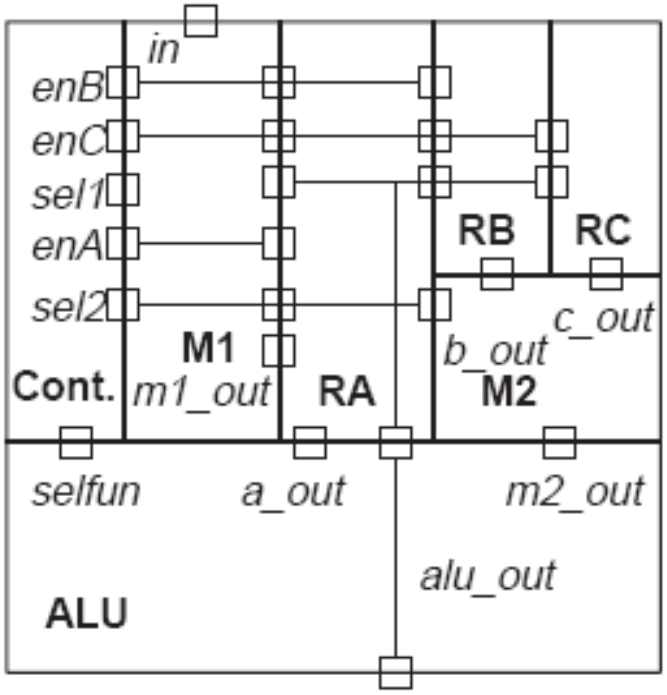
- Blocks inside a pad frame
- Routing inside, between blocks
- Different-sized blocks more difficult than standard cells to place and route
- Blocks
  - Hard, soft, semi-soft
  - Rectangular, L-shaped, T-shaped, rectilinear
  - Can rotate, mirror, …



Structural Description Circuit



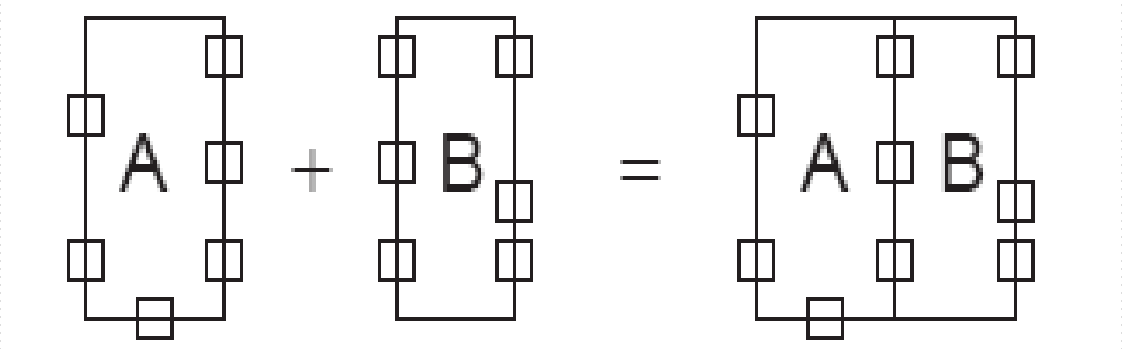
Possible Floorplan



Floorplan Concepts

Terminology

Abutment: establishing connections between cells by putting them directly next to each other, without the necessity of routing.



Terminology

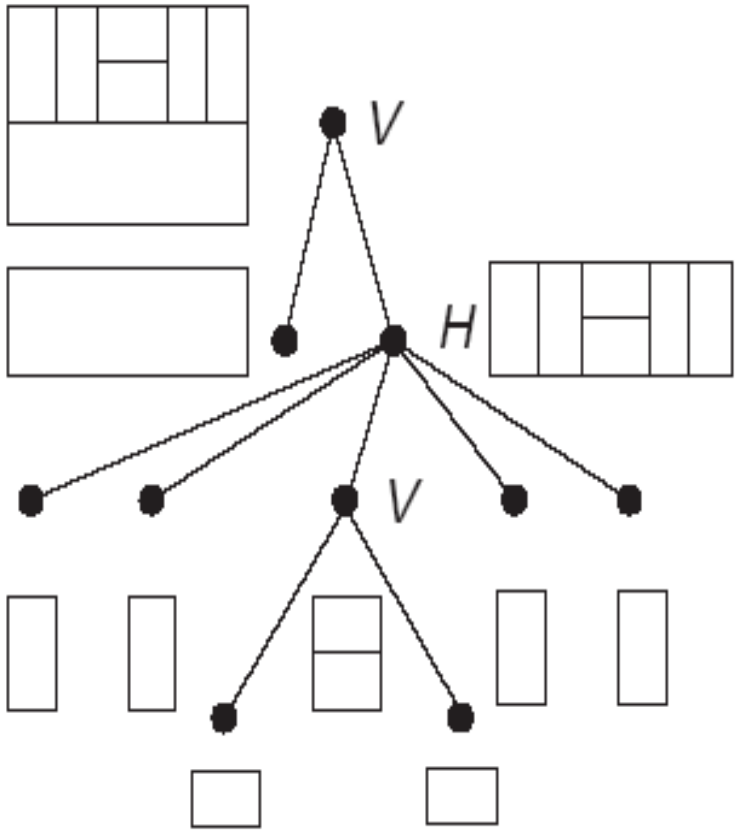
Leaf cell: a cell at the lowest level of the hierarchy; it does not contain any other cell.

Composite cell: a cell that is composed of either leaf cells or composite cells. The entire IC is a composite cell at the highest level.

For the sake of simplicity let us assume all leaf cells and composite cells are rectangular in shape.

Terminology

- If the children of all composite cells can be obtained by bisecting the cell horizontally or vertically, the floor plan is called a slicing floorplan
- Slicing floorplans can be represented by a **slicing tree**.
- In a slicing tree, all cells (except for the top-level cell)have a parent, and all composite cells have children.
- A slicing floorplan is also called a **floorplan of order 2**.

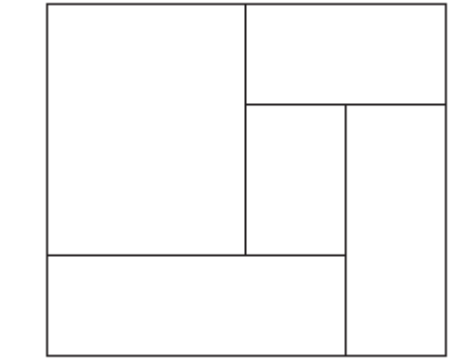


Terminology

Wheel or spiral floorplans

- Not all the floorplans are slicing type. In a wheel floor plan, the children of a cell can not be obtained by bisections.

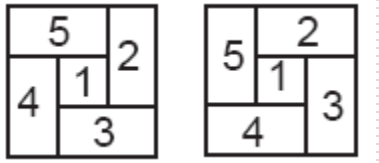
- Taking the shape of a wheel floorplan and its mirror image as the basis of operators leads to hierarchical descriptions of order 5.

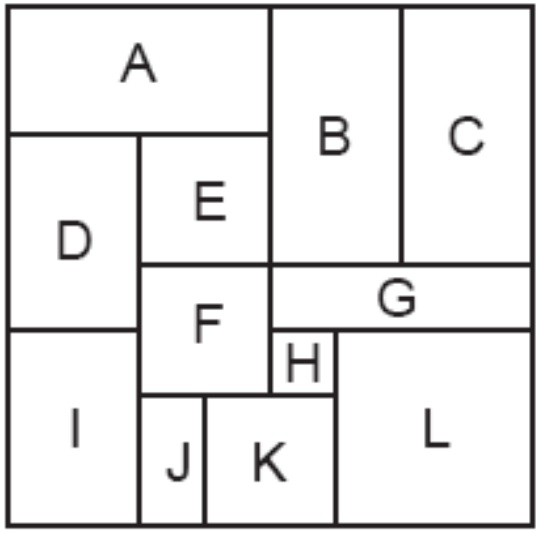


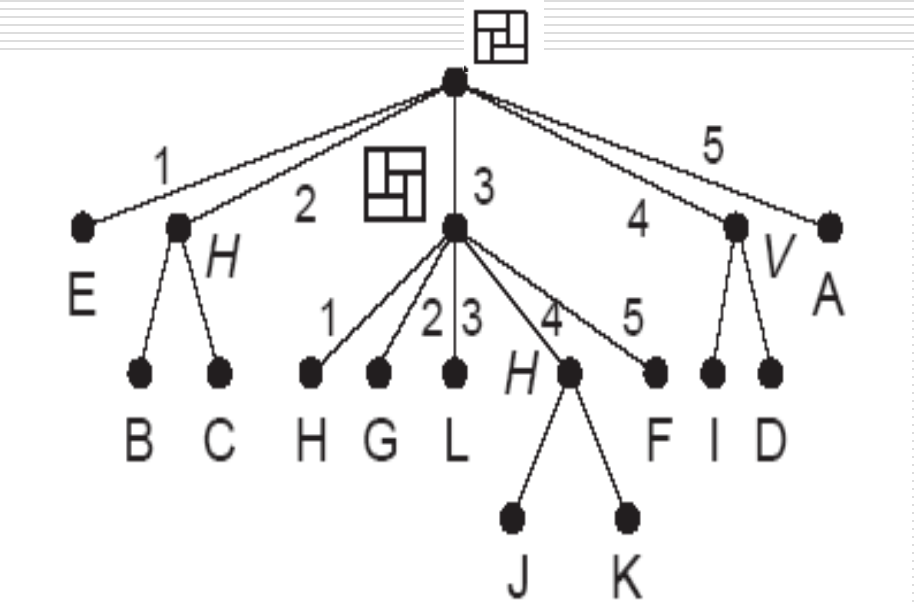
This floorplan is not slicing type!

Terminology

Order - 5 Floorplan Example







Optimization Problem in Floorplan

Floorplan generation: this is the problem of e.g. constructing the slicing tree, given a structural description. It can be done by adapting methods known from placement such as min-cut partitioning.

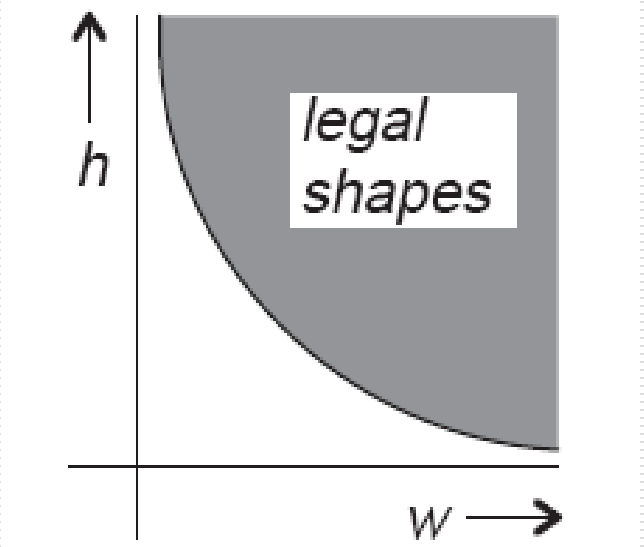
Generation of flexible cells:

Floorplan sizing: optimization of the floorplan area taking advantage of the flexibility in the cells.

Shape function & Floorplan Sizing

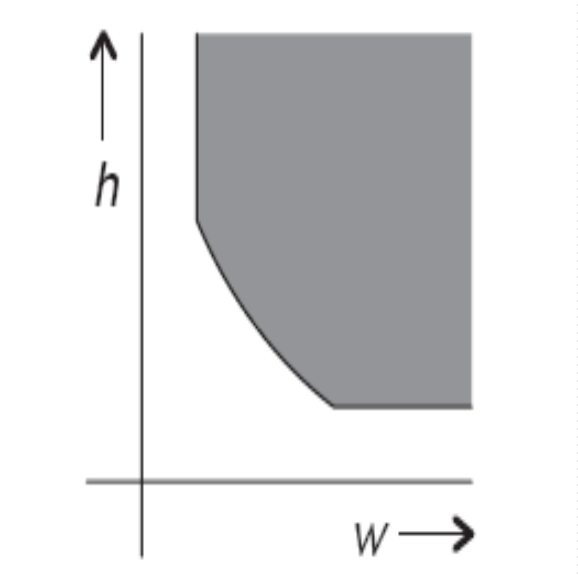
SHAPE FUNCTIONS (1)

- Flexible cells imply that cells can assume different aspect ratios.
- One can assume that all implementations of a cell have same minimum area A .
- $hw = A$, where h is the height and w is the width of the cell
- The minimal height given as a function of the width is called shape function of cell. h(w)=A/w



SHAPE FUNCTIONS (2)

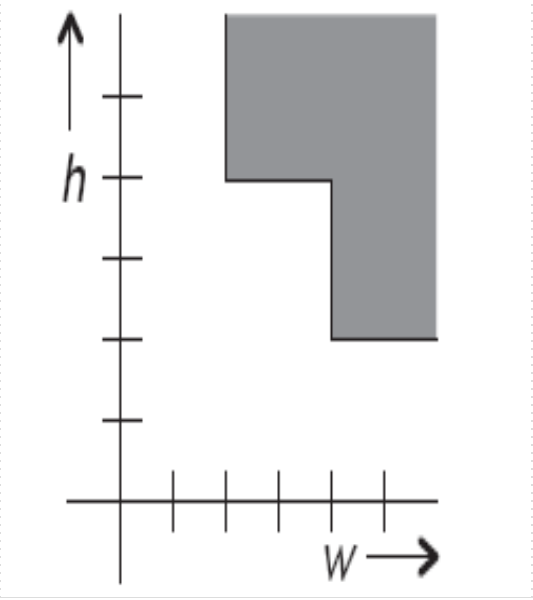
- Very thin cells are not desirable and often not feasible to design.
- The shape function is a combination of a hyperbola and two straight lines.



SHAPE FUNCTIONS (3)

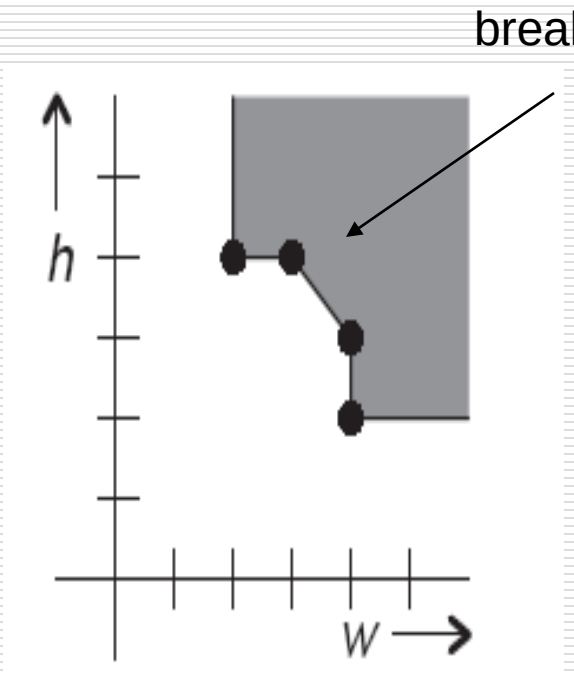
- Leaf cells are built from discrete transistors: it is not realistic to assume that the shape function follows the hyperbola continuously.

- In an extreme case, a cell is rigid: it can only be rotated and mirrored during floorplanning or placement. Such a cell is called an **inset cell.**



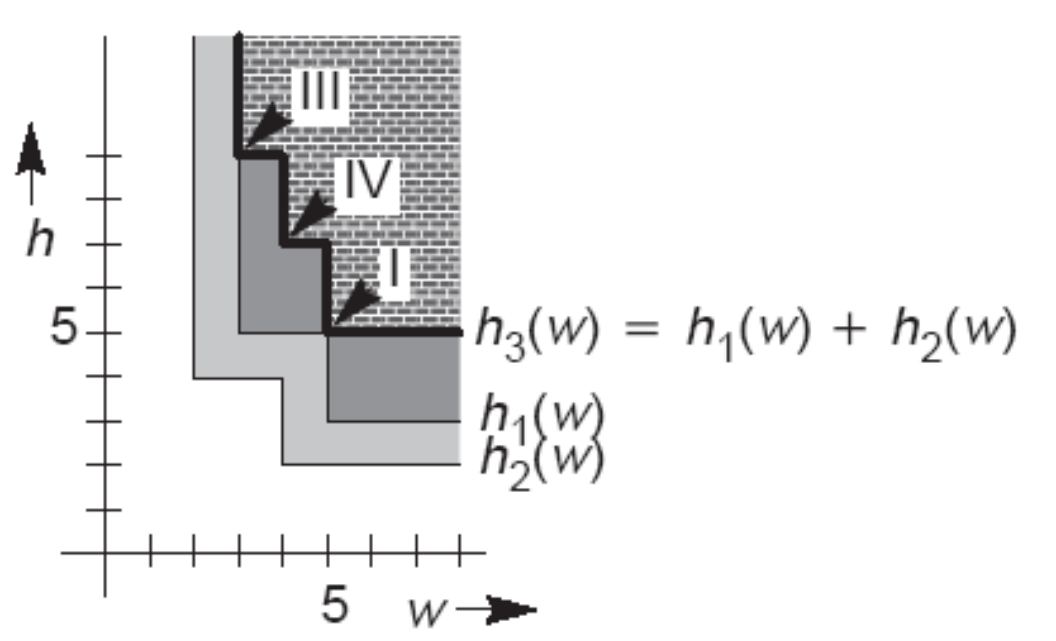
SHAPE FUNCTIONS (4)

- In general, a piecewise linear function can be used to approximate any shape function.
- The points where the function changes its direction, are called the break points of the piecewise linear function.



ADDITION FOR VERTICAL ABUTMENT

Composition by vertical abutment → the addition of shape functions.



SIZING ALGORITHM

- The shape functions of all leaf cells are given as piecewise linear functions.
- Traverse the slicing tree in order to compute the shape functions of all composite cells (bottom-up composition).
- Choose the desired shape of the top-level cell
- As the shape function is piecewise linear, only the break points of the function need to be evaluated, when looking for the minimal area.
- Propagate the consequences of the choice down to the leaf cells (top-down propagation).

DERIVING SHAPES OF CHILDREN

A choice for the minimal shape of composite cell fixes the shapes of its children cells.