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.

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!

What is Synthesis?

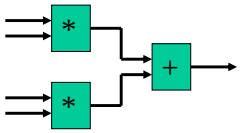
- Synthesis is the automatic mapping from a high-level description to a low-level description
  - gates to transistors
  - AHDL or VHDL to gates
  - Matlab to AHDL/VHDL (?)
- Synthesis is important because
  - it raises designer productivity
  - it allows design-space exploration
  - it improves time-to-market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

What is architectural synthesis?

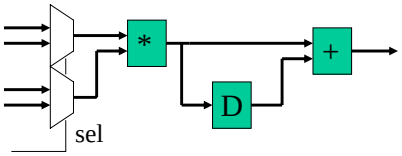
- Current synthesis comes in two main "flavours", depending on what is the input description and what is the output description
  - logic synthesis
    - given Boolean equations, map them into gates
  - architectural ("high-level") synthesis
    - given a description of circuit behaviour, create an architecture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

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

Researchers Wanted!

- The Circuits and Systems group has active research in this field
  - Are you a 1st or 2/1 student?
  - Do you want to contribute to knowledge in:
    - FPGAs?
    - Synthesis algorithms?
    - High-performance parallel computation?
  - I have openings for up to two PhD students in
    - Architectures, algorithms, and hardware for optimization (in collaboration with Control, Cambridge, and NTU).
    - Email gac1@ic.ac.uk

Advance Warning

- I promise to try and avoid tiredness induced grumpiness.
- Help me:
  - Talk only when asking **me** questions
  - Turn off mobile phones
  - Come on time

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
```

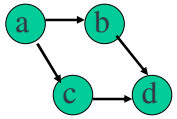
```
xl = x + dx;
ul = u – 3*x*u*dx –
     3*y*dx;
yl = y + u*dx;
c = xl < a;
```


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

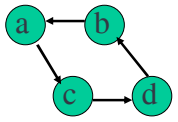
Directed Graphs



V = {a,b,c,d}

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

This graph is acyclic

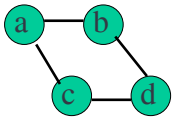


V = {a,b,c,d}

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

This graph is cyclic

Undirected Graph



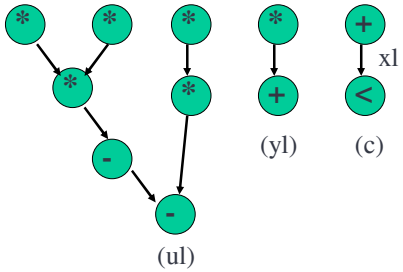
V = {a,b,c,d}

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;
```



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;
```

→

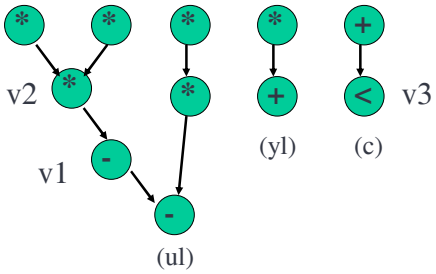
```
xl = x + dx;
t1 = 3*x;
t2 = u*dx;
t3 = t1*t2;
t4 = 3*y;
t5 = t4*dx;
t6 = u – t3;
ul = t6 - t5;
t7 = u*dx;
yl = y + t7;
c = xl < a;
```


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

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

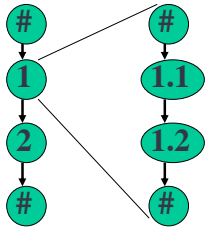


Data Flow Graphs

- 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

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



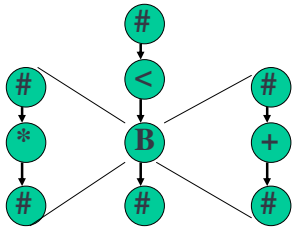
e.g.

1) have a beer 1.1) go to pub
2) have a curry 1.2) order and drink

Conditionals

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

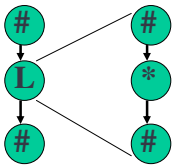
```
a = b < c;
if (a) then
  d = b * c;
else
  d = b + c;
```



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;
```

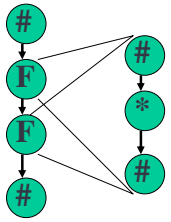


Function Calls

- 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

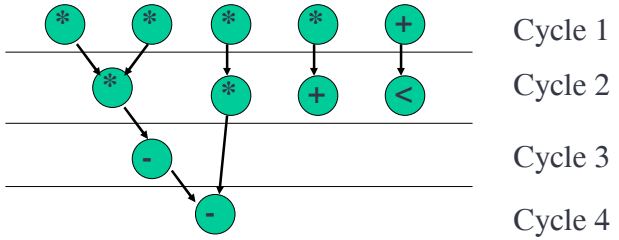
```
a = fun(x);
b = fun(y);
```

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



A Simple Schedule

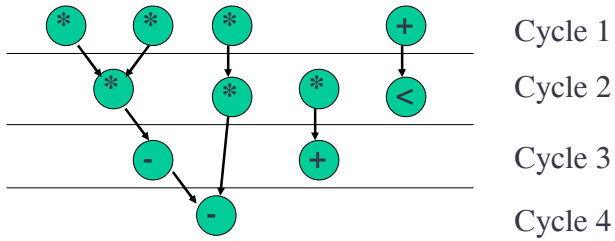
- What if we had unlimited hardware? How fast could we make our algorithm go?
- Let’s assume that each operation takes one clock cycle



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

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

Schedule: Definition

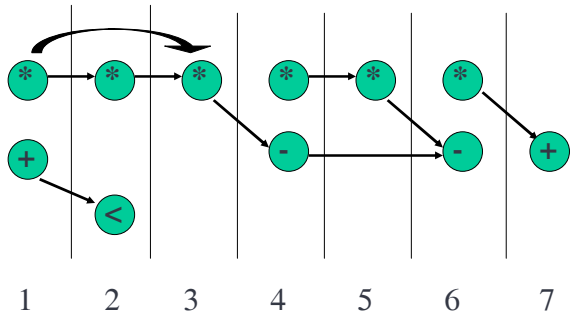
- With each node v in the graph G(V,E), let us associate an execution time $d(v)$
- A schedule of this graph is a function S:V→N where for all edges $(v_1, v_2) \in E$,

  $S(v_2) \geq S(v_1) + d(v_1)$
- For each node v, S(v) denotes the start time of the relevant task

Resource Constrained Schedules

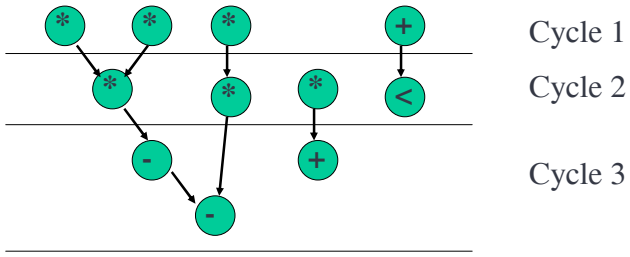
- Often our scheduling task consists of finding a schedule that will complete in a short time, subject to constraints on the amount of hardware available
- This is called “Resource Constrained Scheduling”
- Example: Schedule the differential equation code such that we need no more than:
  - one multiplier,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

Resource Constrained Schedules



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.



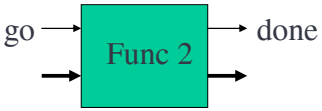
Scheduling with Chaining

- 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

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

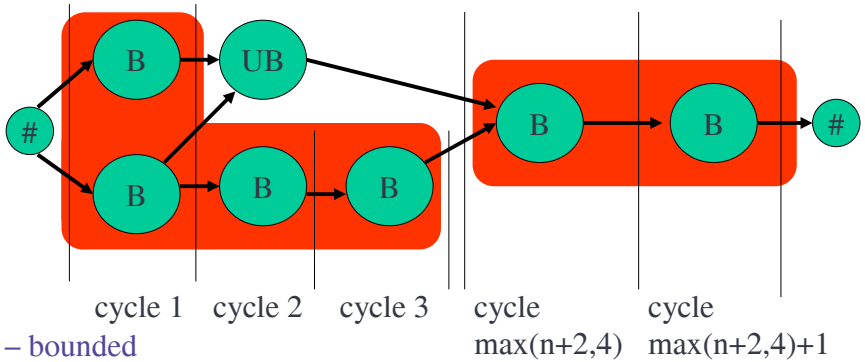
  - Unbounded-delay tasks

```
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

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

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 = {multiplier, adder, ALU}

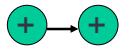
Resource Sharing

- 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.

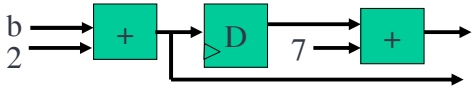
Resource Sharing

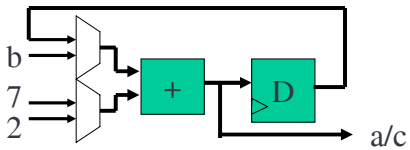
- Consider the code below and its scheduled DFG

```
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

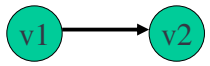
Need to generate select signals

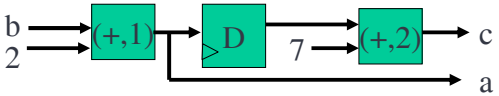
Resource Binding

- 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

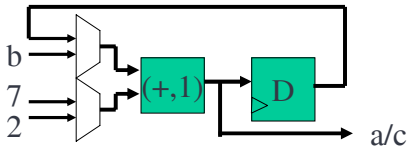
- Revisiting our example…





Y(v1) = (+,1)

Y(v2) = (+,2)

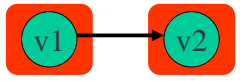


Y(v1) = (+,1)

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 $G'(V, E \cup E_B)$



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

$E_B = \{ \{v1\}, \{v2\} \}$

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

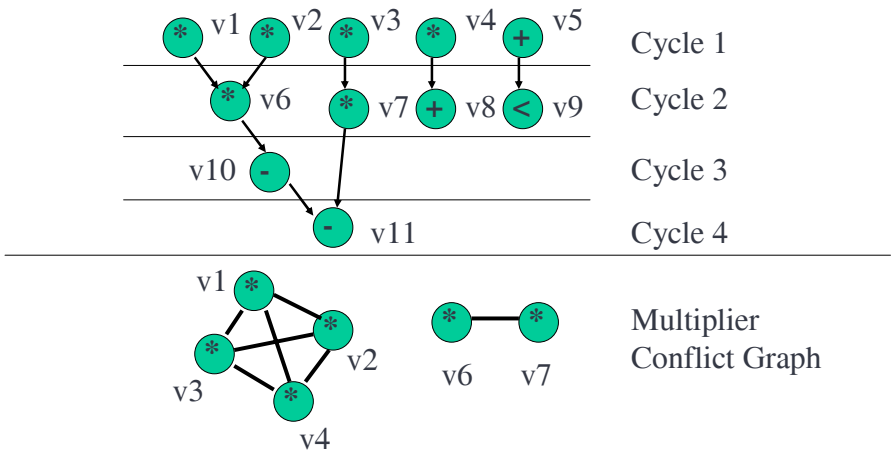
$E_B = \{ \{v1,v2\} \}$

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

Conflict Graphs

- Our example from Lecture 1



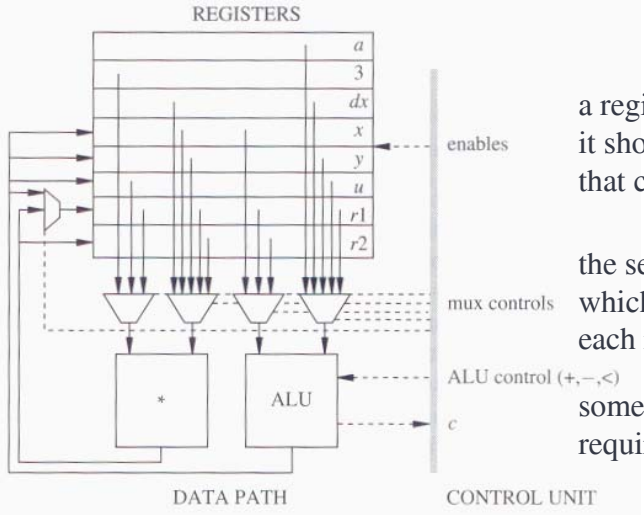
Conflict Graphs

- 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…)

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

Architectural Templates



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

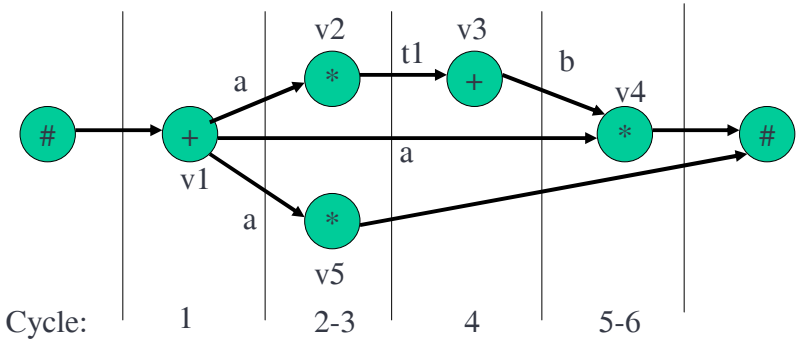
the select-lines decide which register to send to each resource

some resources may require additional control

Worked Problem

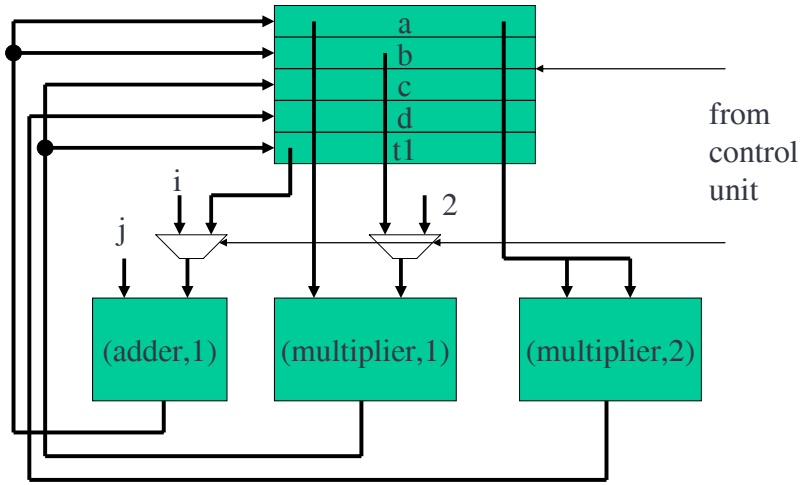
- Consider the following code:
  - a = i+j; b = 2*a + j; c = a * b; d = a*a;
  - (a) construct a CDFG for the code
  - (b) schedule the graph so that each operation starts as soon as it can, assuming each multiplication takes two cycles and each addition takes one cycle
  - (c) if you have the resource type set R = {adder, 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) = (adder,1) Y(v2) = (multiplier,1)
Y(v3) = (adder,1) Y(v4) = (multiplier,1)
Y(v5) = (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

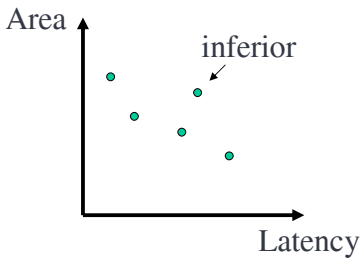
- There can be several objectives when performing a circuit design
  - small area
  - low latency
  - [high throughput] (for pipelined circuits)
  - high max clock rate
  - low power
- Often these objectives are conflicting, and we must trade-off between them

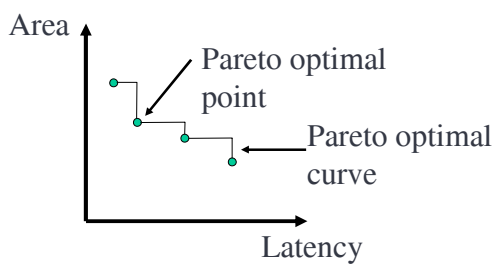
Design space parameters

- 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)





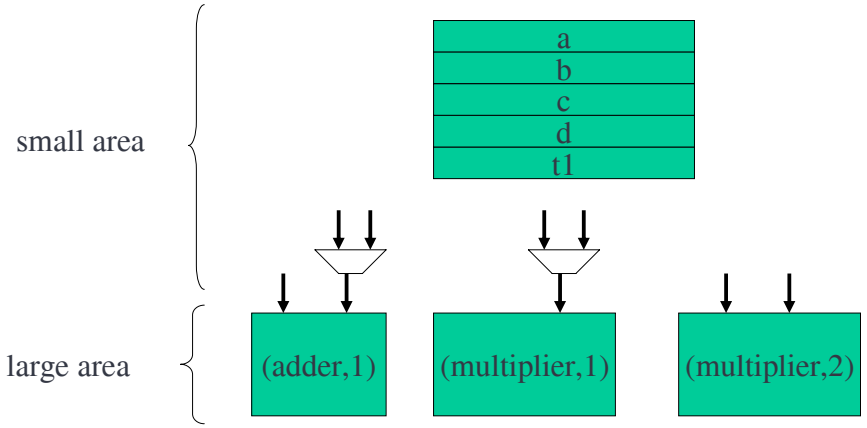
Why estimation?

- 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*

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: $A = A_{add} N_{add} + A_{mult} N_{mult} + \ldots$
  - Latency: known from schedule
- Often DSP circuits tend to be resource dominated
- Example: the worked example from last lecture…

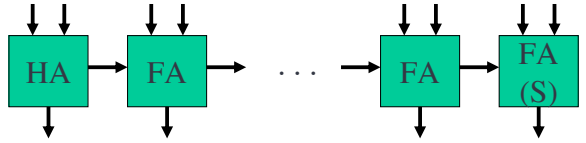
Resource domination



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

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:



$$\text{Area} = A_{HA} + A_{FA(S)} + (n\text{-}2)A_{FA}$$

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

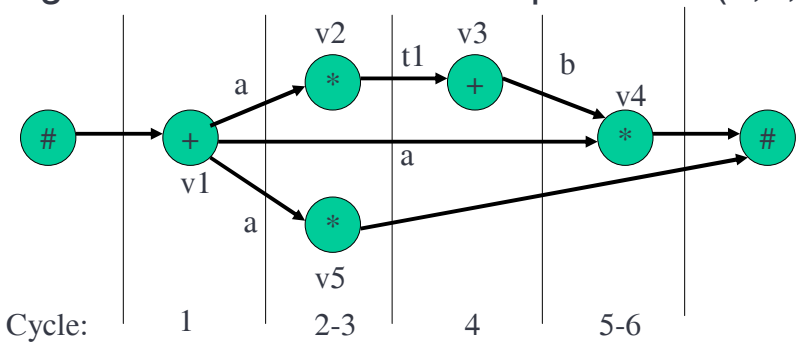
(would be different for a CLA)

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)

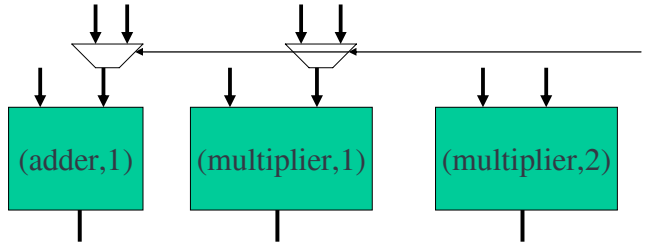


Registers

- 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

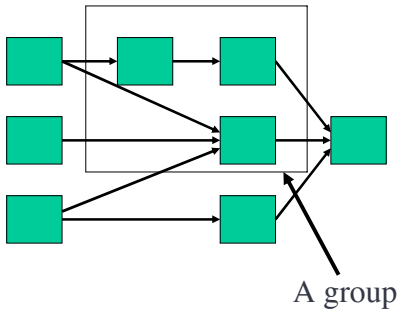


- The number and size of multiplexers required is known from the binding
- Multiplexers can consume area and add delay – the delay added can depend on the number of MUX inputs

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

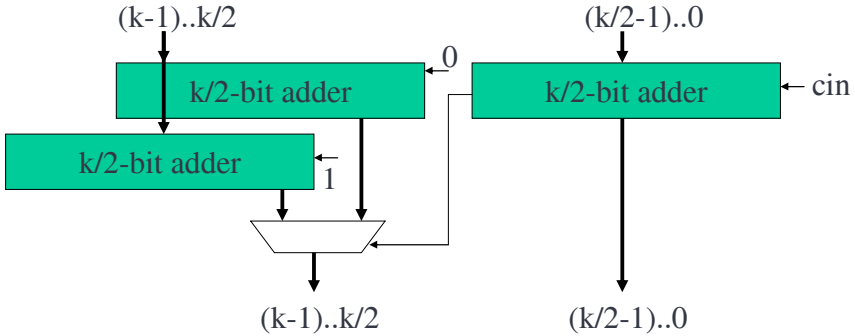
- Some examples of Rent constants
  - SRAM: $\beta = 0.12$, $K = 6$
  - µP: $\beta = 0.45$, $K = 0.82$
  - Gate Array: $\beta = 0.50$, $K = 1.9$

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 = $3A_{FA}(k/2) + A_{MUX}(k/2) = c_1 k$
- Delay = $T_{FA}(k/2) + T_{MUX} = c_2 k + c_3$

- 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 β=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 ≈ (1e6)^0.25 ≈ 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 ≈ 2(5e5)^0.25 ≈ 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

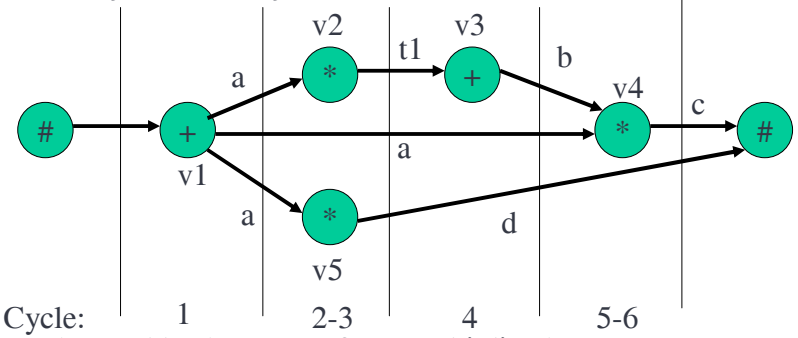
- Part of a 4-lecture introduction
  - Scheduling
  - Resource binding
  - Area and performance estimation
  - Control unit synthesis
- This lecture covers
  - Microcode and microcode optimization
  - Hardwired control
  - Control with interacting state machines

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;



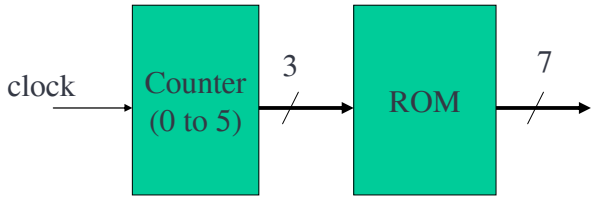
Y(v1) = (adder,1) Y(v2) = (multiplier,1)
Y(v3) = (adder,1) Y(v4) = (multiplier,1)
Y(v5) = (multiplier,2)

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 ($\lceil \log_2 \#\text{cycles} \rceil$)



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)

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

Horizontal Microcode

- This is known as "horizontal" microcode
  - # states << # 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

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

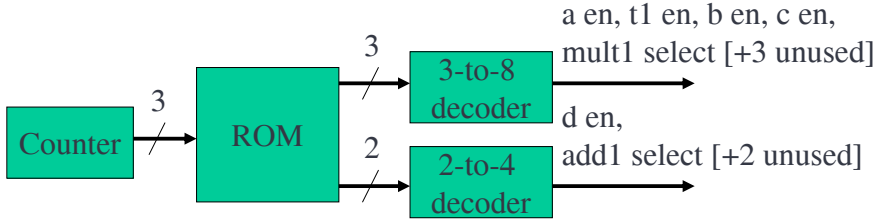
- If we didn't require any control signals concurrently, we would only require a $\lceil \log_2(\#\text{control sigs}+1) \rceil$-bit ROM data bus
  - we could then use an *n* to 2^n decoder to generate the control signals
  - (revision… an *n* to 2^n decoder asserts one of 2^n different possible outputs 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:

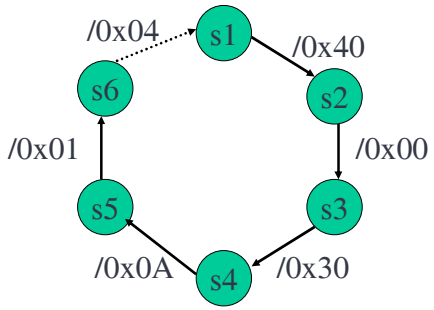
Address	Data	Address	Data
0x0	0b11100	0x3	0b10110
0x1	0b00000	0x4	0b01100
0x2	0b11011	0x5	0b10000

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

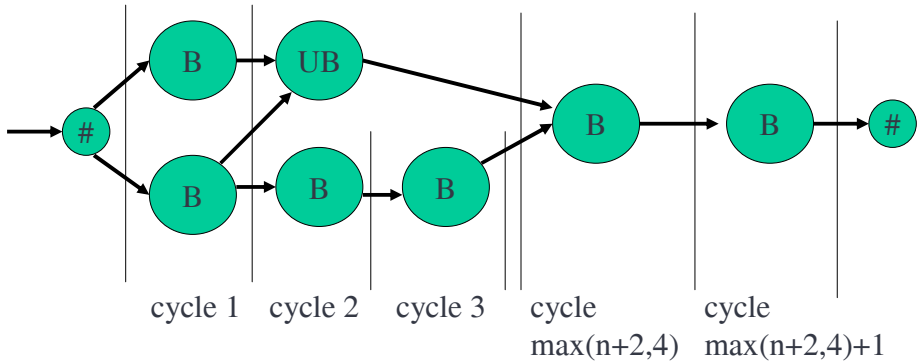
Bounded Latency Example

- 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



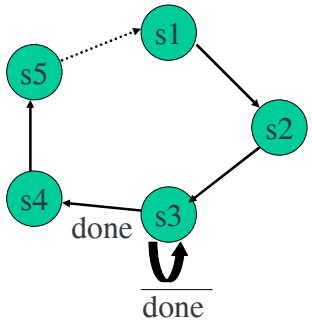
Unbounded Latency Operations

- 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



(FSM outputs not shown)

Local Controllers

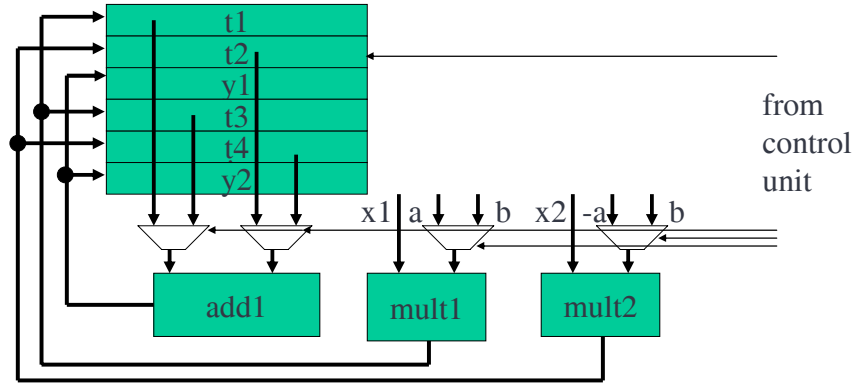
- So far we have assumed that there is one large controller for the whole circuit
  - this can be efficient as it allows the overhead of controller design and implementation to be shared by many control signals
  - however it may be impractical for large circuits due to the need to route control signals 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. (x1, x2 are inputs; y1, y2 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)

```
t1 = a*x1;    // Cycle 1, mult1      t3 = b*x1;    // Cycle 2, mult1
t2 = b*x2;    // Cycle 1, mult2      t4 = -a*x2;   // Cycle 2, mult2
y1 = t1 + t2; // Cycle 2, add1       y2 = t3 + t4; // Cycle 3, add1
```



Worked Example (b)

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

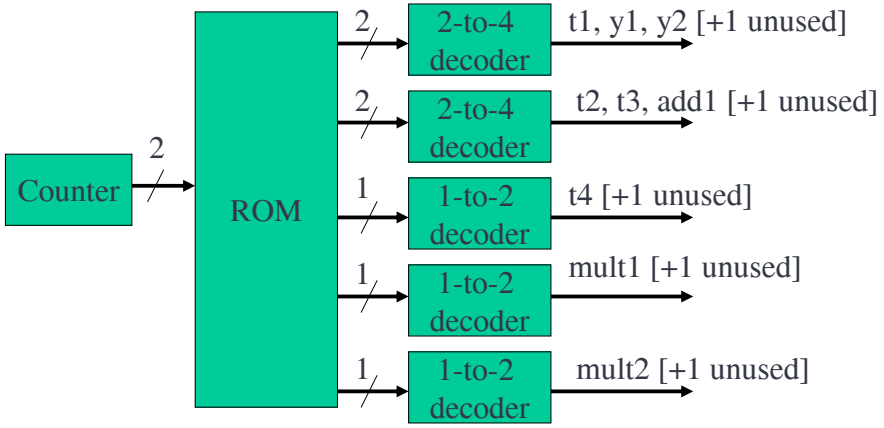
- assume data bus ordering (MSB to LSB): enables (t1,t2,y1,t3,t4,y2); select lines (add1, mult1, mult2)

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

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} => need 2 bits
- Set 2: {t2, t3, add1} => need 2 bits
- Set 3: {t4} => need 1 bit
- Set 4: {mult1} => need 1 bit
- Set 5: {mult2} => need 1 bit

Worked Example (c)



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		

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

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

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