High Level Synthesis

Main Tasks:

- Compile to internal form
- Compiler Optimizations
- Scheduling
- Allocation
- Binding

We've talked about these

Now we'll talk about these

Allocation and Binding:

Functional Allocation/Binding

How many functional units, how are they mapped to functional steps in the algorithm

Interconnect Allocation/Binding

How are the functional units and storage units connected: tri-state buses? Point-to-point?

Storage Allocation/Binding

How are the variables stored?

Architectural Module:

First thing you need to decide:

What sort of architecture will you create?

Need to decide things like:

What sort of registers will you use?
What sort of interconnect scheme will you use?
What sort of timing model will you assume?

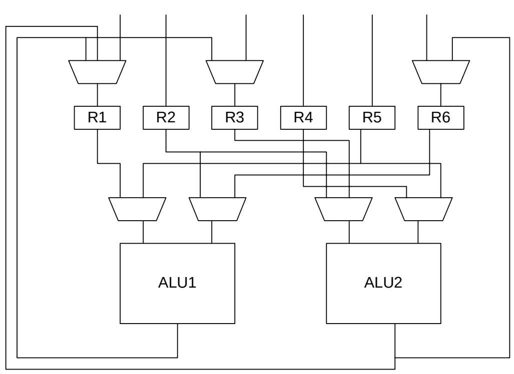
Usually a given CAD tool only targets one type of architecture (decision is not made on a circuit-by-circuit basis)

Example:

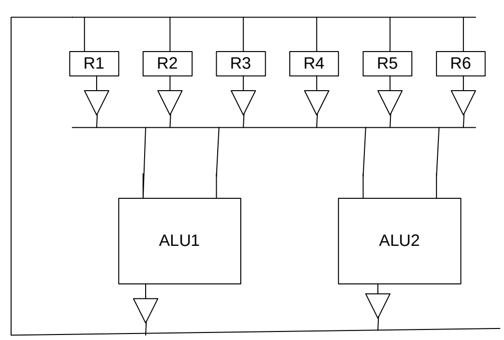
Say we want to implement the following schedule:

Cycle 1: R3 <= ALU1(R1, R2); R1<=ALU2(R3,R4)
Cycle 2: R1 <= ALU1(R5, R6); R6<=ALU2(R2,R5)
Cycle 3: R3 <= ALU1(R1, R6)

Next two slides: bus-based vs. mux-based interconnect



A) Mux-Based



B) Bus-Based

Timing Model Assumptions:

Three timing models assumed by various tools:

- Simple timing model (as shown earlier)
- Functional unit outputs registered
- Functional unit inputs registered
- Functional unit inputs and outputs registered

All but the first model allows some sort of overlap or pipelining. Need to consider this during allocation.

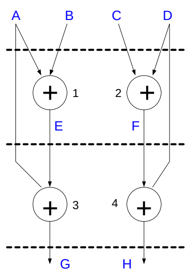
Operator Chaining:

The use of input and output registers on the functional units allows for operator chaining.

Outputs from a functional unit that are going to go to a second functional unit during the next operation, don't need to be written back to a general-purpose register.

Opportunities for chaining should be found during "compiler optimizations" and should be considered during scheduling.

Functional Unit Binding:



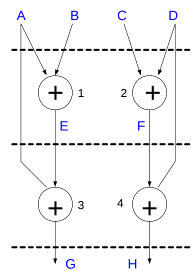
We can clearly do this with two adders

Two pairings:which is better?

a) Op 1 and 3 share adder
   Op 2 and 4 share adder

b) Op 1 and 4 share adder
   Op 2 and 3 share adder

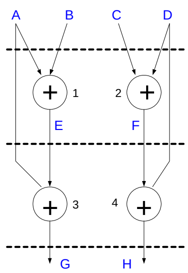
Storage Unit Binding:



Two optimizations:

- Variables with lifetimes that do not overlap may share variable (eg. maybe A and G can be the same variable)
- Variables that are not accessed simultaneously can be in same reg file

Interconnect Binding:



Wire used to transfer from B in step S1 can be also be used to transfer from E in step S2.

Wire used to transfer data to E in step 1 must be distinct from wire used to transfer to F in step 1.

The three problems are dependent:

Best solution must consider all three

But, usually, decisions are made in isolation

To come:

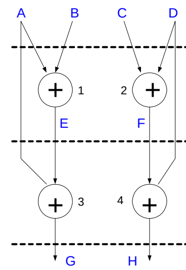
- greedy algorithm that finds very non-optimal solution considering all three at same time
- some "better" algorithms that consider one at a time

Greedy Algorithm:

Datapath = Ø
while (components left to add) {
  consider all components that can be added
  choose component that increases the “cost” of the
    datapath the least
  add it to the datapath
}

Cost of datapath: area or speed usually

Example:



Things to add to datapath:
- 4 adder functions
- 8 registers

Cost function for example =
  (# registers) 2 +
  (# adders) * 4 +
  (# muxes) * 1

Example: continued...

Start with Register (arbitrarily A):

Reg 1: A

Current datapath cost = 2

Example: continued...

Now, to find the cheapest thing to do:
- consider an adder: increase in cost = 4
- consider a new register: increase in cost = 1
- consider a variable that can share current register: increase in cost = 0 (it might mean adding a mux later, but for now, we don’t know)

So, choose a variable that can share current register, say g

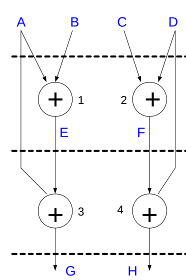
Example: continued...

The new datapath looks like this now:

Reg 1: A, G

Current datapath cost = 2

Example continued:

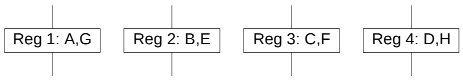


No other variables can share current register, so next step is to:
- add an adder
- add a new register

Adding a new register is cheaper, so add register for variable B

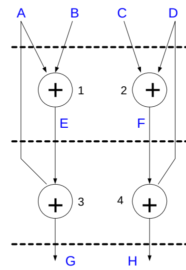
Example: continued...

After 6 more steps:



Current datapath cost = 8

Example Continued:

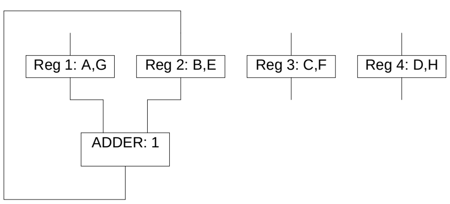


All we have left are the four adder functions.

Each one incurs a cost of 4, so arbitrarily choose Adder 1

Example: continued...

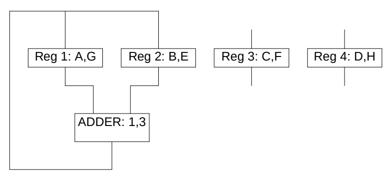
After we add the adder (note: no mux needed yet)



Current datapath cost = 12

Example: continued...

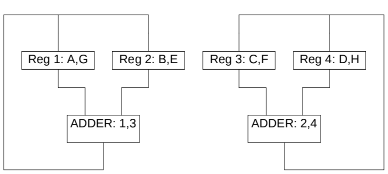
Adder three can be added for free, since it can be shared with the first adder



Current datapath cost = 12

Example: continued...

After two more steps, we are done:



Current datapath cost = 14

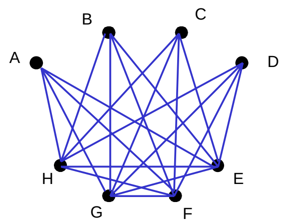
A better algorithm:

Consider solving the storage allocation problem in isolation:

Determine the lifetime of each variable, and construct a graph G(V,E):

- V: each node is one variable
- E: an edge exists between two nodes if the corresponding variables are not used in any control step (ie. the lifetimes do not overlap)

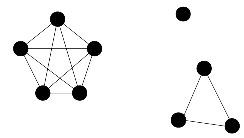
Our Previous Example:



A better algorithm for storage allocation:

Definition:

Clique: A clique of G(V,E) is a subset of nodes V' ⊆ V such that all nodes in V' are connected to each other

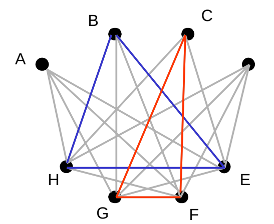


A Better Algorithm for Storage Allocation:

Algorithm: Given the graph, partition it into maximum-sized cliques. Each clique represents variables that can be assigned to the same register.

The clique partitioning problem is a well-known graph theory problem that is NP-Complete.

Our Previous Example:

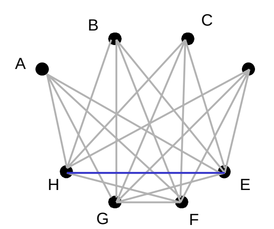


B, H, and E can be implemented using 1 register
C, G, and F can be implemented using 1 register

Heuristic to find Clique Partitioning:

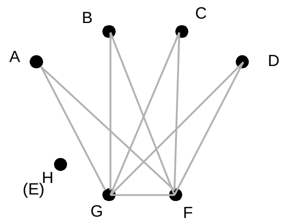
while there are edges left in the graph {
    select two nodes *i* and *j* such that:
        i and *j* are connected by an edge
        they have the maximum number of common neighbours
    merge *i* and *j* into one "super-node" *k*
    delete all edges to *i* and *j* (now *k*)
    add back in edges that go to a common neighbour
}

Our Previous Example:



E and H share 4 common neighbours

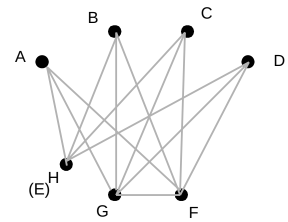
Our Previous Example:



Merge E and H, and delete all edges to E and H

Page 31

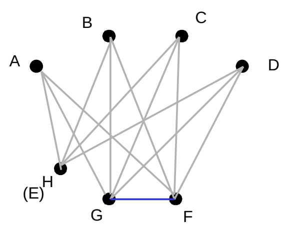
Our Previous Example:



Add Back Edges that go to common neighbour

Page 32

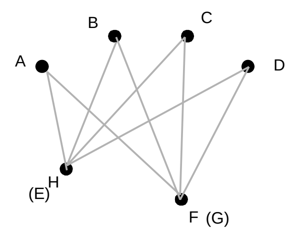
Our Previous Example:



Repeat: E and F share 4 common neighbours

Page 33

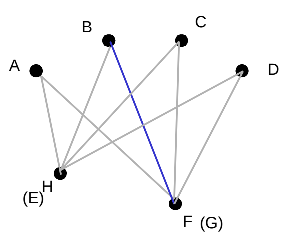
Our Previous Example:



Merge G and F (into F)

Page 34

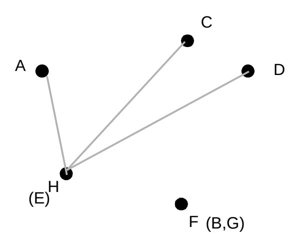
Our Previous Example:



Repeat: B and F share no common neighbours
(but none do, at this point)

Page 35

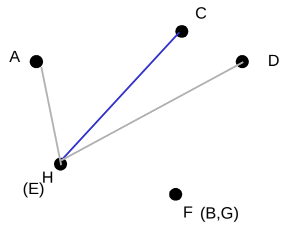
Our Previous Example:



Merge B into F (note: we delete all edges that do
not go to common neighbours)

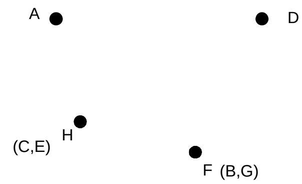
Page 36

Our Previous Example:



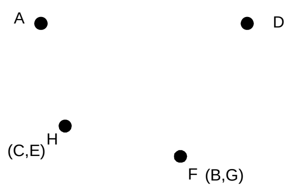
Repeat: Choose C and H

Our Previous Example:



Merge C into H (note: delete all edges that do not go to common neighbours)

Our Previous Example:



So our allocation will contain 4 variables

Our Previous Example:

So our allocation will contain 4 variables:

Variable 1: A

Variable 2: D

Variable 3: C, E, H

Variable 4: B, F, G

Clique Partitioning

Can use this same algorithm for:

Functional Unit Allocation: each node is an operation. An edge between two nodes exists if

- both operations can be done at the same time
- both can be done using the same physical unit

Interconnect Allocation: each node is a data transfer, An edge between two nodes exists if the two data transfers are not done at the same time (meaning they can share an interconnect path).

Iterative methods:

One approach:
- find a solution using any of these methods
- try swapping greedily to get a better solution

Probabilistic Methods: simulated annealing

Future work in this area:

Should integrate these algorithms with a scheduler (maybe have a “fast allocator” that can be used during the scheduling operation)

Better cost functions: we can take interconnect into account, but maybe we could do a better job of it

Better architecture models: a good designer can always beat a synthesis tool today, since he/she can choose an appropriate architecture model

Page 43