Register Sharing

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

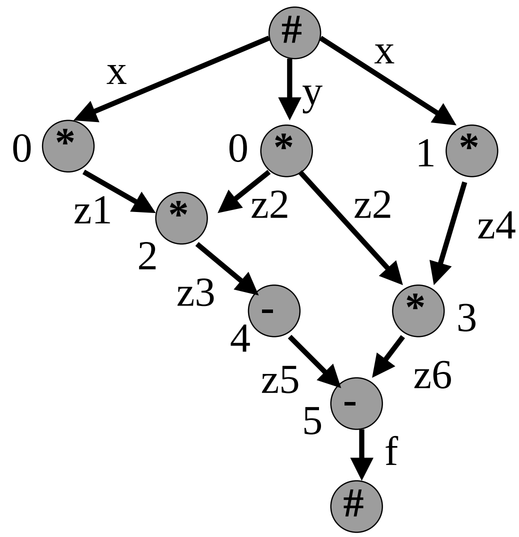
Register Sharing

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

Lifetime Analysis

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

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

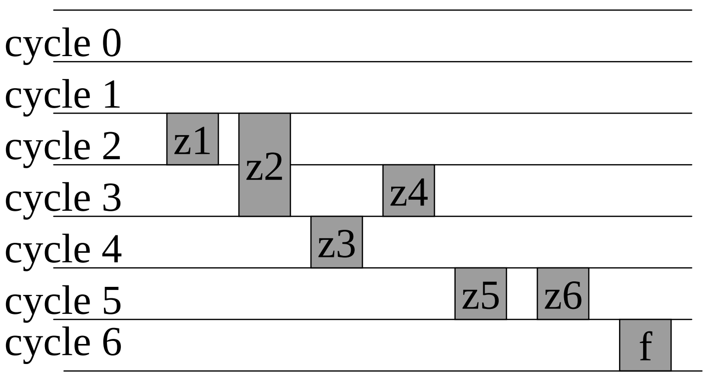


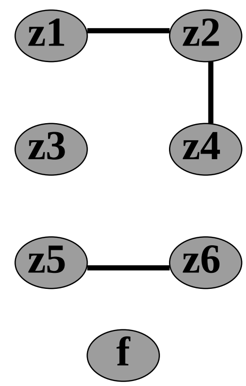
Lifetime Analysis

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

Register Conflict Graph

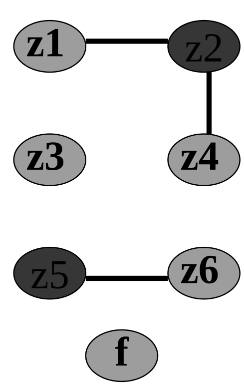
- Two results cannot share a register if their lifetimes overlap
  – we can thus create a register conflict graph just like the resource conflict graph used in the previous lecture





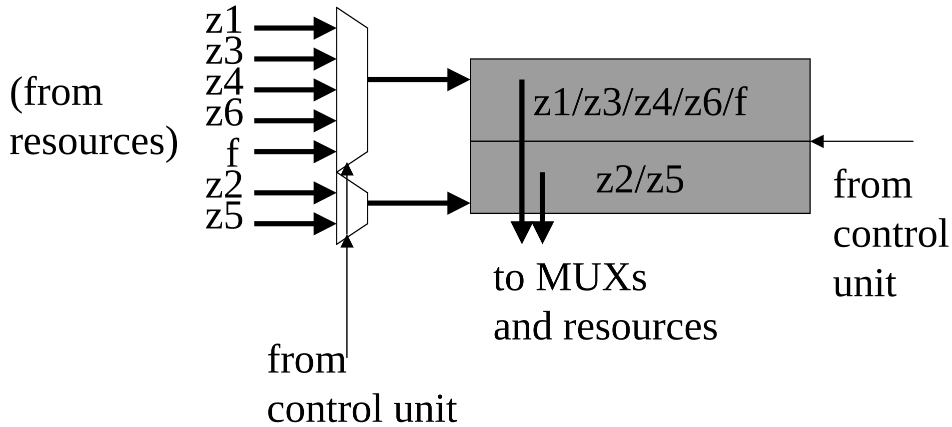
Register Conflict Graph

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



Example Datapath

- So what would the datapath be for that design?

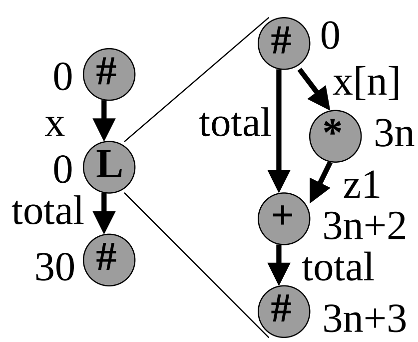


- Note the multiplexers on the register inputs
  - sharing resources leads to MUXs on resource inputs
  - sharing registers leads to MUXs on register inputs

Register sharing for loops

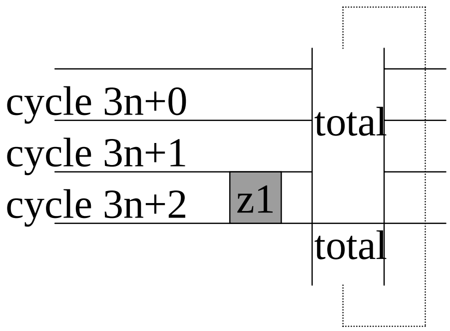
- As with resource sharing, things get more complicated for hierarchical CDFGs
  - we will not consider the general problem
  - but we will examine the effect of loops to give you a glimpse
- Consider the following sum-of-squares code and scheduled CDFG

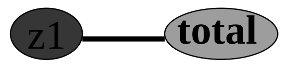
```
total = 0;
for n=0 to 9
  z1 = x[n]*x[n];
  total = total + z1;
end
```



Register sharing for loops

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





Register sharing for loops

- Because of the “circular arc” wrap around effect with some variables, the conflict graphs for hierarchical CDFGs are not always interval graphs

- Colouring such general graphs is NP-hard, requiring the use of our colouring heuristic (or similar)

Summary

- We have investigated register sharing:
  - Variable lifetime calculation
  - Register conflict graphs
  - Non-hierarchical register sharing
  - Hierarchical register sharing: the loop problem

- Next lecture we will look at the module selection problem

Suggested Problems

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

Module Selection

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

Module Selection

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

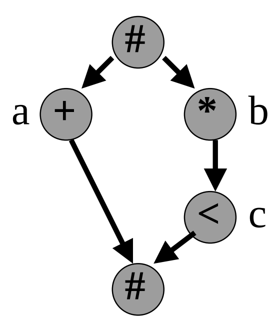
Interactions

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

Interactions

- For example, consider the code and CDFG below

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



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

Interactions

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

Interactions

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

ILP Formulation

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

ILP Formulation

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

ILP Formulation

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

ILP Formulation

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

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

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

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

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

- 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

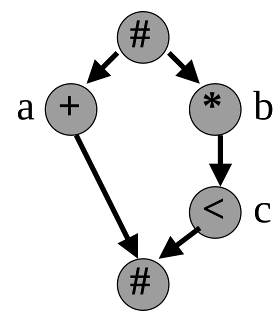
$$\forall (v', v) \in E,$$

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

- 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:

$$\text{minimize}:$$
$$2b_{1,*} + 1b_{1,+} + 1b_{1,<} + 1.5(b_{1,ALU} + b_{2,ALU})$$

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:

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

Summary

- This lecture covers
  - The module selection problem
  - Module selection / scheduling / binding interaction
  - An ILP formulation

- Next lecture we will examine perspectives for the future of architectural synthesis

Suggested Problems

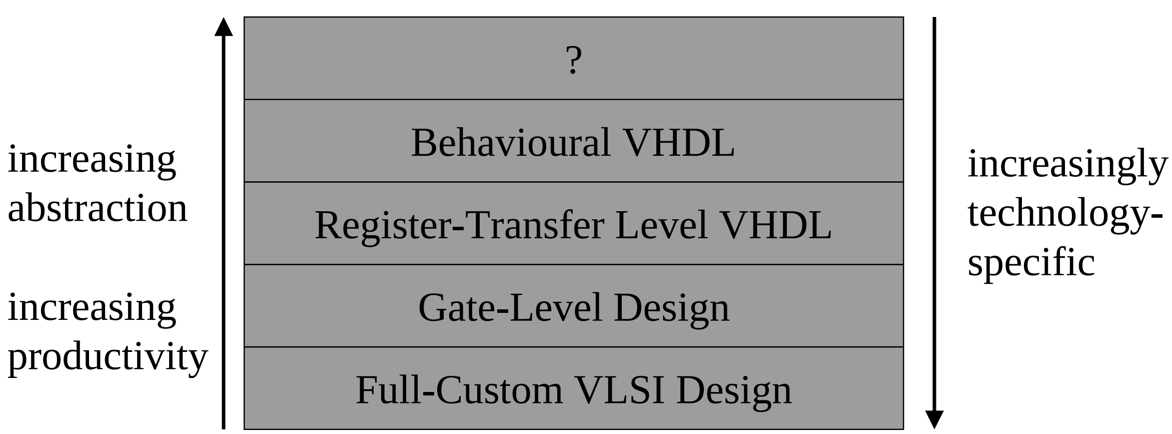
- Download a copy of lp_solve from the website given at the start of Lecture 11, and solve the ILP example
  - what is the minimum possible cost? (*)
  - how many adders, multipliers, comparators and ALUs does it use? (*)
  - how many variables and constraints are there? (*)
  - how do you think the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

Perspectives I

- The final portion of the course covers
  - – Scheduling algorithms
  - – Resource sharing algorithms
  - – Module selection
  - – Perspectives for the future
- This lecture (part one of two) covers
  - – Abstract design representations
  - – Word-length optimization
  - – Number representations

Levels of Abstraction in Design

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



Why [not?] C

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

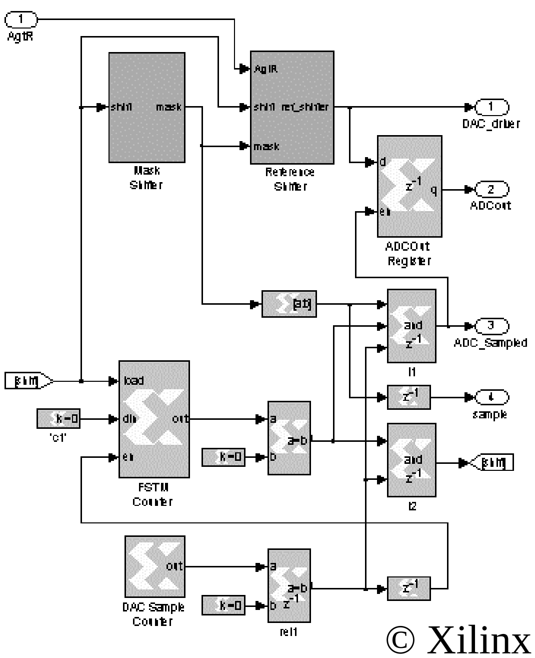
Why [not?] C

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

Simulink

- I believe a more promising approach is to target specific problem domains
  - Simulink is widely used in Control and DSP, so use it as a specification format in these domains
  - We have developed a tool for synthesis from Simulink (http://infoeng.ee.ic.ac.uk/~gac1/pubs.html)
  - 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

© Xilinx

Matlab

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

Mathematical Specifications

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

Word-Length Optimization

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

Word-Length Optimization

- This problem is one of my research areas
- Our research has produced a tool (Synoptix)
  - synthesizes a low-area implementation by selecting the internal word-lengths appropriately
  - input format is Simulink
  - output format is structural VHDL
  - http://infoeng.ee.ic.ac.uk/~gac1/pubs.html
- Our tool is currently restricted to systems with certain mathematical properties
  - we are actively researching extensions

Logarithmic Representations

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

Residue Number Systems

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

Residue Number Systems

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

Number System Selection

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

Summary

- This lecture (part one of two) has covered
  - Abstract design representations
  - Word-length optimization
  - Number representations

- Next lecture will continue to examine some future directions for architectural synthesis

Perspectives II

- The final portion of the course covers
  – Scheduling algorithms
  – Resource sharing algorithms
  – Module selection
  – Perspectives for the future
- This lecture (part two of two) covers
  – Function approximation
  – Mathematical transformations
  – Hardware / Software partitioning
  – Memory synthesis

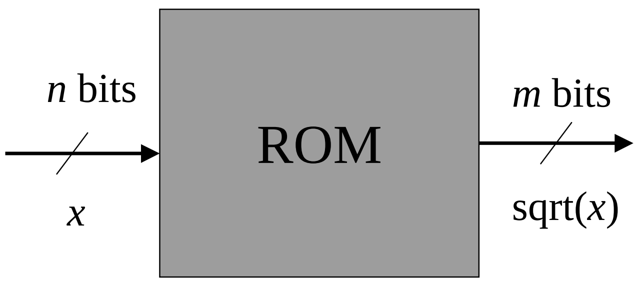
Function Approximation

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

Function Approximation

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

- Simple lookup table approach:

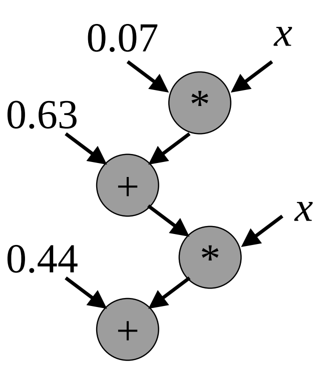


Size ∝ $m2^n$
Speed ∝ $1/n$ [maybe]
Error ∝ 2^{-m} + a complex dependence on n

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

Function Approximation

- Polynomial approximation:
  – Over [1,2], sqrt(x) ≈ $0.44 + 0.63x + 0.07x^2$
  $= 0.44 + x(0.63 + 0.07x)$



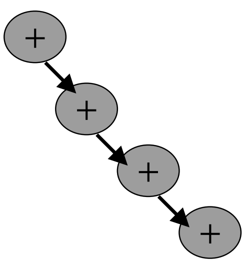
- Many tradeoffs are possible
  - how many bits used to represent coefficient?
  - how many bits to represent internal variables?
  - how many polynomial terms?
  - what type of approximation?
    - worst-case, or average case?

Function Approximation

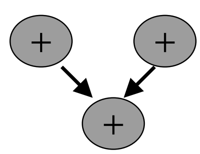
- Different solutions will have different area, arithmetic error, power, and speed characteristics
- The challenge is to decide automatically when to use which type of function approximation
  - we have started to investigate this issue (http://infoeng.ee.ic.ac.uk/~ns999)

Mathematical Transformations

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



Can be scheduled in 4 time units using a single adder



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

Mathematical Transformations

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

Mathematical Transformations

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

Hardware / Software Partitioning

- Large scale designs of embedded systems typically have a hardware portion and a software portion
- The designer must decide which tasks are best done in software, and which in hardware
  - software can be slow, power-hungry, and cheap
  - hardware can be fast, power-efficient, and expensive
  - hardware can only be significantly faster if the application can be parallelized
- Could this task be done automatically?
  - Our research group has been addressing this problem for configurable hardware based on Field-Programmable Gate Arrays (FPGAs)
  - http://infoeng.ee.ic.ac.uk/~tw1

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
  - e.g. what variables should be stored where in memory in order to minimize power consumption?
  - http://infoeng.ee.ic.ac.uk/~samhet

Summary

- This lecture (part two of two) has covered
  – Function approximation
  – Mathematical transformations
  – Hardware / Software partitioning
  – Memory synthesis

- Next lecture will summarize the entire course, and allow you to focus on topics for revision

Course Summary

- There is no more new material in the course
- This lecture summarizes
  – Scheduling (ASAP/ALAP and list), binding, and estimation
  – Register sharing
  – Techniques for control synthesis
  – Graphs and combinatorial problems
  – Algorithms and intractability
  – Integer linear programming and its applications
    - ILP scheduling, module selection
  – Path problems

ASAP / ALAP Scheduling

- We use a control/data-flow graph (CDFG) or data-flow graph (DFG) to represent the data dependencies between operations
- To "schedule" is to select an appropriate start time for each operation in the CDFG
- Each operation could start as soon as its predecessors complete (its ASAP time)
- It must start early enough to allow time for all its successors to complete (its ALAP time)
- We can select a time between these two
  – aim to minimize the number of resources required

List Scheduling

- List Scheduling is one way of selecting start times
- Comes in two flavours
  – resource constrained (RC): find the shortest schedule satisfying resource constraints
  – latency constrained (LC): find the schedule requiring fewest resources satisfying latency constraint
- RC algorithm:
  – At each time step, select some operations which could execute (use ALAP-based “urgency”), without exceeding resource constraints
- LC algorithm:
  – Start off with one resource of each type. At each time step, if we need to increase # resources to schedule operations with zero *slack*, do so

Resource Binding

- “Binding” refers to selecting which resource should perform which operation
- Construct conflict graphs – operations are in conflict if they overlap in time
- A *colouring* of the conflict graph is thus a resource binding
- Non-hierarchical CDFGs
  – the conflict graph can be coloured in polynomial time (left-edge algorithm)
- Hierarchical CDFGs
  – colouring the conflict graph is *NP*-hard due to conditionals and multiple calls to the same function

Estimation

- We must be able to estimate the "low level" effects of our "high level" decisions
  - e.g. we have assumed in many examples that
    Silicon Area = k_1 * #mults + k_2 * #adds
- In reality, assumptions like the one above are flawed
  - doesn't account for multiplexer and controller overheads
- Circuits for which this makes little difference are *resource dominated*

Register Sharing

- If we have a large number of intermediate results, we may consider sharing registers as well as resources
- Register sharing rests on the idea of "lifetime analysis" borrowed from compilers
- Construct a register conflict graph, with edges between variables overlapping in lifetime
- Non-hierarchical CDFGs
  - register conflict graphs can be coloured using left-edge
- Hierarchicial CDFGs
  - no longer interval graphs due to "circular arc" effect of variables carried by loop
- MUX overhead of register sharing could be signficant

Control Synthesis

- The control unit decides the source of the resource inputs
  – done by controlling MUX select lines at inputs to resources
- The control unit must also decide the destination of the resource output
  – done by controlling register enable lines
- When there is register sharing, the control unit must also decide which resource output should be loaded into the register
  – done by controlling MUX select lines at inputs to registers
- Different ways of designing control units
  – microcode (may require *microcode optimization* to keep ROM small)
  – hardwired control (requires FSM design)

Graphs & Combinatorial Problems

- A clique is a complete subgraph
- The clique number of a graph is the size (# nodes) of its largest clique
- Colouring a graph is assigning a label to each nodes so that no two adjacent nodes share the same colour
- In general, minimum colouring is *NP*-hard. But for interval graphs, colouring can be solved in polynomial time
  – left edge algorithm

Algorithms & Intractability

- A problem is a general question to be answered, usually possessing several parameters, whose values are left unspecified
- An *instance* of a problem is obtained by specifying particular values for all parameters, an *algorithm* is a general step-by-step procedure for solving problems
- An algorithm is O(f(n)) if its worst case performance is bounded by k f(n) for large n
- P = set of problems solvable in polynomial time
- NP = set of problems with solutions checkable in polynomial time
- NP-complete = set of problems in NP at least as hard as any others in NP
- NP-hard = set of problems at least as hard as any in NP

Integer Linear Programming

- ILP allows us to use a formal model to find optimal solutions to problems
- A feasible solution is one which satisfies the constraints
- An optimal solution is one which both satisfies the constraints, and minimizes (maximizes) the objective function
- By casting both the objective function and the constraints as linear functions
  – we can apply standard solving tools

ILP Scheduling

- Unique start time: $$\forall v \in V : \sum_{t=ASAP_v}^{ALAP_v} x_{vt} = 1$$
- Data dependencies:

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

- Resource constraints:

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

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

ILP Scheduling

- Resource-constrained objective function:

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

- Latency-constrained objective function:

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

Module Selection

- Module selection is selecting which *type* of resource should perform each operation
- Module selection, scheduling, and binding are all inter-dependent
  - one approach is to perform them simultaneously
- Objective function:

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

- Binding constraints:

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

Module Selection

- Resource constraints:

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

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

- Dependencies:

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

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

Path Problems

- Shortest & longest path problems have many applications in synthesis
- Bellman's equations:

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

- For a directed acyclic graph (DAG)
  - Bellman's equations can be solved by applying (*) in order
- For a general graph
  - Bellman's equations are cross-coupled. Use Liao-Wong.

Summary of Summary

- This lecture has covered
  - Scheduling (ASAP/ALAP and list), binding, and estimation
  - Register sharing
  - Techniques for control synthesis
  - Graphs and combinatorial problems
  - Algorithms and intractability
  - Integer linear programming and its applications
    - ILP scheduling, module selection
  - Path problems
- There are no more formal lectures – if time we will have at least one revision lecture