Digital System Design & Synthesis
Architectural Synthesis

Dr. Mahdi Abbasi
Computer Engineering Department
Bu Ali Sina University

[Adapted from slides of Prof. G. De Micheli: Synthesis & Optimization of Digital Circuits]

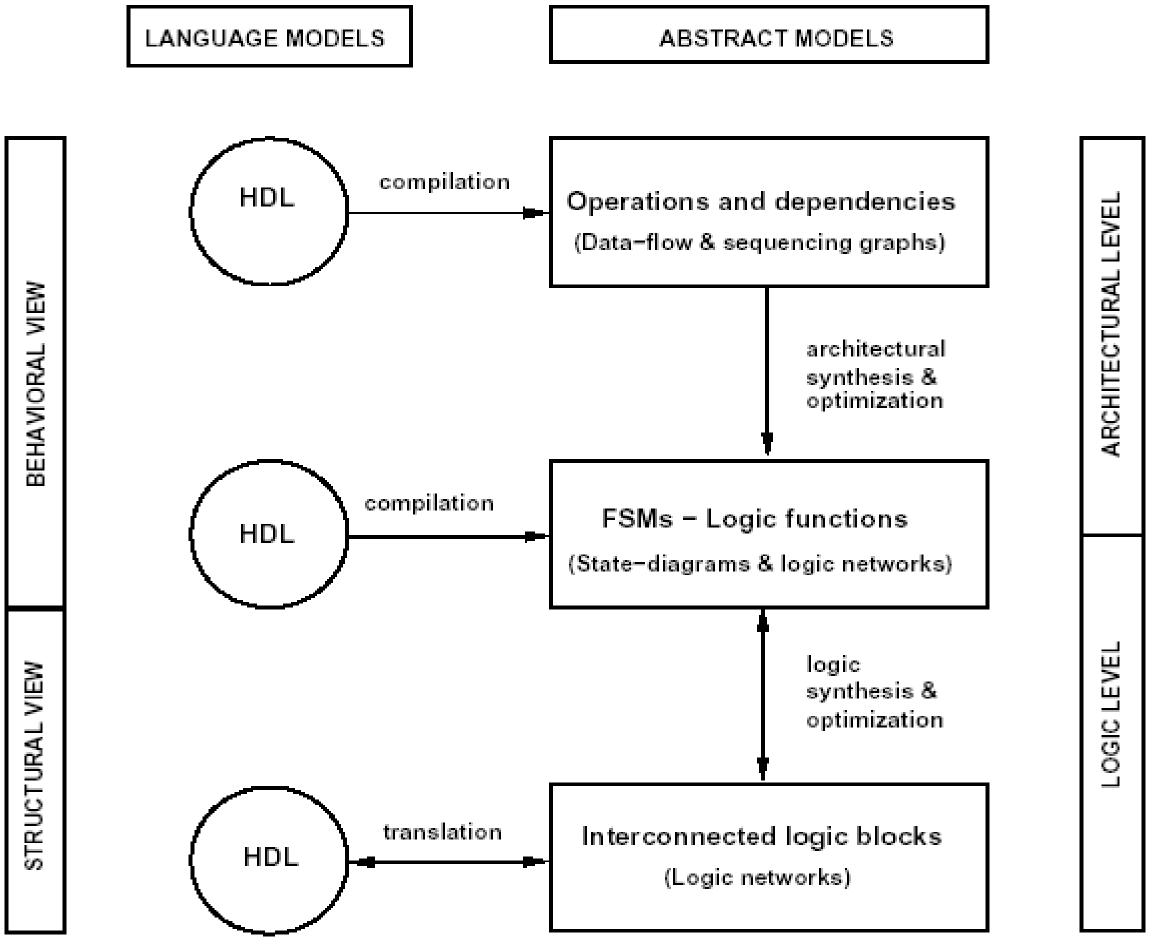
Outline

- Motivation
- Dataflow graphs & Sequencing graphs
- Resources
- Synthesis in temporal domain: Scheduling
- Synthesis in spatial domain: Binding
- Scheduling Models
    - Unconstrained scheduling
    - Scheduling with timing constraints
    - Scheduling with resource constraints
- Algorithmic Solution to the Optimum Binding Problem
- Register Binding Problem

Synthesis

- Transform behavioral into structural view.
- Architectural-level synthesis
  - Architectural abstraction level.
  - Determine macroscopic structure.
  - Example: major building blocks like adder, register, mux.
- Logic-level synthesis
  - Logic abstraction level.
  - Determine microscopic structure.
  - Example: logic gate interconnection.

Synthesis and Optimization



LANGUAGE MODELS
ABSTRACT MODELS
BEHAVIORAL VIEW
STRUCTURAL VIEW
ARCHITECTURAL LEVEL
LOGIC LEVEL
HDL
compilation
Operations and dependencies
(Data–flow & sequencing graphs)
architectural synthesis & optimization
HDL
compilation
FSMs – Logic functions
(State–diagrams & logic networks)
logic synthesis & optimization
HDL
translation
Interconnected logic blocks
(Logic networks)

Architectural-Level Synthesis Motivation

- Raise input abstraction level.
  - Reduce specification of details.
  - Extend designer base.
  - Self-documenting design specifications.
  - Ease of modifications and extensions.
- Reduce design time.
- Explore and optimize macroscopic structure
  - Series/parallel execution of operations.

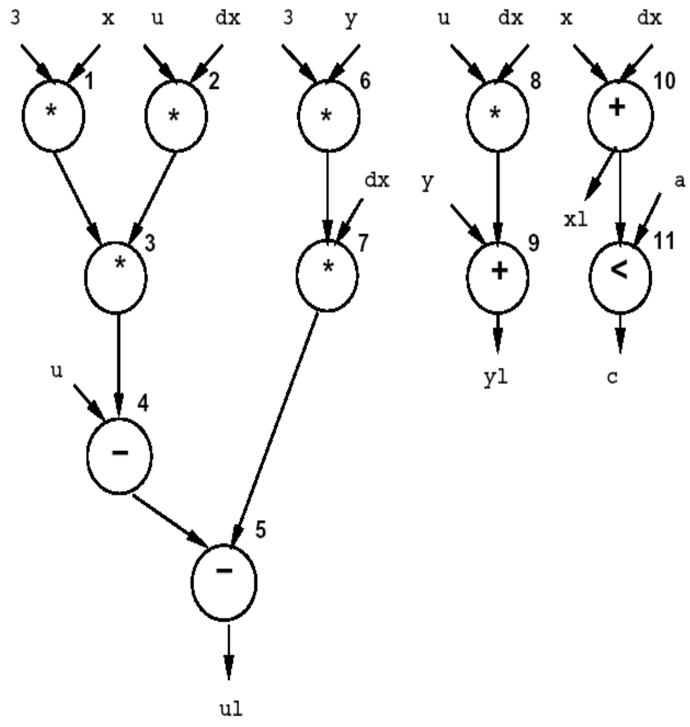
Architectural-Level Synthesis

- Translate HDL models into sequencing graphs.
- Behavioral-level optimization
  - Optimize abstract models independently from the implementation parameters.
- Architectural synthesis and optimization
  - Create macroscopic structure
    - data-path and control-unit.
  - Consider area and delay information of the implementation.

Dataflow Graphs …

- Behavioral views of architectural models.
- Useful to represent data-paths.
- Graph
  - Vertices = operations.
  - Edges = dependencies.
- Dependencies arise due
  - Input to an operation is result of another operation.
  - Serialization constraints in specification.
  - Two tasks share the same resource.

$$\begin{aligned} xl &= x + dx \\ ul &= u - (3 \cdot x \cdot u \cdot dx) - (3 \cdot y \cdot dx) \\ yl &= y + u \cdot dx \\ c &= xl < a \end{aligned}$$

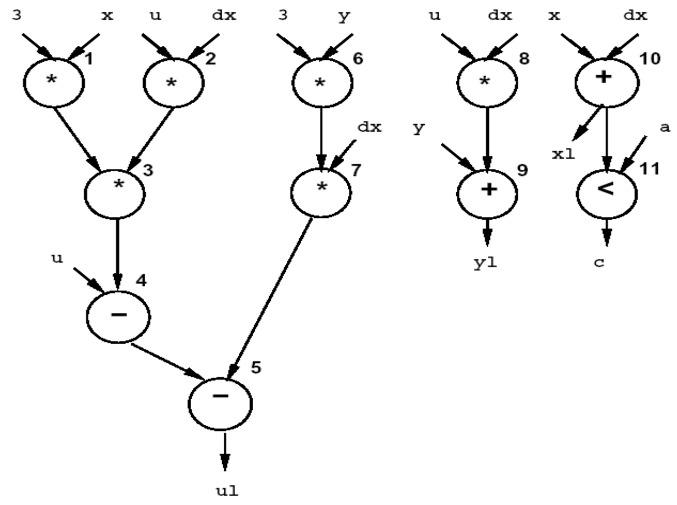


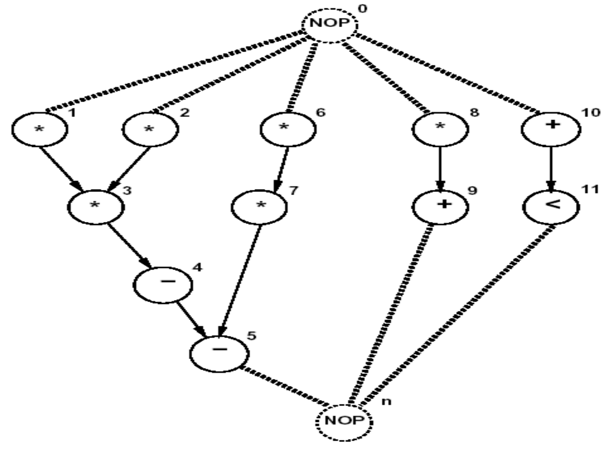
… Dataflow Graphs

- Assumes the existence of variables who store information required and generated by operations.
- Each variable has a *lifetime* which is the interval from *birth* to *death.*
- *Variable birth* is the time at which the value is generated.
- *Variable death* is the latest time at which the value is referenced as input to an operation.
- Values must be preserved during life-time.

Sequencing Graphs

- Useful to represent data-path and control.
- Extended dataflow graphs
  - Control Data Flow Graphs (CDFGs).
  - Polar: source and sink.
  - Operation serialization.
  - Hierarchy.
  - Control-flow commands
    - branching and iteration.
- Paths in the graph represent concurrent streams of operations.

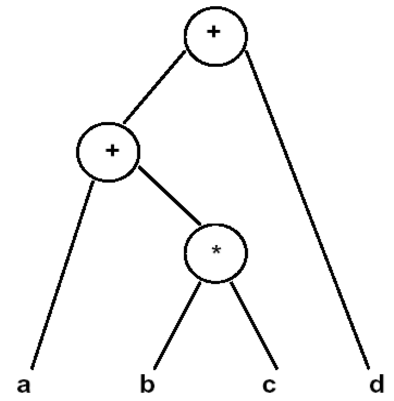




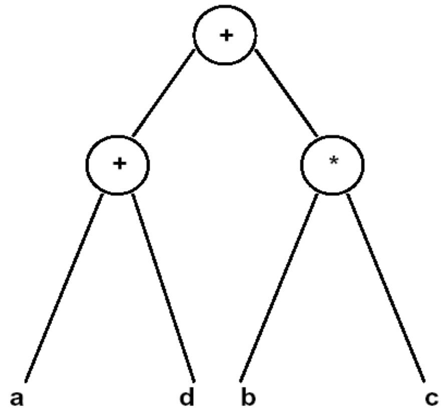
Behavioral-Level Optimization

- Tree-height reduction using commutativity and associativity
- x = a + b * c + d =>   x = (a + d) + b * c

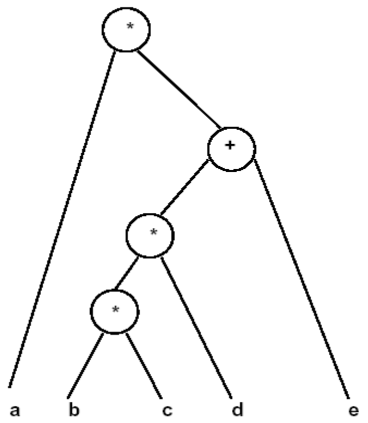
- Tree-height reduction using distributivity
- x = a * (b * c * d + e) => x = a * b * c * d + a * e



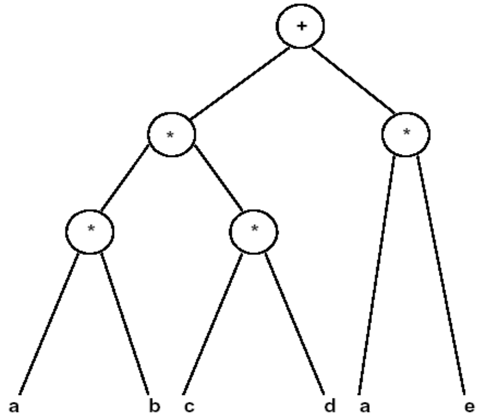
(a)



(b)



(a)



(b)

Architectural Synthesis and Optimization

- Synthesize macroscopic structure in terms of building-blocks.
- Explore area/performance trade-offs
  - maximum performance implementations subject to area constraints.
  - minimum area implementations subject to performance constraints.
- Determine an optimal implementation.
- Create logic model for data-path and control.

Circuit Specification for Architectural Synthesis

- Circuit behavior
  - Sequencing graphs.
- Building blocks
  - Resources.
    - Functional resources: process data (e.g. ALU).
    - Memory resources: store data (e.g. Register).
    - Interface resources: support data transfer (e.g. MUX and Buses).
- Constraints
  - Interface constraints
    - Format and timing of I/O data transfers.
  - Implementation constraints
    - Timing and resource usage.
      - Area
      - Cycle-time and latency

Resources

- Functional resources: perform operations on data.
  - Example: arithmetic and logic blocks.
  - Standard resources
    - Existing macro-cells.
    - Well characterized (area/delay).
    - Example: adders, multipliers, ALUs, Shifters, ...
  - Application-specific resources
    - Circuits for specific tasks.
    - Yet to be synthesized.
    - Example: instruction decoder.
- Memory resources: store data.
  - Example: memory and registers.
- Interface resources
  - Example: busses and ports.

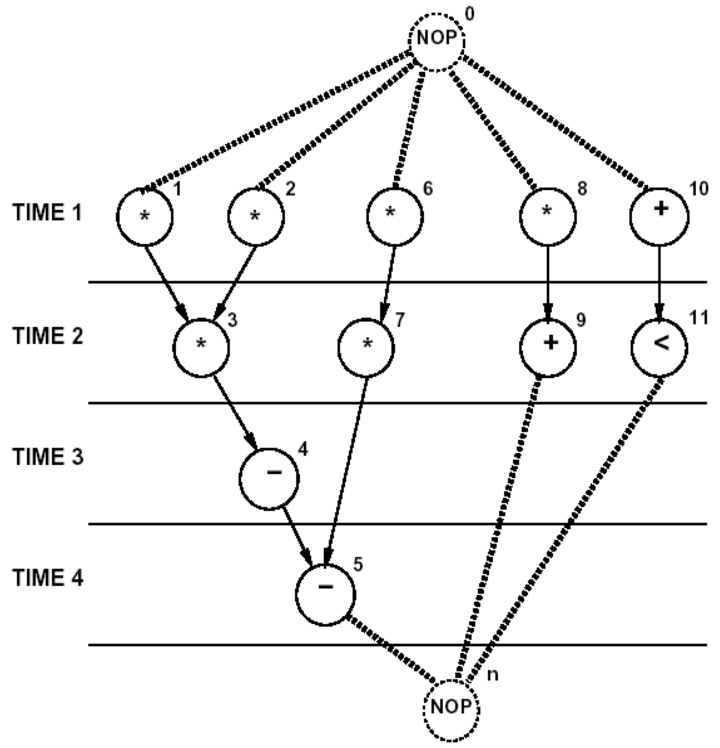
Resources and Circuit Families

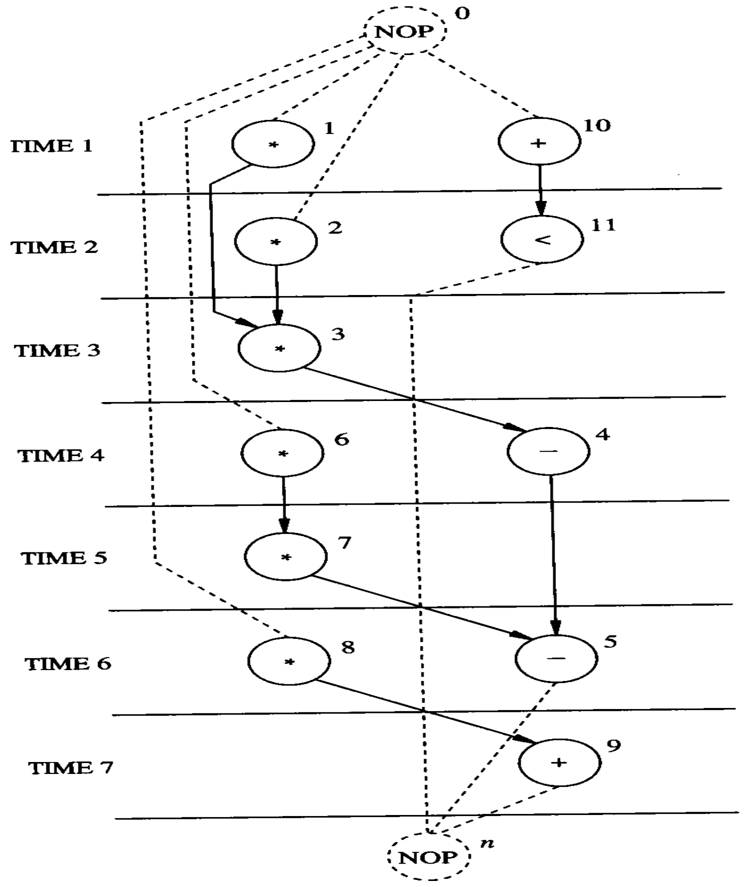
- Resource-dominated circuits
  - Area and performance depend on few, well-characterized blocks.
  - Example: DSP circuits.
- Non resource-dominated circuits
  - Area and performance are strongly influenced by sparse logic, control and wiring.
  - Example: some ASIC circuits.

Synthesis in the Temporal Domain: Scheduling

- Scheduling
  - Associate a start-time with each operation.
  - Satisfying all the sequencing (timing and resource) constraints.
- Goal
  - Determine area/latency trade-off.
  - Determine latency and parallelism of the implementation.
- Scheduled sequencing graph
  - Sequencing graph with start-time annotation.
- Unconstrained scheduling.
- Scheduling with timing constraints.
- Scheduling with resource constraints.

Scheduling …

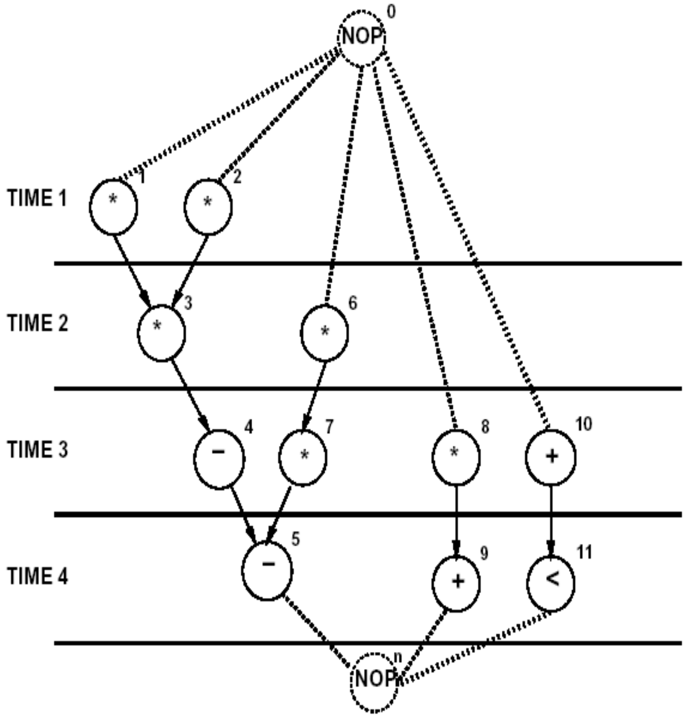




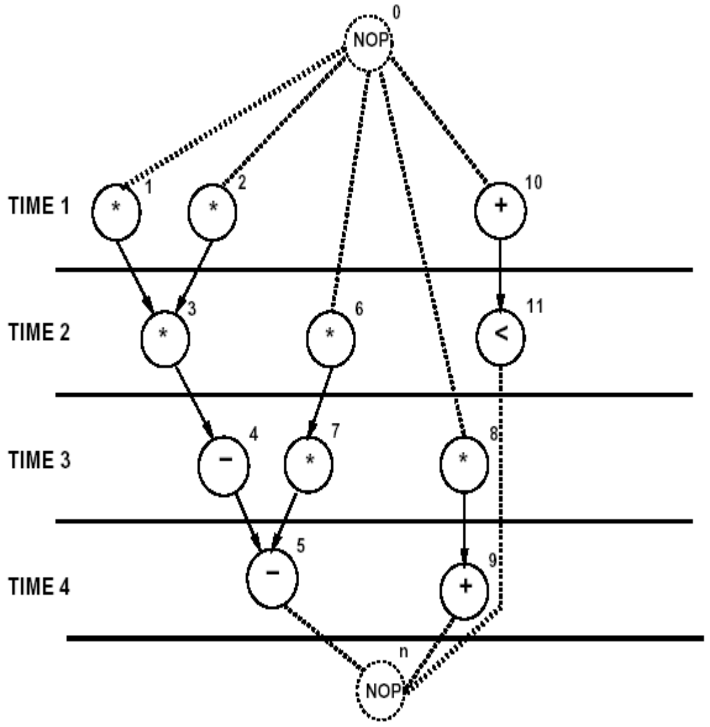
4 Multipliers, 2 ALUs

1 Multiplier , 1 ALU

… Scheduling



2 Multipliers, 3 ALUs

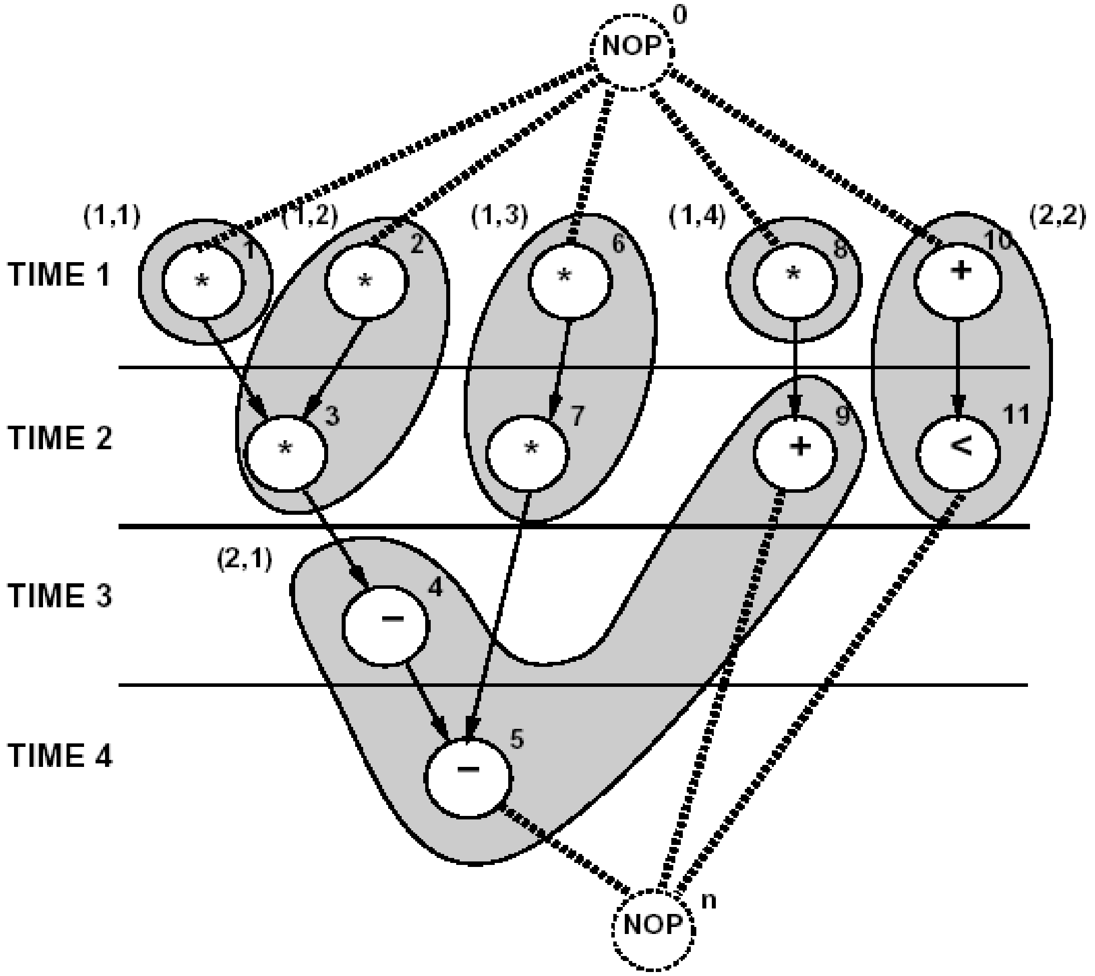


2 Multipliers, 2 ALUs

Synthesis in the Spatial Domain: Binding

- Binding
  - Associate a resource with each operation with the same type.
  - Determine area of the implementation.
- Sharing
  - Bind a resource to more than one operation.
  - Operations must not execute concurrently.
- Bound sequencing graph
  - Sequencing graph with resource annotation.

Example: Bound Sequencing Graph



Performance and Area Estimation

- Resource-dominated circuits
  - Area = sum of the area of the resources bound to the operations.
    - Determined by binding.
  - Latency = start time of the sink operation (minus start time of the source operation).
    - Determined by scheduling
- Non resource-dominated circuits
  - Area also affected by
    - registers, steering logic (routing network), wiring and control.
  - Cycle-time also affected by
    - steering logic (routing network), wiring and (possibly) control.

Scheduling

- Circuit model
  - Sequencing graph.
  - Cycle-time is given.
  - Operation delays expressed in cycles.
- Scheduling
  - Determine the start times for the operations.
  - Satisfying all sequencing (timing and resource) constraints.
- Goal
  - Determine area/latency trade-off.
- Scheduling affects
  - Area: maximum number of concurrent operations of same type is a lower bound on required hardware resources.
  - Performance: concurrency of resulting implementation.

Scheduling Models

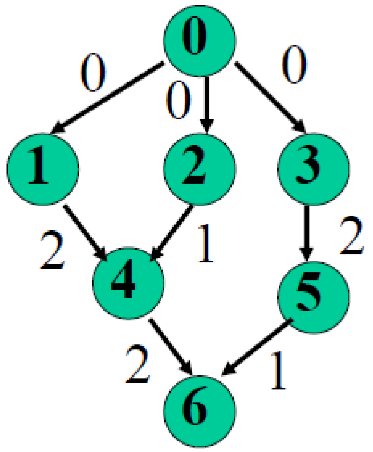
- Unconstrained scheduling.
- Scheduling with timing constraints
  - Latency.
  - Detailed timing constraints.
- Scheduling with resource constraints.
- Simplest scheduling model
  - All operations have bounded delays.
  - All delays are in cycles.
    - Cycle-time is given.
  - No constraints - no bounds on area.
  - Goal
    - Minimize latency.

Some materials:

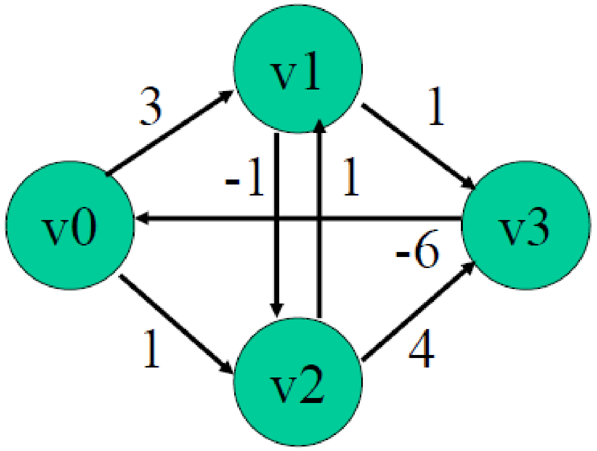
Graphs!

Edge Weighted Graphs

- An edge-weighted graph is a graph $G(V,E)$ together with a weighting function $w: E \to \mathbb{R}$
- We can represent this graphically by annotating each edge $e \in E$ with its weight $w(e)$



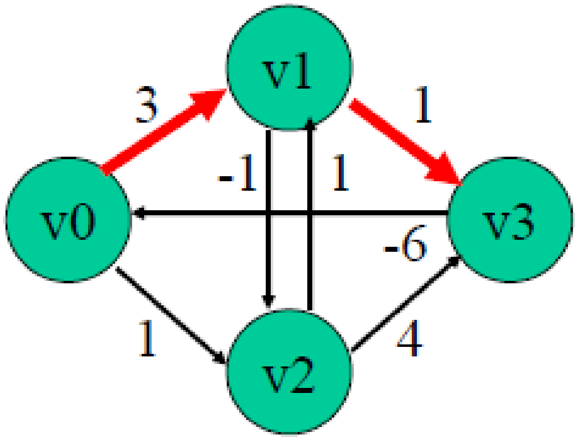
An edge weighted DAG



An edge-weighted graph with cycles

Shortest and longest path

- A *path* through a graph is an alternating sequence of vertices and edges



- A path between vertices v0 and v3, with total edge weight 3+1 = 4 has been highlighted

Shortest and longest path (cont'd)

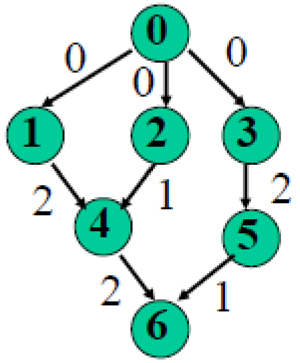
- The *longest path* problem is to find a path of maximum total weight between a given "source" vertex and any other vertex in the graph
  - the shortest path problem is defined similarly
  - we will consider only longest path problems – shortest path can then be achieved by inverting all weights

    $w'(e) = -w(e)$
- Bellman's equations define the total weight of any vertex *v*

$$s_v = \max_{(u,v)\in E}(s_u + w(u,v))$$

longest path through a DAG

- The longest path through a DAG is an easier problem than the equivalent for a general graph
- This is because we can find an order of nodes to visit such that the right-hand side of each Bellman's equation is known
- For our example DAG, let's choose vertex 0 as our source. Then $s_0 = 0$. If we now proceed to apply Bellman's equations in the order $(s_1, s_2, s_3, s_4, s_5, s_6)$, we can determine the total weight for each node
  - $s_1 = 0$, $s_2 = 0$, $s_3 = 0$, $s_4 = 2$, $s_5 = 2$, $s_6 = 4$
- Note that this would not work with an arbitrary order. We must calculate s_v before s_u for all $(v,u) \in E$
- For a graph with cycles, it is not possible to find such an order

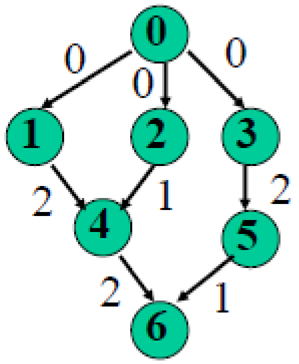


longest path through a DAG - Algorithm

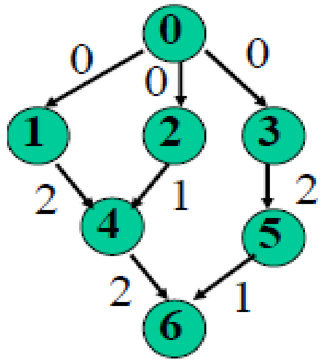
- Below is one possible algorithm (apologies to the recursion-phobics)

```
Algorithm DAG_Longest_Path( G(V,E), source )
    Set $s_{source}$ = 0;
    foreach $v \in V$
        Find_DAG_Path( G(V,E), v );
end DAG_Longest_Path

Algorithm Find_DAG_Path( G(V,E), v )
    if already know $s_v$
        return;
    else
        foreach $(u,v) \in E$
            Find_DAG_Path( G(V,E), u )
        Apply Bellman's equation to find $s_v$
end Find_DAG_Path
```



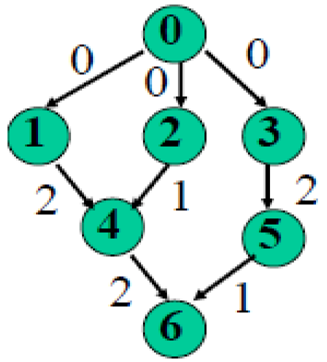
longest path through a DAG - example



- Let's assume the vertices are stored in V in an arbitrary order – say (4,1,2,3,5,0,6)
- A call to **DAG_Longest_Path(G(V,E), 0)** will set s_0 = 0, and then follow the following execution profile

```
1. Find_DAG_Path( G(V,E), 4 )
    1. Find_DAG_Path( G(V,E), 1 )
        1. Find_DAG_Path( G(V,E), 0 )
        2. Calculate $s_1$ = 0
    2. Find_DAG_Path( G(V,E), 2 )
        1. Find_DAG_Path( G(V,E), 0 )
        2. Calculate $s_2$ = 0
    3. Calculate $s_4$ = 2
```

longest path through a DAG - example



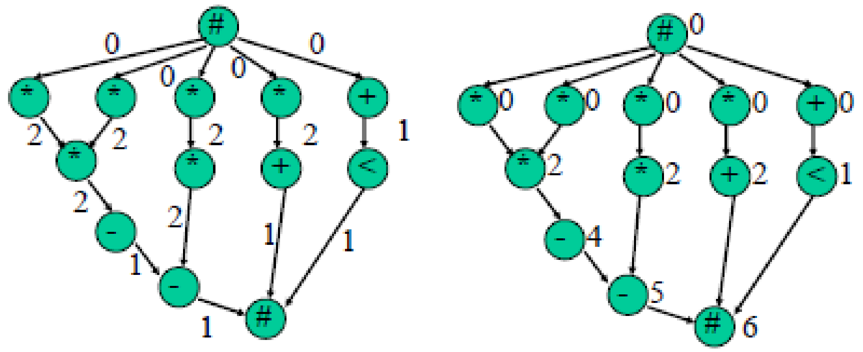
```
2. Find_DAG_Path( G(V,E), 1 )
3. Find_DAG_Path( G(V,E), 2 )
4. Find_DAG_Path( G(V,E), 3 )
    1. Find_DAG_Path( G(V,E), 0 )
    2. Calculate $s_3$ = 0
5. Find_DAG_Path( G(V,E), 5 )
    1. Find_DAG_Path( G(V,E), 3 )
    2. Calculate $s_5$ = 2
6. Find_DAG_Path( G(V,E), 0 )
7. Find_DAG_Path( G(V,E), 6 )
    1. Find_DAG_Path( G(V,E), 4 )
    2. Find_DAG_Path( G(V,E), 5 )
    3. Calculate $s_6$ = 4
```

ASAP Scheduling algorithm

- The simplest type of scheduling occurs when we wish to optimize the overall latency of the computation and do not care about the number of resources required
- This can be achieved by simply starting each operation in a CDFG as soon as its predecessors have completed
- This strategy gives rise to the name ASAP for "As Soon As Possible"

ASAP Scheduling algorithm

- Let's label each edge in the CDFG with the latency of the node producing that edge
- Then scheduling under ASAP is equivalent to finding the longest path between each operation and the source node
- Since a CDFG is a DAG, we can use the DAG longest path algorithm
- Consider the original example and assume that multiplication takes two cycles, whereas addition and comparison take one cycle



- Applying the DFG algorithm to finding the longest path between the start and end nodes leads to the scheduled start times on the right-hand diagram

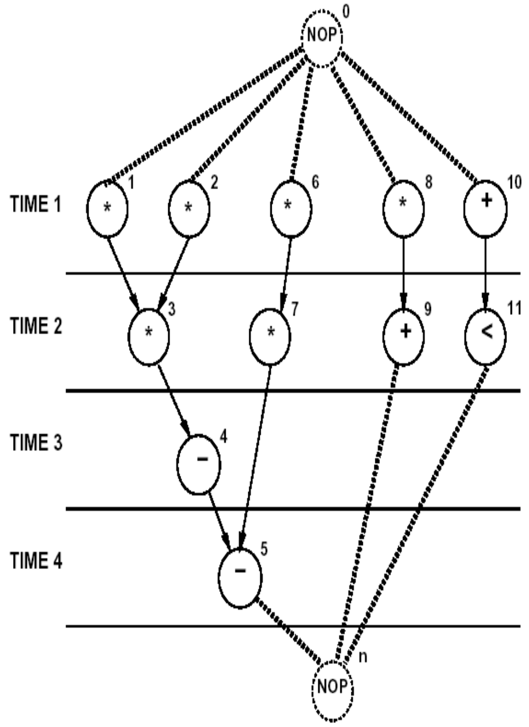
Minimum-Latency Unconstrained Scheduling Problem

- Given a set of operations V with integer delays D and a partial order on the operations E
- Find an integer labeling of the operations $\varphi : V \rightarrow Z^+$, such that
  - $t_i = \varphi(v_i)$,
  - $t_i \geq t_j + d_j \quad \forall\, i, j$ s.t. $(v_j, v_i) \in E$
  - and t_n is *minimum*.
- Unconstrained scheduling used when
  - Dedicated resources are used.
  - Operations differ in type.
  - Operations cost is marginal when compared to that of steering logic, registers, wiring, and control logic.
  - Binding is done before scheduling: resource conflicts solved by serializing operations sharing same resource.
  - Deriving bounds on latency for constrained problems.

ASAP Scheduling Algorithm

- Denote by *t^s* the start times computed by the *as soon as possible (ASAP)* algorithm.
- Yields *minimum* values of start times.

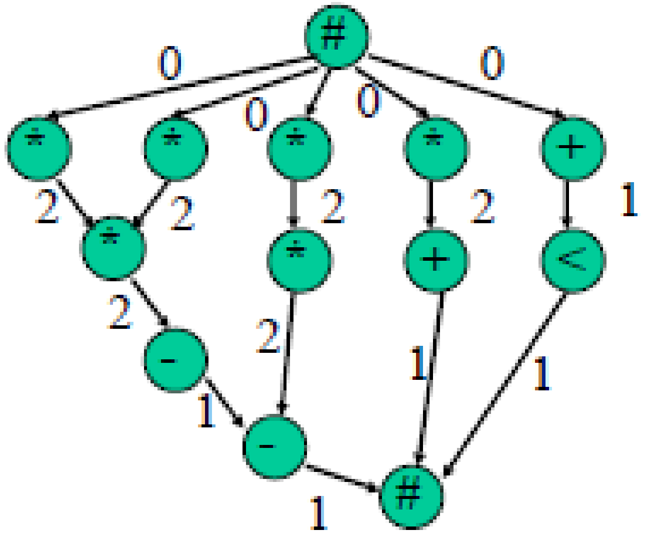
```
ASAP ( $G_s(V,E)$) {
    Schedule $v_0$ by setting $t_0^S = 1$;
    repeat {
        Select a vertex $v_i$ whose pred. are all scheduled;
        Schedule $v_i$ by setting $t_i^S = \max_{j:(v_j,v_i)\in E} t_j^S + d_j$;
    }
    until ($v_n$ is scheduled) ;
    return ($t^S$);
}
```



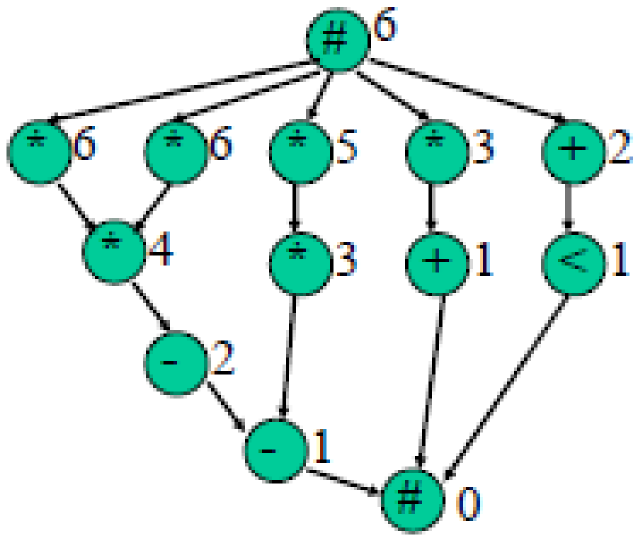
ALAP Scheduling

- The ASAP algorithm schedules each operation at the earliest opportunity. Given an overall latency constraint, it is equally possible to schedule operations at the latest opportunity.
- This leads to the concept of As-Late-As-Possible (ALAP) scheduling.
- ALAP scheduling can be performed by seeking the longest path between each operation and the end or "sink" node.
- We will re-examine the example, under the same delay assumptions, with an overall latency constraint of 6 clock cycles.

ALAP Scheduling



Edge-weighted CDFG

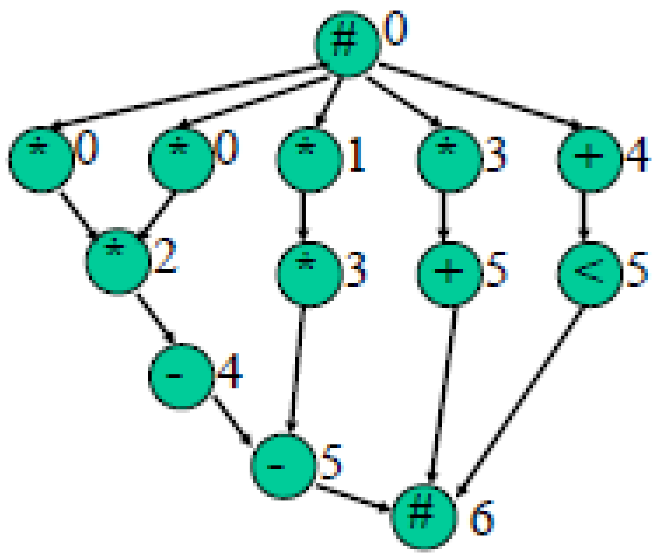


Longest paths to sink node

- The ALAP schedule start times can be derived by subtracting the longest path time from the desired overall latency constraint

ALAP Scheduling

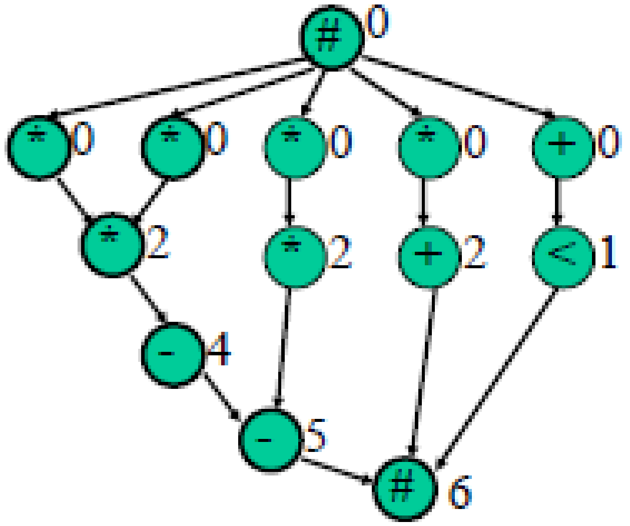
ALAP Scheduling

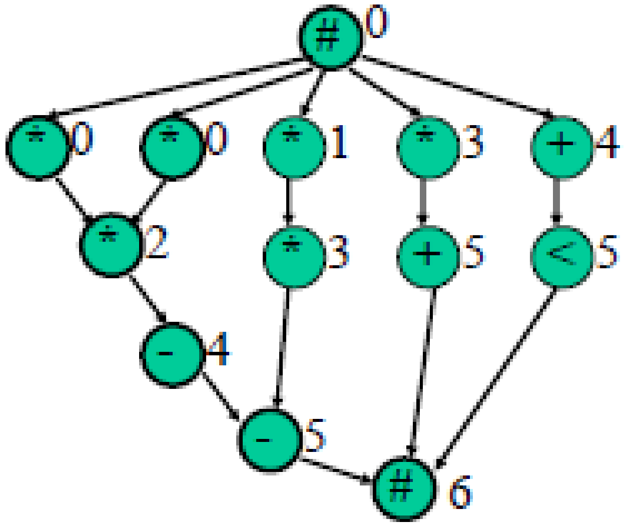


- Here are the ALAP start times. You can see that each operation starts at the latest opportunity possible to still meet 6 cycles overall

ALAP Scheduling

- Let's compare the ASAP and ALAP schedules:



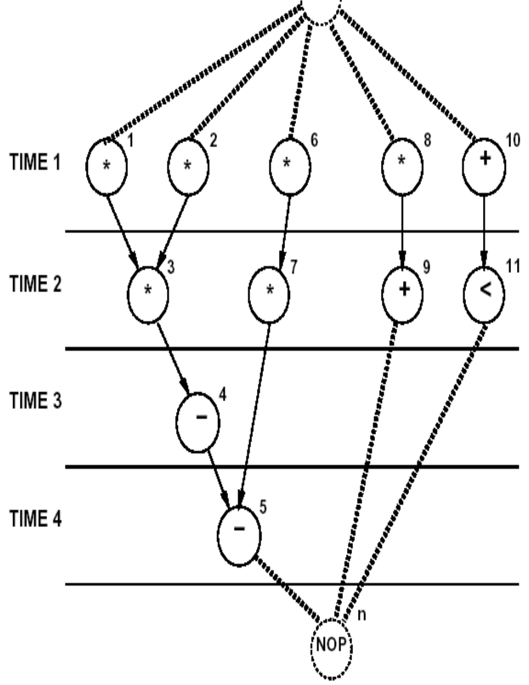


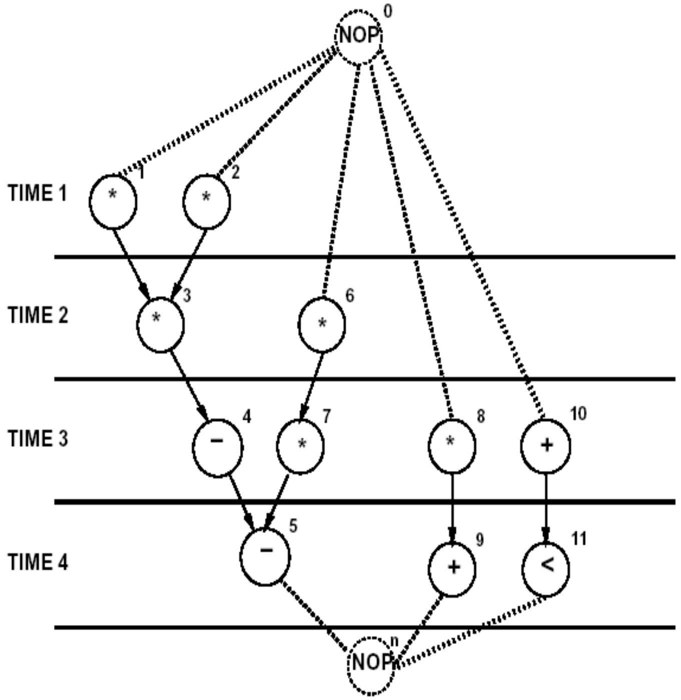
- The highlighted nodes have equal ASAP and ALAP times. For all others there is a difference of at least once cycle.

ALAP Scheduling Algorithm

- Denote by t^L the start times computed (ALAP) algorithm.
- Yields *maximum* values of start times.
- Latency upper bound λ

```
ALAP( $G_s(V,E),\bar{\lambda}$) {
    Schedule $v_n$ by setting $t_n^L = \bar{\lambda} + 1$;
    repeat {
        Select vertex $v_i$ whose succ. are all scheduled;
        Schedule $v_i$ by setting $t_i^L = \min_{j:(v_i,v_j)\in E} t_j^L - d_i$ ;
    }
    until ($v_0$ is scheduled) ;
    return ($t^L$);
}
```





ALAP Scheduling- Mobility

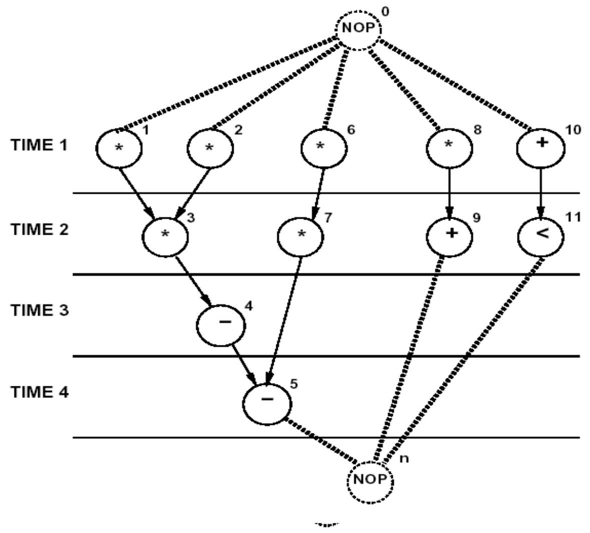
- The difference between the ALAP and ASAP times for an operation is called the *operation mobility* or *slack*.
- Mobility measures how free we are to move the operation into different time-slots.
- Operations with zero mobility are *critical operations*, and together form the *critical path*, which determines how fast our circuit will run.
- More sophisticated scheduling algorithms will take advantage of positive mobility to balance the resource requirements over time.

Latency-Constrained Scheduling

- ALAP solves a latency-constrained problem.
- Latency bound can be set to latency computed by ASAP algorithm.
- Mobility
  - Defined for each operation.
  - Difference between ALAP and ASAP schedule.
  - Zero mobility implies that an operation can be started only at one given time step.
  - Mobility greater than 0 measures span of time interval in which an operation may start.
- Slack on the start time.

Example

- Operations with zero mobility
  - **{v1, v2, v3, v4, v5}.**
  - **Critical path.**
- Operations with mobility one
  - **{v6, v7}.**
- Operations with mobility two
  - **{v8, v9, v10, v11}**



Scheduling under Resource Constraints

- Classical scheduling problem.
  - Fix area bound - minimize latency.
- The amount of available resources affects the achievable latency.
- Dual problem
  - Fix latency bound - minimize resources.
- Assumption
  - All delays bounded and known.

Minimum Latency Resource-Constrained Scheduling Problem

- Given a set of ops V with integer delays D, a partial order on the operations E, and upper bounds $\{a_k; k = 1, 2, \dots, n_{res}\}$
- Find an integer labeling of the operations $\varphi : V \rightarrow Z^+$, such that
  - $t_i = \varphi(v_i)$,
  - $t_i \geq t_j + d_j \quad \forall$ i, j s.t. $(v_j, v_i) \in E$

$$|\{v_i | \mathcal{T}(v_i) = k \text{ and } t_i \leq l < t_i + d_i\}| \leq a_k \qquad \mathcal{T} : V \rightarrow \{1,2, \dots n_{res}\}$$
$$\forall \text{types } k = 1, 2, \dots, n_{res} \text{ and } \forall \text{ steps } l$$

- Number of operations of any given type in any schedule step does not exceed bound.

Scheduling under Resource Constraints

- Intractable problem.
- Algorithms
  - Exact
    - Integer linear program.
  - Approximate
    - List scheduling.

Types of timing constraints

- As well as an overall latency constraint, other types of timing constraint are important
- Consider these examples [DeMicheli94]
  - A circuit reads data from a bus, performs a computation, and writes the result back onto the bus. The bus interface specifies that the data must be written exactly three cycles after the read
  - A circuit has two independent streams of operations, constrained to communicate simultaneously to external circuits by providing two pieces of data at two interfaces. The cycle in which the data are made available is irrelevant, although the simultaneity of the data is essential.

Types of timing constraints

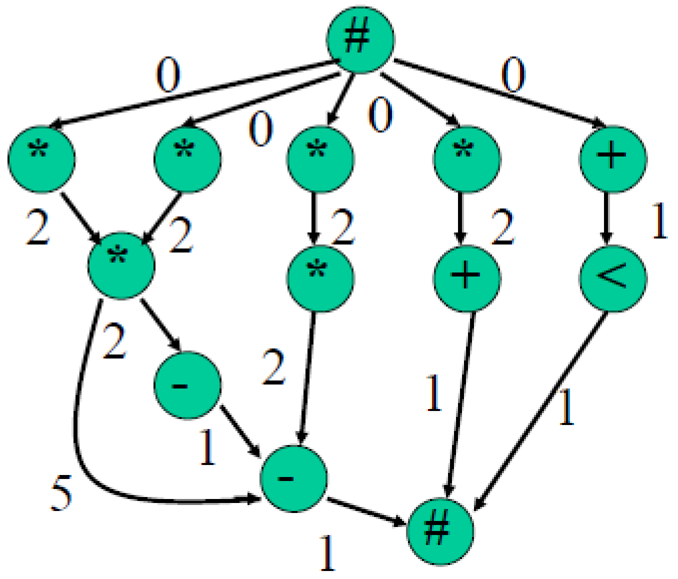
- We will consider two types of constraint
  - a minimum timing constraint l_{ij} between operations v_i and v_j: $S(v_j) \geq S(v_i) + l_{ij}$
  - a maximum timing constraint u_{ij} between operations v_i and v_j : $S(v_j) \leq S(v_i) + u_{ij}$
- These constraints are sufficient to model the situations on the previous slide, in addition to many others. Solutions for previous slide:
  - set both min and max of 3 cycles between read and write
  - set both min and max of 0 cycles between the two writes

Modelling timing constraints

- How can we incorporate these timing constraints within our sequencing graph-based model, and how do they affect the schedule?
- From the sequencing graph $G(V,E)$, we construct an edge-weighted *constraint graph* $G_C(V,E_C)$, where $E \subset E_C$:
  - the edge weights for edges in E are the same as before (i.e. the delay of the node producing that edge)
  - we add extra edges to model the timing constraints

Modelling timing constraints

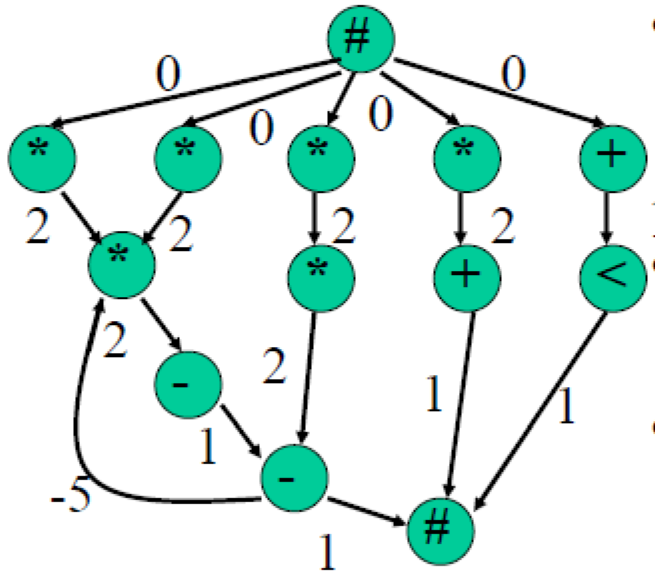
- Minimum timing constraints can simply be modelled by adding an extra edge (v_i, v_j) with weight l_{ij}



- By adding the curved edge with weight 5, the subtraction operation cannot start for at least 5 cycles after the multiplication starts

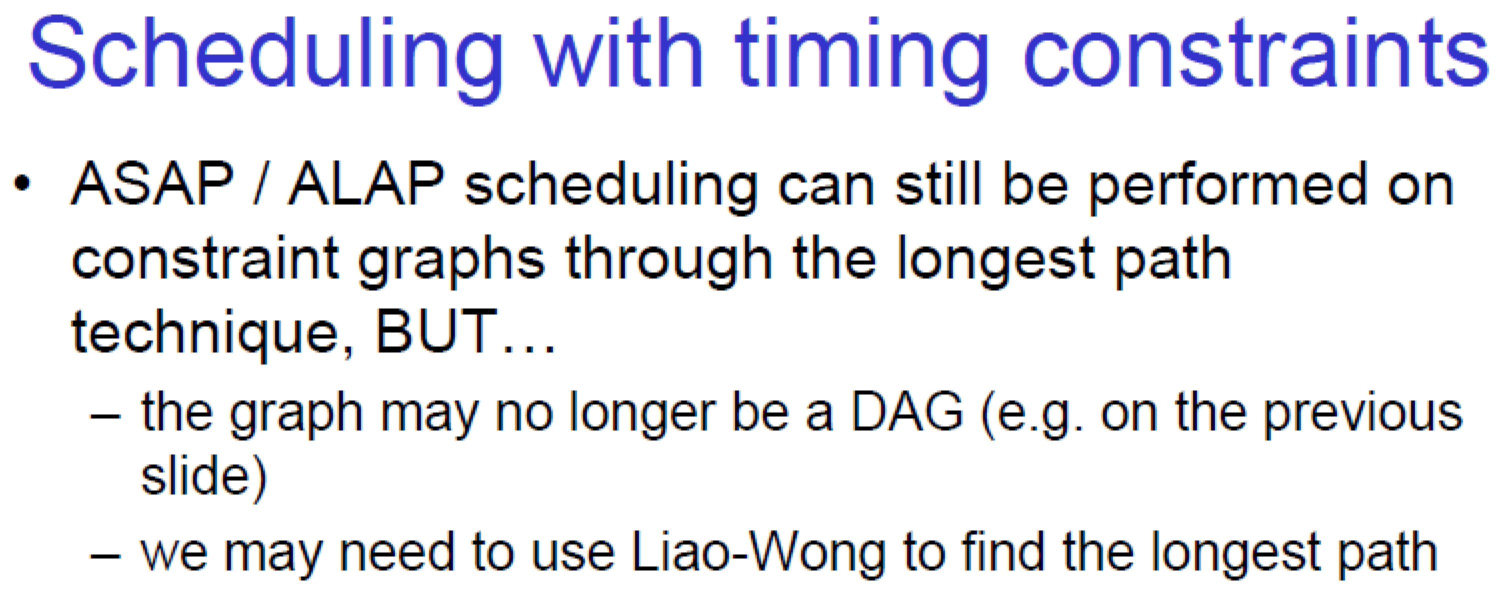
Modelling timing constraints

- Maximum timing constraints can be modelled by adding an extra edge (v_j, v_i) with weight -u_{ij}



- Now the multiplication cannot occur before -5 cycles after the subtraction starts
- S(mult) ≥ S(sub) – 5, i.e. S(sub) ≤ S(mult) + 5
- The subtraction cannot occur later than five cycles after the multiplication starts

Modelling timing constraints



Scheduling with timing constraints

- ASAP / ALAP scheduling can still be performed on constraint graphs through the longest path technique, BUT…
  - the graph may no longer be a DAG (e.g. on the previous slide)
  - we may need to use Liao-Wong to find the longest path

Resource Constrained Scheduling

- The following problem is given the name “resource constrained scheduling”:
  - Given a library of resources, and a constraint on the maximum number of each type of resource to be used in the implementation, find a schedule of minimum latency

Resource Constrained Scheduling

- Let R denote the set of resource types,
  - e.g. R = {add, mult, ALU}
- Let the bound on the number of each resource type $r \in R$ be a_r
- In list scheduling, we schedule operations by considering each clock-cycle in turn
  - $U_{t,r}$ is used to denote the set of operations of type r whose predecessors have already completed by cycle t – the candidate set
  - $T_{t,r}$ is used to denote the set of operations of type r started, but not completed by cycle t

Resource Constrained Scheduling- Algorithm

```
Algorithm RC_ListSchedule( $G(V,E)$, $R$, $a$ ) {
  set $t = 0$;
  repeat {
    foreach $r \in R$ {
      determine $U_{t,r}$;
      determine $T_{t,r}$;
      select $Y \subseteq U_{t,r}$ s.t. $|Y| + |T_{t,r}| \leq a_r$;
      set $S(v) = t$ for all $v \in Y$;
    }
    set $t = t+1$;
  } until all nodes scheduled
  return( $S$ );
}
```

Resource Constrained Scheduling- Algorithm

- At each clock cycle, the candidate set represents those operations we *could* schedule
- From the candidate set, we select a subset *Y*, which we *do* schedule
- The constraint on selection of *Y* is that we can never have more than a_r operations of type r executing simultaneously

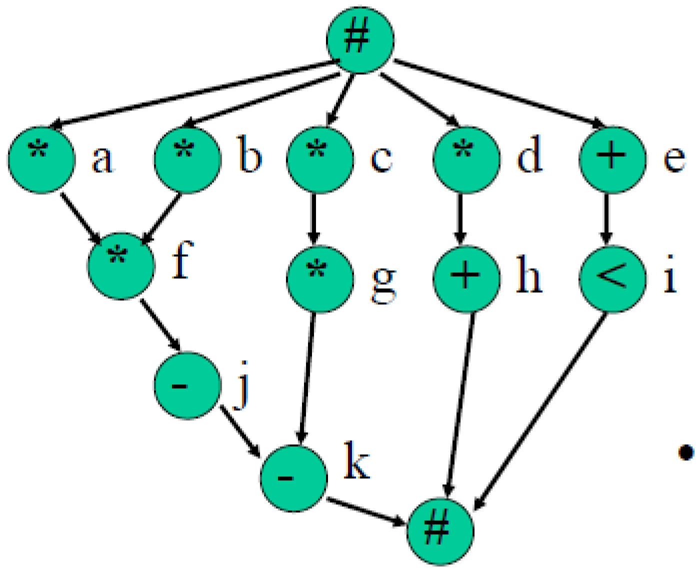
- Notice that as $a_r \to \infty$ for all $r \in R$, the list schedule approaches an ASAP schedule

Resource Constrained Scheduling- Algorithm

- Notice that the algorithm is not fully defined, as we haven’t said how to pick *Y*
- The most common way to pick *Y* is to prefer to schedule the most urgent operations first
- Urgency is typically defined in terms of the minimum latency ALAP schedule time – the lower the ALAP time, the more urgent the operation is

Resource Constrained Scheduling- Algorithm example

- Let's re-visit our familiar differential equation example



- Consider scheduling under the resource set $R = \{*, +/-, <\}$, where the delay of +/- and < is 1 cycle, and the delay of * is 2 cycles

- We will perform a list-schedule with $a_{*}=2$, $a_{+/-}=2$, $a_{<}=1$

Resource Constrained Scheduling- Algorithm example

- $t = 0$
  - $U_{0,*} = \{a,b,c,d\}$, $U_{0,+/-} = \{e\}$, $U_{0,<} = \varnothing$
  - $T_{0,*} = \varnothing$, $T_{0,+/-} = \varnothing$, $T_{0,<} = \varnothing$
  - For +/-, easy to select $Y = \{e\}$
  - For *, we have a choice. ALAP times for a,b,c,d are 0,0,1,3, respectively (see Lecture 9). So most urgent are $Y = \{a,b\}$
  - For <, there is nothing to schedule $Y = \varnothing$
  - $S(a) = 0$, $S(b) = 0$, $S(e) = 0$

Resource Constrained Scheduling- Algorithm example

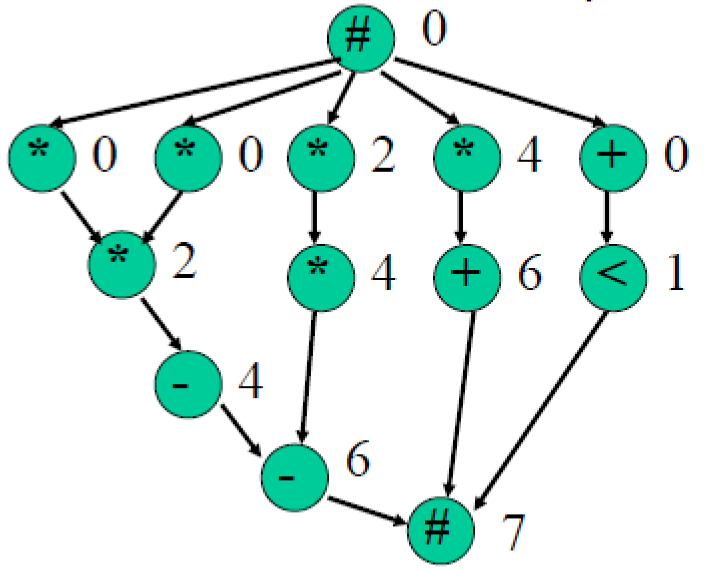
- $t = 1$
  - $U_{1,*} = \{c,d\}$, $U_{1,+/-} = \varnothing$, $U_{1,<} = \{i\}$
  - $T_{1,*} = \{a,b\}$, $T_{1,+/-} = \varnothing$, $T_{1,<} = \varnothing$
  - For +/-, $Y = \varnothing$
  - For *, $Y = \varnothing$ (all resources busy)
  - For <, $Y = \{i\}$
  - $S(i) = 1$

Resource Constrained Scheduling- Algorithm example

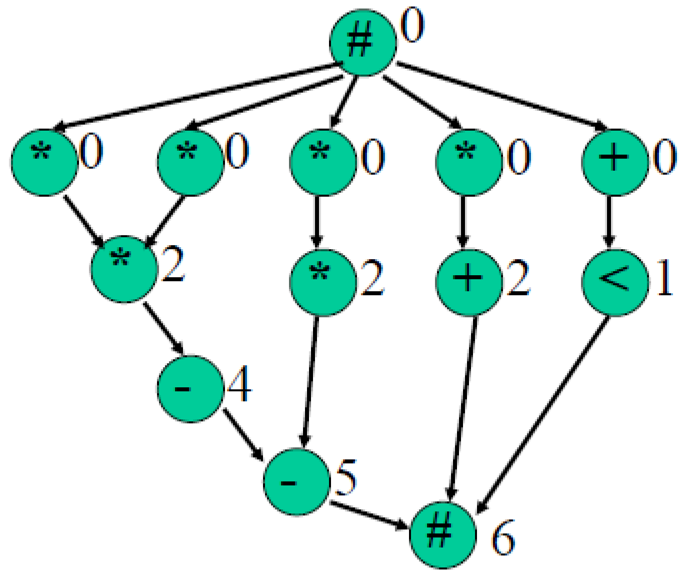
- t = 2
  - $U_{2,*}$ = {c,d,f}, $U_{2,+/-} = \varnothing$, $U_{2,<} = \varnothing$
  - $T_{2,*} = \varnothing$, $T_{2,+/-} = \varnothing$, $T_{2,<} = \varnothing$
  - For +/-, $Y = \varnothing$
  - For *, ALAP times for c,d,f are 1,3,2 respectively. Y = {c,f}
  - For <, $Y = \varnothing$
  - S(c) = 2, S(f) = 2

Resource Constrained Scheduling- Algorithm example

- If we continue this process until the algorithm terminates
  - we take once cycle longer than ASAP (but can use half the number of multipliers)



List-scheduled times



ASAP times

Latency Constrained Scheduling

- The dual problem is “latency constrained scheduling”:
  - Given a library of resources, and a constraint on the maximum overall latency of the schedule, find a schedule using the minimum number of resources of each type
- This problem is also NP-hard (the same proof holds), so again heuristics are used to attack the problem
- Let λ denote the desired maximum latency

Latency Constrained Scheduling

```
Algorithm LC_ListSchedule( $G(V,E)$, $R$, $\lambda$ ) {
  perform ALAP( G(V,E), $\lambda$ );
  set $a_r = 1$ for all $r \in R$;
  set t = 0;
  repeat {
    foreach $r \in R$ {
      determine $U_{t,r}$;
      determine $T_{t,r}$;
      determine slack $s_v = ALAP_v - t$ for all $v \in U_{tr}$;
      set $Y_1 = \{v \in V: s_v = 0\}$;
      set $a_r = \max( a_r , |Y_1| + |T_{tr}| )$;
      select $Y_2 \subseteq U_{tr}$, s.t. $|Y_1 \cup Y_2| + |T_{tr}| \leq a_r$;
      set $S(v) = t$ for all $v \in Y_1 \cup Y_2$;
    }
    set $t = t+1$;
  } until all nodes scheduled
  return( $S$, $a$ );
}
```

Latency Constrained Scheduling

- This algorithm works by constantly refining the "maximum" number of resources it allows
  - we start with one resource of each type
  - this is changed if the desired latency is not achievable
- For each cycle, we calculate the *slack* of the candidate operations
  - slack is the difference between the last cycle an operation could be scheduled in and the current cycle
  - if the slack of an operation is zero, it must clearly be scheduled immediately, even if that means increasing the number of resources allowed

Latency Constrained Scheduling

- Such “forced” scheduled nodes are placed in set Y_1
- It may also be possible to schedule additional nodes, without increasing the resource requirements further. These are placed in Y_2, and selected on the basis of urgency, as with the resource-constrained algorithm

Latency Constrained Scheduling- example

- As an example, we will again consider the differential equation CDFG
  - The ASAP schedule gave a minimum schedule length of 6 cycles. It had up to 4 "*", 1 "+" and 1 "<" operating in parallel
  - Let's see whether latency constrained list scheduling can do better than that
- We will execute LC_ListSchedule(G(V,E), R, 6)
- The ALAP times for this example have already been determined in Lecture 9, and are:
  - a: 0, b: 0, c: 1, d: 3, e: 4, f: 2, g: 3, h: 5, i: 5, j: 4, k: 5

Latency Constrained Scheduling- example

- t = 0
  - $U_{0,*} = \{a,b,c,d\}$, $U_{0,+/-} = \{e\}$, $U_{0,<} = \varnothing$
  - $T_{0,*} = \varnothing$, $T_{0,+/-} = \varnothing$, $T_{0,<} = \varnothing$
  - $s_a = 0$, $s_b = 0$, $s_c = 1$, $s_d = 3$, $s_e = 4$
  - For *, $Y_1 = \{a,b\}$; for +/-, $Y_1 = \varnothing$; for <, $Y_1 = \varnothing$
  - $a_* = 2$; others unchanged
  - For *, $Y_2 = \varnothing$; for +/-, $Y_2 = \{e\}$; for <, $Y_2 = \varnothing$
  - $S(a) = 0$, $S(b) = 0$, $S(e) = 0$

Latency Constrained Scheduling- example

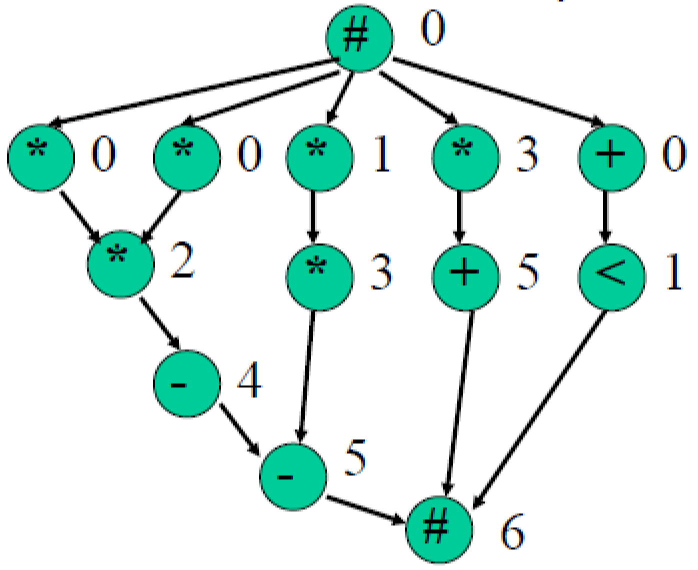
- t = 1
  - $U_{1,*}$ = {c,d}, $U_{1,+/-} = \emptyset$, $U_{1,<}$ = {i}
  - $T_{1,*}$ = {a,b}, $T_{1,+/-} = \emptyset$, $T_{1,<} = \emptyset$
  - $s_c = 0$, $s_d = 2$, $s_i = 4$
  - For *, Y_1 = {c}; for +/-, $Y_1 = \emptyset$; for <, $Y_1 = \emptyset$
  - $a_* = 3$; others unchanged
  - For *, $Y_2 = \emptyset$; for +/-, $Y_2 = \emptyset$; for <, Y_2 = {i}
  - S(c) = 1, S(i) = 1

Latency Constrained Scheduling- example

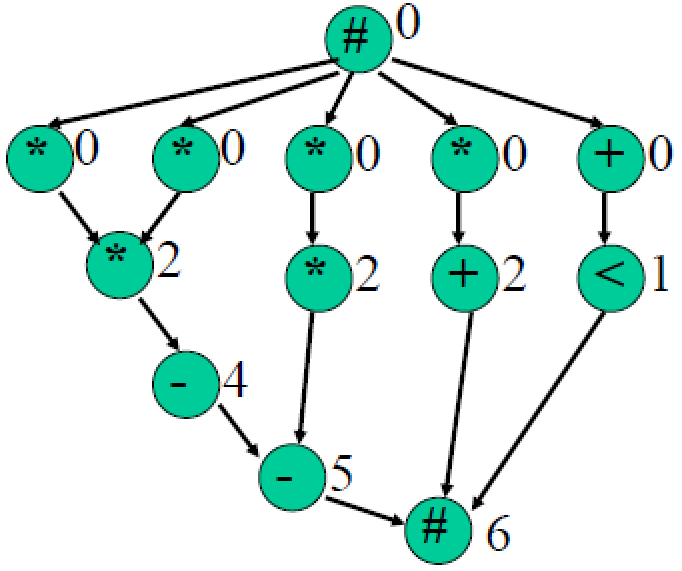
- t = 2
  - $U_{2,*}$ = {f,d}, $U_{2,+/-} = \varnothing$, $U_{2,<} = \varnothing$
  - $T_{2,*}$ = {c}, $T_{2,+/-} = \varnothing$, $T_{2,<} = \varnothing$
  - $s_f = 0$, $s_d = 1$
  - For *, Y_1 = {f}; for +/-, $Y_1 = \varnothing$; for <, $Y_1 = \varnothing$
  - all resource constraints unchanged
  - For *, Y_2 = {d}; for +/-, $Y_2 = \varnothing$; for <, $Y_2 = \varnothing$
  - S(f) = 2, S(d) = 2

Latency Constrained Scheduling- example

- If we continue this process until the algorithm terminates
  - schedule has the same latency as ASAP, but requires 3 rather than 4 multipliers



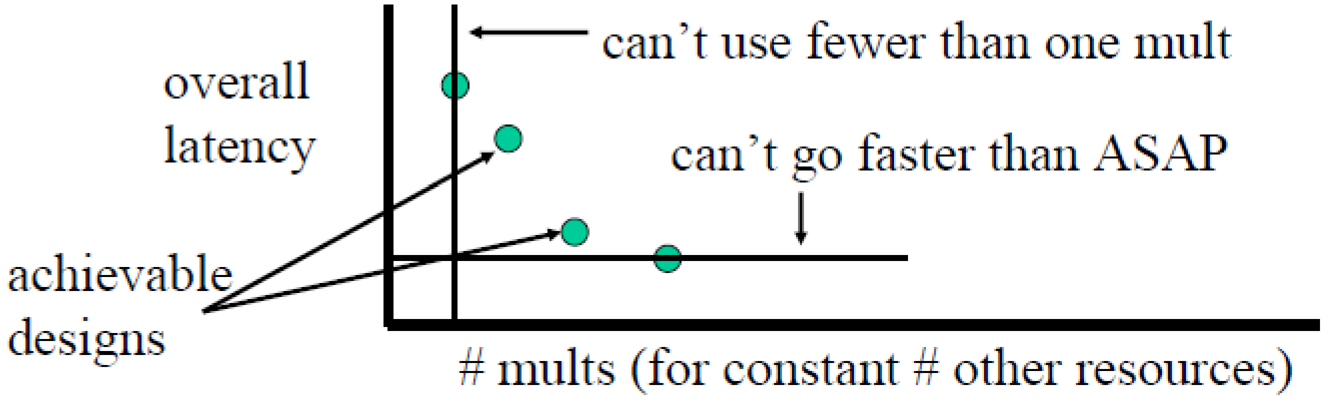
List-scheduled times



ASAP times

Area- Speed Trade offs

- In general, if we allow more resources, the schedule may have a shorter latency
- Similarly, if we allow a longer latency, the schedule may require fewer resources
- This leads to the concept of an area / speed tradeoff
  - one of a designers most important jobs is to explore this curve – and architectural synthesis tools can help



Allocation and Binding

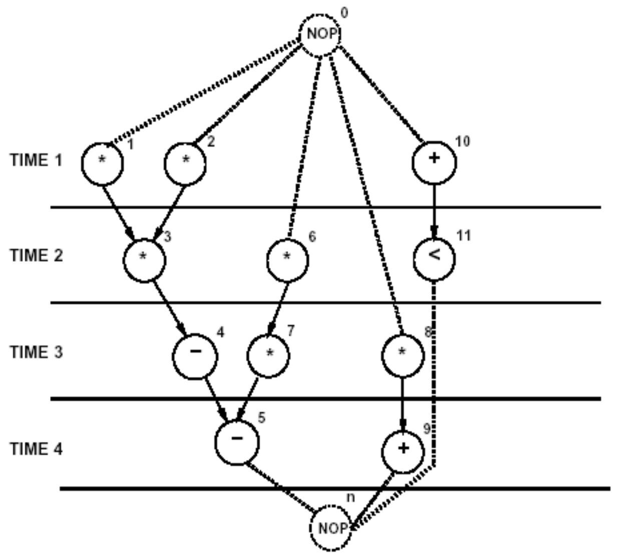
- Allocation
  - Determine number of resources needed
- Binding
  - Mapping between operations and resources.
- Sharing
  - Assignment of a resource to more than one operation.
- Optimum binding/sharing
  - Minimize the resource usage.

Optimum Sharing Problem

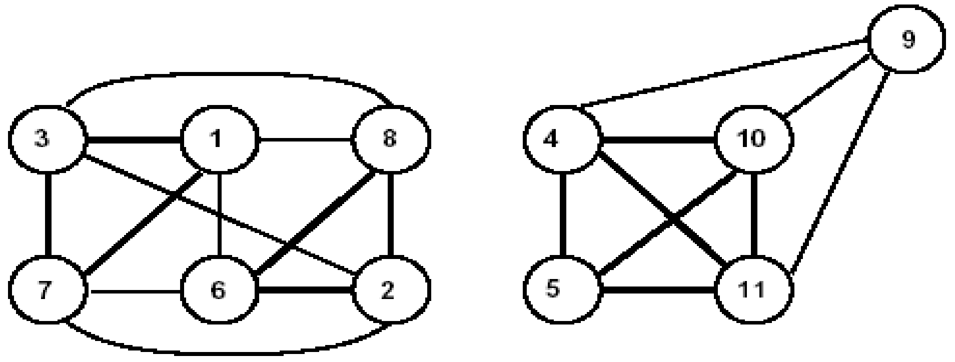
- Scheduled sequencing graphs.
  - Operation concurrency well defined.
- Consider operation types independently.
  - Problem decomposition.
  - Perform analysis for each resource type.
- Minimize resource usage.

Compatibility and Conflicts

- Operation compatibility
  - Same resource type.
  - Non concurrent.
- Compatibility graph
  - Vertices: operations.
  - Edges: compatibility relation.
- Conflict graph
  - Complement of compatibility graph.



Compatibility graphs



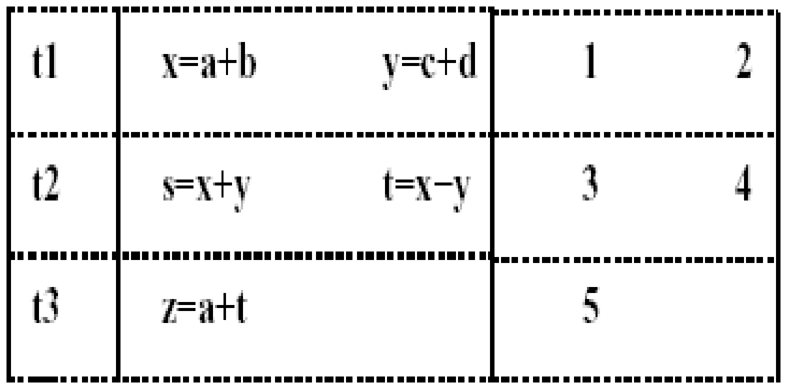
Multiplier

ALU

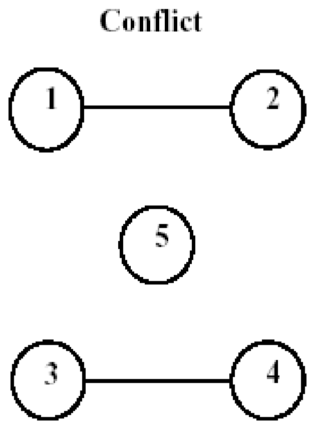
Algorithmic Solution to the Optimum Binding Problem

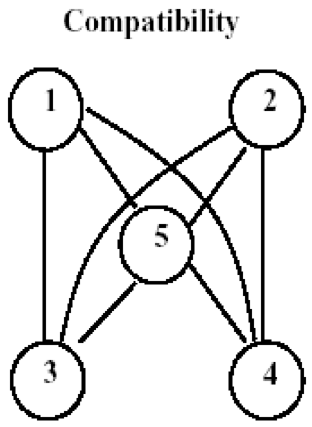
- Compatibility graph.
  - Partition the graph into a minimum number of cliques.
  - Find clique cover number.
- Conflict graph.
  - Color the vertices by a minimum number of colors.
  - Find chromatic number.
- NP-complete problems - Heuristic algorithms.

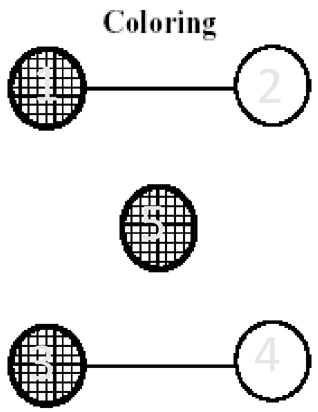
Example

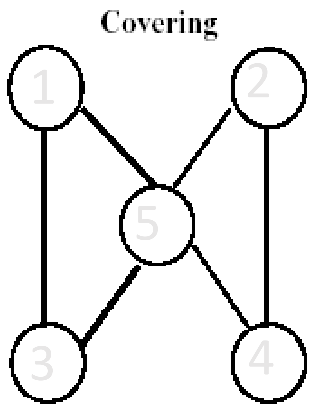


t1	x=a+b	y=c+d	1	2
t2	s=x+y	t=x−y	3	4
t3	z=a+t		5	









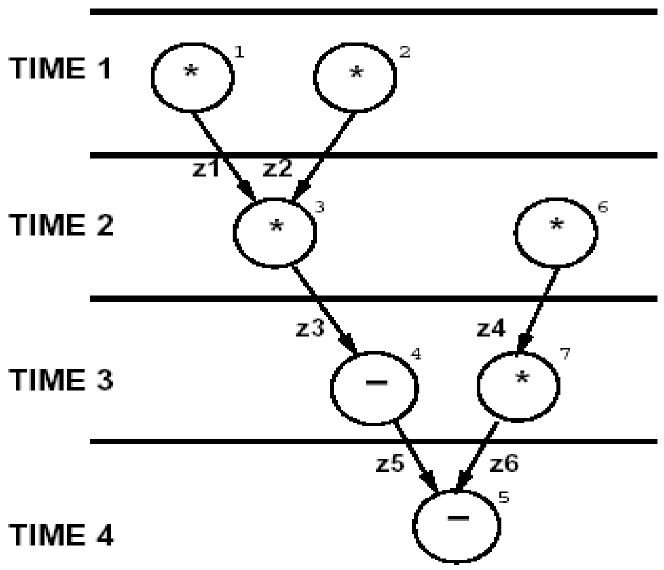
ALU1: 1, 3, 5
ALU2: 2, 4

Register Binding Problem

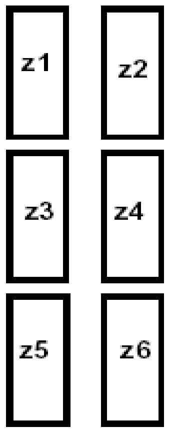
- Given a schedule
  - Lifetime intervals for variables.
  - Lifetime overlaps.
- Conflict graph (interval graph).
  - Vertices $\leftrightarrow$ variables.
  - Edges $\leftrightarrow$ overlaps.
- Find minimum number of registers storing all the variables.
- Compatibility graph.

Example

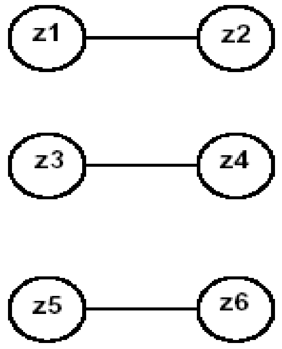
- Six intermediate variables that need to be stored in registers {z1, z2, z3, z4, z5, z6}
- Six variables can be stored in two registers



(a)



(b)

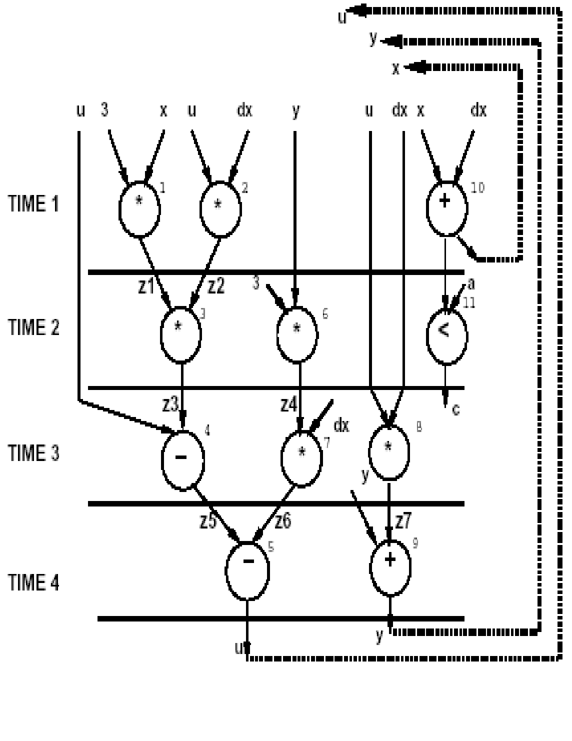


(c)

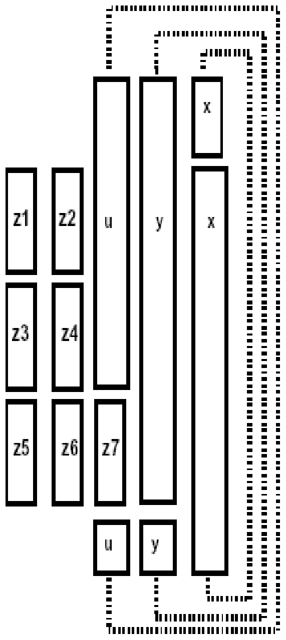
Example

- 7 intermediate variables, 3 loop variables, 3 loop invariants
- 5 registers suffice to store 10 intermediate loop variables

```
diffeq {
    read $(x, y, u, dx, a)$;
    repeat {
        $xl = x + dx$;
        $ul = u - (3 \cdot x \cdot u \cdot dx) - (3 \cdot y \cdot dx)$;
        $yl = y + u \cdot dx$;
        $c = x < a$;
        $x = xl; u = ul; y = yl$;
        }
    until ( c ) ;
write $(y)$;
}
```



(a)



(b)