

Synthesis of Digital Architectures

- Self-contained course
 - no previous requirements beyond compulsory courses
 - all material is described in these notes and during the lectures
- Course materials will draw on several texts
 - Giovanni De Micheli, “Synthesis and Optimization of Digital Circuits”, McGraw-Hill, 1994
 - A good description of most of the topics covered
 - Also covers logic synthesis, not covered in this course
 - Keshab K. Parhi, “VLSI Digital Signal Processing Systems”, Wiley-Interscience, 1999
 - Useful for retiming, and a slightly different perspective
 - Sabih H. Gerez, “Algorithms for VLSI Design Automation”, Wiley, 1999.
 - Useful for a general overview, and some details on floorplanning
 - M. McFarland, A. Parker, R. Camposano, “The High-Level Synthesis of Digital Systems”, Proc. IEEE, Vol. 78, No. 2, Feb 1990
 - R. Camposano, “From Behavior to Structure: High-Level Synthesis”, IEEE Design & Test of Computers, October 1990.

1/8/2007

Introductory Lecture

gac1

1

Administrivia

- Approximately 20 1-hour lectures
- An assessed “mini project”
 - small groups develop some synthesis software
- Course website
 - <http://cas.ee.ic.ac.uk/~gac1/Synthesis>
- Room 903, gac1@ic.ac.uk
 - Questions / discussion welcome, but please email first!

1/8/2007

Introductory Lecture

gac1

2

What is Synthesis?

- Synthesis is the automatic mapping from a high-level description to a low-level description
 - gates to transistors
 - AHDL or VHDL to gates
 - Matlab to AHDL/VHDL (?)
- Synthesis is important because
 - it raises designer productivity
 - it allows design-space exploration
 - it improves time-to-market
 - it is a very big industry
- Synthesis is also a fun real-world application of some nice “game-like” parts of mathematics

1/8/2007

Introductory Lecture

gac1

3

What is architectural synthesis?

- Current synthesis comes in two main “flavours”, depending on what is the input description and what is the output description
 - logic synthesis
 - given Boolean equations, map them into gates
 - architectural (“high-level”) synthesis
 - given a description of circuit behaviour, create an architecture
- This course will give you a good understanding of architectural synthesis

1/8/2007

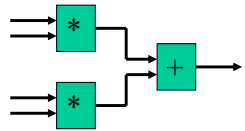
Introductory Lecture

gac1

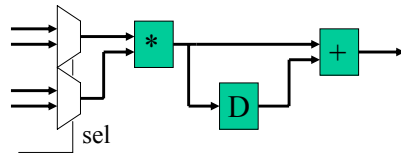
4

Example: Architectural Synthesis

- Problem:
 - create a circuit capable of implementing the behaviour “ $y[n] := a[n] * b[n] + c[n] * d[n]$ ”
- Possible solutions:



(a) fast, big
one result per cycle



(b) slower, smaller (?)
one result per 2 cycles

Course Syllabus

- Introduction to architectural synthesis
 - scheduling
 - when should I execute each operation?
 - resource allocation and binding
 - how many of each computational unit should I use in my design, and which unit should do which task?
 - area and performance estimation
 - how big will my design be and how fast will it run?
 - control unit synthesis
 - how can I design the controller to tell each unit what it should be doing at each time?

Course Syllabus

- Introductory graph theory and combinatorial optimization
 - what is a graph, and how can we use one?
 - tractable and intractable problems
 - longest path through a graph
 - colouring graphs
 - finding complete subgraphs
 - integer linear programming

Course Syllabus

- Scheduling algorithms
 - As Soon As Possible / As Late As Possible
 - list scheduling
 - scheduling with integer linear programs
 - affine loop scheduling
 - retiming
- Resource sharing algorithms
 - interval graph colouring
 - register sharing
 - resource sharing with integer linear programs
- Other topics
 - function approximation
 - floorplanning
- Subject perspectives and revision

Introduction: Scheduling

- Part of a 4-lecture introduction
 - Scheduling
 - Resource binding
 - Area and performance estimation
 - Control unit synthesis
- This lecture covers
 - The relationship between code and operations
 - Data flow and control data flow graphs
 - Modelling of conditionals and loops
 - Resource constrained scheduling
 - Scheduling with chaining
 - Synchronization

Example Code Fragment

- Because we're engineers, we've written some code to solve a differential equation

```

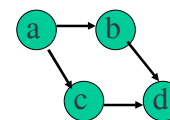
begin diffeq
  read( x, y, u, dx, a );
  repeat {
    xl = x + dx;
    ul = u - 3*x*u*dx - 3*y*dx;
    yl = y + u*dx;
    c = xl < a;
    x = xl; u = ul; y = yl;
  } until( c );
  write( y );
end diffeq
    
```

Graphs and Models

- We want to express this code in a way that maintains the essential information
- Graphs are useful for describing such models
- A graph $G(V,E)$ is a pair (V,E) , where V is a set and E is a binary relation on V .
- Elements of V are called vertices, elements of E are called edges.
- A graph can be undirected or directed depending on whether an edge is an unordered or ordered pair.

Graphs and Models

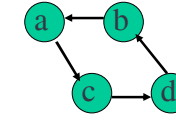
Directed Graphs



$$V = \{a,b,c,d\}$$

$$E = \{(a,b),(a,c),(b,d),(c,d)\}$$

This graph is acyclic

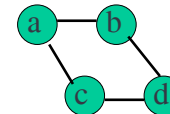


$$V = \{a,b,c,d\}$$

$$E = \{(b,a),(a,c),(d,b),(c,d)\}$$

This graph is cyclic

Undirected Graph



$$V = \{a,b,c,d\}$$

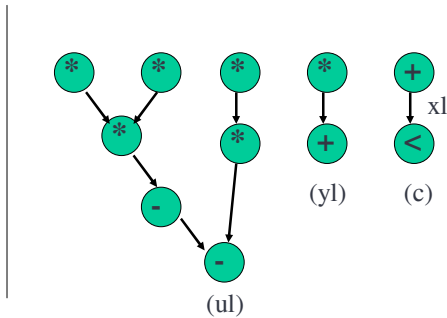
$$E = \{\{a,b\},\{a,c\},\{b,d\},\{c,d\}\}$$

Data Flow Graphs

- A data flow graph (DFG) represents the way in which data flows through a computation

```

xl = x + dx;
ul = u - 3*x*u*dx -
      3*y*dx;
yl = y + u*dx;
c = xl < a;
    
```



1/12/2006

Lecture1

gac1

5

Data Flow Graphs

- What we have done is to break up the algorithm so that we only use standard 2-input operators

```

xl = x + dx;
ul = u - 3*x*u*dx -
      3*y*dx;
yl = y + u*dx;
c = xl < a;
    
```



```

xl = x + dx;
t1 = 3*x;
t2 = u*dx;
t3 = t1*t2;
t4 = 3*y;
t5 = t4*dx;
t6 = u - t3;
ul = t6 - t5;
t7 = u*dx;
yl = y + t7;
c = xl < a;
    
```

1/12/2006

Lecture1

gac1

6

Data Flow Graphs and Compilation

- Splitting into basic operations is necessary for both hardware and software implementations
- For software, such a procedure is performed by the compiler. Each of the steps can be performed by an assembly instruction.
- Assuming 1 instruction per clock cycle, our code would take 11 cycles to execute. The data flow graph shows us how we can speed this up by taking advantage of *parallelism*
- The “value” of each edge into a node in the graph must be known before the computation described by the node can be performed

1/12/2006

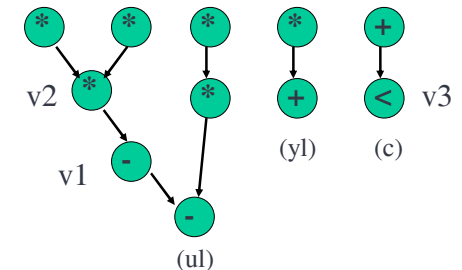
Lecture1

gac1

7

Data Flow Graphs and Parallelism

- Operation v1 needs to know the result of operation v2 before it can proceed
- Operation v3 doesn't depend on v2 at all so we could do it at the same time if we had enough hardware to spare



1/12/2006

Lecture1

gac1

8

Data Flow Graphs

- Formally, a dataflow graph is a directed graph $G(V,E)$ whose vertex set is in one-to-one correspondence with the set of tasks, and whose edge set is in correspondence with the transfer of data from one task to another.
- Data flow graphs express the maximum parallelism available
- They do not allow us to express loops or conditionals

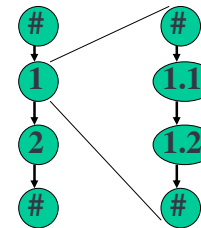
1/12/2006

Lecture1 gac1

9

Control Data Flow Graphs

- Structures which can also represent conditionals and loops are known as control data flow graphs (CDFGs)
- CDFGs are hierarchical graphs where each level of hierarchy is an acyclic DFG together with two additional nodes representing the start and end of the task



e.g.

- 1) have a beer
- 2) have a curry

- 1.1) go to pub
- 1.2) order and drink

1/12/2006

Lecture1 gac1

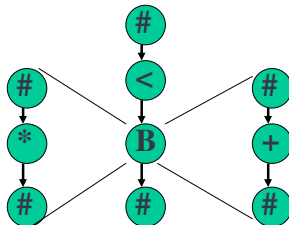
10

Conditionals

- Conditionals can be represented by introducing a task "B" (for branch) with two alternative expansions in the lower level of hierarchy

```

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



1/12/2006

Lecture1 gac1

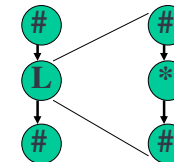
11

Loops

- Loops can be represented by introducing a task "L" (for loop)
- The "L" task tests the loop condition at each iteration, and does the necessary updates

```

for i = 1 to 3
  a = a * b;
    
```



1/12/2006

Lecture1 gac1

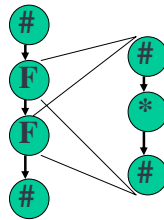
12

Function Calls

- Function calls can be represented by introducing a task “F” (for function)
- The F-task calls the function body represented by the lower level in the hierarchy

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

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



1/12/2006

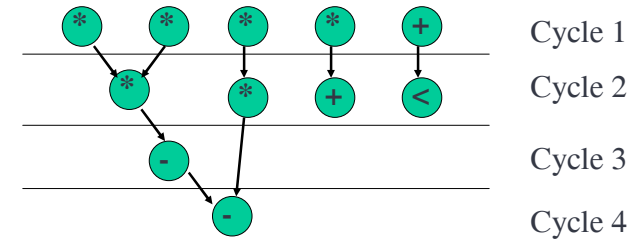
Lecture1

gac1

13

A Simple Schedule

- What if we had unlimited hardware? How fast could we make our algorithm go?
- Let's assume that each operation takes one clock cycle



We need at least four clock cycles. This schedule requires at least 4 multipliers, 1 adder/subtractor, and one comparator

1/12/2006

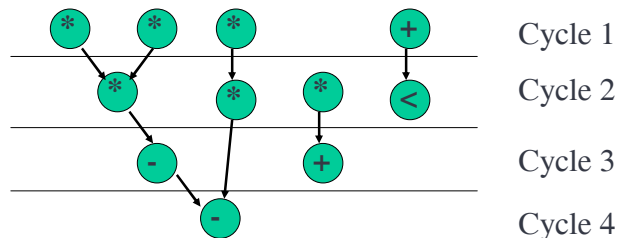
Lecture1

gac1

14

Improving our schedule

- By shuffling around the execution of tasks, we can reduce the number of resources required



This schedule requires at least 3 multipliers, 1 adder/subtractor, and one comparator

1/12/2006

Lecture1

gac1

15

Schedule: Definition

- With each node v in the graph $G(V,E)$, let us associate an execution time $d(v)$
- A schedule of this graph is a function $S:V \rightarrow \mathbb{N}$ where for all edges $(v_1, v_2) \in E$,
 $S(v_2) \geq S(v_1) + d(v_1)$
- For each node v , $S(v)$ denotes the start time of the relevant task

1/12/2006

Lecture1

gac1

16

Resource Constrained Schedules

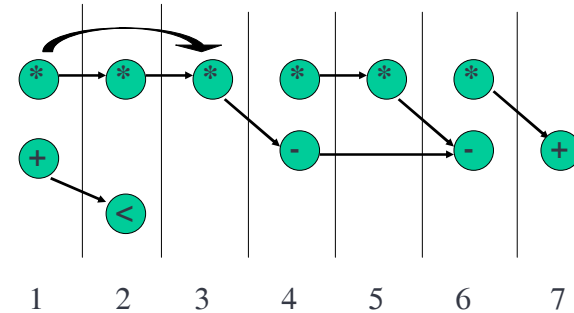
- Often our scheduling task consists of finding a schedule that will complete in a short time, subject to constraints on the amount of hardware available
- This is called “Resource Constrained Scheduling”
- Example: Schedule the differential equation code such that we need no more than:
 - one multiplier, one adder/subtractor, one comparator

1/12/2006

Lecture1 gac1

17

Resource Constrained Schedules



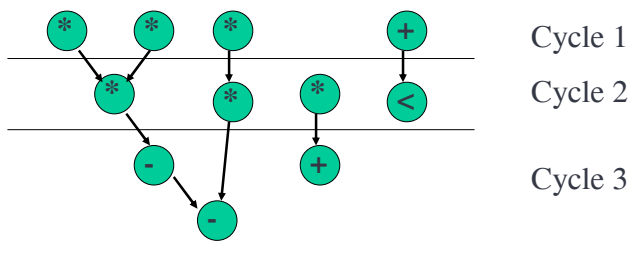
1/12/2006

Lecture1 gac1

18

Scheduling with Chaining

- So far, our basic unit of time has been the clock cycle.
- What if a subtraction only takes 0.5 clock cycles? We can do two subtractions in a single clock cycle.



1/12/2006

Lecture1 gac1

19

Scheduling with Chaining

- The advantages of chaining
 - We can reduce the latency of our schedule
 - The latency is the total number of clock cycles req'd
 - We can avoid registers
 - Each data transfer across a clock cycle needs a register to store temporary data
- The disadvantage of chaining
 - We have to work with sub-clock cycle units
 - Design of scheduling algorithms becomes more complex

1/12/2006

Lecture1 gac1

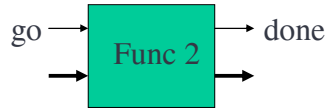
20

Synchronization

- Multiplications and additions have predictable delays
 - We can incorporate them into the scheduling model
- Some blocks do not, e.g. data-dependent iterations (while (a) { ... })



A hardware block with predictable delay



A hardware block with unpredictable delay

Synchronization

- Tasks with unpredictable delay can be classified as

- Bounded-delay tasks

```
for( i=0; i<50; i++ ) {
    ...
    if( something ) continue;
}
```

A simple solution: We could just assume that this task takes its worst-case time

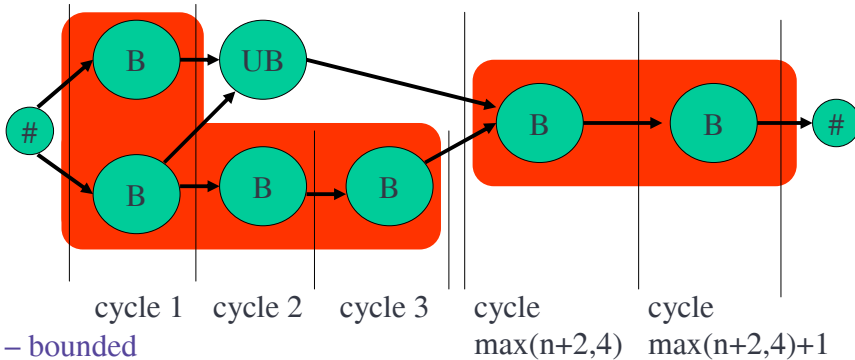
- Unbounded-delay tasks

```
while(x) {
    ...
    x = <something complicated>
}
```

There is no (easily calculable) worst-case time

Synchronization

- Methods are req'd to deal with unbounded tasks (they could also be applied to bounded tasks)



B – bounded

UB – unbounded, actual delay = n

Summary

- This lecture has covered
 - The relationship between code and operations
 - Data flow and control data flow graphs
 - Modelling of conditionals and loops
 - Resource constrained scheduling
 - Scheduling with chaining
 - Synchronization
- Later in the course, we will be exploring algorithms to do scheduling automatically

Introduction: Binding

- Part of a 4-lecture introduction
 - Scheduling
 - Resource binding
 - Area and performance estimation
 - Control unit synthesis
- This lecture covers
 - Resources and resource types
 - Resource sharing and binding
 - Graph models of resource binding
 - Conflict graphs
 - Templates for architectural synthesis
 - A complete worked problem

1/12/2006

Lecture2

gac1

1

Resources

- We refer to a piece of hardware that can perform a specific function as a “resource”
 - e.g. a 16x16-bit multiplier, a PCI interface
- An operation could be performed on one of several resources
 - e.g. a multiplication could be performed on one of two physically distinct multipliers
 - e.g. an addition could be performed by a special-purpose adder, or an ALU.
- We are distinguishing here between the operation, and the resource that will execute that operation

1/12/2006

Lecture2

gac1

2

Resource Types

- The “type” of a resource denotes its ability to perform different operations
 - A multiplier can do multiplications
 - An adder can do additions
 - An ALU can do comparisons and additions
- The resource type set R consists of all the different resource types we have available
 - $R = \{\text{multiplier, adder, ALU}\}$

1/12/2006

Lecture2

gac1

3

Resource Sharing

- Just because we have n additions in an algorithm, we don't need n adders
 - In traditional sequential processors, we use just a single adder to do all the additions in our program
 - This is possible because we have scheduled them – an adder is only used for one addition at one time
- Using the same resource to perform several different operations is “resource sharing”
- Advantage: can save area and peak power.
- Disadvantage: can make things slower and use more energy.

1/12/2006

Lecture2

gac1

4

Resource Sharing

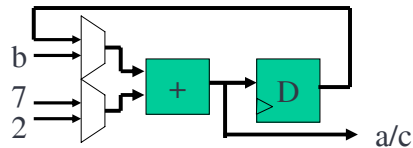
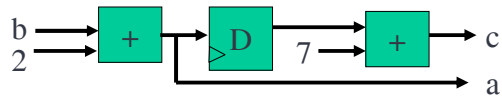
- Consider the code below and its scheduled DFG

```

a = b + 2;
c = a + 7;
    
```



- We could use two adders or one shared adder



One fewer adder but 2 more MUXs, possibly worse max clock rate

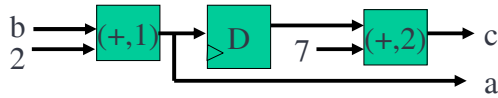
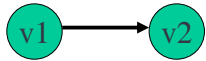
Need to generate select signals

Resource Binding

- Resource binding is the process of deciding which resource should perform which operation
- Cartesian product
 - \times denotes the Cartesian product of two sets
 - $A \times B = \{ (a,b) \mid a \in A, b \in B \}$
 - e.g. $\{a,b\} \times \{1,2\} = \{(a,1),(a,2),(b,1),(b,2)\}$
- A resource binding is a function $Y: V \rightarrow R \times N$

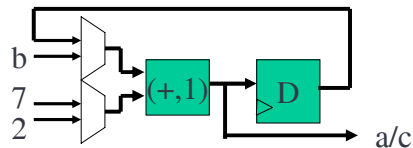
Resource Binding

- Revisiting our example...



$Y(v1) = (+,1)$

$Y(v2) = (+,2)$



$Y(v1) = (+,1)$

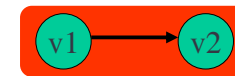
$Y(v2) = (+,1)$

Binding Graphs

- A hypergraph extends the notion of a graph by allowing edges to be incident to any number of nodes
- We can represent a bound CDFG or DFG by a hypergraph $G'(V, E \cup E_B)$



$E = \{ (v1,v2) \}$
 $E_B = \{ \{v1\}, \{v2\} \}$



$E = \{ (v1,v2) \}$
 $E_B = \{ \{v1,v2\} \}$

Conflict Graphs

- Sometimes we must bind operations to different resources
 - e.g. if they execute at the same time
- Such information can be represented using conflict graphs
- These have the same nodes as the corresponding DFG or CDFG.
- An edge corresponds to a conflict
 - two nodes connected by an edge cannot be bound to the same resource

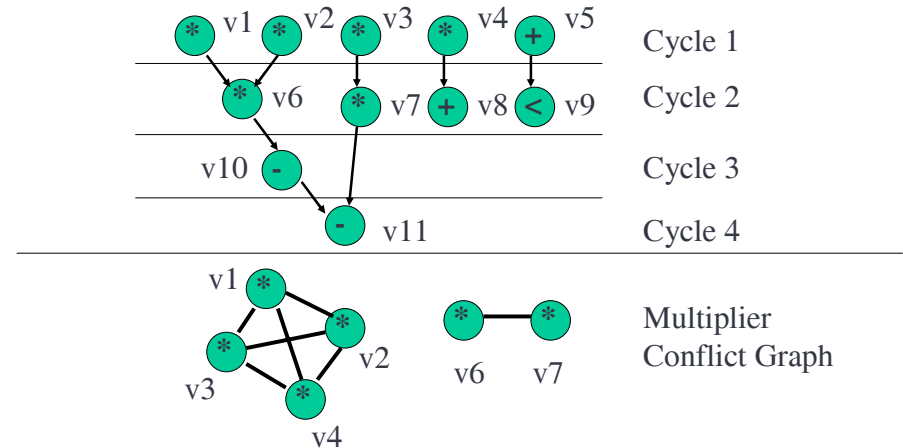
1/12/2006

Lecture2 gac1

9

Conflict Graphs

- Our example from Lecture 1



1/12/2006

Lecture2 gac1

10

Conflict Graphs

- In this example, the structure of the conflict graph is very simple
 - two disjoint sets of nodes, each one fully connected within itself
- This is because all operations took a single cycle – with multicycle operations, conflict graphs become more interesting and important (a later lecture...)

1/12/2006

Lecture2 gac1

11

Architectural Templates

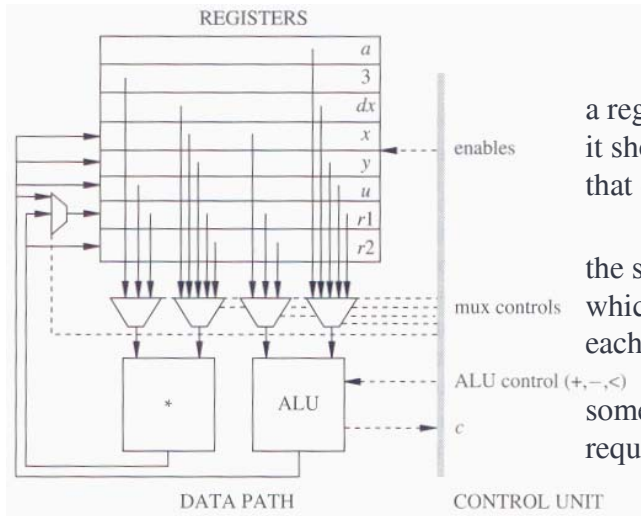
- Once we have a schedule S and a resource binding Y, we know all we need to construct our circuit
- In order to do this, the synthesis tool needs to have a “template” in mind
- We will be working with register bus-based architectures: in one clock cycle
 - values are read from registers, pass through multiplexers, and get steered to the right resource
 - the operations are performed
 - the results are written back into the registers

1/12/2006

Lecture2 gac1

12

Architectural Templates



a register is enabled when it should be written in that clock cycle

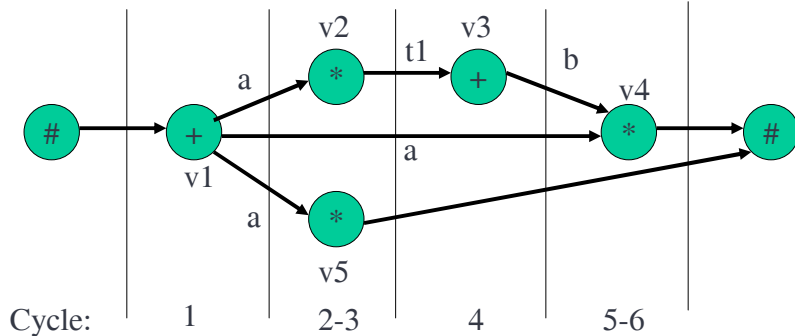
the select-lines decide which register to send to each resource

some resources may require additional control

Worked Problem

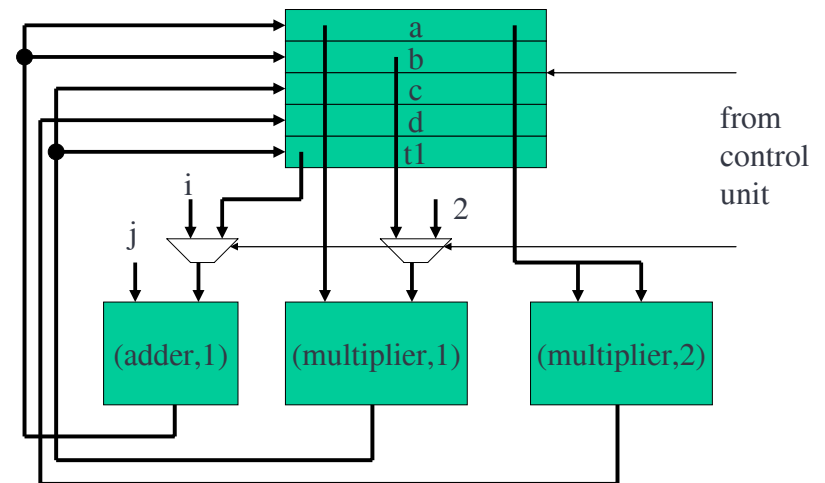
- Consider the following code:
 - $a = i+j$; $b = 2*a + j$; $c = a * b$; $d = a*a$;
 - (a) construct a CDFG for the code
 - (b) schedule the graph so that each operation starts as soon as it can, assuming each multiplication takes two cycles and each addition takes one cycle
 - (c) if you have the resource type set $R = \{\text{adder}, \text{multiplier}\}$, construct a resource binding for this example
 - (d) draw the completed data-path
 - (e) suggest a way you could save area

Worked Problem (a-c)



$Y(v1) = (\text{adder}, 1)$ $Y(v2) = (\text{multiplier}, 1)$
 $Y(v3) = (\text{adder}, 1)$ $Y(v4) = (\text{multiplier}, 1)$
 $Y(v5) = (\text{multiplier}, 2)$

Worked Problem (d)



Worked Problem (e)

- Area could be saved by scheduling v5 in cycles 4-5, and v4 in 6-7, at the penalty of one clock cycle
- (actually if we pipelined one of the multipliers, we wouldn't have to pay any penalty...)

Summary

- This lecture has covered
 - Resources and resource types
 - Resource sharing and binding
 - Graph models of resource binding
 - Conflict graphs
 - Templates for architectural synthesis
 - A complete worked problem
- Later in the course, we will be examining algorithms to perform automatic binding

Suggested Problems

- De Micheli, Problems 4.11, Q5 (assume all additions take one cycle) (**)
- For the binding hypergraph shown in De Micheli, Fig. 4.5, construct a datapath design (you may label your registers in any way) (*)

Introduction: Estimation

- Part of a 4-lecture introduction
 - Scheduling
 - Resource binding
 - Area and performance estimation
 - Control unit synthesis
- This lecture covers
 - Design space and the estimation problem
 - Resource domination
 - Estimation in general circuits
 - Rent's rule

Design space parameters

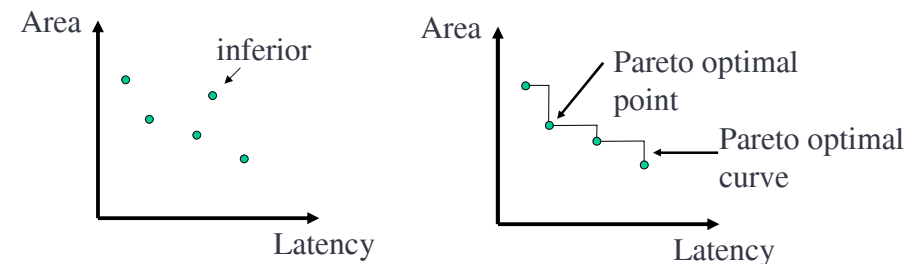
- There can be several objectives when performing a circuit design
 - small area
 - low latency
 - [high throughput] (for pipelined circuits)
 - high max clock rate
 - low power
- Often these objectives are conflicting, and we must trade-off between them

Design space parameters

- We can imagine a 4-dimensional space defined by (area, latency, max clock rate, power)
- Each design we could choose can be plotted as a point in this space
- Which design is “the best”?
 1. Area = 1, Latency = 2, MCR = 2, Power = 1
 2. Area = 2, Latency = 2, MCR = 1, Power = 1
 3. Area = 2, Latency = 1, MCR = 1, Power = 1
- Design 1 is better than design 2
- Design 1 could be better than design 3, depending on whether it's area, latency, or MCR we're most interested in.

Design space parameters

- Design 2 is known as an “inferior design”
 - it is dominated, in all objectives, by another design
- Graphically, we can imagine trading off area for latency (possibly by using more resources to reduce the number of clock cycles)



Why estimation?

- We are not just trying to create a working circuit, but one which meets some constraints on area, power, latency, etc.
- Each time we make a high-level design decision, we want to have an estimate of the effect of this decision on these objectives.
 - e.g. If I use 5 multipliers rather than 4, how will the power consumption change?
- We don't want to have to build the circuit and measure the power consumption – we need a *model*

1/12/2006

Lecture3 gac1

5

Resource domination

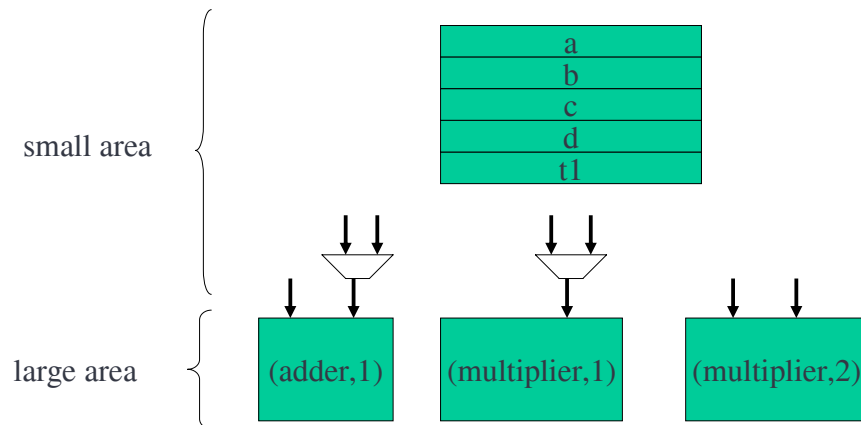
- For some “resource dominated” circuits, the area, speed, power, etc. are all a function of the resources
 - multiplexers, registers, etc. have an insignificant impact
- Estimation for these circuits is easy
 - Area: add up the area consumed by each resource: $A = A_{\text{add}} N_{\text{add}} + A_{\text{mult}} N_{\text{mult}} + \dots$
 - Latency: known from schedule
- Often DSP circuits tend to be resource dominated
- Example: the worked example from last lecture...

1/12/2006

Lecture3 gac1

6

Resource domination



$$\text{Total Area} = A_{\text{add}} + 2A_{\text{mult}}$$

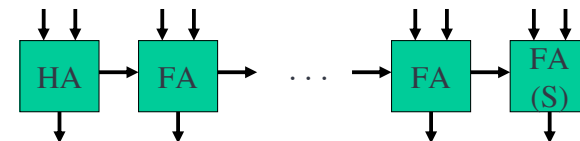
1/12/2006

Lecture3 gac1

7

Example: Adder Models

- How do we estimate the area and speed of each resource?
- Typically we have a model of how a resource is constructed, for example an n-bit ripple-carry adder:



$$\text{Area} = A_{\text{HA}} + A_{\text{FA(S)}} + (n-2)A_{\text{FA}} \quad (\text{would be different for a CLA})$$

$$\text{Delay} = T_{\text{HA}} + T_{\text{FA(S)}} + (n-2)T_{\text{FA(C)}}$$

1/12/2006

Lecture3 gac1

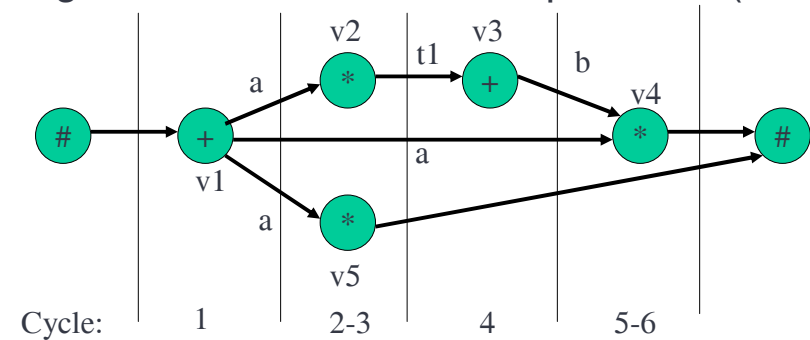
8

General circuits

- For non-resource dominated circuits, several other components can impact on the design objectives
 - registers
 - multiplexers
 - wiring
 - control unit

Registers

- Registers are required to store intermediate data – returning to our previous example, 3 registers are needed for temp. results (a,b,t1)

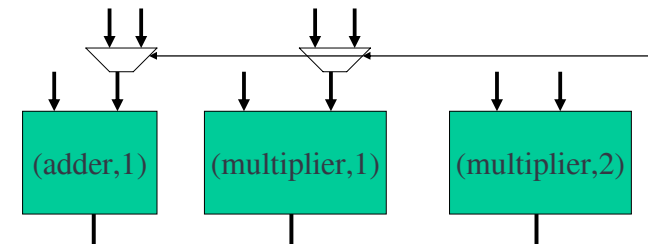


Registers

- We don't always need as many registers as there are temporary variables
- If registers are expensive, we could share registers, just like we share resources
 - t1 and b do not overlap in "lifetime" – we could use the same register for both
- The number of temporary variables provides a good "first guess" for the number of registers we'll need in our design

Multiplexers

- Multiplexers are needed to steer the right operands to the right resources

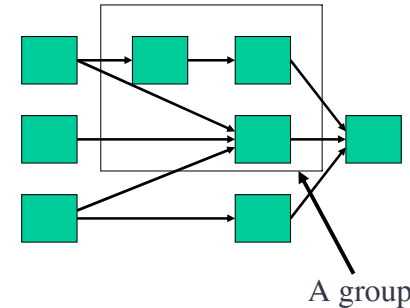


- The number and size of multiplexers required is known from the binding
- Multiplexers can consume area and add delay – the delay added can depend on the number of MUX inputs

Wiring

- Even wires add area, delay, and significant power consumption to a circuit
- It has been estimated that in the next few years it will take 10 clock cycles just for a signal to cross a chip
- Unfortunately wiring is hard to estimate
 - we need the binding but also the physical layout
- Rent's rule can provide a high-level model
 - Relates the amount of interconnect to the number of gates in an area
 - $N = KG^\beta$ (N = no. pins, G = no. gates)

Rent's Rule



Rent's rule gives a rough estimate how many wires will cross the boundaries of any given group.

This can be used as an estimate of wiring length

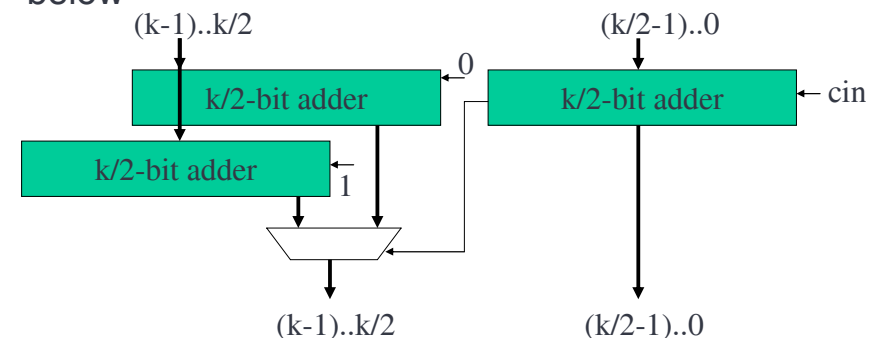
- Some examples of Rent constants
 - SRAM: $\beta = 0.12$, $K = 6$
 - μP : $\beta = 0.45$, $K = 0.82$
 - Gate Array: $\beta = 0.50$, $K = 1.9$

Control unit

- The control unit, which provides the select lines to MUXs and enable lines to registers, itself consumes area and power
- The size of the control unit can vary significantly depending on the amount of looping and branching in the algorithm
 - often DSP algorithms have very simple control
- The control unit can also impact on the max. clock rate
- We will investigate control unit synthesis in the next lecture
 - for now, let's simply state that the size of a controller tends to grow with
 - the number of activation signals (selects, enables)
 - the length of the schedule

Worked Example 1

- A k -bit carry-select adder has the structure shown below



- Assume:
 - each constituent adder is a ripple-carry design

Worked Example 1

- Derive models for the area and delay of the circuit, in terms of the wordlength k
- Compare the area and delay with a k -bit ripple carry design
- Area = $3A_{FA}(k/2) + A_{MUX}(k/2) = c_1k$
- Delay = $T_{FA}(k/2) + T_{MUX} = c_2k + c_3$
- Compared to a k -bit ripple carry design, the area is always larger. The delay is smaller, so long as
 - $T_{MUX} < T_{FA}(k/2)$ (almost certainly true for reasonable k)

Worked Example 2

- You are designing a circuit with gate-level Rent constants $\beta=0.25$, $k = 1$
- There are a total of 1M gates in your design. How many pins would you expect your chip to have?
 - Pins $\approx (1e6)^{0.25} \approx 32$
- Your chip is too big and you must split it over two chips. How many pins would you now expect?
 - Let's assume an equal split. Then:
 - Pins $\approx 2(5e5)^{0.25} \approx 53$
- The main source of power consumption in your design is driving the external pins. Estimate the increase in power due to using two chips.
 - Increase (%) = $(53 - 32)/32 = 66\%$ (assuming all pins equal)

Summary

- This lecture has covered
 - Design space and the estimation problem
 - Resource domination
 - Estimation in general circuits
 - Rent's rule
- Next lecture will examine the synthesis of control units

Introduction: Control Synthesis

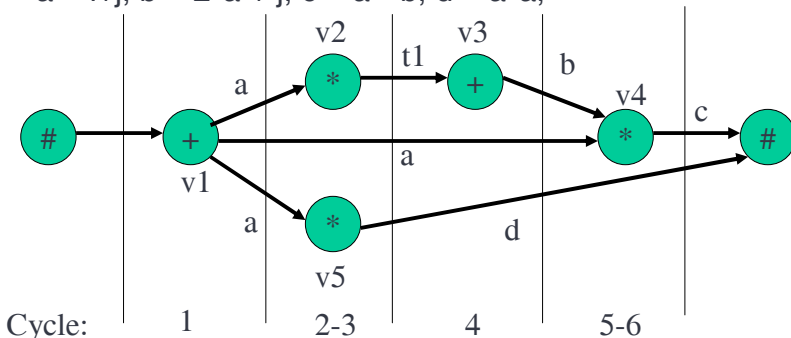
- Part of a 4-lecture introduction
 - Scheduling
 - Resource binding
 - Area and performance estimation
 - **Control unit synthesis**
- This lecture covers
 - Microcode and microcode optimization
 - Hardwired control
 - Control with interacting state machines

Why control synthesis?

- Our datapath designs have included
 - multiplexers, to steer data to the correct resource
 - register enable signals, to select when a register should store an intermediate value
- These signals have to be generated from somewhere – this is the *control unit*
- Once we know the schedule and binding for an algorithm, we have enough information to design the controller

A Control Unit

- Let's take another look at the worked example from Lect 2
 - $a = i+j$; $b = 2*a + j$; $c = a * b$; $d = a*a$;



Cycle: 1 2-3 4 5-6

$Y(v1) = (\text{adder}, 1)$ $Y(v2) = (\text{multiplier}, 1)$

$Y(v3) = (\text{adder}, 1)$ $Y(v4) = (\text{multiplier}, 1)$

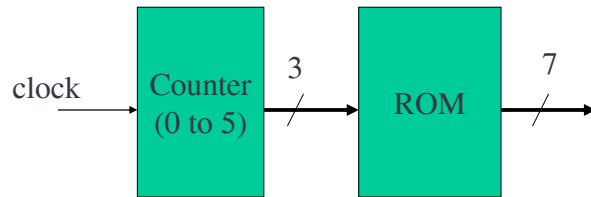
$Y(v5) = (\text{multiplier}, 2)$

A Control Unit

- The control unit for this example must:
 - Cycle 1: Enable register “a”, ensure the inputs of “adder1” are “i” and “j”
 - Cycle 2: Ensure the inputs of “multiplier1” are 2 and “a”
 - Cycle 3: Enable registers “t1” and “d”
 - Cycle 4: Enable register “b”, ensure the inputs of “adder1” are “t1” and “j”
 - Cycle 5: Ensure the inputs of “multiplier1” are “a” and “b”
 - Cycle 6: Enable register “c”

Microcode vs Hardwired control

- This is the behaviour of a sequential circuit
 - circuit has no inputs apart from clock for this example
 - circuit has 7 binary outputs (5 enables and 2 select lines
 - don't need a select line for dedicated resource “multiplier2”)
- We could build this circuit as a microcode-based controller with 3 address lines ($\lceil \log_2 \# \text{cycles} \rceil$)



1/12/2006

Lecture4 gac1

5

Microcode vs Hardwired control

- Alternatively we could build a finite state machine (FSM) implementation specifically for this sequence
- The choice between these two implementation schemes is a *logic* synthesis problem – we will not consider it in detail
- Hardwired FSM design is itself a major topic
 - could be faster, smaller, lower power than the corresponding microcode controller
 - more complex to design
 - less flexible (if you make the microcode ROM programmable)
 - more flexible (if your design has unbounded latency nodes)

1/12/2006

Lecture4 gac1

6

Horizontal Microcode

- For the example we've been working with, let's construct the ROM contents
 - assume the following ordering of data outputs (MSB to LSB): a enable, t1 enable, d enable, b enable, c enable, adder1 select, multiplier1 select
 - assume a 3-bit up counter, initialized to 0
- ROM micro-program

Address	Data	Address	Data
0x0	0x40	0x3	0x0A
0x1	0x00	0x4	0x01
0x2	0x30	0x5	0x04

1/12/2006

Lecture4 gac1

7

Horizontal Microcode

- This is known as “horizontal” microcode
 - # states \ll # control signals (usually)
 - ROM has much greater width than height
- We have great freedom with a horizontal microcode
 - we design a controller for *any* schedule and *any* binding in this way
 - design process is simple
- However, the ROM might be large

1/12/2006

Lecture4 gac1

8

Microcode Optimization

- By adding an extra (combinational) stage to our controller, we can often reduce the size of the ROM required



- The challenge is to design the ROM and decoder carefully to keep n small and optimize our speed, power, and area

Microcode Optimization

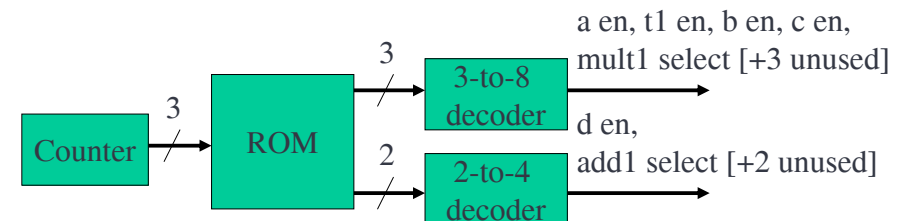
- If we didn't require any control signals concurrently, we would only require a $\lceil \log_2(\#control\ sigs+1) \rceil$ -bit ROM data bus
 - we could then use an n to 2^n decoder to generate the control signals
 - (revision... an n to 2^n decoder asserts one of 2^n different possible outputs depending on the n -bit binary encoding of the input. For example a 2-to-4 decoder has truth table 00->0001, 01->0010, 10->0100, 11->1000)
- Why (+1)?
 - because we may not want to assert *any* control signals in some clock cycles

Microcode Optimization

- But if we don't allow concurrent control signals then we don't allow parallelism!
 - the advantage of a hardware implementation is destroyed
- Solution:
 - Partition the set of control signals into subsets which are not required concurrently
- For our example, one possible partition is:
 - {a enable, t1 enable, b enable, c enable, multiplier1 select}, {d enable, adder1 select}
- We can encode each of these partitions separately

Microcode Optimization

- We need $\lceil \log_2 6 \rceil = 3$ bits to encode the first subset, and $\lceil \log_2 3 \rceil = 2$ bits to encode the second subset
- Our controller now looks like this



- We have saved two bits of ROM data bus!

Microcode Optimization

- Assume we order the ROM databus (MSB to LSB): first subset encoding, second subset encoding
- Assume we order the outputs of each decoder in the order shown on the figure in the prev. slide
- A suitable ROM program is now:

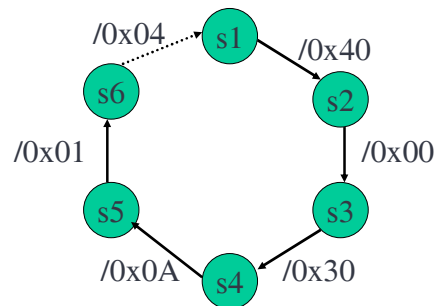
Address	Data	Address	Data
0x0	0b11100	0x3	0b10110
0x1	0b00000	0x4	0b01100
0x2	0b11011	0x5	0b10000

Hardwired FSM synthesis

- Alternatively, we could view controller design as a standard FSM synthesis problem
- One state per clock cycle
- Most importantly, it is easy to specify FSM behaviour for sequencing graphs with nodes of unbounded latency
- The same optimization technique applied to microcode can also be applied to the FSM design

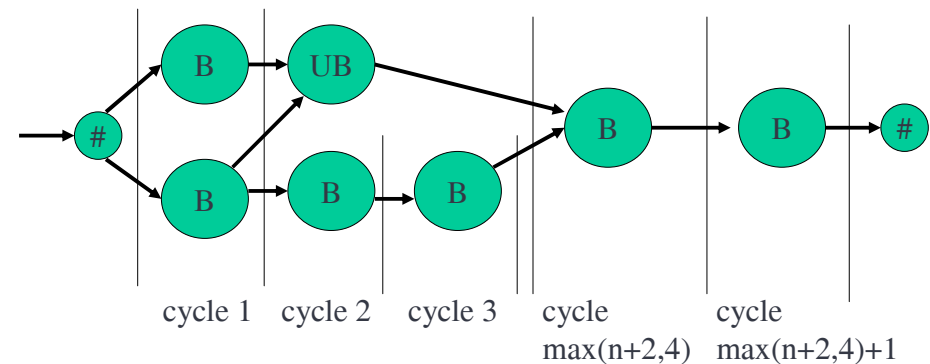
Bounded Latency Example

- Considering our previous example as an FSM leads to the following state transition graph, which can easily be coded in your favourite hardware description language



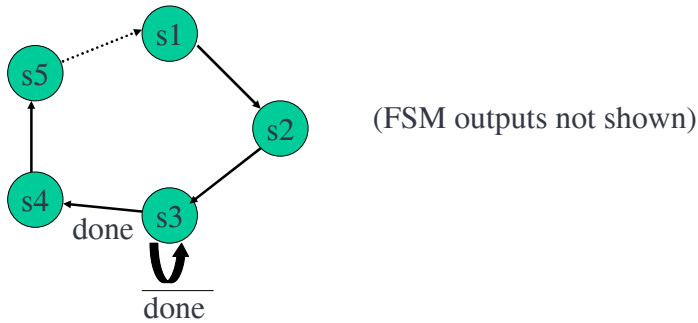
Unbounded Latency Operations

- Let's consider the unbounded latency example from Lecture 1



Unbounded Latency Example

- It is easy to create an FSM for this example
 - we must wait for “done” signal from unbounded latency resource => we have inputs to the controller as well as outputs



Local Controllers

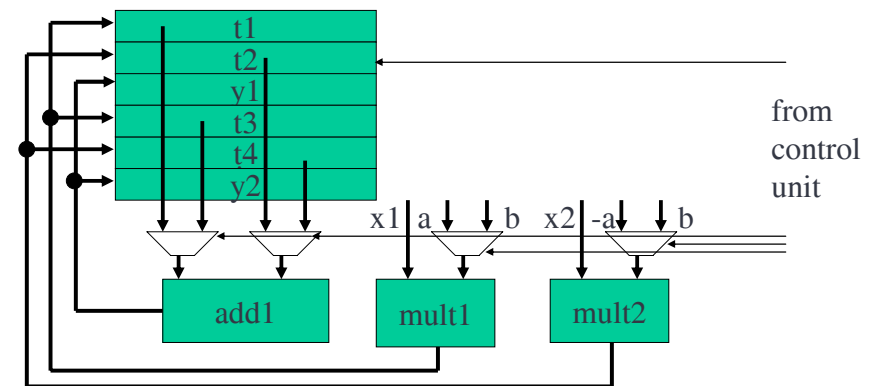
- So far we have assumed that there is one large controller for the whole circuit
 - this can be efficient as it allows the overhead of controller design and implementation to be shared by many control signals
 - however it may be impractical for large circuits due to the need to route control signals across the chip
- Each control signal (or set of control signals) could have its own controller
- Such a control unit is called “distributed control”. Using one large controller is called “centralized control” or “lumped control”

Worked Example

- You are designing a hardware implementation of a discrete cosine transform (DCT) algorithm. The inner loop of your algorithm is shown overleaf, along with a schedule and binding. (x_1, x_2 are inputs; y_1, y_2 are outputs). Both multipliers and adders take one clock cycle.
 - (a) draw a datapath for this circuit
 - (b) design a horizontal microcode-based controller for this circuit
 - (c) by making your design “non-horizontal”, minimize the size of ROM required
 - (d) re-design your controller so that each functional unit has its own controller, which controls the select-lines for its input multiplexers, and the enable lines for its output registers

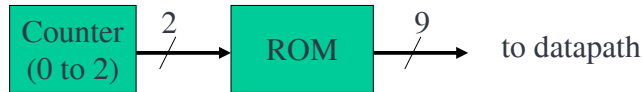
Worked Example (a)

$t_1 = a * x_1;$ // Cycle 1, mult1 $t_3 = b * x_1;$ // Cycle 2, mult1
 $t_2 = b * x_2;$ // Cycle 1, mult2 $t_4 = -a * x_2;$ // Cycle 2, mult2
 $y_1 = t_1 + t_2;$ // Cycle 2, add1 $y_2 = t_3 + t_4;$ // Cycle 3, add1



Worked Example (b)

- 3 cycles $\Rightarrow \lceil \log_2 3 \rceil = 2$ bit ROM address
- 6 registers + 3 mux select lines = 9 control signals



- assume data bus ordering (MSB to LSB): enables (t1,t2,y1,t3,t4,y2); select lines (add1, mult1, mult2)

Address	Data	Address	Data
0x0	0b110000000	0x2	0b000001100
0x1	0b001110011		

1/12/2006

Lecture4

gac1

21

Worked Example (c)

- In general, finding the subsets that minimize the size of ROM is a “hard” problem (more on the meaning of “hard” in a future lecture...)
- We will therefore find a “good”, but not necessarily optimum set of subsets
- Set 1: {t1, y1, y2} \Rightarrow need 2 bits
- Set 2: {t2, t3, add1} \Rightarrow need 2 bits
- Set 3: {t4} \Rightarrow need 1 bit
- Set 4: {mult1} \Rightarrow need 1 bit
- Set 5: {mult2} \Rightarrow need 1 bit

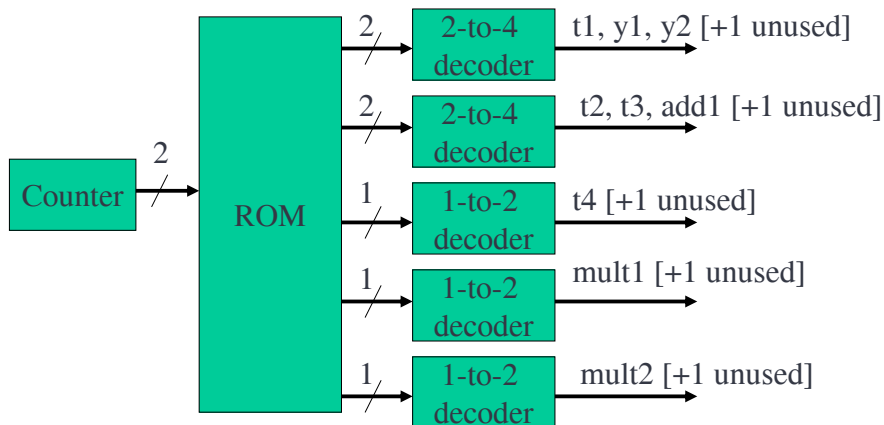
1/12/2006

Lecture4

gac1

22

Worked Example (c)



1/12/2006

Lecture4

gac1

23

Worked Example (c)

- Applying the techniques described in the lecture results in the table below (for the order in the figure on the prev. slide) This results in compression from 9 to 7 data bits.
- Clearly this could be further compressed, as from below bit6=bit4, bit5=bit3, bit2=bit1=bit0. Together this results in compression from 9 to 3 data bits.

Address	Data	Address	Data
0x0	0b1111000	0x2	0b0101000
0x1	0b1010111		

1/12/2006

Lecture4

gac1

24

Worked Example (d)

- For a horizontal implementation, simply pick the relevant output bits from the lumped-controller implementation
- Assume orderings (MSB to LSB):
 - add1 controller: add1, y1, y2
 - mult1 controller: mult1, t1, t3
 - mult2 controller: mult2, t2, t4

Address	add1 Data	mult1 Data	mult2 Data
0x0	0b000	0b010	0b010
0x1	0b010	0b101	0b101
0x2	0b101	0b000	0b000

1/12/2006

Lecture4

gac1

25

Summary

- This lecture has covered
 - Microcode and microcode optimization
 - Hardwired control
 - Control with interacting state machines
 - A detailed worked example
- During the next lecture we will start to look at some of the mathematical framework which helps us do architectural synthesis

1/12/2006

Lecture4

gac1

26

Suggested Problems

- Some multiplication algorithms have data-dependent delay. Assume we want to use one such multiplier, and one adder (with delay 1 cycle) to implement the differential equation inner-loop introduced in Lecture 1.
 - (a) perform a scheduling, treating the multipliers as elements of unknown latency
 - (b) sketch the datapath of the resulting design
 - (c) draw a FSM state-transition diagram for your design

1/12/2006

Lecture4

gac1

27

Graphs & Combinatorial Problems

- A new part of the course – will cover the more theoretical aspects required in later lectures
 - **Graphs, cliques, and colouring**
 - Algorithms and intractability
 - Linear programming and integer linear programming
 - Shortest and longest path algorithms
- This lecture covers
 - Definition of graph (revision), clique, and clique number
 - Graph colouring, chromatic number
 - Interval graphs

Graphs

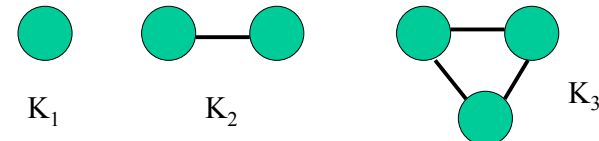
- Formal Definition:
 - A *graph* G is a finite nonempty set V together with an [irreflexive], symmetric relation E on V
- The relation E relates vertices to other vertices and is known as the edge relation, or “edge set”
- If relation E is symmetric, it means that
 - $(a,b) \in E \Rightarrow (b,a) \in E$
 - an edge has no concept of “direction”
- In mathematics, an edge relation is usually considered irreflexive:
 - $\neg \exists a : (a,a) \in E$
 - engineers often relax this constraint (hence the brackets)

Directed Graphs

- Formal definition:
 - A *directed graph* G is a finite nonempty set V together with an [irreflexive] relation E on V
 - This time the concept of direction is implicit, as we *could* have $(a,b) \in E$ and $(b,a) \notin E$
- You may see directed graphs referred to as “digraphs”

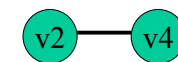
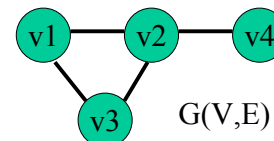
Cliques

- A *complete graph* is a special type of graph where all possible edges are in the edge set



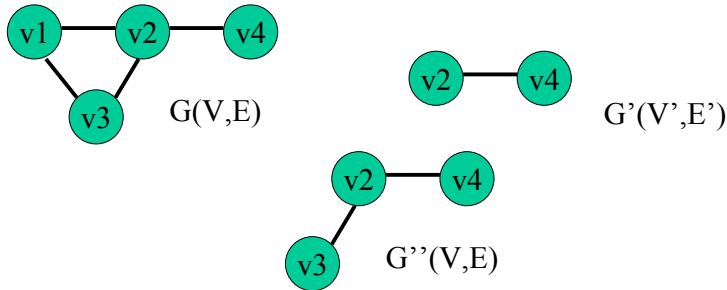
- A subgraph $G'(V',E')$ of a graph $G(V,E)$ is a graph whose vertex and edge sets obey

- $V' \subseteq V, E' \subseteq E$



Cliques

- A *clique* is a complete subgraph



- G' is a clique. G'' is not a clique (but it is a subgraph of G)

1/15/2007

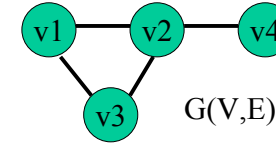
Lecture5

gac1

5

Clique Number

- The *clique number* $\omega(G)$ of a graph G is the size of the node set of its largest clique



- This graph has cliques with the following node subsets:
 - $\{v_1\}$, $\{v_2\}$, $\{v_3\}$, $\{v_4\}$, $\{v_1, v_2\}$, $\{v_1, v_3\}$, $\{v_2, v_3\}$, $\{v_2, v_4\}$, $\{v_1, v_2, v_3\}$
- Its clique number is 3

1/15/2007

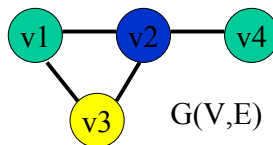
Lecture5

gac1

6

Graph Colouring

- Graph colouring is the process of labelling each node of a graph such that no two connected nodes share the same label



- The graph above is coloured with three different colours
- Graph colouring can model many problems
 - e.g. colouring a conflict graph (Lecture 2) will result in a resource binding

1/15/2007

Lecture5

gac1

7

A Colouring Algorithm

- A simple algorithm for colouring a graph is given below

```

Colour_Graph( G(V,E) )
begin
  foreach  $v \in V$  {
     $c = 1$ ;
    while  $\exists (v, v') \in E : v' \text{ has colour } c$ 
       $c = c + 1$ ;
    label  $v$  with colour  $c$  }
end
    
```

- This will always correctly colour a graph, but the number of distinct colours used depends on the order in which the nodes are visited

1/15/2007

Lecture5

gac1

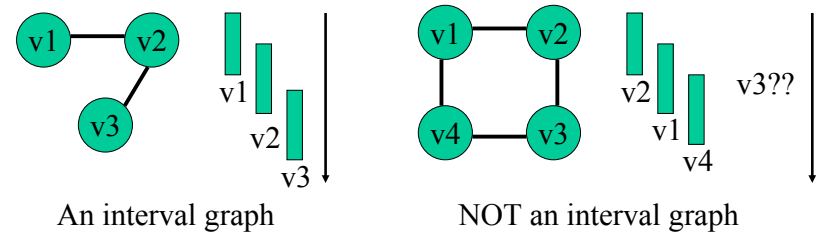
8

Chromatic Number

- The smallest number of colours with which it is possible to colour a graph G is called its chromatic number $\chi(G)$
- For a general graph, finding $\chi(G)$ is a “hard” problem
 - the algorithm presented does not guarantee a colouring with $\chi(G)$ colours
 - we’ll be discussing “hard” problems next lecture
- In resource binding, the chromatic number tells us the minimum number of distinct resources required
- Since every node in a clique must be coloured differently to every other node in a clique,
 - $\omega(G) \leq \chi(G)$

Interval Graphs

- Luckily, not all graphs are “hard” to colour. One type of graph which is easy to colour with the minimum number of colours is an “interval graph”
- An *interval graph* is a graph 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



An interval graph

NOT an interval graph

The Left Edge Algorithm

- The left-edge algorithm colours interval graphs optimally.
- Let us denote by l_i and r_i the left-most and right-most point of the interval corresponding to vertex v_i .

Left_Edge($G(V,E)$)

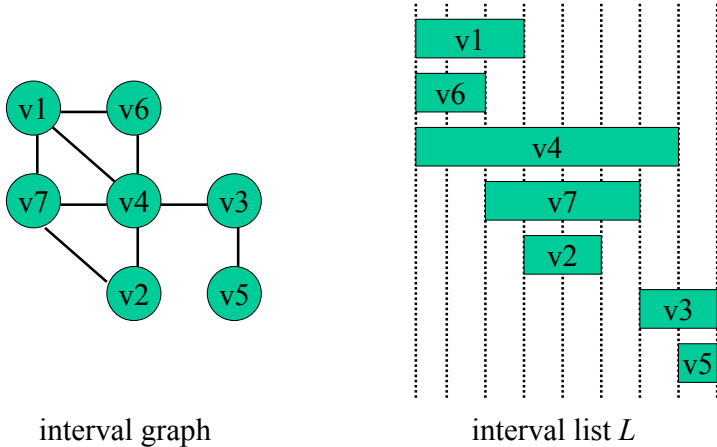
```

begin
  sort nodes in ascending order of left edge – store in L
  c := 1;
  while( not all vertices have been coloured ) {
    r := 0;
    while(  $\exists$  an element s 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
       $L := L \setminus \{v_s\}$ ;
    }
    c := c + 1;
  }
end
    
```

The Left Edge Algorithm

- Some set theory:
 - \setminus represents set subtraction
 - $X \setminus Y = \{ z : z \in X \wedge z \notin Y \}$
- The left edge algorithm tries to colour as many intervals as possible with one colour, before moving on to the next colour
- Left Edge was originally introduced to pack wire segments tightly on a VLSI layout. It is now used for many other purposes – particularly resource binding.

Left Edge - Example



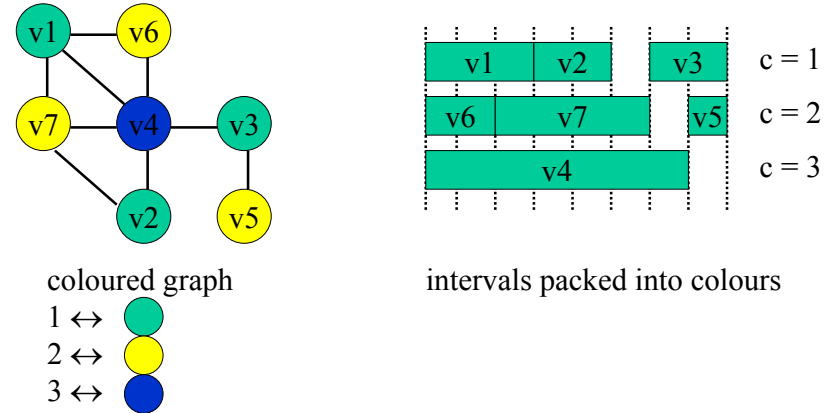
1/15/2007

Lecture5

gac1

13

Left Edge - Example



1/15/2007

Lecture5

gac1

14

Summary

- This lecture has covered
 - graphs and digraphs
 - cliques and clique number
 - colouring and chromatic number
 - interval graphs and the Left Edge algorithm
- Next lecture will examine the ideas behind designing “good” algorithms, and what it means for a problem to be “hard”

1/15/2007

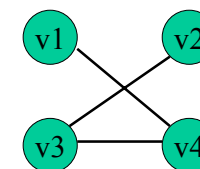
Lecture5

gac1

15

Suggested Problems

- For the graph below, apply the general colouring algorithm for the following two vertex orders. Compare and contrast your results. (*)
 - (a) $(v1, v2, v3, v4)$
 - (b) $(v1, v4, v3, v2)$
- By applying the left-edge algorithm, or otherwise, demonstrate that one of the two orders above results in an optimum colouring (*)



1/15/2007

Lecture5

gac1

16

Algorithms and Intractability

- Part of our 4-lecture “theory break”
 - Graphs, cliques, and colouring
 - Algorithms and intractability
 - Linear programming and integer linear programming
 - Shortest and longest path algorithms
- This lecture covers
 - The definition of an “algorithm”
 - Polynomial-time and intractability
 - P and NP
 - Polynomial reduction, NP-completeness and NP-hardness

1/15/2007

Lecture6

gac1

1

The Purpose of This Lecture

- Synthesis is all about writing algorithms to solve problems in digital design
- This lecture will consider some of the more theoretical aspects concerning
 - problems, algorithms, and complexity
- We will formalize what is meant by a “hard” problem
- You will **not** be required to prove the hardness of any unseen problems as part of this course
- You may be required to describe the ideas of hardness

1/15/2007

Lecture6

gac1

2

Problems and Instances

- We have already discussed several problems and algorithms. We will now take a few minutes to formalize these concepts
- A *problem* is a general question to be answered, usually possessing several parameters, whose values are left unspecified
 - e.g. Can I schedule a DFG $G(V,E)$ to complete within λ cycles using at most n multipliers?
- An *instance* of a problem is obtained by specifying particular values for all parameters
 - e.g. Can I schedule the DFG given in Lecture 1, slide 5, to complete within 10 cycles using at most 2 multipliers?

1/15/2007

Lecture6

gac1

3

“Hard” Problems



“I can't find an efficient algorithm, I guess I'm just too dumb.”

[Garey & Johnson 1979]

1/15/2007

Lecture6

gac1

4

“Hard” Problems



“I can’t find an efficient algorithm, because no such algorithm is possible!”

[Garey & Johnson 1979]

1/15/2007

Lecture6

gac1

5

“Hard” Problems



“I can’t find an efficient algorithm, but neither can all these famous people.”

[Garey & Johnson 1979]

1/15/2007

Lecture6

gac1

6

Algorithms and Efficiency

- An *algorithm* is a general step-by-step procedure for solving problems
- An algorithm is said to *solve* a problem Π if the algorithm can be applied to any instance of Π and is guaranteed to always produce a solution for that instance
- An *efficient* algorithm is one that solves the problem “quickly”
 - there are other factors such as memory usage, but we will ignore these

1/15/2007

Lecture6

gac1

7

Complexity

- Usually, we can describe the worst-case performance of an algorithm as a function of the “size” n of the problem instance
- We generally are concerned with the “big picture” of how performance scales with size (especially for large sizes), rather than specific execution times
- The Big-Oh notation allows us to express this behaviour
 - $O(n)$, $O(n^2)$, $O(e^n)$
- An algorithm is $O(f(n))$ if its worst case performance is bounded by $k f(n)$ for large n

1/15/2007

Lecture6

gac1

8

Complexity

- Example: A (good) algorithm to add n numbers will be $O(n)$
- Example: An algorithm to sort n numbers in order. You may be familiar with
 - quicksort: $O(n^2)$
 - heapsort: $O(n \log n)$
- Example: An algorithm which considers all possible k -colourings that a graph could have would be $O(k^n)$

Polynomial vs Exponential Time

- A polynomial-time algorithm is one which has $O(p(n))$ for some polynomial $p(\cdot)$.
- An exponential-time algorithm is any algorithm which is not polynomial-time.
- Clearly for large n , exponential-time algorithms take much longer than polynomial-time algorithms
 - the main distinction is thus: “is this algorithm exponential (bad) or polynomial (good)?”
 - the order of the polynomial is of secondary concern
- All problems which can be solved by polynomial algorithms are said to belong to the class P

Nondeterministic Polynomial Time

- To complicate matters, computer scientists have come up with another class, NP (nondeterministic polynomial).
- A problem is in NP if a *solution* to the problem can be *checked* in polynomial time
 - this doesn't mean it has to be solvable in polynomial time
- Example:
 - scheduling $G(V,E)$ in time λ given resource constraints may or may not be solvable in polynomial time
 - it is clear that given a schedule, we could check in polynomial time that it is a valid schedule and it completes within λ cycles

Want to Earn Some Money?

- The problem “does $P = NP?$ ” is unsolved
- If you solve it you will
 - be famous
 - win \$2,000,000 from www.claymath.org
- ...but don't let it distract you from your degree!

Polynomial Reduction

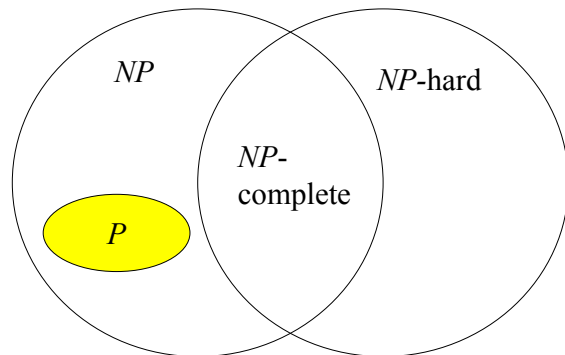
- Many interesting and difficult problems (like scheduling) are in NP but we don't know whether they're in P
- Since it is generally hard to prove that a given problem is not in P , we instead concentrate on proving that its "at least as hard" as a known hard problem
- If we can transform any instance of a hard problem Π^H into an instance of our problem Π , and that transformation can be done in polynomial time, then
 - if we can solve Π , we can solve $\Pi^H \Rightarrow \Pi$ is also hard!

NP-completeness & NP-hardness

- There are some problems which are in NP and which are known to be at least as hard as any other problem in NP .
 - these are called NP -complete
- NP -complete problems are of particular interest, as if a solution to any NP -complete problem can be found in polynomial time then $P = NP$
- A problem which is at least as hard as an NP -complete problem is called NP -hard
 - this is our formal definition: for "hard problem" read "NP-hard problem"

A Hierarchy of Problems

- Assuming $P \neq NP$, this is how our "world of problems" looks



Proving Hardness

- Proving NP -hardness requires two stages
 - pick a known NP -hard problem
 - demonstrate a transformation from this problem to your problem
- There are some NP -complete problems which form the basis of many proofs. We will look at one: Partition
- Partition: Given a finite set A and a measure $s(a) \in \mathbb{Z}^+$ for each $a \in A$, is there a subset $A' \subseteq A$ such that the following equation holds?

$$\sum_{a \in A'} s(a) = \sum_{a \in A - A'} s(a)$$

Proving Hardness

- An example instance of “partition”:
 - $A = \{v1, v2, v3\}$ with $s(v1) = 1$, $s(v2) = 2$, $s(v3) = 1$
 - for this instance, the answer is clearly “yes”:
 - $A' = \{v2\}$ or $A' = \{v1, v3\}$

Example: Scheduling is *NP*-hard

- To finish off, we’ll prove the *NP*-hardness of an example problem (a simple form of scheduling)
- Our simple scheduling problem has no data dependencies and only one type of operation
- Remember that you won’t be asked to do such a proof for an unseen problem, but this proof has been included
 - for completeness
 - to give a more “practical” end to a highly theoretical lecture
 - to justify past and future comments about scheduling being a “hard” task to perform

Scheduling is *NP*-hard

- Let’s start by defining our problem:
 - given a finite set A of operations, a latency $d(a) \in \mathbb{Z}^+$ for each $a \in A$, a number $m \in \mathbb{Z}^+$ of resources, and a deadline $\lambda \in \mathbb{Z}^+$
 - is there a schedule such that all operations complete within the deadline and no more than m resources are used?

Scheduling is *NP*-hard

- Let us rephrase the question:
 - is there a partition $A = A_1 \cup A_2 \cup \dots \cup A_m$ of A into m disjoint subsets such that

$$\max_{1 \leq i \leq m} \left\{ \sum_{a \in A_i} d(a) \right\} \leq \lambda$$

- A_i represents the set of operations assigned to processor i , and no two operations can be executed at the same time on a single resource

Scheduling is NP-hard

- Let's consider a special case of our problem, for $m=2$ and $\lambda = \frac{1}{2} \sum_{a \in A} d(a)$
- Then the problem reduces to:
 - given a finite set A , and a value $d(a) \in \mathbb{Z}^+$ for each $a \in A$
 - is there a partition into 2 disjoint subsets A' and $A - A'$ such that

$$\max \left\{ \sum_{a \in A'} d(a), \sum_{a \in A - A'} d(a) \right\} \leq \frac{1}{2} \sum_{a \in A} d(a)$$

Scheduling is NP-hard

- Rewriting, we require

$$\frac{1}{2} \max \left\{ \sum_{a \in A'} d(a) - \sum_{a \in A - A'} d(a), \sum_{a \in A - A'} d(a) - \sum_{a \in A'} d(a) \right\} \leq 0$$

- But for any k , $\max(k, -k) \leq 0 \Rightarrow k = 0$, so we require

$$\sum_{a \in A'} d(a) = \sum_{a \in A - A'} d(a)$$

- But this is the “partition” problem. So “partition” is a special case of our problem and hence our problem is NP-hard

Summary

- This lecture has covered
 - The definition of an “algorithm”
 - Polynomial-time and intractability
 - P and NP
 - Polynomial reduction, NP-completeness and NP-hardness
- Next lecture we will look at the (NP-hard!) problem of Integer Linear Programming (ILP) and how we can use ILP solving software to help us optimize our hardware

[Integer] Linear Programming

- Part of our 4-lecture “theory break”
 - Graphs, cliques, and colouring
 - Algorithms and intractability
 - Linear programming and integer linear programming
 - Shortest and longest path algorithms
- This lecture covers
 - Mathematical programming, integer / mixed-integer programming, and linear programming
 - Slack variables
 - Application example: Capital budgeting

1/15/2007

Lecture7

gac1

1

Mathematical Programming

- Mathematical “programming” is the name given to the branch of mathematics that considers the following optimization problem:

$$\max f(x), \quad x \in S \subseteq R^n$$

- Here R^n represents the set of n -dimensional vectors of real numbers, and f is a real-valued function defined on S . S is the *constraint set* and f is the *objective function*.
- By choosing f and S appropriately, we can model a wide variety of real-life problems in this way.

1/15/2007

Lecture7

gac1

2

Feasibility and Optimality

- Any $x \in S$ is called a *feasible solution*
- If there is an $x^o \in S$ such that
$$f(x) \leq f(x^o) \text{ for all } x \in S$$
then x^o is called an *optimal solution*
- The aim is to find an optimal solution for a given f and S

1/15/2007

Lecture7

gac1

3

Integer Programming

- An integer programming problem is one where S is restricted to have only integer values

$$S \subseteq Z^n \subseteq R^n$$

- A mixed integer programming problem is one where some elements of S are restricted to integers
- Integer programming problems are typically harder than the equivalent real problem. You can gain an intuition why by considering the following problems
 - find the value of x minimizing $\cos(x/5)$
 - 5π
 - find the integer value of x minimizing $\cos(x/5)$
 - $\text{round}(5\pi)$? $\text{round}(5\pi + 10\pi)$? ...

1/15/2007

Lecture7

gac1

4

Linear Programming

- Problems where f and S are restricted to linear form are of particular interest

$$f(x) = c^T x, \quad S = \{ x \mid Ax = b, x \geq 0 \}$$

- c is an $n \times 1$ vector, A is an $m \times n$ matrix and b is an $m \times 1$ vector
- Imposing the linearity constraints restricts the domain of problems, but allows us to use known solution techniques
- For general x , these problems can be solved exactly (e.g. Simplex technique). For integer x , the problem is *NP*-complete.

Why Are We Interested?

- We are interested in expressing problems as integer or mixed integer linear programs because
 - it provides a way to formalize the problem
 - we can apply known general techniques to solve the problem
 - lots of software exists to solve MILPs (e.g. `lp_solve`, available free from the web)
- I will be introducing ILP formulations for scheduling and resource binding in later lectures

Modelling Complex Problems

- At first glance, linear constraints may seem very restrictive – this is not necessarily the case, if you build your model carefully.
- Here are three types of constraint that could be useful in synthesis
 - inequalities (e.g. $x_1 + x_2 \leq b_1$, rather than $x_1 + x_2 = b_1$)
 - dichotomy (e.g. $x_1 + x_2 \leq b_1$ OR $x_3 + x_4 \leq b_2$)
 - conditionals (e.g. $x_1 + x_2 \leq b_1 \Rightarrow x_3 + x_4 \leq b_2$)
- We will only be considering the first in this brief introduction. If you wish to use the others,
 - R.S. Garfinkel and G.L. Nemhauser, “Integer Programming”, Wiley and Sons, 1972

Inequality

- Inequality constraints can easily be introduced by adding an extra variable
- For example, consider the program:
$$\max 2x_1 + 3x_2 \text{ subject to } x_1 + x_2 \leq 10$$
This is the same as
$$\max 2x_1 + 3x_2 \text{ subject to } x_1 + x_2 + x_3 = 10$$
- For “ \geq ”, we would insert $(-x_3)$ into the constraint
- The extra variable is called a slack variable – it does not appear in the objective function
- Because this is so straight-forward, many ILP solving programs allow you to express constraints with inequality directly. From now on, we will use inequalities freely without considering slack variables explicitly

Example: Capital Budgeting

- From Garfinkel and Nemhauser (1972):
 - A firm has n projects that it would like to undertake, but due to budget limitations, not all can be selected. In particular, project j has a value of c_j , and requires an investment of a_{ij} in the time period i , $i=1, \dots, m$. The capital available in time period i is b_i .
 - Problem: Maximize the total value, subject to budget constraints

Example: Capital Budgeting

- Let's introduce a set of variables x_j , which we interpret as:
 - $x_j = 1 \Rightarrow$ project j is selected
 - $x_j = 0 \Rightarrow$ project j is not selected
- Then the objective function can be formulated as

$$\sum_{j=1}^n c_j x_j$$

- The constraints are

$$\sum_{j=1}^n a_{ij} x_j \leq b_i, i = 1, \dots, m; \quad x_j \leq 1, j = 1, \dots, n$$

Summary

- This lecture has covered
 - Mathematical programming, integer / mixed-integer programming, and linear programming
 - Slack variables
 - Application example: Capital budgeting
- Next lecture (the last in our “theory break”), looks at finding the shortest and longest path through a graph

Path Problems and Algorithms

- Part of our 4-lecture “theory break”
 - Graphs, cliques, and colouring
 - Algorithms and intractability
 - Linear programming and integer linear programming
 - **Shortest and longest path algorithms**
- This lecture covers
 - Edge-weighted graphs, shortest and longest path problems
 - Longest path through a DAG
 - Longest path through a general graph: Liao-Wong
 - Longest path as a LP

1/15/2007

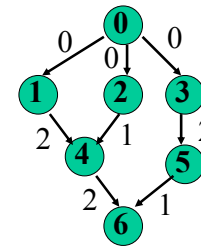
Lecture8

gac1

1

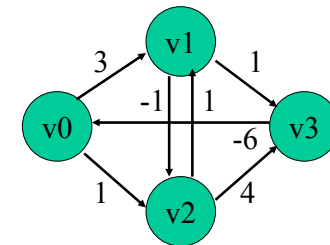
Edge Weighted Graphs

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



An edge weighted DAG

1/15/2007



An edge-weighted graph with cycles

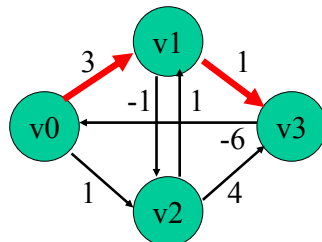
Lecture8

gac1

2

Shortest and Longest Path

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



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

1/15/2007

Lecture8

gac1

3

Shortest and Longest Path

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

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

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

1/15/2007

Lecture8

gac1

4

Longest Path Through a DAG

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

DAG Algorithm

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

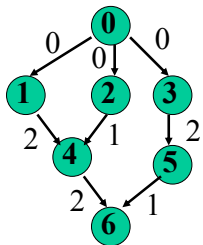
```

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

```

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

DAG example



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

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

DAG Example

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

General Longest Path

- Many algorithms to find the longest path of general graphs have been proposed in the literature
- We will consider Liao and Wong's algorithm as it is very efficient for cases where the graph edge set $E \cup F$ can be partitioned into a "forward" edge set E and a feedback edge set F where $G(V, E)$ is a DAG and $|E| \gg |F|$
 - this is often the case with graphs arising in synthesis – we will consider some of these in future lectures

1/15/2007

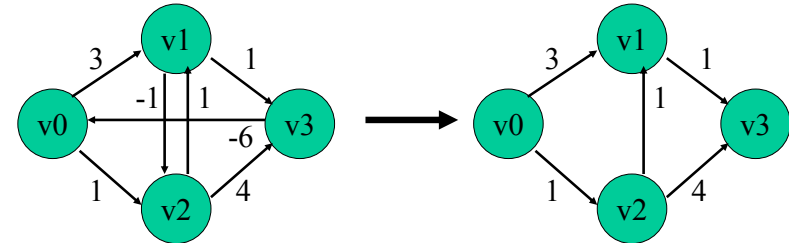
Lecture8

gac1

9

Example Edge Set Partition

- Consider our example graph. If we remove the edges labelled "-1" and "-6", we obtain a DAG



- The remaining edges form the set E , whereas the two we removed form the set F

1/15/2007

Lecture8

gac1

10

General Algorithm

```

Algorithm Liao_Wong(  $G(V, E \cup F)$ , source )
  for j = 1 to  $|F| + 1$  {
    DAG_Longest_Path(  $G(V, E)$ , source);
    flag = TRUE;
    foreach (u, v) in F {
      if  $s_v < s_u + w(u, v)$  {
        flag = FALSE;
         $E = E \cup \{ (source, v) \}$ ;
         $w(source, v) = s_u + w(u, v)$ ;
      }
    }
    if ( flag ) return;
  }
end Liao_Wong

```

1/15/2007

Lecture8

gac1

11

Algorithm Description

- Liao Wong first applies the DAG algorithm on the forward edges only. If no feedback edge provides a longer path alternative, the algorithm terminates
- If a longer path alternative is found, the algorithm models this as an extra forward edge directly from the source
- This process is repeated, until no more changes to the edge set are necessary
- It is provable that if the graph contains no cycles where the sum of weights around the cycle is positive, the outer loop need only be executed at most $|F|+1$ times.

1/15/2007

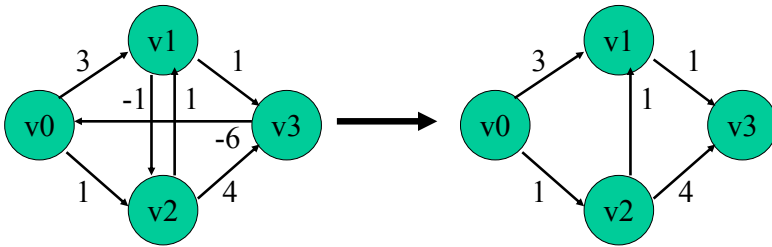
Lecture8

gac1

12

General Example

- Let us examine our example graph



- Performing our initial DAG longest path, with v0 as the source, leads to
 - $s_{v_0} = 0, s_{v_1} = 3, s_{v_2} = 1, s_{v_3} = 5$

1/15/2007

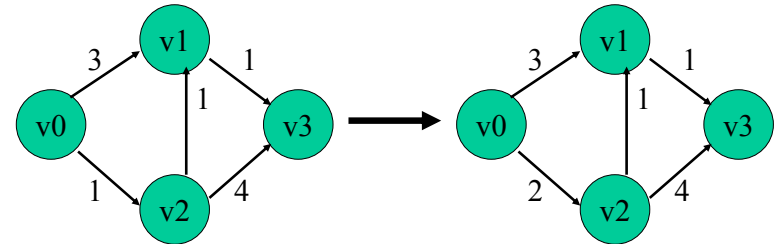
Lecture8

gac1

13

General Example

- We now examine each of the feedback edges in turn
 - for edge (v3,v0), $s_{v_0} \geq s_{v_3} - 6$ ($0 \geq -1$), so no change needs to be made
 - for edge (v1,v2), $s_{v_2} < s_{v_1} - 1$ ($1 < 2$), so we must insert a new forward edge (v0,v2) with weight 2 [in this example, (v0,v2) is already in E, so we just modify the weight]



1/15/2007

Lecture8

gac1

14

General Example

- Calculating the longest path on the modified DAG leads to
 - $s_{v_0} = 0, s_{v_1} = 3, s_{v_2} = 2, s_{v_3} = 6$
- Examining each feedback edge in turn
 - for edge (v3,v0), $s_{v_0} \geq s_{v_3} - 6$ ($0 \geq 0$), so no change needs to be made
 - for edge (v1,v2), $s_{v_2} \geq s_{v_1} - 1$ ($2 \geq 2$), so no change needs to be made
- At this point, the algorithm terminates as no changes are necessary

1/15/2007

Lecture8

gac1

15

Longest Path as a LP

- To keep up our interest in LP, let's formulate the longest path problem as a LP
- Let's revisit Bellman's equations:

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

- A necessary condition for satisfaction is:

$$\forall (u,v) \in E, \quad s_v \geq s_u + w(u,v) \quad (*)$$

- The minimum values of s_v that satisfy (*) are the solutions to Bellman's equations

1/15/2007

Lecture8

gac1

16

Longest Path as a LP

- We can write this as:

minimize $\sum_{v \in V} s_v$ subject to:

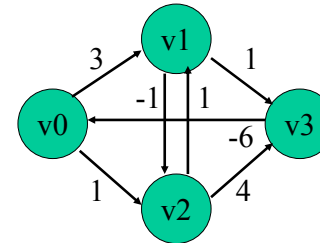
$$s_v \geq s_u + w(u, v) \text{ for all } (u, v) \in E$$

$$\text{and } s_{source} = 0$$

- This is a standard LP formulation (c.f. lecture 7), which can easily be cast in matrix notation $A\mathbf{x} \geq \mathbf{b}$ if required

LP Example

- For our general graph example, the LP objective function and constraints are given below



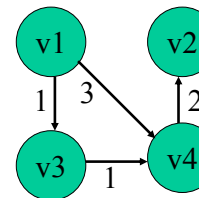
- minimize $s_0 + s_1 + s_2 + s_3$
subject to:
 $s_1 \geq s_0 + 3$; $s_1 \geq s_2 + 1$
 $s_2 \geq s_0 + 1$; $s_2 \geq s_1 - 1$
 $s_3 \geq s_1 + 1$; $s_3 \geq s_2 + 4$
 $s_0 \geq s_3 - 6$; $s_0 = 0$

Some Applications

- Longest and shortest path problems have many real-life applications, including
 - Circuits: Determining the critical path in a circuit, and hence the performance of that circuit
 - Transport: Finding the (shortest/cheapest/least fuel) route between two places
 - Networking and Comms: Shortest path through a network

Worked Example

- Consider the edge-weighted graph shown below



- (a) determine the longest path from v_1 to all other vertices in the graph
- (b) if an edge (v_2, v_3) with weight $w(v_2, v_3) = -4$ were added, how would this affect the longest paths?

Worked Example

- (a) It should be easy to see that $s_{v_1} = 0$, $s_{v_2} = 5$, $s_{v_3} = 1$, $s_{v_4} = 3$ (verify by applying Bellman's equations in the order (v_1, v_3, v_4, v_2))
- (b) This edge would close a cycle $\{v_3, v_4, v_2\}$. We therefore use Liao-Wong to determine whether any change has occurred to the longest paths. Examining the feedback edge (v_2, v_3) , we see that $s_{v_3} \geq s_{v_2} - 4$ ($1 \geq 5 - 4$) and therefore the extra edge has not affected the longest paths.

Summary

- This lecture has covered
 - Edge-weighted graphs, shortest and longest path problems
 - Longest path through a DAG
 - Longest path through a general graph: Liao-Wong
 - Longest path as a LP
- This brings us to the end of our “theory break”. Next lecture will look at scheduling digital circuits.

Suggested Problems

- Find the *shortest* path through the DAG used as an example in this lecture (*)
- Try to apply the Liao-Wong algorithm to find the *shortest* path through the cyclic graph example. Does it work? If not, why not? (***)
- In the cyclic example, change the weight of edge (v_3, v_0) to -4 . Now apply Liao-Wong to the *shortest* path problem. (*)

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

1/22/2007

Lecture9

gac1

1

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”

1/22/2007

Lecture9

gac1

2

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

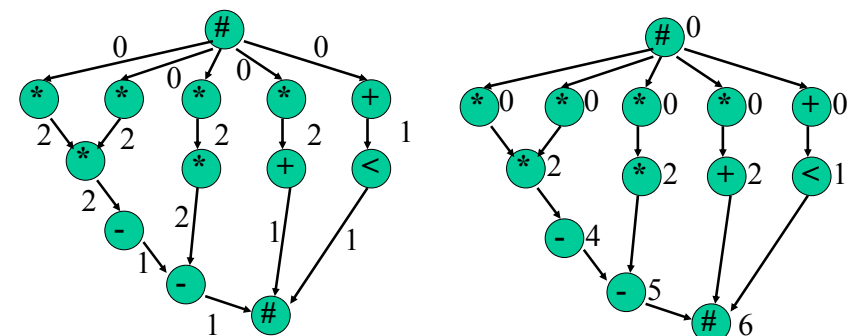
1/22/2007

Lecture9

gac1

3

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

1/22/2007

Lecture9

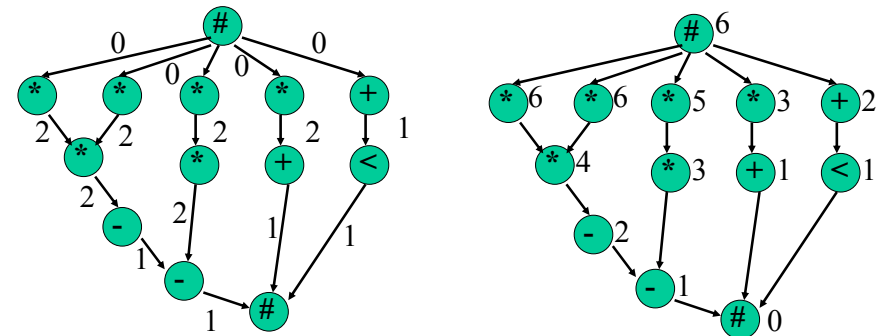
gac1

4

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

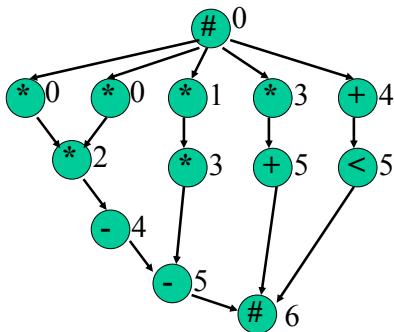


Edge-weighted CDFG

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

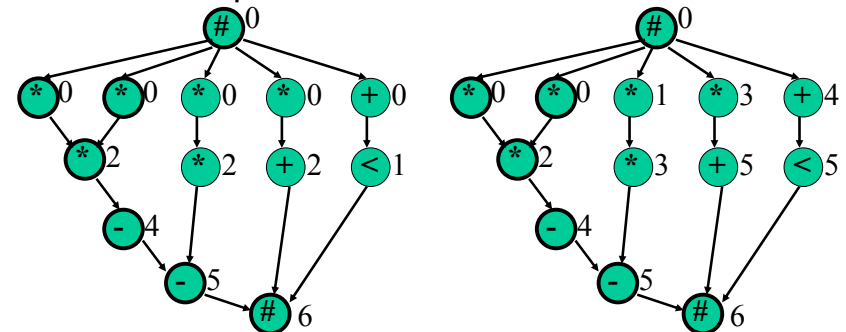
ALAP Scheduling



- 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

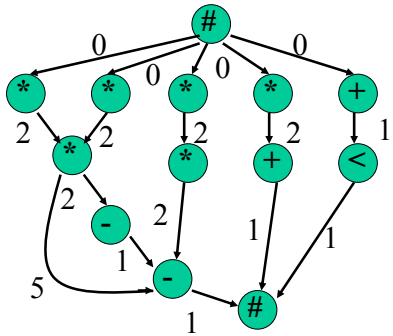
- We will consider two types of constraint
 - a minimum timing constraint l_{ij} between operations v_i and v_j : $S(v_j) \geq S(v_i) + l_{ij}$
 - a maximum timing constraint u_{ij} between operations v_i and v_j : $S(v_j) \leq S(v_i) + u_{ij}$
- These constraints are sufficient to model the situations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

1/22/2007

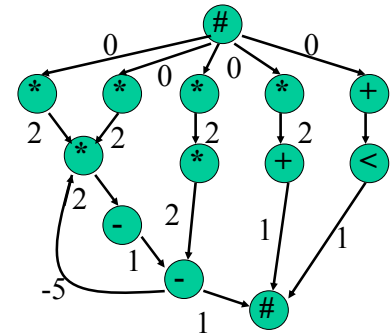
Lecture9

gac1

13

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
- $S(\text{mult}) \geq S(\text{sub}) - 5$, i.e. $S(\text{sub}) \leq S(\text{mult}) + 5$
- The subtraction cannot occur later than five cycles after the multiplication starts

1/22/2007

Lecture9

gac1

14

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

1/22/2007

Lecture9

gac1

15

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

1/22/2007

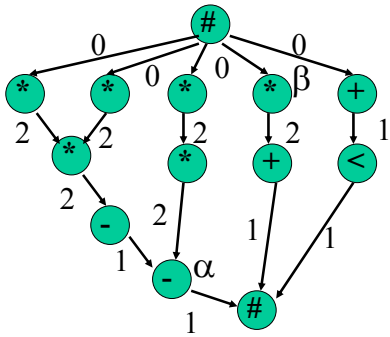
Lecture9

gac1

16

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

1/22/2007

Lecture10

gac1

1

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

1/22/2007

Lecture10

gac1

2

Resource Constrained Scheduling

- Let R denote the set of resource types,
 - e.g. $R = \{\text{add, mult, ALU}\}$
- 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

1/22/2007

Lecture10

gac1

3

Resource Constrained Algorithm

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

1/22/2007

Lecture10

gac1

4

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

1/22/2007

Lecture10

gac1

5

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

1/22/2007

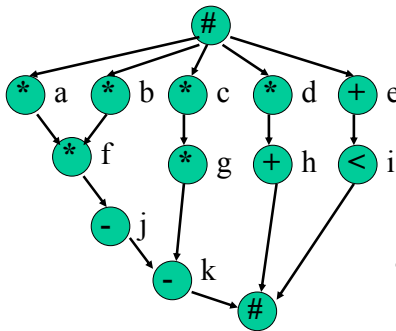
Lecture10

gac1

6

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 $a_*=2$, $a_{+/-}=2$, $a_<=1$



1/22/2007

Lecture10

gac1

7

Resource Constrained Example

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

1/22/2007

Lecture10

gac1

8

Resource Constrained Example

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

1/22/2007

Lecture10

gac1

9

Resource Constrained Example

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

1/22/2007

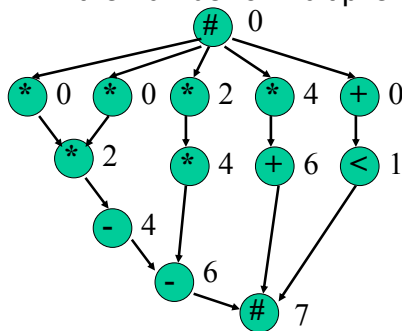
Lecture10

gac1

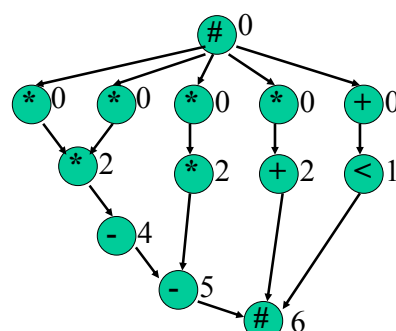
10

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

1/22/2007

Lecture10

gac1

11

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

1/22/2007

Lecture10

gac1

12

Latency Constrained Algorithm

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

1/22/2007

Lecture10

gac1

13

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

1/22/2007

Lecture10

gac1

14

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

1/22/2007

Lecture10

gac1

15

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

1/22/2007

Lecture10

gac1

16

Latency Constrained Example

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

1/22/2007

Lecture10

gac1

17

Latency Constrained Example

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

1/22/2007

Lecture10

gac1

18

Latency Constrained Example

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

1/22/2007

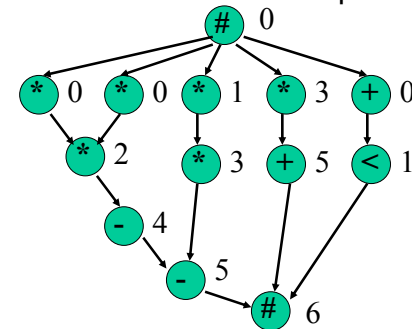
Lecture10

gac1

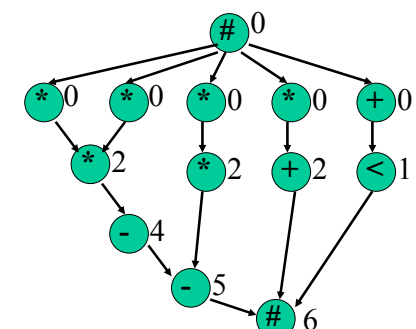
19

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

1/22/2007

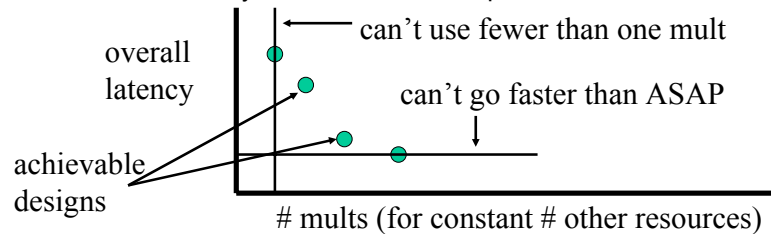
Lecture10

gac1

20

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



1/22/2007

Lecture10

gac1

21

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

1/22/2007

Lecture10

gac1

22

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-schedulingAre 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 (***)

1/22/2007

Lecture10

gac1

23

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

1/22/2007

Lecture11

gac1

1

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

1/22/2007

Lecture11

gac1

2

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

1/22/2007

Lecture11

gac1

3

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

1/22/2007

Lecture11

gac1

4

Binary Decision Variables

- We will use a trick often used in ILP formulations: to introduce binary decision variables
- We will use x_{vt} ($v \in V, t \in \{\text{ASAP}_v, \text{ASAP}_v+1, \dots, \text{ALAP}_v\}$), 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

$$\forall v \in V : \sum_{t=\text{ASAP}_v}^{\text{ALAP}_v} x_{vt} = 1$$

- 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

$$\forall (v', v) \in E : \sum_{t=\text{ASAP}_v}^{\text{ALAP}_v} t \cdot x_{vt} \geq \sum_{t=\text{ASAP}_{v'}}^{\text{ALAP}_{v'}} t \cdot x_{v't} + d_{v'}$$

- 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

$$\forall r \in R, \forall t \in \{0, \dots, \lambda\},$$

$$\sum_{v \in V : T(v)=r} \sum_{t' \in \{t-d_v+1, \dots, t\} \cap \{\text{ASAP}_v, \dots, \text{ALAP}_v\}} x_{vt'} \leq a_r$$

- 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

1/22/2007

Lecture11

gac1

9

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)

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

1/22/2007

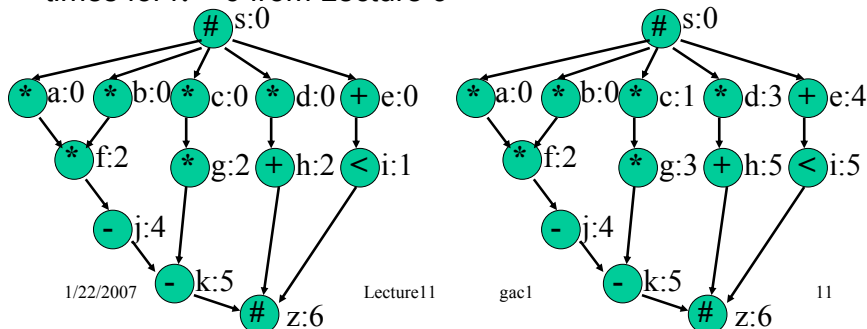
Lecture11

gac1

10

Example ILP

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



1/22/2007

Lecture11

gac1

11

Example ILP

- First, let's 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

1/22/2007

Lecture11

gac1

12

Example ILP

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

$$x_{s0} = 1$$

$$x_{a0} = 1$$

$$x_{b0} = 1$$

$$x_{c0} + x_{c1} = 1$$

$$x_{d0} + x_{d1} + x_{d2} + x_{d3} = 1$$

$$x_{e0} + x_{e1} + x_{e2} + x_{e3} + x_{e4} = 1$$

$$x_{f2} = 1$$

$$x_{g2} + x_{g3} = 1$$

$$x_{h2} + x_{h3} + x_{h4} + x_{h5} = 1$$

$$x_{i1} + x_{i2} + x_{i3} + x_{i4} + x_{i5} = 1$$

$$x_{j4} = 1$$

$$x_{k5} = 1$$

$$x_{z6} = 1$$

1/22/2007

Lecture11

gac1

13

Example ILP

- The next constraints are dependency constraints:

$$0 \cdot x_{a0} \geq 0 \cdot x_{s0} + 0$$

$$0 \cdot x_{b0} \geq 0 \cdot x_{s0} + 0$$

$$0 \cdot x_{c0} + 1 \cdot x_{c1} \geq 0 \cdot x_{s0} + 0$$

$$0 \cdot x_{d0} + 1 \cdot x_{d1} + 2 \cdot x_{d2} + 3 \cdot x_{d3} \geq 0 \cdot x_{s0} + 0$$

$$0 \cdot x_{d0} + 1 \cdot x_{d1} + 2 \cdot x_{d2} + 3 \cdot x_{d3} + 4 \cdot x_{d4} \geq 0 \cdot x_{s0} + 0$$

$$2 \cdot x_{f2} \geq 0 \cdot x_{a0} + 2$$

$$2 \cdot x_{f2} \geq 0 \cdot x_{b0} + 2$$

$$2 \cdot x_{g2} + 3 \cdot x_{g3} \geq 0 \cdot x_{c0} + 1 \cdot x_{c1} + 2$$

$$2 \cdot x_{h2} + 3 \cdot x_{h3} + 4 \cdot x_{h4} + 5 \cdot x_{h5} \geq 0 \cdot x_{d0} + 1 \cdot x_{d1} + 2 \cdot x_{d2} + 3 \cdot x_{d3} + 2$$

$$1 \cdot x_{i1} + 2 \cdot x_{i2} + 3 \cdot x_{i3} + 4 \cdot x_{i4} + 5 \cdot x_{i5} \geq 0 \cdot x_{e0} + 1 \cdot x_{e1} + 2 \cdot x_{e2} + 3 \cdot x_{e3} + 4 \cdot x_{e4} + 1$$

1/22/2007

Lecture11

gac1

14

Example ILP

- Dependency constraints continued...

$$4 \cdot x_{j4} \geq 2 \cdot x_{f2} + 2$$

$$5 \cdot x_{k5} \geq 2 \cdot x_{g2} + 3 \cdot x_{g3} + 2$$

$$6 \cdot x_{z6} \geq 2 \cdot x_{h2} + 3 \cdot x_{h3} + 4 \cdot x_{h4} + 5 \cdot x_{h5} + 1$$

$$6 \cdot x_{z6} \geq 1 \cdot x_{i1} + 2 \cdot x_{i2} + 3 \cdot x_{i3} + 4 \cdot x_{i4} + 5 \cdot x_{i5} + 1$$

$$5 \cdot x_{k5} \geq 4 \cdot x_{j4} + 1$$

$$6 \cdot x_{z6} \geq 5 \cdot x_{k5} + 1$$

1/22/2007

Lecture11

gac1

15

Example ILP

- Resource constraints:

$$r = <, t = 1 : x_{i1} \leq a_{<}$$

$$r = <, t = 2 : x_{i2} \leq a_{<}$$

$$r = <, t = 3 : x_{i3} \leq a_{<}$$

$$r = <, t = 4 : x_{i4} \leq a_{<}$$

$$r = <, t = 5 : x_{i5} \leq a_{<}$$

$$r = +/-, t = 0 : x_{e0} \leq a_{+/-}$$

$$r = +/-, t = 1 : x_{e1} \leq a_{+/-}$$

$$r = +/-, t = 2 : x_{e2} + x_{h2} \leq a_{+/-}$$

$$r = +/-, t = 3 : x_{e3} + x_{h3} \leq a_{+/-}$$

$$r = +/-, t = 4 : x_{e4} + x_{h4} + x_{j4} \leq a_{+/-}$$

$$r = +/-, t = 5 : x_{h5} + x_{k5} \leq a_{+/-}$$

1/22/2007

Lecture11

gac1

16

Example ILP

- More resource constraints:

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

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

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

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

- 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 Ip_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

1/22/2007

Lecture11

gac1

1

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
```

1/22/2007

Lecture11

gac1

2

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 $i = (i_1, i_2, \dots, i_n)^T$.
 - We will assume l_j and u_j are affine, i.e.

$$l_j(i) = l_j^T i + l_j^0,$$

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

1/22/2007

Lecture11

gac1

3

The Unrolling “Solution”

- Before going further, 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.

1/22/2007

Lecture11

gac1

4

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 $Ai \leq b$, known as a *convex polytope*.

Iteration Space

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

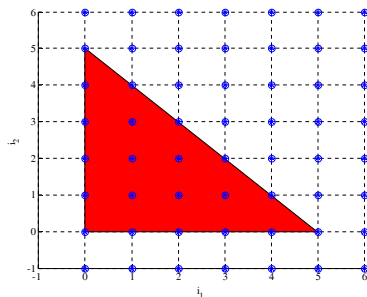
$$A = \begin{pmatrix} -1 & 0 & \dots & 0 & 0 \\ +1 & 0 & \dots & 0 & 0 \\ \underline{l}_{11} & -1 & \dots & 0 & 0 \\ -\underline{u}_{11} & +1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \underline{l}_{n1} & \underline{l}_{n2} & \dots & \underline{l}_{n(n-1)} & -1 \\ -\underline{u}_{n1}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

Iteration Space

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

- Graphical example:

```
for  $i_1 = 0$  to 5
  for  $i_2 = 0$  to  $5 - i_1$ 
    ...
  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  $i_1 = 1$  to 100
  for  $i_2 = 0$  to 100
    s[  $i_1$  ][  $i_2$  ] = s[  $i_1 - 1$  ][  $i_2$  ] + c[  $i_1$  ][  $i_2$  ] * x[  $i_2$  ]
  end
end
```

- In this code, iteration (i_1, i_2) must execute after iteration $(i_1 - 1, i_2)$ due to dependence carried by access to array “s”.
- In the unrolled CFG, 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:

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

- 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” \Leftrightarrow 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

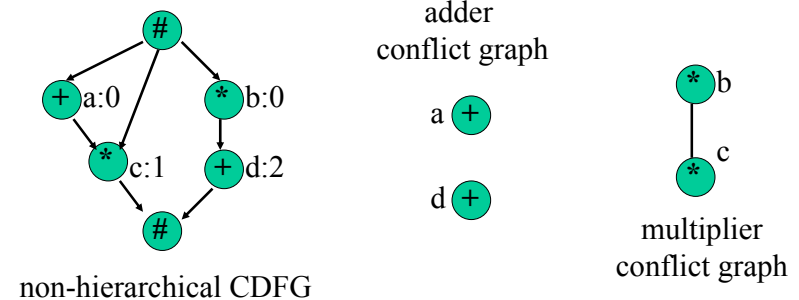
- We will consider some approaches for sharing resources between operations
- Non-hierarchical and hierarchical CDFGs will be considered separately
 - problem has different complexity
- Remember that hierarchical CDFGs can be used to represent the following (Lecture 1)
 - conditionals
 - loops
 - function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



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  $I_s > r$  ) {
       $v_s :=$  first node in L with  $I_s > r$ ;
       $r := r_s$ ;
      label  $v_s$  with colour c
       $L := L \setminus \{v_s\}$ ;
       $c := c + 1$ ;
    }
  }
end
    
```

Left-Edge: Example

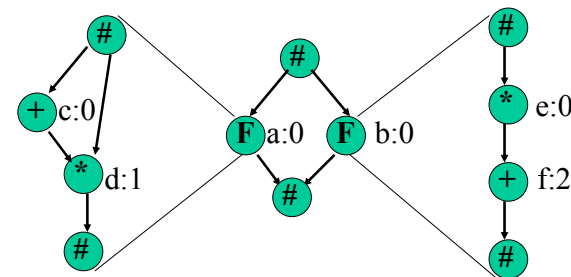
- Taking the previous example:



- 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

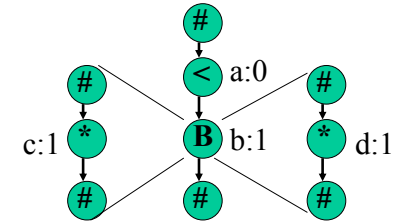
- How do we perform resource sharing?
 - a naïve approach would be to perform resource sharing on each level of the hierarchy in turn
 - for our example, this would lead to one multiplier and one adder for each function: one more adder than we needed for the non-hierarchical version
- We should try to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

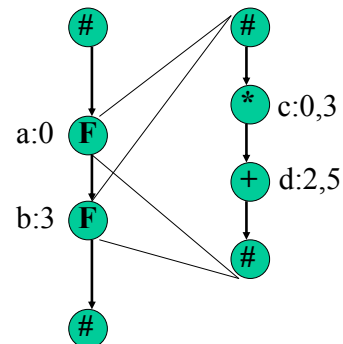
Multiple Function Calls

- 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  $v \in V$  {
     $c = 1$ ;
    while  $\exists (v, v') \in E$ :  $v'$  has colour  $c$ 
       $c = c + 1$ ;
    label  $v$  with colour  $c$  }
end
  
```

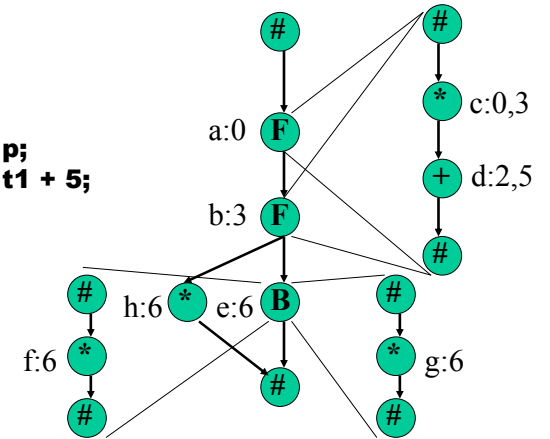
- We will apply it to an example with conditionals and multiple function calls

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)

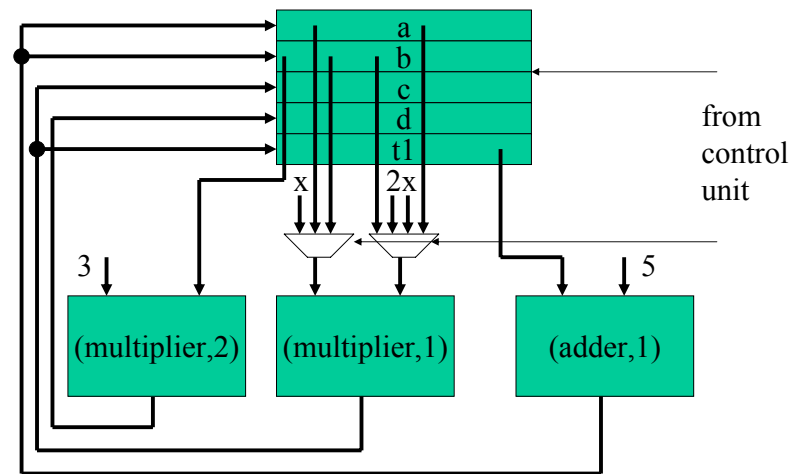


multiplier conflict graph

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

1/22/2007

Lecture13

gac1

1

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

1/22/2007

Lecture13

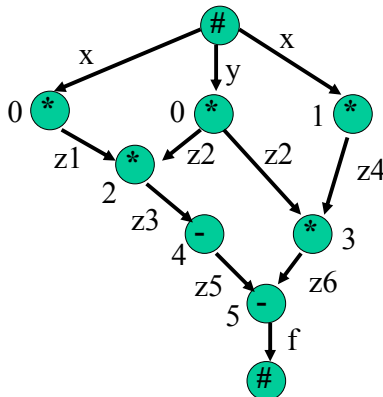
gac1

2

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;
```



1/22/2007

Lecture13

gac1

3

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"

1/22/2007

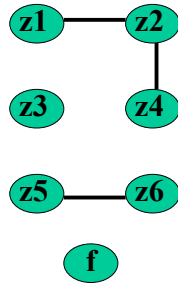
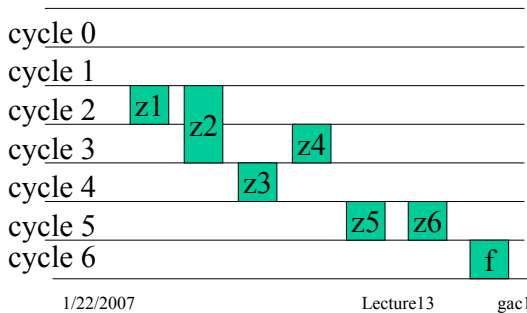
Lecture13

gac1

4

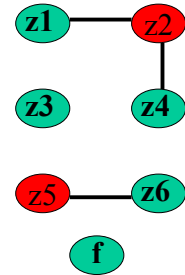
Register Conflict Graph

- 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



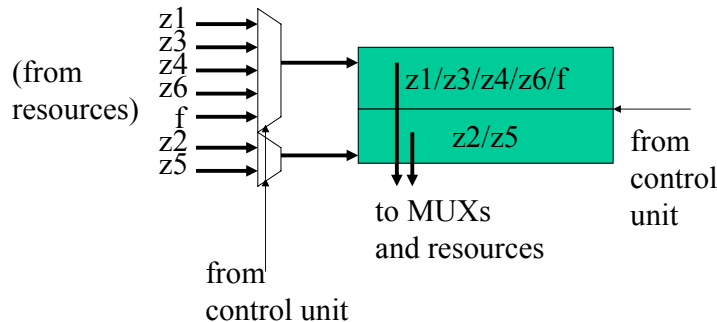
Register Conflict Graph

- 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?

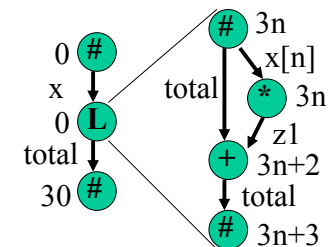


- Note the multiplexers on the register inputs
 - sharing resources leads to MUXs on resource inputs
 - sharing registers leads to MUXs on register inputs

Register sharing for loops

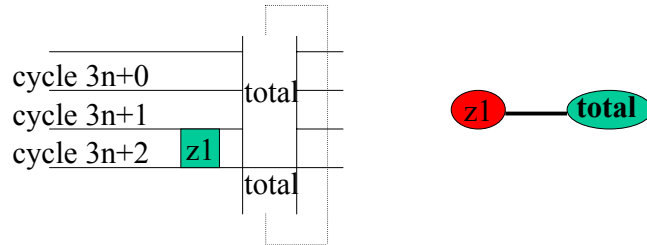
- As with resource sharing, things get more complicated for hierarchical CDFGs
 - we will not consider the general problem
 - but we will examine the effect of loops to give you a glimpse
- Consider the following sum-of-squares code and scheduled CDFG

```
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



1/22/2007

Lecture13

gac1

9

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)

1/22/2007

Lecture13

gac1

10

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

1/22/2007

Lecture13

gac1

11

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

1/22/2007

Lecture13

gac1

12

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

1/22/2007

Lecture14

gac1

1

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

1/22/2007

Lecture14

gac1

2

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

1/22/2007

Lecture14

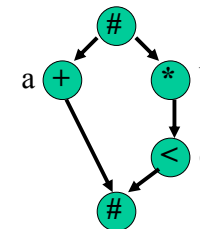
gac1

3

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

1/22/2007

Lecture14

gac1

4

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

1/22/2007

Lecture14

gac1

5

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

1/22/2007

Lecture14

gac1

6

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 $i \in \{1, \dots, a_r\}$ of resource type $r \in R$
 - one variable x_{vtir} exists for all $v \in V, t \in \{\text{ASAP}_v, \dots, \text{ALAP}_v\}, r \in T(v), i \in \{1, \dots, a_r\}$

1/22/2007

Lecture14

gac1

7

ILP Formulation

- $T(v)$ is the *type set* of operation v . For our previous example, $T(*) = *$; $T(<) = \{\text{ALU}, <\}$; $T(+)= \{\text{ALU}, +/-\}$
- 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

1/22/2007

Lecture14

gac1

8

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:

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

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

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

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

- 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

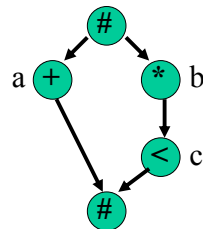
$$\forall (v', v) \in E,$$

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

- 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:

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

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

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

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

ILP Example

- More resource constraints:

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

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

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

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

ILP Example

- More resource constraints:

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

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

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

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

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

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

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

1/22/2007

Lecture15

gac1

1

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

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

1/22/2007

Lecture15

gac1

2

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)

1/22/2007

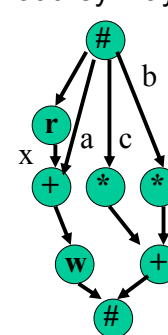
Lecture15

gac1

3

Motivation

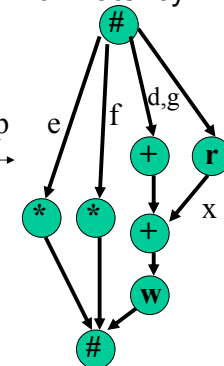
- Comparing the CDFGs of the two inner loops, we can see that they may have different minimum latency.



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

1/22/2007

potential speedup



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

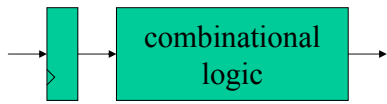
gac1

Lecture15

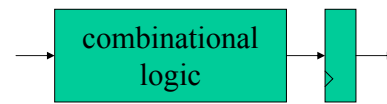
4

Retiming an operator

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



... has identical behaviour to ...



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

1/22/2007

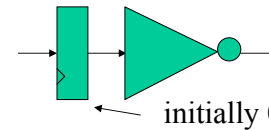
Lecture15

gac1

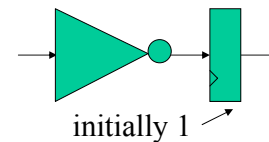
5

The initialization problem

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



... has identical behaviour to ...



1/22/2007

Lecture15

gac1

6

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.

1/22/2007

Lecture15

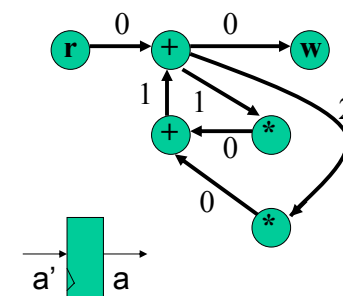
gac1

7

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...

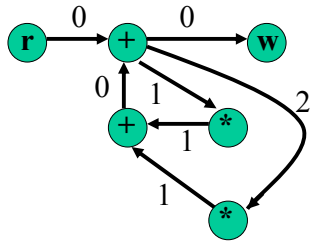
1/22/2007

Lecture15

gac1

8

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 a 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.

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

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

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

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

1/22/2007

Lecture15

gac1

13

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 $(s_v + \text{delay of node } v)$.
- (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.

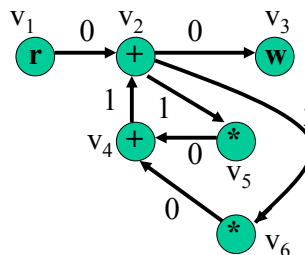
1/22/2007

Lecture15

gac1

14

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 $s_{v_1} = 0$, $s_{v_2} = 0$, $s_{v_3} = 1$, $s_{v_4} = 2$, $s_{v_5} = 0$, $s_{v_6} = 0$, $L = 3$

- The retimed example also corresponds to a feasible solution, with $s_{v_1} = 0$, $s_{v_2} = 1$, $s_{v_3} = 2$, $s_{v_4} = 0$, $s_{v_5} = 0$, $s_{v_6} = 0$, $L = 2$: an improvement!

1/22/2007

Lecture15

gac1

15

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.

1/22/2007

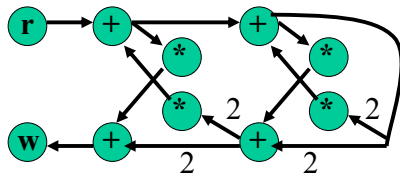
Lecture15

gac1

16

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

1/22/2007

Lecture16

gac1

1

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.

1/22/2007

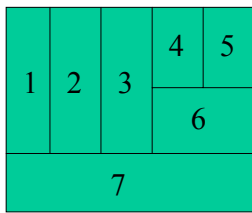
Lecture16

gac1

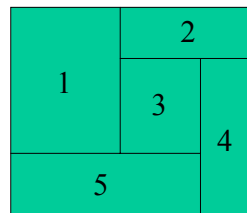
2

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

1/22/2007

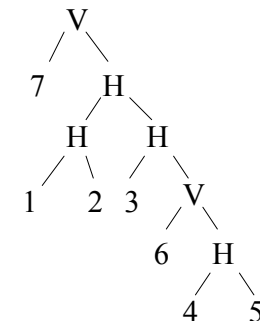
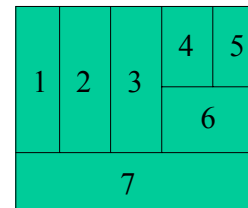
Lecture16

gac1

3

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.



1/22/2007

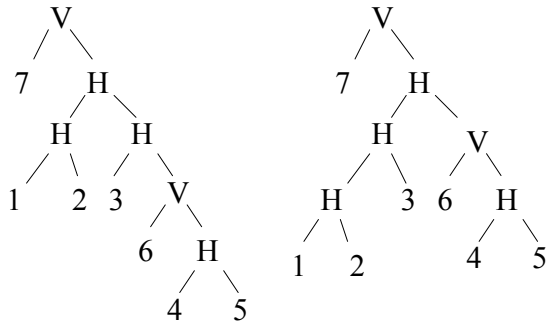
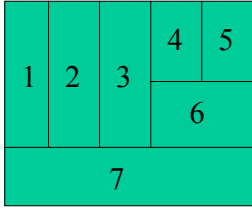
Lecture16

gac1

4

Skewed Slicing Trees

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

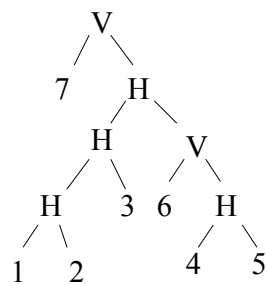



Both slicing trees are valid representations

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: $\text{Polish}(Y) + \text{Polish}(Z) + "X"$ 
- Polish expression for leaf is leaf value.
- For tree on the left: "712H3H645HVHV"

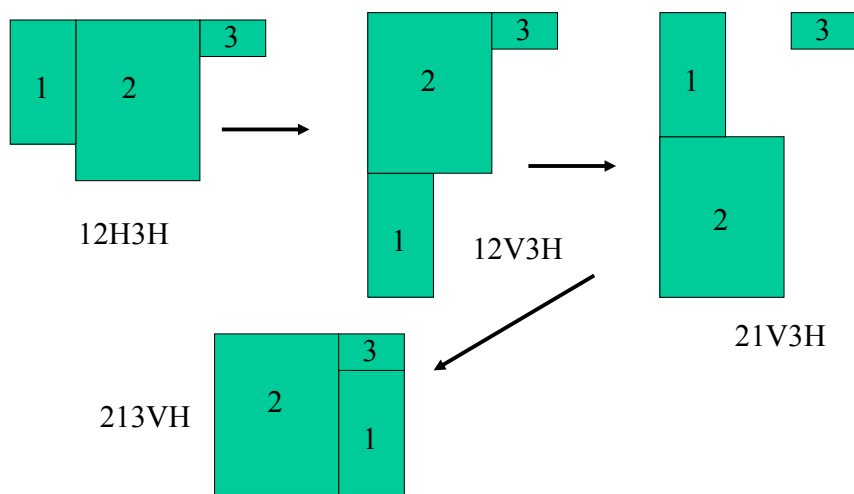
- A skewed slicing tree corresponds to a Polish expression where
 - no two consecutive operators (H/V) are of the same type.

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 moves¹?
 - 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



1/22/2007

Lecture16

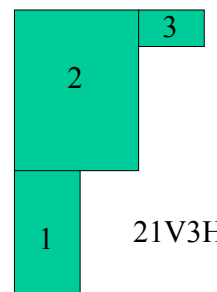
gac1

9

Area Computation

- How to tell whether a move improves area?

- $\text{Height}(XYH) = \max(\text{Height}(X), \text{Height}(Y))$
- $\text{Width}(XYH) = \text{Width}(X) + \text{Width}(Y)$
- $\text{Height}(XYV) = \text{Height}(X) + \text{Height}(Y)$
- $\text{Width}(XYV) = \max(\text{Width}(X), \text{Width}(Y))$



$$\begin{aligned} \text{Height}(21V3H) &= \max(\text{Height}(21V), \text{Height}(3)) \\ &= \max(\text{Height}(2) + \text{Height}(1), \text{Height}(3)) \end{aligned}$$

$$\begin{aligned} \text{Width}(21V3H) &= \text{Width}(21V) + \text{Width}(3) \\ &= \max(\text{Width}(2), \text{Width}(1)) + \text{Width}(3) \end{aligned}$$

1/22/2007

Lecture16

gac1

10

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

1/22/2007

Lecture16

gac1

11

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

1/22/2007

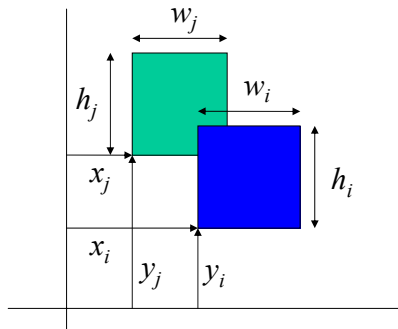
Lecture16

gac1

12

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.

$$x_i + P\delta_{ij} + P\eta_{ij} \geq x_j + w_j \quad (1)$$

$$x_j + P(1 - \delta_{ij}) + P\eta_{ij} \geq x_i + w_i \quad (2)$$

$$y_i + P\delta_{ij} + P(1 - \eta_{ij}) \geq y_j + h_j \quad (3)$$

$$y_j + P(1 - \delta_{ij}) + P(1 - \eta_{ij}) \geq y_i + h_i \quad (4)$$

$$\delta_{ij}, \eta_{ij} \in \mathbf{B}$$

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.

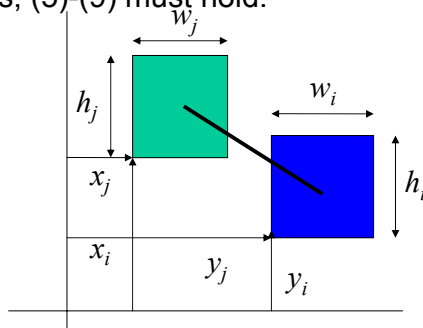
$$x_i + 0.5w_i - x_j - 0.5w_j \leq W_{ij}^h \quad (5)$$

$$-x_i - 0.5w_i + x_j + 0.5w_j \leq W_{ij}^h \quad (6)$$

$$y_i + 0.5h_i - y_j - 0.5h_j \leq W_{ij}^v \quad (7)$$

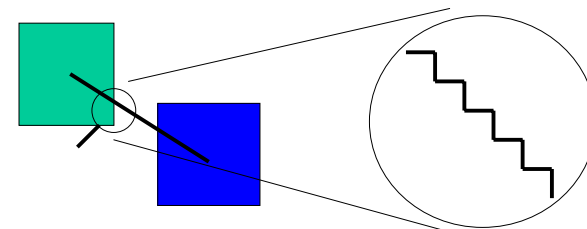
$$-y_i - 0.5h_i + y_j + 0.5h_j \leq W_{ij}^v \quad (8)$$

$$W_{ij} = W_{ij}^h + W_{ij}^v \quad (9)$$



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.

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

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

- 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-problems¹.
 - sub-problems can be stitched into suboptimal solutions.

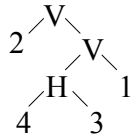
¹Sutanthavibul, Schragowitz, and Rosen, IEEE Trans CAD 10(6), 1991.
Smith, Constantinides, and Cheung, Proc. Field-Programmable Logic, 2005 (in the context of FPGA design).

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

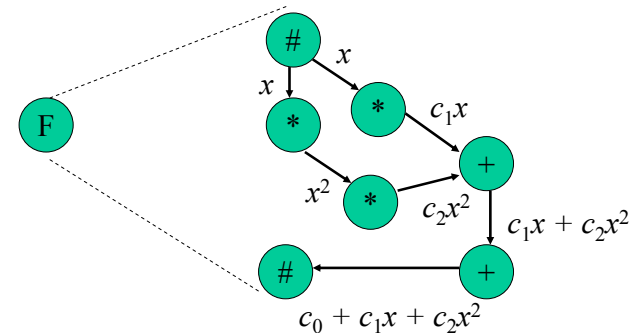
- Throughout much of the course, we have used multiplication and addition as the key operations
- There are typically pre-designed library blocks for adder and multiplier resources
- Not necessarily the case for more complex functions: $\sin(x)$, $\cos(x)$, e^x , etc.
- In this lecture we investigate 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
 - $f(x) = c_0 + c_1x + c_2x^2 + \dots + c_nx^n$
- This suggests a solution to our problem: find a polynomial “close enough” to the function, and then use mults and adds to evaluate it

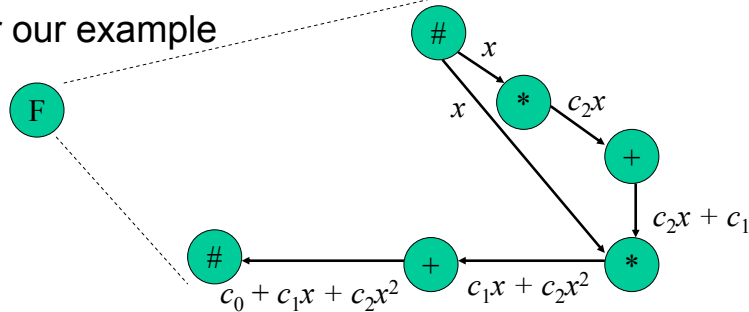
A Simple Evaluation Scheme

- 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
 - $f(x) = c_0 + c_1x + c_2x^2 + \dots + c_nx^n$
 - re-write: $f(x) = (\dots((c_nx + c_{n-1})x + c_{n-2})x + \dots + c_1)x + c_0$
- 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:
 - to within a worst case error ϵ , i.e. $\max_x |f(x) - g(x)| < \epsilon$
 - in the least-squares sense, i.e.

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

- $w(x)$ 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.

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

- Then it is easy to calculate a_i

Least-Squares Approximations

- If we define the inner product

$$\langle f, g \rangle = \int_x f(x)g(x)dx$$
- Then the coefficients minimizing the least-squares error are

$$a_i = \frac{\langle f, \phi_i \rangle}{\langle \phi_i, \phi_i \rangle}$$

Least-Squares Approximations

- Proof: We are trying to minimize

$$\begin{aligned}
 E &= \int_x \left(f(x) - \sum_{i=0}^n a_i \phi_i(x) \right)^2 dx \\
 &= \int_x f^2(x) - 2 \sum_{i=0}^n a_i \int_x f(x) \phi_i(x) dx + \sum_{i=0}^n \sum_{j=0}^n a_i a_j \int_x \phi_i(x) \phi_j(x) dx \\
 &= \int_x f^2(x) - 2 \sum_{i=0}^n a_i \langle f, \phi_i \rangle + \sum_{i=0}^n a_i^2 \langle \phi_i, \phi_i \rangle
 \end{aligned}$$

1/22/2007

Lecture16

gac1

9

Least-Squares Approximations

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

$$\begin{aligned}
 \frac{\partial E}{\partial a_i} &= -2 \langle f, \phi_i \rangle + 2a_i \langle \phi_i, \phi_i \rangle = 0 \\
 \Rightarrow a_i &= \frac{\langle f, \phi_i \rangle}{\langle \phi_i, \phi_i \rangle}
 \end{aligned}$$

- This ease of derivation makes least-squares solutions popular

1/22/2007

Lecture16

gac1

10

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

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

1/22/2007

Lecture16

gac1

11

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

1/22/2007

Lecture16

gac1

12

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

1/22/2007

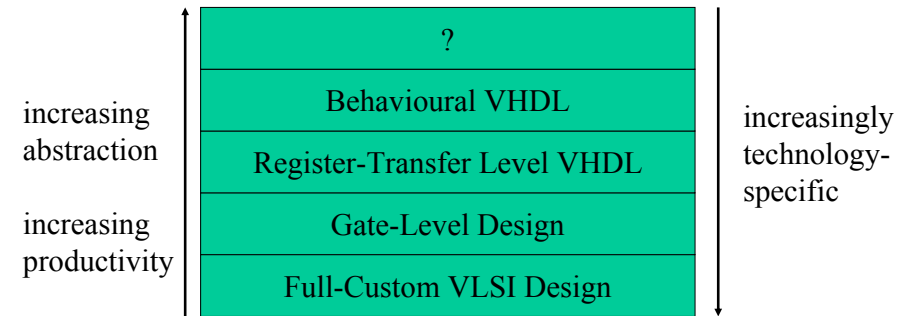
Lecture17

gac1

1

Levels of Abstraction in Design

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



1/22/2007

Lecture17

gac1

2

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

1/22/2007

Lecture17

gac1

3

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

1/22/2007

Lecture17

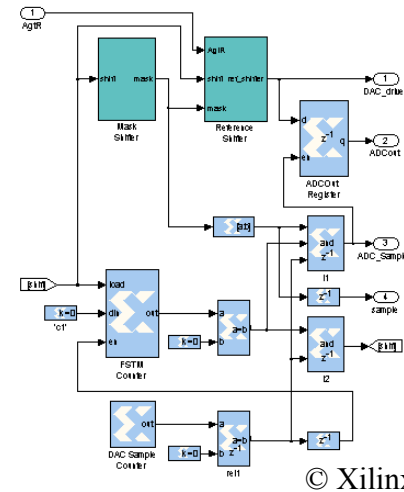
gac1

4

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



- 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 to 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, \dots, m_n
- Represent x as a list ($x \bmod m_1, x \bmod m_2, \dots, x \bmod m_n$)
 - we can represent up to $m_1 m_2 \dots m_n$ numbers uniquely like this
 - we can perform arithmetic on the list of numbers, e.g. for $n=2, m_1=3, m_2=5$: $4 = (1,4)$, $3 = (0,3)$, $4*3 = (1*0, 4*3) = (0, 12 \bmod 5) = (0, 2)$

Residue Number Systems

- Key point: We can do arithmetic on each of the list elements *in parallel*
 - if $\max(\lceil \log_2 m_1 \rceil, \lceil \log_2 m_2 \rceil, \dots, \lceil \log_2 m_n \rceil) < \lceil \log_2(m_1 m_2 \dots m_n) \rceil$, 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

1/22/2007

Lecture18

gac1

1

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

1/22/2007

Lecture18

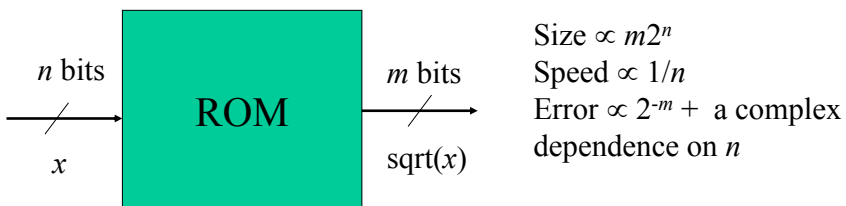
gac1

2

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:



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

1/22/2007

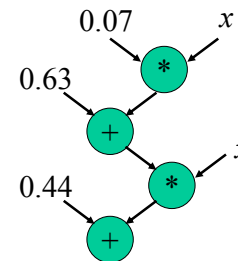
Lecture18

gac1

3

Function Approximation

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



- 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?

1/22/2007

Lecture18

gac1

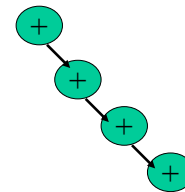
4

Function Approximation

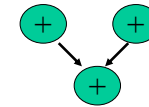
- Different solutions will have different area, arithmetic error, power, and speed characteristics
- The challenge is to decide automatically when to use which type of function approximation
 - we have started to investigate this issue (Dr Nalin Sidahao and Mr Gareth Morris)

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 $127x \rightarrow 128x - x = (x \ll 7) - x$
 - “ $\ll 7$ ” 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]

1/22/2007

Lecture18

gac1

9

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)

1/22/2007

Lecture18

gac1

10

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).

1/22/2007

Lecture18

gac1

11

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

1/22/2007

Lecture18

gac1

12