1 Abstract Models

1.1 Structure

At it's highest level, a *structure* is an interconnection of various components.

Formally, we define an *incidence structure* as an abstract system consisting of two types of objects, and a single relationship between these types of objects. There are man

There are different kinds of incidence structures corresponding to different ways to model something.

1.1.1 Modules/Pins + Nets + Incidence relation

A natural way to model hardware is via *modules* and *nets*, with their interconnections described by *incidence relations*.

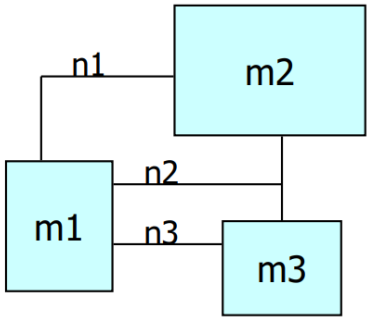


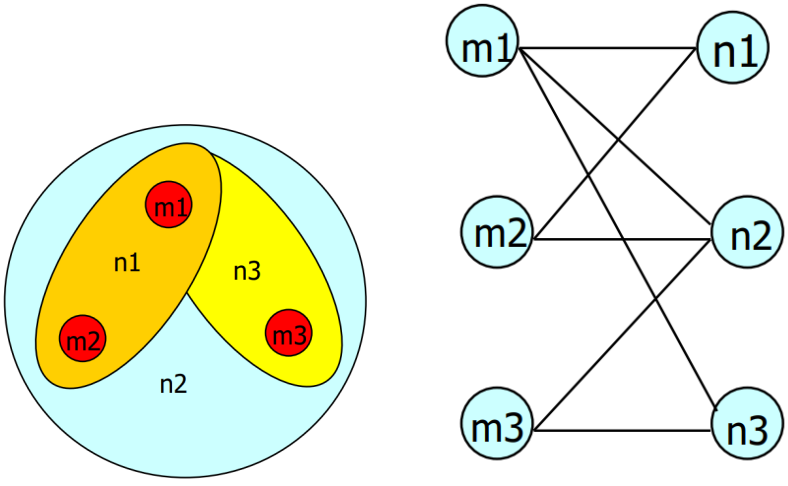
Figure 1: Modules, nets and pins

Seen in Fig1 above, we term net n_1 as being incident on modules m_1 and m_2

1.1.2 Hypergraph, Bipartite Graph

Another way to model is to use graphs, where *vertices* correspond to modules and *edges* correspond to nets.

When a particular net n_i is incident on a module m_j, there is an associated edge e_k connecting n_i and m_j, where each of these edges denotes an incidence relation.



(a) Hypergraph (b) Bipartite graph

Figure 2: Graph-based modelling

- Hypergraph is a generalization of a graph, where any edge can join **any** number of vertices
- Bipartite graph is a graph where every edge connects a vertex in U to one in V
  - Notice that modules $\{m_1, m_2, m_3\}$ are placed on one side, nets $\{n_1, n_2, n_3\}$ placed on the other side. Therefore modules **cannot** be directly connected to other modules; they required a net interconnection. Similarly nets **cannot** be directly connected to other nets; if they were, it would just be considered the same net!

1.1.3 Incidence Matrix, Netlists

From the discussion above, we notice the idea of incidence relations. A natural way of representing incidence relations is to use *incidence matrices*, as seen in Fig3 below.

	n1	n2	n3
m1	1	1	1
m2	1	1	0
m3	0	1	1

Figure 3: Incidence Matrix

In practical applications, incidence matrices tend to be *sparse*. Therefore a more efficient representation is to downgrade from *matrix* to a net*list*. Accoridngly we have two types of netlists, ...

- Net-oriented; Enumerating all modules of a given net
- Module-oriented; Enumerating all nets of a given module
  - eg. $m1 : n1, n2, n3$
    $m2 : n1, n2$
    $m3 : n2, n3$

1.1.4 Hierarchy

There is the concept of *hierarchy* within a structure, where submodules may be connected together to create a larger module.

- *Leaf* modules are *primitives* (ie. Module that does not have any submodules 'within it')
  - Usually primitives are used to describe something that is already available in the cell library
- *Non-leaf* modules are composed from a set of modules (termed it's *submodules*)

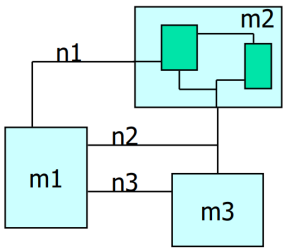
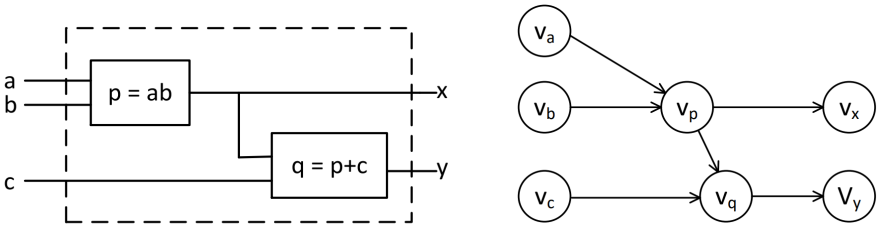


Figure 4: m_1 and m_3 are leaf modules
m_2 is non-leaf module (composed of it's submodules)

1.2 Logic Network

Logic networks are defined to be *blocks* interconnected with **directional** nets

- Each *block* is modelled by a boolean function (ie. combinational logic)
- Since each block is implementing combinational logic, a logic network is purely **combinational**
- More specifically, each block is a *multi-input, single-output* **leaf** module
- In the special case where the blocks correspond to cell library elements, we term it a *mapped network*



(a) Logic Network (b) Logic Network graph

Figure 5: Different ways of representing Logic Networks

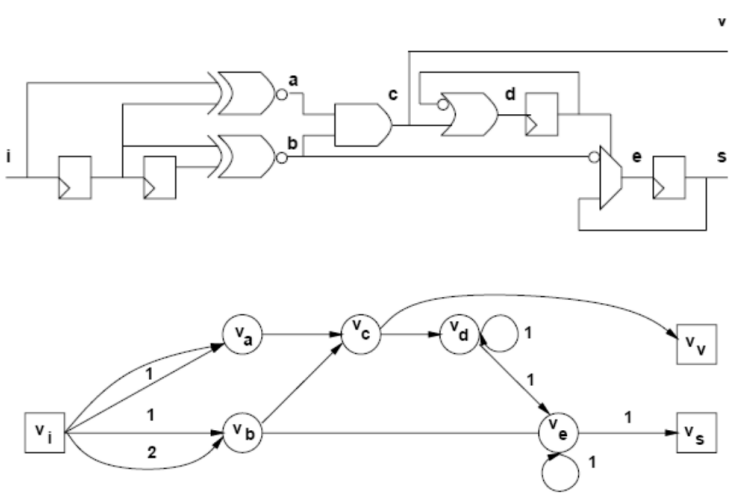
Logic networks have structural and behavioral semantics

- Structural semantics can be induced by the interconnection of blocks
- Behavioral semantics can be extracted from the boolean expression of each block

1.2.1 Synchronous Logic Network

Extends the concept of logic networks, to include synchronous elements.

To do that, we allow the edges to have weights (denoting delays).



$$
\begin{aligned}
a^{(n)} &= i^{(n)} \overline{\oplus}\, i^{(n-1)} \\
b^{(n)} &= i^{(n-1)} \overline{\oplus}\, i^{(n-2)} \\
c^{(n)} &= a^{(n)} b^{(n)} \\
d^{(n)} &= c^{(n)} + d'^{(n-1)} \\
e^{(n)} &= d^{(n)} e^{(n-1)} + d'^{(n)} b'^{(n)} \\
v^{(n)} &= c^{(n)} \\
s^{(n)} &= e^{(n-1)}
\end{aligned}
$$

$$
\begin{aligned}
a &= i \,\overline{\oplus}\, i@1 \\
b &= i@1 \,\overline{\oplus}\, i@2 \\
c &= a\ b \\
d &= c + d@1' \\
e &= d\ e@1 + d'\ b' \\
v &= c \\
s &= e@1
\end{aligned}
$$

Figure 6: Synchronous Logic Network (as seen in L3)

1.3 Finite-State Machine

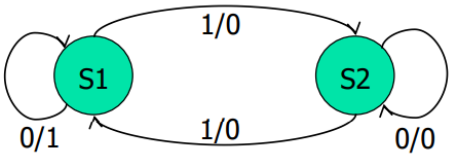
A **purely behavioral** model.

Can be described by ...

- A set of primary input patterns X
- A set of primary output patterns Y
- A set of states S
- Two functions ...
  1. State transition function; $\delta : X \times S \to Y$
  2. Output function (Mealy); $\lambda : X \times S \to Y$
  3. Output function (Moore); $\lambda : S \to Y$

State	Inp	N-State	Outp
S_1	0	S_1	1
S_1	1	S_2	0
S_2	0	S_2	0
S_2	1	S_1	0

(a) State table



(b) State diagram

Figure 7: Different ways of representing FSMs

1.4 Dataflow Graph (DFG)

A **purely behavioral** model, used to represent *data-paths*.

Vertices represent operations, edges represent dependencies

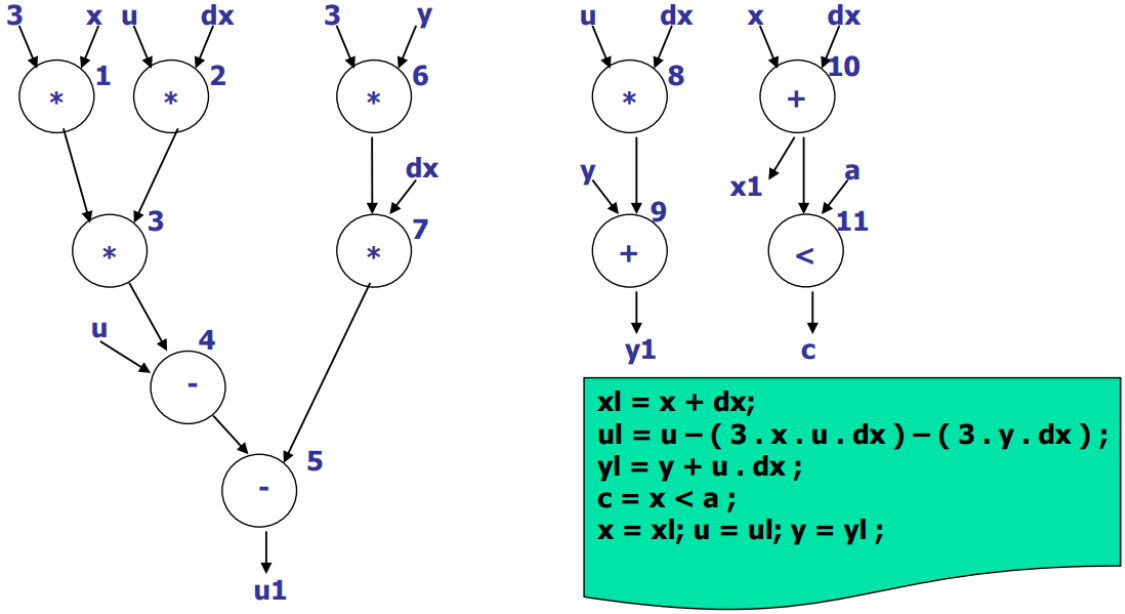


Figure 8: DFG

DFG is also useful to see where registers might need to be inserted (since the results of operating on data need to be saved into memory eventually)

However, note that DFGs only say what the operations are and what the dependencies are. It does not give us *control information* (eg. some operations might only be done on a conditional basis).

1.5 Sequencing Graph (CDFG)

A **purely behavioral** model, used to represent *data-paths* **and** *control-paths* (ie. enchanced version of DFG).

Note that oftentimes the CDFG is shown without data annotations, especially when doing *scheduling* (when to perform operation) or *binding* (where to perform operation).

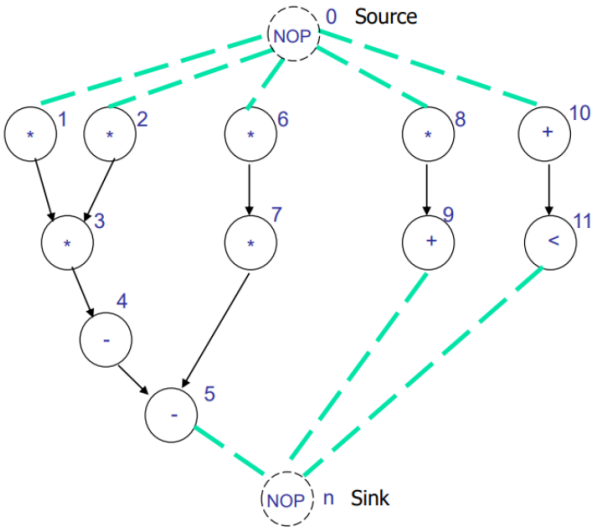


Figure 9: CDFG. The *NOPs* mean that the hardware isn't doing anything before/after the operations

CDFG's can furthur show us ...

- Hierarchy
- Control-flow commands (e.g *branching*, *iteration*)

1.5.1 Sequencing Graphs: Hierarchy

What is "inside" a vertex?

In a CDFG, vertices may contain other 'sub-vertices' (ie. operations) within them. Therefore, it performs simiar to a function call in programming.

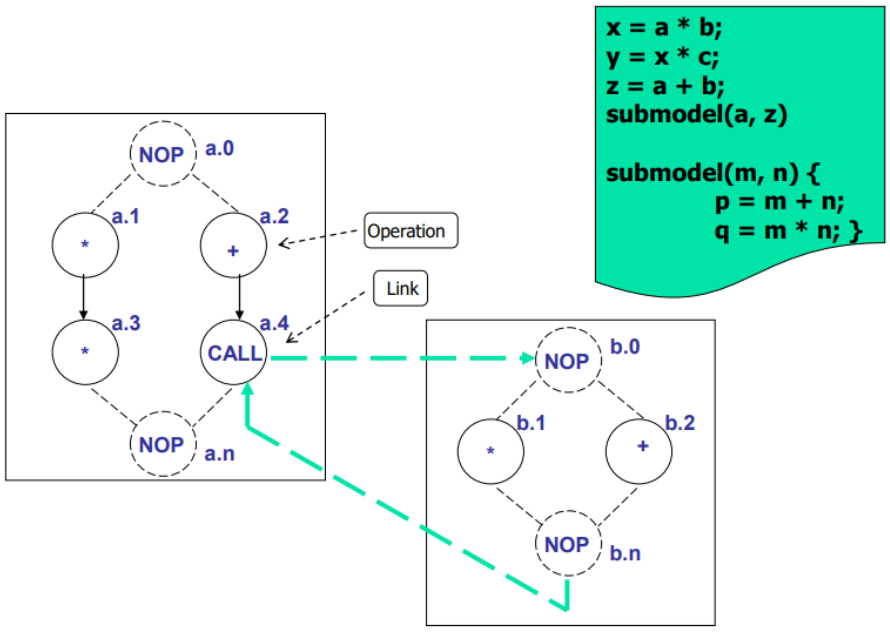
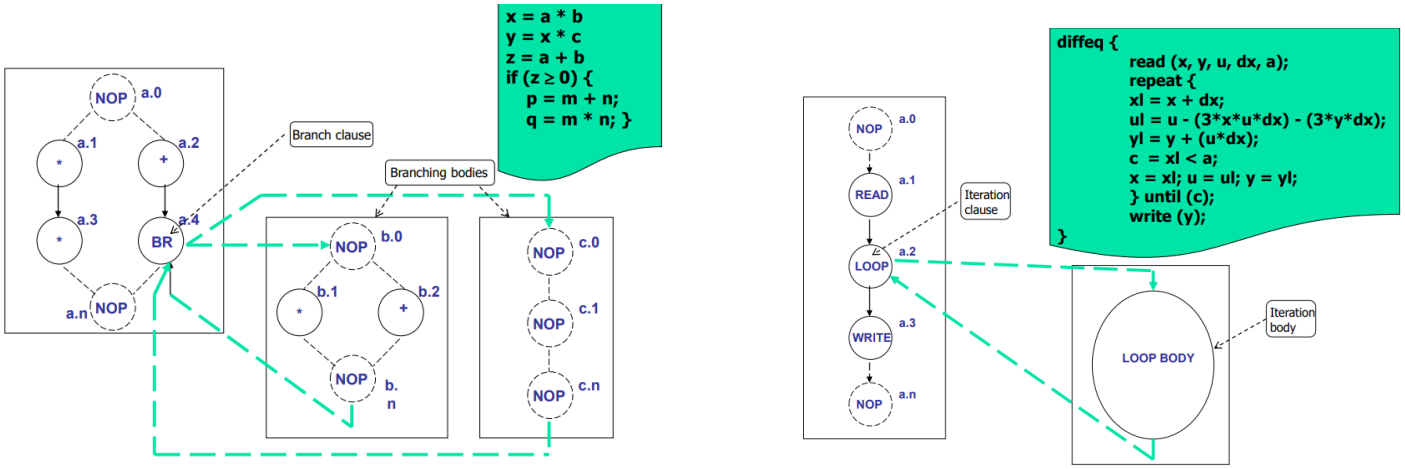


Figure 10: "Function calls" in a CDFG

1.5.2 Sequencing Graphs: Control Information



(a) Branching

(b) Iteration

Figure 11: Control information in a CDFG

1.5.3 Sequencing Graphs: Semantics

What can a CDFG tell us?

Vertices (representing operations) can be in either of the three states ...

1. *Waiting* for execution
2. *Executing*
3. *Completed* execution

From the CDFG's POV, a vertex can be executed as soon as all of it's **immediate** predecessors have completed execution (we term this *precedence constraints*). However, is this really true?

- CDFG's are only concerned with the behavioral aspects, there is no notion of structural aspects (availability of hardware to actually execute the operation)

1.5.4 Vertex attributes

What can the vertices of a CDFG tell us?

- Area cost
  - eg. A multiplication vertex '*' has an associated hardware area cost.

    Note that the hardware cost may not be fixed (eg. Multiplication units may be implemented combinationally or sequentially, which have different area costs)
- Delay cost, furthur categorized as ...
  - Propagation delay (denoted in seconds); Corresponding to *combinational* delays
    - Combinational logic has no concept of 'cycles', everything is done within one 'cycle'. Therefore the concept of 'execution cycles' is not applicable to combinational logic.
  - Execution delay (denoted in *number of cycles*); Corresponding to textitsequential delays
    - Execution delays may be *data-dependent* (eg. runtime of booth's multiplication algorithm is data-dependent)
    - Execution delays may be *bounded* (eg. branching) or *unbounded* (eg. while-loops where exit condition of the loop is itself modified by the loop)

1.5.5 CDFG estimates

What area and delay estimates can we make from a CDFG?

Let's use the CDFG from Fig9 as an example (copied here for convenience)

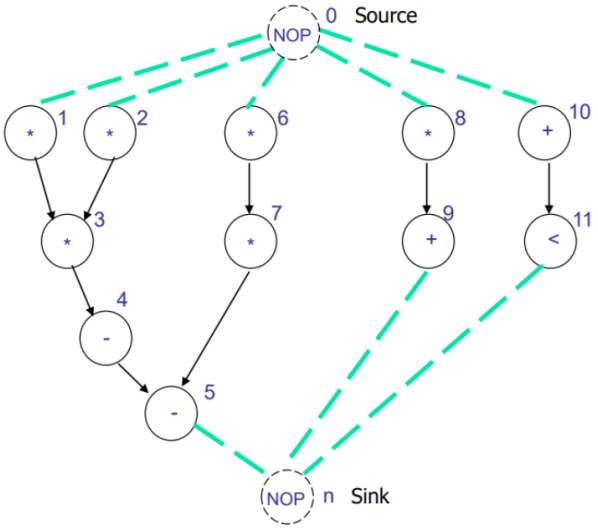


Figure 12: CDFG graph. Assume that area and delay are determined entirely by the ALUs and MULs

- Area estimates; Only the **upper bound** is meaningful, lower bound is simply (1 ALU, 1 MUL)
  - Upper bound is st. each operation has dedicated hardware (ie. no sharing of hardware resources, termed *dedicated binding*). Therefore, upper bound (ie. maximal area) is 6 MULs, 5 ALUs
- Delay estimates
  - Lower bounded by the longest path from *source* to *sink* **in terms of delay**.
    - Note that this is not to be confused with critical path delay of a combinational circuit (vertices of a CDFG are usually not connected back-to-back in a combinational fashion)
    - Furthur note that the delay is lower bounded when the area is upper bounded.
  - Upper bounded (meaningfully) by the scenario where only 1 ALU is available, 1 MLU is available (a meaningless delay estimate is ∞, where no hardware units arae available).
    - If we are upper bounded by only having 1 functional unit of each type, can we look at the CDFG and derive the optimal delay? This problem is actually the *scheduling problem*, and is NP-hard.

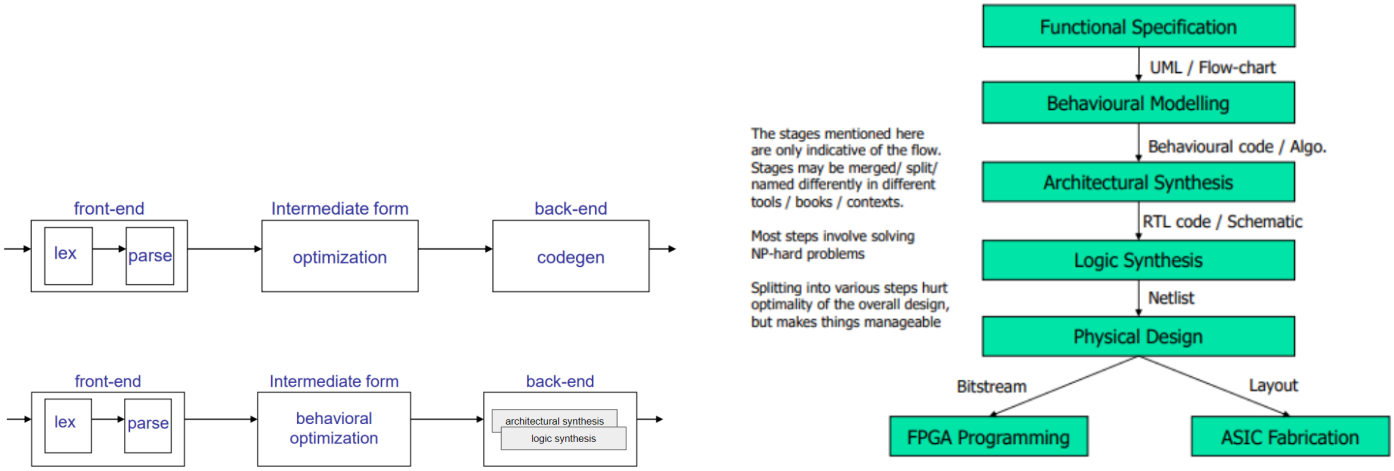
2 Behavioral Synthesis (High-Level Synthesis)

Synthesizing *behavioral* (not necessarily HDL) code rather than RTL code.

'Synthesis' in this context is not the same as Vivado's 'synthesis'.

2.1 Behavioral Synthesis Steps

Note that it shares many similarities with software compilation



(a) Software compilation vs Hardware synthesis

(b) Simplified FPGA/ASIC design flow

Figure 13

Software compilation ...

1. Compile program into machine form
2. Optimize intermediate form
3. Generate target code for architecture

Hardware compilation ...

1. Elaborate HDL model into sequencing graph
2. *Behavioral-level optimization*; Optimize sequencing graph
   - Optimize abstract models **independently** from *implementation* parameters
3. *Architectural synthesis* and *optimization*; **Macroscopic** level
   - Converts a *behavioral* model to a macroscopic *structural* model
   - Utilizes macroscopic building blocks (Adders, Registers, ...)
   - Involves *scheduling* (delay considerations) and *binding* (area considerations)
   - Transcompiles optimized TLM to RTL (described using HDL) design
4. *Logic synthesis*; **Microscopic level**
   - Turns RTL design into design implementation (in terms of the technology library's building blocks)
     - For ASICs, building blocks are usually gates
     - For FPGAs, building blocks are CLBs (which can offer higher level of functionality compared to ASICs)
   - Apply furthur constraints (eg. Logical port to physical pin mapping)

2.2 Front-end (Lexers and Parsers)

2.2.1 Lexers (Lexical analysis)

Conversion of text into meaningful *lexical* tokens

2.2.2 Parser (Syntactic analysis)

Generates a *parse tree* from lexical tokens

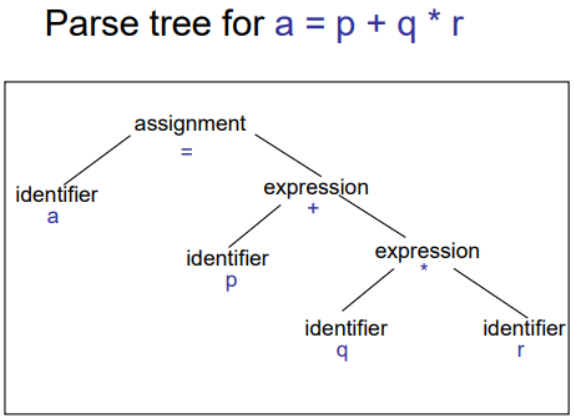


Figure 14: Parse tree

2.2.3 Semantic Analysis

Checks if your parse tree makes *semantic* sense

- Data-flow and control-flow analysis
- Type checking
- Resolving and checking arithmetic and relational operators
- ...

2.3 Behavioral-level optimizations

Semantic-preserving transformations aimed at simplifying the model, Applied to parse-trees during or after their generation.

Divided into *data-flow* based transformations and *control-flow* based transformations ...

2.3.1 Some data-flow based transformations

- Tree-height reduction
  - Applied to arithmetic operations only
  - Goal is to balance the parse tree, in order to exploit hardware parallelism

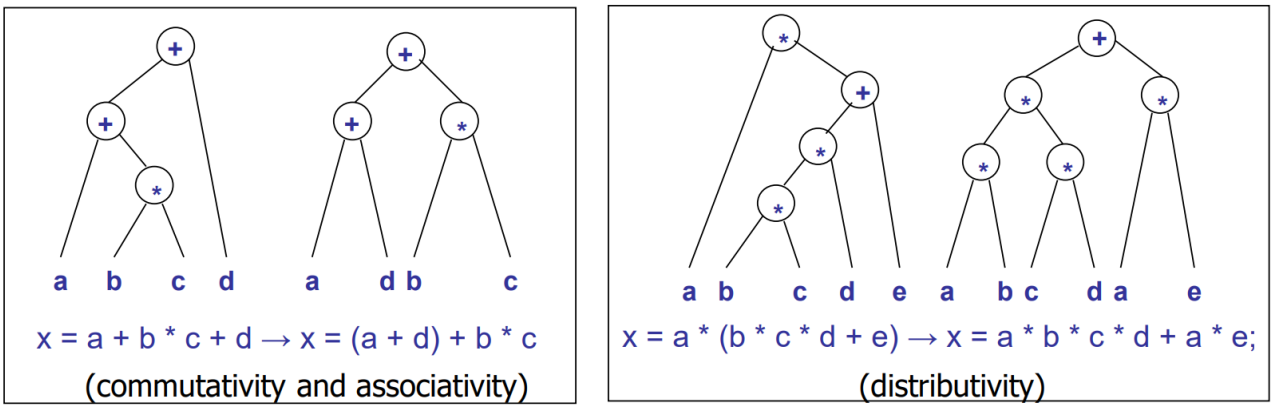


Figure 15: Tree-height reduction

- Propagation

```
a = 0; b = a + 1; c = 2 * b;
a = 0; b = 1; c = 2;
```

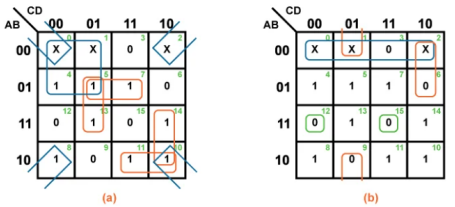
(a) *Constant* propagation; a can be evaluated at compile time (instead of runtime)

```
a = x; b = a + 1; c = 2 * x;
a = x; b = x + 1; c = 2 * x;
```

(b) *Variable* propagation; a and b can be evaluated **simultaneously** now

Figure 16: Different propagation techniques

- Sub-expression elimination



(a) *Logic* expressions can be simplified with K-Maps

```
a = x + y; b = a +1; c = x + y
a = x + y; b = a + 1; c = a;
```

(b) *Arithmetic* expressions can be simplified (like memoization)

Figure 17: Different sub-expression eliminations

- Operator-strength reduction
  - Complexity of operations is . . . $\xrightarrow[\text{Increasing level of complexity}]{\text{Assignment; Bitwise logical operations(AND,OR,...); Variable shift/Arithmetic operations; Mult; Div}}$

```
a = x^2,  b = 3 * x;
a = x * x;  t = x << 1; b = x + t;
```

Figure 18: Use 'weaker' operations if possible

  - Intuitively, the degree of complexity has to do with how much can be done in parallel
    - Suppose we perform bitwise AND between two 8bit inputs. The operations on each bit can be done **independently**, which makes it less 'complex'

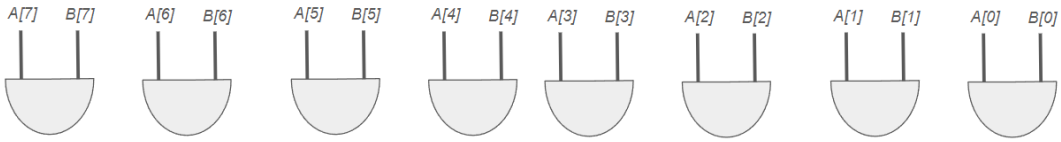


Figure 19: AND operation, done in parallel

    - Suppose we are performing addition. The operations cannot be done independently, there is data dependency on the carry bit. This makes it more 'complex'

$$\begin{array}{r} 1\ 1\ 1 \\ +\ 0\ 1\ 1 \\ \hline 1\ 0\ 1\ 0 \end{array}$$

Figure 20: Addition operation. Note that we must wait for the carry bit to propagate through

2.3.2 Some control-flow based transformations

- Modal expansion (ie. Inline function)
  - *Flattens* the hierarchy, removing module boundaries and encouraging cross-module optimizations (that would not have been obvious before)
  - Removes the need for *Program Counter(PC)* to jump abruptly, avoiding control hazards and etc ...

x = a + b; y = a * b; z = foo (x , y);
foo(p,q) { t=q - p; return (t); }

By expanding foo
x = a + b; y = a*b; z = y – x;

Figure 21: Modal expansion

- Conditional expansion
  - Very useful for **logical** expressions, removes the need to evaluate condition during runtime
  - Note that it may preclude hardware sharing

y = ab; if (a) {x = b + d; } else { x = bd; }
- Can be expanded to : x = a (b + d) + a'bd
- And simplified as : y = ab; x = y + d (a + b)

Figure 22: Conditional expansion. Note that this is a boolean function, not an arithmetic function

- Loop expansion
  - Applicable to loops with data-**independent** exit conditions

for (I = 0; I < 3; I ++) { a[I] = b[I] + 1; }

Expanded to
a[0] = b[0] + 1; a[1] = b[1] + 1; a[2] = b[2] + 1;

Figure 23: Loop unrolling

2.4 Architectural Synthesis

Implementation of *structural* design from (behaviorally optimized) *behavioral* CDFGs.

More formally, we can *synthesize* a **macroscopic** structure because we have ...

1. Circuit behavior (Sequencing graph)
2. Cells ('Building blocks') from technology library, fully characterized in terms of area and execution delay
3. Timing and area/resource usage constraints; Which will be utilized to do scheduling and binding

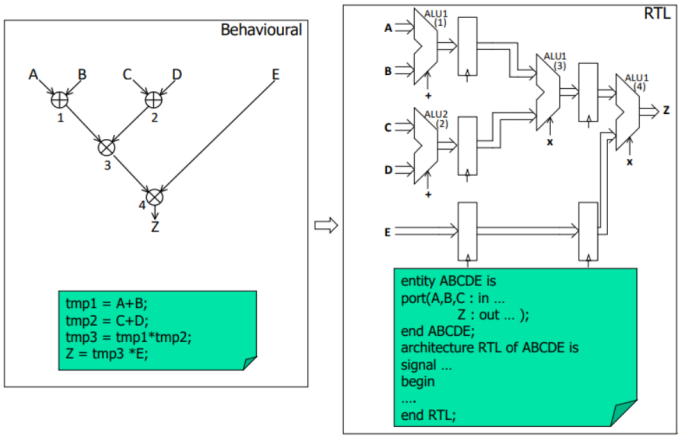


Figure 24: Behavioral to Structural

2.4.1 Technology library cells

Can be broadly split into *data-path* and *control-unit* cells

Data-path cells consist of ...

- Arithmetic and logic blocks
  - Functional units that perform operations on data
- Memory and registers
  - Storage resources to store data
- External buses and ports
  - Interconnection of different blocks together
- Muxes and **internal** buses
  - Steering logic to appropriately channel data **within** a block

Control-unit cells need to control the flow of data **within** the datapath, utilizing ...

- Mux *selects*
- Register write-*enable*s
- Functional unit *activation/enable*, and operation *selection* signals

2.4.2 Implementation constraints

Can be broadly split into *timing* constraints and *resource* constraints

Timing constraints could come in the form of ...

- Cycle-time
  - What is the frequency that your circuit should operate at?
    - If we have single-cycle operations, then the logic must have it's result by the next clock edge
    - Multi-cycle operations (eg. pipelined computations) relax the above constraint
- Latency between a set of operations (ie. Set of operations must complete within 'x' amount of cycles)
- Time spacing between operation pairs (ie. Operations must be spaced 'x' amount of cycles apart)

Resource constraints could come in the form of ...

- Allocation
  - Only given limited number of functional units to achieve some functionality
- Partial binding (ie. some operations are unalterably tied to some functional units)

2.4.3 Scheduling and Binding - Overview

In architectural synthesis, we have an area/performance trade-off.

An optimal implementation will maximize performance subject to area constraints (or vice versa). But how do we achieve this optimal implementation?

This is done via *scheduling* and *binding*. Place vertices of CDFG (representing operand-operations) in ...

- Time (Scheduling); Determine which operand-operation occurs during each clock cycle
- Space (Binding); Determine which hardware resource should be utilized
  - Map operand-operations to functional units (*Function binding*)
  - Map variables to storage units (*Storage binding*)
  - Map data transfers to buses (*Connection binding*)

2.4.4 Scheduling

When doing scheduling (ie. Synthesis in the *temporal* domain) on a CDFG graph, the ...

- Edges denote the *latency* of the vertex (we assume no edge label means latency of 1)
- *Granularity* is in terms of *cycles*, not seconds

Scheduling associates a start-time for each operation (vertex) in the sequencing graph. From this, we can determine the *latency* and *parallelism* of the implementation.

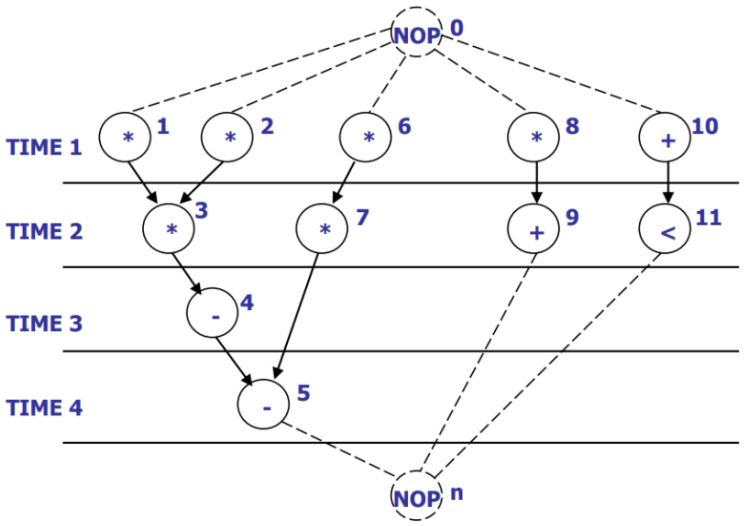


Figure 25: Sequencing graph that has been scheduled

2.4.4.1 Mathematical model of scheduling

Let's describe scheduling mathematically.

1. Define our graph objects ...
   - Vertices $V = \{v_0, v_1, \ldots, v_n\}$; v_0 is source node, v_n is sink node
   - Node latencies $M = \{\mu_0, \mu_1, \ldots, \mu_n\}$ st. $\mu_i \in Z$ (Node latency cannot be negative)
   - Operation start times $T = \{\tau_0, \tau_1, \ldots, \tau_n\}$ st. $\tau_i \in Z^+$ (ie. Begin counting from 1)
2. Then, define the function $\varphi : V \to Z^+$ as $\varphi(v_i) = \tau_i$ st ...
   - τ_i (denoting the *operation start time*) obeys $\tau_i \geq \tau_j + \mu_j$; $\forall (v_j, v_i) \in$ *valid edge*
   - This is just saying that an operation can only start when it's precedence constraint is fufilled (ie. Operators before it have completed their computation and are ready with results)
3. **Overall** latency $\mu_{overall} = \tau_n - \tau_0$;
4. Scheduling is the task of optimally determining T, subject to precedence constraints specified in the sequencing graph

2.4.4.2 Different kinds of scheduling

- ASAP scheduling (Optimize for the lowest $\mu_{overall}$, not factoring in functional unit availability)
  - Unconstrained hardware availability (We have infinite functional units)
    - Operation is only delayed by precedence constraints, not by hardware availability
  - Maximally constrained hardware availability (We have only one of each type of functional unit)
    - We are still optimizing for lowest $\mu_{overall}$. This is actually an NP-hard combinatorial optimization problem.

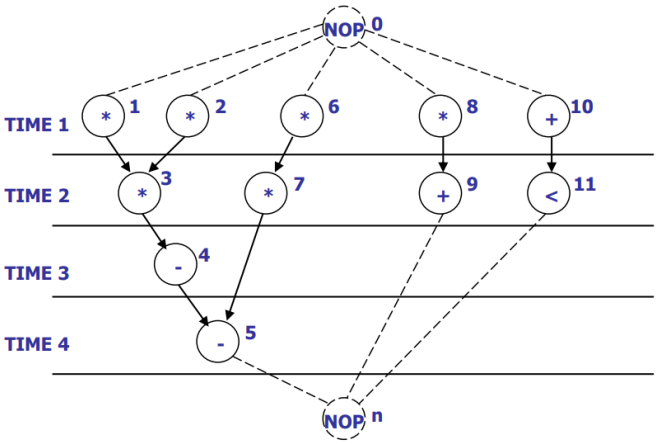


Figure 26: Unconstrained
$\mu_{overall} = 5 - 1 = 4$
Minimal resource usage: x4 MUL units, x2 ALU units

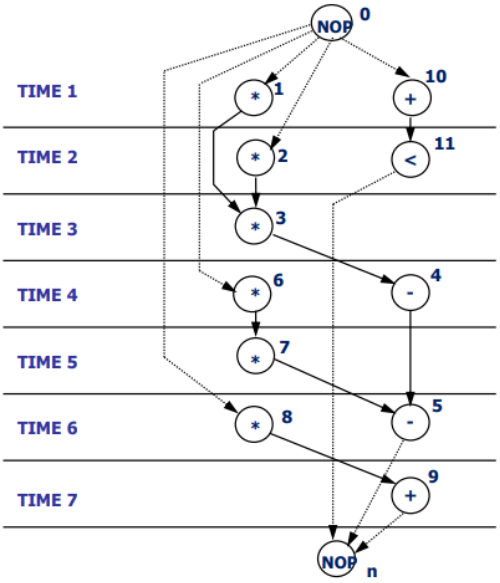


Figure 27: Maximally constrained
$\mu_{overall} = 8 - 1 = 7$

- ALAP scheduling
  - Scheduling under unconstrained-ASAP is equivalent to finding longest path between each vertex and **source** node.

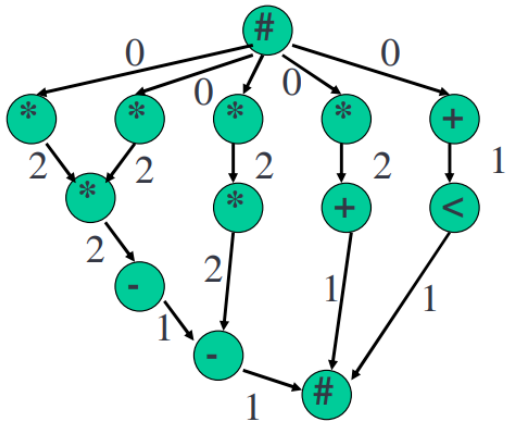


Figure 28: Edge-weighted CDFG

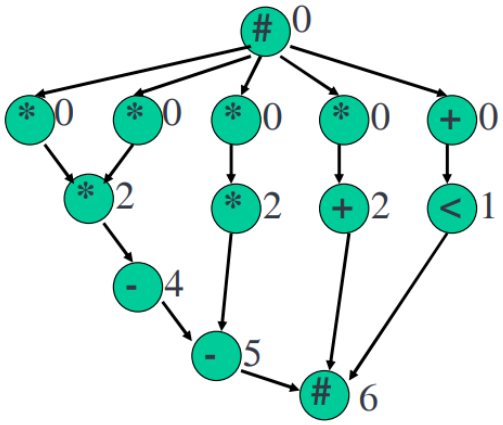


Figure 29: Applying longest path algorithm leads to ASAP start-times

  - Unconstrained-ALAP is then the opposite; We schedule operations at the latest opportunity. This can be performed by seeking the longest path between each vertex and **sink** node, then subtracting longest path time from desired $\mu_{overall}$

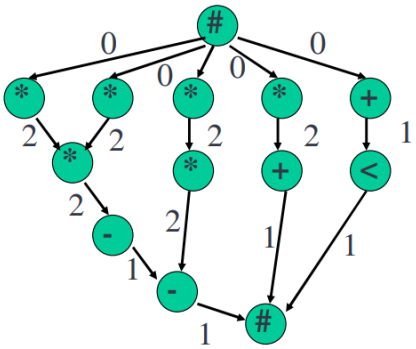


Figure 30: Edge-weighted CDFG

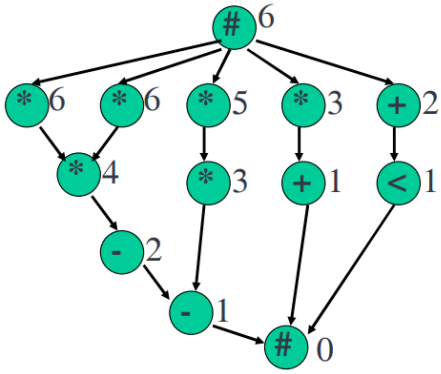


Figure 31: Longest path to sink node

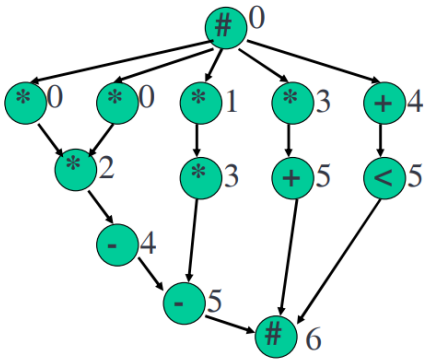


Figure 32: Supposing $\mu_{overall} = 6$

  - The difference between ALAP and ASAP time for a **vertex** is termed the *operation mobility* or *slack*.

    Mobility measures how free we are to move vertices into different time-slots. Operations with zero mobility are *critical* operations, and together they form the *critical path*, which determines how fast our circuit can run.

- Scheduling with chaining
  - What if we allow two or more **combinational** operations to execute in the same cycle?
  - In the example below, suppose ...
    - Mult: 35s
    - Others: 25ns
    - Cycle time: 50ns
  - Then, we can squeeze x2 ALU units back-to-back into a single clock cycle
  - Note that the latency has decreased, but now we require x2 ALUs minimally
  - Note also that there is a cycle-time and latency tradeoff. Compared to Fig25, we notice that Fig33 has a longer cycle-time (but shorter latency)

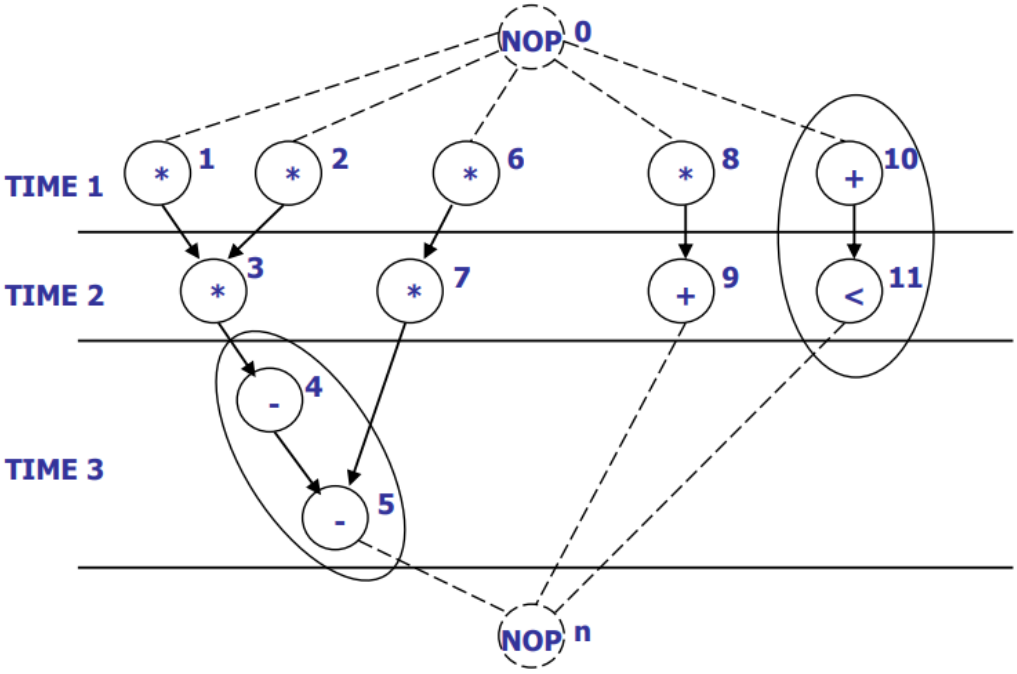


Figure 33: Chained operations

2.4.5 Binding

When doing binding (ie. Synthesis in the *spatial* domain) on a **scheduled** CDFG graph, we ...

- Associate a resource (described as a **2-tuple** (*type*, *instance*) annotation beside the vertex) with each operation. From this, we can determine the *area* of implementation.
  - *Type* denotes the type of functional unit (eg. ALU, Adder, MUL, ...)
  - *Instance* denotes the instance of a **specific type** of functional unit. (eg. ALU1, ALU2, ...)
- Operations bound to the same resource, belong to the same hypergraph.

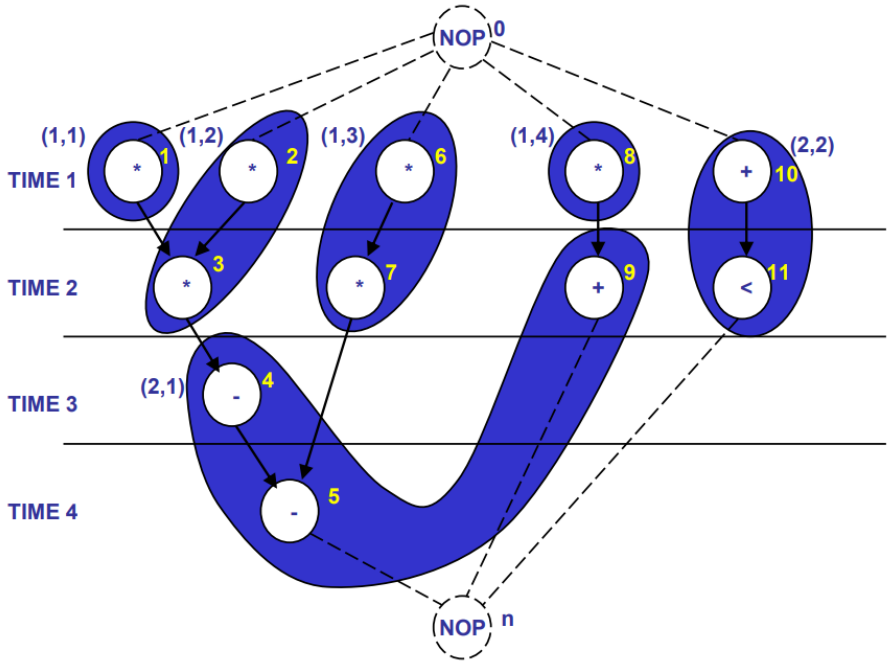


Figure 34: Binding a scheduled sequencing graph

The key point to notice is that when we bind a resource to more than one operation, the operations must not execute concurrently. Intuitvely this makes sense, since the same hardware cannot be shared amongst two operations in the same clock cycle.

2.4.5.1 Mathematical model of Binding

Let's describe binding mathematically.

1. Define our graph objects ...
   - Vertices $V = \{v_1, v_2, \ldots, v_n\}$; v_0 is source node, v_n is sink node
   - Types $Q = \{q_1, q_2, \ldots, q_{num\_types}\}$
   - Instances (**of a specific type** q_i) $R = \{r_1, r_2, \ldots, r_{num\_instances}\}$ st. $r_i \in Z^+$
2. Define *resource binding* **function** $\beta : V \rightarrow Q \times Z^+$ as $\beta(v_i) = (q_i, r_i)$
   - Plainly, $\beta(v_i) = (q_i, r_i)$ means an operation (corresponding to v_i) is implemented by the r_i^{th} instance of functional unit type q_i
   - β is *one-to-one* $\longrightarrow$ We have dedicated binding
     - One instance of one functional unit type, implements only one vertex
   - β is *many-to-one* $\longrightarrow$ Resource sharing **is** happening
     - Multiple vertices are mapped to the same resource (ie. same instance of same functional unit type)
3. Define *implementation* **relation** $\theta : V \rightarrow Q$ as $\theta(v_i) = q_i$ st. $q_i \in Q$
   - Plainly, $\theta(v_i) = q_i$ means that the i^{th} operation is implemented by functional unit type q_i
   - θ is *one-to-many* $\longrightarrow$ We have a choice of *which* functional unit type we wish to utilize
     - eg. If v_i is an addition operation, it could be implemented by ALU or Adder functional unit

- θ is *many-to-one* $\longrightarrow$ Multiple operations are mapped to the same functional unit type, resource sharing is **possible** (not **confirmed**)
  - Suppose that operations v_i and v_j do not overlap. Then they can both use the same instance of the same functional unit type q_k, which is resource sharing.
  - Suppose that operations v_i and v_j overlap. Then they must use different instances of the same functional unit type q_k, meaning resource sharing is not possible

2.4.5.2 Example of binding

$\beta(v_1)$	(1,1)
$\beta(v_2)$	(1,2)
$\beta(v_3)$	(1,3)
$\beta(v_4)$	(2,1)
$\beta(v_5)$	(2,2)
$\beta(v_6)$	(1,4)
$\beta(v_7)$	(1,5)
$\beta(v_8)$	(1,6)
$\beta(v_9)$	(2,3)
$\beta(v_{10})$	(2,4)
$\beta(v_{11})$	(2,5)

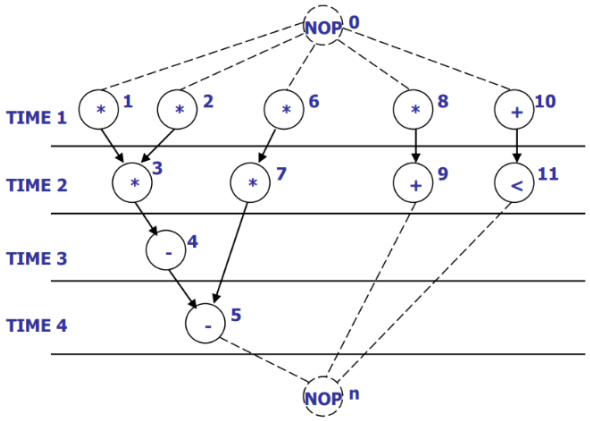


Figure 35: Dedicated binding
6 MULs, 5 ALUs

$\beta(v_1)$	(1,1)
$\beta(v_2)$	(1,2)
$\beta(v_3)$	(1,2)
$\beta(v_4)$	(2,1)
$\beta(v_5)$	(2,1)
$\beta(v_6)$	(1,3)
$\beta(v_7)$	(1,3)
$\beta(v_8)$	(1,4)
$\beta(v_9)$	(2,1)
$\beta(v_{10})$	(2,2)
$\beta(v_{11})$	(2,2)

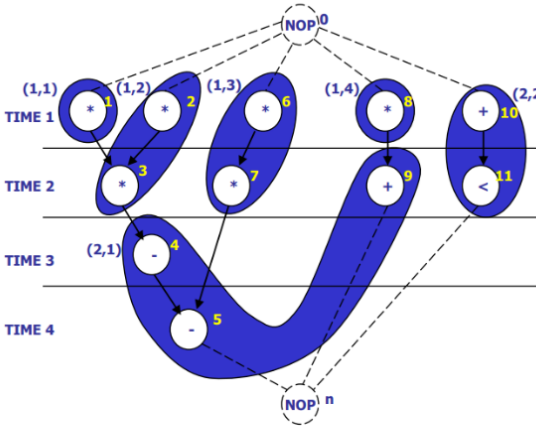


Figure 36: Binding with resource sharing
4 MULS, 2 ALUs

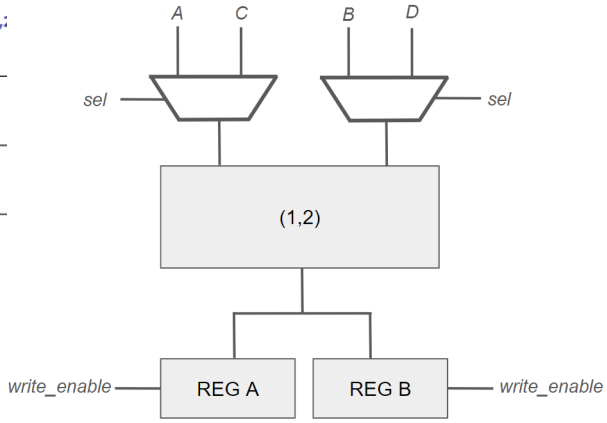


Figure 37: More complex control logic

Resource sharing means that multiplexers and registers need more complicated control signals to manage dataflow. Therefore, a major benefit of utilizing dedicated binding is that it results in greatly simplified control logic.

Consider vertices 2 and 3 in Fig36. Suppose vertex 2 is a (A*B) operation, vertex 3 is a (C*D) operation. We can observe that due to resource-sharing, the shared (1,2) MUX functional unit requires multiplexers (with appropriate *sel* to select which data to operate on), as well as different registers (with different *write enables* to store data)

3 Data-path Synthesis, Control-path Synthesis

Once a complete binding has been done, we will need to synthesize our Data-path and Control-path as the last step

3.1 Data-path Synthesis

- Defining the interconnection amongst ...
  - Functional units (Multiplier, ALU, etc...)
  - Steering resources (MUXs, Buses)
  - Memory Resources (Registers, Memory arrays)
  - **Data-dependent** condition signals (ie. Branching, iteration)

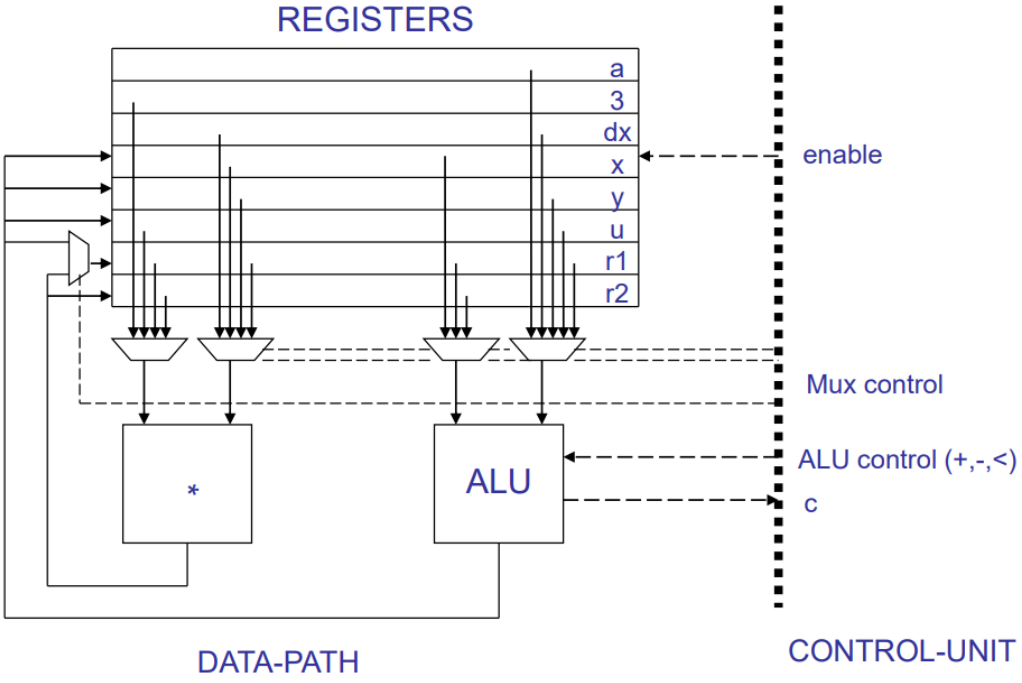


Figure 38: Datapath for 1 MUX, 1 ALU system
Only 2 intermediate variables at any one time, stored in $r1$ and $r2$

3.2 Control-path Synthesis

Suppose we want to synthesize the control signals for the scheduled graph below

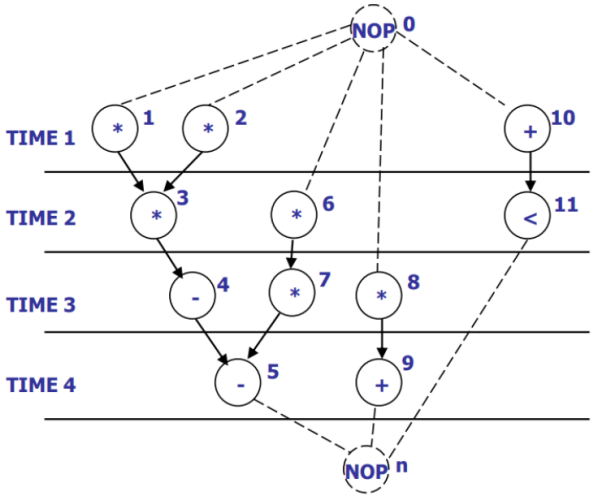
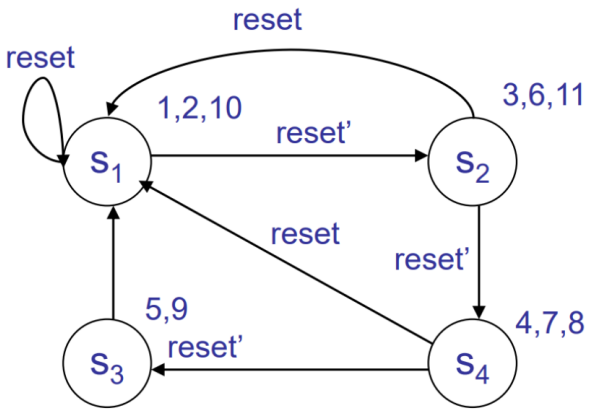


Figure 39: Assume dedicated binding (for simplicity)

There are two ways to do it - Hardwired FSM or Microcode.

3.2.1 Hardwired FSM

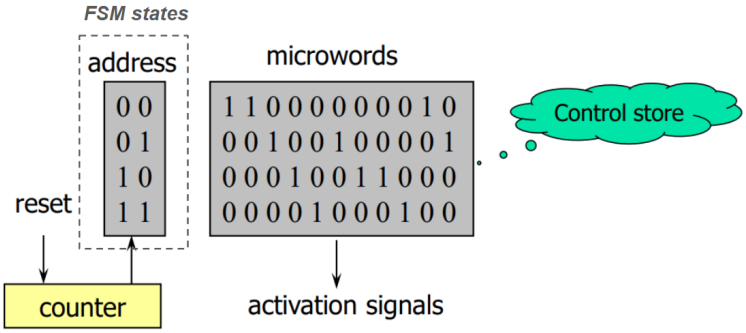
Classical state machine that we are used to



3.2.2 Microcode

- One activation signal per resource (thus 11 bits required for 11 resources)
- 4 states (each potentially controlling up to 11 resources) requires $log_2(4) = 2$ bit ROM counter

3.2.2.1 Horizontal Microcode



3.2.2.2 Vertical Microcode

We can *compress* horizontal microcode (which suffers from sparsity) to form *vertical microcode*

- Allows only one resource to be activated per time-step (if memory is single-port), which can be solved by either ...
  - Lengthening the schedule
  - Reading multiple microcodes per schedule step (requires multi-port memory)

- $\log_2 n_{act}$ bits for n_{act} activation signals
- Needs decoders at ROM output to decode activation signals

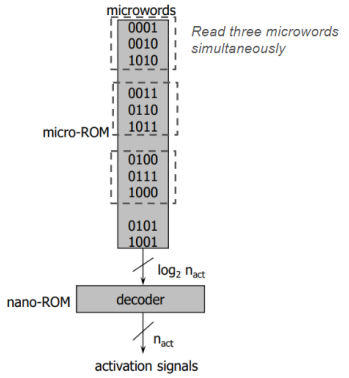


Figure 40: Vertical Microcode
eg. *1010* means that we want to activate resource 10

- Group operations : {v1, v3, v4}, {v2}, {v6, v7, v5}, {v8, v9}, {v10, v11}
  - 5 fields (A to E)
- Each field has no-op (encoded 0 or 00)

Field	Operation	Code
A	v_1	01
A	v_3	10
A	v_4	11
B	v_2	1
C	v_6	01
C	v_7	10
C	v_5	11
D	v_8	01
D	v_9	10
E	v_{10}	01
E	v_{11}	10

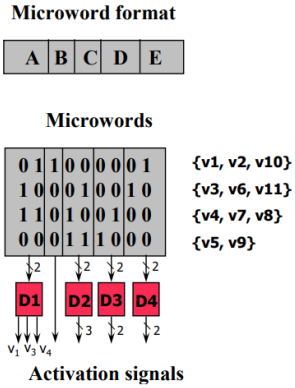


Figure 41: Vertical Microcode with Grouping