ASAP and ALAP scheduling

- We're now entering the final portion of the course
  - Scheduling and retiming
  - Resource sharing algorithms
  - Floorplanning
  - Function Approximation
  - Perspectives for the future
- This lecture covers
  - The ASAP scheduling algorithm
  - The ALAP scheduling algorithm and operation slack
  - Introducing timing constraints into schedules

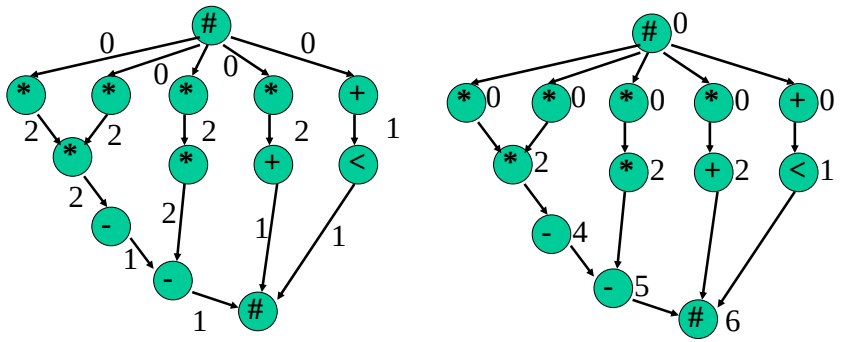
ASAP Scheduling

- 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

- 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 presented in Lecture 8
- Consider the original example from Lecture 1, and assume that multiplication takes two cycles, whereas addition and comparison take one cycle

ASAP Scheduling



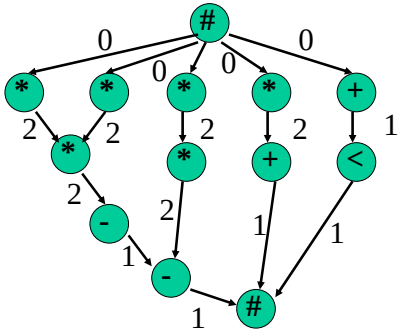
Edge weighted CDFG Scheduled start times

- 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

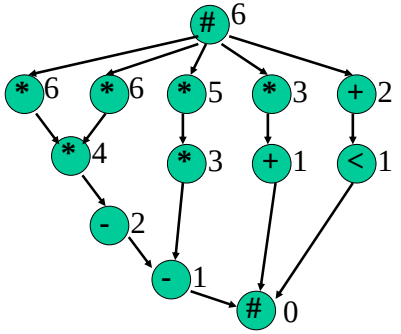
ALAP Scheduling

- 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



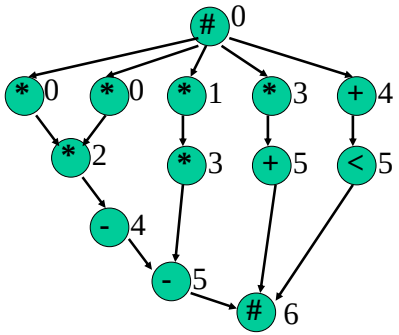
Edge-weighted CDFG



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

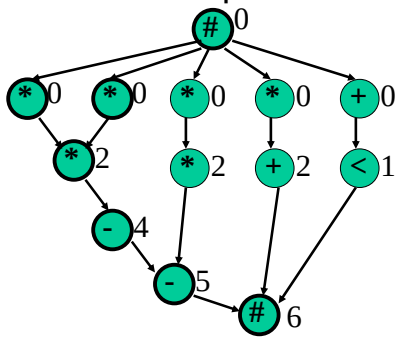
ALAP Scheduling

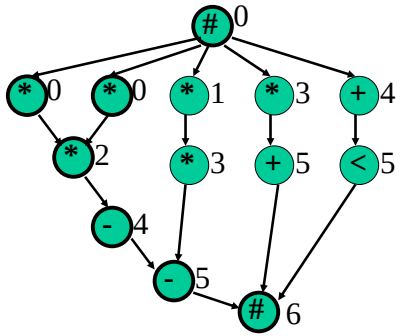


- Here are the ALAP start times. You can see that each operation starts at the latest opportunity possible to still meet 6 cycles overall

Mobility

- 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.

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.

Types of Timing Constraint

- 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 Constraint

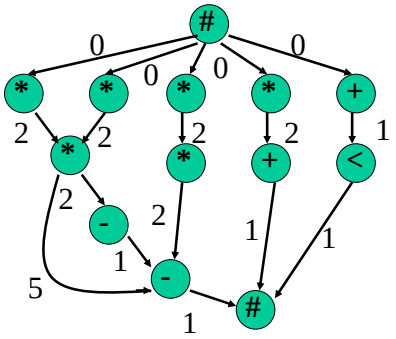
- We will consider two types of constraint
  - a minimum timing constraint l_{ij} between operations v_i and v_j: $S(v_j) \geq S(v_i) + l_{ij}$
  - a maximum timing constraint u_{ij} between operations v_i and v_j : $S(v_j) \leq S(v_i) + u_{ij}$
- These constraints are sufficient to model the situations 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

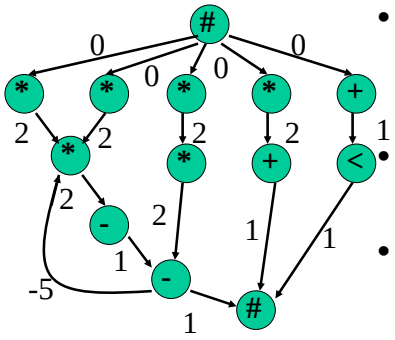
- Minimum timing constraints can simply be modelled by adding an extra edge (v_i, v_j) with weight l_{ij}



- 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

Scheduling with timing constraints

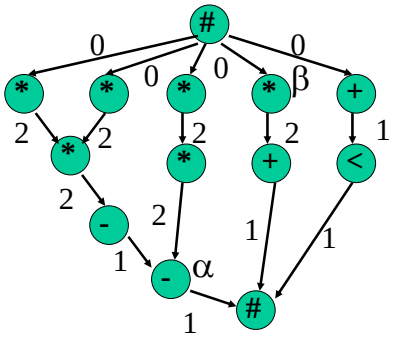
- ASAP / ALAP scheduling can still be performed on constraint graphs through the longest path technique, BUT…
  - the graph may no longer be a DAG (e.g. on the previous slide)
  - we may need to use Liao-Wong to find the longest path

Summary

- This lecture has covered
  - The ASAP scheduling algorithm
  - The ALAP scheduling algorithm and operation slack
  - Introducing timing constraints into schedules
- Next lecture will look at list scheduling, an heuristic method to find a short schedule given constraints on the number of each type of resource available

Suggested Problem

- Consider again the differential equation example from Lecture 1, repeated again below.



- It is required that the subtraction operation marked (α) begin no later than 3 cycles after the multiplication operation marked (β)
- Compare the ALAP schedules with and without this constraint

More Suggested Problems

- DeMicheli, Chapter 5, Problems 2 and 3 (note that DeMicheli refers to a combined min and max constraint between the source vertex and an operation as a “release time” constraint)

List Scheduling

- The final portion of the course covers
  - Scheduling and retiming
  - Resource sharing algorithms
  - Floorplanning
  - Function Approximation
  - Perspectives for the future
- This lecture covers
  - resource constrained scheduling and latency constrained scheduling
  - the resource-constrained list-scheduling algorithm
  - the latency-constrained list-scheduling algorithm

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
- This problem is NP-hard (proof in Lecture 6), so generally heuristics are used to attack the problem
  - we will also be looking at a way to find an optimum solution next lecture

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 Algorithm

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

Resource Constrained Algorithm

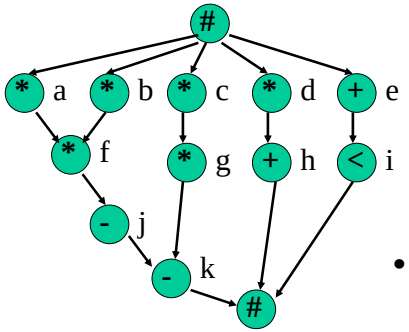
- At each clock cycle, the candidate set represents those operations we *could* schedule
- From the candidate set, we select a subset Y, which we *do* schedule
- The constraint on selection of Y is that we can never have more than a_r operations of type r executing simultaneously
- Notice that as $a_r \rightarrow \infty$ for all $r \in R$, the list schedule approaches an ASAP schedule

Resource Constrained 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 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 Example

- t = 0
  - $U_{0,*}$ = {a,b,c,d}, $U_{0,+/-}$ = {e}, $U_{0,<}$ = ∅
  - $T_{0,*}$ = ∅, $T_{0,+/-}$ = ∅, $T_{0,<}$ = ∅
  - 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 = ∅
  - S(a) = 0, S(b) = 0, S(e) = 0

Resource Constrained Example

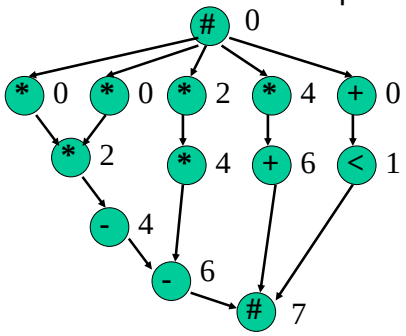
- $t = 1$
  - $U_{1,*} = \{c,d\}$, $U_{1,+/-} = \varnothing$, $U_{1,<} = \{i\}$
  - $T_{1,*} = \{a,b\}$, $T_{1,+/-} = \varnothing$, $T_{1,<} = \varnothing$
  - For +/-, $Y = \varnothing$
  - For *, $Y = \varnothing$ (all resources busy)
  - For <, $Y = \{i\}$
  - $S(i) = 1$

Resource Constrained Example

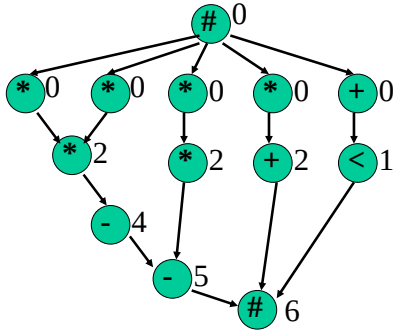
- t = 2
  - $U_{2,*} = \{c,d,f\}$, $U_{2,+/-} = \varnothing$, $U_{2,<} = \varnothing$
  - $T_{2,*} = \varnothing$, $T_{2,+/-} = \varnothing$, $T_{2,<} = \varnothing$
  - For +/-, $Y = \varnothing$
  - For *, ALAP times for c,d,f are 1,3,2 respectively. $Y = \{c,f\}$
  - For <, $Y = \varnothing$
  - $S(c)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

Resource Constrained 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 from Lect 9

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 Algorithm

```
Algorithm LC_ListSchedule( G(V,E), R, λ ) {
  perform ALAP( G(V,E), λ );
  set a_r = 1 for all r ∈ R;
  set t = 0;
  repeat {
    foreach r ∈ R {
      determine U_t,r;
      determine T_t,r;
      determine slack s_v = ALAP_v – t for all v ∈ U_tr;
      set Y_1 = {v ∈ V: s_v = 0};
      set a_r = max( a_r , |Y_1| + |T_tr| );
      select Y_2 ⊆ U_tr, s.t. |Y_1 ∪ Y_2| + |T_tr| ≤ a_r;
      set S(v) = t for all v ∈ Y_1 ∪ Y_2;
    }
    set t = t+1;
  } until all nodes scheduled
  return( S, a );
}
```


Latency Constrained Algorithm

- 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 Algorithm

- 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 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 Example

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

Latency Constrained Example

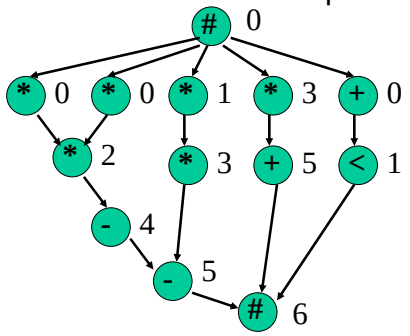
- t = 1
  - $U_{1,*}$ = {c,d}, $U_{1,+/-}$ = ∅, $U_{1,<}$ = {i}
  - $T_{1,*}$ = {a,b}, $T_{1,+/-}$ = ∅, $T_{1,<}$ = ∅
  - $s_c = 0$, $s_d = 2$, $s_i = 4$
  - For *, Y_1 = {c}; for +/-, Y_1 = ∅; for <, Y_1 = ∅
  - a_* = 3; others unchanged
  - For *, Y_2 = ∅; for +/-, Y_2 = ∅; for <, Y_2 = {i}
  - S(c)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

Latency Constrained Example

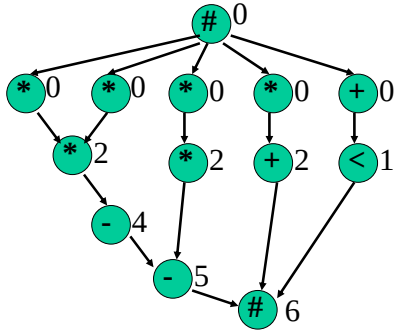
- t = 2
  - $U_{2,*}$ = {f,d}, $U_{2,+/-}$ = ∅, $U_{2,<}$ = ∅
  - $T_{2,*}$ = {c}, $T_{2,+/-}$ = ∅, $T_{2,<}$ = ∅
  - $s_f = 0$, $s_d = 1$
  - For *, Y_1 = {f}; for +/-, Y_1 = ∅; for <, Y_1 = ∅
  - all resource constraints unchanged
  - For *, Y_2 = {d}; for +/-, Y_2 = ∅; for <, Y_2 = ∅
  - S(f) = 2,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

Latency Constrained Example

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



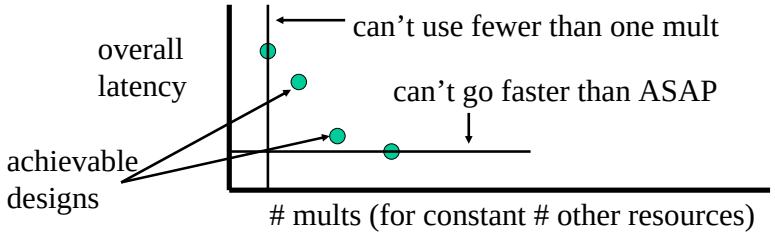
List-scheduled times



ASAP times from Lect 9

Area / Speed Tradeoffs

- 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



Summary

- This lecture has covered
  - resource constrained scheduling and latency constrained scheduling
  - the resource-constrained list-scheduling algorithm
  - the latency-constrained list-scheduling algorithm
  - area / speed tradeoffs
- Next lecture will look at optimum scheduling methods, using Integer Linear Programming

Suggested Problems

1. Re-visit the differential equation example. For two +/- resources and one < resource, draw the complete Area / Speed tradeoff curves achieved by applying
   - resource-constrained list-scheduling
   - latency-constrained list-scheduling

   Are they the same? Account for any differences (**)
2. Write a program to perform one of the list-scheduling algorithms and test it on some CDFGs of your own invention (***)

Optimum Scheduling

- The final portion of the course covers
  - Scheduling and retiming
  - Resource sharing algorithms
  - Floorplanning
  - Function Approximation
  - Perspectives for the future
- This lecture covers
  - Optimum scheduling: why ILP?
  - Integer linear program model
  - Example ILP and solution

Optimum Scheduling

- Last lecture we looked at an heuristic scheduling technique: list scheduling
- We may also wish to know the optimum result for a given scheduling problem
  - optimum results are only achievable for small problems, as resource-constrained scheduling is *NP*-hard
  - if we design a heuristic, and it achieves near-optimal schedules for small problems, we are usually more confident it will do well for large problems
  - optimum results form a “baseline” against which we can compare heuristics

Why ILP?

- Integer Linear Programming is useful to achieve optimum results because
  - it lets us formalize the problem
  - it gives a structure to the problem: what is the objective function, what are the constraints, how many are there, what are their nature?
  - we can use ILP solvers such as lp_solve (ftp://ftp.es.ele.tue.nl/pub/lp_solve/) to solve problems once they are in ILP format

Notation

- We will use the following notation, mainly carried over from previous lectures
  - $S(v)$: the scheduled start time of node v
  - d_v: the delay (latency) of node v
  - a_r: the maximum number of resources of type r
  - $T(v)$: the type of node v
  - R: the set of resource types
  - λ: the maximum overall latency
  - $ASAP_v$ ($ALAP_v$): the ASAP time (ALAP time) under overall latency λ
  - x_{vt}: binary decision variable (see next slide)
  - c_r: the cost of a resource of type r

Binary Decision Variables

- We will use a trick often used in ILP formulations: to introduce binary decision variables
- We will use x_{vt} ($v \in V$, $t \in \{ASAP_v, ASAP_v+1, \ldots, ALAP_v\}$, with $x_{vt} = 1$ iff node v is scheduled to start at time t, i.e. $x_{vt} = 1 \Leftrightarrow S(v) = t$
- These will allow us to formulate the resource constraints as *linear* functions of x_{vt}
- Note that if we are doing resource-constrained scheduling, we may not know λ. Since it is an upper bound, we can use RC list scheduling to obtain it.

Ensuring a Unique Start Time

- Our first constraint needs to be to ensure that each operation starts at only one time

$$\forall v \in V : \sum_{t=ASAP_v}^{ALAP_v} x_{vt} = 1$$

- Because x_{vt} are constrained to be binary variables, this means that exactly one time-index is true for each operation

Specifying Data Dependencies

- Of course we can't allow operations to start before their predecessors in the CDFG have completed

$$\forall (v', v) \in E : \sum_{t=ASAP_v}^{ALAP_v} t \cdot x_{vt} \geq \sum_{t=ASAP_{v'}}^{ALAP_{v'}} t \cdot x_{v't} + d_{v'}$$

- Each edge in the CDFG defines one of these constraints
- Each summation represents the start time of the particular node (v on the LHS, v' on the RHS)

Specifying Resource Constraints

- No more than a_r operations of type r can simultaneously execute

$$\forall r \in R, \forall t \in \{0, \ldots, \lambda\},$$
$$\sum_{v \in V : T(v) = r} \quad \sum_{t' \in \{t - d_v + 1, \ldots, t\} \cap \{ASAP_v, \ldots, ALAP_v\}} x_{vt'} \leq a_r$$

- The first summation is over all nodes of type r
- The second summation is over a time "window" covering all start cycles t' for which the operation would still be executing by cycle t

Resource-Constrained Objective Function

- Under these constraints, the resource-constrained scheduling problem can be solved by minimizing the overall latency (we fix a_r)

$$\min : \sum_{t=ASAP_{v_z}}^{ALAP_{v_z}} t \cdot x_{v_z t}$$

- Here, v_z represents the "end" or "sink" node in the CDFG

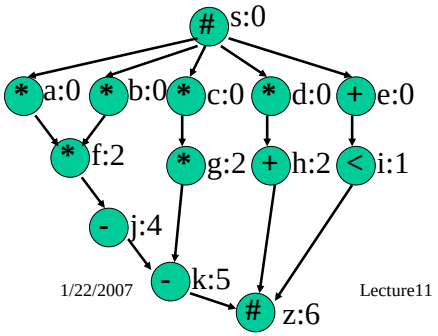
Latency-Constrained Objective Function

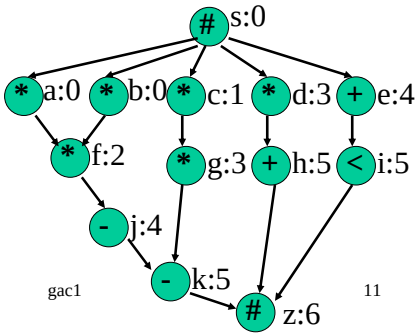
- Under the same constraints, the latency-constrained scheduling problem can be solved by minimizing the cost of the resources required (we fix λ)

$$\min : \sum_{r \in R} c_r a_r$$

Example ILP

- We will build an ILP for the differential equation solver as an example
- We will formulate the latency-constrained problem for λ = 6, the minimum possible latency
- To refresh your memories, here are the ASAP and ALAP times for λ = 6 from Lecture 9





Example ILP

- First, lets examine what variables we have:

$$\{x_{s0}, x_{a0}, x_{b0}, x_{c0}, x_{c1}, x_{d0}, x_{d1}, x_{d2},$$
$$x_{d3}, x_{e0}, x_{e1}, x_{e2}, x_{e3}, x_{e4}, x_{f2}, x_{g2},$$
$$x_{g3}, x_{h2}, x_{h3}, x_{h4}, x_{h5}, x_{i1}, x_{i2}, x_{i3},$$
$$x_{i4}, x_{i5}, x_{j4}, x_{k5}, x_{z6}\}$$

- Operations with large mobility give rise to a large number of variables

Example ILP

- The first constraints are unique-start-time constraints:

$$x_{s0} = 1$$
$$x_{a0} = 1$$
$$x_{b0} = 1$$
$$x_{c0} + x_{c1} = 1$$
$$x_{d0} + x_{d1} + x_{d2} + x_{d3} = 1$$
$$x_{e0} + x_{e1} + x_{e2} + x_{e3} + x_{e4} = 1$$
$$x_{f2} = 1$$
$$x_{g2} + x_{g3} = 1$$
$$x_{h2} + x_{h3} + x_{h4} + x_{h5} = 1$$
$$x_{i1} + x_{i2} + x_{i3} + x_{i4} + x_{i5} = 1$$
$$x_{j4} = 1$$
$$x_{k5} = 1$$
$$x_{z6} = 1$$

Example ILP

- The next constraints are dependency constraints:

$$0 \cdot x_{a0} \geq 0 \cdot x_{s0} + 0$$
$$0 \cdot x_{b0} \geq 0 \cdot x_{s0} + 0$$
$$0 \cdot x_{c0} + 1 \cdot x_{c1} \geq 0 \cdot x_{s0} + 0$$
$$0 \cdot x_{d0} + 1 \cdot x_{d1} + 2 \cdot x_{d2} + 3 \cdot x_{d3} \geq 0 \cdot x_{s0} + 0$$
$$0 \cdot x_{d0} + 1 \cdot x_{d1} + 2 \cdot x_{d2} + 3 \cdot x_{d3} + 4 \cdot x_{d4} \geq 0 \cdot x_{s0} + 0$$
$$2 \cdot x_{f2} \geq 0 \cdot x_{a0} + 2$$
$$2 \cdot x_{f2} \geq 0 \cdot x_{b0} + 2$$
$$2 \cdot x_{g2} + 3 \cdot x_{g3} \geq 0 \cdot x_{c0} + 1 \cdot x_{c1} + 2$$
$$2 \cdot x_{h2} + 3 \cdot x_{h3} + 4 \cdot x_{h4} + 5 \cdot x_{h5} \geq 0 \cdot x_{d0} + 1 \cdot x_{d1} + 2 \cdot x_{d2} + 3 \cdot x_{d3} + 2$$
$$1 \cdot x_{i1} + 2 \cdot x_{i2} + 3 \cdot x_{i3} + 4 \cdot x_{i4} + 5 \cdot x_{i5} \geq 0 \cdot x_{e0} + 1 \cdot x_{e1} + 2 \cdot x_{e2} + 3 \cdot x_{e3} + 4 \cdot x_{e4} + 1$$

Example ILP

- Dependency constraints continued…

$$4 \cdot x_{j4} \geq 2 \cdot x_{f2} + 2$$
$$5 \cdot x_{k5} \geq 2 \cdot x_{g2} + 3 \cdot x_{g3} + 2$$
$$6 \cdot x_{z6} \geq 2 \cdot x_{h2} + 3 \cdot x_{h3} + 4 \cdot x_{h4} + 5 \cdot x_{h5} + 1$$
$$6 \cdot x_{z6} \geq 1 \cdot x_{i1} + 2 \cdot x_{i2} + 3 \cdot x_{i3} + 4 \cdot x_{i4} + 5 \cdot x_{i5} + 1$$
$$5 \cdot x_{k5} \geq 4 \cdot x_{j4} + 1$$
$$6 \cdot x_{z6} \geq 5 \cdot x_{k5} + 1$$

Example ILP

- Resource constraints:

$$r = <, t = 1 : x_{i1} \leq a_{<}$$
$$r = <, t = 2 : x_{i2} \leq a_{<}$$
$$r = <, t = 3 : x_{i3} \leq a_{<}$$
$$r = <, t = 4 : x_{i4} \leq a_{<}$$
$$r = <, t = 5 : x_{i5} \leq a_{<}$$
$$r = +/-, t = 0 : x_{e0} \leq a_{+/-}$$
$$r = +/-, t = 1 : x_{e1} \leq a_{+/-}$$
$$r = +/-, t = 2 : x_{e2} + x_{h2} \leq a_{+/-}$$
$$r = +/-, t = 3 : x_{e3} + x_{h3} \leq a_{+/-}$$
$$r = +/-, t = 4 : x_{e4} + x_{h4} + x_{j4} \leq a_{+/-}$$
$$r = +/-, t = 5 : x_{h5} + x_{k5} \leq a_{+/-}$$

Example ILP

- More resource constraints:

$$r = *, t = 0 : x_{a0} + x_{b0} + x_{c0} + x_{d0} \le a_*$$

$$r = *, t = 1 : x_{a0} + x_{b0} + x_{c0} + x_{c1} + x_{d0} + x_{d1} \le a_*$$

$$r = *, t = 2 : x_{c1} + x_{d1} + x_{d2} + x_{f2} + x_{g2} \le a_*$$

$$r = *, t = 3 : x_{d2} + x_{d3} + x_{f2} + x_{g2} + x_{g3} \le a_*$$

- Objective function:
  - let's assume the cost of a mult is "2", and that of an adder and comparator is "1":

$$\min : 2a_* + a_{+/-} + a_<$$

Example ILP

- This (rather long!) example contains 29 binary decision variables and 3 resource allocation variables (total = 32) and 44 constraints
- For even this small example, the ILP model is quite sizable
  - ILP is only really practical for solving small problems

Summary

- This lecture has covered
  - Optimum scheduling: why ILP?
  - Integer linear program model
  - Example ILP and solution
- Next lecture will move off the subject of scheduling, and start to consider algorithms for resource sharing

Suggested Problems

- Download a copy of lp_solve from the website given at the start of the lecture, and solve the ILP example
  - what is the minimum possible cost?
  - how many adders, multipliers, and comparators does it use?
  - how does that compare with a latency-constrained list-schedule?

Affine Scheduling

- The final portion of the course covers
  - Scheduling and retiming
  - Resource sharing algorithms
  - Floorplanning
  - Function Approximation
  - Perspectives for the future
- This lecture covers
  - Scheduling nested loops: the affine approach

Nested Loop Programs

- So far, we have only looked at scheduling "straight-line" code
  - Loops can be trivially scheduled by repeating the schedule of the loop body.
  - However, this is not always the most efficient way.
- We shall now consider nested loop programs:

```
for i1 = l1 to u1
  for i2 = l2(i1) to u2(i1)
    ...
    for in = ln(i1,...,in-1) to un(i1,...,in-1)
          S1: first statement
          ...
          Sk: kth statement
    end for
    ...
  end for
end for
```


Affine Nested Loop Programs

- To simplify notation, we will discuss scheduling *statements*, rather than operations
  - Equivalent if each statement contains a single operation.
- Our scheduling procedures so far would allocate a start time $S(u)$ to each statement u in the inner loop
  - loops will run sequentially.
- We can do better if we make a (practical) restriction on the functions l_j and u_j
  - Let us denote $i = (i_1, i_2, ..., i_n)^T$.
  - We will assume l_j and u_j are affine, *i.e.*

    $l_j(i) = \underline{l_j}^T i + l_j^0$,

    $u_j(i) = \underline{u_j}^T i + u_j^0$.

The Unrolling "Solution"

- Before going further, it let us consider an easy alternative:
  - "unroll" all the loops, *i.e.* convert to straight-line code,
  - Use one of our previous scheduling algorithms.
- Problem:
  - Size of unrolled code exponential in *n.*
  - As a result, optimal scheduling infeasible, heuristic scheduling overwhelmed, massive FSM.

Affine Schedules

- The alternative is to define a scheduling function $S(i,v)$: the start time of statement v in iteration i.
- If we impose a particular functional form on $S(i,v)$, the problem becomes tractable
  - Ensure $S(i,v)$ is "affine-by-statement":
    $S(i,v) = t_v^T i + t_v^0$.
- The domain of the function S is $V \times \mathcal{IS}$, where $\mathcal{IS}$ denotes the iteration space.
- For an affine loop nest, $\mathcal{IS}$ is the set of integral points inside $Ai \le b$, known as a *convex polytope.*

Iteration Space

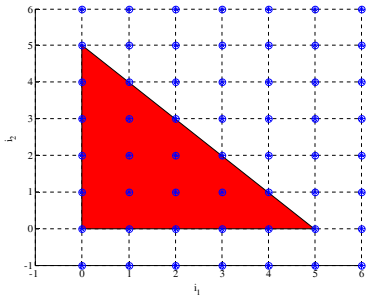
- This is because the lower and upper iteration bounds impose linear constraints on i:

$$A = \begin{pmatrix} -1 & 0 & \cdots & 0 & 0 \\ +1 & 0 & \cdots & 0 & 0 \\ \underline{l}_{11} & -1 & \cdots & 0 & 0 \\ -\underline{u}_{11} & +1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \underline{l}_{n1} & \underline{l}_{n2} & \cdots & \underline{l}_{n(n-1)} & -1 \\ -\underline{u}_{n1} & -\underline{u}_{n2} & \cdots & -\underline{u}_{n(n-1)} & +1 \end{pmatrix} \quad b = \begin{pmatrix} -l_1^0 \\ u_n^0 \\ -l_2^0 \\ u_n^0 \\ \vdots \\ -l_n^0 \\ u_n^0 \end{pmatrix}$$

Iteration Space

- Geometrically:
  - each constraint (a row in A and b) cuts n-dimensional space with an $(n-1)$-dimensional hyperplane.
- Graphical example:

```
for i1 = 0 to 5
  for i2 = 0 to 5 – i1
    ...
  end for
end for
```



Dependences

- As before, the key issue in scheduling is to respect data dependences ('flow' dependences).
  - We shall now consider inter-iteration data dependences.
  - Typically, these are carried by array accesses.

```
for i1 = 1 to 100
  for i2 = 0 to 100
    s[ i1 ][ i2 ] = s[ i1 – 1 ][ i2 ] + c[ i1 ][ i2 ]*x[ i2 ]
  end
end
```

  - In this code, iteration (i_1,i_2) must execute after iteration $(i_1\text{-}1,j)$ due to dependence carried by access to array "s".
  - In the unrolled CDFG, this would be a normal edge.

Constant dependences

- Each of the dependences imposes a linear constraint on t_v
  - For our example, there is only one statement, so we shall drop the "*v*" subscript, and denote the delay of this statement by *d*. Then:

$$t^T \begin{pmatrix} i_1 \\ i_2 \end{pmatrix} \geq t^T \begin{pmatrix} i_1 - 1 \\ i_2 \end{pmatrix} + d \Rightarrow (1 \quad 0)t \geq d$$

  - In this example, there is nothing in the constraint $(1\ 0)t \geq d$ that depends on *i* or *j*; this is a *constant dependence*.

Constant dependences

- Constant dependences make life easier
  - One linear constraint per statement
  - Any feasible solution to the corresponding linear set of constraints is a valid schedule!
  - We could define an appropriate objective function, depending on what we're trying to optimize – overall latency, etc.
  - More complex techniques exist to deal with non-constant (but still affine!) dependences
    - P. Feautrier, "Some Efficient Solutions to the Affine Scheduling Problem I: One-Dimensional Time", *Int. J. Parallel Programming* 21(5), 1992, pp. 313-347.

Example Objective

- We have our constraints: what about an objective function?
  - Instance *i* of statement *v* completes by $t_v^T i + t_v^0 + d(v)$.
  - This linear function of *i* will be maximized at *one of the vertices*.
  - For each vertex *i*, introduce a constraint $\lambda \geq t_v^T i + t_v^0 + d(v)$.
  - Min latency objective is then just min: λ.

Limitations

- Affine scheduling sub-optimal, e.g. the code below, where n is some constant known at synthesis time.

```
for i = 0 to n
  for j = 0 to i
    s = s + a(i,j)
  end for
end for
```

- The code is completely sequential. The best (non-affine) schedule is $S(i,j) = i(i+1)/2 + j$, giving overall latency $n(n+3)/2$. The best affine schedule $S(i,j) = ni + j$, which is much worse (approx twice as slow), at $n(n+1)$.
- Can use multi-dimensional "time" ⇔ polynomial schedules.

Summary

- This lecture has covered
  - Affine nested loop programs
  - Affine schedules
  - Constant and affine dependences
  - The vertex method
  - Limitations of affine schedules.
- Next lecture will move off the subject of scheduling, and start to consider algorithms for resource sharing.

Suggested Problems

- Consider the code below.
- Determine the flow dependences, and construct a linear program to schedule this code.
  - Assume each statement takes a single cycle

```
for i = 1 to 10
  for j = i to 2*i
    x[ i ][ j ] = x[ i - 1][ j ] * x[ i ][ j – 1]
```

Resource Sharing

- The final portion of the course covers
  - Scheduling and retiming
  - Resource sharing algorithms
  - Floorplanning
  - Function Approximation
  - Perspectives for the future
- This lecture covers
  - Non-hierarchical CDFGs
  - Hierarchical CDFGs

Introduction

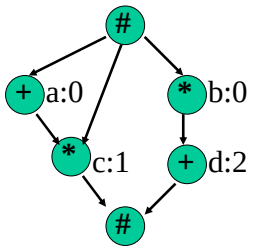
- We will consider some approaches for sharing resources between operations
- Non-hierarchical and hierarchical CDFGs will be considered separately
  - problem has different complexity
- Remember that hierarchical CDFGs can be used to represent the following (Lecture 1)
  - conditionals
  - loops
  - function calls

Resource Conflict Graph

- The one fundamental restriction on sharing resources:
  - two operations executing simultaneously cannot be executed on the same resource
- This leads to the concept of “resource conflict”
- Two operations are in resource conflict if they overlap in execution time
- A resource conflict graph uses the same node set as the CDFG, but uses a set of undirected edges such that: (Lecture 2)
  - two operations are joined by an edge iff they are in resource conflict

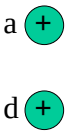
Non-Hierarchical CDFGs

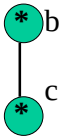
- For non-hierarchical CDFGs (i.e. those with just one level of hierarchy), such a conflict graph is simple



non-hierarchical CDFG

adder conflict graph





multiplier conflict graph

Graph Structure

- Conflict graphs for non-hierarchical CDFGs are *interval graphs*
- Recall from Lecture 5 that an interval graph is one whose vertices can be put in one-to-one correspondence with a set of intervals, such that two vertices are connected by an edge iff the corresponding intervals intersect
- Also recall from Lecture 5 that such graphs are colourable easily in polynomial time using the *left-edge* algorithm

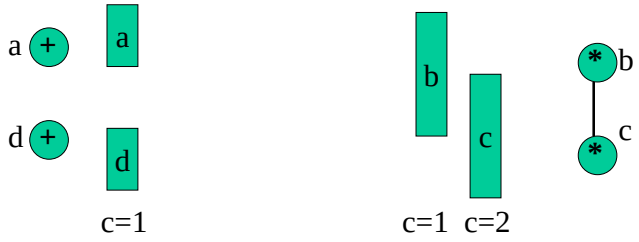
Solution via Left-Edge

- We can therefore find an optimum binding using left-edge, reproduced below from Lecture 5
  - use the scheduled start and end times as the left and right "edges", respectively

```
Left_Edge( G(V,E) )
begin
 sort nodes in ascending order of left edge – store in L
 c := 1;
 while( not all vertices have been coloured ) {
  r := 0;
  while( there is a vertex in L with l_s > r ) {
  v_s := first node in L with l_s > r;
  r := r_s;
  label v_s with colour c
  L := L \ {v_s}; }
 c := c + 1; }
end
```


Left-Edge: Example

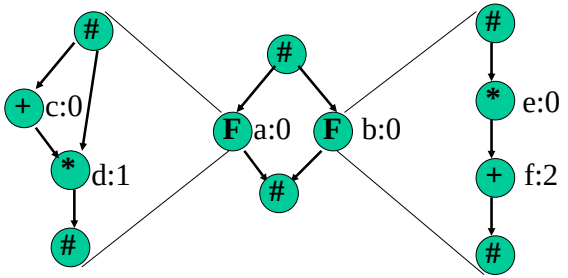
- Taking the previous example:



- So use one adder to do both a and d, but different multipliers to do b and c
- Formally, Y(a) = (+,1); Y(b) = (*,1); Y(c)=(*,2); Y(d)=(+,1)

Hierarchical CDFGs

- Consider a simple hierarchical CDFG with function calls, performing the same function as the previous example



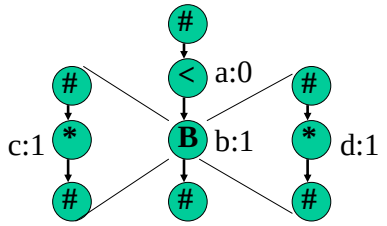
Hierarchical CDFGs

- How do we perform resource sharing?
  - a naïve approach would be to perform resource sharing on each level of the hierarchy in turn
  - for our example, this would lead to one multiplier and one adder for each function: one more adder than we needed for the non-hierarchical version
- We should try to share resources across the levels of hierarchy

Conditionals

- Conditionals help us share resources, as the two branches ("if" and "else") are never needed simultaneously

```
a = b < c;
if (a) then
  d = b * b;
else
  d = c * c;
```



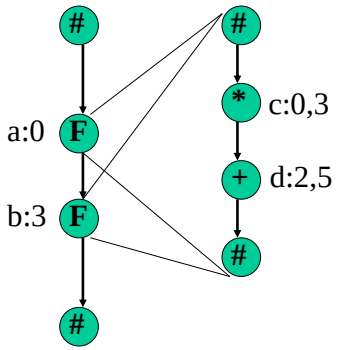
- Operations c and d are not in resource conflict, although they have the same type and "overlap" in time

Multiple Function Calls

- Multiple calls to the same function complicate matters, as operations can have several execution times

```
a = fun(x);
b = fun(a);
```

```
fun(p) {
 return p*p + 5;
}
```



Graph Properties

- Conditionals and multiple function calls change the structure of the conflict graph
  - it no longer must be an interval graph
  - the left-edge algorithm is therefore no longer applicable
- We need an heuristic approach to colouring the graph
  - one such algorithm is given in Lecture 5

Colouring Heuristic

- Here is the colouring heuristic from Lecture 5:

```
Colour_Graph( G(V,E) )
begin
  foreach v ∈ V {
    c = 1;
    while ∃(v,v') ∈ E : v' has colour c
        c = c + 1;
    label v with colour c }
end
```

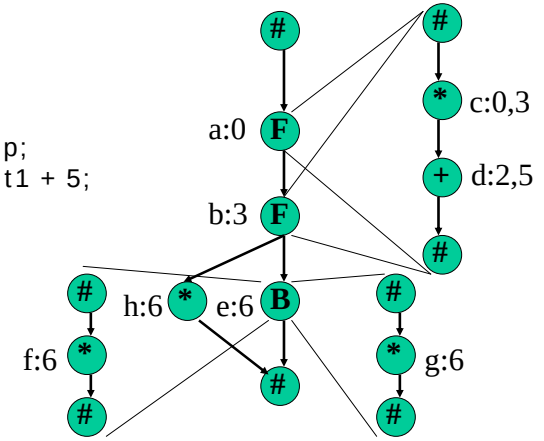
- We will apply it to an example with conditionals and multiple function calls

Hierarchical Example

- Here is a more complex scheduled CDFG

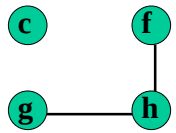
```
a = fun(x);
b = fun(a);
if (y) then
  c = b * b;
else
  c = 2 * b;
d = 3*b;
```

```
fun(p) {
 t1 = p*p;
 return t1 + 5;
}
```



Hierarchical Example

- Remember f and g don't conflict (if / else)

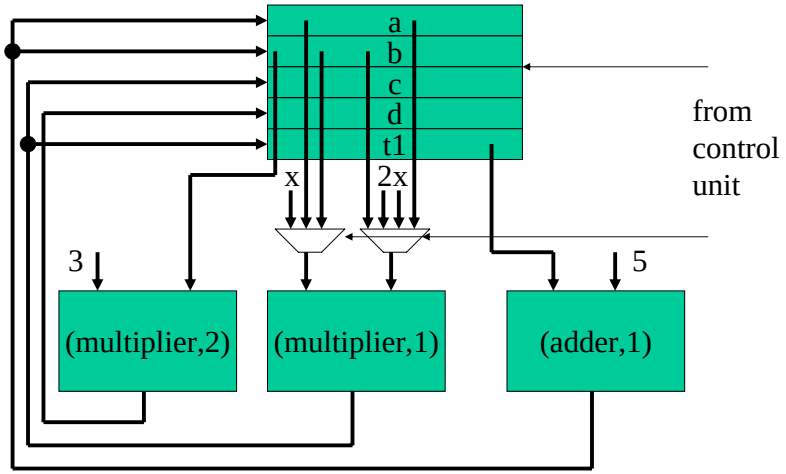


multiplier conflict graph

adder conflict graph

- Let's colour the multiplier nodes in the order: c, f, g, h
  - c gets colour 1; f gets colour 1; g gets colour 1; h gets colour 2
  - we need two mults and an add

Example Datapath



Summary

- We have investigated resource sharing for both
  - Non-hierarchical CDFGs
  - Hierarchical CDFGs
- Next lecture we will look at register sharing

Suggested Problems

- Perform a resource binding for the list-scheduled differential equation example from Lecture 10 and draw the completed datapath (*)
- Design a controller for this datapath (*)
- Discuss resource binding for conditionals within conditionals (****)
- Discuss a possible approach to resource binding for loops (****)
- De Micheli, Problems 6.11, No. 1 (conflict graphs only) (*)

Register Sharing

- The final portion of the course covers
  - Scheduling and retiming
  - Resource sharing algorithms
  - Floorplanning
  - Function Approximation
  - Perspectives for the future
- This lecture covers
  - The register sharing problem
  - Variable lifetime calculation
  - Register conflict graphs
  - Non-hierarchical register sharing
  - Hierarchical register sharing: the loop problem

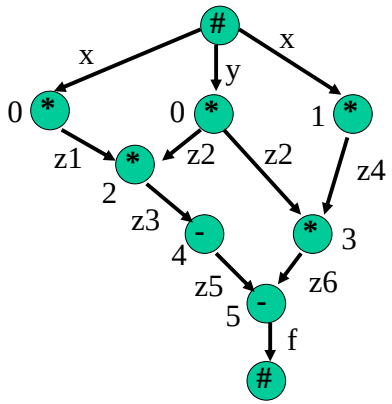
Register Sharing

- We have discussed sharing of arithmetic resources
  - registers also consume silicon area
- Registers are required for each intermediate result passed across a clock-cycle boundary
- So far, we have used a distinct register for each intermediate result
  - but we could share registers if results are not needed at the same time

Lifetime Analysis

- Consider the code and scheduled CDFG below
  - it has inputs x and y, and output f

```
z1 = 2*x;
z2 = 3*y;
z3 = z1*z2;
z4 = x*x;
z5 = z3 – 2;
z6 = z2*z4;
f = z5 – z6;
```

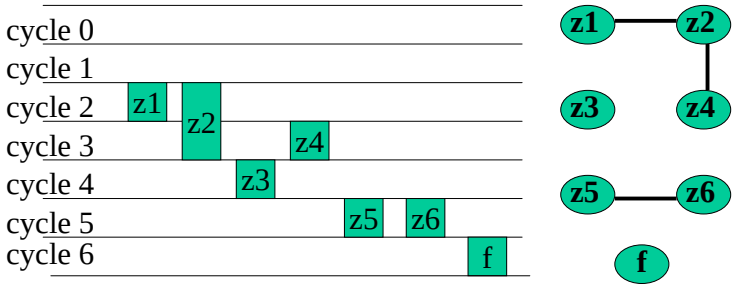


Lifetime Analysis

- Let's analyse the lifetime for which each result is required
  - z1 is produced during cycle 1 and consumed during cycle 2
  - z2 is produced during cycle 1 and consumed both during cycle 2 and cycle 3
  - z3 is produced during cycle 3 and consumed during cycle 4
  - z4 is produced during cycle 2 and consumed during cycle 3
  - z5 is produced during cycle 4 and consumed during cycle 5
  - z6 is produced during cycle 4 and consumed during cycle 5
  - f is produced during cycle 5 and consumed at some unknown time
- A register must be allocated to each result from the period AFTER production, to the period DURING the last consumption
  - this is the variable "lifetime"

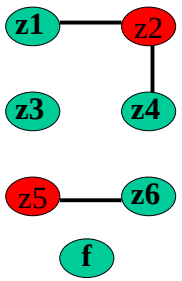
Register Conflict Graph

- Two results cannot share a register if their lifetimes overlap
  - we can thus create a register conflict graph just like the resource conflict graph used in the previous lecture



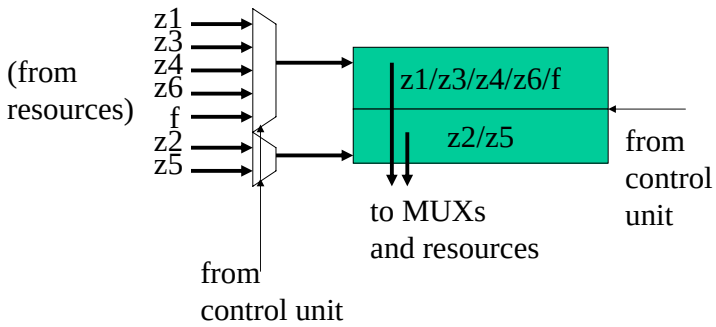
Register Conflict Graph

- As with resource sharing, for the non-hierarchical case the register conflict graph is an interval graph
  - optimum solution through the left-edge algorithm
- Our example conflict graph can be coloured with only two colours
  - only two registers are required
  - z1, z3, z4, z6 and f share a register
  - z2 and z5 share a register



Example Datapath

- So what would the datapath be for that design?

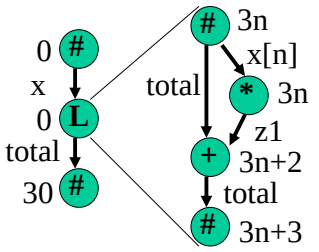


- Note the multiplexers on the register inputs
  - sharing resources leads to MUXs on resource inputs
  - sharing registers leads to MUXs on register inputs

Register sharing for loops

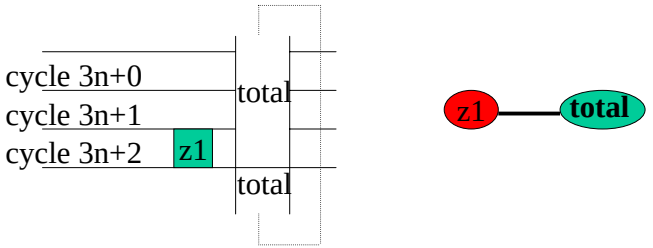
- As with resource sharing, things get more complicated for hierarchical CDFGs
  - we will not consider the general problem
  - but we will examine the effect of loops to give you a glimpse
- Consider the following sum-of-squares code and scheduled CDFG

```
total = 0;
for n=0 to 9
  z1 = x[n]*x[n];
  total = total + z1;
end
```



Register sharing for loops

- The result “total” is required to keep its value BETWEEN loop iterations
  - it is produced at cycles 3,6,9,…30 (excluding the initialization) and consumed at cycles 2,5,8,…,29, and at an unknown time after cycle 30



Register sharing for loops

- Because of the “circular arc” wrap around effect with some variables, the conflict graphs for hierarchical CDFGs are not always interval graphs
- Colouring such general graphs is NP-hard, requiring the use of our colouring heuristic (or similar)

Summary

- We have investigated register sharing:
  - Variable lifetime calculation
  - Register conflict graphs
  - Non-hierarchical register sharing
  - Hierarchical register sharing: the loop problem
- Next lecture we will look at the module selection problem

Suggested Problems

- Perform a resource binding, and thus complete the partial example datapath given this lecture (*)
- To what extent can the registers be shared in the resource-constrained list-scheduled example of Lecture 10? (*)
- How important is register sharing? (think about it…) (***)
- Consider what problems, if any, you may have extending the framework discussed in this lecture to (****)
  - function calls (with one call per function)
  - function calls (with unlimited calls per function)
  - conditionals

Module Selection

- The final portion of the course covers
  - Scheduling and retiming
  - Resource sharing algorithms
  - Floorplanning
  - Function Approximation
  - Perspectives for the future
- This lecture covers
  - The module selection problem
  - Module selection / scheduling / binding interaction
  - An ILP formulation

Module Selection

- So far, we have considered only one resource type capable of performing each operation, e.g.
  - an adder/subtractor performs additions or subtractions
  - a multiplier performs multiplications
- We could have different possibilities, e.g.
  - either an adder/subtractor or an ALU could perform an addition
  - either a ripple-carry adder or a carry-lookahead adder could perform an addition
- Module selection is the task of selecting an appropriate *type* of resource to perform each operations

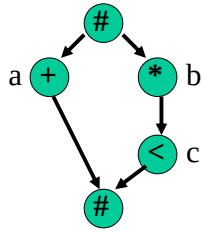
Interactions

- Ideally, we would like to perform module selection before scheduling
  - different resource types for a given operation may have different latencies
  - we need to know the latency (or at least an upper bound) before we can schedule
- However, ideally we would like to combine module selection and resource binding
  - we don't know which operations can share resources until we know the resource type of each operation
  - delaying module selection until binding will help us find a low-area implementation

Interactions

- For example, consider the code and CDFG below

```
z1 = x*2;
f1 = z1 < 3;
f2 = x+2;
```



- Assume we have the following library:
  - Adder: 1 area unit / latency 1 cycle, Comparator: 1 area unit / latency 1 cycle, ALU: 1.5 area units / latency 2 cycles, Multiplier: 2 area units / latency 2 cycles

Interactions

- We may wish to implement
  - a in an adder, c in a comparator
  - a and c in ALUs
- The second option is only useful if the operations can share a *single* ALU, otherwise it is a waste of area and latency
- We don't know if they can share a single ALU until after scheduling
  - we should perform module selection after scheduling
- But we don't know the latencies until module selection
  - we should perform module selection before scheduling

Interactions

- Since we perform scheduling before binding, there is clearly a contradiction
  - we want to do module selection early in the design flow
  - we want to do module selection late in the design flow
- One solution is to perform scheduling, module selection, and resource binding concurrently as a single problem
  - advantage: leads to high-quality solutions
  - disadvantage: leads to a complex problem to solve

ILP Formulation

- It is relatively straightforward to extend our ILP scheduling approach to consider the combined problem
- Rather than using variables x_{vt} to indicate the scheduling of operation v at time t
  - we assume we know an upper bound a_r on the number of resources required of type $r \in R$
  - use x_{vtir} to indicate the scheduling of operation v at time t on instance $i \in \{1,\ldots,a_r\}$ of resource type $r \in R$
  - one variable x_{vtir} exists for all $v \in V$, $t \in \{ \text{ASAP}_v, \ldots, \text{ALAP}_v \}$, $r \in T(v)$, $i \in \{1,\ldots, a_r\}$

ILP Formulation

  - $T(v)$ is the *type set* of operation v. For our previous example, $T(*) = *$; $T(<) = \{ALU,<\}$; $T(+) = \{ALU, +/-\}$
- The module selection problem is thus choosing a single member of $T(v)$ for each $v \in V$
  - We will combine module selection, scheduling, and binding, to achieve an optimum result
- In addition to x_{vtir}, we will use a binary variable b_{ir} for each instance of each resource type
  - $b_{ir} = 1 \Leftrightarrow$ instance i of resource type r is used by *at least one* operation
  - as before, we will use c_r to denote the cost of a resource of type r

ILP Formulation

- Unlike the ILP scheduling in Lecture 11, a CDFG node does not have a fixed delay
  - it depends on which resource type implements the operation
- For this reason, we associate delays with resource types: type *r* has delay d_r
- There is at least one resource type with minimum delay $d_{\min v}$
- The ASAP and ALAP scheduling is performed by assuming each operation has its minimum delay

ILP Formulation

- We will also introduce one more symbol which will make the formulation easier to follow:
- *W* represents the set of all times that any operation could possibly start at:

$$W = \bigcup_{v \in V} \{ ASAP_v, ..., ALAP_v \}$$

Objective Function

- We are now in a position to formulate the “minimum cost” objective function:

$$\text{minimize}: \sum_{r \in R} c_r \sum_{i=1}^{a_r} b_{ir}$$

Binding Constraints

- Each operation must be mapped to a single instance of a single resource type, operating at a single time:

$$\forall v \in V, \quad \sum_{r \in T(v)} \sum_{i=1}^{a_r} \sum_{t=ASAP_v}^{ALAP_v - d_r + d_{\min v}} x_{vtir} = 1$$

- Note that an operation with ALAP time $ALAP_v$ cannot execute later than $ALAP_v - d_v + d_{\min v}$ when performed on a resource with delay d_r

Resource Constraints

- No one instance of any resource type can execute more than one operation at a time
  - indeed, if the instance is unused, no operations may execute on that instance

$$\forall t \in W, \forall r \in R, \forall i \in \{1,...,a_r\},$$

$$\sum_{v \in V : r \in T(v)} \sum_{t' \in \{t,...,t+d_r-1\} \cap \{ASAP_v,...,ALAP_v - d_r + d_{\min v}\}} x_{vt'ir} \leq b_{ir}$$

- As before, the 2nd summation is over a "time window" during which operations could overlap

Dependencies

- As previously, we need to encode each dependency in the CDFG

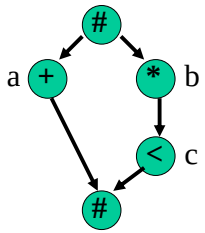
$$\forall (v', v) \in E,$$

$$\sum_{r \in T(v)} \sum_{i=1}^{a_r} \sum_{t=ASAP_v}^{ALAP_v - d_r + d_{\min v}} t \cdot x_{vtir} \geq \sum_{r \in T(v')} \sum_{i=1}^{a_r} \sum_{t=ASAP_{v'}}^{ALAP_{v'} - d_r + d_{\min v'}} (t + d_r) \cdot x_{v'tir}$$

- The main difference with the previous formulation is simply bringing the execution delay into the RHS summations, as it depends on the resource type

ILP Example

- To illustrate the method, we will complete an ILP for the simple example earlier this lecture
  - let $a_* = 1$, $a_+ = 1$, $a_< = 1$, $a_{ALU} = 2$
  - (we can't use more resource than operations of that type)
  - note that a_{ALU} is overkill, as we mentioned earlier
  - let $d_* = 2$, $d_+ = 1$, $d_< = 1$, $d_{ALU} = 2$
  - let $c_* = 2$, $c_+ = 1$, $c_< = 1$, $c_{ALU} = 1.5$
  - let $\lambda = 4$ (not a tight constraint)
  - then $ASAP_a = 0$, $ASAP_b = 0$, $ASAP_c = 2$, $ALAP_a = 3$, $ALAP_b = 1$, $ALAP_c = 3$



ILP Example

- So $W = \{0,1,2,3\} \cup \{0,1\} \cup \{2,3\} = \{0,1,2,3\}$
- Our objective function is then:

minimize :

$$2b_{1,*} + 1b_{1,+} + 1b_{1,<} + 1.5(b_{1,ALU} + b_{2,ALU})$$

ILP Example

- Binding constraints:

$$v = a: \quad x_{a,0,1,+} + x_{a,1,1,+} + x_{a,2,1,+} + x_{a,3,1,+} + x_{a,0,1,ALU} + x_{a,1,1,ALU} + x_{a,2,1,ALU} + x_{a,0,2,ALU} + x_{a,1,2,ALU} + x_{a,2,2,ALU} = 1$$

$$v = b: \quad x_{b,0,1,*} + x_{b,1,1,*} = 1$$

$$v = c: \quad x_{c,2,1,<} + x_{c,3,1,<} + x_{c,2,1,ALU} + x_{c,2,2,ALU} = 1$$

ILP Example

- Resource constraints:

$$t = 0, r = +, i = 1: \quad x_{a,0,1,+} \le b_{1,+}$$

$$t = 1, r = +, i = 1: \quad x_{a,1,1,+} \le b_{1,+}$$

$$t = 2, r = +, i = 1: \quad x_{a,2,1,+} \le b_{1,+}$$

$$t = 3, r = +, i = 1: \quad x_{a,3,1,+} \le b_{1,+}$$

ILP Example

- More resource constraints:

$$t = 0, r = *, i = 1: \quad x_{b,0,1,*} + x_{b,1,1,*} \le b_{1,*}$$

$$t = 1, r = *, i = 1: \quad x_{b,1,1,*} \le b_{1,*}$$

$$t = 2, r = <, i = 1: \quad x_{c,2,1,<} \le b_{1,<}$$

$$t = 3, r = <, i = 1: \quad x_{c,3,1,<} \le b_{1,<}$$

ILP Example

- More resource constraints:

$$t = 0, r = ALU, i = 1: \quad x_{a,0,1,ALU} + x_{a,1,1,ALU} \le b_{1,ALU}$$

$$t = 0, r = ALU, i = 2: \quad x_{a,0,2,ALU} + x_{a,1,2,ALU} \le b_{2,ALU}$$

$$t = 1, r = ALU, i = 1: \quad x_{a,1,1,ALU} + x_{a,2,1,ALU} + x_{c,2,1,ALU} \le b_{1,ALU}$$

$$t = 1, r = ALU, i = 2: \quad x_{a,1,2,ALU} + x_{a,2,2,ALU} + x_{c,2,2,ALU} \le b_{2,ALU}$$

$$t = 2, r = ALU, i = 1: \quad x_{a,2,1,ALU} + x_{c,2,1,ALU} \le b_{1,ALU}$$

$$t = 2, r = ALU, i = 2: \quad x_{a,2,2,ALU} + x_{c,2,2,ALU} \le b_{2,ALU}$$

ILP Example

- Dependency constraint:

$$v' = b, v = c: \quad 2x_{c,2,1,<} + 3x_{c,3,1,<} +$$
$$2x_{c,2,1,ALU} + 2x_{c,2,2,ALU} \geq (0+2)x_{b,0,1,*} + (1+2)x_{b,1,1,*}$$

Summary

- This lecture has covered
  - The module selection problem
  - Module selection / scheduling / binding interaction
  - An ILP formulation
- Next lecture we will examine the retiming problem.

Suggested Problems

- Download a copy of lp_solve from the website given at the start of Lecture 11, and solve the ILP example
  - what is the minimum possible cost? (*)
  - how many adders, multipliers, comparators and ALUs does it use? (*)
  - how many variables and constraints are there? (*)
  - how do you think the number of variables and constraints vary with the size of the CDFG? (***)

Retiming

- The final portion of the course covers
  - Scheduling and retiming
  - Resource sharing algorithms
  - Floorplanning
  - Function Approximation
  - Perspectives for the future
- This lecture covers
  - Retiming: motivation and definitions
  - Delay-weighted DFGs
  - Retiming for clock period minimization

Motivation

- Our concentration so far has been on synthesising "straight-line code" or single loop iterations
- We have also briefly generalized this using CDFGs
- Often, algorithms will contain loop-carried dependencies, e.g. this IIR filter:

```
a = 0; b = 0; c = 0;
while( true ) {
 read x;
 y = x + a;
 a' = 0.1*b + 0.2*c;
 b' = y;
 c' = b;
 a = a'; b = b'; c = c';
 write y;
}
```

An IIR filter with transfer function

$$H(z) = \frac{1}{1 - 0.1z^{-2} - 0.2z^{-3}}$$

Motivation

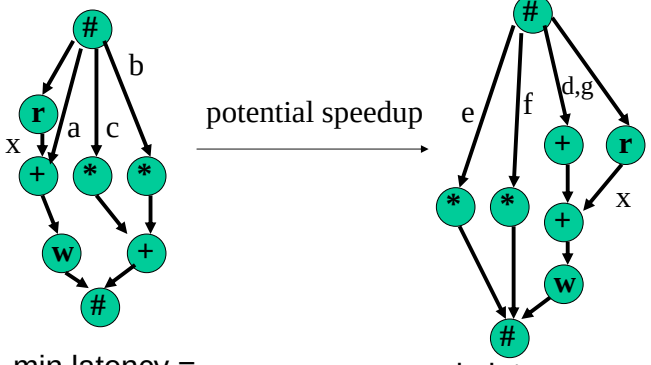
- There is an alternative way of writing this code:

```
d = 0; e = 0; f = 0; g = 0;
while( true ) {
 read x;
 y = x + d + g;
 d' = 0.1*e;
 e' = y;
 f' = e;
 g' = 0.2*f;
 d = d'; e = e'; f = f'; g = g';
 write y;
}
```

(We will soon see how you can prove the equivalence)

Motivation

- Comparing the CDFGs of the two inner loops, we can see that they may have different minimum latency.

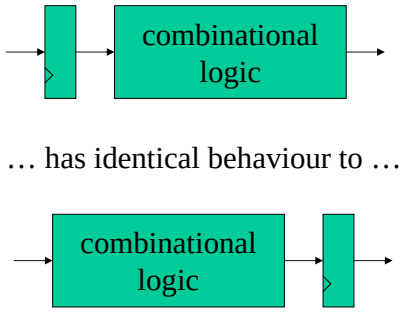


min latency = $\max\{T_r+T_w, T_*\}+T_+$

min latency = $\max\{T_*, 2T_+ + T_w, T_r+T_+ + T_w\}$

Retiming an operator

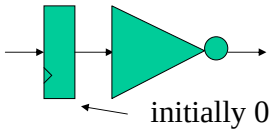
- This type of code transformation is called *retiming*, and derives from the following simple observation:



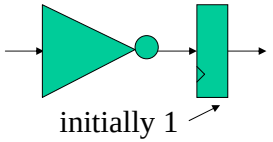
- We can move a register through an operation without affecting the "outside world" view of behaviour

The initialization problem

- We must, however, give some thought to the initialization of the system
- For example,



… has identical behaviour to …



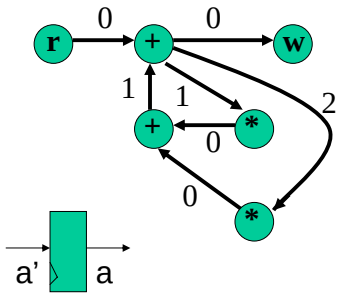
- This is fine for forward retiming, i.e. moving the register from an input to an output.
- Backward retiming requires there to be an appropriate set of inputs that generate the desired output

The delay-weighted DFG

- To be able to formally reason about retiming issues, we need to represent the entire loop as a form of DFG, including information on loop-carried dependencies.
- We will do this by an edge-weighted DFG, where each edge weight represents the number of iterations delay on that edge. We will call this a *delay-weighted DFG*.
- Note that when we have a loop-carried dependency, the delay-weighted DFG will contain a cycle.

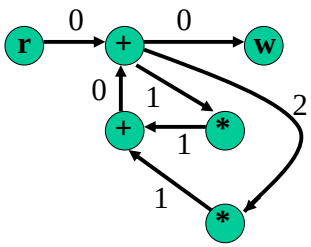
Delay-Weighted DFG

```
a = 0; b = 0; c = 0;
while( true ) {
  read x;
  y = x + a;
  a' = 0.1*b + 0.2*c;
  b' = y;
  c' = b;
  a = a'; b = b'; c = c';
  write y;
}
```



- This is our original example and its delay-weighted DFG
- Noting that the only output of the lower adder has weight 1, we can retime backwards across this adder, resulting in…

Delay-Weighted DFG




```
d = 0; e = 0; f = 0; g = 0;
while( true ) {
  read x;
  y = x + d + g;
  d' = 0.1*e;
  e' = y;
  f' = e;
  g' = 0.2*f;
  d = d'; e = e'; f = f'; g = g';
  write y;
}
```

- … which corresponds to our modified example

Approaching the problem

- We can associate the nodes V with a retiming value r: $V \rightarrow Z$ which denotes the number of clock cycles that node has been moved "forwards in time"
- If we denote by w: $E \rightarrow Z$ the original weight, and w_r: $E \rightarrow Z$ the retimed weight, then
  for all $(u,v) \in E$, $w_r(u,v) = w(u,v) + r(v) - r(u)$
- A *feasible* retiming is one for which for all $(u,v) \in E$, $w_r(u,v) \geq 0$ (since we can't have a negative number of registers)

Retiming for Clock-Period Min

- There are several reasons why we may wish to retime, including for speed and for minimization of registers.
- We will address retiming for clock-period minimization, i.e. clock frequency maximization.
- The maximum clock frequency is determined by the worst-case combinational delay between any two registers, or from an input to a register, or from an register to an output.
- Let us denote by $d(v)$ the combinational delay of node v, and we will assume all nodes are combinational.

Retiming problem formulation

- We must therefore have the notion of a combinational path, i.e. a path that does not pass through any registers.
  - $w_r(u,v) = 0 \Rightarrow$ combinational path.

An ILP Solution

- We can modify the LP for longest-path given in Lecture 8 to:
- Minimize L s.t.

$$s_v \geq s_u + d(u) + w_r(u,v)N \text{ for all } (u,v) \in E \quad (1)$$

$$s_v + d(v) \leq L \text{ for all } v \in V \quad (2)$$

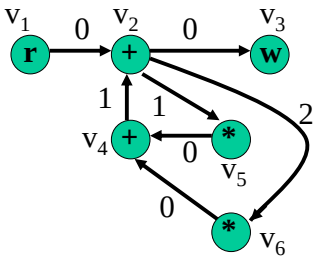
$$w_r(u,v) = w(u,v) + r(v) - r(u) \geq 0 \text{ for all } (u,v) \in E \quad (3)$$

$$r(v) \in Z \text{ for all } v \in V \quad (4)$$

An ILP Solution

- Here *N* is a "large-enough" negative number.
- L corresponds to the longest combinational path, a fact guaranteed by (2), which ensures it is at least as large as the largest (s_v + delay of node *v*).
- (1) is simply an extension of Bellman's equations. If $w_r(u,v) = 0$, it is a direct implementation of Bellman's. $w_r(u,v) > 0$, (1) is satisfied no matter what (due to N being large, and w_r being integer (4)).
- Finally, (3) combines the definition of $w_r(u,v)$ with the feasibility constraint.

Example



- Let's say $d(v_2) = d(v_4) = 1$, $d(v_1) = d(v_3) = 0$, $d(v_5) = d(v_6) = 2$
- If the retiming left the graph unchanged, then $r(v_1)=r(v_2)=r(v_3)=r(v_4)=r(v_5)=r(v_6)=0$
- It should be easily verifiable that (1)-(4) are satisfied in this case, with $s_{v1} = 0$, $s_{v2} = 0$, $s_{v3} = 1$, $s_{v4} = 2$, $s_{v5} = 0$, $s_{v6} = 0$, $L = 3$

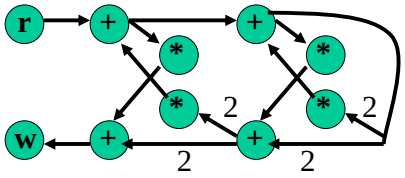
- The retimed example also corresponds to a feasible solution, with $s_{v1} = 0$, $s_{v2} = 1$, $s_{v3} = 2$, $s_{v4} = 0$, $s_{v5} = 0$, $s_{v6} = 0$, $L = 2$: an improvement!

Summary

- This lecture has covered
  - Retiming: motivation and definitions
  - Delay-weighted DFGs
  - Retiming for clock-period minimization
- The next lecture will investigate the floorplanning problem.

Suggested Problems

- Is the retiming shown in the example optimal?
- The edge-weighted DFG of a two-stage lattice filter is shown below: retime the DFG to improve the clock rate given that the delay of a multiplier is 2ns, the delay of an adder is 1ns, and the delay of an I/O node is 0ns.



(unlabelled edges have zero weight)

Floorplanning

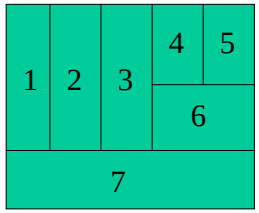
- The final portion of the course covers
  - Scheduling algorithms
  - Resource sharing algorithms
  - Module selection
  - Retiming
  - Floorplanning
  - Function approximation
  - Perspectives for the future
- This lecture covers
  - The floorplanning problem
  - Slicing and non-slicing floorplans and representations
  - Heuristic and ILP solutions

Motivation

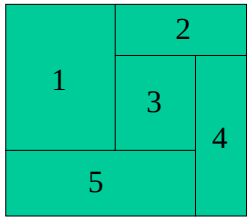
- In recent years, we have moved to deep sub-micron design.
- Wiring delays have started to compete with (and sometimes overtake) logic delay.
  - it is important to be able to estimate wiring delay early in the design process.
- We need an early idea of geometrical layout on silicon
  - a *floorplan*.
- Floorplanning becomes part of architectural synthesis.

Slicing Floorplans

- Floorplans are typically categorised into
  - slicing floorplans or non-slicing floorplans
- Slicing floorplan
  - obtainable by repeated bisection of rectangular cells
  - simplifies representation and optimization



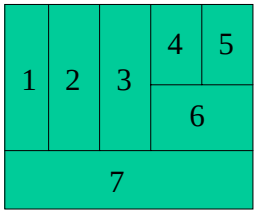
A slicing floorplan

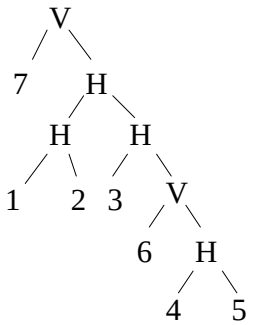


A non-slicing floorplan

Slicing Tree Representation

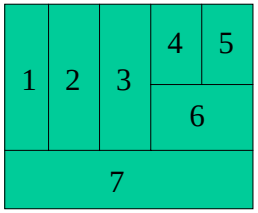
- A *slicing tree* is a binary tree representation of a slicing floorplan
  - a leaf is a resource to be floorplanned
  - other nodes indicate how to compose their children: vertically, or horizontally.

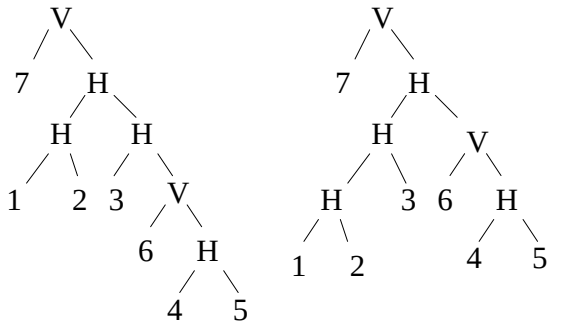




Skewed Slicing Trees

- Unfortunately, slicing trees are not unique representations of the floorplan.



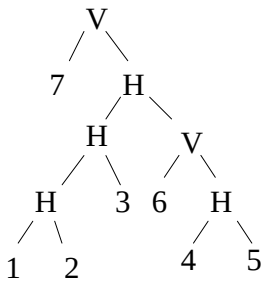


Both slicing trees are valid representations

Skewed Slicing Trees

- A skewed slicing tree has the following property
  - no node and its right-child have the same type
- Every slicing floorplan has a unique skewed slicing tree.
- How to represent the trees in a floorplanning algorithm?
  - we can represent it as a string, called a *Polish expression*.

Polish Expressions



- Polish expression for:
  Polish(Y)+Polish(Z)+“X”

- Polish expression for leaf is leaf value.
- For tree on the left:
  “712H3H645HVHV”

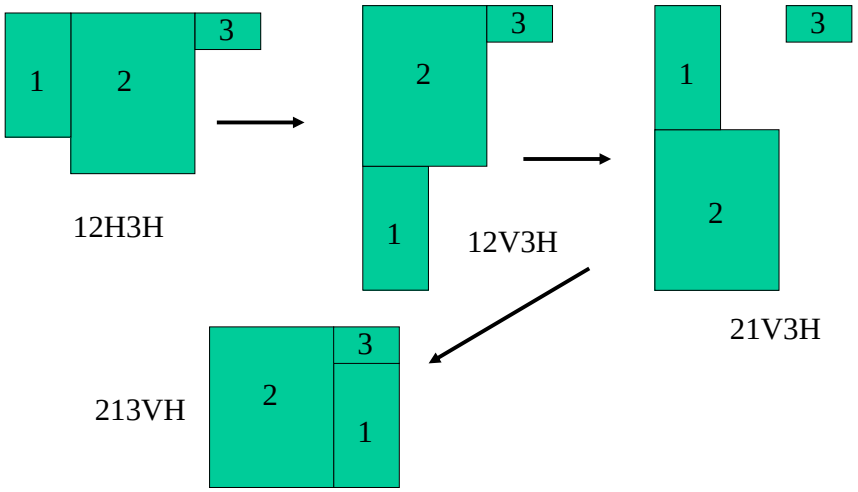
- A skewed slicing tree corresponds to a Polish expression where
  - no two consecutive operators (H/V) are of the same type.

Floorplan Optimization

- We have a compact and unique representation of a slicing floorplan. How to optimize for smallest area?
- A common approach:
  - start with a random floorplan
  - improve it based on certain well-defined “moves”
- What moves[1]?
  - Swap two adjacent operands (leaf nodes) in the Polish expression.
  - Take a chain of consecutive operators, e.g. “HVHV”, and complement it, e.g. “VHVH”.
  - Swap an adjacent operator and operand. (But make sure still a skewed tree!)

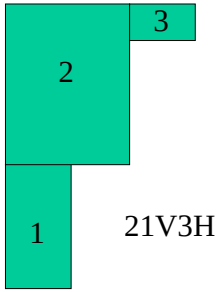
[1] Moves from Prof. Hai Zhou

Floorplan Optimization



Area Computation

- How to tell whether a move improves area?
  - Height(XYH) = max(Height(X), Height(Y))
  - Width(XYH) = Width(X) + Width(Y)
  - Height(XYV) = Height(X) + Height(Y)
  - Width(XYV) = max(Width(X), Width(Y))



Height(21V3H) = max(Height(21V), Height(3))
= max(Height(2) + Height(1), Height(3))

Width(21V3H) = Width(21V) + Width(3)
= max(Width(2), Width(1)) + Width(3)

Simulated Annealing

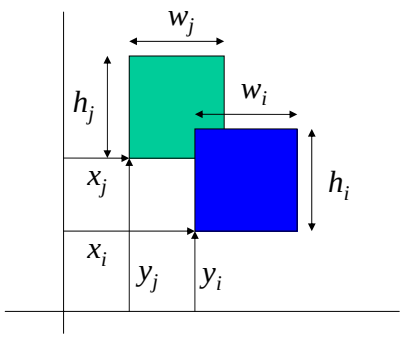
- In our example, not all moves improved area
  - not good enough to just “pick the best move” each time
- *Simulated annealing* is often used
  - pick a move at random.
  - if it improves area, do it.
  - if it doesn’t improve area, *maybe* do it.
- Probability of selecting a move that does not improve area
  - reduces with area penalty for move
  - decreases (for a fixed area penalty) with iteration number

An ILP Approach

- We can also take an ILP approach to the floorplanning problem
  - guaranteed optimal solutions
  - slicing and non-slicing floorplans within a single framework
  - poor execution-time scaling

An ILP Approach

- Resources cannot overlap



$$x_i \geq x_j + w_j \quad (1)$$
$$x_j \geq x_i + w_i \quad (2)$$
$$y_i \geq y_j + h_j \quad (3)$$
$$y_j \geq y_i + h_i \quad (4)$$

- We need to ensure that *at least one* of (1)-(4) holds

An ILP Approach

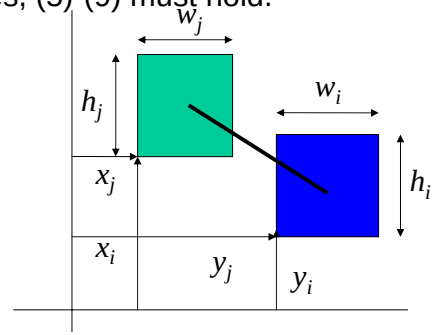
- Although each constraint is linear, "at least one of" causes us a problem.
- A solution: *all* constraints below hold.
  - P is a big enough positive number, e.g. max chip dimension. For all $(i,j) \in R^2$, (1) to (4) must hold.

$$x_i + P\delta_{ij} + P\eta_{ij} \geq x_j + w_j \quad (1)$$
$$x_j + P(1 - \delta_{ij}) + P\eta_{ij} \geq x_i + w_i \quad (2)$$
$$y_i + P\delta_{ij} + P(1 - \eta_{ij}) \geq y_j + h_j \quad (3)$$
$$y_j + P(1 - \delta_{ij}) + P(1 - \eta_{ij}) \geq y_i + h_i \quad (4)$$
$$\delta_{ij},\ \eta_{ij} \in \boldsymbol{B}$$

Good Floorplanning

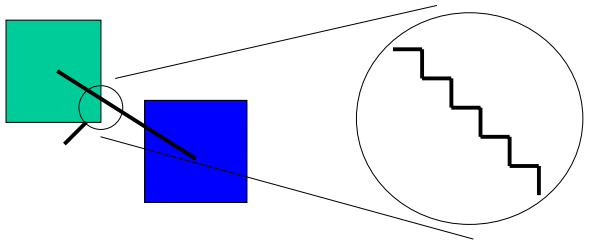
- Some floorplans are better than others
  - place resources that communicate close to each other.
- Given a maximum wire-length W_{ij} for each pair $(i,j) \in R^2$ of connected resources, (5)-(9) must hold.

$$x_i + 0.5w_i - x_j - 0.5w_j \leq {W_{ij}}^h \quad (5)$$
$$-x_i - 0.5w_i + x_j + 0.5w_j \leq {W_{ij}}^h \quad (6)$$
$$y_i + 0.5h_i - y_j - 0.5h_j \leq {W_{ij}}^v \quad (7)$$
$$-y_i - 0.5h_i + y_j + 0.5h_j \leq {W_{ij}}^v \quad (8)$$
$$W_{ij} = {W_{ij}}^h + {W_{ij}}^v \quad (9)$$



Good Floorplanning

- Constraints (5) & (6) ensure that horizontal wirelength is no more than ${W_{ij}}^h$.
  - (7) and (8) perform a similar function for vertical wirelength.
- Constraint (9) expresses total wirelength in terms of Manhattan distance.



- ✓ appropriate for most design rules
- ✓ linear

Design Area

- We must ensure that the design fits in chip dimensions X by Y.
  - For all resources $i \in R$, (10) and (11) must hold.

$$x_i + w_i \leq X \qquad (10)$$

$$y_i + h_i \leq Y \qquad (11)$$

- If the chip *aspect ratio* is given, $Y = kX$ (12).
  - Objective is then min: X
- If aspect ratio is not given, we have min: XY
  - problem: nonlinear objective

Linearization

- Two standard approaches
  - iterate: solve "min: X" with Y fixed, many times for different values of Y.
  - approximate:
    $XY \approx X' Y' + (X - X')Y' + (Y - Y') X'$ for
    $X \approx X'$ and $Y \approx Y'$.
  - (or some combination of the two).
- More recently, convex (nonlinear) optimization techniques have started to appear.

ILP Approaches

- The approach has a (very) large execution time: $O(n^2)$ integer variables.
  - techniques have been proposed to break down into sub-problems[1].
  - sub-problems can be stitched into suboptimal solutions.

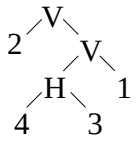
[1]Sutanthavibul, Schragowitz, and Rosen, IEEE Trans CAD 10(6), 1991.
Smith, Constantinides, and Cheung, Proc. Field-Programmable Logic, 2005 (in the context of FPGA design).

Summary

- This lecture has introduced floorplanning
  - motivation: deep-submicron era
  - slicing vs non-slicing floorplans
  - Polish expressions
  - optimizing moves
  - an ILP approach
- The next lecture will look at function approximation.

Suggested Problems

- Draw the floorplan represented by the following slicing tree:



- Convert this tree into a skewed slicing tree.
- Write the Polish expression for the skewed tree.
- Identify one of the three moves proposed in this lecture that could be applied to obtain an optimal area floorplan for the given resource dimensions.
  - Resource 1: Height = 2, Width = 2
  - Resource 2: Height = 2, Width = 1
  - Resource 3: Height = 1, Width = 1
  - Resource 4: Height = 1, Width = 1

Beyond Mults and Adds

- The final portion of the course covers
  - Scheduling and retiming
  - Resource sharing algorithms
  - Function Approximation
  - Floorplanning
  - Perspectives for the future
- This lecture covers
  - Polynomial approximations
  - Evaluation methods
  - Approximation methods

Function Evaluation

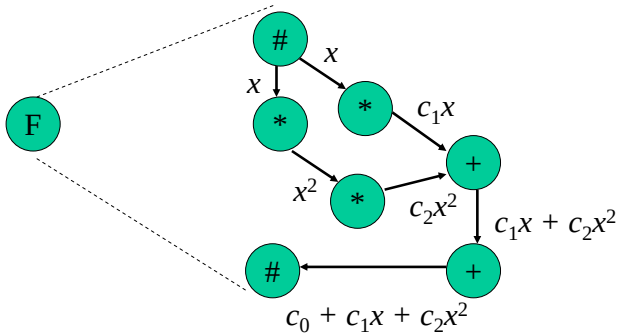
- Throughout much of the course, we have used multiplication and addition as the key operations
- There are typically pre-designed library blocks for adder and multiplier resources
- Not necessarily the case for more complex functions: sin(x), cos(x), e^x, etc.
- In this lecture we investigate how to evaluate these functions

Polynomial Approximations

- Let us return to our main operations: addition, and multiplication
- What different functions of a variable x can be produced through addition and multiplication alone?
  - polynomials in x
  - $f(x) = c_0 + c_1x + c_2x^2 + \ldots + c_nx^n$
- This suggests a solution to our problem: find a polynomial "close enough" to the function, and then use mults and adds to evaluate it

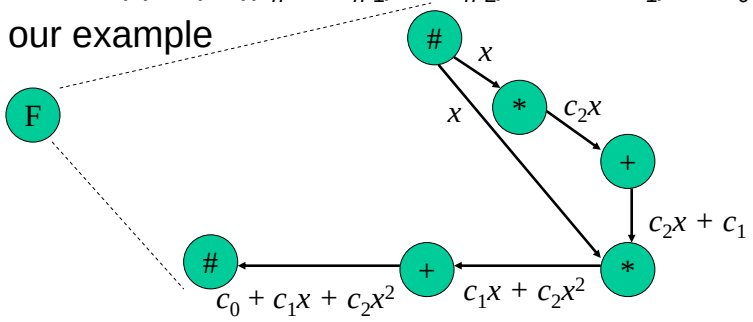
A Simple Evaluation Scheme

- Let's use a 2nd order polynomial as an example
  - $f(x) = c_0 + c_1x + c_2x^2$
  - how can we evaluate this polynomial?



Horner's Scheme

- Horner's scheme is a method to reduce the number of operations involved
  - $f(x) = c_0 + c_1x + c_2x^2 + \ldots + c_nx^n$
  - re-write: $f(x) = (\ldots((c_nx + c_{n-1})x + c_{n-2})x + \ldots + c_1)x + c_0$
- For our example



Finding Polynomial Coefficients

- For any function f(x), we want to find the set of polynomial coefficients so that the polynomial function g(x) is "close enough" to f(x)
- What is "close enough"? Could be:
  1. to within a worst case error ε, i.e. $\max_x |f(x) - g(x)| < \varepsilon$
  2. in the least-squares sense, i.e.

$$\int_x w(x)(f(x) - g(x))^2 dx < \varepsilon$$

  - w(x) is a "weight" function, which allows us to place greater emphasis on errors some ranges of *x*

Least-Squares Approximations

- We can construct

$$g(x) = \sum_{i=0}^{n} a_i \phi_i(x)$$

  - where $\phi_i(x)$ is a known polynomial of degree i
- If we choose a set of *orthogonal* polynomials $\phi_i(x)$, i.e.

$$\forall i \neq j, \quad \int_x \phi_i(x)\phi_j(x)dx = 0$$

- Then it is easy to calculate a_i

Least-Squares Approximations

- If we define the inner product

$$< f, g >= \int_x f(x)g(x)dx$$

- Then the coefficients minimizing the least-squares error are

$$a_i = \frac{< f, \phi_i >}{< \phi_i, \phi_i >}$$

Least-Squares Approximations

- Proof: We are trying to minimize

$$E = \int_x \left(f(x) - \sum_{i=0}^{n} a_i \phi_i(x) \right)^2 dx$$

$$= \int_x f^2(x) - 2\sum_{i=0}^{n} a_i \int_x f(x)\phi_i(x)dx + \sum_{i=0}^{n}\sum_{j=0}^{n} a_i a_j \int_x \phi_i(x)\phi_j(x)dx$$

$$= \int_x f^2(x) - 2\sum_{i=0}^{n} a_i < f, \phi_i > + \sum_{i=0}^{n} a_i^2 < \phi_i, \phi_i >$$

Least-Squares Approximations

- Proof (cont'd): Differentiate w.r.t. a_i and set equal to zero

$$\frac{\partial E}{\partial a_i} = -2 < f, \phi_i > +2a_i < \phi_i, \phi_i >= 0$$

$$\Rightarrow a_i = \frac{< f, \phi_i >}{< \phi_i, \phi_i >}$$

- This ease of derivation makes least-squares solutions popular

Legendre Polynomials

- There are many sets of orthogonal polynomials with different properties
- Two common ones are the Legendre and the Chebyshev-I polynomials, both defined over [-1,1]
- Legendre polynomials have a weight $w(x) = 1$ and can be defined by

$$\phi_i(x) = \frac{1}{2^i i!} \frac{d^i}{dx^i} (x^2 - 1)^i$$

Chebyshev Polynomials

- Chebyshev-I polynomials have weighting function $w(x) = (1-x^2)^{-1/2}$ and can be defined by:

$$\phi_i(x) = 2^{i-1} \prod_{k=1}^{i} \left\{ x - \cos\left[\frac{(2k-1)\pi}{2i} \right] \right\}$$

- Your choice of orthogonal polynomials should depend on which parts of the function domain you require to be highly accurate

Summary

- This lecture has covered
  - Polynomial approximations
  - The Horner's scheme evaluation method
  - Least squares approximation
  - Legendre and Chebyshev-I orthogonal polynomials
- In the next lecture, we will discuss floorplanning.
- The work by my ex-Ph.D. student Dr. Nalin Sidahao was used extensively to prepare this lecture.

Suggested Problems

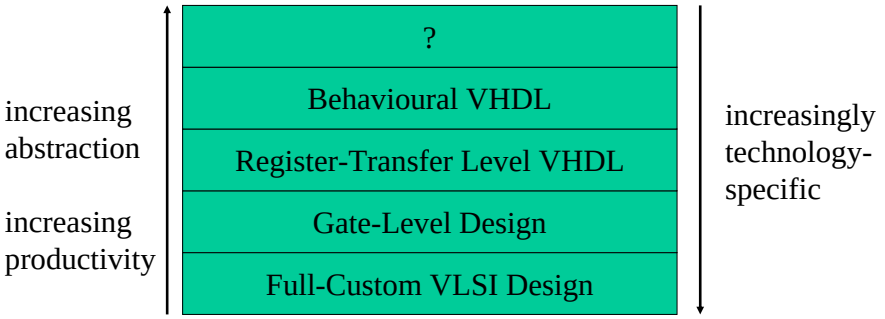
- What is the least-squares error when fitting the function $f(x) = \sin(\pi(x+1)/4)$ over [-1,1] using a polynomial of 3rd order constructed as a weighted sum of Legendre polynomials?
- Derive a formula for the number of multipliers required using Horner's scheme for polynomial evaluation
- The critical path of the Horner's scheme evaluation can be reduced, possibly at the cost of more operations, by different approaches. Can you derive one such scheme?

Perspectives I

- The final portion of the course covers
  - Scheduling and retiming
  - Resource sharing algorithms
  - Function Approximation
  - Floorplanning
  - Perspectives for the future
- This lecture (part one of two) covers
  - Abstract design representations
  - Word-length optimization
  - Number representations

Levels of Abstraction in Design

- Most of our examples have used a C-like imperative language as the original design specification



Why [not?] C

- One of the main candidates for "?" on the previous slides is C
- Advantage: There are lots of C programmers, and even more C code
- Disadvantage: C was designed for a single processor
  - no concept of parallelism, so we would need to automatically detect all parallelism
  - sometimes C is not a natural representation – we have had to sequentialize an algorithm, only to have to re-parallelize it

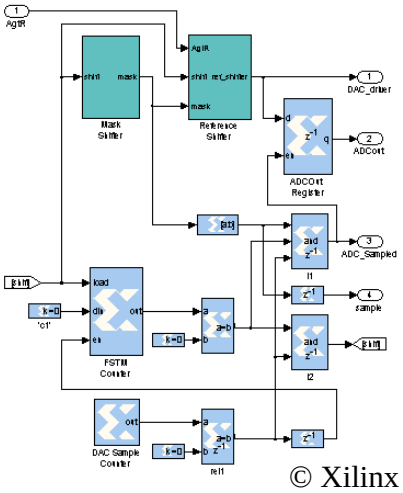
Why [not?] C

- One compromise is to extend C
  - Celoxica (http://www.celoxica.com) has a product for synthesis from "C with extensions"
  - You can add explicit parallelism with the "par" keyword
- Some aspects of C are particularly troublesome for automatic analysis and efficient hardware generation
  - Synthesis of code containing pointers has only recently been addressed (c. 2000) (http://akebono.stanford.edu/users/nanni/research/sys)
  - For this reason, pointerless Java has been sometimes suggested as an alternative

Simulink

- I believe a more promising approach is to target specific problem domains
  - Simulink is widely used in Control and DSP, so use it as a specification format in these domains
  - We have developed a tool for synthesis from Simulink (http://cas.ee.ic.ac.uk/~gac1/)
  - Recently technology manufacturers are getting interested in this approach (http://www.xilinx.com/xlnx/xil_prodcat_product.jsp?title=system_generator)

Example in Simulink



© Xilinx

- Already in DFG form!
- Modelling loops, etc. is not as natural
- Ideal for data-intensive applications
  - DSP
  - Communications

Matlab

- Probably the widest used tool for DSP algorithm development
- Has complex control structures (while, etc) like C
  - so comparatively hard to map efficiently
  - also has implicit parallelism in matrix statements, e.g. A = B + C for matrices: each element can be done in parallel – in C, we would have to write as a loop
- A Matlab-based synthesis tool is in development at Northwestern University (http://www.ece.northwestern.edu/cpdc/Match/Match.html)

Mathematical Specifications

- Possibly the “ultimate” future for synthesis of DSP systems
- DSP algorithms are typically defined as a set of equations
  - a designer will then map this to a Matlab or Simulink description
- We could aim higher – for direct synthesis from the equations themselves
  - plenty of scope for research here!

Word-Length Optimization

- Simulink, Matlab, some C and mathematical specifications share something not present in hardware languages
  - in numerical computations, often everything is a high-precision floating point number
  - for hardware, we want to trim the precision down the the minimum (high speed, low area, low power)
- Word-length optimization problem:
  - Choose a suitable word-length for each internal variable, in order to minimize area (or power, or maximize speed) *subject to acceptable arithmetic error*

Word-Length Optimization

- This problem is one of my original research areas
- Our research has produced two tools (Synoptix, Right-Size)
  - synthesizes a low-area implementation by selecting the internal word-lengths appropriately
  - input format is Simulink
  - output format is structural VHDL
  - http://cas.ee.ic.ac.uk/~gac1
  - LTI systems, differentiable nonlinear systems
- Actively researching the use of word-length optimization for power consumption minimization
  - EPSRC funded research, Dr. Altaf Abdul Gaffar and Mr. Jonathan Clarke.

Logarithmic Representations

- Using standard two's complement representation is not always the most efficient
- In an algorithm with many additions but few divisions and multiplies, standard representation may suffice
- In an algorithm with few additions but many multiplies and divisions, a logarithmic representation may be better
  - log(a/b) = log(a) – log(b); log(ab) = log(a) + log(b)
- We may still have to do conversion in and out of log-form
  - overheads could outweigh advantages

Residue Number Systems

- Residue number systems also may be a possible route to fast circuitry
- Choose n relatively prime numbers m_1, m_2, …, m_n
- Represent x as a list (x mod m_1, x mod m_2, …, x mod m_n)
  - we can represent up to $m_1 m_2 \ldots m_n$ numbers uniquely like this
  - we can perform arithmetic on the list of numbers, e.g. for n=2, m_1=3, m_2=5: 4 = (1,4), 3 = (0,3), 4*3 = (1*0,4*3) = (0,12 mod 5) = (0, 2)

Residue Number Systems

- Key point: We can do arithmetic on each of the list elements *in parallel*
  - if max($\lceil \log_2 m_1 \rceil, \lceil \log_2 m_2 \rceil, \ldots, \lceil \log_2 m_n \rceil$) < $\lceil \log2(m_1 m_2 \ldots m_n) \rceil$, we can get speed advantages
  - the delay of an arithmetic component depends on the worst-case delay of each list element
  - for our example, max($\lceil \log_2 3 \rceil, \lceil \log_2 5 \rceil$) = 3 < 4 = $\lceil \log_2 15 \rceil$
  - however the area of the design may increase
  - for our example, we need a 2-bit and a 3-bit adder rather than a single 4-bit adder (roughly 25% larger)

Number System Selection

- Ideally, a synthesis tool would select automatically which portions of the circuit are best implemented using
  - standard bit-parallel representation
  - bit-serial representation (or something between)
  - logarithmic representation
  - residue representation
  - fixed point
  - floating point (IEEE standard – or something else?)
- Such a tool would have to take into account the overhead of converting from one format to another
- This is an open research topic

Summary

- This lecture (part one of two) has covered
  - Abstract design representations
  - Word-length optimization
  - Number representations
- Next lecture will continue to examine some future directions for architectural synthesis

Perspectives II

- The final portion of the course covers
  - Scheduling and retiming
  - Resource sharing algorithms
  - Function Approximation
  - Floorplanning
  - Perspectives for the future
- This lecture (part two of two) covers
  - Function approximation
  - Mathematical transformations
  - Hardware / Software partitioning
  - Memory synthesis
  - Synthesis of Reconfigurable Architectures

Function Approximation

- During this lecture course, we have often used multiplication and addition as exemplary operations
- Sometimes we are interested in incorporating more complex functions like $\sin(x)$ or $e^{\cos(x)}$
- We could simply extend our current approach, if we have a library of designs for such functions
  - however there are many different methods for implementing a given function in hardware
  - we could use a ROM as a lookup-table
  - we could express the function using a polynomial approximation, and then implement it using adds and mults

Function Approximation

  - we could express the function using a rational approximation, and then implement it using adds, mults, and a divide

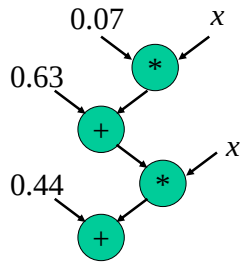
- Simple lookup table approach:

Size $\propto m2^n$
Speed $\propto 1/n$
Error $\propto 2^{-m}$ + a complex dependence on n

- Choose m and n to trade-off area/error/speed

Function Approximation

- Polynomial approximation:
  - Over [1,2], $\text{sqrt}(x) \approx 0.44 + 0.63x + 0.07x^2$
    $= 0.44 + x(0.63 + 0.07x)$



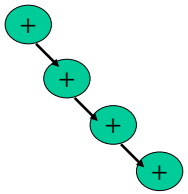
- Many tradeoffs are possible
  - how many bits used to represent coefficient?
  - how many bits to represent internal variables?
  - how many polynomial terms?
  - what type of approximation?
    - worst-case, or average case?

Function Approximation

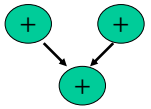
- Different solutions will have different area, arithmetic error, power, and speed characteristics
- The challenge is to decide automatically when to use which type of function approximation
  - we have started to investigate this issue (Dr Nalin Sidahao and Mr Gareth Morris)

Mathematical Transformations

- There are certain mathematical transformations which may be used to obtain different speed / area tradeoffs
- For a simple example, ((a+b)+c)+d = (a+b) + (c+d)
  - addition is associative
- Comparing the LHS and RHS as DFGs,



Can be scheduled in 4 time units using a single adder



Can be scheduled in 2 time units, if we use two adders

Mathematical Transformations

- Another typical transformation is “strength reduction”
  - try to replace high-area / low-speed / high-power operators by a combination of low-area / high-speed / low-power operators
- For example $127x \rightarrow 128x - x = (x<<7) - x$
  - “<<7” represents a left-shift by 7 bits
  - shifting in hardware is cheap: just wires
  - subtraction is cheap
  - multiplication is expensive

Mathematical Transformations

- The challenge is to decide, given constraints on area, error, power and speed for the overall design, which transformations to apply where
- There may be hidden pitfalls
  - just because a transformation is valid for real numbers doesn’t make it valid for binary representations
  - in an 8-bit 2’s complement representation, numbers can range from –128 to 127. (120+120)-150 may flag an overflow, but (120-150)+120 won’t

Hardware / Software Partitioning

- Large scale designs of embedded systems typically have a hardware portion and a software portion
- The designer must decide which tasks are best done in software, and which in hardware
  - software can be slow, power-hungry, and cheap
  - hardware can be fast, power-efficient, and expensive
  - hardware can only be significantly faster if the application can be parallelized
- Could this task be done automatically?
  - Our research group has been addressing this problem for configurable hardware based on Field-Programmable Gate Arrays (FPGAs) [Dr. Theerayod Wiangtong]

Memory Synthesis

- We have concentrated in the course on the area, speed, and power associated with arithmetic units
- In many applications, memory accesses consume significant power and slow down the application
- Memory itself can also consume a significant proportion of silicon area
- Recently, our research group has been investigating ways to use memory more efficiently
  - what variables should be stored where in memory in order to minimize power consumption? (Dr. Sambuddhi Hettiaratchi)
  - How to design customised parallel caches which match the characteristics of the algorithm (Mr. Su-Shin Ang)

Synthesis of Reconfigurable Architectures

- We have covered techniques to synthesise application specific architectures.
  - this architecture could then be implemented on an ASIC (expensive for small volume!)
  - or on an FPGA (expensive for large volume)
- FPGAs are cost effective for small volumes
  - able to spread fixed costs over a large range of designs
  - but how to decide the architecture of the FPGA *itself*?
- Fixed-function blocks: multipliers, RAMs
  - limited flexibility, high performance, small footprint
- What proportion of multipliers, RAMs, fine-grain logic, and other components are appropriate?
  - Synthesise an FPGA architecture suitable for synthesising AS architectures!
  - New and exciting research field. (Mr. Alastair Smith).

Summary

- This lecture (part two of two) has covered
  - Function approximation
  - Mathematical transformations
  - Hardware / Software partitioning
  - Memory synthesis
  - Reconfigurable architectures
- Next lecture will summarize the entire course, and allow you to focus on topics for revision