

VLSI System Design

Part V : High-Level Synthesis(3)

Lecturer : Tsuyoshi Isshiki

Dept. Information and Communications Engineering,
Tokyo Institute of Technology
issiki@ict.e.titech.ac.jp

High-Level Synthesis Flow

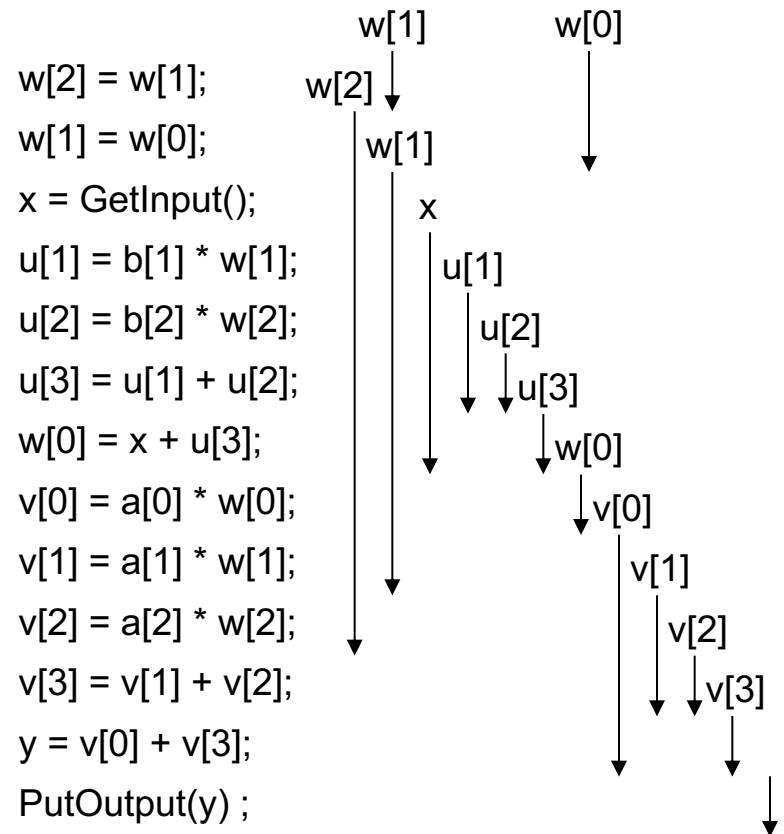
- A) Design capture (HDLs, C/C++, signal-flow graph, etc)
- B) Compilation to internal representation
 - Data-flow graph (DFG)
 - Control-flow graph (CFG)
 - Control-data-flow graph (CDFG)
- C) Resource allocation
 - Specify available functional units
- D) Operation scheduling
 - Assign each operation to control steps
- E) Resource binding
 - Assign each data to registers
 - Assign each operation to functional units

Resource Binding

- Datapath architecture construction
 - ✓ Functional units
 - ✓ Registers
 - ✓ Interconnect (busses, multiplexers)
 - ✓ Memory
- Resource binding is the process of allocating each resource instances to computational elements (operations, data)

Data Lifetime (1)

- Recall *data lifetime* within basic blocks:
 - Starts after the data assignment operation
 - Ends after the last operation using the data
- Used for constructing data-flow graph for operation scheduling.
- Actually not sufficient for register binding because data can be alive across basic block boundaries.

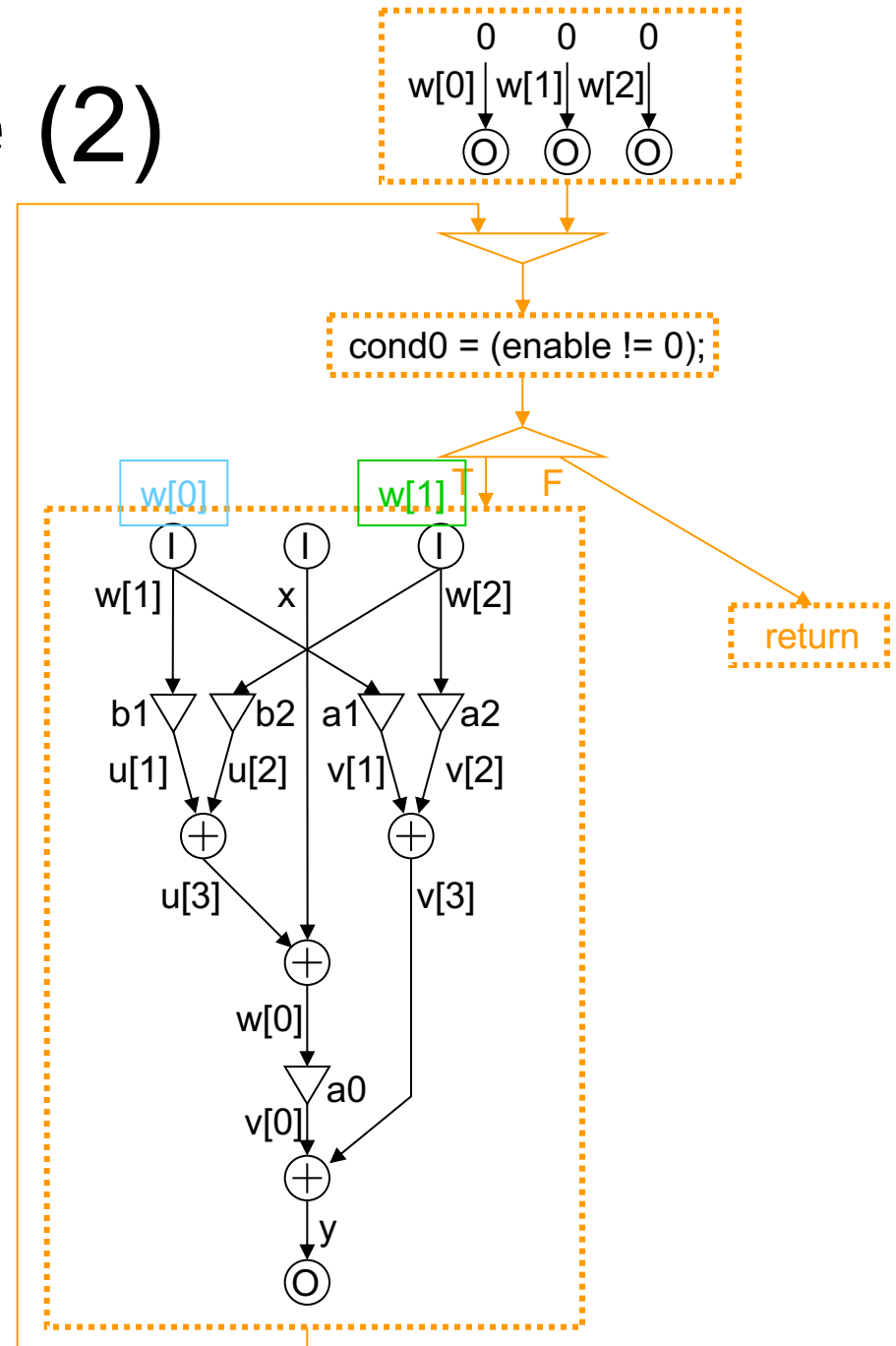
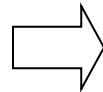


Data Lifetime (2)

- *Where does the input variables in the basic block come from??*
- `x = GetInput()` is the actual input node to the system.
- What about `(w[2] = w[1])` and `(w[1] = w[0])` ??

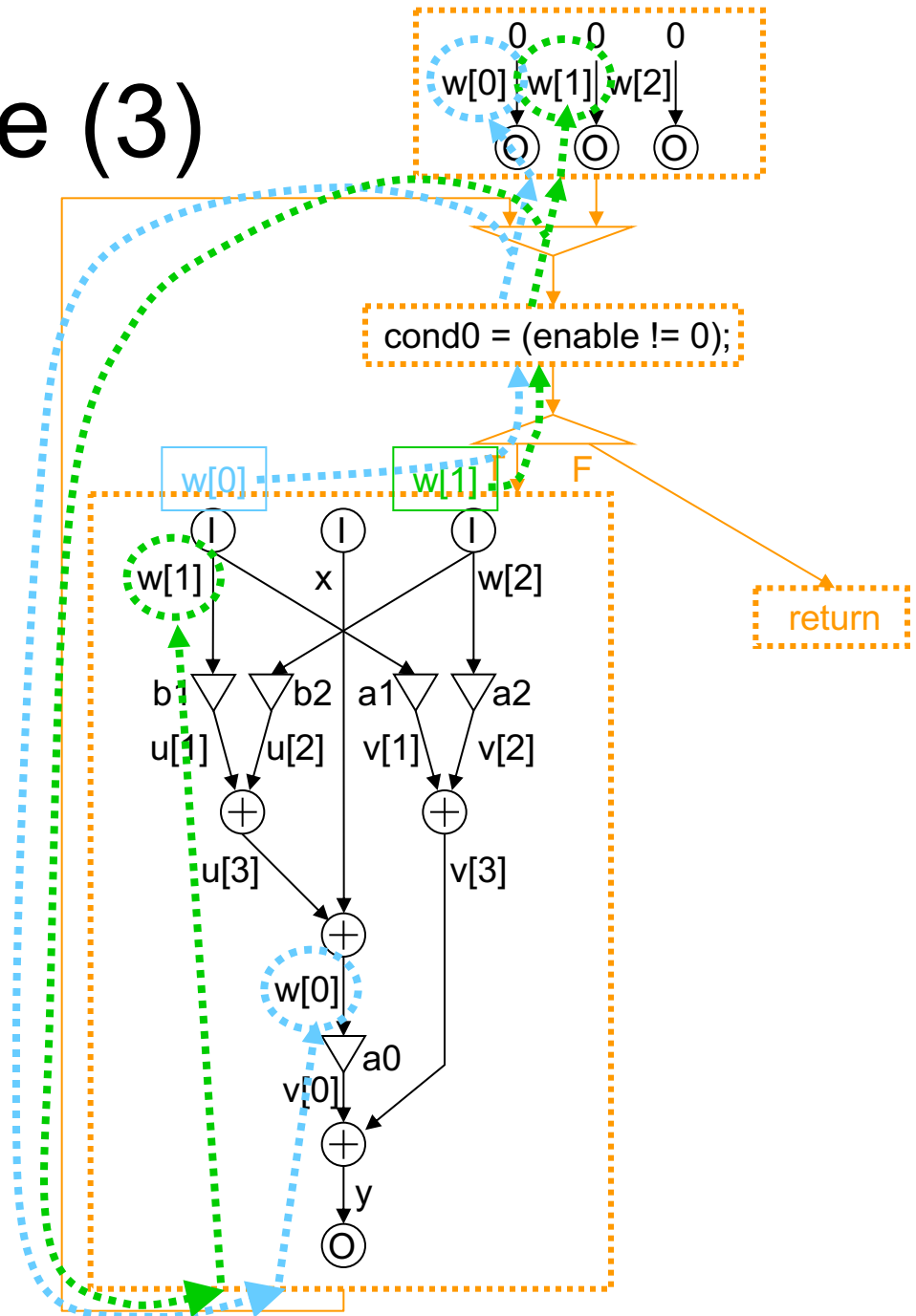
```

w[0] = 0; w[1] = 0; w[2] = 0;
while (enable != 0){
    w[2] = w[1];
    w[1] = w[0];
    x = GetInput();
    u[1] = b[1] * w[1];
    u[2] = b[2] * w[2];
    u[3] = u[1] + u[2];
    w[0] = x + u[3];
    v[0] = a[0] * w[0];
    v[1] = a[1] * w[1];
    v[2] = a[2] * w[2];
    v[3] = v[1] + v[2];
    y = v[0] + v[3];
    PutOutput(y);
}
    
```



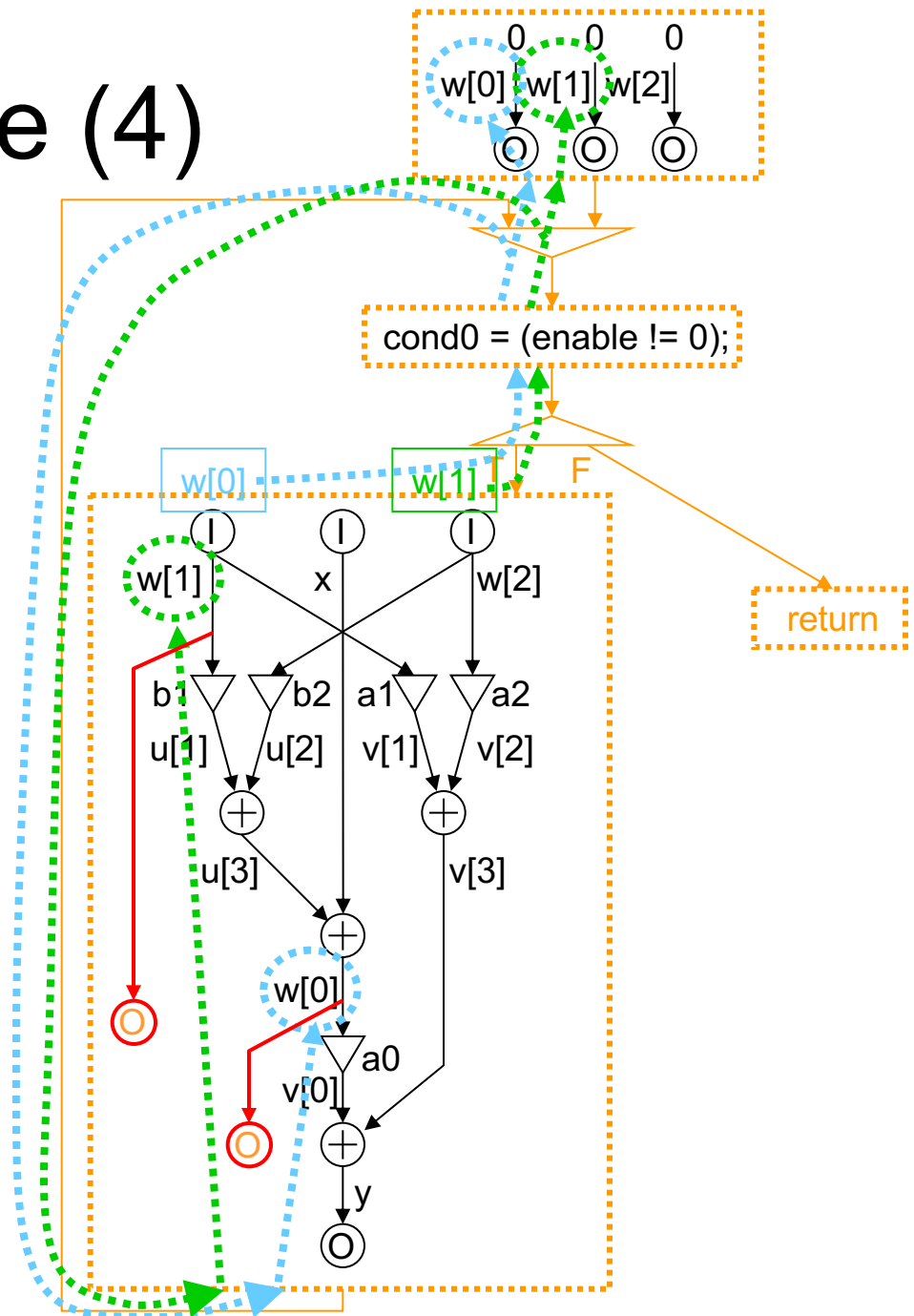
Data Lifetime (3)

- For each implicit inputs in the basic-block (variables without source within basic block), traverse the control-flow graph backwards until the basic-block which generates the data is reached.
- When join-node is reached, traversal must *fork* to each of the join sources.



Data Lifetime (4)

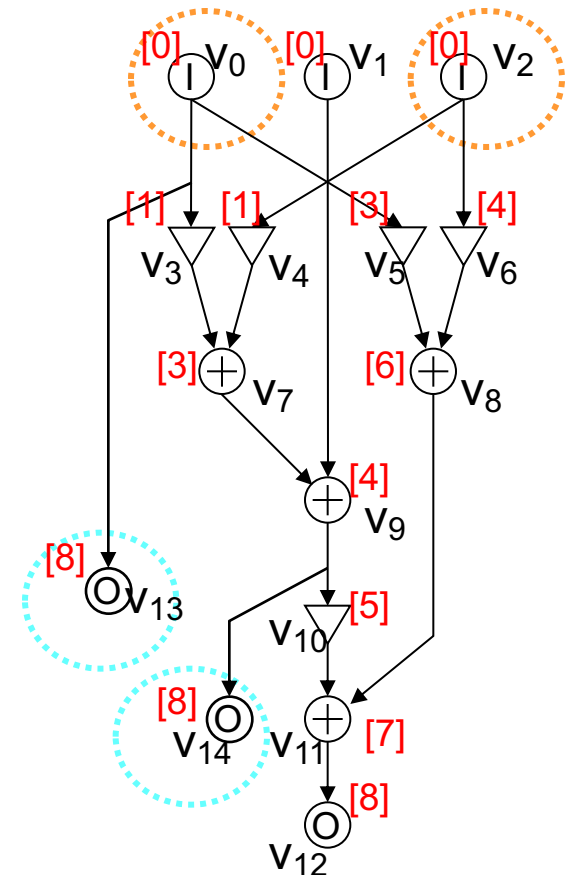
- For the basic-block which generates the concerned input data, add output node for that data (if the output node does not exist)



Data Lifetime After Scheduling (1)

□ Scheduling of input/output nodes :

- All input nodes whose data lifetimes cross the basic-block boundary need to be scheduled at time $t = 0$.
- All output nodes whose data lifetimes cross the basic-block boundary need to be scheduled at time $t = T_{max} - 1$.
- IO nodes whose data lifetimes *do not* cross the basic-block boundaries is not restricted here (but actually is dependent on the external devices connected to these nodes)

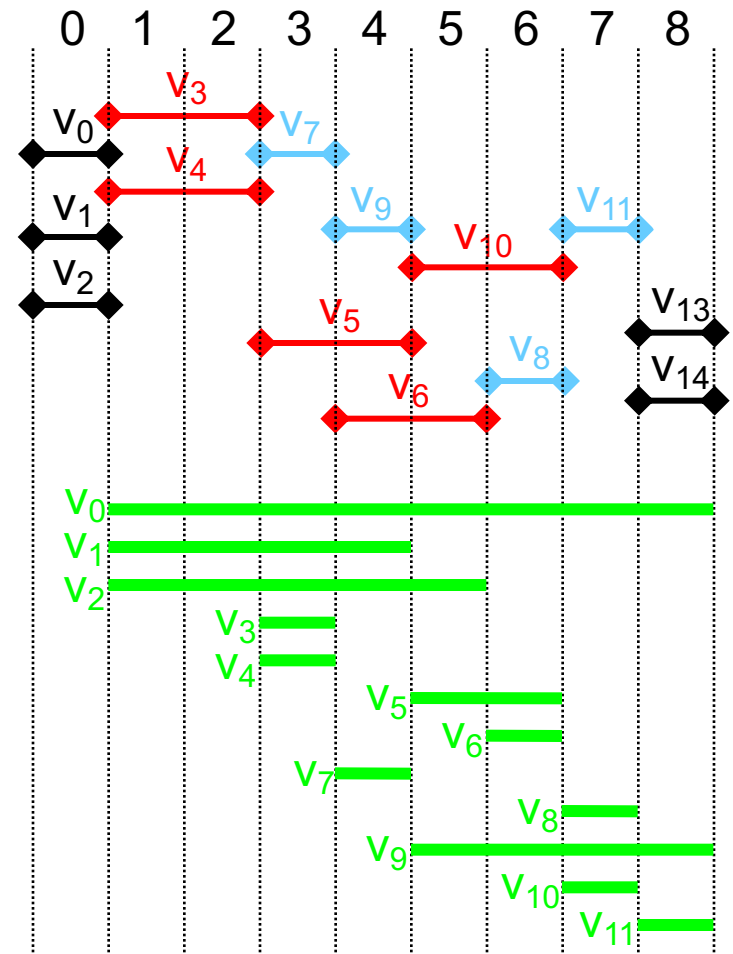
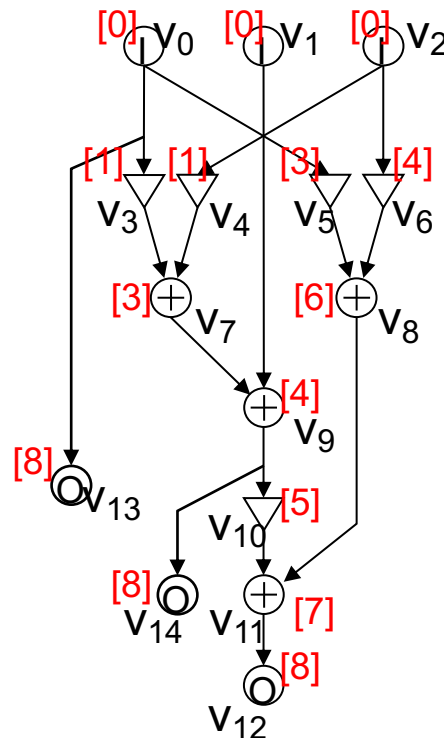


Interval Graph for Data Lifetime

- $l(v_j) = [l_{min}(v_j), l_{max}(v_j)]$: lifetime of output data of v_j
- $l_{min}(v_j) = \sigma(v_j) + \delta(v_j)$
- $l_{max}(v_j) = \max \{ \sigma(v_i) + \delta(v_i) - 1 \mid (v_j, v_i) \in E \}$

✓ Example :

- ◆ $l_{min}(v_1) = 1$
- ◆ $l_{max}(v_1) = 4$



Register Binding Problem

□ Problem input :

- ✓ List of data lifetimes $L = \{l(v) \mid v \in V\}$
- ✓ Set of registers $R = \{r_i \mid i = 0, 1, \dots, |R| - 1\}$

□ Register binding λ is a mapping of operations $v \in V$ to the register set R

$$\lambda : V \rightarrow R$$

□ such that all the output data lifetimes of the operations mapped to a register does not overlap

$$\begin{aligned} \square \quad & l_{\min}(v_j) > l_{\max}(v_i) \text{ or } l_{\max}(v_j) < l_{\min}(v_i) \\ \square \quad & \text{for all } \lambda(v_j) = \lambda(v_i) \end{aligned}$$

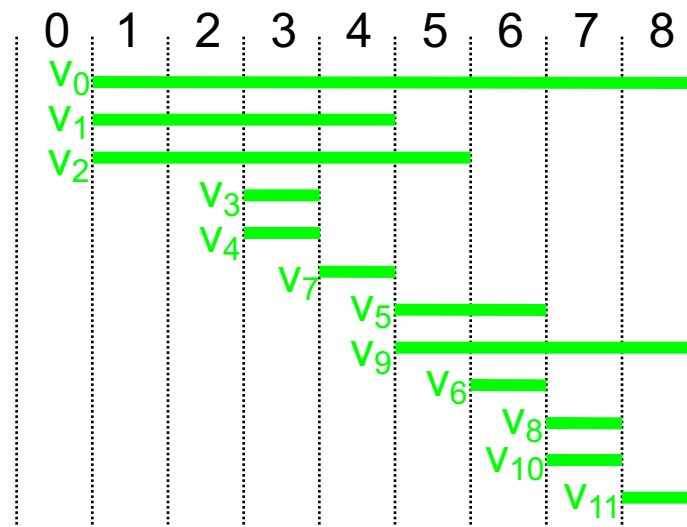
□ Objective :

- ✓ Minimize the number of registers required $|R|$ to hold all output variables

Left-Edge Algorithm (1)

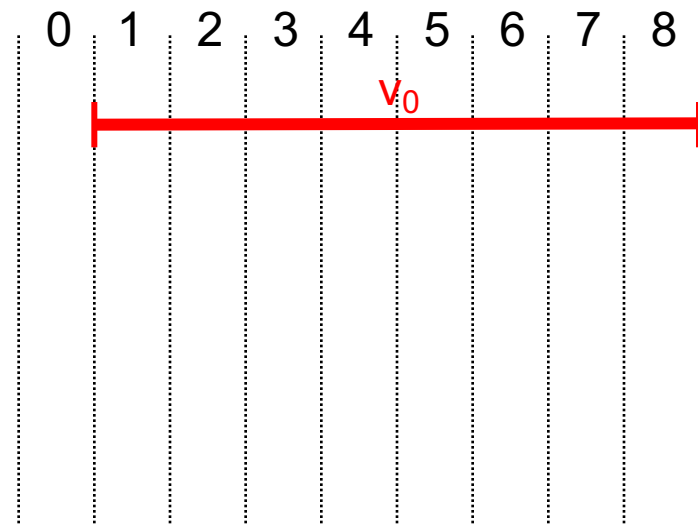
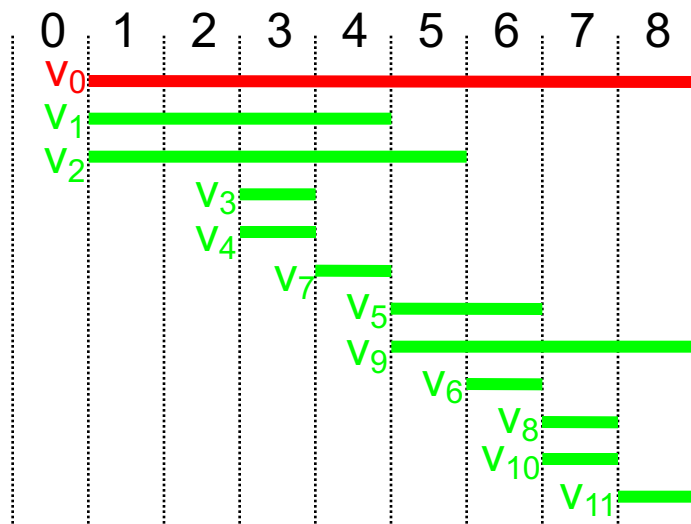
1. L : Operation list V sorted in the increasing order of $l_{min}(v)$
2. $k = -1$
3. $L' = \phi$ (temporal list of operations)
4. Select the first element v in V which satisfies $l_{min}(v) > k$
 - If such v does not exist, go to 8
5. Remove v from L and add to L'
6. $k = l_{max}(v)$
7. Go to 4
8. Add register r and assign all operations in L'
9. If L is not empty, go to 2. Otherwise END

Sorted operation list : L



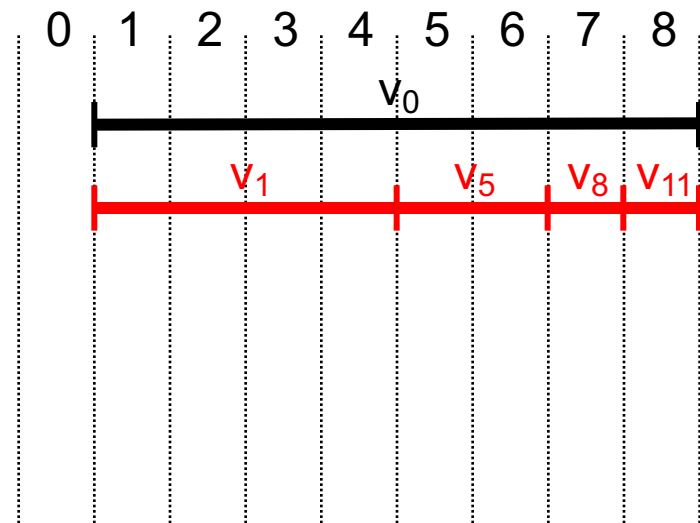
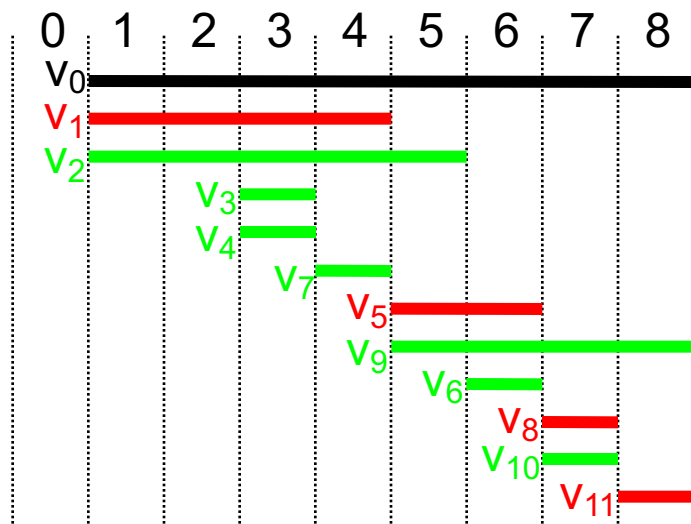
Left-Edge Algorithm (2)

1. L : Operation list V sorted in the increasing order of $l_{min}(v)$
2. $k = -1$
3. $L' = \phi$ (temporal list of operations)
4. Select the first element v in V which satisfies $l_{min}(v) > k$
 - If such v does not exist, go to 8
5. Remove v from L and add to L'
6. $k = l_{max}(v)$
7. Go to 4
8. Add register r and assign all operations in L'
9. If L is not empty, go to 2. Otherwise END



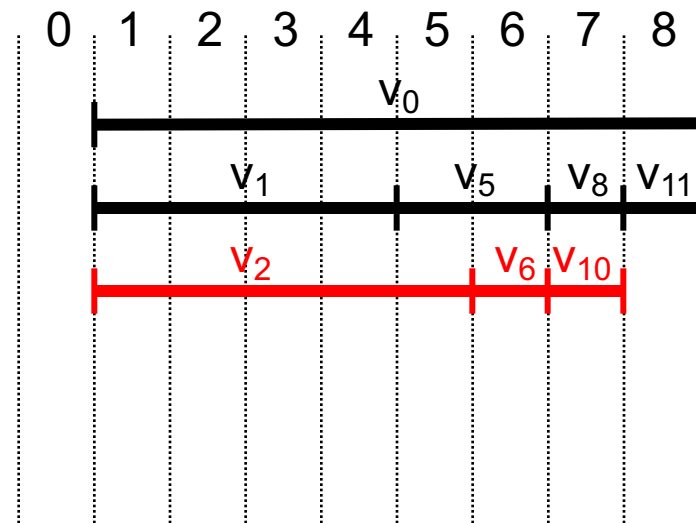
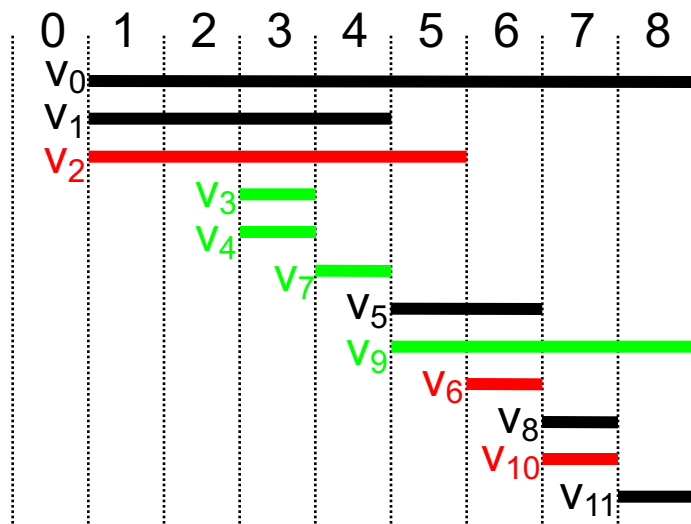
Left-Edge Algorithm (3)

1. L : Operation list V sorted in the increasing order of $l_{min}(v)$
2. $k = -1$
3. $L' = \phi$ (temporal list of operations)
4. Select the first element v in V which satisfies $l_{min}(v) > k$
 - If such v does not exist, go to 8
5. Remove v from L and add to L'
6. $k = l_{max}(v)$
7. Go to 4
8. Add register r and assign all operations in L'
9. If L is not empty, go to 2. Otherwise END



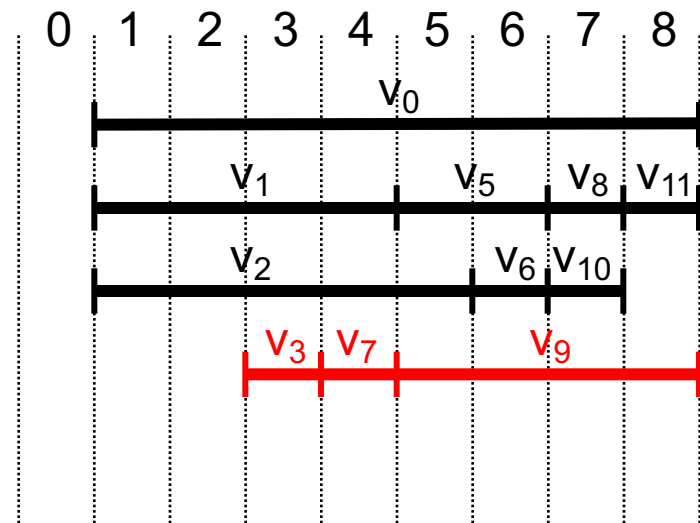
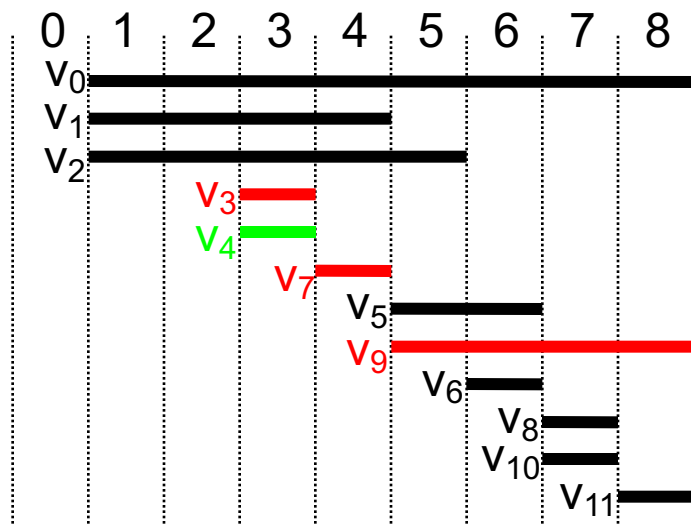
Left-Edge Algorithm (4)

1. L : Operation list V sorted in the increasing order of $l_{min}(v)$
2. $k = -1$
3. $L' = \phi$ (temporal list of operations)
4. Select the first element v in V which satisfies $l_{min}(v) > k$
 - If such v does not exist, go to 8
5. Remove v from L and add to L'
6. $k = l_{max}(v)$
7. Go to 4
8. Add register r and assign all operations in L'
9. If L is not empty, go to 2. Otherwise END



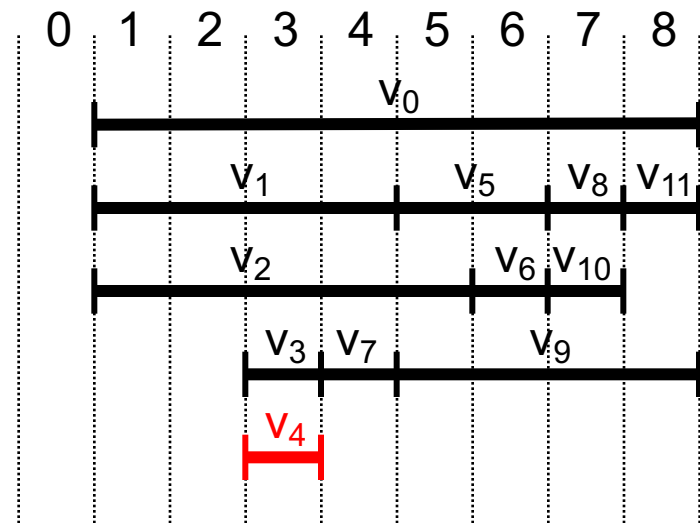
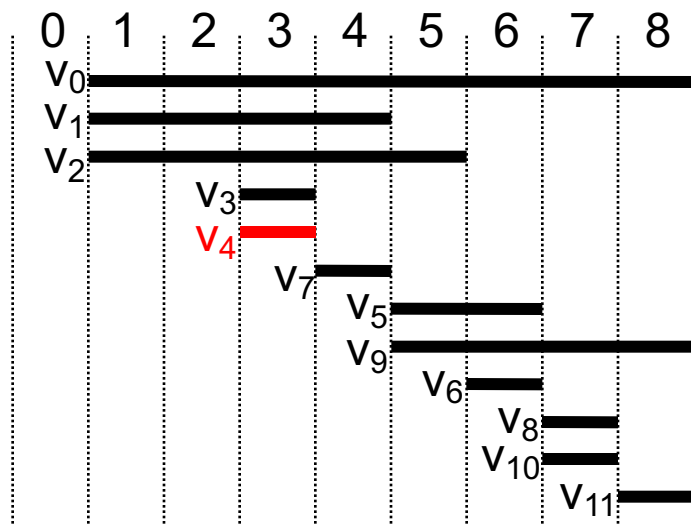
Left-Edge Algorithm (5)

1. L : Operation list V sorted in the increasing order of $l_{min}(v)$
2. $k = -1$
3. $L' = \phi$ (temporal list of operations)
4. Select the first element v in V which satisfies $l_{min}(v) > k$
 - If such v does not exist, go to 8
5. Remove v from L and add to L'
6. $k = l_{max}(v)$
7. Go to 4
8. Add register r and assign all operations in L'
9. If L is not empty, go to 2. Otherwise END



Left-Edge Algorithm (5)

1. L : Operation list V sorted in the increasing order of $l_{min}(v)$
2. $k = -1$
3. $L' = \phi$ (temporal list of operations)
4. Select the first element v in V which satisfies $l_{min}(v) > k$
 - If such v does not exist, go to 8
5. Remove v from L and add to L'
6. $k = l_{max}(v)$
7. Go to 4
8. Add register r and assign all operations in L'
9. If L is not empty, go to 2. Otherwise END



Properties of Left-Edge Algorithm

- ❑ Even though it is a greedy algorithm, left-edge algorithm produces an optimal solution in terms of number of registers
 - Minimum number of registers required is the maximum number of data lifetimes overlapping within the scheduling time set T
 - It can be shown that left-edge algorithm always produces a binding solution with the maximum number overlapping data lifetimes
- ❑ Limitations of left-edge algorithm
 - Can only handle interval graph
 - Cannot take into account factors other than the number of registers into the problem formulation. (number of registers is not the only hardware cost)

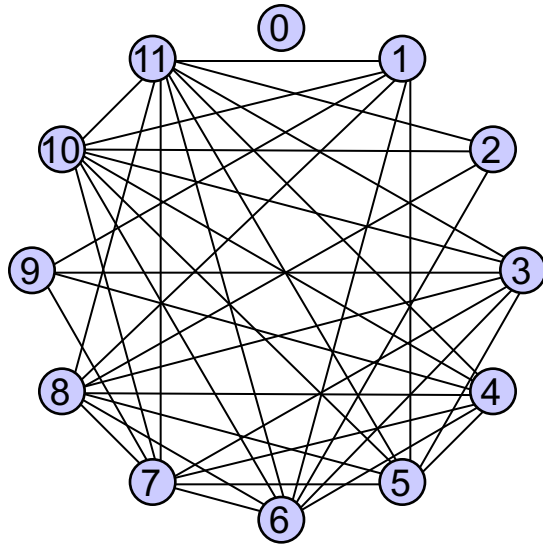
Task-To-Agent Problem (1)

- ❑ Generalization of resource binding problem
 - Task : operation, data, data transfer
 - Agent : functional unit, register, bus
- ❑ Task compatibility and task conflict
 - Two tasks are *compatible* if they can be assigned to the same agent.
 - Otherwise, they are in *conflict*.
- ❑ Task-to-agent problem is to assign tasks to agents such the number of agents are minimized where all tasks assigned to an agent are compatible.

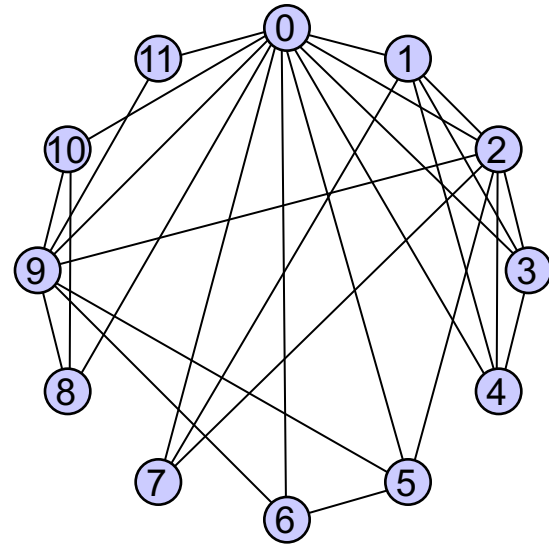
Task-To-Agent Problem (2)

- ❑ Compatibility graph $G_p(V_p, E_p)$
 - Vertices denote tasks
 - Presence of an edge $(v_i, v_j) \in E_p$ indicates that vertices v_i and v_j are compatible
- ❑ Conflict graph $G_f(V_f, E_f)$
 - Vertices denote tasks
 - Presence of an edge $(v_i, v_j) \in E_f$ indicates that v_i and v_j are in conflict
- ❑ Conflict graph $G_f(V_f, E_f)$ is a complement graph of compatibility graph $G_p(V_p, E_p)$
 - $V_p = V_f$ (same vertex set)
 - $G(V_p, E_p \cup E_f)$: **complete graph** (there are edges to every pair of vertices)

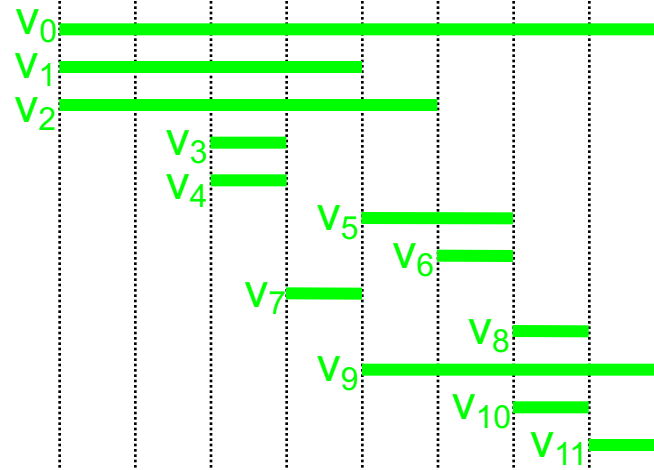
Compatibility Graph vs Conflict Graph



Compatibility Graph



Conflict Graph



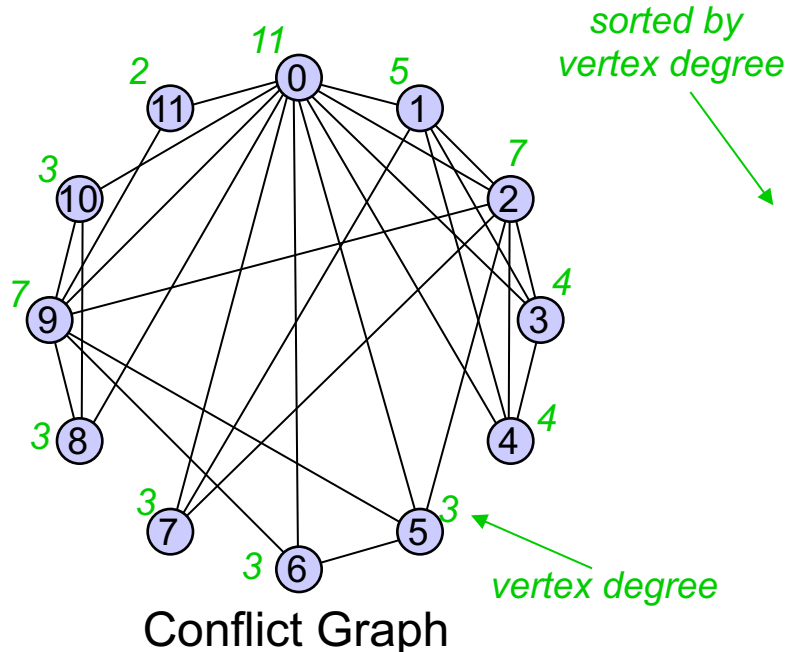
Graph Coloring Problem

- ❑ Problem input : **conflict graph** $G_f(V_f, E_f)$
- ❑ Assign a color to each vertex so that adjacent vertex pair each has a different color, and the total number of color is minimized (color \rightarrow agent)
- ❑ General graph coloring problem is NP-complete
- ❑ Special instances of graphs can be optimally colored
 - \rightarrow Conflict graph constructed from **interval graph**
 - \rightarrow Left-edge algorithm gives the optimal graph coloring solution in polynomial time
 - \rightarrow In general, task-to-agent problem may not be from an interval graph, in which case left-edge algorithm cannot be applied

Greedy Graph Coloring Algorithm

□ Basic greedy algorithm :

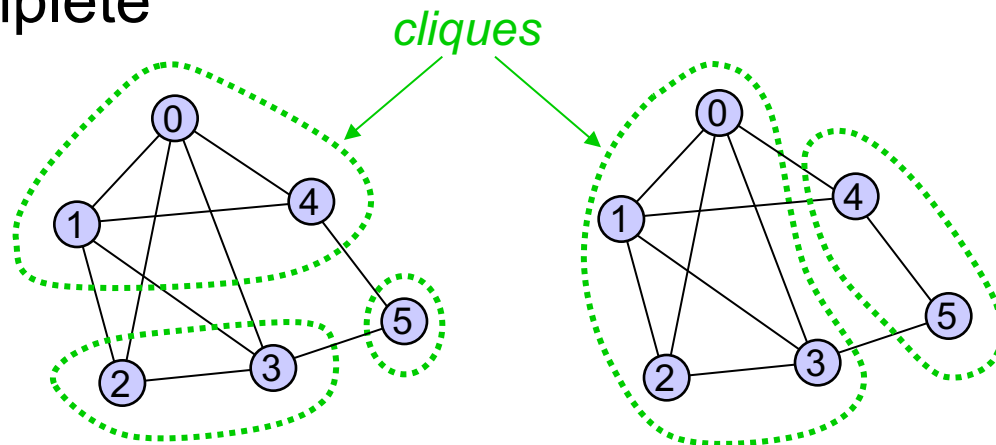
- Sort V in some order (such as *vertex degree* : number of edges connected to that vertex)
- For each vertex $v \in V$ (in the sorted order), assign the color with the minimum index which is not assigned to any of the vertices adjacent to v .



	color 0	color 1	color 2	color 3	color 4
0 (11)	0				
2 (7)	0	2			
9 (7)	0	2	9		
1 (5)	0	2	9,1		
3 (4)	0	2	9,1	3	
4 (4)	0	2	9,1	3	4
5 (3)	0	2	9,1	3,5	4
6 (3)	0	2	9,1	3,5	4,6
7 (3)	0	2	9,1	3,5,7	4,6
8 (3)	0	2,8	9,1	3,5,7	4,6
10 (3)	0	2,8	9,1	3,5,7,10	4,6
11 (2)	0	2,8,11	9,1	3,5,7,10	4,6

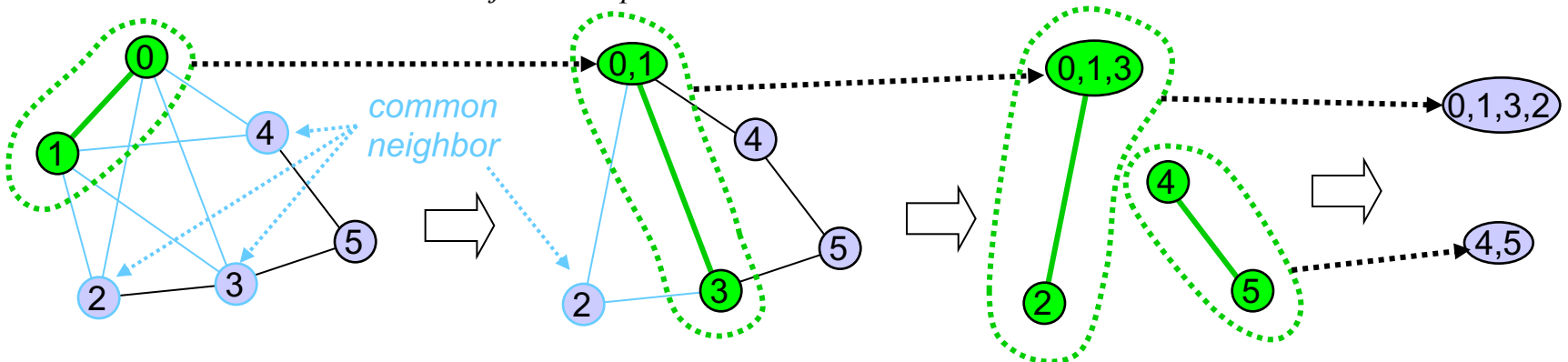
Clique Partitioning Algorithm (1)

- ❑ Problem input : **compatible graph** $G_p(V_p, E_p)$
- ❑ Partition $G_p(V_p, E_p)$ into cliques so that the total number of cliques is minimized (clique \rightarrow agent)
 - Clique : complete subgraph (there is an edge for all vertex pairs in the subgraph \rightarrow all vertices in a clique are compatible)
- ❑ General clique partitioning problem is NP-complete



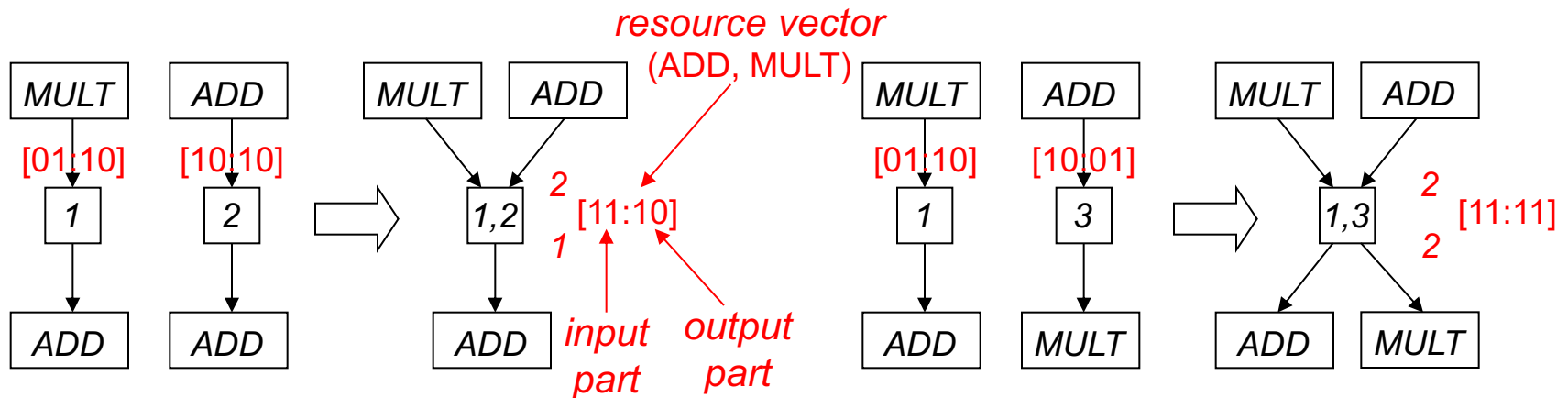
Clique Partitioning Algorithm (2)

- ❑ **Super-vertex** : group of vertex which form a clique (vertex is also a super-vertex)
- ❑ If two edges (v_i, v_k) and (v_j, v_k) exist, then vertex v_k is the **common neighbor** for v_i and v_j
- ❑ Algorithm :
 - A) Choose an edge $(v_i, v_j) \in E_p$ with the most common neighbors and merge v_i and v_j into a super-vertex $v_{i,j}$ (Remove (v_i, v_j) from E_p)
 - B) Add a set of edges $\{(v_{i,j}, v_k) \mid v_k : \text{common neighbor of } v_i \text{ and } v_j\}$ to E_p



Considering Interconnect Cost in Register Binding

- ❑ Interconnect cost (simple estimate) → sum of input and output arcs to the register
 - Assume that operations with the same type is assigned to the same hardware (because functional unit binding not done)
 - Interconnect cost = # (input op-types) + # (output op-types)
 - Resource vector : each bit denotes whether the corresponding functional unit is connected (1) or not connected (0)
 - Merge interconnect cost : bit-wise OR on the input/output *resource vectors* and counting 1s in the vectors.
 - Use interconnect cost when there are several merge candidates

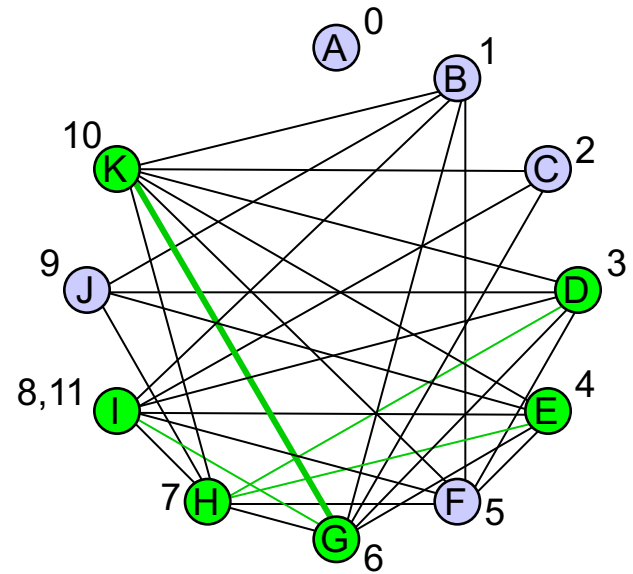
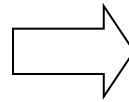
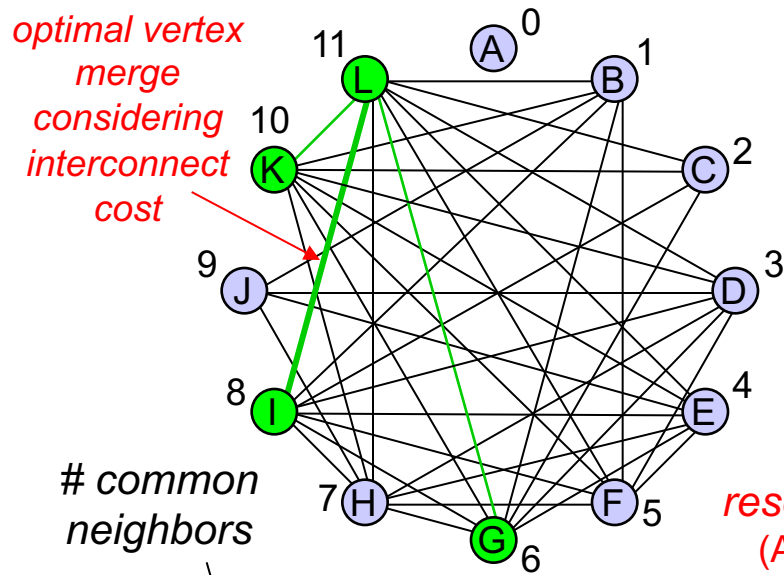


Clique Partitioning Algorithm (3)

optimal vertex
merge
considering
interconnect
cost

common
neighbors

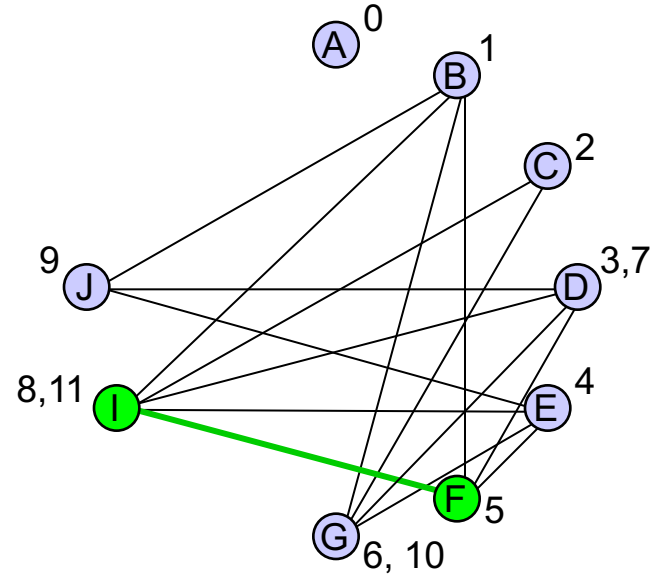
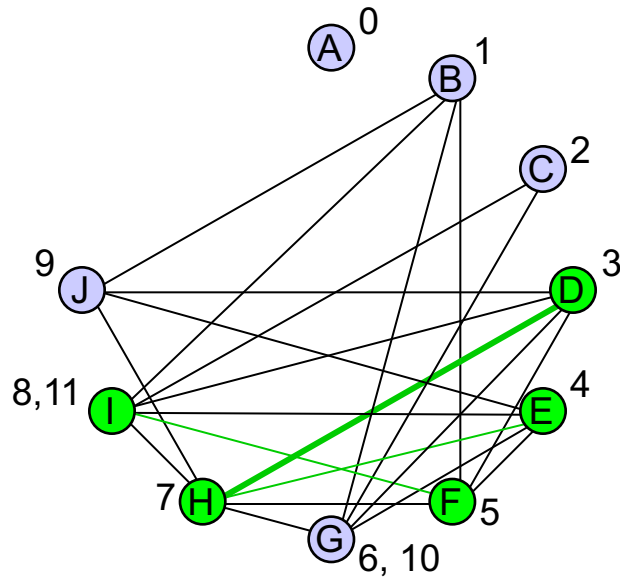
resource vector
(ADD, MULT)



A	B	C	D	E	F	G	H	I	J	K	L	
*	-	-	-	-	-	-	-	-	-	-	-	A: (0) [00:01]
-	*	-	-	-	3	3	-	3	0	3	4	B: (1) [00:10]
-	-	*	-	-	-	3	-	2	-	2	3	C: (2) [00:01]
-	-	-	*	-	4	4	6	4	1	4	5	D: (3) [01:10]
-	-	-	-	*	4	4	6	4	1	4	5	E: (4) [01:10]
-	3	-	4	4	*	-	5	5	-	5	6	F: (5) [01:10]
-	3	3	4	4	-	*	5	6	-	6	7	G: (6) [01:10]
-	-	-	6	6	5	5	*	5	2	5	6	H: (7) [10:10]
-	3	2	4	4	5	6	5	*	-	-	7	I: (8) [10:10]
-	0	-	1	1	-	-	2	-	*	-	-	J: (9) [10:01]
-	3	2	4	4	5	6	5	-	-	*	7	K: (10) [01:10]
-	4	3	5	5	6	7	6	7	-	7	*	L: (11) [10:00]

A	B	C	D	E	F	G	H	I	J	K	
*	-	-	-	-	-	-	-	-	-	-	A: (0) [00:01]
-	*	-	-	-	2	2	-	2	0	2	B: (1) [00:10]
-	-	*	-	-	-	2	-	1	-	1	C: (2) [00:01]
-	-	-	*	-	3	3	5	3	1	3	D: (3) [01:10]
-	-	-	-	*	3	3	5	3	1	3	E: (4) [01:10]
-	2	-	3	3	*	-	4	4	-	4	F: (5) [01:10]
-	2	2	3	3	-	*	4	5	-	5	G: (6) [01:10]
-	-	-	5	5	4	4	*	4	2	4	H: (7) [10:10]
-	2	1	3	3	4	5	4	*	-	-	I: (8,11) [10:10]
-	0	-	1	1	-	-	2	-	*	-	J: (9) [10:01]
-	2	1	3	3	4	5	4	-	-	*	K: (10) [01:10]

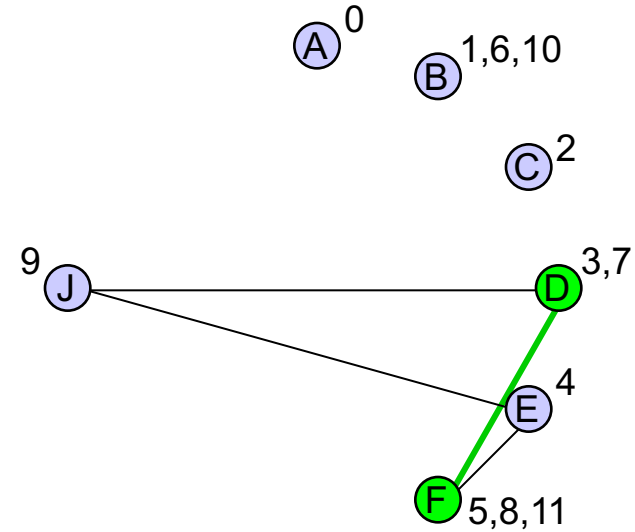
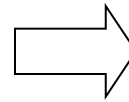
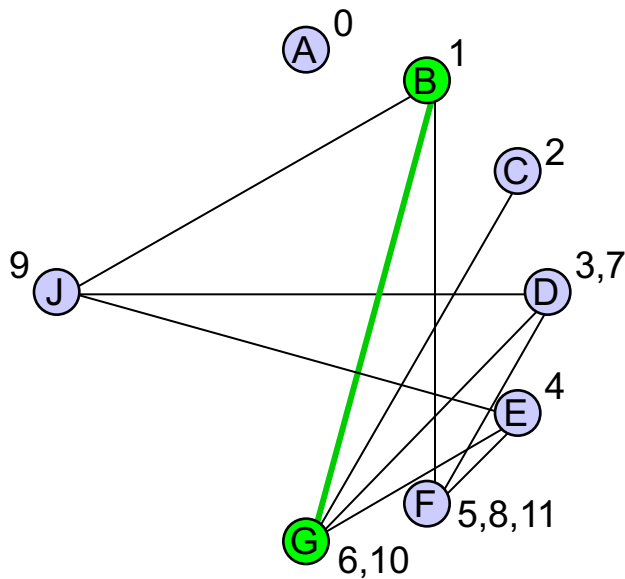
Clique Partitioning Algorithm (4)



A	B	C	D	E	F	G	H	I	J	
*	-	-	-	-	-	-	-	-	-	A: (0) [00:01]
-	*	-	-	-	1	0	-	1	0	B: (1) [00:10]
-	-	*	-	-	-	0	-	0	-	C: (2) [00:01]
-	-	-	*	-	2	1	4	2	1	D: (3) [01:10]
-	-	-	-	*	2	1	4	2	1	E: (4) [01:10]
-	1	-	2	2	*	-	3	4	-	F: (5) [01:10]
-	0	0	1	1	-	*	2	-	-	G: (6,10) [01:10]
-	-	-	4	4	3	2	*	3	2	H: (7) [10:10]
-	1	0	2	2	4	-	3	*	-	I: (8,11) [10:10]
-	0	-	1	1	-	-	2	-	*	J: (9) [10:01]

A	B	C	D	E	F	G	I	J	
*	-	-	-	-	-	-	-	-	A: (0) [00:01]
-	*	-	-	-	1	0	1	0	B: (1) [00:10]
-	-	*	-	-	-	0	0	-	C: (2) [00:01]
-	-	-	*	-	1	0	1	0	D: (3, 7) [11:10]
-	-	-	-	*	1	0	1	0	E: (4) [01:10]
-	1	-	1	1	*	-	3	-	F: (5) [01:10]
-	0	0	0	0	-	*	-	-	G: (6,10) [01:10]
-	1	0	1	1	3	-	*	-	I: (8,11) [10:10]
-	0	-	0	0	-	-	-	*	J: (9) [10:01]

Clique Partitioning Algorithm (5)



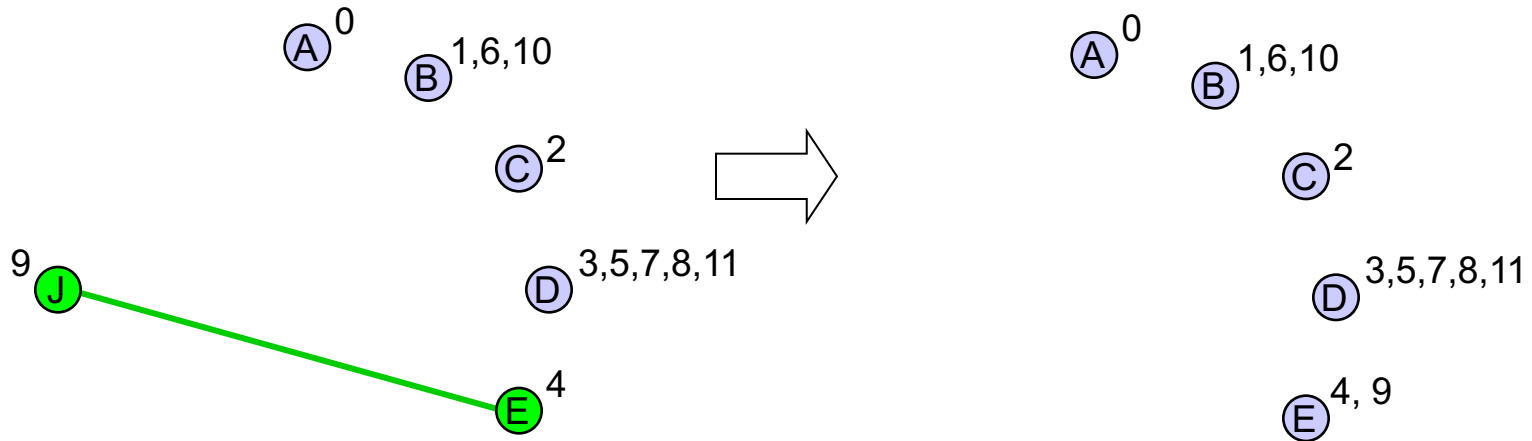
A B C D E F G J

=====								
*	-	-	-	-	-	-	-	A: (0) [00:01]
-	*	-	-	-	0	0	0	B: (1) [00:10]
-	-	*	-	-	-	0	-	C: (2) [00:01]
-	-	-	*	-	0	0	0	D: (3, 7) [11:10]
-	-	-	-	*	0	0	0	E: (4) [01:10]
-	0	-	0	0	*	-	-	F: (5, 8, 11) [11:10]
-	0	0	0	0	-	*	-	G: (6, 10) [01:10]
-	0	-	0	0	-	-	*	J: (9) [10:01]

A B C D E F J

=====							
*	-	-	-	-	-	-	A: (0) [00:01]
-	*	-	-	-	-	-	B: (1, 6, 10) [01:10]
-	-	*	-	-	-	-	C: (2) [00:01]
-	-	-	*	-	0	0	D: (3, 7) [11:10]
-	-	-	-	*	0	0	E: (4) [01:10]
-	-	-	0	0	*	-	F: (5, 8, 11) [11:10]
-	-	-	0	0	-	*	J: (9) [10:01]

Clique Partitioning Algorithm (6)



A	B	C	D	E	J	
*	-	-	-	-	-	A: (0) [00:01]
-	*	-	-	-	-	B: (1, 6,10) [01:10]
-	-	*	-	-	-	C: (2) [00:01]
-	-	-	*	-	-	D: (3, 5, 7, 8,11) [11:10]
-	-	-	-	*	0	E: (4) [01:10]
-	-	-	-	0	*	J: (9) [10:01]

A	B	C	D	E	
*	-	-	-	-	A: (0) [00:01]
-	*	-	-	-	B: (1, 6,10) [01:10]
-	-	*	-	-	C: (2) [00:01]
-	-	-	*	-	D: (3, 5, 7, 8,11) [11:10]
-	-	-	-	*	E: (4, 9) [11:11]

Final register binding result →

A : v_0

B : v_1, v_6, v_{10}

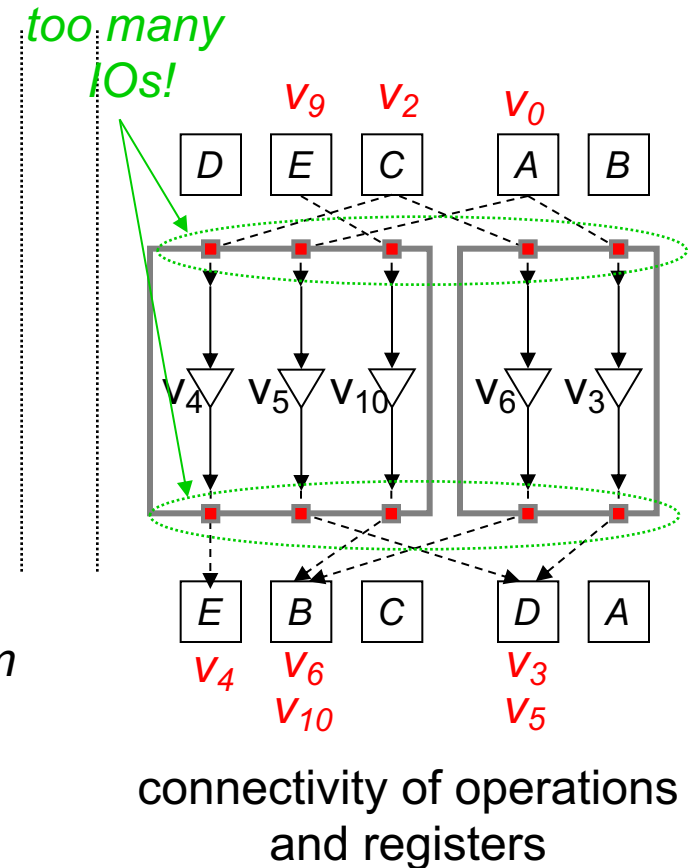
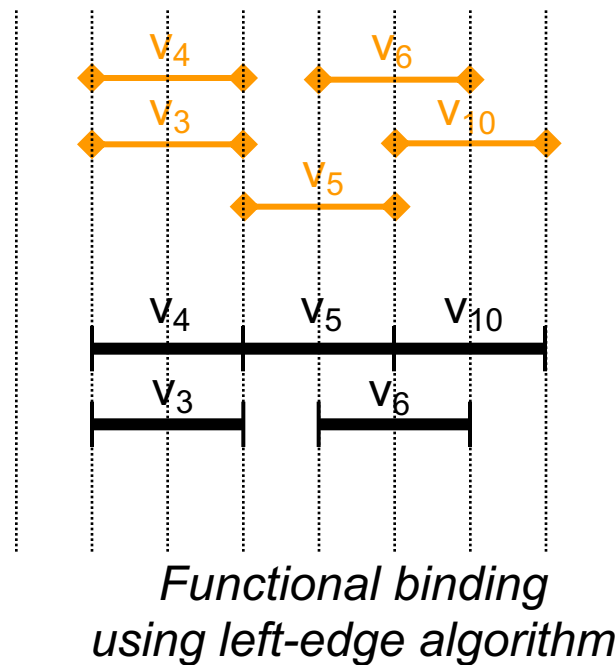
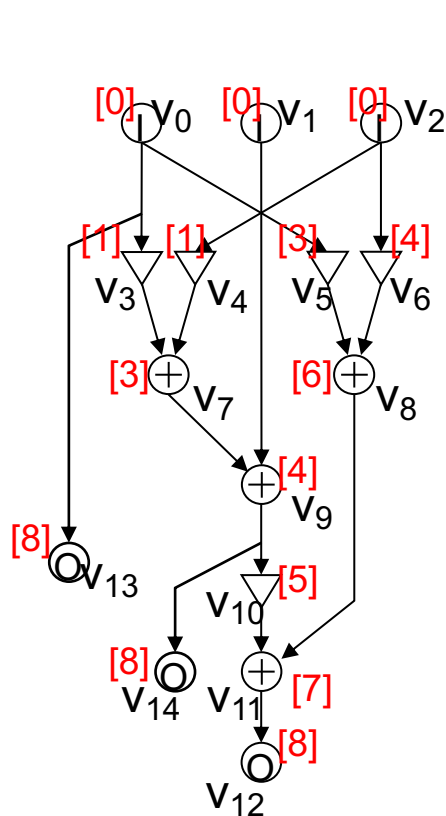
C : v_2

D : $v_3, v_5, v_7, v_8, v_{11}$

E : v_4, v_9

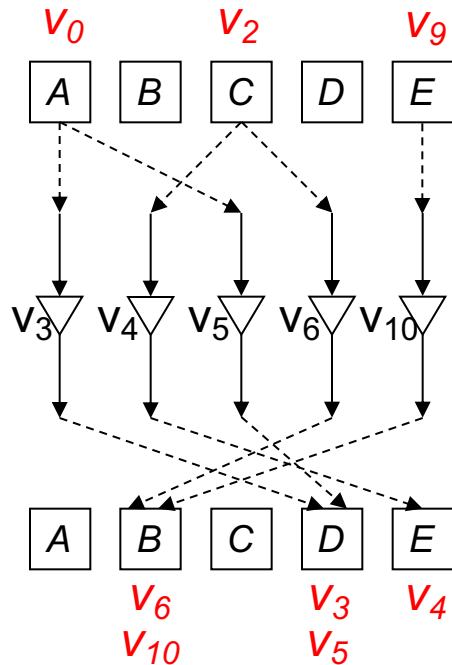
Functional Unit Binding (1)

- ❑ Allocate each operation to functional unit instance
 - Left-edge algorithm cannot consider interconnect cost
 - Interconnect cost : # of IO ports

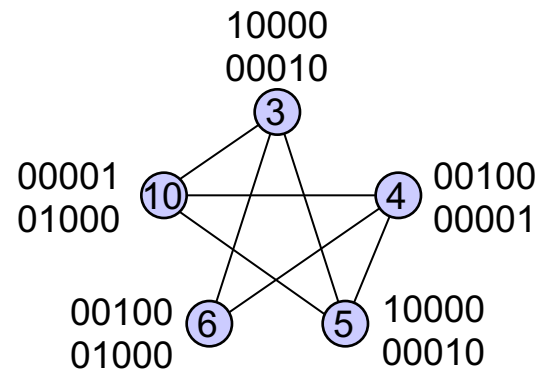
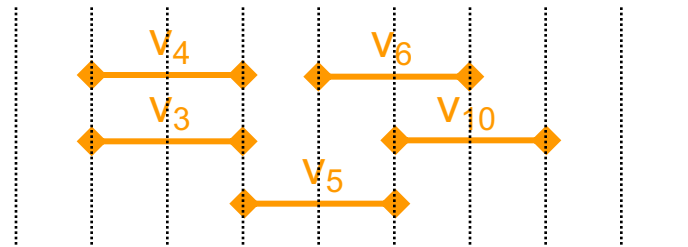


Functional Unit Binding (2)

- Use clique partitioning and consider the interconnect cost
 - Resource vector : each element represent register

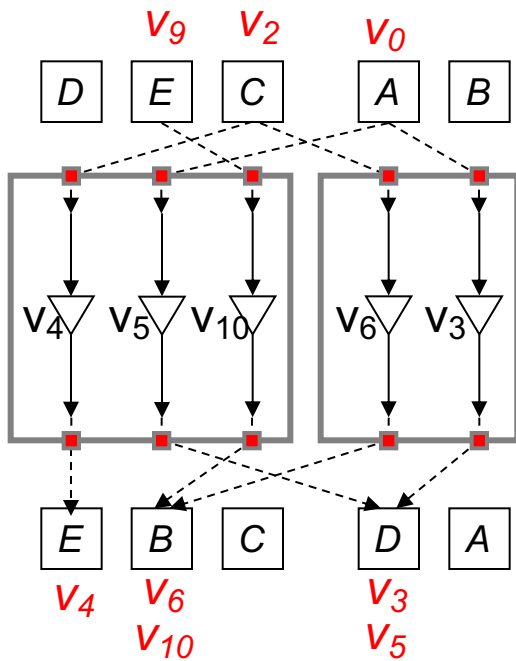
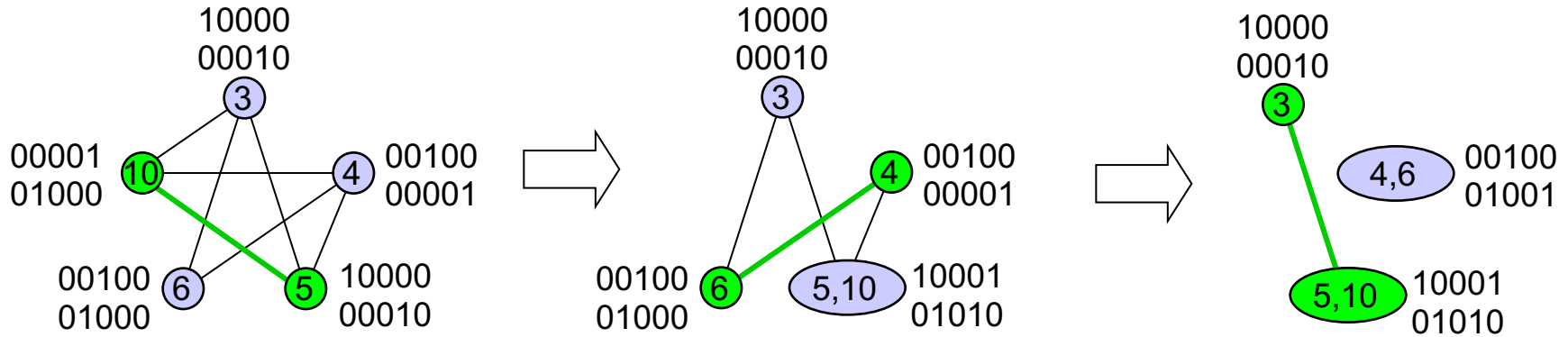


connectivity of operations
and registers

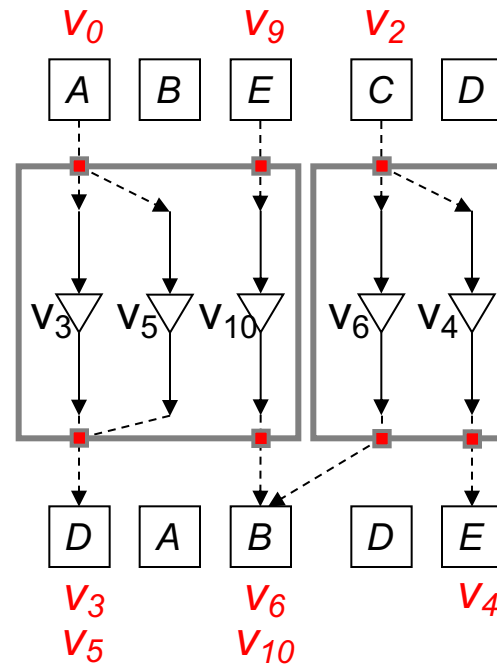


compatibility graph

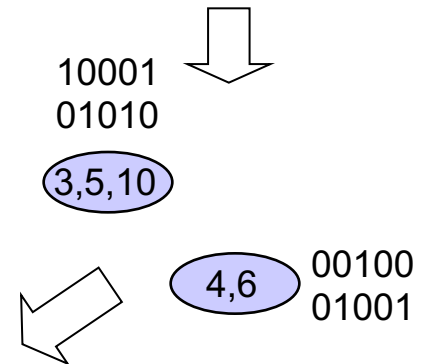
Functional Unit Binding (3)



FU binding with left-edge algorithm



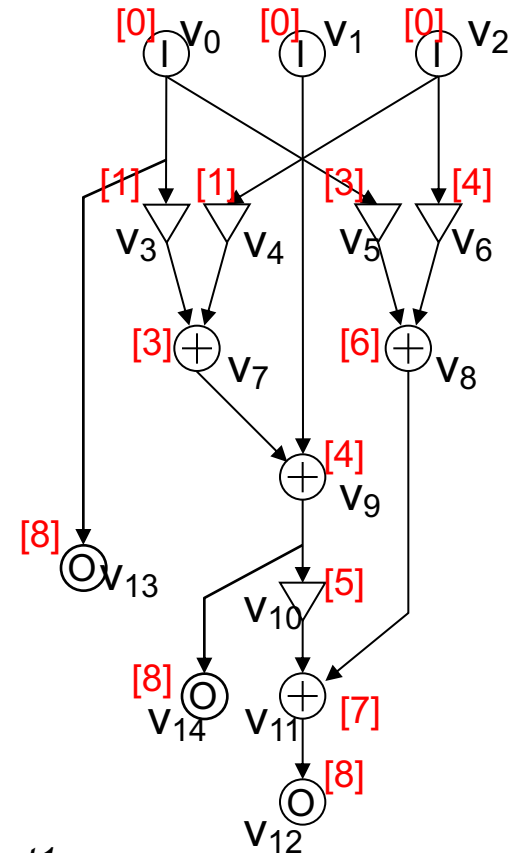
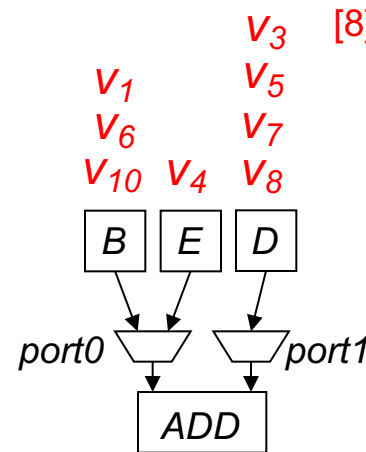
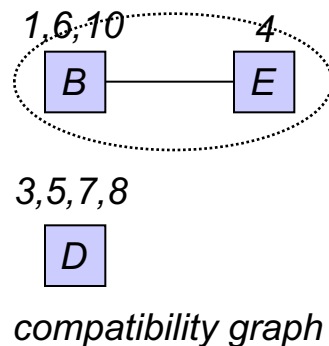
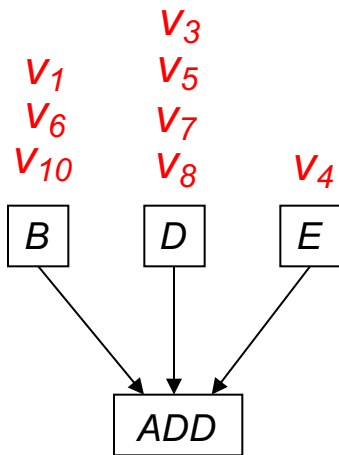
FU binding with clique partitioning



Disadvantage using clique partitioning : minimum # of FUs is NOT guaranteed! (but often OK)

Port Binding (1)

- ❑ Assign data to ports for units with multiple ports (ex:adder)
 - input data simultaneously used for an operation are not compatible (must be assigned to different ports)
 - Use clique partitioning : works for this example → *but does not work in general*

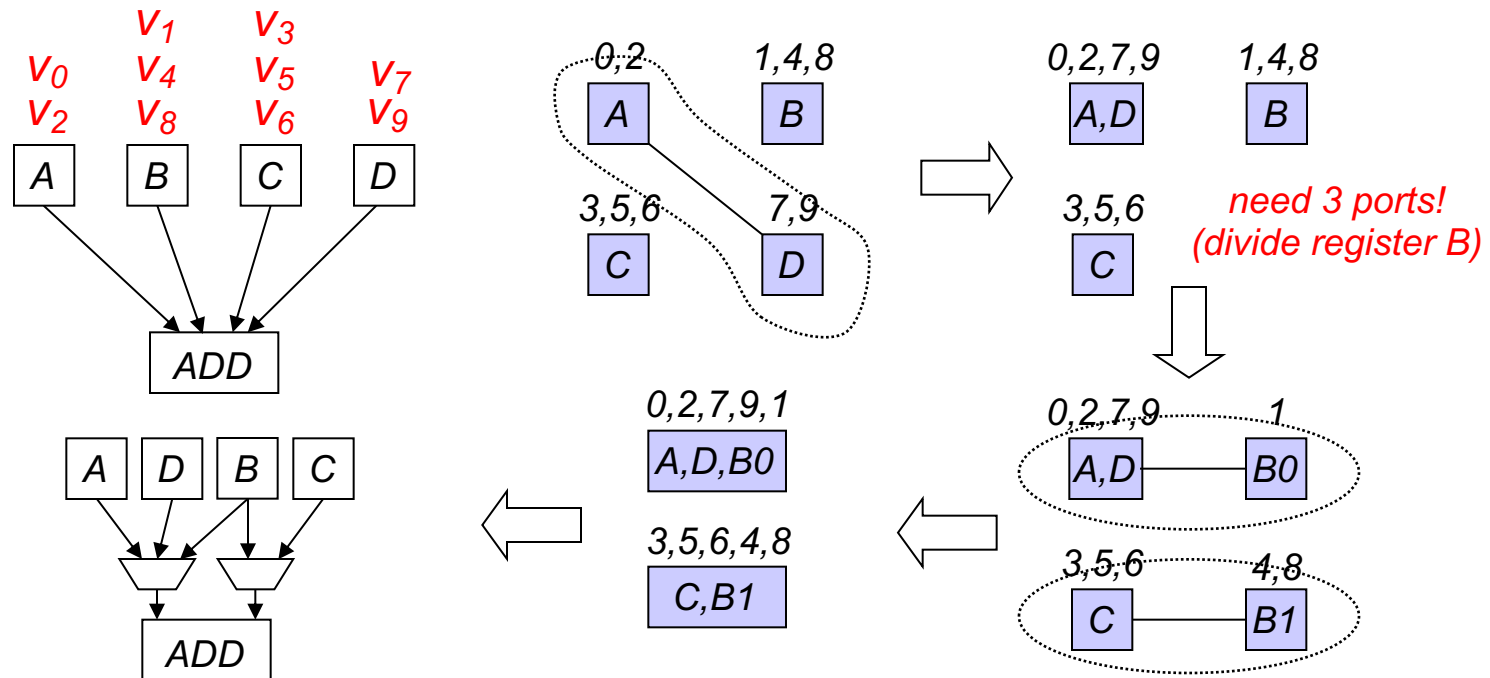


Port Binding (2)

□ Consider 5 additions

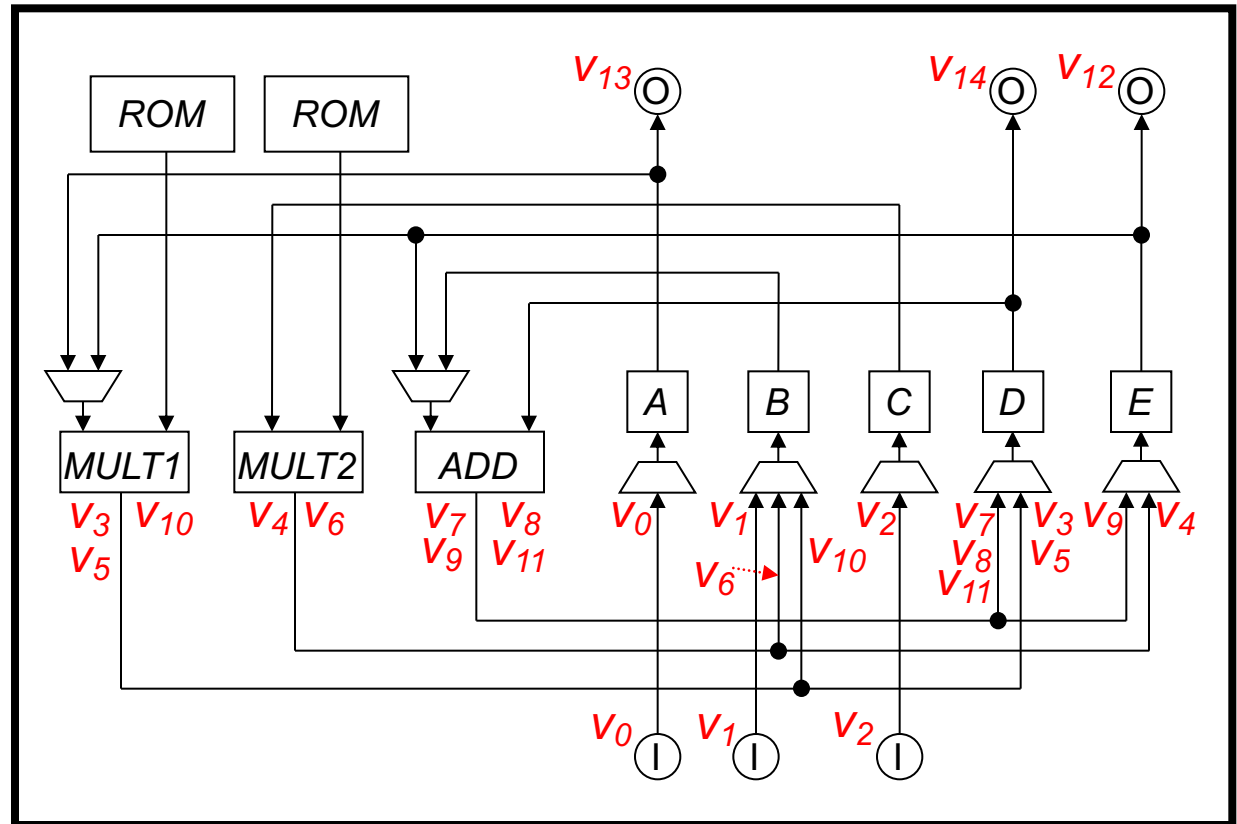
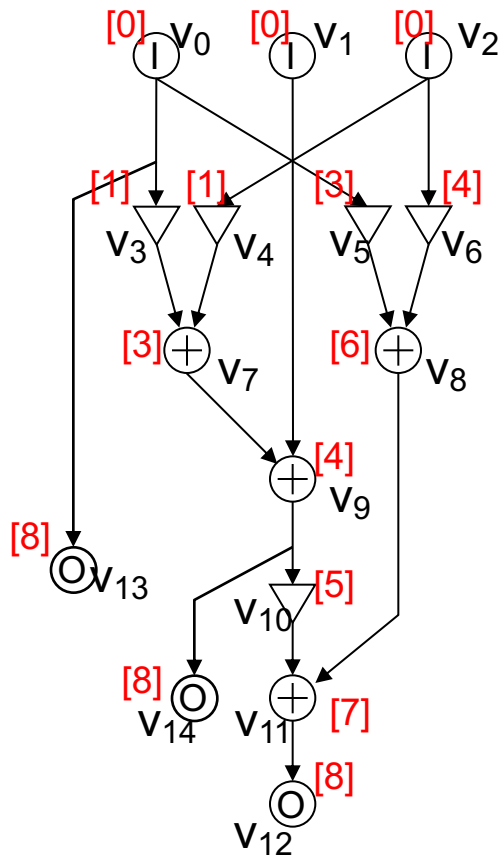
$$V_0 + V_1, V_2 + V_3, V_4 + V_5, V_6 + V_7, V_8 + V_9$$

- Registers A : (v_0, v_2), B : (v_1, v_4, v_8), C : (v_3, v_5, v_6), D : (v_7, v_9)
- Cannot directly allocate registers to 2 ports (register B need to be allocated to both ports)



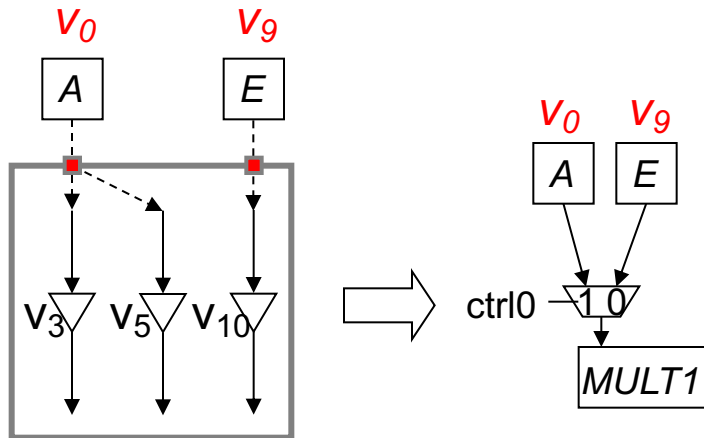
Datapath Construction

- After resource binding, construct the datapath
 - Netlist composed of functional units, registers, multiplexers, etc.



Controller Generation (1)

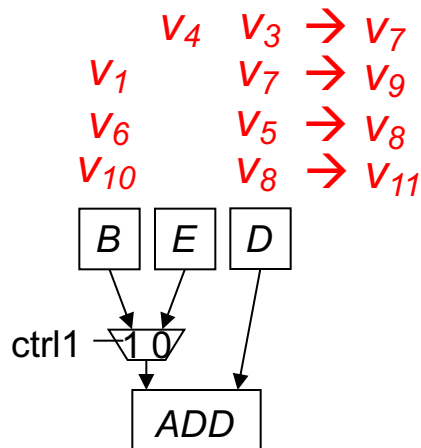
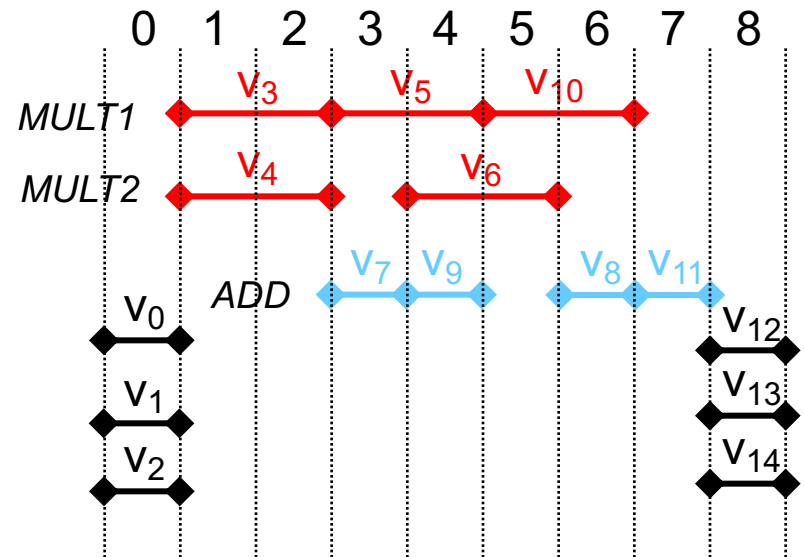
- Control signal generation for functional units



MULT1

```

always@(state)
  if (state == S1 || state == S2 ||
      state == S3 || state == S4)
    ctrl0 = 1;
  else if (state == S5 || state == S6)
    ctrl0 = 0;
  else ctrl0 = 1b'x;
    
```



```

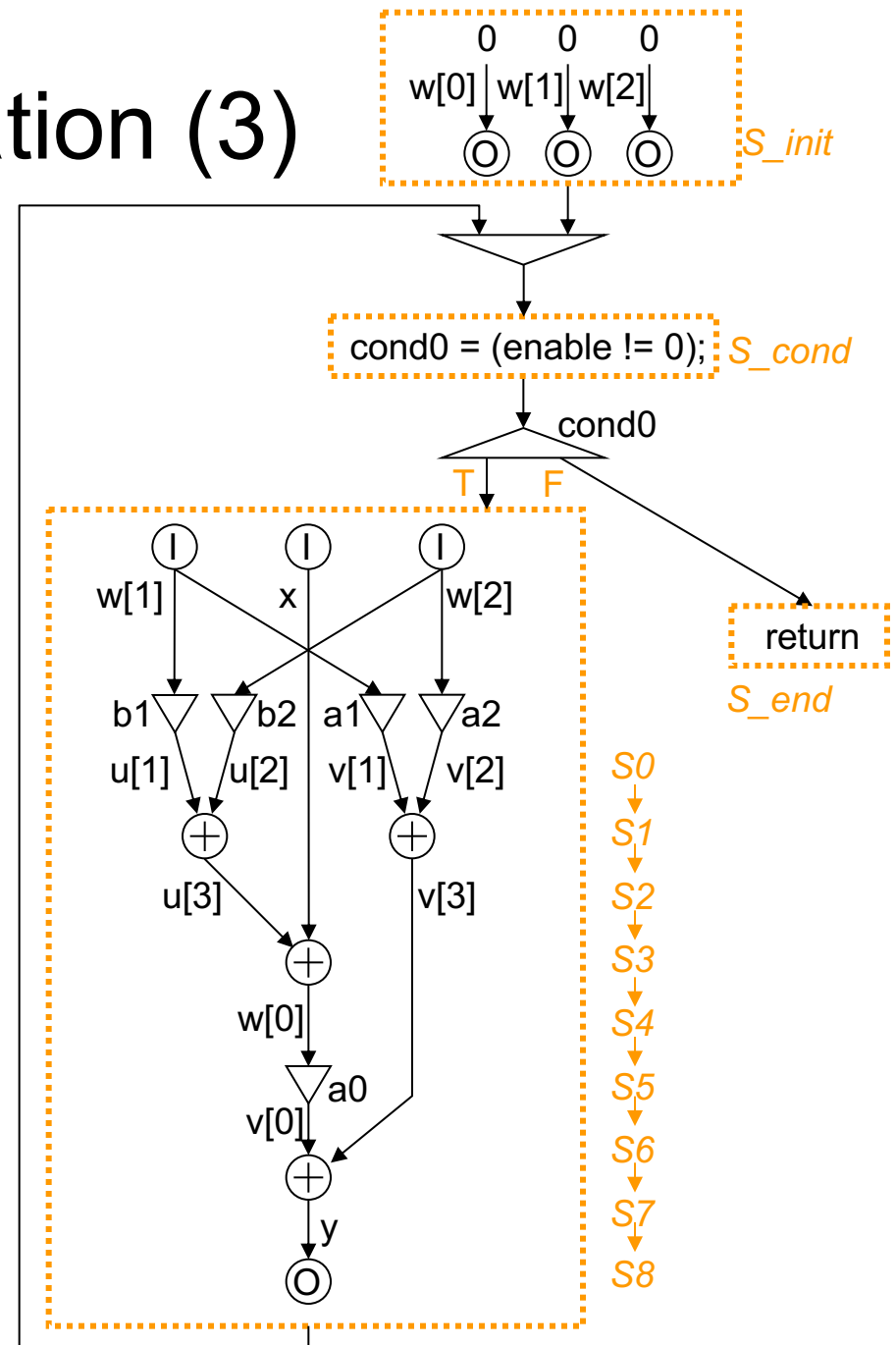
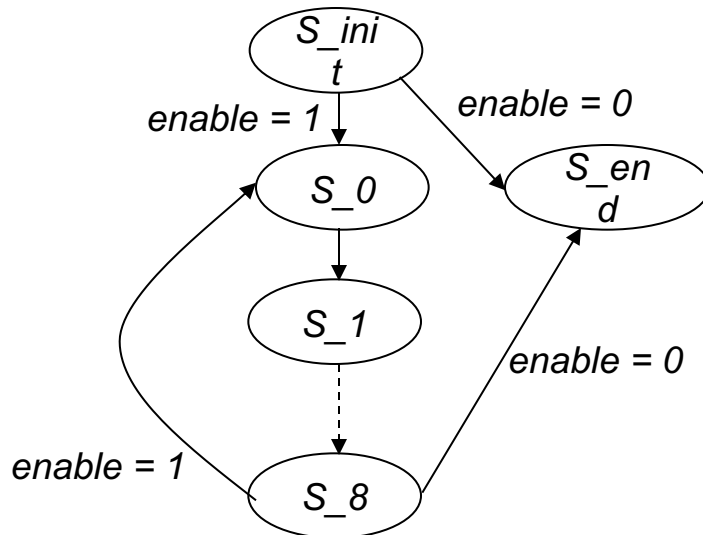
always@(state)
  if (state == S4 || state == S6 ||
      state == S7)
    ctrl1 = 1;
  else if (state == S3)
    ctrl1 = 0;
  else ctrl1 = 1b'x;
    
```


Controller Generation (3)

- Finite state machine construction

```

initial state = S_init;
always@(posedge clk)
  if (state == S_init || state == S8)
    if (enable != 0) state = S0;
    else state = S_end;
  else if (state == S0) state = S1;
  else if (state == S1) state = S2;
  else if ....
    
```



High-Level Synthesis Summary (1)

□ High-Level Synthesis flow :

- A) Design capture (HDLs, C/C++, signal-flow graph, etc)
- B) Compilation to internal representation (Control-Data-Flow Graph)
- C) Resource allocation
- D) Operation scheduling
- E) Resource binding
- F) RTL description generation
 - ✓ Datapath construction
 - ✓ Controller generation

High-Level Synthesis Summary (2)

- ❑ Scheduling and a series of resource bindings (registers, functional units, ports, etc.) are very closely related
 - Consider the register cost, interconnect cost, and other hardware costs during scheduling → can incorporate these costs as *forces* in force-directed scheduling
 - Simultaneous/iterative scheduling and binding → more accurate hardware cost evaluation on actual binding and feedback to the scheduler

High-Level Synthesis Summary (3)

- ❑ Extracting parallelism beyond basic-blocks
 - Loop pipelining (overlap scheduling of successive loop iterations)
 - Well studied in DSP applications
 - Becomes complicated when loop includes control-flows
 - Multiple control-flow execution
 - Not well studied yet in terms of automatic synthesis
 - Key issue is how to extract global parallelism (related to the description language issues)
 - Manually done by the hardware designers