Design Verification and Test of Digital VLSI Circuits NPTEL Video Course

Module-II

Lecture-I

Introduction to HLS: Scheduling, Allocation and Binding Problem

Introduction

•Any VLSI design we start with specifications and the first step is to obtain the Register Transfer Level (RTL) circuit.

•RTL circuit is obtained from specifications using High Level Synthesis (HLS) algorithms. As specifications are processed by HLS algorithms, they need to be represented using some modeling language.

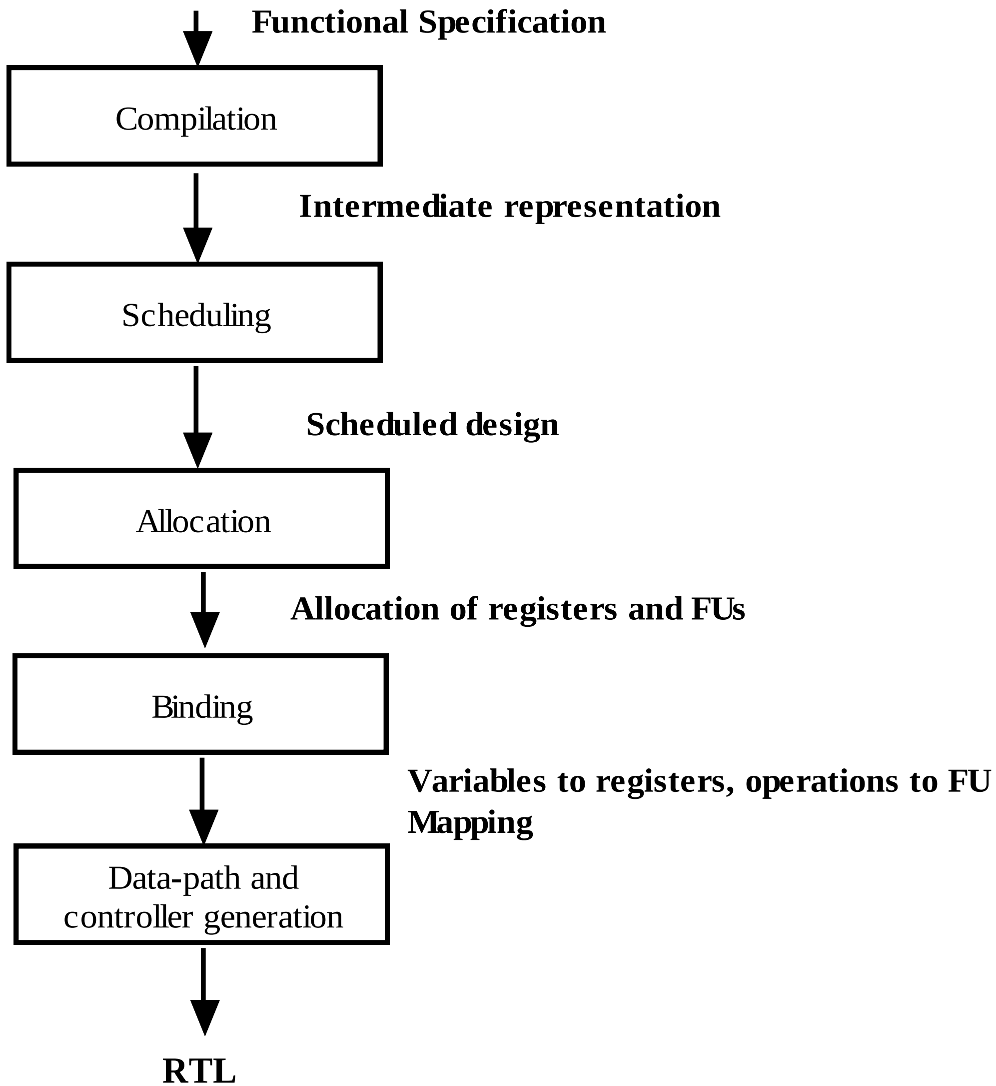
•Control and Data Flow Graph (CDFG), is one of the most widely accepted modeling paradigm for specifications that are processed by HLS tools.

•Transformation techniques in the CDFGs, which lead to efficient circuit implementation in terms of area, frequency, power etc. HLS takes as input, the optimized CDFG, performs Scheduling, Allocation, Binding and generates RTL design.

•In this module we will study algorithms pertaining to these steps--Scheduling, Allocation, and Binding. To start with, in this lecture, we introduce HLS and problem definition of Scheduling, Allocation and Binding.

Introduction to HLS

•A behavioural description (i.e., functional specifications) is used as the starting point for HLS. It specifies the behaviour in terms of operations, assignment statements, and control constructs in a Hardware Description Language (HDL) .



Introduction to HLS

The first step in HLS is compilation of the HDL and transformation into an internal representation.

Most HLS techniques use Control and Data Flow Graph (CDFG) as the representation, because it contains both the data flow and the control flow.

This process also includes a series of compiler like optimizations namely, dead code elimination, redundant expression elimination etc.

Further, it also applies hardware-library specific transformations such as, use of incrementers instead of adders, use of shifters instead of multipliers etc.

It may be noted that in the last module, we have studied these compilation and transformation steps. Sometimes we call these steps as pre-processing phase for HLS, where the optimized CDFG is provided to HLS engine. In some literatures, however, we include these pre-processing steps in the HLS procedure.

Introduction to HLS

The second step of the HLS, which plays a key role in transforming a CDFG (i.e., behavioral) representation into a RTL (i.e., structural) representation, is operation-scheduling (called just “scheduling” in HLS terminology).

Scheduling involves assigning operations of the CDFG to so-called control steps. A control step usually corresponds to a cycle of the system clock, the basic time unit of a synchronous digital system.

The third step is Allocation, which chooses functional units and storage elements from the design library. The design library has several alternatives for a given functional unit or a storage unit. For example, for a functional unit like adder, there can be many options like ripple-carry adder, carry-look-ahead-adder etc. Similarly, for storage elements there can be different types of registers like registers with only resets, registers with both pre-sets and resets, registers with pre-sets, resets and load etc. Among the alternatives, the allocation algorithm must select the one that matches the design constraints best and maximizes the optimization objective.

Introduction to HLS

The fourth step is Binding. After the functional operations and storage operations are scheduled and components from design library are selected for such operations (allocation), then comes the role of binding. Binding assigns operations to functional units, variables to storage elements and data transfers to wires or buses such that data can be correctly computed and passed, according to the scheduling.

The final step of HLS is data-path and controller generation. Depending upon the scheduling and the binding information, interconnection between the circuit modules of the data-path components are set up; this is called data-path generation. Further, an FSM is generated to control all the micro-operations required to control data-flow in the data-path; this is called controller generation.

Scheduling Problem

The scheduling problem involves determining the sequence in which the operations are executed to produce a control step schedule, which specifies the operations that execute in each control step.

Let O be the set of all operations to be scheduled, which are obtained from the HDL code. If there is an operation $o_j \in O$ which depends on the result of another operation an $o_i \in O$, then o_i must finish its execution before operation o_j can begin. In such a case we say that there is a data dependency between the two operations o_i and o_j and o_i is an immediate predecessor of o_j. Data dependency results in a precedence constraint between the two dependant operations in scheduling. In other words, an operator can be scheduled only after all its predecessors are scheduled.

Scheduling Problem

For any HLS platform, there exists a module library comprising circuits for different functionalities like adder, multipliers, registers etc. Further, the library also has information regarding different parameters of the modules namely, frequency, area, power etc. Let T be the set of different types of modules that are available. For a given operation o, the type of the operation is determined by a type function $Ty : O \rightarrow T$; $Ty(o) = t$ implies that operation o can operate on module of type t.

Based on the above basic formulations we will discuss the following four types of scheduling problems

- Un-Constrained Scheduling (UCS) problem
- Time Constrained Scheduling (TCS) problem
- Resource Constrained Scheduling (RCS) problem
- Time-Resource Constrained Scheduling (TRCS) problem

Now we elaborate on each of these types using the simple example expression "(a+b+c+d)*e".

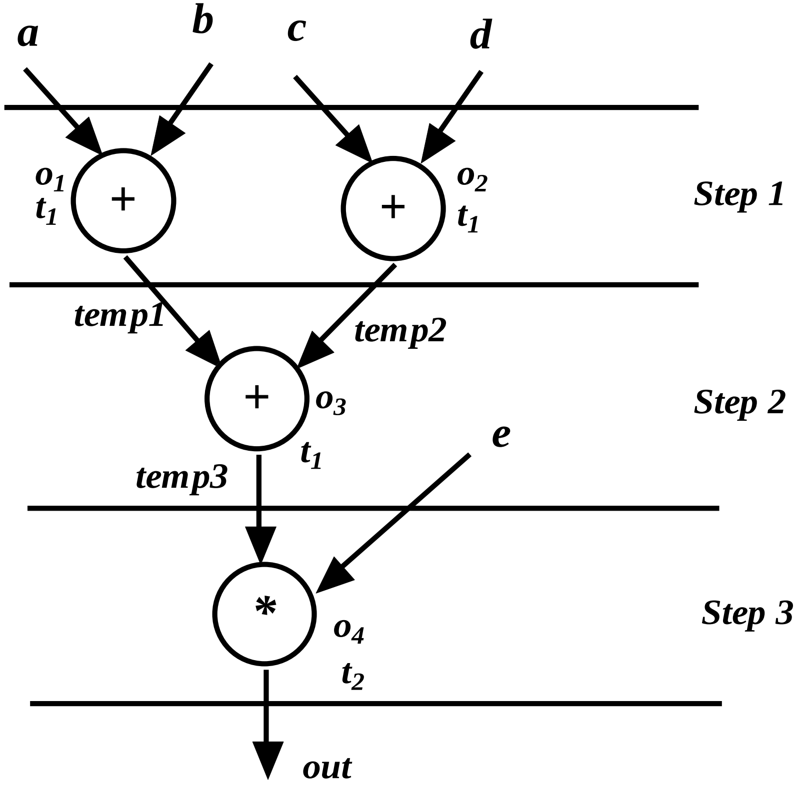
Unconstrained Scheduling (UCS) problem

Given:

A set of operations O, a set T of different types of functional modules, a type function $Ty: O \rightarrow T$ and a partial order on O determined by the precedence constraints.

Find:

A feasible schedule for all elements in O, taking appropriate modules from T and obeying the partial order.



Unconstrained Scheduling (UCS) problem

As the schedule is unconstrained we need to see that all elements in O are scheduled, appropriate modules from T are taken and partial order is maintained. In the above example, there are four operations (3 additions denoted as o_1, o_2, o_3 and 1 multiplication denoted as o_4), all of which are scheduled. Let the library have two types of resources, adders (denoted as t_1) and multipliers (denoted as t_2). It may be noted that appropriate modules from T are taken-- o_1, o_2, o_3 are assigned to t_1 (i.e., adder is assigned to addition operations) and o_4 are assigned to t_2 (i.e., multiplier is assigned to multiplication operation).

As the scheduling is unconstrained, we consider two adder modules (one for o_1 and the other for o_2) and a multiplier module (for o_4). The adder module for o_1 can be reused for o_3. The control steps required is 3.

Time Constrained Scheduling (TCS) problem

Given:

A set of operations O, a set T of different types of functional modules, a type function $Ty: O \rightarrow T$, a time constraint (deadline) D (i.e., maximum control steps) and a partial order on O determined by the precedence constraints.

Find:

A feasible schedule for all elements in O, taking appropriate modules from T, meeting the deadline D and obeying the partial order.

It may be noted that schedule of last example satisfies all requirements of unconstraint scheduling problem and along with that, it satisfies the three steps deadline (of timing constrained problem). Further, we may note that we cannot have a successful schedule if timing constraint is two control steps, as it will lead to violation of partial order. The time-constrained scheduling required two adders (for o_1, o_2, which is reused for o_3) and a multiplier (for o_4) .

Resource Constrained Scheduling (TCS) problem

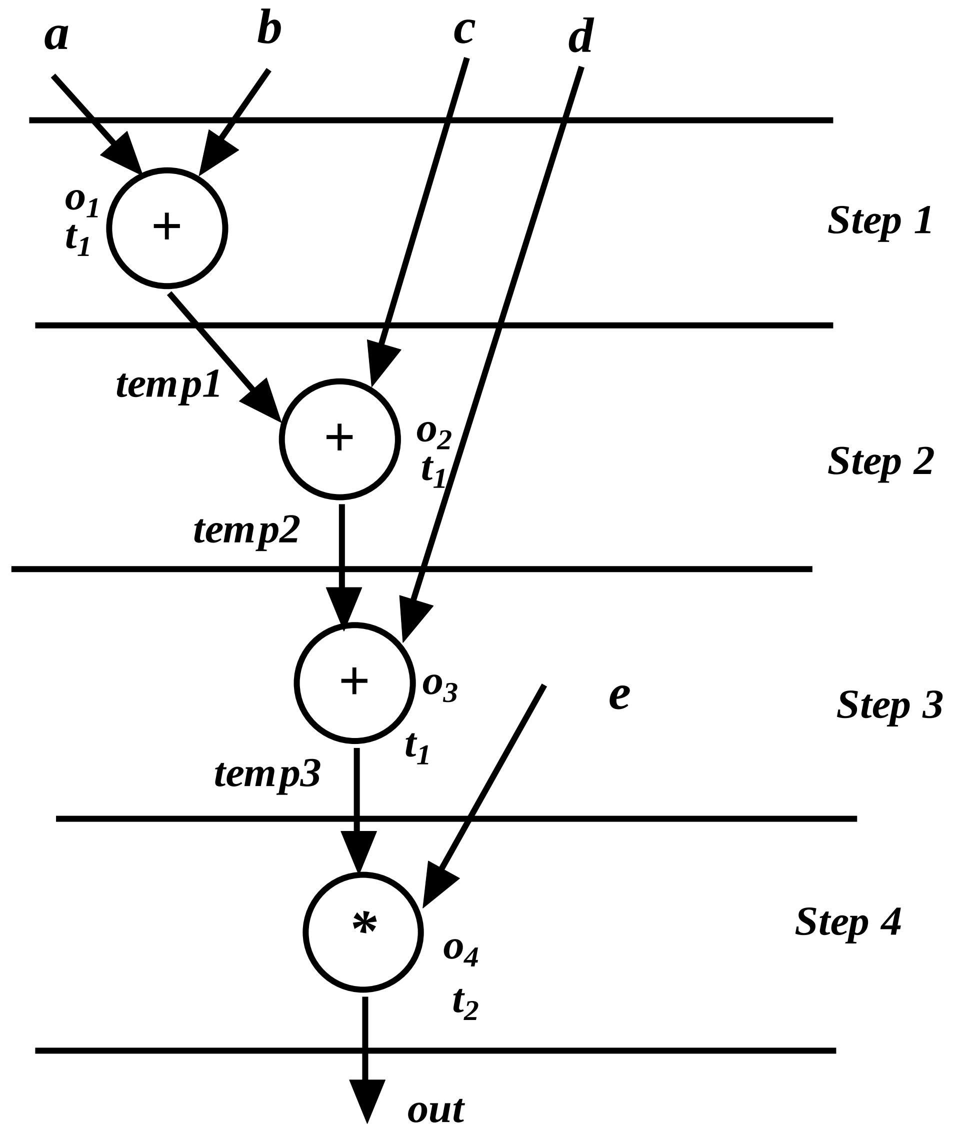
Given:

A set of operations O, a set T of different types of functional modules, a type function $Ty: O \rightarrow T$, resource constraints $\max_k, 1 \le k \le |T|$ for each functional module of type $t_k, 1 \le k \le |T|$ and a partial order on O determined by the precedence constraints.

Find:

A feasible schedule for all elements in O, taking appropriate modules from T, meeting the resource constraints for each functional module type and obeying the partial order.

Resource Constrained Scheduling (TCS) problem



Resource Constrained Scheduling (TCS) problem

Illustrates a resource-constrained scheduling involving, one adder and one multiplier, for expression (a+b+c+d)*e.

As the schedule is resource-constrained we need to see that all elements in O are scheduled, appropriate modules from T are taken, partial order is maintained and recourse utilization does not cross the limit.

As there is one adder module (for o_1, o_2, o_3) and a multiplier module (for o_4), we cannot schedule o_1 and o_2 in one control step. So we schedule o_1 is step1 and o_2 in step2. To maintain the partial order, o_3 is scheduled in step3 and o_4 is scheduled in step4; it may be noted that these operators cannot be scheduled earlier.

Therefore, the number of control steps is 4. Due to meeting the resource constraint, we cannot have a schedule in 3 steps

Time Resource Constrained Scheduling (TCS) problem

Given:

A set of operations O, a set T of different types of functional modules, a type function $Ty : O \rightarrow T$, a time constraint (deadline) D, resource constraints $\max_k, 1 \le k \le |T|$ for each functional module of type $t_k, 1 \le k \le |T|$ and a partial order on O determined by the precedence constraints.

Find:

A feasible schedule for all elements in O, taking appropriate modules from T, meeting the deadline D, meeting the resource constraints for each functional module type and obeying the partial order.

In time resource constraint scheduling, we need to meet both timing and resource constraints.

Allocation Problem

•Once a schedule is made (i.e., type of operators are determined along with their quantity), the allocation task determines the *"exact"* operator modules, available in the design library, to be used in implementation of the operators. Also, the area, power, frequency is determined after allocation.

•A typical design library can be represented as a table given below. It has description regarding the type of modules (i.e., functionality), sub-types (namely, fast, slow, typical etc.), area, power, frequency etc. In case of a modern sub-micron technology, a design library has many more entries namely, leakage power, current etc.

Allocation Problem

Sl. No	Name of Module	Type	Sub-type	Frequency	Area	Power
1	Adder-slow	t_1	$t_1 - S$	F_{t_1-S}	A_{t_1-S}	P_{t_1-S}
2	Adder-fast	t_1	$t_1 - F$	F_{t_1-F}	A_{t_1-F}	P_{t_1-F}
3	Multiplier-slow	t_2	$t_2 - S$	F_{t_2-S}	A_{t_2-S}	P_{t_2-S}
4	Multiplier-fast	t_2	$t_2 - F$	F_{t_2-F}	A_{t_2-F}	P_{t_2-F}

It may be noted that a fast module has higher frequency, higher area and higher power compared to its shower counterpart; so $F_{t_1-S} < F_{t_1-F}$, $A_{t_1-S} < A_{t_1-F}$, $P_{t_1-S} < P_{t_1-F}$ and $F_{t_2-S} < F_{t_2-F}$, $A_{t_2-S} < A_{t_2-F}$, $P_{t_2-S} < P_{t_2-F}$.

Allocation Problem

Let us consider the unconstrained schedule of the expression (a+b+c+d)*e, From the output of scheduling we know that o_1, o_2, o_3 are of type t_1 and o_4 is of t_2. Further, we need two modules of type t_1 and one module of type t_2.

Now, depending on requirement of frequency and available area-power overheads, we can select the sub-types for t_1 and t_2. If we have high area and power constraints, then we would use $t_1 - S$ for t_1 and $t_2 - S$ for t_2.

It may be noted that time period of each control step is dependent on module having the lowest frequency because system clock frequency depends on the critical path. In general, a multiplier has much higher area and power requirements compared to an adder. Also, frequency of a multiplier is lower compared to an adder.

Allocation Problem

So, in the example, time period of each control step be $\frac{1}{F_{t_2-S}}$.

Now, if have no area and power constraints, then we would use t_1-F for t_1 and t_2-F for t_2. The time period of each control step is $\frac{1}{F_{t_2-F}}$.

But, in general $F_{t_2-F} < F_{t_1-S}$; frequency of a fast multiplier is generally less compared to even a slow adder. So in spite of allocating fast adders to o_1, o_2, o_3 (consuming high area and power), time period of control step is $\frac{1}{F_{t_2-F}}$, which is not dependent on F_{t_1-S} or F_{t_1-F}. So we can use slow adders without any compromise in overall time period of operation (i.e., time period of control step).

Binding

After all the operations are scheduled and allocation is done, we get information regarding exact type of circuit modules (from the design library) to be used and their numbers.

We have seen in the allocation step, that operations in a control step are performed by different modules, however, modules are shared between operations (of same type) that are in different control steps. In the unconstrained schedule example, an adder module will be shared between o_1 and o_3 or o_2 and o_3. Due to sharing, in addition to operational modules (adders, multipliers etc.), we need multiplexers.

Further, to store variables (*a*,*b*,*c*,*d*,*e*) and intermediate results *(temp1*,*temp2*.*temp3*) we need registers. Like operational modules, registers can be shared, which do not lie in same control step.

All the above-mentioned steps (after scheduling and allocation) fall under Binding.

Binding

The binding task (also called resource-sharing step) assigns the operations and variables to hardware modules. A resource such as an operational module or register can be shared by different operations, data accesses, or data transfers if they are mutually exclusive. For example, two operations assigned to two different control steps are mutually exclusive since they will never execute simultaneously; hence, they can be binded to the same hardware unit. Binding can be classified into three sub-functions:

Storage binding: This step assigns input, output and temporary variables to registers units. Two variables that are not alive simultaneously (i.e., not required in overlapping control steps) in a given control step can be assigned to the same register.

Functional-unit binding: This binding step assigns operations to operational modules (like adder, multiplier etc.). Two operations of same type that are not in a single control step can be assigned to the same operational module.

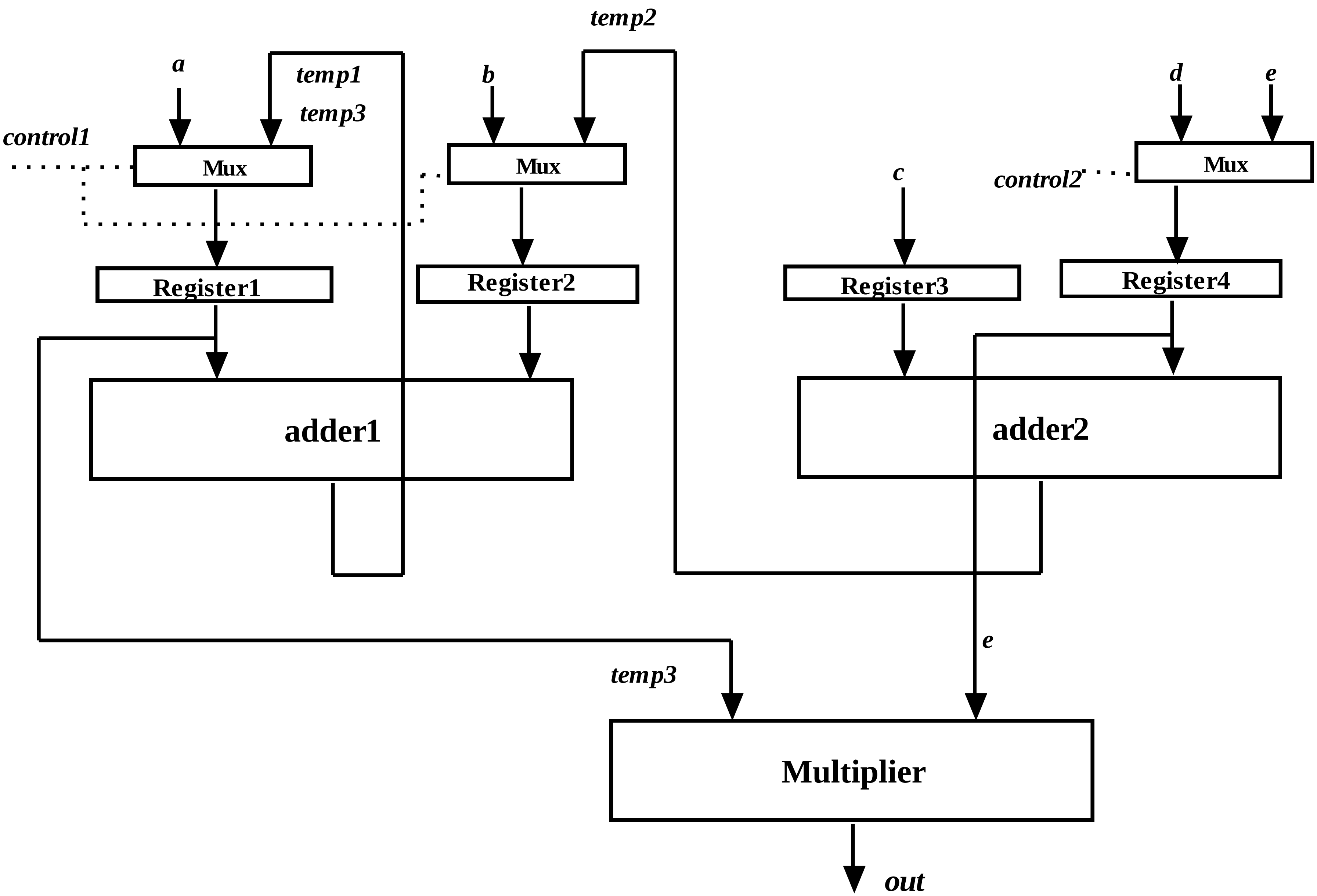
Interconnection binding: This step assigns an interconnection unit such as a multiplexer or a bus to a data transfer.

Binding

Although listed separately, the three sub-functions are intertwined and are to be carried out concurrently for optimal results.

Now, we will illustrate Binding for the unconstrained schedule when allocation is-- two number of modules of type $t_1 - S$ for o_1, o_2, o_2 and one module of type $t_2 - F$ for o_4.

Binding


temp2
a
temp1
temp3
b
d
e
control1
Mux
Mux
c
control2
Mux
Register1
Register2
Register3
Register4
adder1
adder2
e
temp3
Multiplier
out

Binding

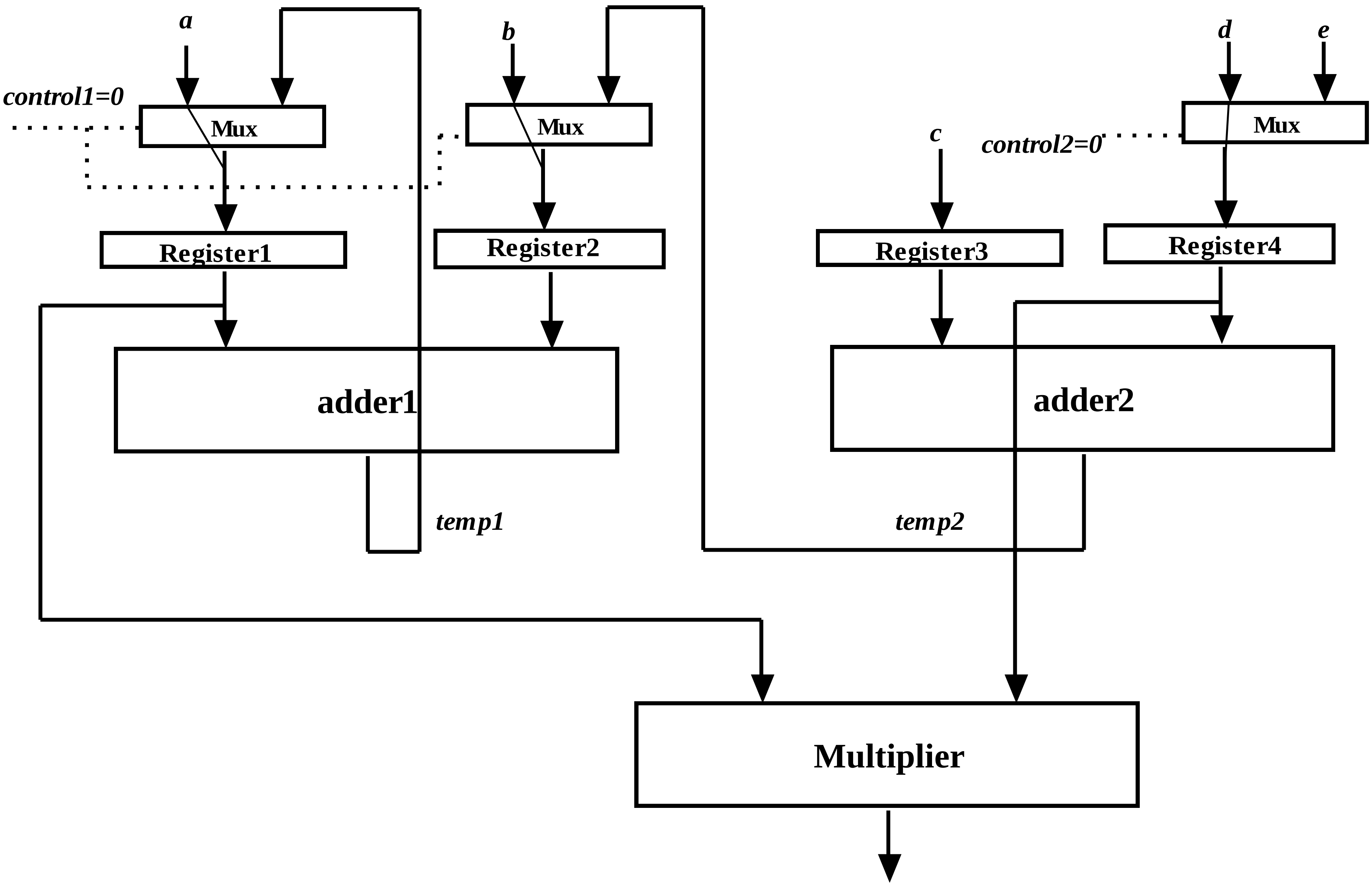
At control step1, we have 4 active variables (*a*,*b*,*c*,*d*), at step2 we have 2 active variables (*temp1*,*temp2*) and at step3 we have 2 active variables (*temp3*,*e*).

So we have a maximum of 4 active variables at step1, thereby leading to the fact that we required 4 registers; *a*,*b*,*c*,*d* cannot share any register. However, registers can be shared between (*a*,*b*,*c*,*d*) and (*temp3*,*e*); (*a*,*b*,*c*,*d*) and (*temp1*,*temp2*); (*temp1*,*temp2*) and (*temp3*,*e*). However, variables listed in the brackets cannot share registers among themselves. As discussed in last section, we have two adder modules and one multiplier module. Based on these facts a possible binding is as follows

Binding

- Binding of o_1 to adder1 and o_2 to adder2 (functional unit binding)
- Binding of o_3 to adder2 (functional unit binding)
- Binding of *a*,*temp1*,*temp3* to register1 (storage binding)
- Binding of *b*,*temp2* to register2 (storage binding)
- Binding of *c* to register3 (storage binding)
- Binding of *d*,*e* to register4 (storage binding)
- Binding of o_4 to multiplier1 (functional unit binding)

Binding-Configuration at Control step1

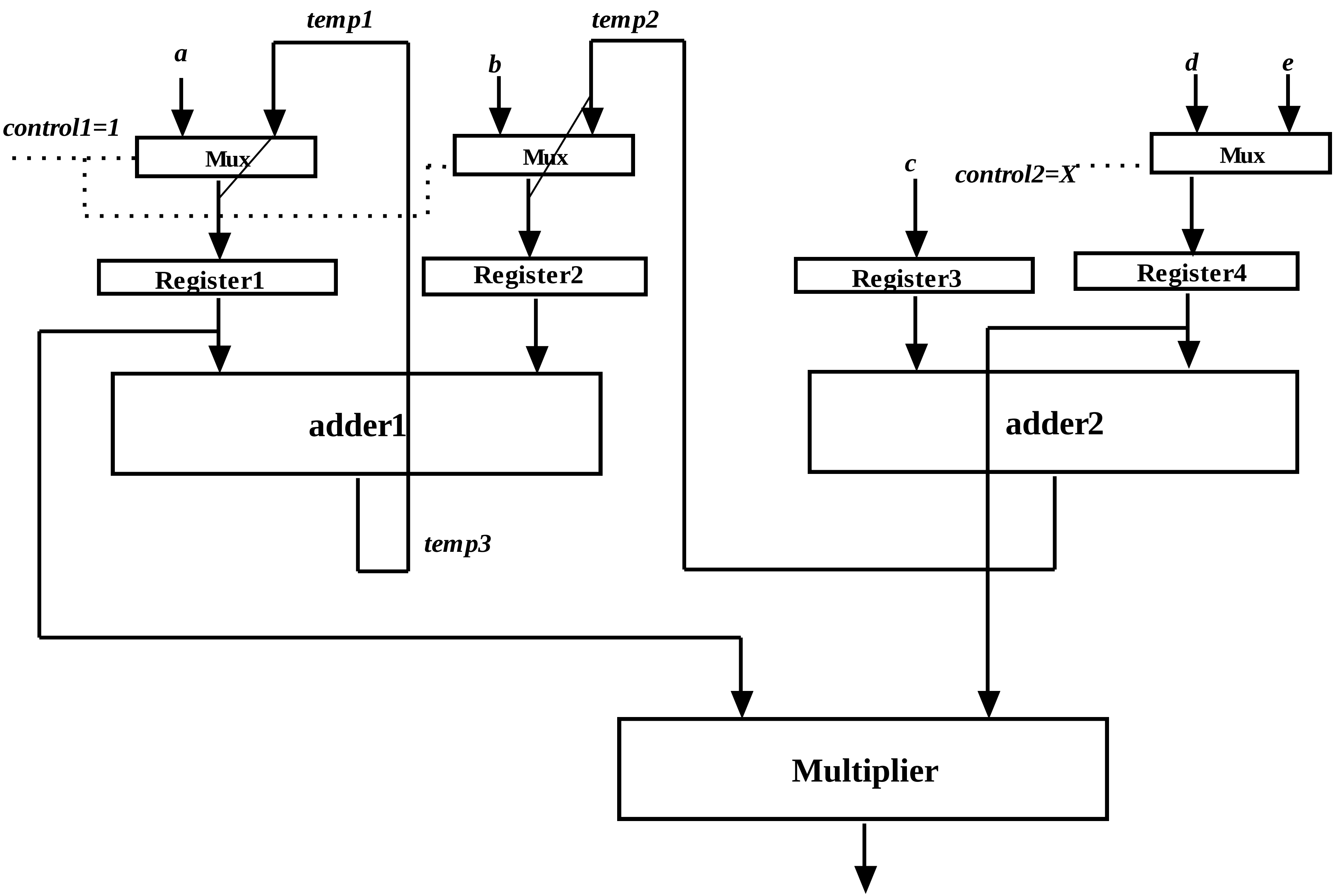


Binding-Configuration at Control step1

- *control1* is 0, thereby binding *a* in register1 and *b* in register2
- *control2* is 0, thereby binding *d* in register4
- Binding *c* to register3
- Binding of o_1 to adder1
- Binding of o_2 to adder2

Under this binding, adder1 generates *temp1* and adder2 generates *temp2*.

Binding-Configuration at Control step2

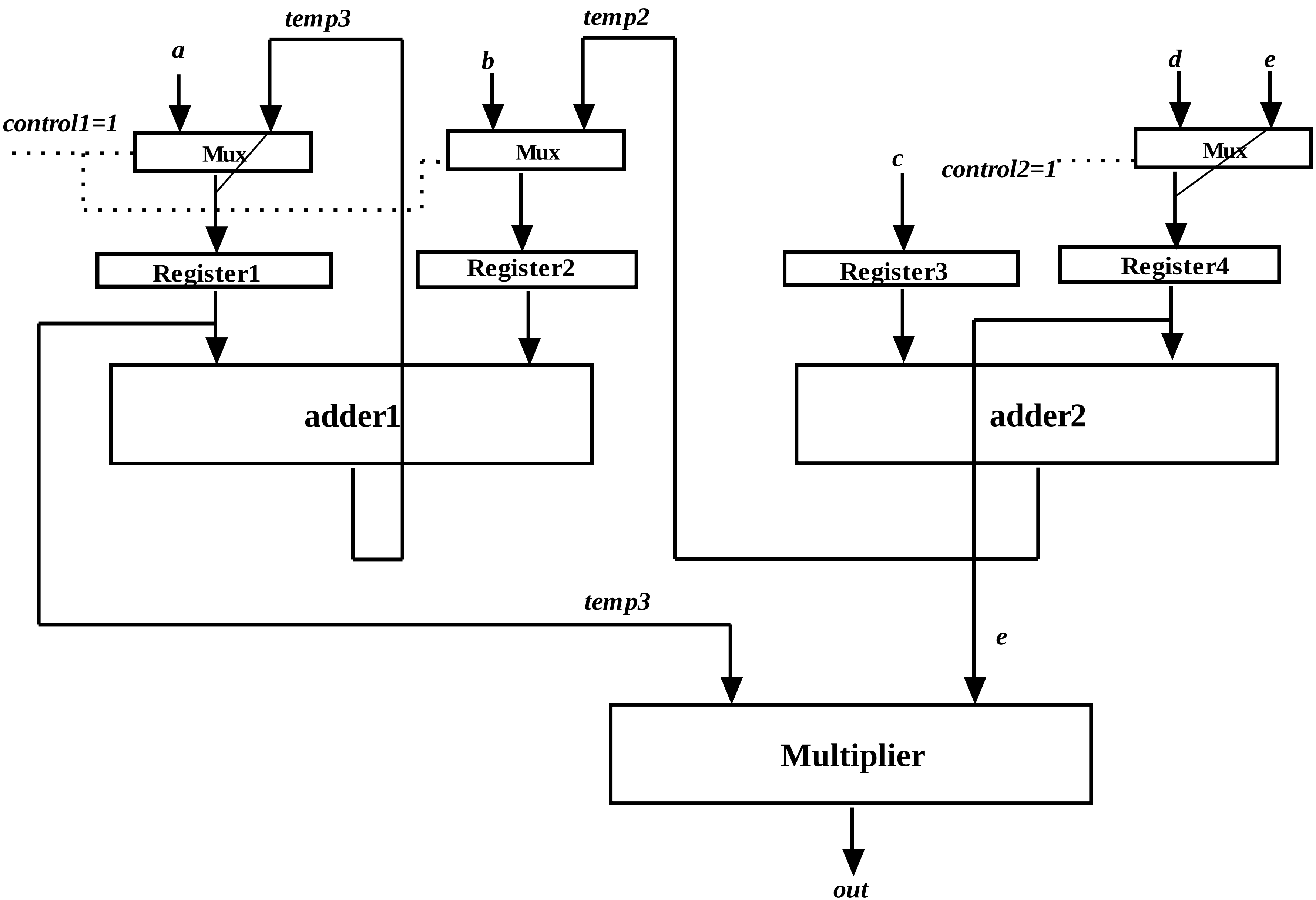


Binding-Configuration at Control step2

- *control1* is 1, thereby binding *temp1* in register1 and *temp2* in register2
- *control2* is X and adder2 is not used. In addition, register3 and register4 are not used.
- Binding of o_3 to adder1

Under this binding, adder1 generates *temp3*

Binding-Configuration at Control step3



Binding-Configuration at Control step3

- *control1* is 1, thereby binding *temp3* in register1; register2 is not used
- *control2* is 1, hereby binding *e* in register4. Register3 is not used.
- Binding of o_4 to multiplier1

Under this binding, multiplier1 generates *out*.

Control Path

For the scheduling, allocation and binding considered in the running example we have the following signal sequences for *control1* and *control2* in the three time steps.

- Step-1: *control1* is 0 and *control2* is 0
- Step-2: *control1* is 1 and *control2* is X
- Step-3: *control1* is 1 and *control2* is 1

We need to develop a sequential circuit having two output bits "*control1*" and"*control2*" and they should have the values "00", "1X" and "11" in three consecutive clock edges. This circuit can be easily design using the concept of state machine implementation

Question and Answer

Question: Among the three sub-steps of HLS, scheduling, allocation and binding, what can be done without information regarding design-library?

Answer: Scheduling and Binding can be done without information regarding design-library. Scheduling assigns control steps to all operations in the CDFG, after satisfying data-dependency between the operations, subject constraints like number of steps, number of modules etc. So none of the parameters are related to design-library. In case of Binding, operations and variables are attached to circuit modules, which are selected from the design library during the allocation phase. As circuit modules are already selected from the design library during the allocation phase, binding can work without any information from the design library.

Design Verification and Test of Digital VLSI Circuits NPTEL Video Course

Module-II

Lecture-II and III

Scheduling Algorithms

Introduction

High Level Synthesis (HLS) involves three sub-parts namely, scheduling, allocation and binding.

In this lecture, we will discuss scheduling algorithms, which automatically assign control steps to operations subject to design constraints.

Scheduling problem can be of four types namely, unconstrained, time constrained, resource constrained and time-resource constrained.

Introduction

•There are many algorithms proposed in the literature that solve these four types of scheduling problem.

•Now, these algorithms can be classified into two types as heuristics and exact. Exact algorithms like Integer Liner Programming for scheduling, provides optimal schedule but consumes high processing time.

•In practical cases, these exact algorithms for HLS take prohibitive amount of execution time. To cater to the execution time issue, several algorithms based on greedy strategies have been developed that make a series of local decisions, selecting at each point the single “best” operation-control step pairing without backtracking or look-ahead. So they may miss the globally optimal solution, however, they do produce results quickly, and those results are generally be sufficiently close to optimal to be acceptable in practice. Such algorithms are called heuristic algorithms (for HLS). Examples for heuristic algorithms for HLS comprise As Soon As Possible (ASAP), As Late As Possible (ALAP), List Scheduling (LS) and Force Directed Scheduling (FDS).

As Soon As Possible Scheduling

As-Soon-As-Possible (ASAP) scheduling is one of the simplest scheduling algorithms used in HLS.

In ASAP scheduling, first the maximum number of control steps that are allowed is determined.

Following that, the algorithm schedules each operation, one at a time, into the earliest possible control step.

In other words, ASAP algorithm schedules operations in the earliest possible control step, subject to satisfying the partial order, i.e., an operation is scheduled if and only if all its predecessors are scheduled in earlier control steps.

If ASAP algorithm can schedule all the operations within the allowed number of control steps, scheduling is successful.

It may be noted that ASAP algorithm does not consider any resource constraints.

As Soon As Possible Scheduling

Algorithm 1: As Soon As possible

Input: Operations O, Maximum number of control steps M.

Output: Control step for each operations, Status of scheduling .

Steps

for each operation $o_i \in O$

DO

if o_i has no immediate predecessors (i.e., computation from inputs)

control_step(o_i) = 1. /* control_step(o_i) indicates control step into which operation o_i is scheduled */

else

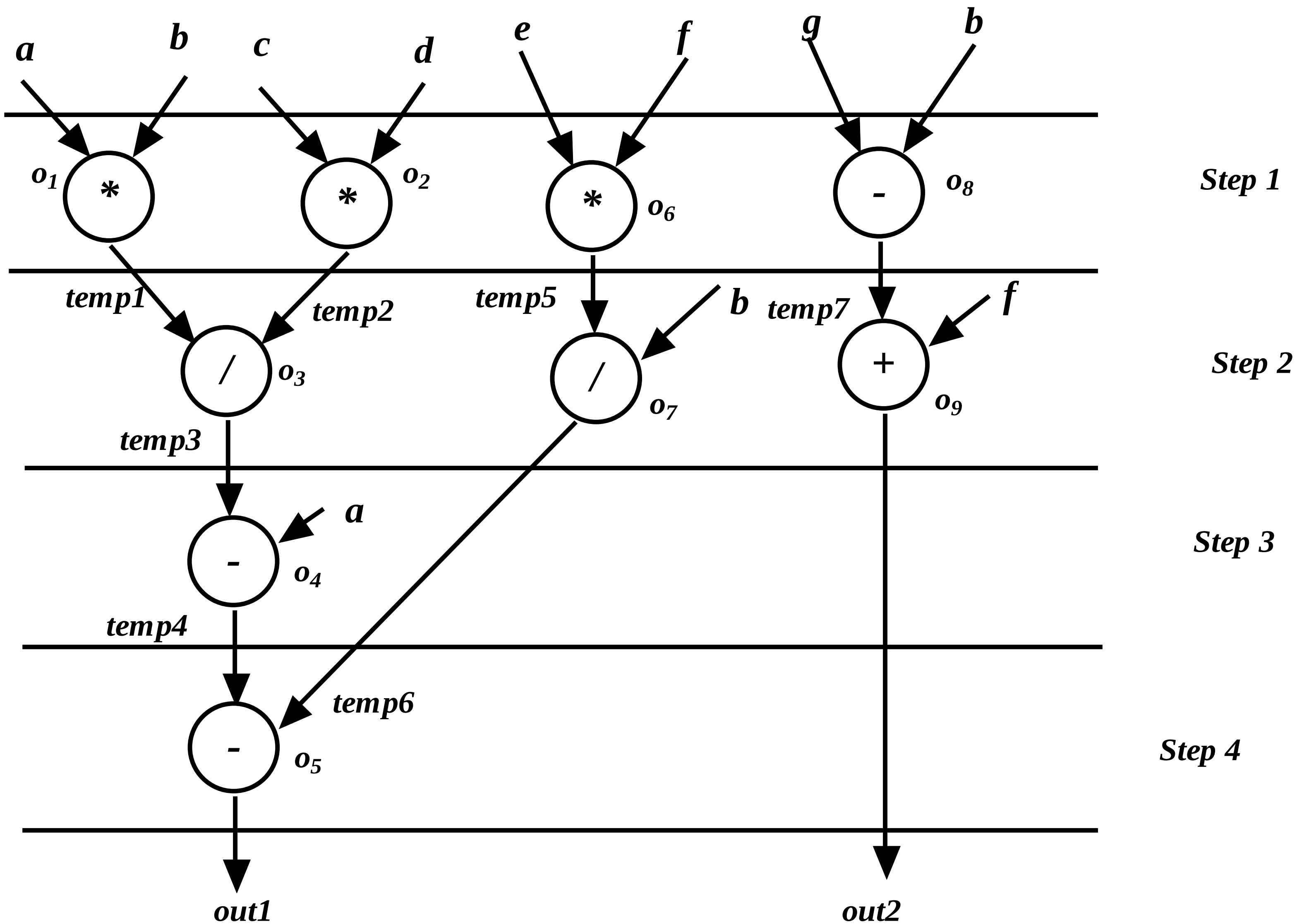
control_step(o_i) = maximum(control_step(o_j))+ 1,where

$o_j \in \{o \mid o$ is immediate predecessor of $o_i\}$.

END

If value of control_step(o_i), $\leq M, o_i \in O$ then Status of scheduling is Successful.

As Soon As Possible Scheduling



ASAP scheduling for “out1=((a*b)/(c*d))-a-((e*f)/b)” and “out2=(g-b)+f”

As Soon As Possible Scheduling

In this case, it may be noted that operations o_1, o_2, o_6, o_8 do not have any direct predecessors, i.e., they depend on input values. So these operations have the control step as 1 (control_step(o_i)=1, $i = 1,2,6,8$). Operation o_3 has o_1, o_2 as predecessors, so, control_step(o_3) = maximum (control_step(o_1),control_step(o_2))+1=2. Similarly, control step assignment for all operations can be explained.

This schedule is complete within 4 steps, thereby making it successful. The resource requirements are—

- Step1: 3 Multipliers + 1 Subtractor
- Step2: 2 Dividers + 1 Adder
- Step3: NIL (subtractor from Step1 can be used)
- Step4: NIL (subtractor from Step1 can be used)

As Late As Possible Scheduling

•As-Late-As-Possible (ALAP) scheduling is almost similar to ASAP, but instead of scheduling operations to early control steps, in ALSP, first the maximum number of control steps that are allowed is determined.

•Following that, the algorithm schedules each operation, one at a time, into the latest possible control step. In other words, ALAP algorithm schedules operations in the latest possible control step, subject to satisfying the (reverse) partial order, i.e., an operation is scheduled if and only if all its successors are scheduled in latter control steps.

•If ALAP algorithm can schedule all the operations within 1st control step (as we move backward), scheduling is successful. It may be noted that like ASAP, ALAP algorithm also does not consider any resource constraints.

As Late As Possible Scheduling

•As-Late-As-Possible (ALAP) scheduling is almost similar to ASAP, but instead of scheduling operations to early control steps, in ALSP, first the maximum number of control steps that are allowed is determined.

•Following that, the algorithm schedules each operation, one at a time, into the latest possible control step. In other words, ALAP algorithm schedules operations in the latest possible control step, subject to satisfying the (reverse) partial order, i.e., an operation is scheduled if and only if all its successors are scheduled in latter control steps.

•If ALAP algorithm can schedule all the operations within 1st control step (as we move backward), scheduling is successful. It may be noted that like ASAP, ALAP algorithm also does not consider any resource constraints.

As Late As Possible Scheduling

Algorithm 2: As Late As possible

Input: Operations O, Maximum number of control steps M .

Output: Control step for each operations, Status of scheduling .

Steps

for each operation $o_i \in O$

DO

if o_i has no immediate successors (i.e., computation generates outputs)

control_step(o_i) = M . /* control_step(o_i) is assigned the last control step */

else

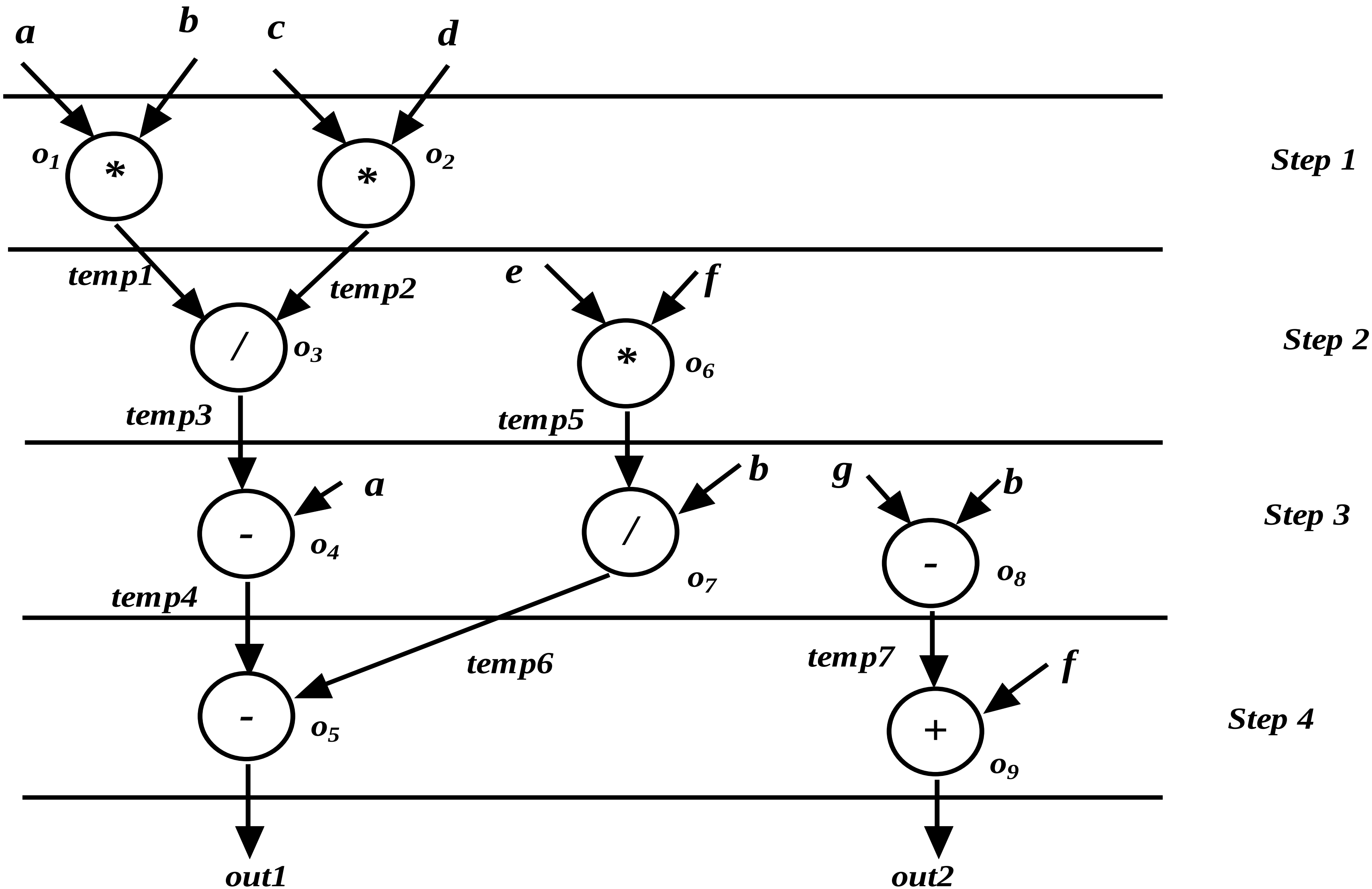
control_step(o_i) = control_step(o_j)- 1, o_j is immediate successor of o_i .

END

If all $o_i \in O$ are scheduled within control step1

then Status of scheduling is Successful

As Late As Possible Scheduling



scheduling for “out1=((a*b)/(c*d))-a-((e*f)/b)” and “out2=(g-b)+f”

As Late As Possible Scheduling

In this case, it may be noted that operations o_5, o_9 do not have any direct successors, i.e., they generate output values. So these operations have the control step as $M = 4$. Operation o_5 is the immediate successor of o_4, so, control_step(o_4) = control_step(o_5)-1=3. Similarly, control step assignment for all operations can be explained.

This schedule is complete within the 1st control step, thereby making it successful. The resource requirements are—

- Step1: 2 Multipliers
- Step2: 1 Dividers + (multiplier from Step1 can be used)
- Step3: 2 subtractors + (divider from Step2 can be used)
- Step4: 1 adder + (subtractor from Step3 can be used)

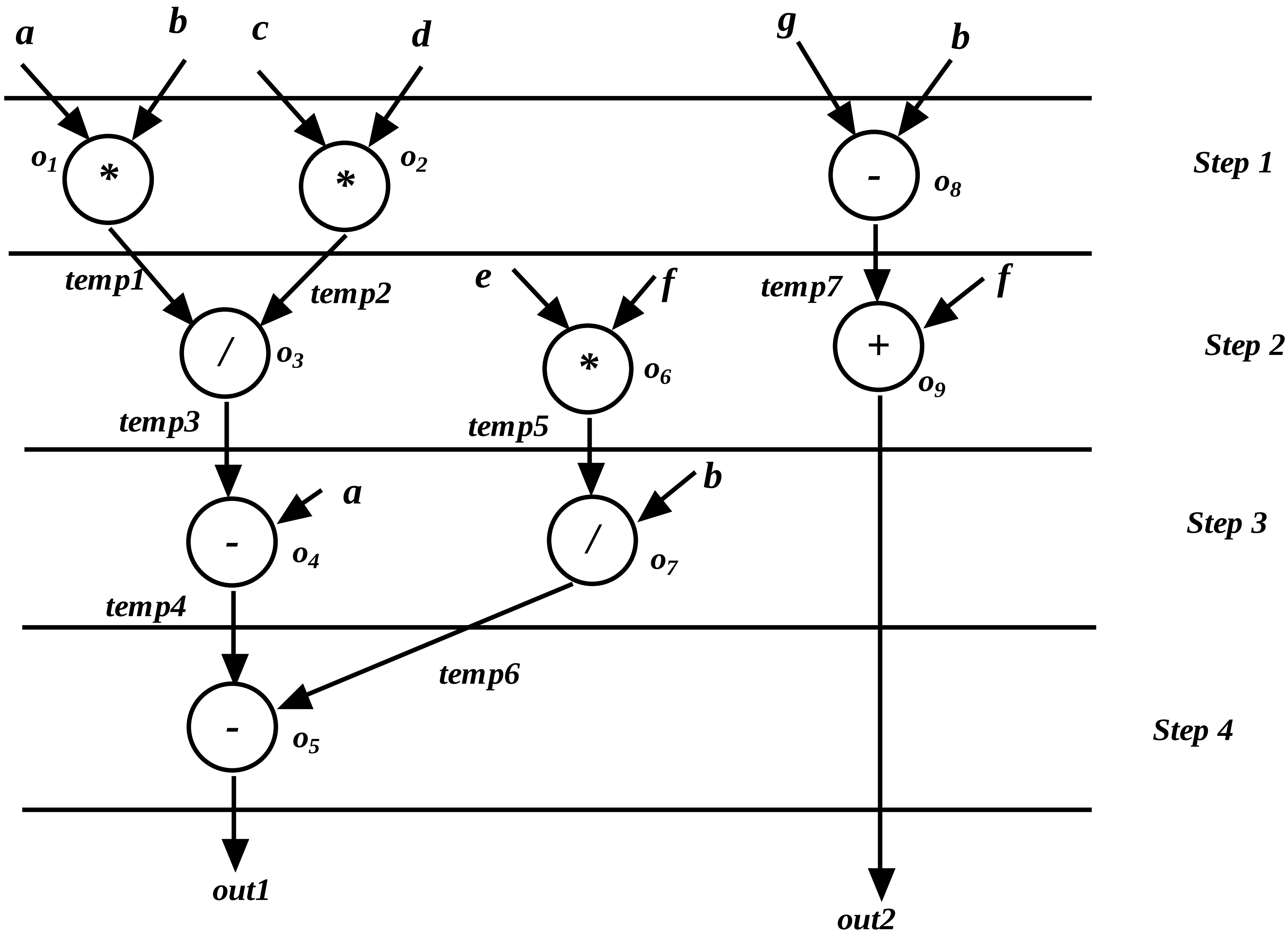
ASAP versus ALAP

If ALAP is compared with ASAP, it may be noted that we have achieved the following.

- Saved 1 Multiplier by delaying o_6 from step1 to step2.
- Saved 1 Divider by delaying o_7 from step2 to step3.
- Increased 1 subtractor by delaying o_8 from step1 to step3.

So, it may be observed that for the subpart of the expression, "(e*f)/b", ALSP is better compared to ASAP. However, for the expression, "out2=(g-b)+f" ASAP is better compared to ALAP. As already mentioned, ALSP and ASAP are heuristics and may not generate an optimal solution. By applying the scheme ALAP for "(e*f)/b" and ASAP for "out2=(g-b)+f", the schedule we obtain for $M = 4$ is shown next.

ASAP versus ALAP



ALAP scheduling for "((e*f)/b)" and ASAP for "out2=(g-b)+f"

ASAP versus ALAP

It may be noted that the resource consumption in this case is as follows.

- Step1: 2 Multipliers + 1 subtractor
- Step2: 1 Divider + (multiplier from Step1 can be used) + 1 adder
- Step3: 1 subtractors + (subtractor from Step1 can be used) + (divider from Step2 can be used)
- Step4: (subtractor from Step3 can be used)

So it may be noted that a schedule which is a “mix of ALAP and ASAP” provides better solution than by the individual algorithms.

Now we will see FDS scheduling algorithm which is motivated from above fact of combining ALAP and ASAP.

Force Directed Scheduling

•FDS starts by first finding ALSP and ALAP scheduling for all the operations. Operations whose ALAP and ASAP schedules are same (i.e., same control step is assigned by both ALAP and ASAP), are not considered to be re-scheduled by FDS as there is no flexibility in their positions.

•Following that, all operations are listed whose ALSP and ASAP schedules are different and the flexible range for such an operation is " [control step assigned by ASAP --to-- control step assigned by ALSP]".

•Now, we schedule these operations in one of their flexible steps, such that total count of operators is minimal.

- •To accomplish this, operations of each type are considered one by one. For a given type of operation, we analyze the total requirement of the number of operators (of the type under question), by considering the combinations of placing the corresponding operations in the steps within their intervals.
- •We select the combination that leads to minimal number of operators.
- •Once we are done with the operation of one type we move for the other types, one by one.

Force Directed Scheduling

Before providing the algorithm for FDS scheduling certain notations are introduced.

- $ASAP_i$: Control step scheduled by ASAP algorithm to operation o_i
- $ALAP_i$: Control step scheduled by ALAP algorithm to operation o_i
- $INTERVAL_i$: [$ASAP_i$ to $ALAP_i$]
- $RANGE_i$: $ALAP_i$ - $ASAP_i$ +1
- $PROB_{i,j}$: Probability of scheduling an operation o_i in control step j, $j \in INTERVAL_i$; $PROB_{i,j} = (RANGE_i)^{-1}$
- $LIST_{k,j}$: Set of all operations of type k in step j, i.e., set comprising all operations o_i of type k such that $j \in INTERVAL_i$.
- $COST_{k,j}$: Number of operators of type k required in step j.

$$COST_{k,j} = \sum_{o_i \in LIST_{k,j}} PROB_{i,j}$$

Force Directed Scheduling

Algorithm 3: Force Directed Scheduling

Input: ALSP and ASAP Scheduling.

Output: Control step for each operations, Status of scheduling .

Steps

From ASAP and ALAP scheduling, for all operations (i.e., $o_i \in O$) compute

$INTERVAL_i$, $RANGE_i$, $PROB_{i,j}$ (for all *j*), $LIST_{k,j}$ (for all *j*), $COST_{k,j}$ (for all *j*).

for each type of operation $k \in K$

DO

BEGIN /*loop finds best steps for all operations of type k */

$\Delta_{k(best)} = \infty$;

best_step=0;

for each operation o_i of type *k* whose $RANGE_i \geq 2$.

BEGIN /*loop finds best step for o_i */

for each $j \in INTERVAL_i$

BEGIN

Force Directed Scheduling

Temporarily schedule o_i in step j and compute the value of $COST_{k,j} (= \Delta_{k(new)})$ due to fixing the schedule of o_i and changes of schedule of other operations due to data dependency.

If $\Delta_{k(new)} < \Delta_{k(best)}$ then assign value of $\Delta_{k(new)}$ to $\Delta_{k(best)}$ and *best_step=j*.

END

Finally schedule o_i in step *best_step* and other operators due to data dependency.

Update for all operations (i.e., $o_i \in O$) $INTERVAL_i$, $RANGE_i$, $PROB_{i,j}$ (for all *j*), $LIST_{k,j}$ (for all *j*), $COST_{k,j}$ (for all *j*).

END

END

Force Directed Scheduling

Now we will illustrate the FDS algorithm with the running example of scheduling "out1=((a*b)/(c*d))-a-((e*f)/b)" and "out2=(g-b)+f".

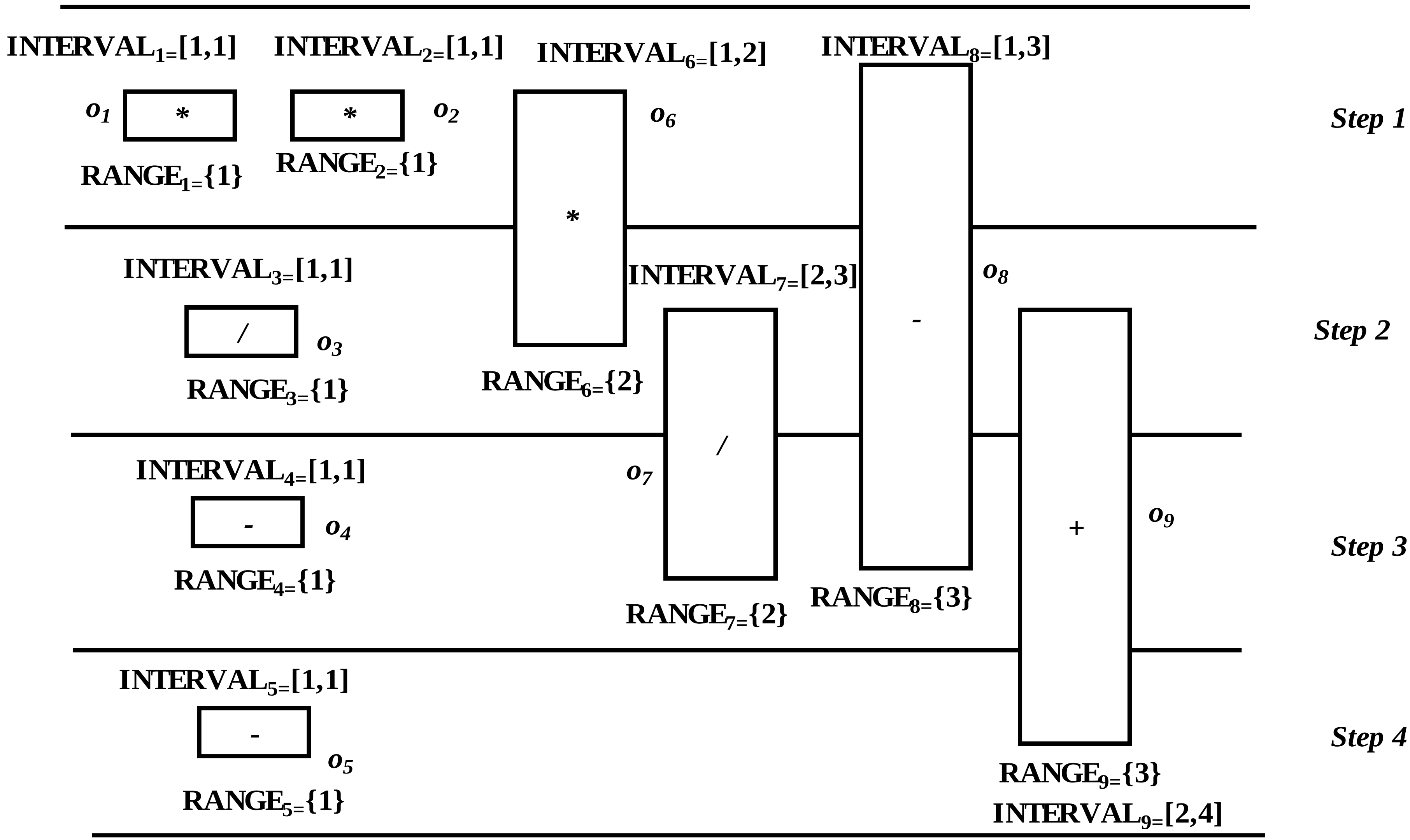
Next Figure illustrates $INTERVAL_i$ and $RANGE_i$, for all the operations. Here we have four types of operators; let *k=1* represent multiplier, *k=2* represent divider, *k= 3* represent subtractor and *k= 4* represent adder.

Now we will illustrate FDS for scheduling all operations of type *k=1*. Computation of $PROB_{i,j}$, for all the multiplication operations are as follows.

- $PROB_{1,1}=1$; $PROB_{2,1}=1$; $PROB_{6,1}=0.5$;
- $PROB_{1,2}=0$; $PROB_{2,2}=0$; $PROB_{6,2}=0.5$;

Further, $LIST_{k,j}$ for the first two steps for multiplication operations are $LIST_{1,1}=\{o_1, o_2, o_6\}$ and $LIST_{1,2}=\{o_6\}$. So, $COST_{1,1}=1+1+0.5=2.5$ and $COST_{1,2}=0.5$.

Force Directed Scheduling



$INTERVAL_i$ and $RANGE_i$ for "out1=((a*b)/(c*d))-a-((e*f)/b)" and "out2=(g-b)+f"

Force Directed Scheduling

Among three multiplication operations, we have freedom only in scheduling o_6 ($RANGE_6 = 2$). If we schedule o_6 in step1 then

- $PROB_{1,1}=1$; $PROB_{2,1}=1$; $PROB_{6,1}=1$;
- $PROB_{1,2}=0$; $PROB_{2,2}=0$; $PROB_{6,2}=0$;
- $LIST_{1,1}=\{o_1, o_2, o_6\}$ and $LIST_{1,2}=\{\}$.
- $COST_{1,1}=3$
- $\Delta_{k(new)}=3$; as $\Delta_{k(new)} < \Delta_{k(best)}, \Delta_{k(best)}$ is assigned 3 and *best_step=1*

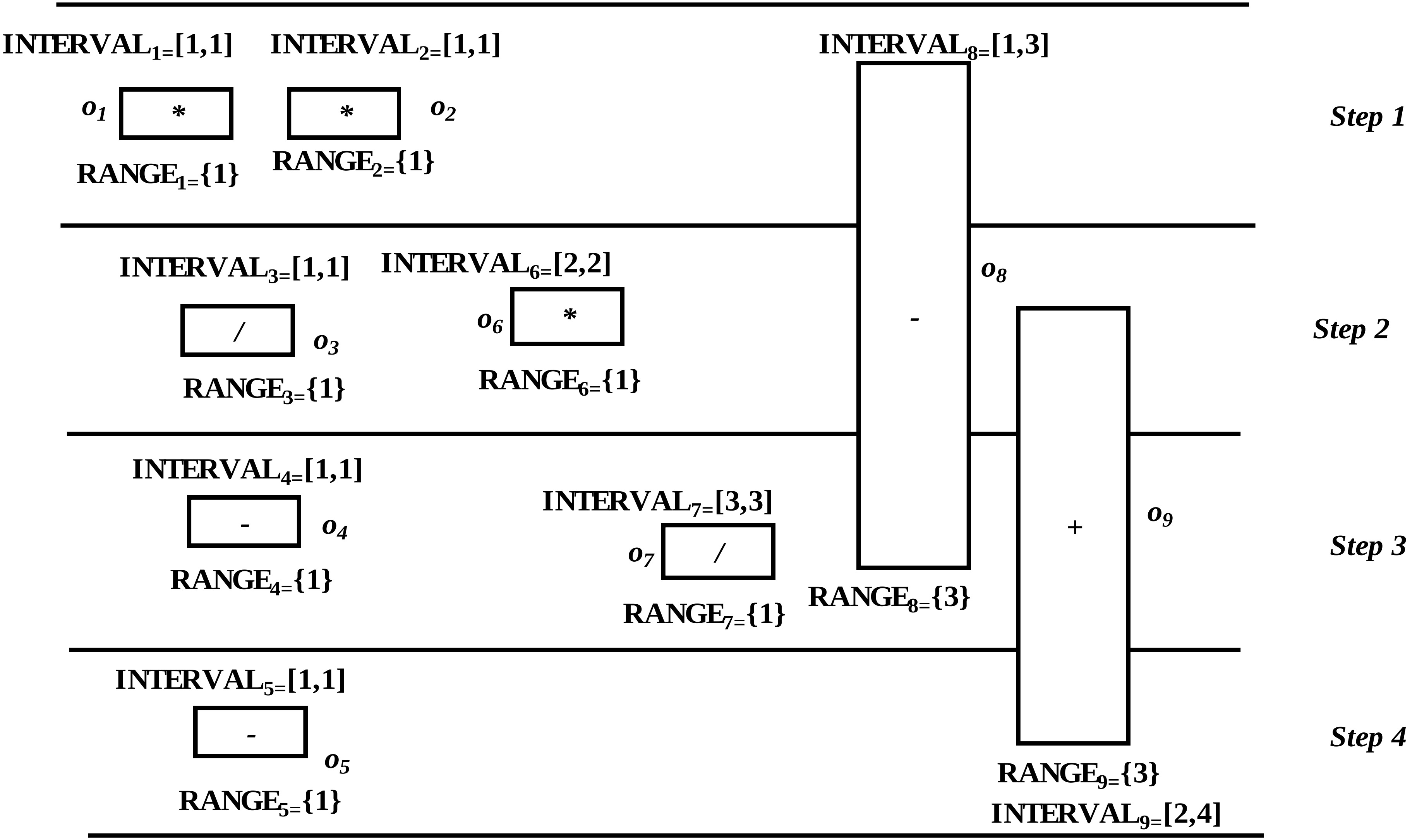
Force Directed Scheduling

We can also schedule o_6 in step2, which results in

- $PROB_{1,1}=1$; $PROB_{2,1}=1$; $PROB_{6,1}=0$;
- $PROB_{1,2}=0$; $PROB_{2,2}=0$; $PROB_{6,2}=1$;
- $LIST_{1,1}=\{o_1, o_2\}$ and $LIST_{1,2}=\{o_6\}$.
- $COST_{1,1}=2$
- $\Delta_{k(new)}=2$; as $\Delta_{k(new)}<\Delta_{k(best)}, \Delta_{k(best)}$ is assigned 1 and *best_step=2*.

So we schedule o_6 in step2, which results in fixing the schedule of o_7 in step3 (due to data dependency); o_7 loses its flexibility. This is shown in next figure .

Force Directed Scheduling



. $INTERVAL_i$ and $RANGE_i$ for "out1=((a*b)/(c*d))-a-((e*f)/b)" and "out2=(g-b)+f" after o_6 is scheduled in step2

Force Directed Scheduling

Similarly, computation of $PROB_{i,j}$, for all the subtraction operations are as follows.

- $PROB_{4,3}$=1.
- $PROB_{8,1}$=0.33; $PROB_{8,2}$=0.33; $PROB_{8,3} = 0.33$;

Further, $LIST_{k,j}$ for the first three steps for subtraction operations are $LIST_{3,1}$={ o_8 } and $LIST_{3,2}$={ o_8 } and $LIST_{3,3}$={ o_4, o_8 }. So, $COST_{3,1} = 0.33$, $COST_{3,2} = 0.33$ and $COST_{3,3} = 1.33$.

Among two subtraction operations we have freedom only in scheduling o_8 ($RANGE_8 = 3$). If we schedule o_8 in step1 then

- $PROB_{4,3}$=1.
- $PROB_{8,1}$=1; $PROB_{8,2}$=0; $PROB_{8,3} = 0$;
- $LIST_{3,1}$={ o_8 }, $LIST_{3,2}$={} and $LIST_{3,3}$={ o_4 }.
- $COST_{3,1} = 1$
- $\Delta_{k(new)} = 1$; as $\Delta_{k(new)} < \Delta_{k(best)}, \Delta_{k(best)}$ is assigned 1 and *best_step=1*

Force Directed Scheduling

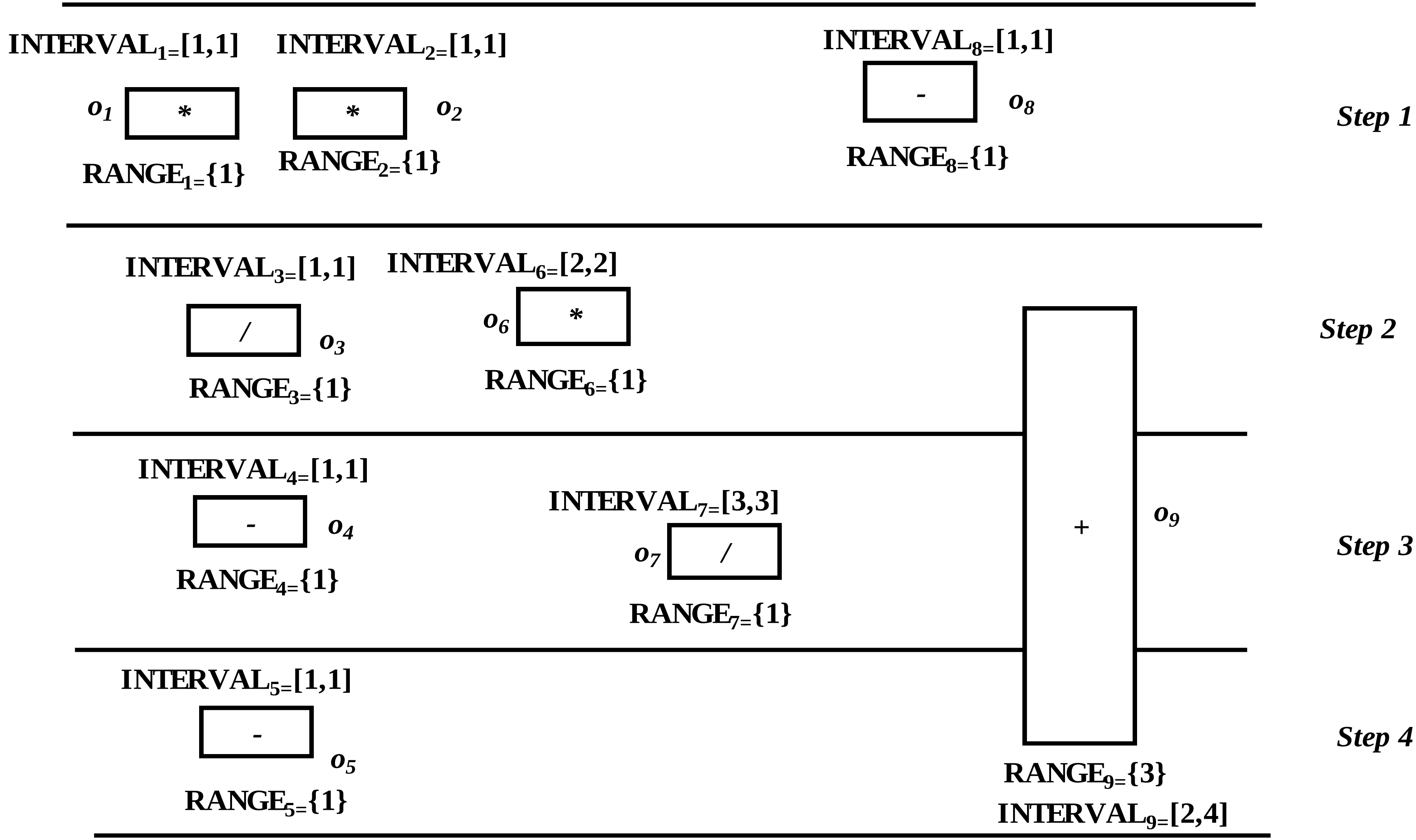
We can also schedule o_8 in step2, which results in

- $PROB_{4,3}$=1.
- $PROB_{8,1}$=0; $PROB_{8,2}$=1; $PROB_{8,3} = 0$;
- $LIST_{3,1}$={}, $LIST_{3,2}$={ o_8 } and $LIST_{3,3}$={ o_4 }.
- $COST_{3,2} = 1$
- $\Delta_{k(new)} = 1$; as $\Delta_{k(new)} = \Delta_{k(best)}$, $\Delta_{k(best)}$ is not changed and *best_step=1*

We can also schedule o_8 in step3, which results in

- $PROB_{4,3}$=1.
- $PROB_{8,1}$=0; $PROB_{8,2}$=0; $PROB_{8,3} = 1$;
- $LIST_{3,1}$={}, $LIST_{3,2}$={} and $LIST_{3,3}$={ o_4, o_8 }.
- $COST_{3,3} = 2$
- $\Delta_{k(new)} = 2$; as $\Delta_{k(new)} > \Delta_{k(best)}$, $\Delta_{k(best)}$ is not changed and *best_step=1*

Force Directed Scheduling



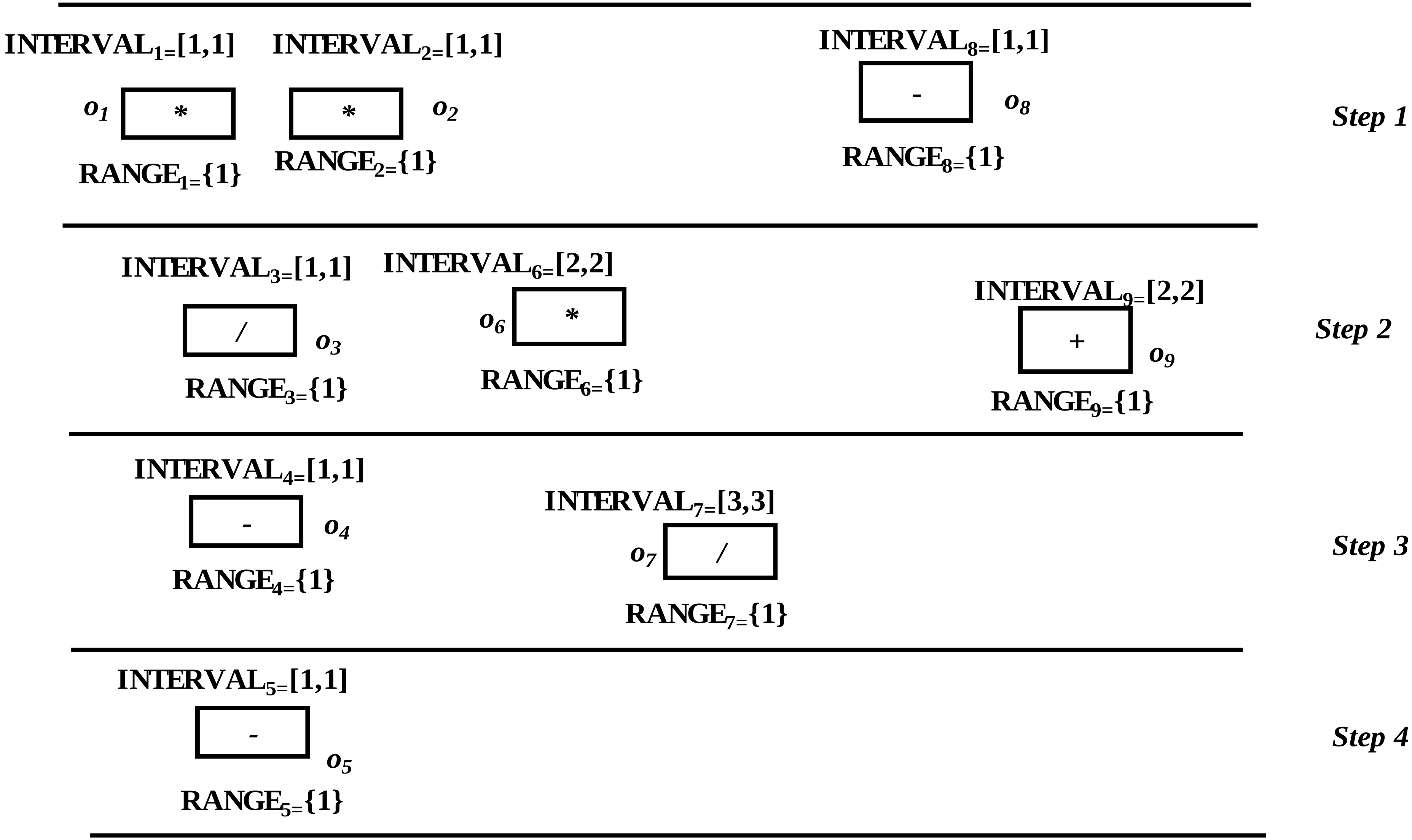
. $INTERVAL_i$ and $RANGE_i$ for "out1=((a*b)/(c*d))-a-((e*f)/b)" and "out2=(g-b)+f" after o_8 is scheduled in step1

Force Directed Scheduling

Similarly, it may be verified that FDS will schedule o_9 in step2; final schedule is shown in next figure.

It may be noted that this schedule is same as the one for ALAP+ASAP. Therefore, FDS obtains an optimal schedule considering a merger of ASAP and ALSP.

Force Directed Scheduling



. $INTERVAL_i$ and $RANGE_i$ for “out1=((a*b)/(c*d))-a-((e*f)/b)” and “out2=(g-b)+f” after o_9 is scheduled in step2

Design Verification and Test of Digital VLSI Circuits NPTEL Video Course

Module-II

Lecture-III

Scheduling Algorithms

List Scheduling

•All the scheduling algorithms we discussed till now were heuristics based on time constants, in terms of number of control steps. Now we discuss another heuristic scheduling algorithm which is resource constrained—List Scheduling.

•Unlike ASAP, ALAP or FDS scheduling, which process operations individually in a fixed order, list scheduling handles each control step individually (in increasing order).

•List scheduling works by trying to schedule "maximum" number of operations in the control step, subject to resource constraints and data dependency.

•During the scheduling process, list scheduling uses a ready list (hence the name) to keep track of data-ready operations subject to data dependency.

•The ready list in a control step comprises those unscheduled operations that can be scheduled into the current control step without violating the data dependency

•As long as there are operations in the ready list that meet the resource constraints, operations are chosen from that list and scheduled into the current control step.

List Scheduling

•If more than one operation from ready list can be scheduled in a control step, but violates resource constraints, then choice among the ready operations is made by a priority function.

One common priority function is "$ALAP_i - ASAP_i$" (i.e., range where the operation can be scheduled). Operations with smaller ranges (i.e., smaller mobility) are given higher priority, since there are fewer possible control steps into which those operations can be scheduled, and since delaying them to a later control step would more likely increase the overall length of the schedule.

List Scheduling

Algorithm 4: List Scheduling

Input: Operations O, Maximum number of operators of type k $k_{\max}$.

Output: Control step for each operations.

Steps

Prepare $READYLIST_1$ (ready list for first step) , which comprises all operations whose predecessors are input variables.

for each operation $o_i \in READYLIST_1$

DO

BEGIN

Schedule all operations $o_i \in READYLIST_1$ in control step1. If there is violation in resource requirement (i.e., number of operations of type k in $READYLIST_1$ is more than $k_{\max}$), then schedule according to priority function.

END

List Scheduling

Till there are operations o_i to be scheduled

DO

BEGIN

Prepare $READYLIST_j$ for the next control step (i.e., scheduling over for step j-1)

Schedule all operations $o_i \in READYLIST_j$ in control step j. If there is violation in resource requirement then schedule according to priority function.

END

List Scheduling

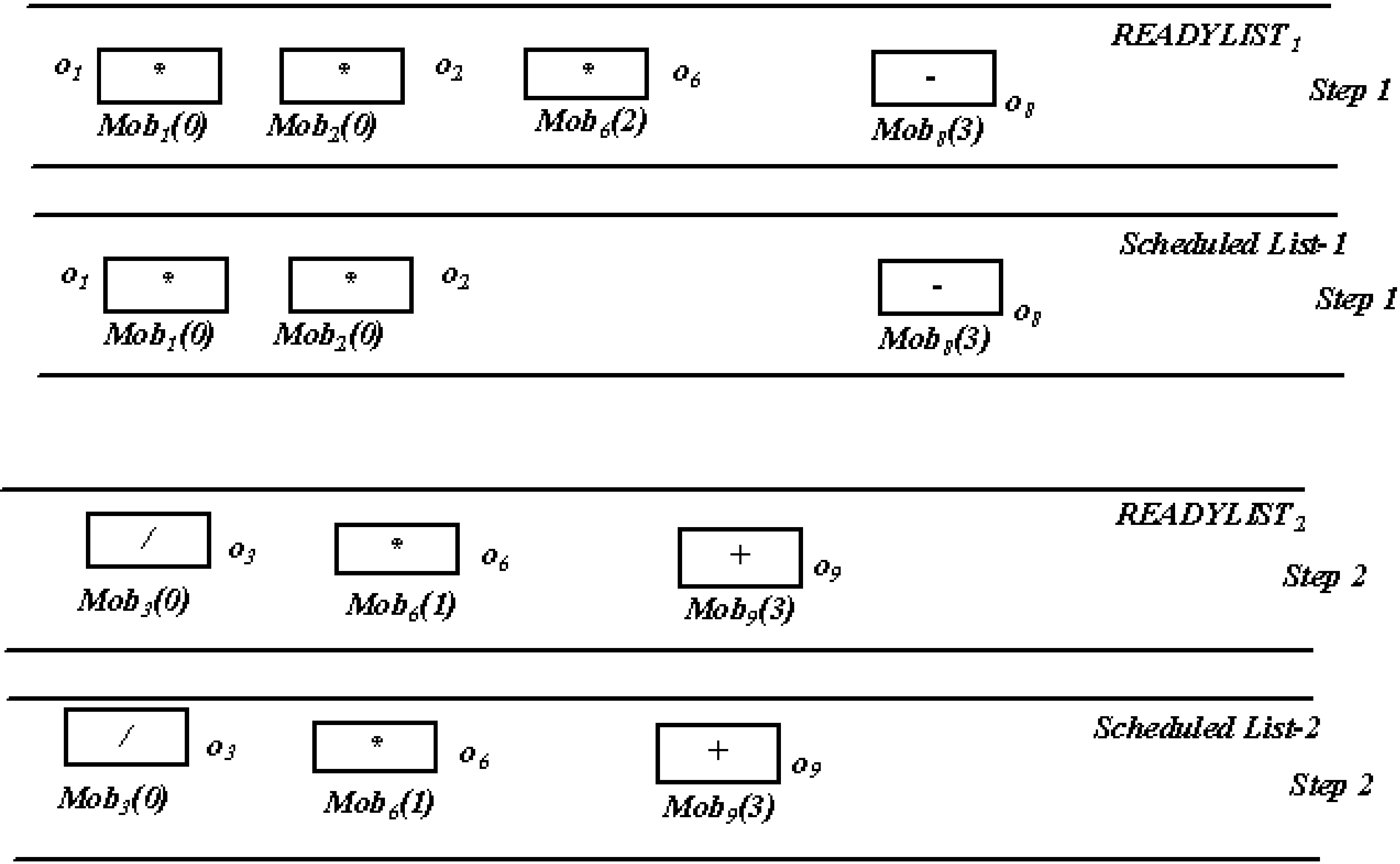
Now we will illustrate list scheduling for the running example of "out1=((a*b)/(c*d))-a-((e*f)/b)" and "out2=(g-b)+f" . Next Figure illustrates the schedule for each control step. We assume that we have two multipliers ($1_{max} = 2$), one divider ($2_{max} = 1$), one subtractor ($3_{max} = 1$) and one adder ($4_{max} = 1$).

List Scheduling

For step 1, the $READYLIST_1$ comprises o_1, o_2, o_6, o_8. We cannot schedule all three multiplication operations o_1, o_2, o_6 as $1_{max} = 2$. Mobility of the operations are shown in the figure. We assume that mobility is the priority function. As mobility of o_1, o_2 is 0, while for o_6 mobility is 2, we schedule only o_1, o_2 in step1. It may be noted that in step1 we can schedule o_8 without violating resource constraint of subtractor.

For step 2, the $READYLIST_2$ comprises o_3, o_6, o_9. We can schedule all three operations o_1, o_2, o_6 as requirement is one multiplier, one divider and one adder, which does not violate resource constraint. The $READYLIST_2$ and scheduled operations in step2 are shown in the figure (third and fourth rows).

List Scheduling



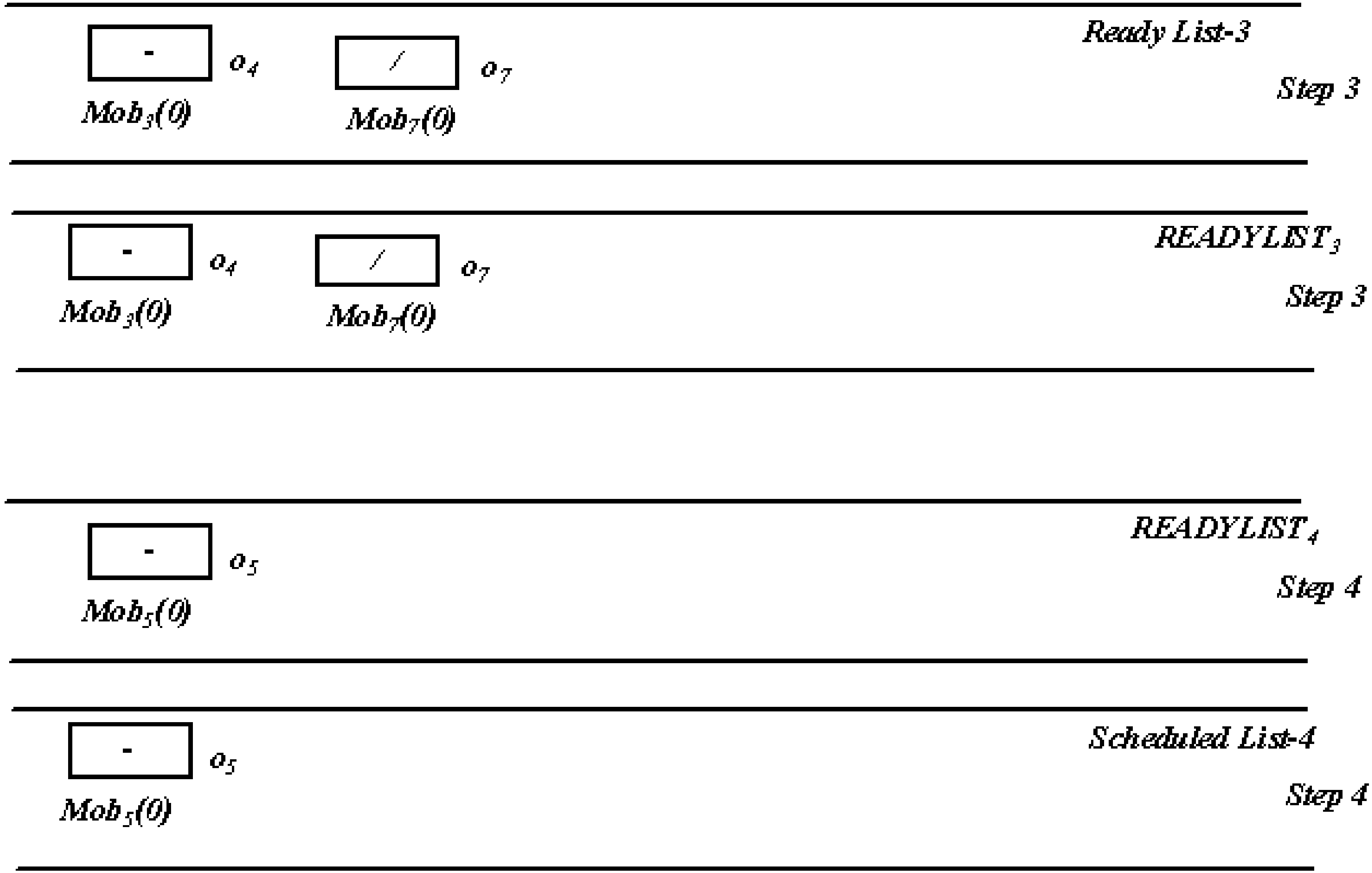
List Scheduling

For step 3, the $READYLIST_3$ comprises o_4, o_7. We can schedule the two operations as requirement is one subtractor and one divider, which does not violate resource constraint. The $READYLIST_3$ and scheduled operations in step3 are shown in the figure (fifth and sixth rows).

For step 4, the $READYLIST_4$ comprises o_5. We can schedule the operation as requirement is one subtractor, which does not violate resource constraint. The $READYLIST_4$ and scheduled operations in step4 are shown in the figure (seventh and eight rows).

As there are no more operations left, list scheduling is complete,

List Scheduling



Integer Linear Programming based Scheduling

•Integer Linear Programming (ILP), have been used to solve a wide range of constraint based optimization problems

•ILP formulation for optimally solving the synthesis problem.

•The biggest advantage of using ILP is the quality of the solution; unlike the heuristics based scheduling algorithms, described earlier, an ILP solver is guaranteed to find an optimal schedule from these formulations. However, this guarantee of quality comes at a price — ILPs cannot, in general, be solved in polynomial time. Thus, the tradeoff is between the guarantee of solution quality and a guarantee of quickly finding a solution.

Integer Linear Programming based Scheduling

- Now we will formulate the scheduling problem as ILP. Unlike the discussion on other scheduling cases (above), we will not give a generalized algorithm to formulate an ILP for a scheduling problem. However, we will consider the running example and formulate ILP for the same and the discussion will give a generalized idea of the procedure of such a formulation.

- In the ILP for scheduling, we have four sets of equations namely,
  (i) to capture the range ([ASAP-ALAP]) in which an operation can be scheduled,
  (ii) requirement that there is no violation of resource constraints,
  (iii) data dependency and
  (iv) optimize the resource requirements.

Range of scheduling

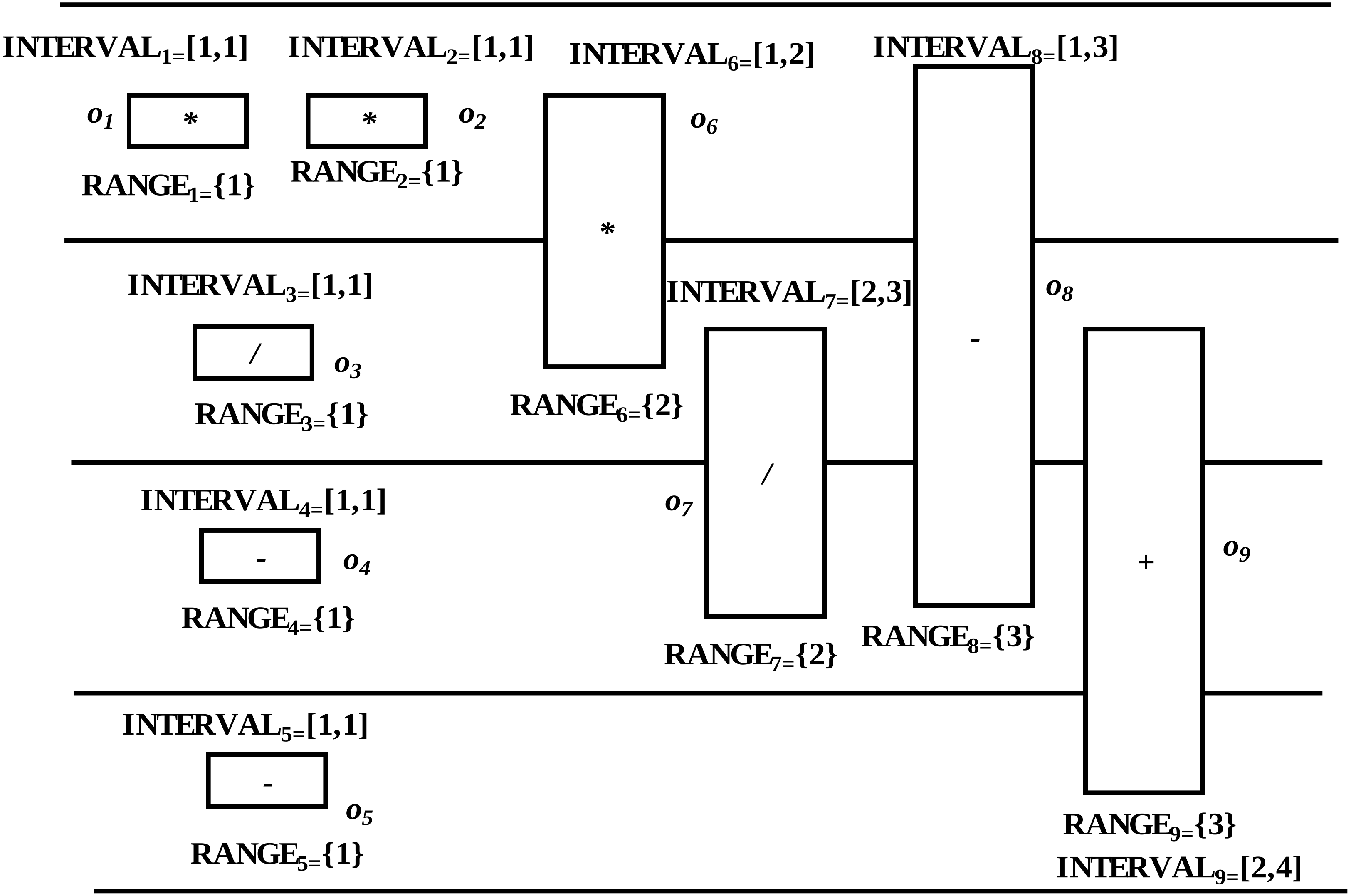
Let $o_{i,j}$ denote the scheduling of operation o_i in step *j*. $o_{i,j}$ is a variable for the ILP.

For scheduling we consider 0-1 ILP, where, in the solution we can have only 0 and 1 values of the variables. In the running example, operation o_1 must be scheduled in step1.

The ILP equation capturing this fact is $o_{1,1} = 1$, which implies that for the ILP solution, variable $o_{1,1}$ can only have the solution as 1.

However, for operation o_6 the equation is $o_{6,1} + o_{6,2} = 1$, which captures the fact that o_6 can be scheduled in step1 or step2. In the ILP solution either, $o_{6,1}$ will have the value 1 and $o_{6,2} = 0$ (implying that o_6 is scheduled in step1) or $o_{6,2}$ will have the value 1 and $o_{6,1} = 0$ (implying that o_6 is scheduled in step2).

Range of scheduling



Range of scheduling

In a similar way, all the equations for the running example which capture range of scheduling are given below:

- $o_{1,1} = 1$ (range of o_1)
- $o_{2,1} = 1$ (range of o_2)
- $o_{3,2} = 1$ (range of o_3)
- $o_{4,3} = 1$ (range of o_4)
- $o_{5,4} = 1$ (range of o_5)
- $o_{6,1} + o_{6,2} = 1$ (range of o_6)
- $o_{7,2} + o_{7,3} = 1$ (range of o_7)
- $o_{8,1} + o_{8,2} + o_{8,3} = 1$ (range of o_8)
- $o_{9,2} + o_{9,3} + o_{9,4} = 1$ (range of o_9)

Requirement :no violation of resource constraints

In the running example, at control step1, we can have three multiplication operations and a subtraction operation.

However, this requirement should not violate the resource constraints.

In the example we have multiplier, divider, subtractor and adder. Let $1_{\max}, 2_{\max}, 3_{\max}, 4_{\max}$ denote the maximum number of multipliers, dividers, subtractors and adders, respectively.

So for the first control step, equation $o_{1,1} + o_{2,1} + o_{6,1} \leq 1_{\max}$, represents that multiplication operations o_1, o_2, o_6 can be scheduled in step1, however, their number must be less than maximum allowed multipliers ($1_{\max}$).

Requirement :no violation of resource constraints

- $o_{1,1} + o_{2,1} + o_{6,1} \leq 1_{\max}$ (multipliers in step1)
- $o_{8,1} \leq 3_{\max}$ (subtractor in step1)
- $o_{6,2} \leq 1_{\max}$ (multiplier in step2)
- $o_{3,2} + o_{7,2} \leq 2_{\max}$ (dividers in step2)
- $o_{8,2} \leq 3_{\max}$ (subtractor in step2)
- $o_{9,2} \leq 4_{\max}$ (adder in step2)
- $o_{4,3} \leq 3_{\max}$ (subtractor in step3)
- $o_{7,3} \leq 2_{\max}$ (divider in step3)
- $o_{8,3} \leq 3_{\max}$ (subtractor in step3)
- $o_{9,3} \leq 4_{\max}$ (adder in step3)
- $o_{5,4} \leq 3_{\max}$ (subtractor in step4)
- $o_{9,4} \leq 4_{\max}$ (adder in step4)

Data dependency

It may be noted that there are some operations in the running example, whose position are fixed namely, o_1, o_2 in step1 and o_3 in step2.

It may be noted that there is data dependency between o_1, o_2 and o_3; o_3 can be scheduled only after o_1, o_2.

However, as the positions of o_1, o_2, o_3 are fixed, we need not write explicit expressions in ILP for their data dependencies. The three equations representing the range of scheduling of o_1, o_2, o_3 ($o_{1,1} = 1$, $o_{2,1} = 1$ and $o_{3,2} = 1$) capture that dependency relation; o_1, o_2 is scheduled in step1 and o_3 is scheduled in step2, thereby satisfying "o_3 can be scheduled only after o_1, o_2".

Data dependency

However, for operations like o_6 (which can be scheduled in step1 or step2) and o_7 (which can be scheduled in step2 or step3), equation $o_{6,1} + o_{6,2} = 1$ states that o_6 can be scheduled in step1 or step2 and equation $o_{7,2} + o_{7,3} = 1$ states that o_7 can be scheduled in step2 or step3.

However, these two equations cannot guarantee that o_6 cannot be scheduled in step2 along with o_7; this will lead to inconsistency as there is dependency between o_6 and o_7.

So for such flexible operations we need equations in ILP representing the dependency.

Data dependency

$(1o_{6,1}+2o_{6,2})-(2o_{7,2}+3o_{7,3})\leq -1$ captures this dependency; the mechanism is explained as follows.

There are four solutions to the schedule of o_6 and o_7, after satisfying the equations representing their ranges ($o_{6,1}+o_{6,2}=1$ and $o_{7,2}+o_{7,3}=1$),

(i) o_6 in step1 and o_7 in step2,

(ii) o_6 in step1 and o_7 in step3,

(iii) o_6 in step2 and o_7 in step2,

(iv) o_6 in step2 and o_7 in step3.

Among the four cases only (iii) is to be avoided; it may be noted that $o_{6,2}=1$ (so, $o_{6,1}=0$) and $o_{7,2}=1$ (so, $o_{7,3}=0$), will not satisfy "$(1o_{6,1}+2o_{6,2})-(2o_{7,2}+3o_{7,3})\leq -1$" (however, will satisfy $o_{6,1}+o_{6,2}=1$ and $o_{7,2}+o_{7,3}=1$).

Data dependency

So, equation $(1o_{6,1}+2o_{6,2})-(2o_{7,2}+3o_{7,3})\leq -1$ could incorporate the data dependency in the IPL.

To generalize, the equation $(1o_{6,1}+2o_{6,2})-(2o_{7,2}+3o_{7,3})\leq -1$, is formulated as follows.

The parts of the equation, "$1o_{6,1}+2o_{6,2}$" and "$2o_{7,2}+3o_{7,3}$" are taken from the range equation for o_6 and o_7, respectively, after multiplying the terms with the corresponding control step number. Then we subtract the sub-equation of the successor from that of the processor and the result should be less than or equal to -1. "less than or equal to -1" corresponds that o_6 is to be predecessor of o_7.

Data dependency equations for the running example are as follows:

- $(1o_{6,1}+2o_{6,2})-(2o_{7,2}+3o_{7,3})\leq -1$
- $(1o_{8,1}+2o_{8,2}+3o_{8,3})-(2o_{9,2}+3o_{9,3}+4o_{9,4})\leq -1$

Optimize the resource requirements

This equation represents the optimization criterion to minimize the resource cost.

In the running example, the equation is

Minimize "$1_{max} + 2_{max} + 3_{max} + 4_{max}$", subject to satisfying all the equations for range, resource constraints and data dependency given above.

Final Solution of the ILP

If we solve the 0-1 ILP using any standard method, we get the following solution:

- $o_{1,1} = 1$
- $o_{2,1} = 1$
- $o_{3,2} = 1$
- $o_{4,3} = 1$
- $o_{5,4} = 1$
- $o_{6,1} = 0; o_{6,2} = 1$
- $o_{7,2} = 0; o_{7,3} = 1$
- $o_{8,1} = 1; o_{8,2} = 0; o_{8,3} = 0$
- $o_{9,2} = 1; o_{9,3} = 0; o_{9,4} = 0$
- $1_{\max} = 2; 2_{\max} = 1; 3_{\max} = 1; 4_{\max} = 1$

Final Solution of the ILP

- It may be noted that this solution corresponds to the schedule of FDS; it can also be determined that this schedule is most optimal in terms of resource requirements if time step constraint is 4.

- Now a question arises, if FDS can do the same schedule then why we need the complex ILP based solution. The answer to this question lies in the fact that FDS may not always lead to optimal solution while ILP guarantees to generate the most optimal solution. In the question and answer section of this lecture, we illustrate a situation when FDS may not generate an optimal solution.

Some issues

First, we assumed that all types of operators have same cost in terms of resource requirements. However, in a real situation some operators involve much more hardware than others. Therefore, our scheduling algorithms need to consider also the cost of hardware of the operators.

Secondly, we assume that each operator takes one control step to do the operation. However, in a real situation some operators take more time to complete the computation

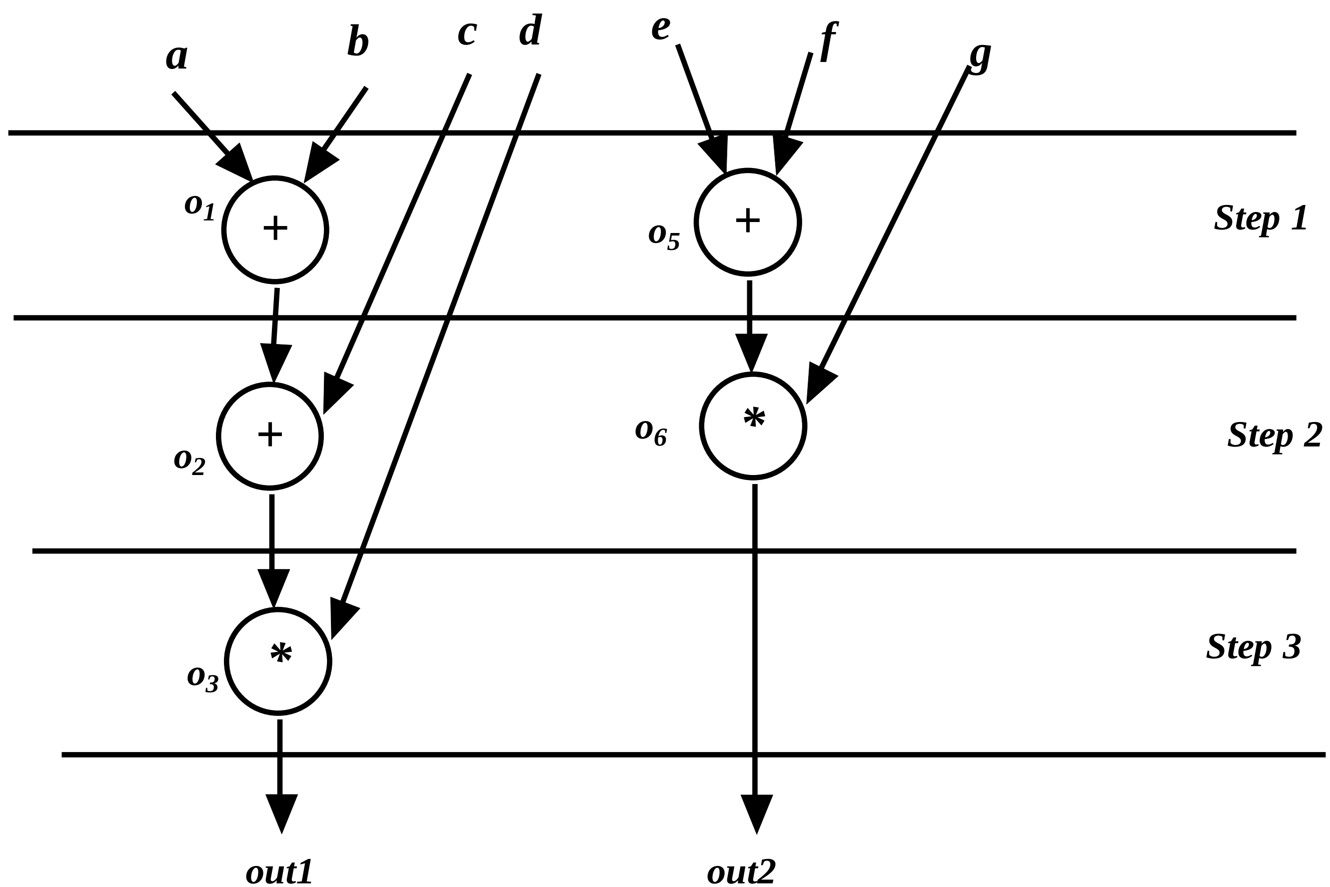
Third, we assume that each operator can perform only one function. However, in practice most of the operators are capable of doing multiple types of operations e.g., an adder can do both addition and subtraction (with slight modification)

Question and Answer

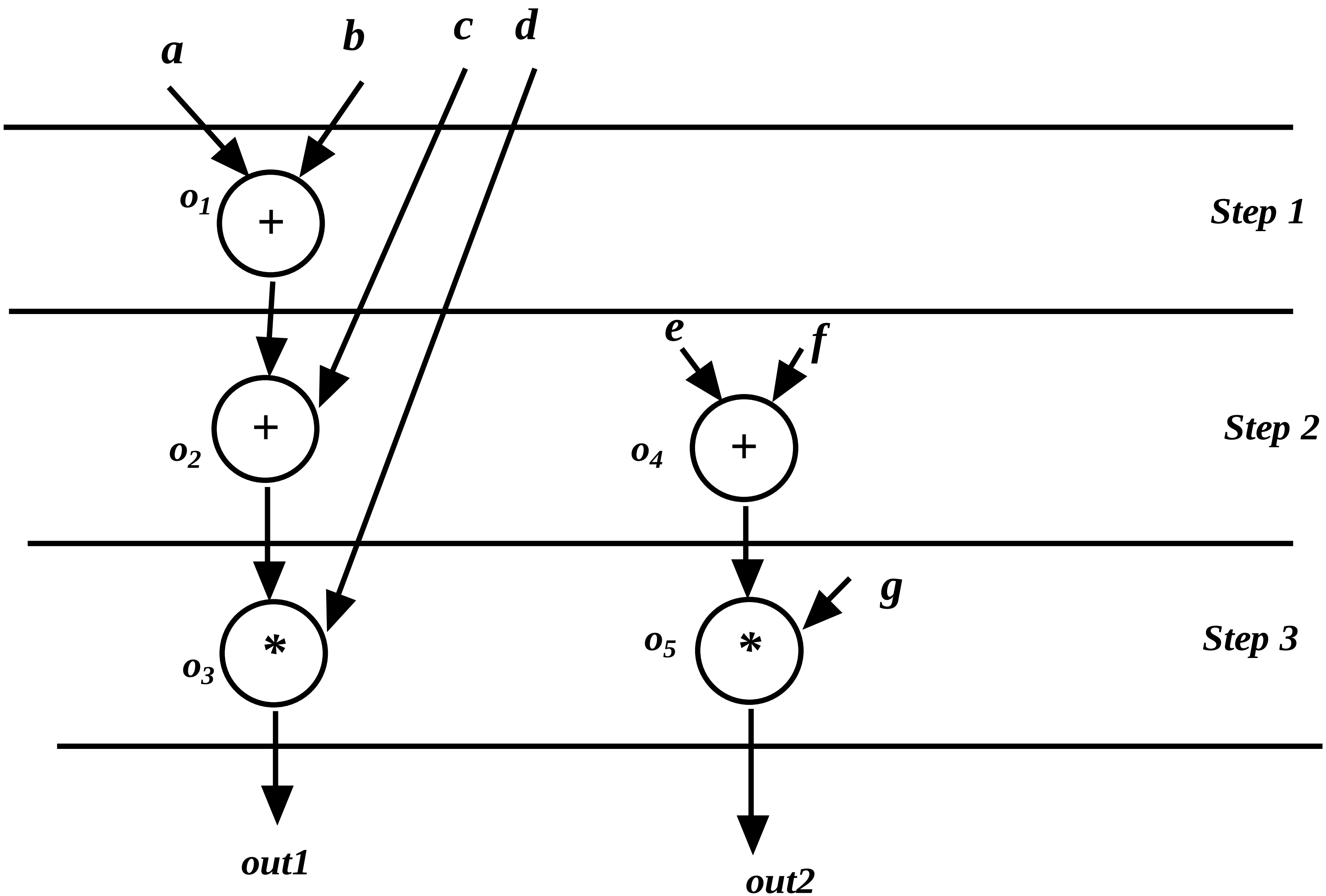
Question: Illustrate an example of scheduling where FDS does not provide optimal results in terms of resource requirements.

Answer

Consider the following expressions “out1=(a+b+c)*d” and “out2=(e+f)*g” . ASAP and ALSP schedule for “out1=(a+b+c)*d” and “out2=(e+f)*g” are shown in next two figures.



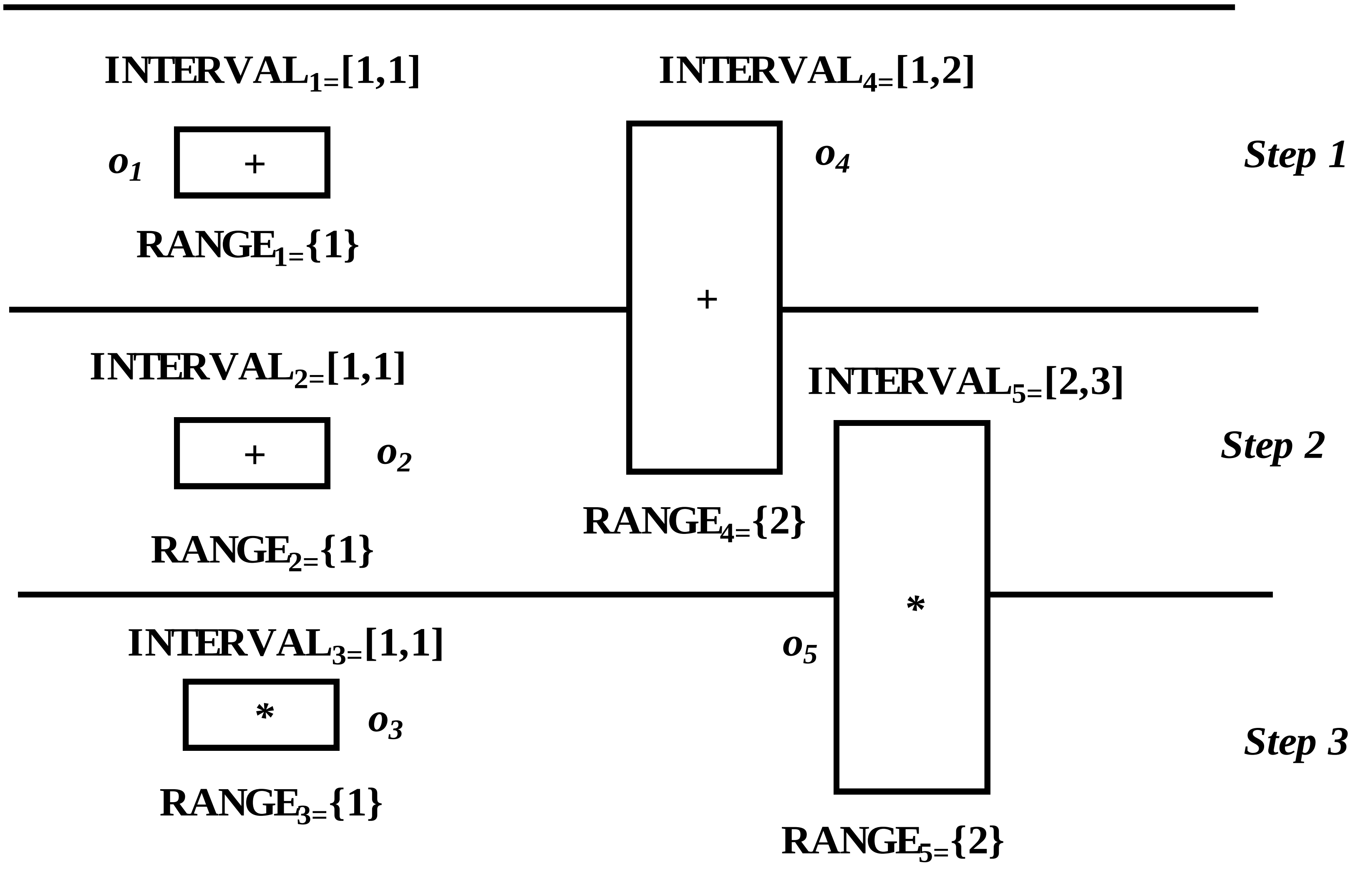
Question and Answer



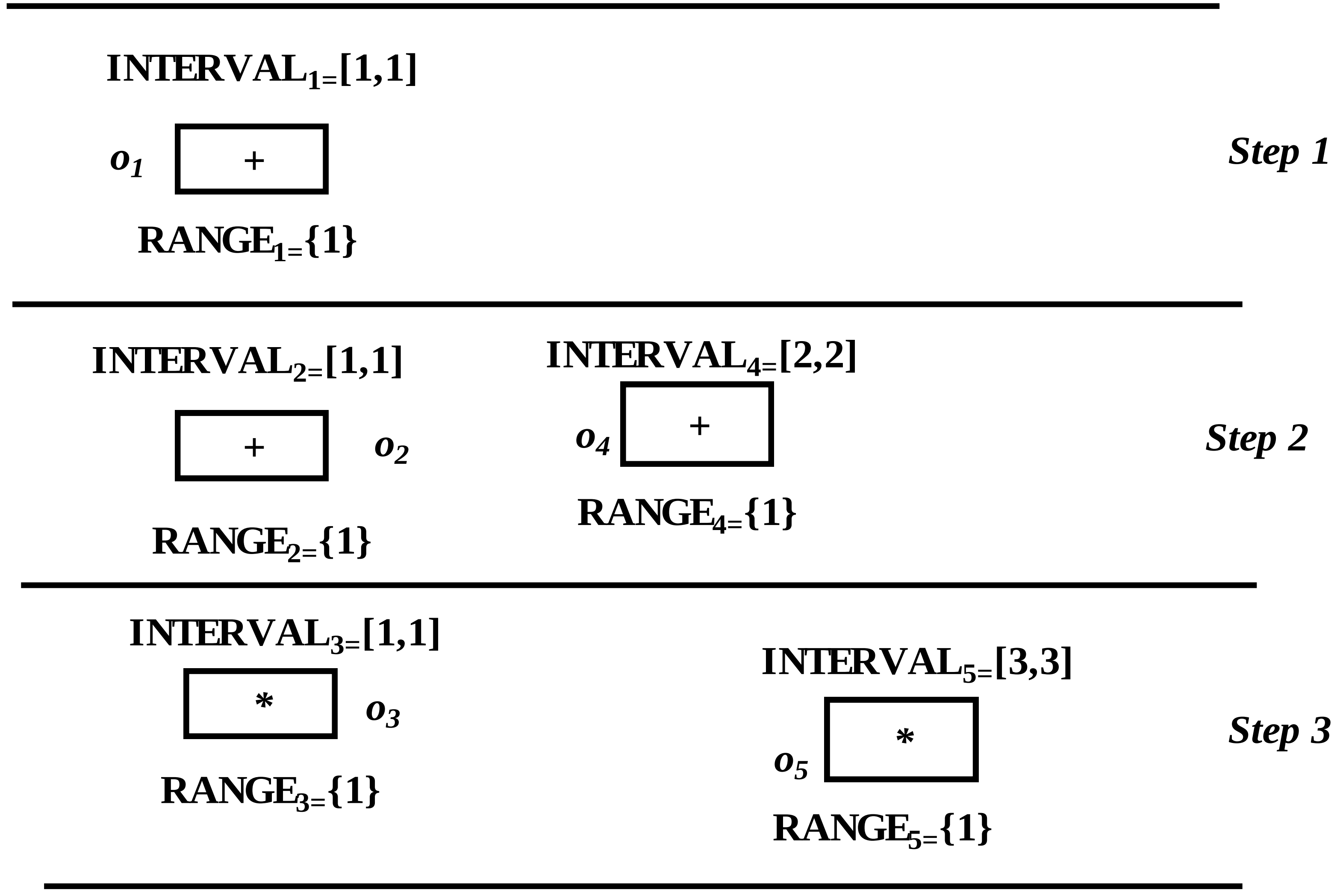
ALAP schedule for "out1=(a+b+c)*d" and "out2=(e+f)*g"

Question and Answer

$INTERVAL_i$ and $RANGE_i$ for "out1=(a+b+c)*d" and "out2=(e+f)*g" are shown in next figure.



Question and Answer



FDS status of “out1=(a+b+c)*d” and “out2=(e+f)*g” after o_4 is scheduled in step2

Question and Answer

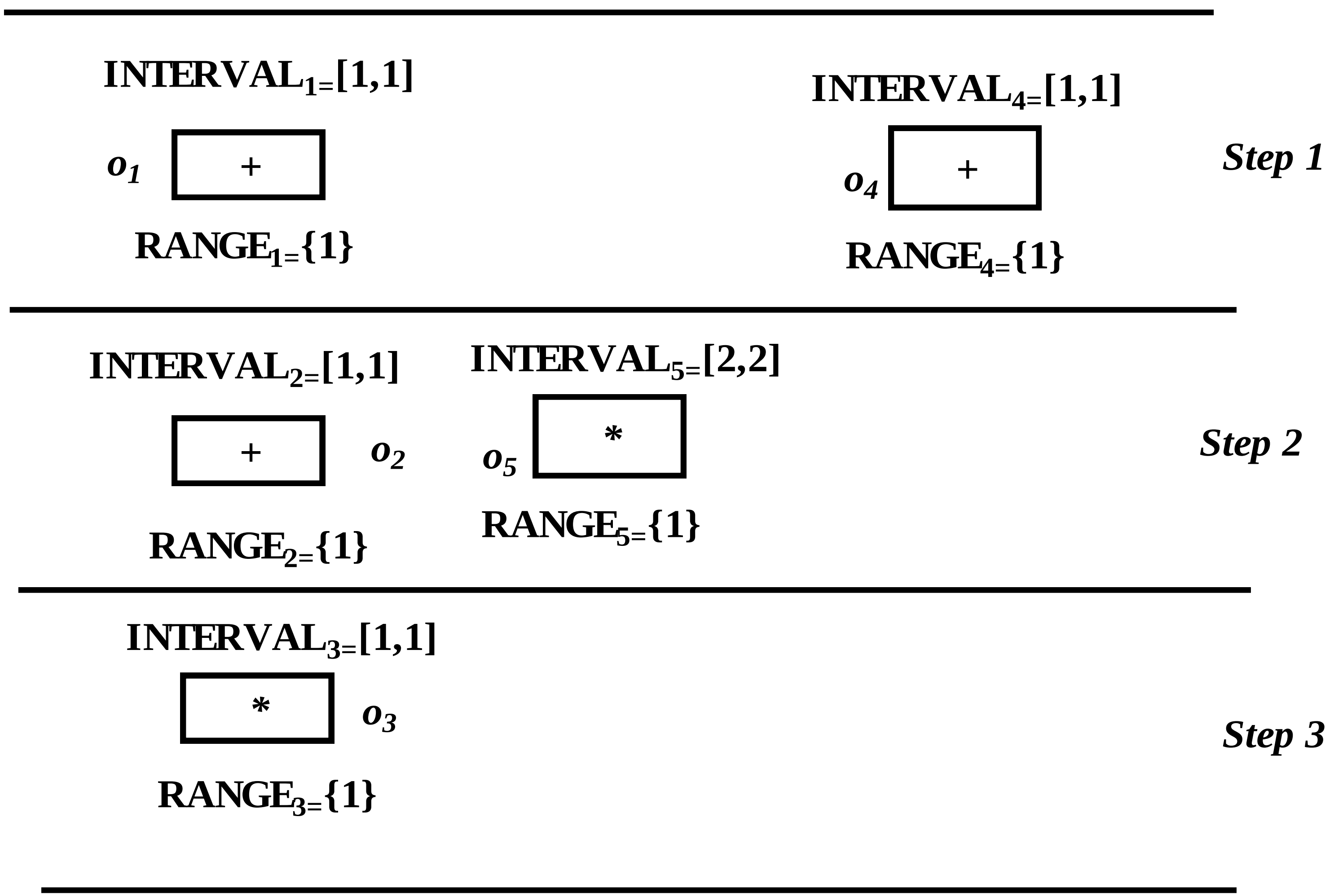
As FDS algorithm does not specify which type of operator to start with, let us consider adder first.

From FDS algorithm it may be noted operation o_4 can be scheduled in either step1 or step2, because both will lead to same cost in terms of resource requirements (number of adder is two).

So let the operation o_4 be scheduled in step2, which implies that operation o_5 will be placed in step3.

Now the total resource requirement is two adders and two multipliers.

Question and Answer



FDS status of “out1=(a+b+c)*d” and “out2=(e+f)*g” after o_5 is scheduled in step2

Question and Answer

Let us see the other option and schedule operation o_4 in step1. Now by FDS operator o_5 will be placed in step2. Now the total resource requirement is two adders and one multiplier. So FDS may provide non-optimal solution;

Design Verification and Test of Digital VLSI Circuits NPTEL Video Course

Module-II

Lecture-IV

Binding and Allocation Algorithms

Introduction

After the scheduling process, which assigns control steps to all the operations, in the allocation step, circuit modules from the design library are selected for executing the operations. Once circuit modules are selected, binding is done, which accomplishes the following:

- Functional unit binding: All arithmetic and logic operations are binded to the specific circuit modules allocated from the design library.

- Storage to register binding: A storage operation is created for each data transfer that crosses a control step boundary. Also, all inputs are to be stored in variables and binded to registers.

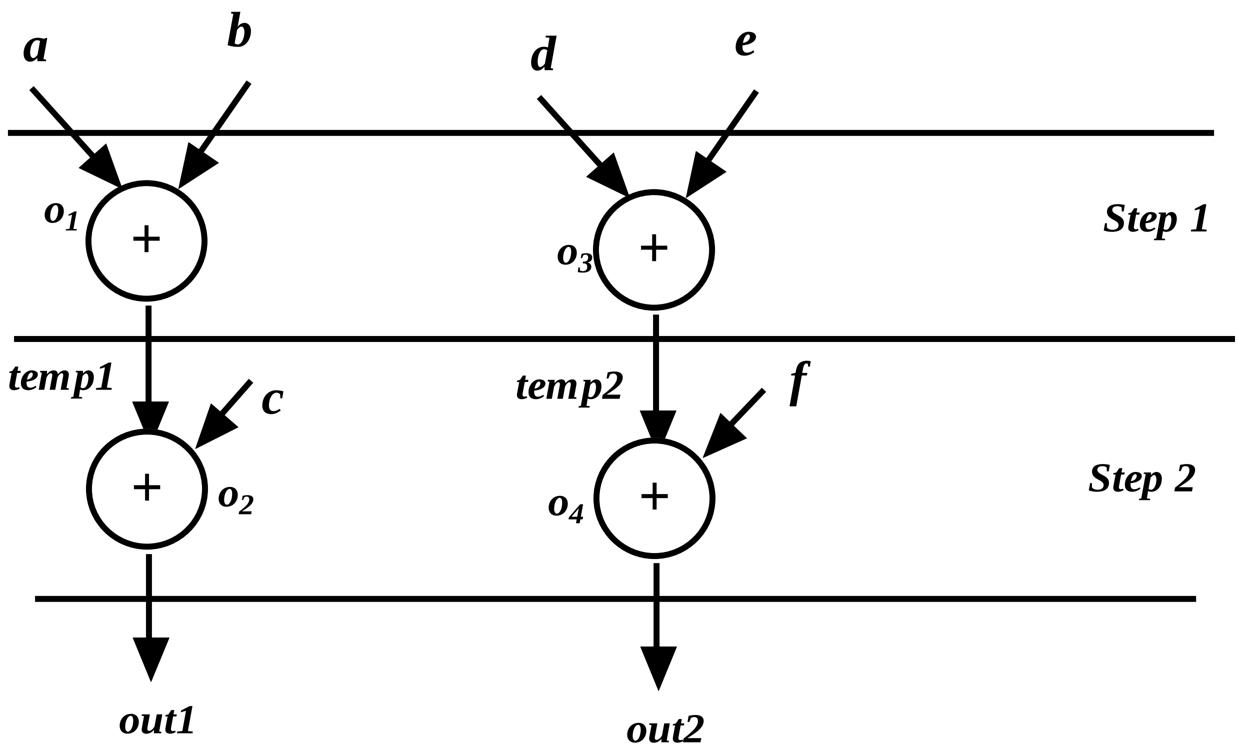
- Data-transfer to interconnect binding: Any data transfer involves an interconnection between source and sink. Therefore, any data transfer is to be binded with an interconnection (from source to destination). In addition, it might be noted that interconnects are shared by data transfers which leads to use of multiplexers in the sources and destinations.

Example: Binding of functional units, storages and data-transfer

A schedule of expressions "out1=a+b+c" and "out2=d+e+f" is shown. Let the allocation be as follows:

- Two (ripple carry) adders
- Four registers (D-flip-flops)

We need two adders because in control step1 (also in step2) two addition operations are scheduled and each need an adder to operate. Also we need four registers because in step1, we need four variables (storage) namely, *a,b,c,d.* These four registers can be re-used in step2 for variables *temp1,c,temp2,f.*



Example: Binding of functional units, storages and data-transfer

Let us consider the following option of binding

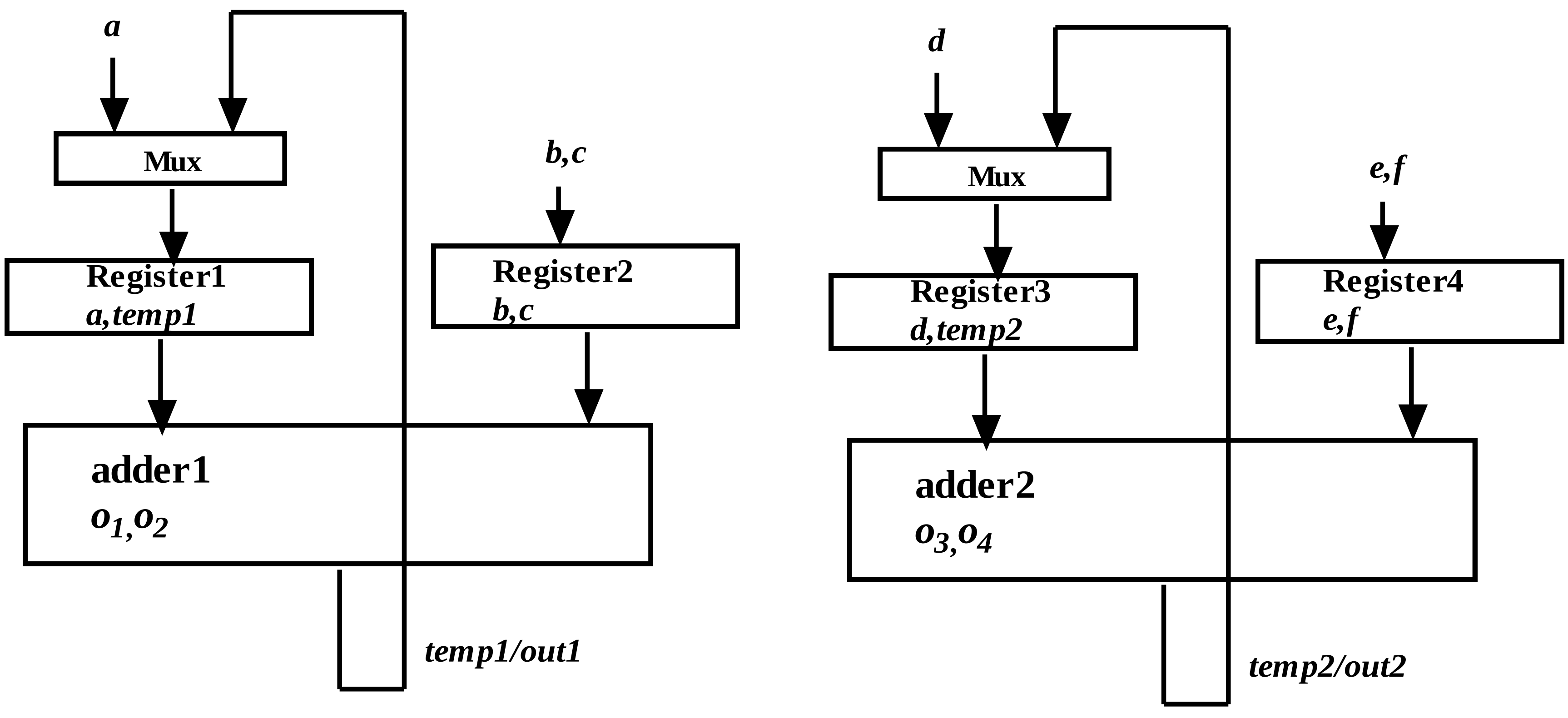
- Operations o_1, o_2 are binded to adder1
- Operations o_3, o_4 are binded to adder2
- Variables *a*,*temp1*,*out1* are binded to register1
- Variables *b*,*c* are binded to register2
- Variables *d*,*temp2*,*out2* are binded to register3
- Variables *e*,*f* are binded to register4

Some of the binding of the data-transfers with the interconnects are as follows:

- adder1 to register1 (via Mux) is binded to data transfer "temp1=a+b"
- Input a to register1 (via Mux) is binded to data transfer "reading a from input bus"
- Input b (and c) to register2 (via Mux) is binded to data transfer "reading b from input bus" ("reading c from input bus")

Example: Binding of functional units, storages and data-transfer

Case 1 of Binding for the schedule above



Example: Binding of functional units, storages and data-transfer

- It may be noted that as two data transfers (point 1 and point 2, above) are binded to regsiter1, we need a multiplexer that feeds to the input of register1. Similarly, we require a multiplexer at input of register3.

- It may be noted that even if two data transfers "reading b from input bus" and "reading c from input bus" are binded to register2, there is no multiplexer at input of register2. This is because we connect the input line to register2, where in step1 we have value of b and in step2 we have value of c.

- For a similar reason we do not require a multiplexer for input of register4.

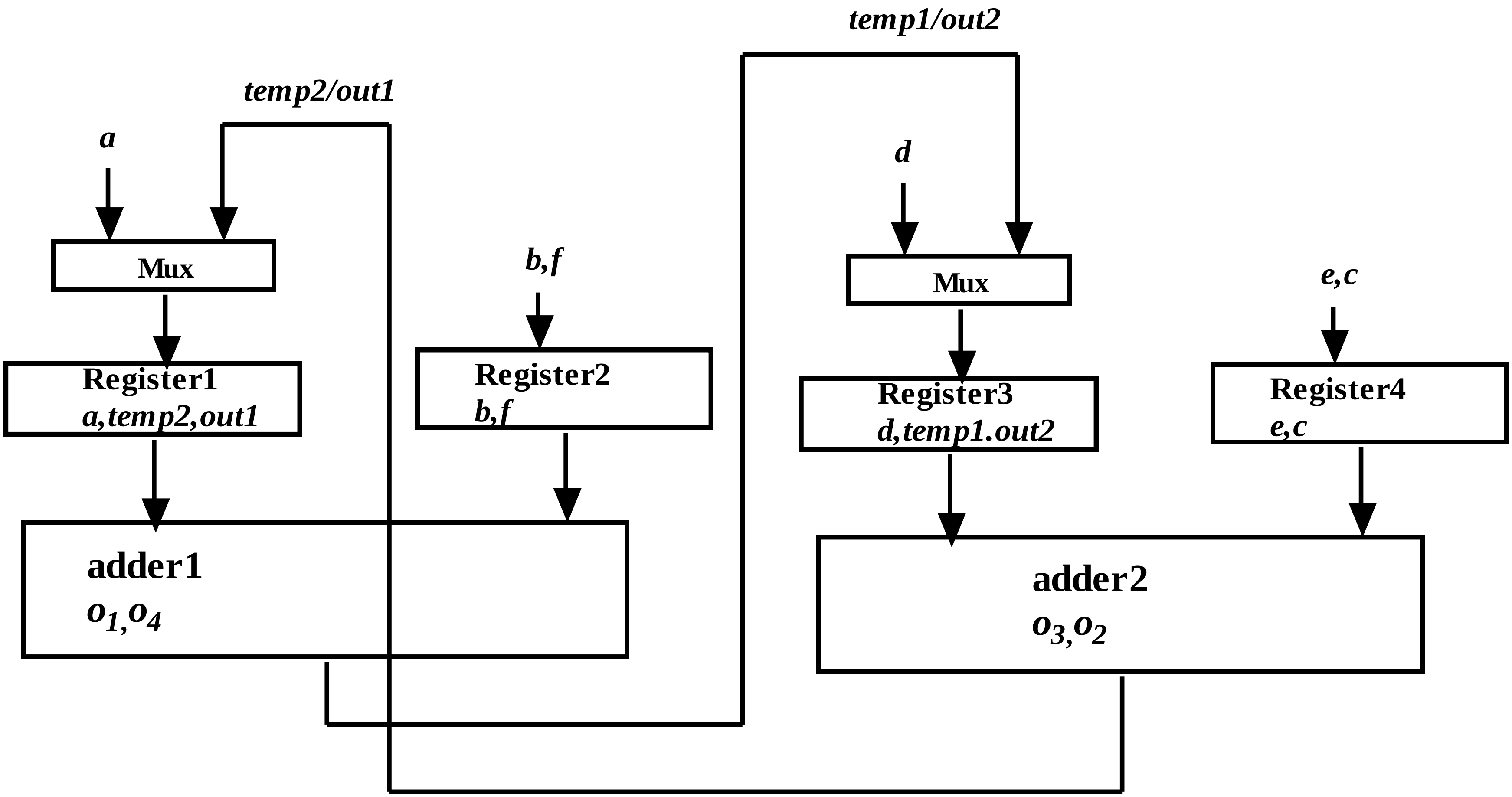
Example: Binding of functional units, storages and data-transfer

Let us consider the another option of binding

- Operations o_1, o_4 are binded to adder1
- Operations o_2, o_3 are binded to adder2
- Variables *a,temp2,out1* are binded to register1
- Variables *b,f* are binded to register2
- Variables *d,temp1,out2* are binded to register3
- Variables *e,c* are binded to register4

The interconnects are illustrated in next figure and can be interpreted in a similar manner as discussed for the last case. It may be noted that in this case also we require two multiplexers in the circuit.

Example: Binding of functional units, storages and data-transfer

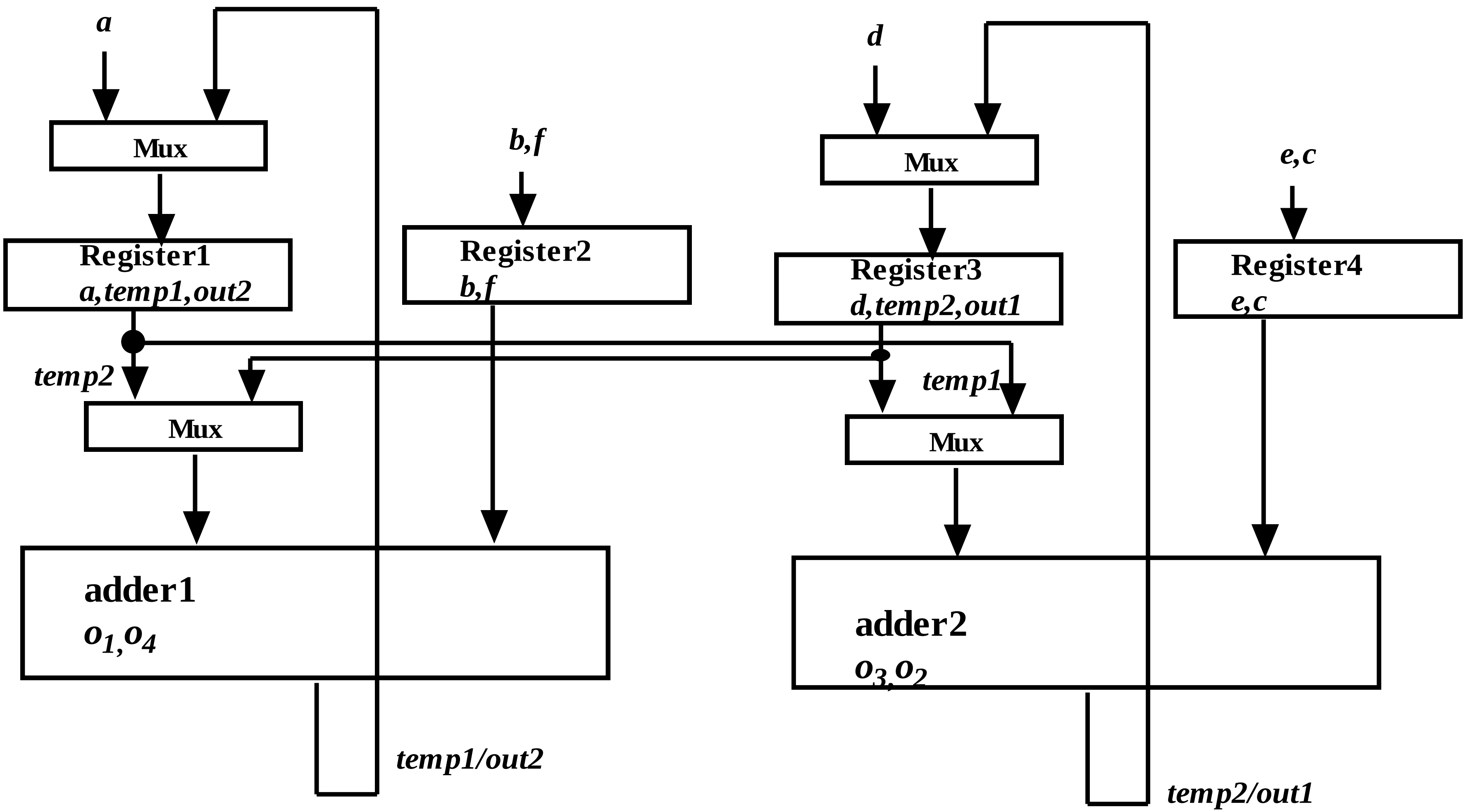


Example: Binding of functional units, storages and data-transfer

Now, let us consider the third option of binding:

- Operations o_1, o_4 are binded to adder1
- Operations o_2, o_3 are binded to adder2
- Variables *a*,*temp1*,*out2* are binded to register1
- Variables *b*,*f* are binded to register2
- Variables *d*,*temp2*,*out1* are binded to register3
- Variables *e*,*c* are binded to register4

Example: Binding of functional units, storages and data-transfer



Example: Binding of functional units, storages and data-transfer

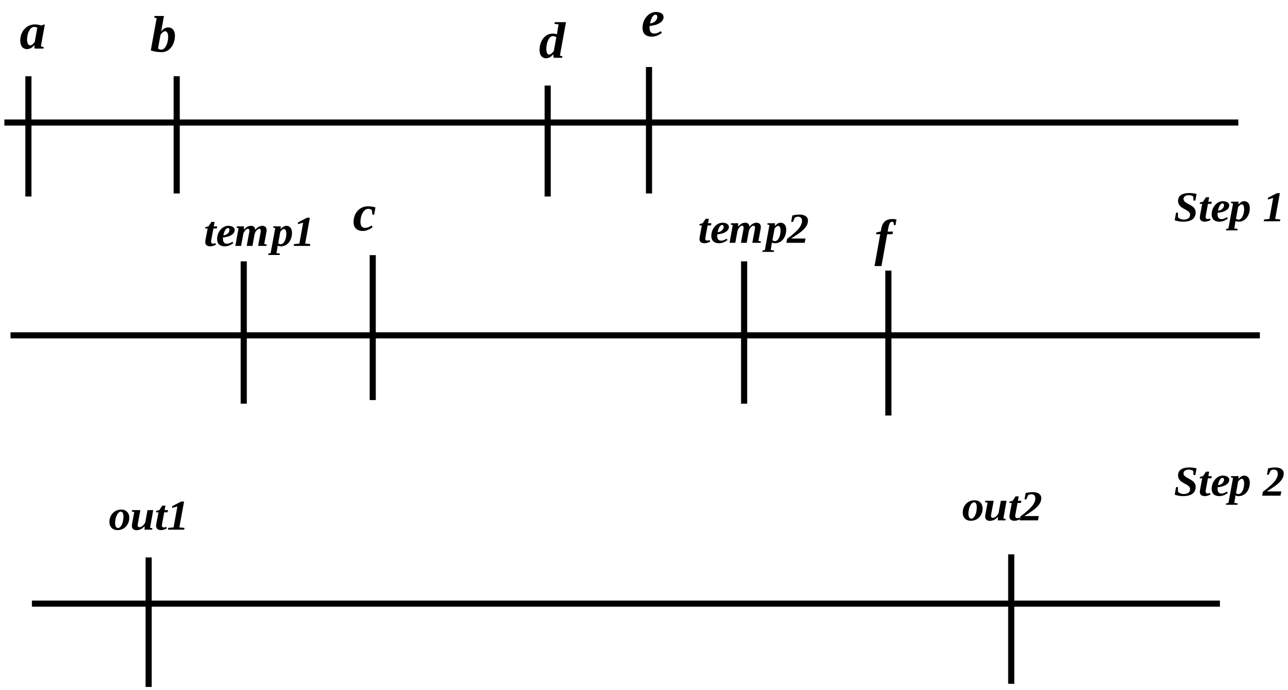
•It may be noted that in this case we require four multiplexers in the circuit. Two multiplexers at inputs of register1 and register3 are added for the same reason as discussed in the last two cases.

•Now we see why two more multiplexers at inputs of both the adders are required. It may be observed from the figure that data transfer “a to operand of adder1” is binded to interconnect “register1 (source)--left input of adder1 (destination)” and “temp2 to operand of adder1” is binded to interconnect “register3 (source)--left input of adder1 (destination)”. As there are two different interconnects for the left input of adder1, we require a multiplexer. Similarly, we require another multiplexer at input of adder2.

•So, it can be concluded that depending on binding, the area taken by interconnects (including multiplexers) varies.

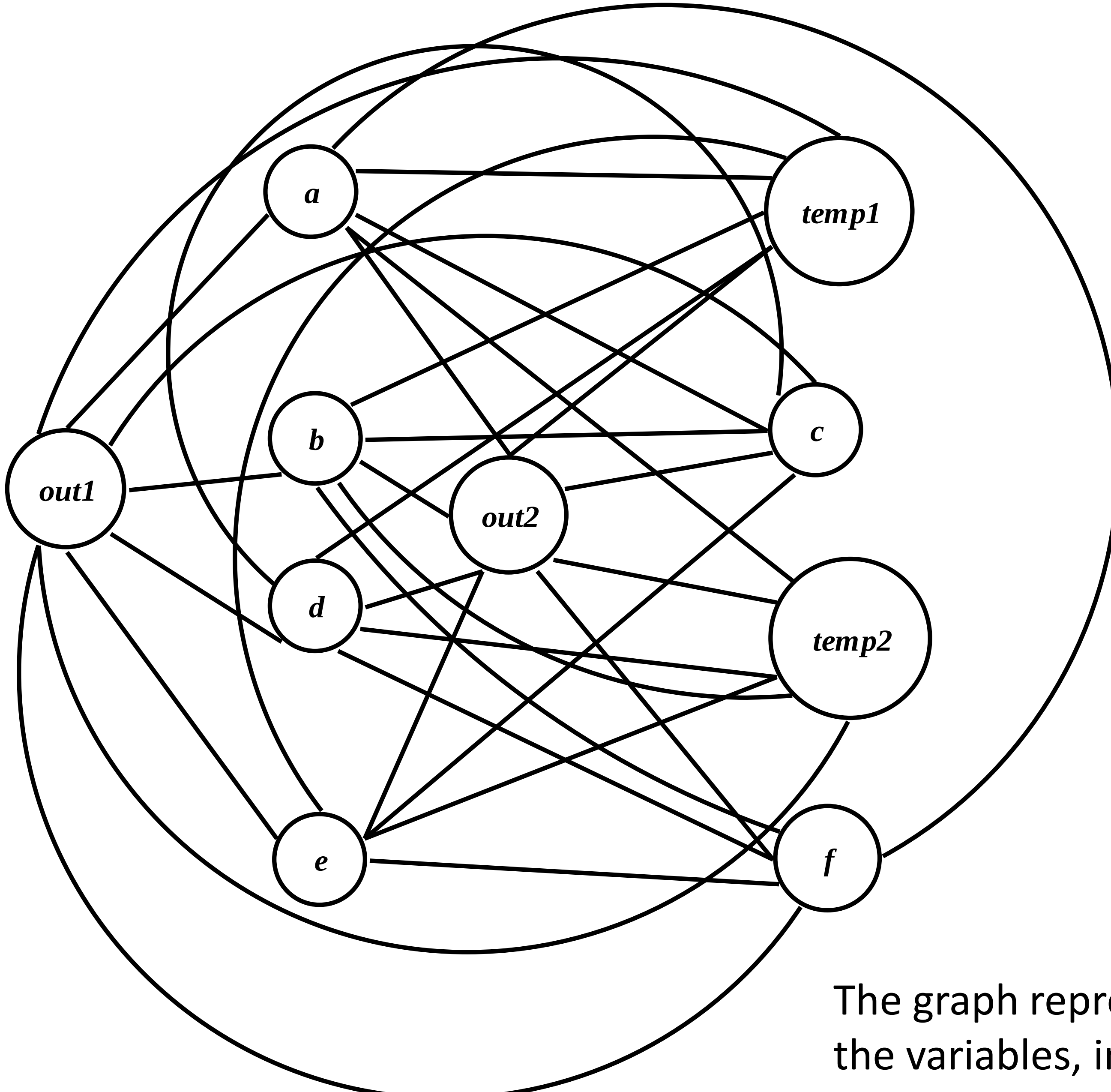
Binding using clique partitioning

In clique partitioning based binding, the operations and variables are modeled in terms of a graph. Each variable (if storage binding is done, or operation, if functional unit binding is done) is modeled by a node in the graph. There is an edge between two nodes only if the lifetime of the variables (or operations) does not overlap.



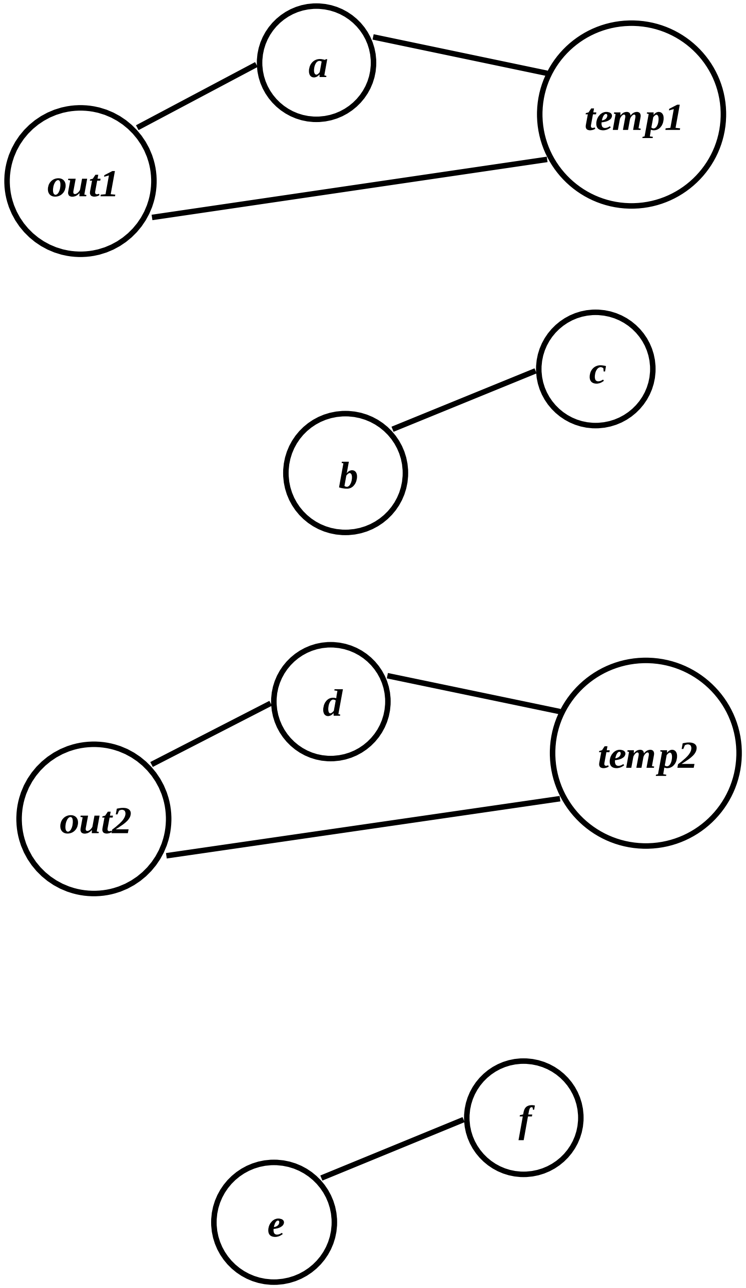
It may be noted that variables *a,b,c,d* are required in step1 only, thereby making their life time only step1. Similarly, life time of variables *temp1,c,temp2,f* is step2 and *out1,out2* are alive only in step3.

Binding using clique partitioning



The graph representation of the variables, in terms of lifetime is illustrated

Binding using clique partitioning



Binding using clique partitioning

•It may be noted that there if two variables exits whose lifetime do not overlap, then they are connected by an edge. For example, *out1* and *a* are connected by an edge while *a* and *b* are not.

•Now, for binding, we need to determine maximal cliques in the graph.

• The clique problem is to find complete subgraphs ("cliques") in a graph, i.e., sets of elements where each pair of nodes is connected.

•For each maximal clique we need a hardware resource of the corresponding type. All variables (or operations) corresponding to the nodes of the maximal clique are binded to the hardware module selected for the clique.

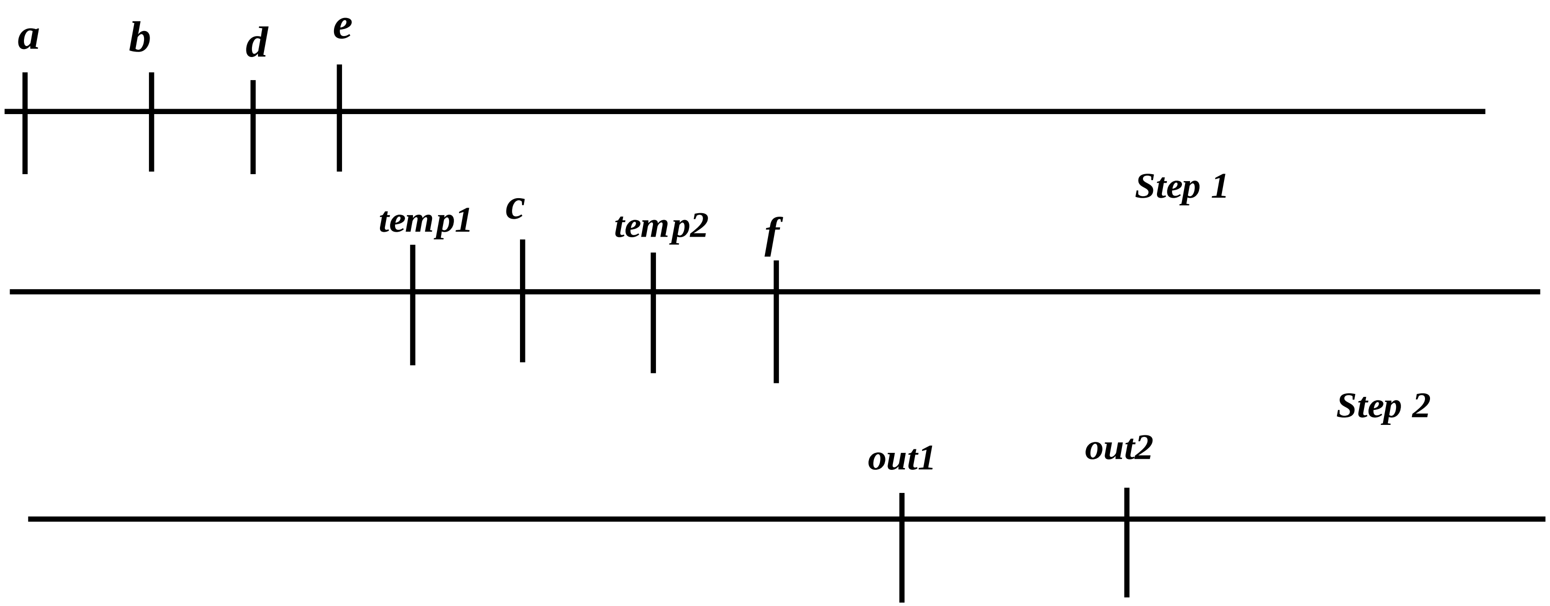
•It may be noted that a maximal clique comprises maximum possible nodes where each of them has an interconnecting edge. Variables (or operations) in a clique can share a resource. If we have maximal cliques then we can have minimal number of modules as more variables (or operations) share a single hardware module.

Binding using Left-Edge Algorithm

•In left edge algorithm, we first short, in ascending order, the variables (or operations) according to the starting step of their life times.

• If there are more than one variable at the same level in the order (because of the same starting control step), then those variables are ordered based on the last control step.

• For example, if there are three variables *a,b,c* where, *a* has life time from step1 to step3, *b* has life time from step1 to step2 and *c* has life time from step2 to step3, then the order is a<b<c. If there are some variables with same start and end control step then they are ordered arbitrarily.

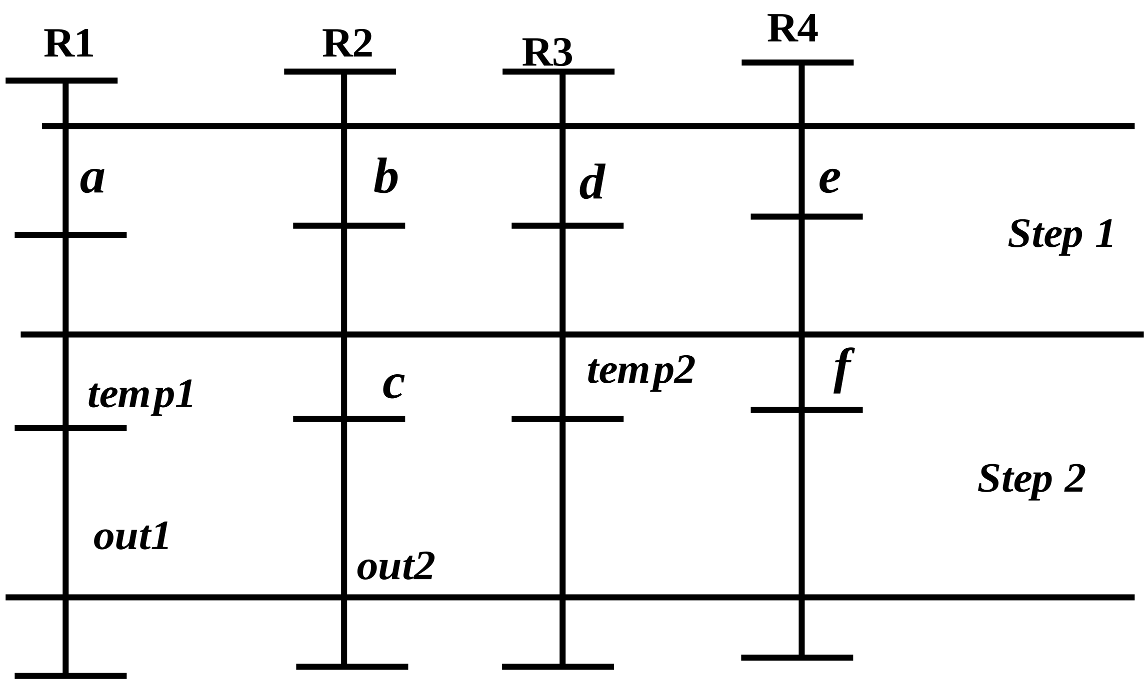


Binding using Left-Edge Algorithm

- Once the variables are arranged, we start with a register and traverse the variables (arranged in order) from left to right.

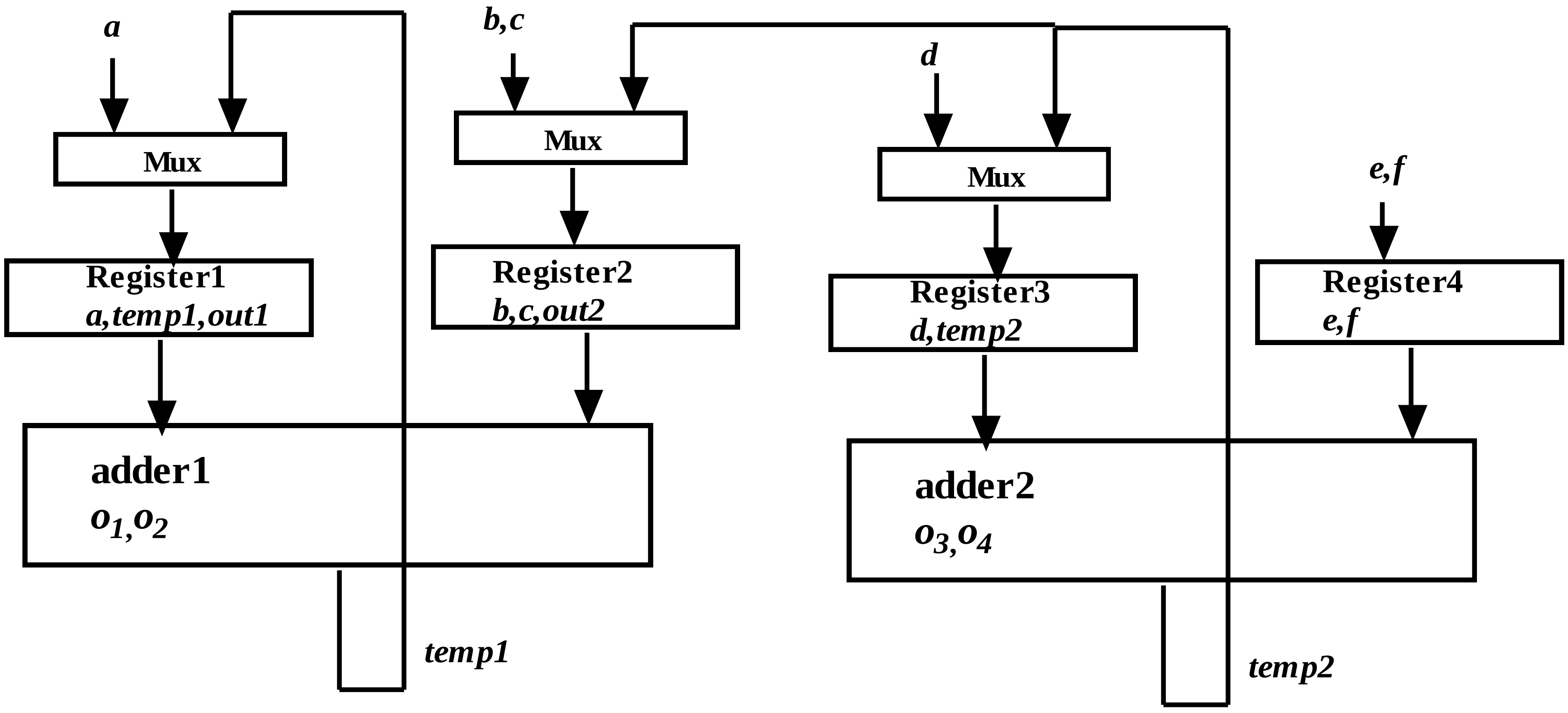
- While traversing, we start filling the register with variables such that there is no overlap in the register. Once the traversal is complete, we delete the variables from the arranged list that are filled in the register.

- If there are variables remaining in the list we take another register and repeat the procedure.



Binding using Left-Edge Algorithm

We take register R1, and in the process of traversal we first start with variable *a*; variable *a* is filled in R1 and it occupies step1 in R1. Following that we traverse variables *b,c,d* but cannot put them in R1 as they would overlap with *a*. Variable *temp1* can be filled in R1 and it occupies step2. Finally variable *out1* is put is R1. As there are more variables, we take another register R2 and repeat the procedure.



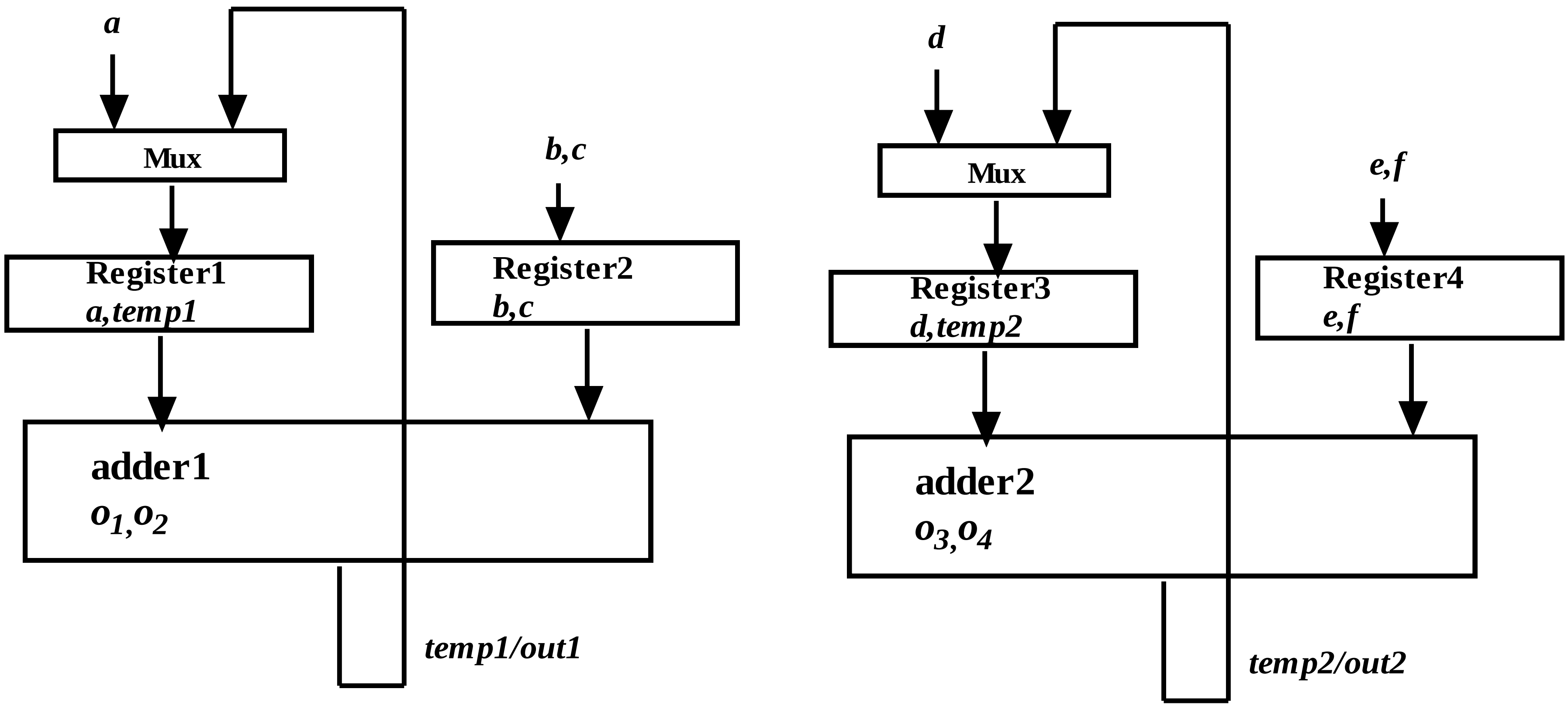
Circuit for the binding

Binding using Iterative Refinement

Binding using iterative refinement, as the name suggests, starts with an arbitrary "feasible" binding and at each step of iteration, variables (or operations) are swapped in between the registers (or operations) such that the new binding remains feasible. If the new binding comprises less interconnect area than the previous one, the new binding replaces the old one. Iteration continues until the interconnect area reaches the desired level or new iterations are not able to improve the area.

For example, we may start with the binding given in last figure. Then we may swap variable *out2* and "NULL" between R2 and R3; this schedule is better than the old one as it requires two multiplexers, while the old one requires three multiplexers. Similarly, we carry on with the iterations by swapping variables until we get the desired interconnect area or we find that there has been no improvement since last few (which can be a user defined threshold) iterations.

Binding using Iterative Refinement



Question and Answer

Question: We know that list scheduling provides optimal binding solution in P-time where as clique partitioning requires exponential time for the same quality of solution. Why, still, clique partitioning is not considered obsolete?

Answer:

While list scheduling provides optimal solution, in terms of resource utilization of variables (i.e., registers) and operations (i.e., operators), in polynomial time, there is no provision of incorporating area of interconnects due to a given binding into the algorithm. However, in case of clique partitioning based solution weights can be assigned to the edges based on area that might result by binding the two operations (or variables) corresponding to the two nodes of the edge under question to a single operator (or register). So most of the area aware binding techniques consider clique partitioning (with required enhancements).

Thank You