Hardware-Software Co-Design: System Partitioning

EE8205: Embedded Computer Systems
http://www.ee.ryerson.ca/~courses/ee8205/

Dr. Gul N. Khan
http://www.ee.ryerson.ca/~gnkhan
Electrical and Computer Engineering
Ryerson University

Overview

- Hardware-Software Codesign
- Task Graph Representations
- Scheduling for Partitioning
- GDL Scheduling and Partitioning
- DADGP-based Partitioning

Introductory Articles on Hardware-Software Partitioning available at the course webpage,

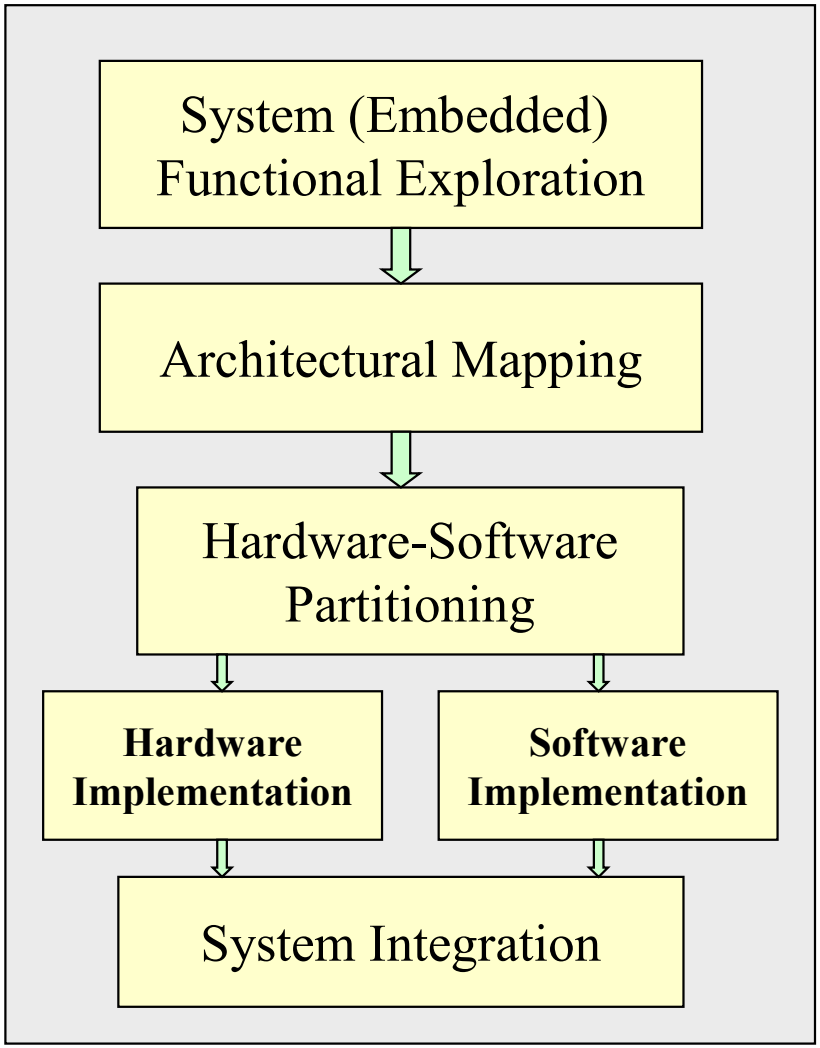
Part of Chapter 7, 5 of the Text by Wayne Wolf

Embedded System Design

Embedded Computer Systems are the ideal candidate for hardware-software codesign.

- Separate HW and SW design has been explored and examined very thoroughly.
- Joint design remains an area of rapidly growing study
- Old embedded devices always built from scratch
  - **within reasonable amount of time**
- Components - smaller and faster **- IP cores**
- Tools required for the product engineer.

Hardware-Software Codesign



- Functional exploration: Define a desired product's requirements and produce a specification of the system behavior.
- Map this specification **onto various hardware and software architectures**
- Partition the functions between silicon and code, and map them **directly to hardware or software components**
- Integrate system **for prototype test.**

HS-Codesign

- Co-Specification: Describe system functionality at the abstract level
- System description is converted into a task graph representation
- HW-SW Partitioning: Take the task graph and decide which components are implemented where/how ?

  i.e. Dedicated hardware,

  Software -- **one CPU or multiple CPUs**

HS-Codesign

- HW-SW Co-Synthesis: Analyze the task graph and decide on the system architecture.

  (incorporates HW/SW partitioning as heart of co-synthesis process)

- HW-SW Co-Simulation: Simulate embedded device's functionality before prototype construction.

- Co-Verification: Mathematical or simulation based verification that device meets requirements.

HW/SW Partitioning

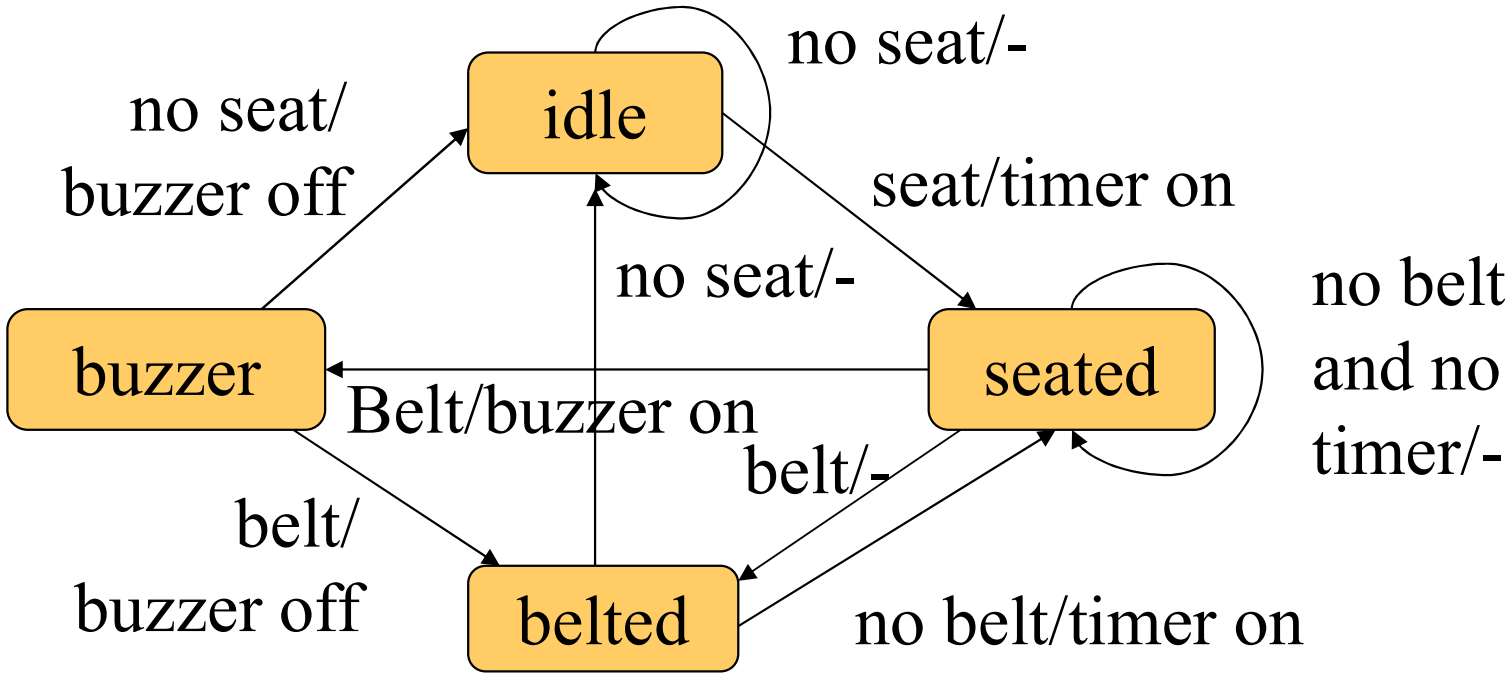
- Both textual and graphical representation like DAG **(Directed Acyclic Graph)** are used to describe system.
- Analyzes task graph to determine each task's placement **(HW or SW)**
- Many algorithms have been developed
- Major problem involves the computation time of the algorithm.

System Design Patterns

Design Pattern: A generalized description of the design of a certain type of program that can also be used for system representation and hardware-software partitioning.

- State Diagram
- Data Flow Graph (DFG)
- Control Data Flow Graph (CDFG)
- Directed Acyclic Graph (DAG) similar to DFG
- Directed Acyclic Data Dependence Graph with Precedence (DADGP) proposed by one of my past Graduate student.

State Machine: Seat-belt System




```
switch (state) {
    case IDLE: if (seat) { state = SEATED; timer_on = TRUE; } break;
    case SEATED: if (belt) state = BELTED;
                 else if (timer) state = BUZZER; break;
    ……
}
```

Data Flow Graph

DFG: Data Flow Graph

- DFG does not represent control
- It models the Basic Block: code or a system block with one entry and exit
- Describes the minimal ordering requirements on operations

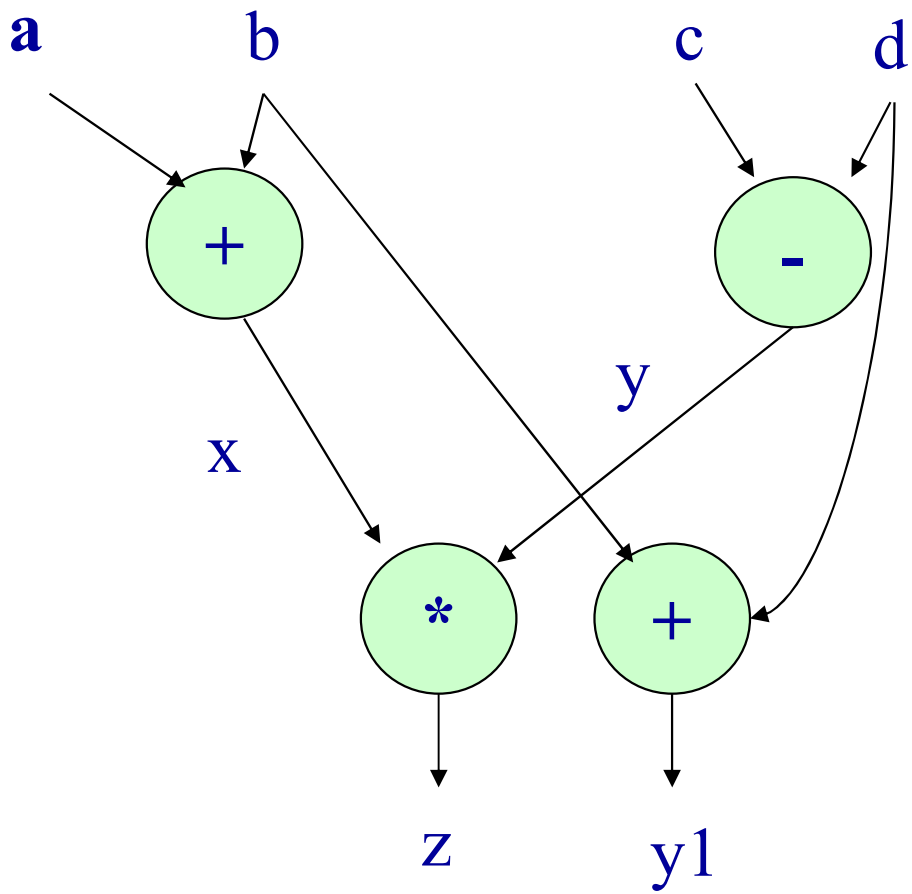
Data Flow Graph: Software Module

x = a + b;

y = c - d;

z = x * y;

y1 = b + d;



DFG

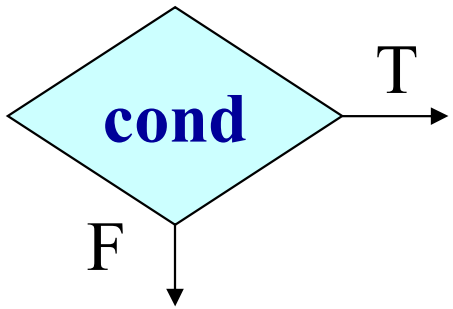
Control Data Flow Graph

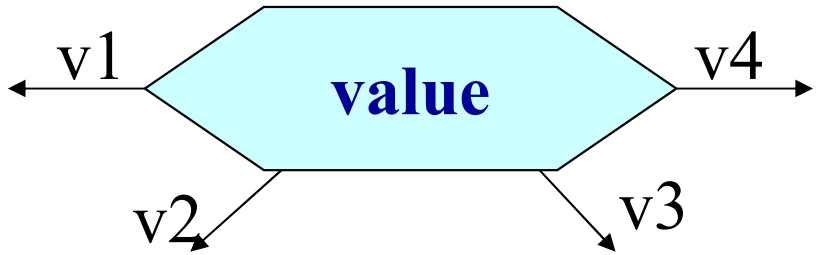
CDFG: represents control and data.

- Uses data flow graphs as components.
- Two types of nodes:
  - **Data Flow Node encapsulate a DFG**

```
x = a + b;
y = c + d
```

  - **Decision Nodes**

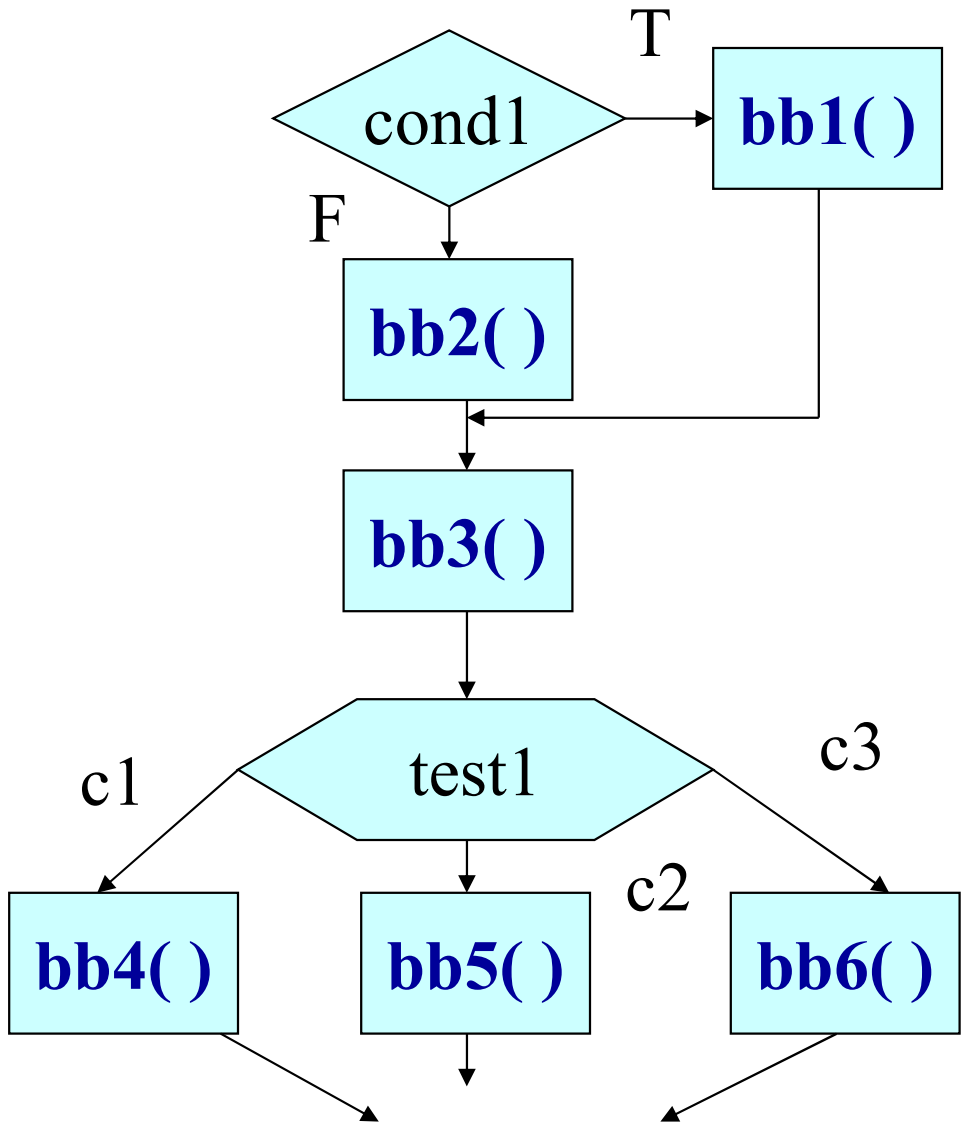




Equivalent Forms

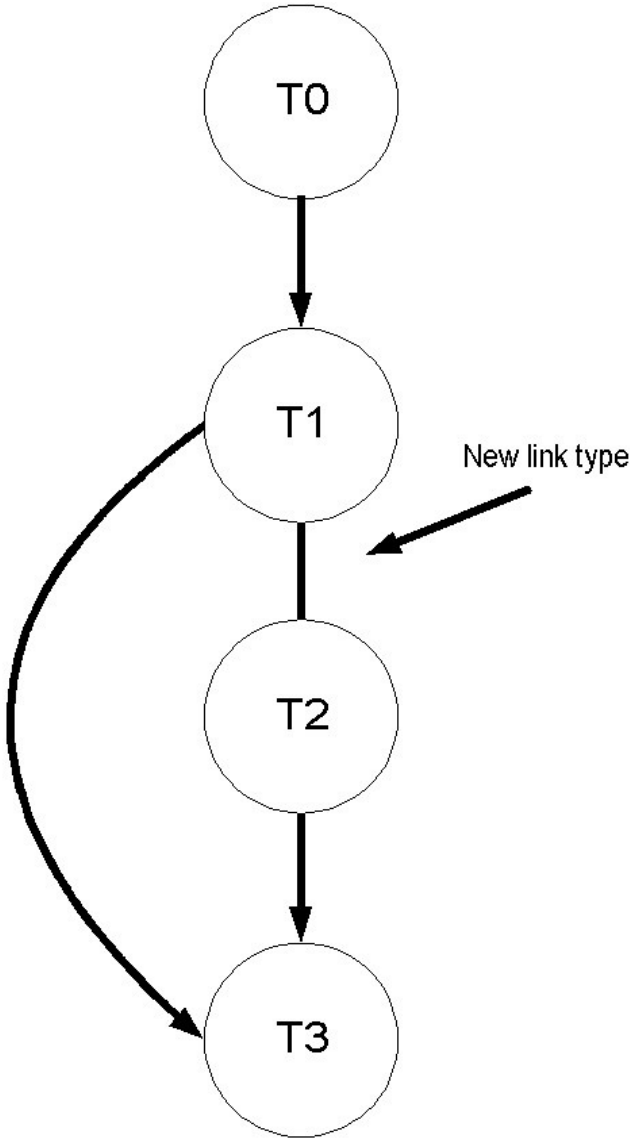
Control Data Flow Graph Example

```
if (cond1) bb1();
 else bb2();
bb3();
switch (test1) {
    case c1: bb4(); break;
    case c2: bb5(); break;
    case c3: bb6(); break;
}
```



DADGP

- Extension of DAG
- New type of link implies no need for data transfer to execute the descendent link.
- Represent variable execution order of tasks T1 and T2



What is **DADGP**

- Directed Acyclic Data dependency Graph with Precedence is an extension of DAG
- DADGP is a super set of DAG
- Two types of edges:
  1) Weighted Dependency edge
  2) Precedence edge

Scheduling for Partitioning

The main input to scheduling for partitioning is a graph representation in the form of DFG **and/or CFG**.

Complex designs contain thousands of both control and data processing operations ranging from:

- Complex arithmetic operations **(multiplication, division)** or logic-level bit-operations.
- All the above interleaved operations by multiple control operations **(if-then-else or case statements)** and loops.

Such designs contain thousands of data-dependencies, basic blocks and control paths.

DFG-based Scheduling & Partitioning

Data-flow based scheduling techniques extract parallelism from the input description (DFG).

- Schedule operations in parallel to satisfy the constraints.
- Two most common DF-based scheduling methods.

  1) List Scheduling (LS): Minimize the number of control steps under resource constraints.

  2) Force-directed Scheduling (FDS): Minimize the number of resource constraints under a fixed number of control steps.

  3) Mixed (FDLS): Force-directed technique is employed as the cost function during list scheduling.

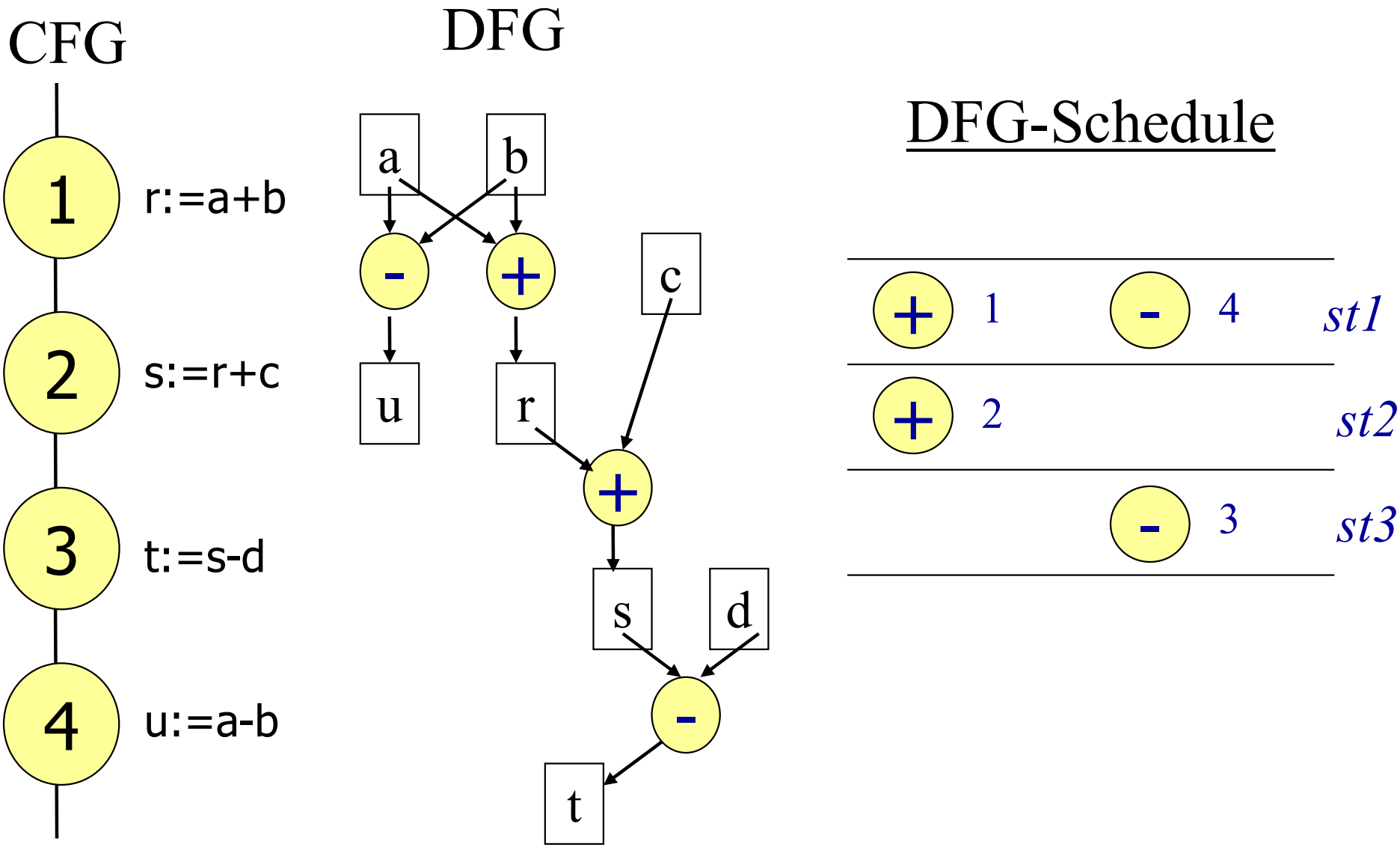
DF-Scheduling

List scheduling algorithm uses a cost function to select the operation to be scheduled from a list.

- DF-approach provides flexible cost-function, and it can be easily adapted to generate resource-constraint as well as time-constraint schedules.
- The cost function can represent any design measure such as HW area, delay, etc.

  The result is only as good as the cost function.
- DF-based algorithms can analyze all the parallelism in the DFG independently.

DF- Scheduling Example

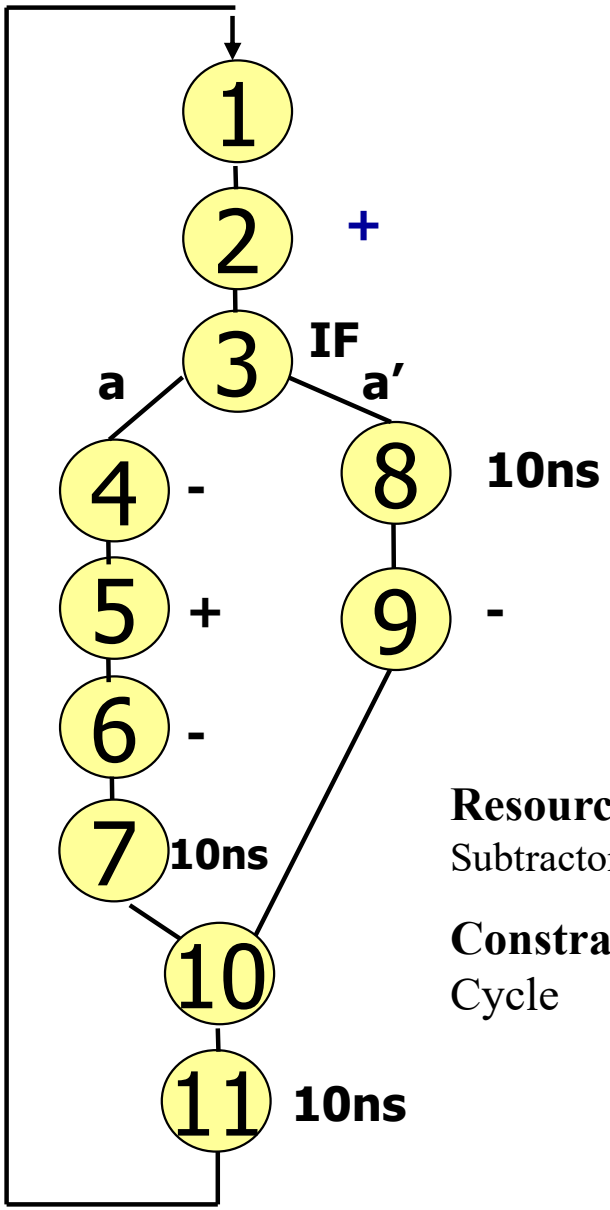


CF-Scheduling

Analyze the sequences of operations in CFG called control flow paths and schedule the CFG with minimum number of control steps in each path.

- Path-based scheduling is one of the main example of this scheme.
- Analyze all the paths in the CFG and schedule each of them independently.
- It minimizes the number of control steps in each path rather than minimizing the number of states.
- Paths in CFG come from loops and conditional operations.

Path-based Scheduling



Path-1 = 1 2 3 4 5 6 7 10 11

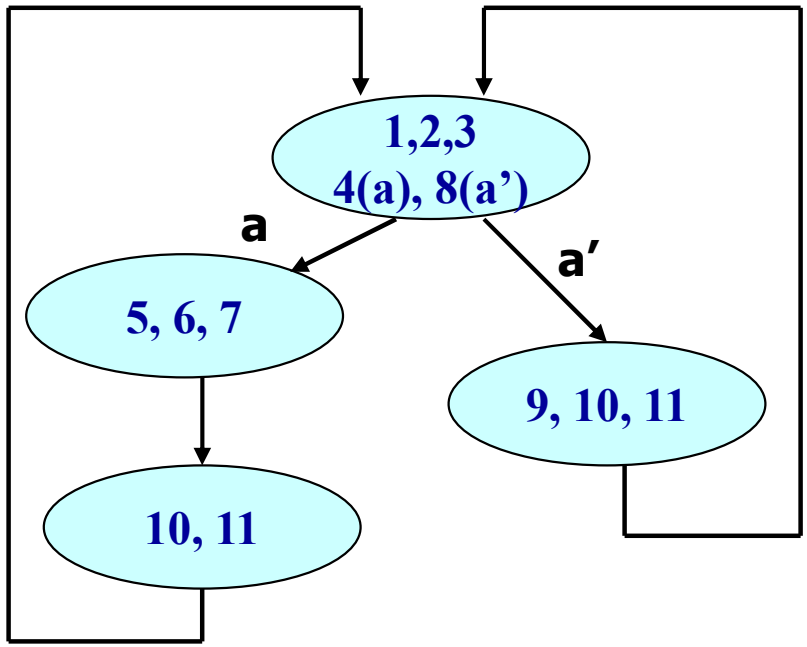
- + - 10 10

Path-2 = 1 2 3 8 9 10 11

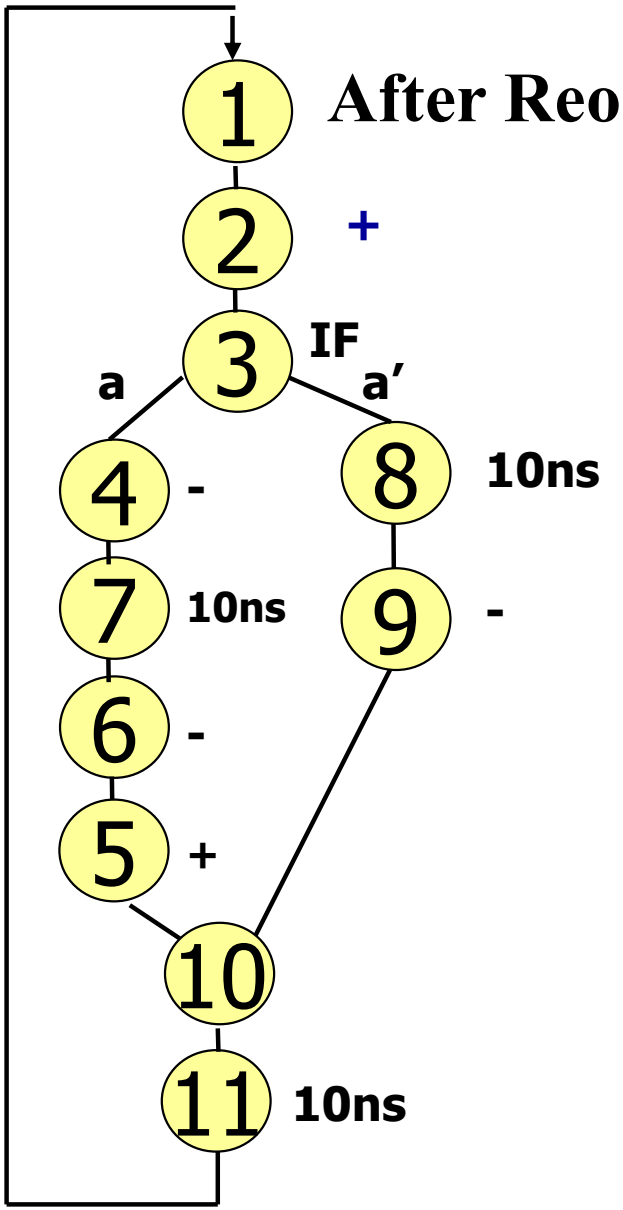
10 10

Resources: One Adder and Subtractor each.

Constraints: 15ns State Cycle



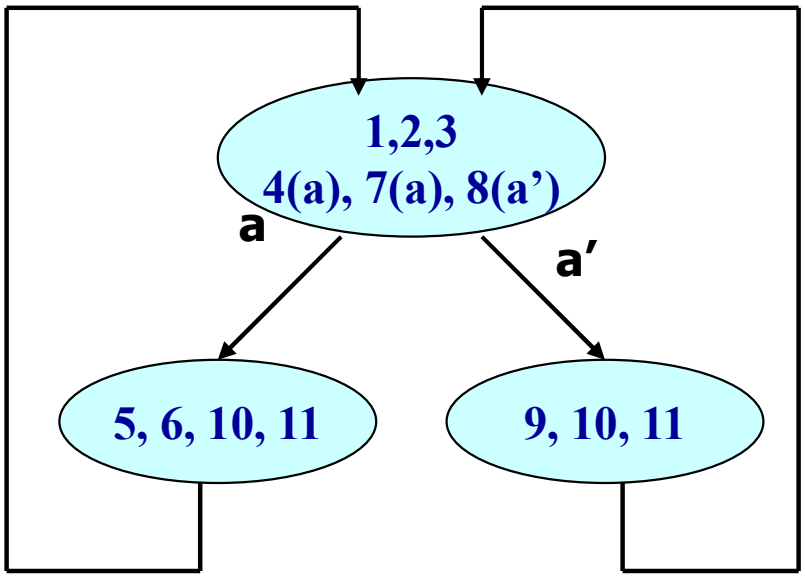
Path-based Scheduling



After Reordering path-1

Path-1 = 1 2 3 4 7 6 5 10 11

+ - 10 - + 10



Partitioning Approaches

Simple one CPU and an ASIC architecture is the most common.

- Early approaches **(mainly heuristic):** Initially assume all tasks mapped to software **(one CPU Hardware)**
- Move tasks to HW incrementally until system requirements **(system or individual task execution time)** are met.
- Other early approaches: Initially all tasks are mapped to dedicated hardware.
- Move tasks incrementally to SW **(CPU)** until system requirements **(system or individual task execution time)** are met.

Optimal Partitioning

- Exhaustive approaches are characterized by attempting all possible combinations there by always selecting the best option.
- Exhaustive approaches are generally computationally intensive, consume huge-time in the range of hours or even days to find an optimal partition.
- Limited to smaller task graphs **(often < 30 nodes)**
  – Large telecom or other embedded systems can have 4000 **or more** nodes

Dynamic Programming

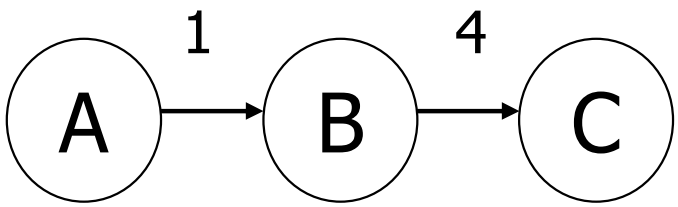
- Recursive, **iterative** algorithm
- Good for problems where calculating all possibilities is computationally infeasible **(good for partitioning!)**
- Problem has to be divided into stages
- Decision required at each stage
- Decisions can alter the current state
- Decisions are independent **(directly)** on past decisions.
- HW/SW Partitioning works well, it can be approached as a recursive, iterative state-based problem.
- Dynamic approaches can yield high quality solutions with very fast run times.

GDL Scheduling for Partitioning

Scheduling is the key part of partitioning process

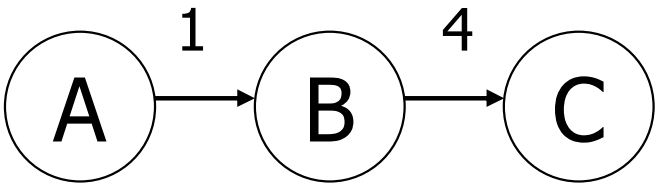
General dynamic level (GDL) scheduling is an extension of typical list scheduling.

- It assigns dynamic priority to nodes and schedule nodes with the highest priority first.
- Dynamic priority assignment is key to GDL scheduling.

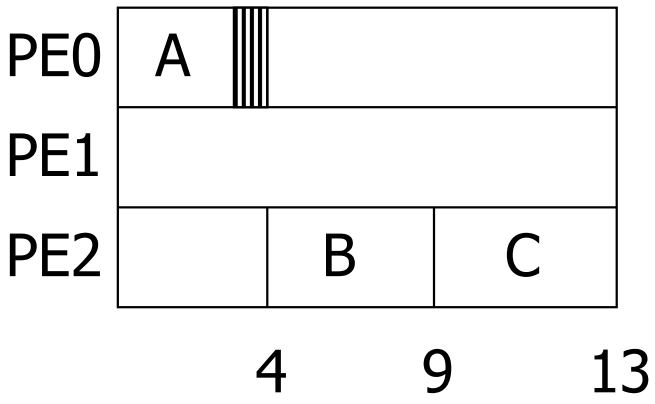


	PE0	PE1	PE2
A	3	6	6
B	5	5	5
C	13	5	4

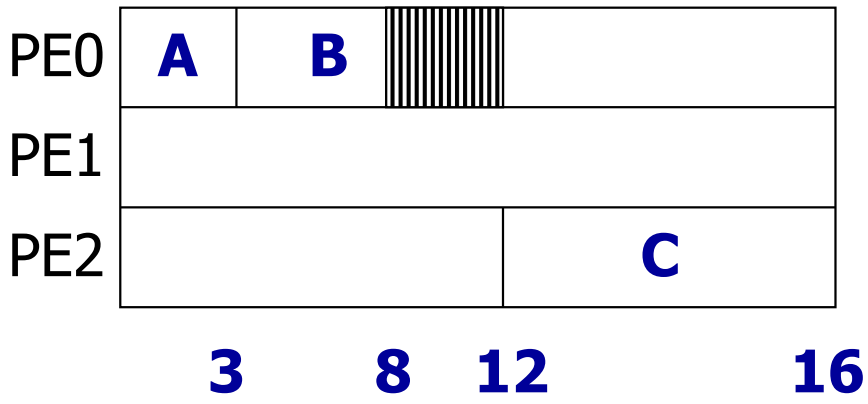
Simple Partitioning Example



	PE0	PE1	PE2
A	3	6	6
B	5	5	5
C	13	5	4

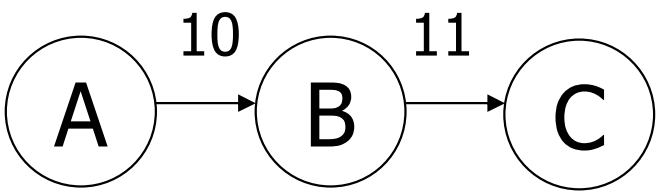


GDL result

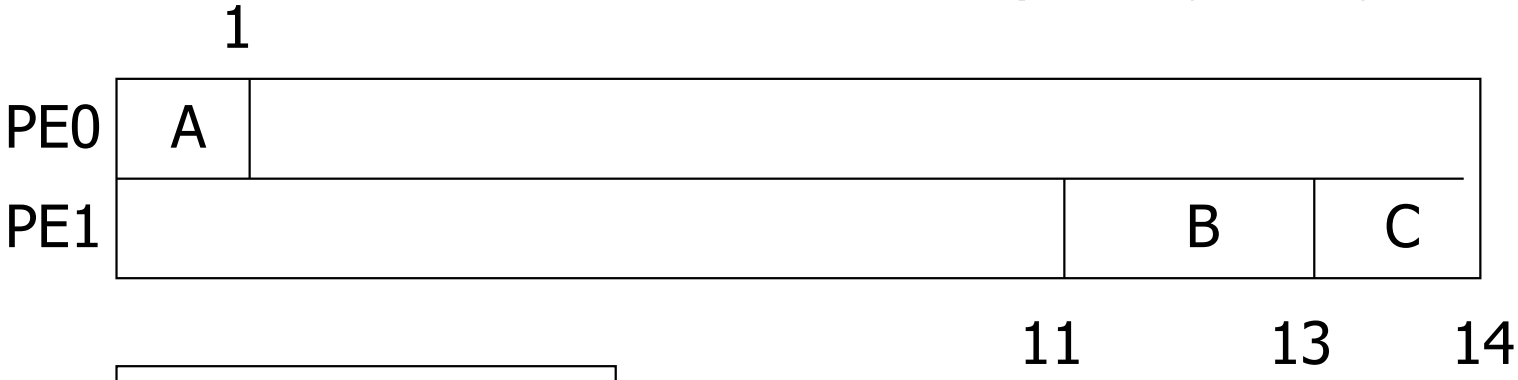


Result of not considering decedents

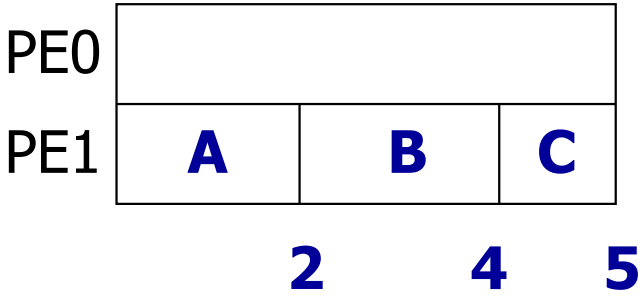
Another Example



	PE0	PE1
A	1	2
B	2	2
C	20	1



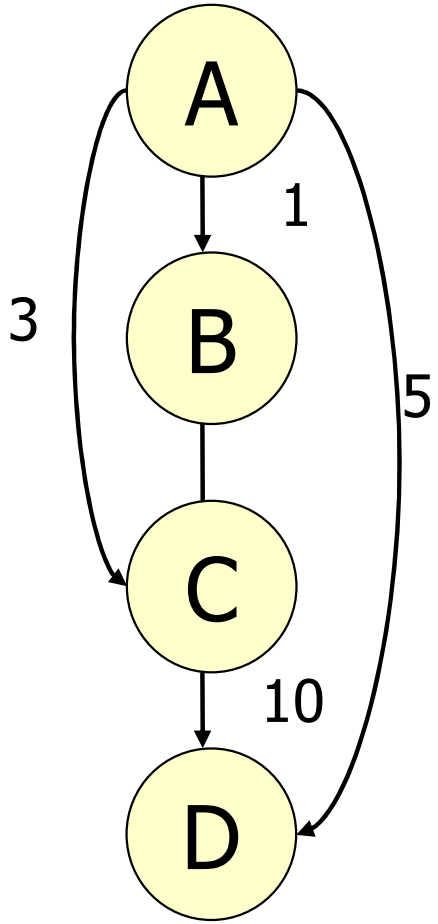
GDL Result



Optimal Solution

DADGP: Directed Acyclic Data Dependency Graph with Precedence

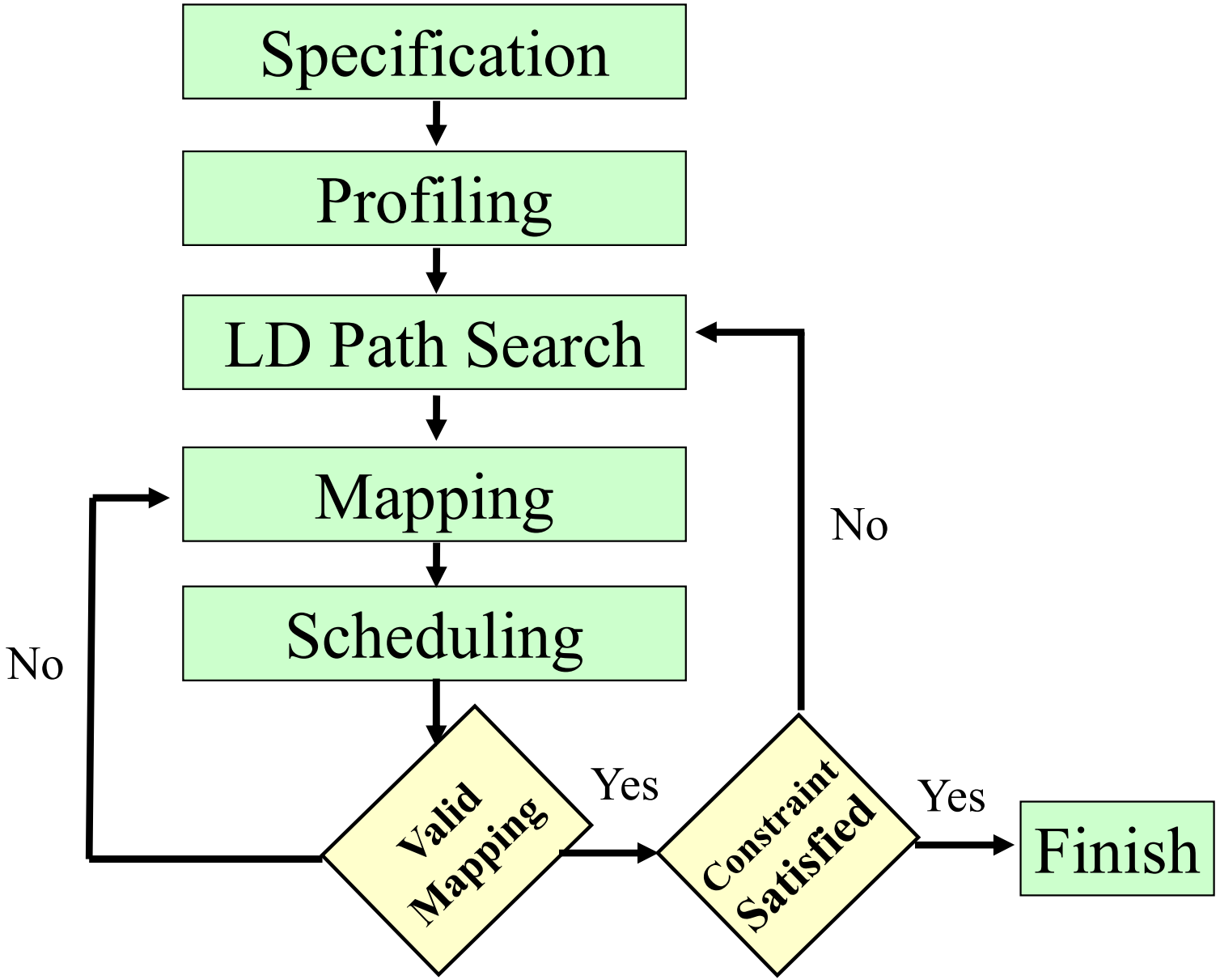
- Arrow represents dependence relationship
- Precedence edge is represented with a line
- Precedence dependency captures the order of execution between nodes and such nodes can be executed in parallel.
- Only necessary parallelism is exposed



Relevant Partitioning Research

- HW-SW Partitioning is a difficult **and NP-hard problem**.
- To find optimal partitioning set, it is very difficult due to many factors affecting the partitioning decision.
- A new partitioning Heuristics are being researched.
- HW/SW Partitioning based on DADGP, Directed Acyclic Data Dependency Graph with Precedence.
- Specified a new task-graph format with less restrictive types of communication links.

DADGP-based Partitioning Structure



DADGP-based Partitioning

i. Profiling and building an initial DADGP
ii. Find the LD_path **(longest delay path)** in DADGP
iii. Mapping of LD-path nodes to hardware
iv. Schedule and if invalid mapping then go to Step iii
v. Update DADGP and calculate the total execution time of target system.
vi. If system constraints **(specified by the user)** are not met then got Step ii, otherwise quit.

Profiling

Profiler collects the following data for each task node (module)

- Hardware/Software execution time
- Hardware Area
- Amount of data transfer
- Execution order
- Data dependencies between nodes

Longest Delay Path Search

Longest Delay path means, longest execution path

- Finding the longest delay path (LD-path) in DADGP is equivalent to finding a bottleneck of the system.

- Minimizes search space for mapping

Mapping

- Maps a node to be implemented as a dedicated hardware unit
- Mapping can change the Longest Delay path, as well as DADGP
- Mapping of a node is valid if implementing that node to Hardware gives the shortest LD-path in the modified DADGP

Scheduling

- Very simple List-based scheduling approach.
- Schedules the earliest node first without violating the resource limit.
- Exposes parallelism and changes the DADGP accordingly.

DADGP-based Scheduling

- Start scheduling from the root of DADGP.
- Traverse down the LD-path tree and schedule the earliest starting time node.
- If the node is connected by a precedence dependency edge, check whether exposing parallelism can eliminate that edge. When an edge is eliminated, DADGP structure may convert to two DADGPs. Roots of the two DADGPs are combined to form a single DADGP with a dummy root node.
- In case of multiple descendants, schedule them forcibly by adding PEs.
- Update the PE resource (HW-SW) library.

Constraints

- Constraints of deadline and cost is given by the system designer.
- Hardware cost is calculated by the gate or transistor count.

  i.e. equivalent to chip or board size.
- Different **granularity** level should be explored if no solution is found.

Varying Granularity

- Task graphs can vary greatly in granularity
- Low-level granularity: each task is a basic operation **(multiply, add, sub, …)**
- High-level granularity: each task is an entire process **(MPEG decode, JPEG encode, . . .)**

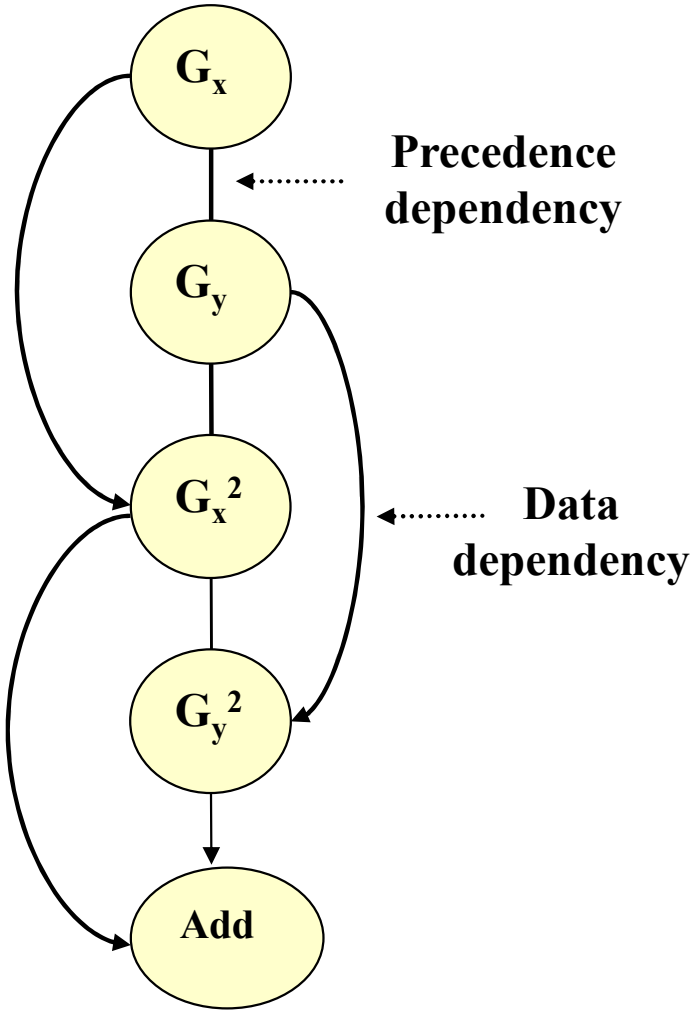
Edge Detection Example

Pair of masks are convolved to estimate gradients, G_x and G_y

Overall $\mathbf{G^2 = (G_x^2 + G_y^2)}$

HW-SW Library

Operation	SW EXE (ms)	HW EXE (ms)	HW Area (gates)
Gradient (Gx or Gy)	9.4	1.4	1200
Square	5.2	0.9	500
Add	3.88	0.3	100



SOBEL Edge Detection

SOBEL masks

-1	0	+1
-2	0	+2
-1	0	+1

Gx

+1	+2	+1
0	0	0
-1	-2	-1

Gy

Input Image

a_{11}	a_{12}	a_{13}		
a_{21}	a_{22}	a_{23}		
a_{31}	a_{32}	a_{33}		

Mask

m_{11}	m_{12}	m_{13}
m_{21}	m_{22}	m_{23}
m_{31}	m_{32}	m_{33}

Output Image

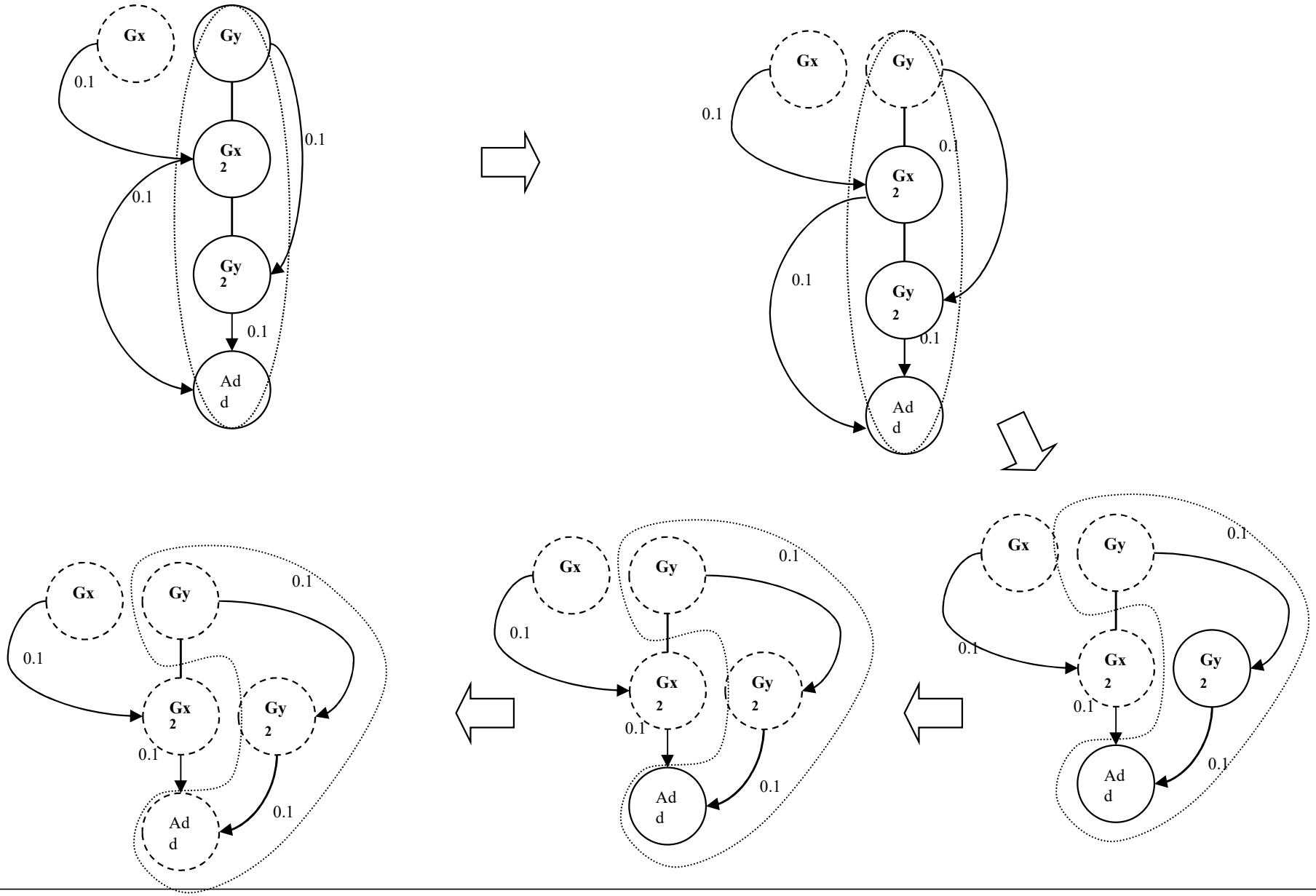
b_{11}	b_{12}	b_{13}		
b_{21}	b_{22}	b_{23}		
b_{31}	b_{32}	b_{33}		

$$b_{22}=(a_{11}*m_{11})+(a_{12}*m_{12})+(a_{13}*m_{13})+(a_{21}*m_{21})+(a_{22}*m_{22})+(a_{23}*m_{23})+(a_{31}*m_{31})+(a_{32}*m_{32})+(a_{33}*m_{33})$$

Sobel Edge Detection

```
main() {
unsigned char image_in[ROWS][COLS];
unsigned char image_out[ROWS][COLS];
int r, c; /* row and column array counters */
int pixel; /* temporary value of pixel */
        /*filter the image and store result in output array */
for (r=1; r<ROWS-1; r++)
 for (c=1; c<COLS-1; c++) {   /* Apply Sobel operator. */
   pixel = image_in[r-1][c+1]-image_in[r-1][c-1]
        + 2*image_in[r][c+1] - 2*image_in[r][c-1]
        + image_in[r+1][c+1] - image_in[r+1][c-1];
   /* Normalize and take absolute value */
   pixel = abs(pixel/4);
   /* Check magnitude */
   if (pixel > Threshold)
   pixel= 255; /*EDGE_VALUE;*/
   /* Store in output array */
   image_out[r][c] = (unsigned char) pixel;
 }
}
```

Edge Detection Solutions



Performance Improvement *vs.* HW area