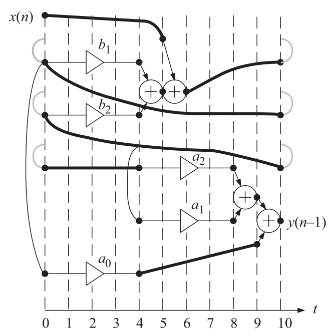
Application Specific Integrated Circuits for Digital Signal Processing
Lecture 7

Oscar Gustafsson

Today's topics

- Scheduling formulations
- Scheduling algorithms
- Memory
- Resource allocation
- Resource assignment

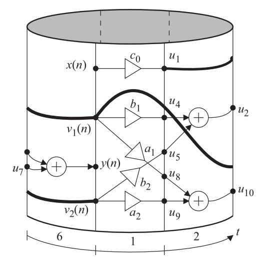
Scheduling Example



- Two possibilities to find a solution with two multipliers
  - Schedule for more than one sample period: increased flexibility
  - Use cyclic scheduling: "ignore" the start and stop time of the schedule

Cyclic Scheduling

- Iterative algorithms will be executed over and over again



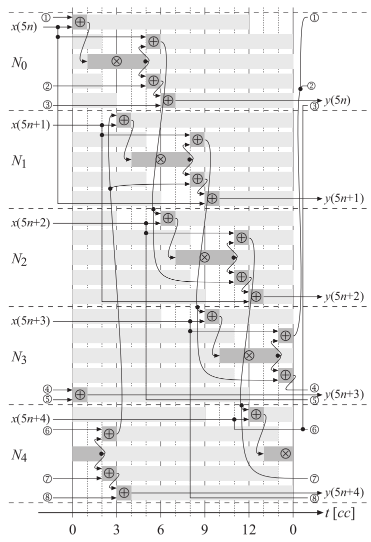
- Really no need for a start and a stop time for the schedule
- Can move operations across the scheduling boundary (equivalent to retiming/pipelining the SFG)
- Operations can be executed across the scheduling boundary

Cyclic scheduling example

- Third-order bit-serial bireciprocal LWDF filter
  - Adaptor coefficient $\alpha = 0.375$
  - Wordlength = 11 bits
  - $T_{sample} = T_{min}$
- Use extra register after each operation to increase clock frequency
  - $T_{L,add} = 1$ clock cycle
  - $T_{L,mult} = W_f + 1 = 4$ clock cycles
- $T_{min} = \frac{4+1+1}{2} = 3$ clock cycles
- $\max\{T_{exe}\} = 15$ clock cycles (latency + wordlength)
- $T_{schedule} = K \times T_{sample} = K \times T_{min} \geq 15 \Rightarrow K = 5$

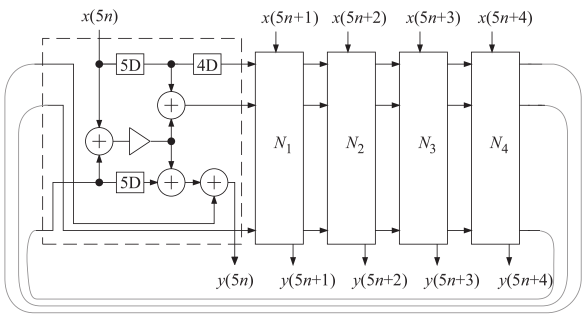
Cyclic scheduling example

- Five concurrent computations
- Dark gray corresponds to latency
- Light gray corresponds to execution time
- Note that there are no shimming delays in the critical loop



Cyclic scheduling example

- Resulting implementation

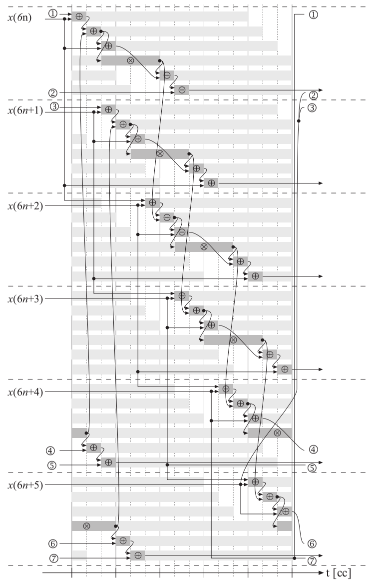


Cyclic scheduling example II

- Apply arithmetic transformations to remove one addition from the loop
- $T_{min} = \frac{4+1}{2} = 2.5$ clock cycles
- $\max\{T_{exe}\} = 15$ clock cycles (latency + wordlength)
- $T_{schedule} = K \times T_{sample} = K \times T_{min} \geq 15 \Rightarrow K = 6$

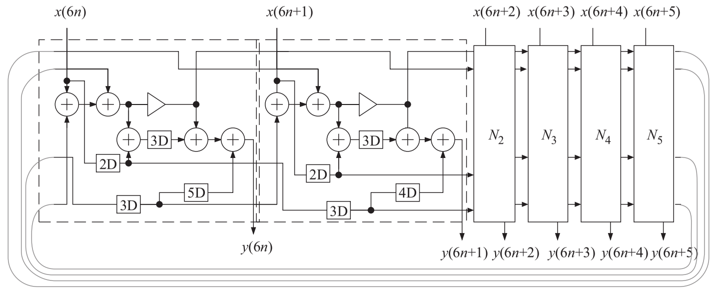
Cyclic scheduling example II

- Six concurrent computations
- The delay between the different outputs varies between two and three clock cycles
- On average 2.5 clock cycles



Cyclic scheduling example II

- Resulting implementation

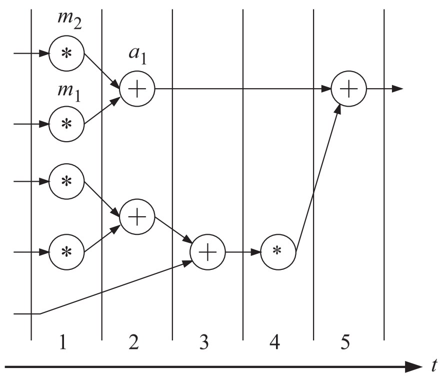


Scheduling algorithms

- Automatic scheduling can be useful to reduce the design time, especially for complex algorithms
- Some definitions
  - Heuristic – method of finding a "good enough" solution quickly
  - Constructive algorithm – an algorithm that constructs a solution
  - Iterative algorithm – an algorithm that refines a previous solution
  - Greedy algorithm – an algorithm that always takes the seemingly best step while not considering later consequences
- Most scheudling problems are NP-hard $\Rightarrow$ exhaustive search required to (find and) prove optimal solution

Scheduling algorithms

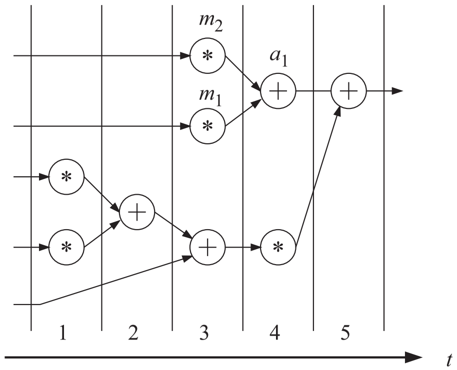
- As-soon-as-possible (ASAP) scheduling



- Shortest possible scheduling time without considering resources
- Typically what is obtained when introducing timing to the precedence graph

Scheduling algorithms

- As-late-as-possible (ALAP) scheduling



- Useful to obtain scheduling ranges (latest possible starttime for operations to finish in time)
- ASAP + ALAP gives the possible ranges for each operator (if cyclic scheduling is not considered)

Scheduling algorithms

- Integer linear programming scheduling
  - Describe schedule using linear relations

$$T_{start,1} + T_{L,1} + Shimming_1 = T_{start,2} \tag{1}$$

  - Can be solved optimally using standard techniques, e.g., branch-and-bound
  - Time consuming for large problems
  - Must describe objective fuction using linear expression

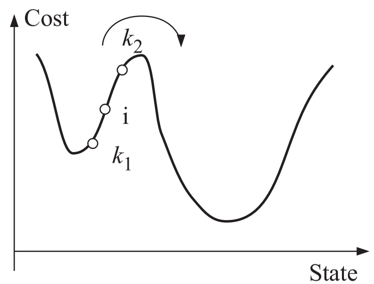
$$minimize \sum Shimming_i \tag{2}$$

Scheduling algorithms

- Earliest deadline scheduling
  - In each time step, schedule the process, processes whose deadline is closest
  - Scheduling time optimal for single PE
- Slack time scheduling
  - Schedule the process whose slack time is the least
  - Slack time = Time to deadline = (remaining) execution time
  - Better than or as good as earliest deadline for more than one PE

Heuristic optimization methods

- Straightforward search methods often get stuck in local minimums



- The methods we will consider try to avoid getting stuck, primarily by using a certain degree of randomization
- Can be used to solve "all" optimization problems
- Often inspired by nature

Simulated annealing

- Simulate the cooling of solid materials reaching equilibrium
- 1. Set temperature, T, and initial solution
  2. Find nearby possible solution
  3. Determine difference in cost between the new and the current solution, Δ_{cost}
  4. If new solution is better take that as the current
  5. If not, do still take it as the current if $e^{\frac{\Delta_{cost}}{T}} > random(0, 1)$
  6. Decrease temperature and go to step 2
- Possible to escape local minimums by accepting a worse solution at certain probability
- Works well for many problems, but one will have to fine tune the parameters
  - How fast to decrease the temperature
  - What is nearby?
  - ...

Genetic algorithms

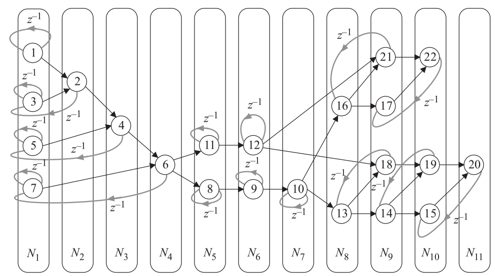
- Simulated evolution
- Solutions represented as genes
- Set of solutions acting as parents
- These parents are combined and mutated to form new solutions (children)
- The best solutions (strongest individuals) are used as parents for the next generation
- Parameters
  - Size of generation?
  - Combination and mutation rates?
  - ...

Some more

- Tabu search
  - Greedy local search, go in the best direction
  - The latest solutions are "Tabu", i.e., one is not allowed to go back to those
- Bug swarm optimization
  - Solutions are "bugs" that flies towards the best solution but at the same time a bit at random
- Ant Colony optimization
- Artifical Immune System Optimization
- ...

Interpolator Case Study

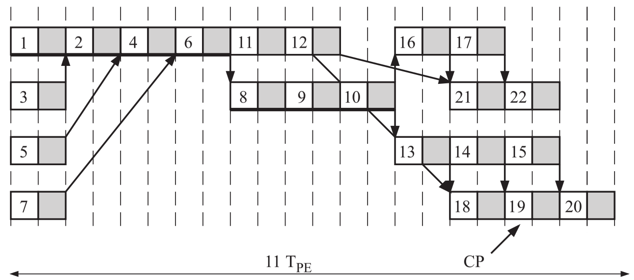
- Adaptor operation is chosen as PE
- Precedence graph



- Pipelined bit-serial adaptor has $T_L = 48$ clock cycles and $T_{exe} = 24$ clock cycles
- Select a time unit to be 24 clock cycles

Interpolator Case Study

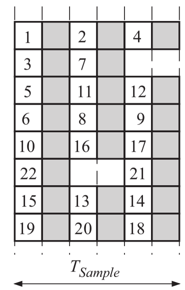
- Initial computation graph



- Can be folded to six time units input sample periods (identical to introducing pipelining at multiples of six time units)
- This is because six time units is larger than T_{min}

Interpolator Case Study

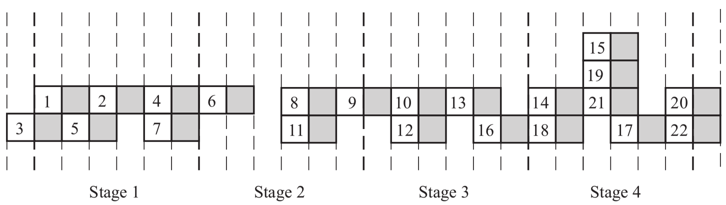
- Folded schedule



- Requires that eight adaptor operations start at the same time
- $2 \times 8 = 16$ concurrent memory reads
- $2 \times 8 = 16$ concurrent memory writes
- Reschedule

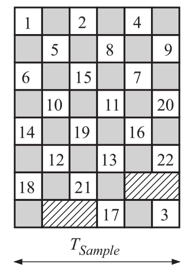
Interpolator Case Study

- Improved schedule



Interpolator Case Study

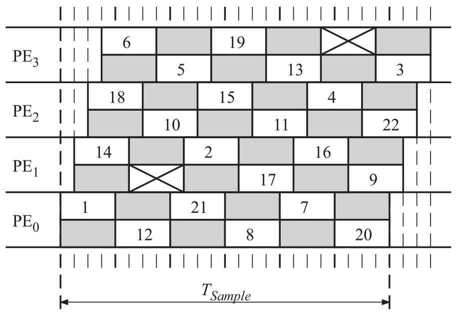
- Improved folded schedule



- Requires that four adaptor operations start at the same time
- $2 \times 4 = 8$ concurrent memory reads
- $2 \times 4 = 8$ concurrent memory writes
- Increase time resolution and skew each row of operations

Interpolator Case Study

- Improved skewed folded schedule



Memory

- The data storage/memory should perform the following task
  - At time T_1 receive a data and store it until time T_2 when it is returned
  - Typically, there are multiple data and some data may be returned more than once
- The problem is to determine a memory system that uses as few resources as possible
- The memory can be implemented either using RAMs or flip-flops

Memory

- The first problem is to determine how many memories are required
  - Based on the amount of concurrent memory accesses
  - Either use multiple read and write ports or multiple memories (or both)
  - Multiple write-ports are often costly
  - Multiple read-ports are easier (especially when using flip-flops as all data can be accessed in parallel)

Memory

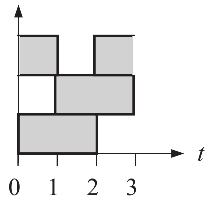
- The second problem is to determine the number of cells/flip-flops to store the required amount of data and which storage to use when
  - Variables not overlapping in time can share resource
  - Number of concurrently stored variables puts a lower limit on the amount of resources

Resource allocation and assignment

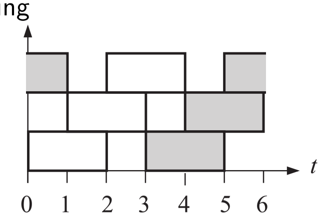
- Resource allocation
  - Processes without overlapping execution times can share the same resource
  - Variable storage has an execution time identical to the time the variable must be stored
- Resource assignment
  - Select which process is performed on which resource
  - Impact on communication (buses, switches etc)
- Goal
  - Maximize resource utilization
  - Minimize total cost (including e.g. communication cost)

Resource allocation and assignment

- Largest number of concurrent processes determines the number of resources required
- Not always true!



- No valid allocation using two PEs
- Solution – unfolding



Algorithms for resource allocation and assignment

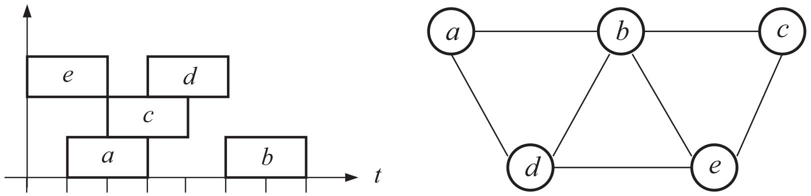
- Often a valid assignment is obtained without an explicit allocation step
- Algorithms
  - Clique partitioning – well studied graph theoretic problem, NP hard but optimal
  - Left-edge algorithm – constructive heuristic

Clique partitioning

- Two alternatives to forming the graph representation the possible sharing
  - Inclusion graph – focus on processes that can share resource
  - Exclusion graph – focus on processes that can not share resource
- The inclusion graph is the complement graph of the exclusion graph and vice versa
  - Nodes which are connected in the inclusion graph are not connected in the exclusion graph and vice versa

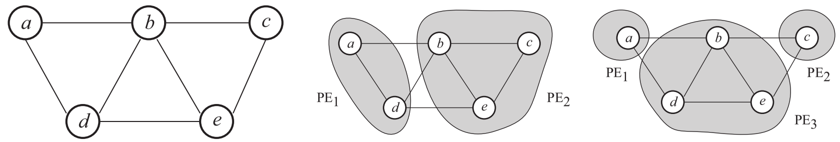
Clique partitioning using inclusion graphs

- Form an undirected graph where
  - Each process corresponds to a node
  - Two processes that can share the same resource are connected with an edge
- Example



Clique partitioning using inclusion graphs

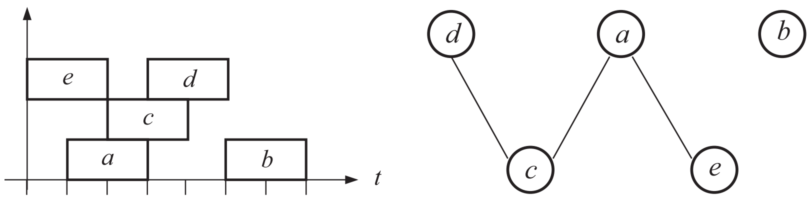
- A clique is a fully connected subgraph, i.e., all nodes have edges to all other nodes in the subgraph
- This means that all processes in a clique can share resource



- Finding the minimum number of cliques is an NP-hard problem
- Well studied in literature so one can use existing approximation algorithms

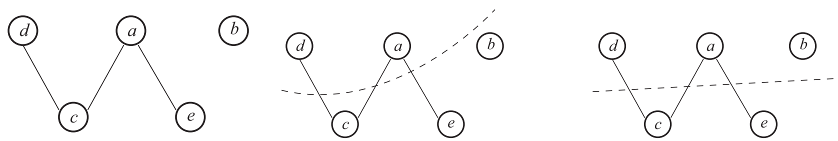
Clique partitioning using exclusion graphs

- Form an undirected graph where
  - Each process corresponds to a node
  - Two processes that can not share the same resource are connected with an edge
- Example



Clique partitioning using exclusion graphs

- We would like to disconnect the connected nodes
- Find cuts that crosses all edges at least once
- Nodes/processes between/outside of the cuts can share the same resource



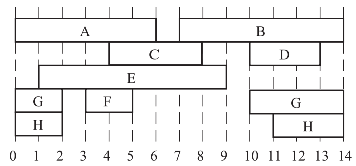
- Equivalent to coloring the graph
  - Find color assignments such that two connected nodes have different colors
  - Nodes/processes with the same color can share resource
- Often convenient to select the graph with fewest edges when determining if to use inclusion or exclusion graphs

Left-edge algorithm

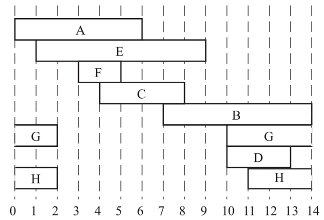
- Fast (but not optimal for cyclic execution times hence a heuristic)
- Constructive (run once, no previous solution) and greedy (pick the seemingly best process) algorithm
  1. Sort processes according to start time (equal start time ⇒ sort on longest execution time)
  2. Assign first process to a free resource
  3. Assign the first process with non-overlapping life-time to the same resource
  4. If process assigned and processes remaining go to step 3
  5. If no process could be assigned and processes remaining add resource and go to step 2
  6. If not processes remaining we are done

Left-edge algorithm example

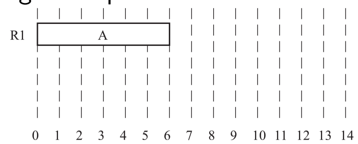
Initial processes



Sort processes based on start time

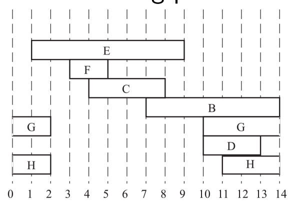


Assign first process to a free resource

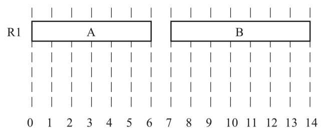


Left-edge algorithm example

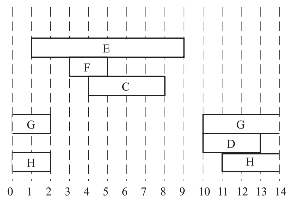
Remaining processes



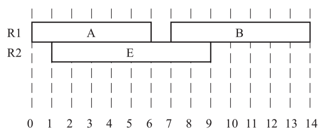
Assign first non-overlapping process



Remaining processes

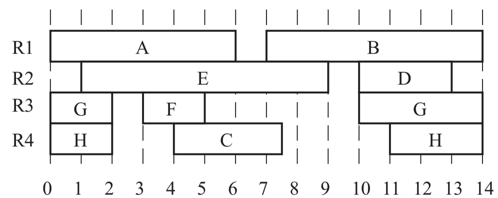


No non-overlapping processes so add a new resource and assign first process



Left-edge algorithm example

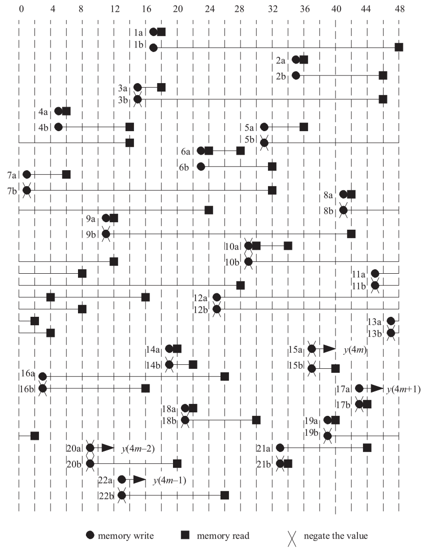
- Continue along the same lines to end up with the final result



- Four resources required

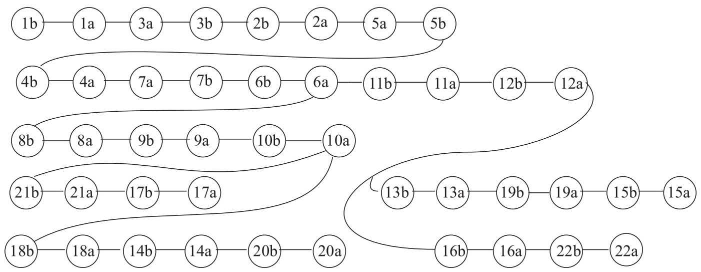
Interpolator case study

- Extract all memory variables
- Delay output from adaptor with one clock cycle to avoid concurrent reading and writing
- To avoid multi-port memories use two memories
- Partition memory variables with identical read or write times into different memories



Interpolator case study

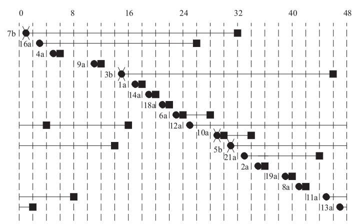
- Create exclusion graph based on concurrent memory accesses



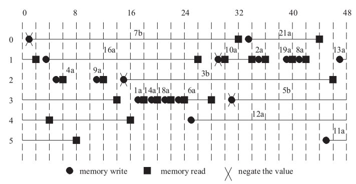
- Find a cut that crosses each edge once
- Results in two different sets of memory variables

Interpolator case study

- Use left edge algorithm for each set of memory variables
- Memory 1

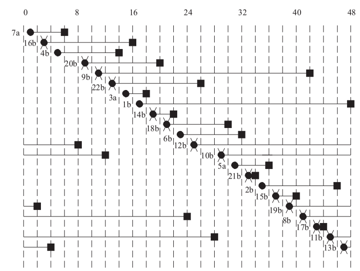


- Resulting memory assignment using six memory positions

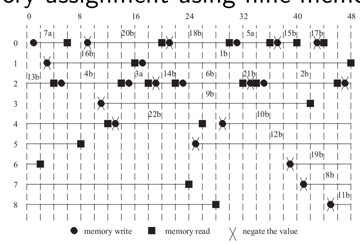


Interpolator case study

- Memory 2

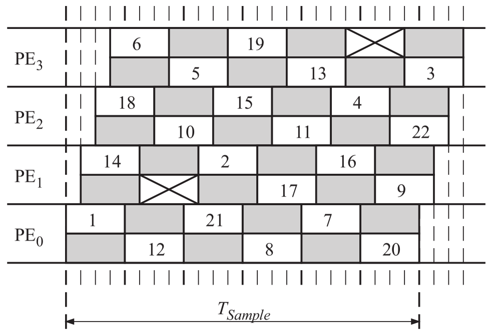


- Resulting memory assignment using nine memory positions



A case where the left edge algorithm is not optimal

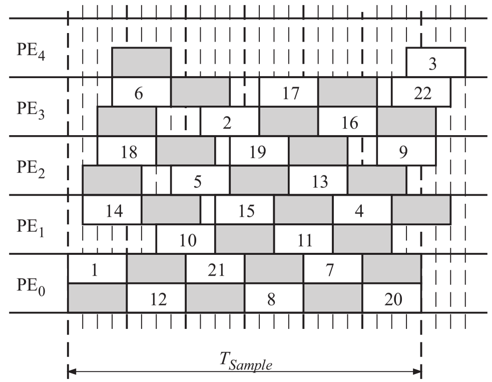
- Consider the final schedule for the interpolator case study



- Apply the left-edge algorithm

A case where the left edge algorithm is not optimal

- The “empty” slots causes problems
- Resulting resource allocation and assignment



- Slightly better results may be obtained for different start times (use the cyclic property)