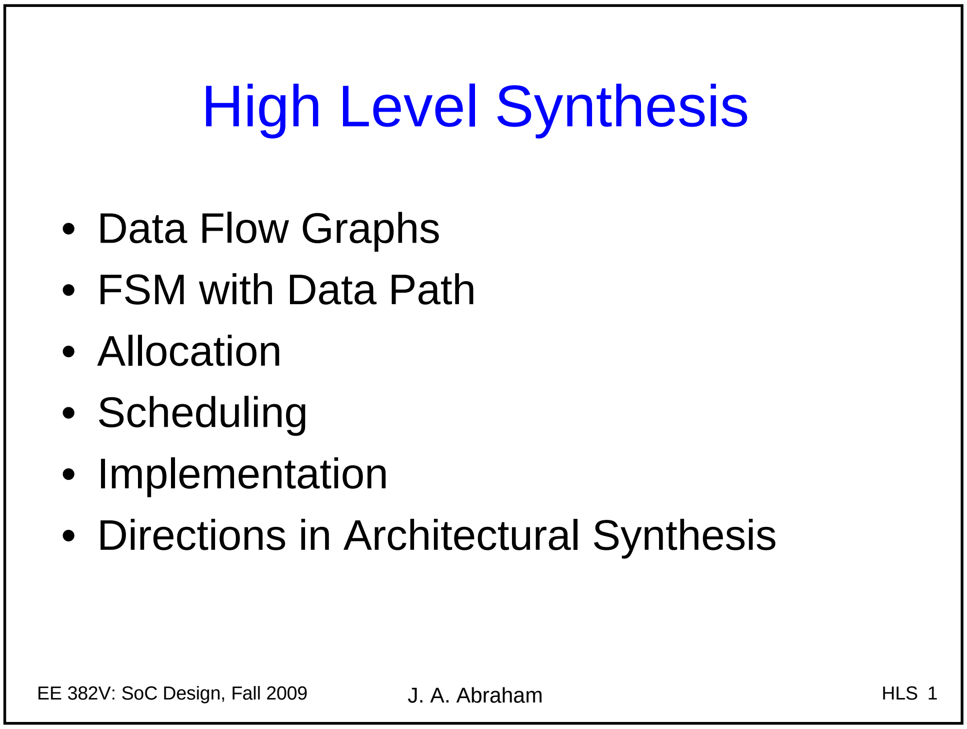
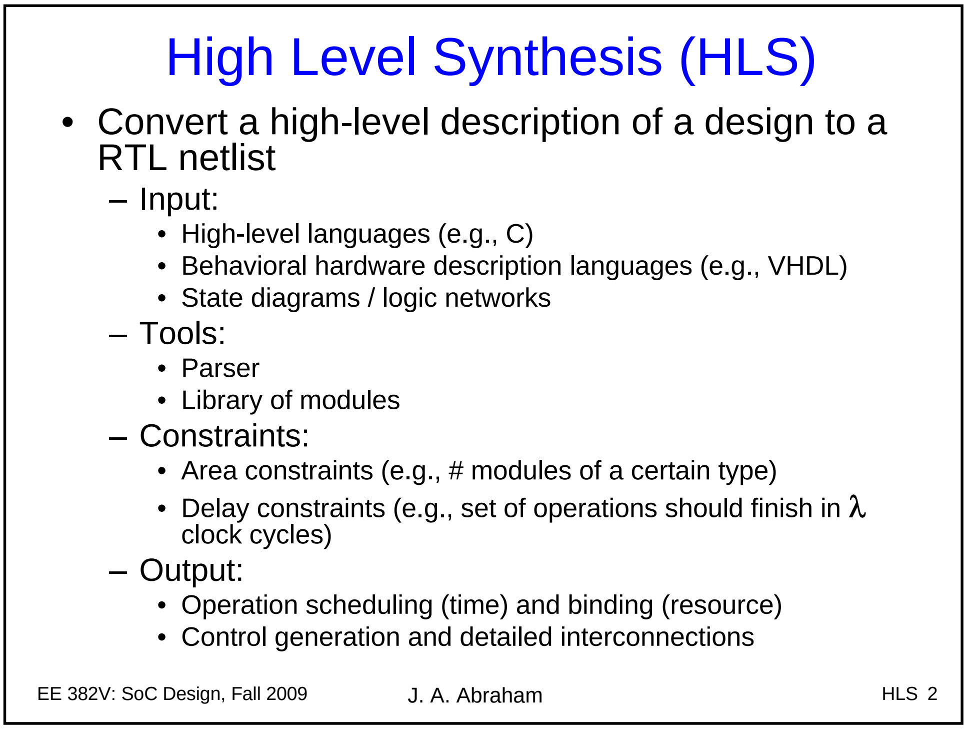


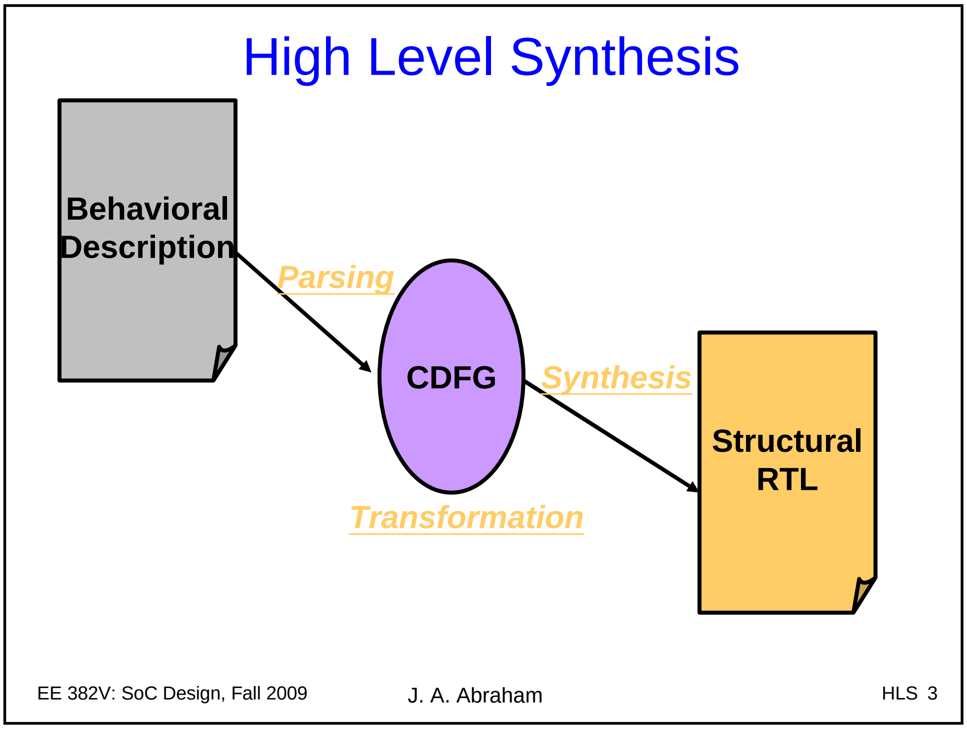
High Level Synthesis

- Data Flow Graphs
- FSM with Data Path
- Allocation
- Scheduling
- Implementation
- Directions in Architectural Synthesis



High Level Synthesis (HLS)

- Convert a high-level description of a design to a RTL netlist
  - Input:
    - High-level languages (e.g., C)
    - Behavioral hardware description languages (e.g., VHDL)
    - State diagrams / logic networks
  - Tools:
    - Parser
    - Library of modules
  - Constraints:
    - Area constraints (e.g., # modules of a certain type)
    - Delay constraints (e.g., set of operations should finish in λ clock cycles)
  - Output:
    - Operation scheduling (time) and binding (resource)
    - Control generation and detailed interconnections



High Level Synthesis

Behavioral Description

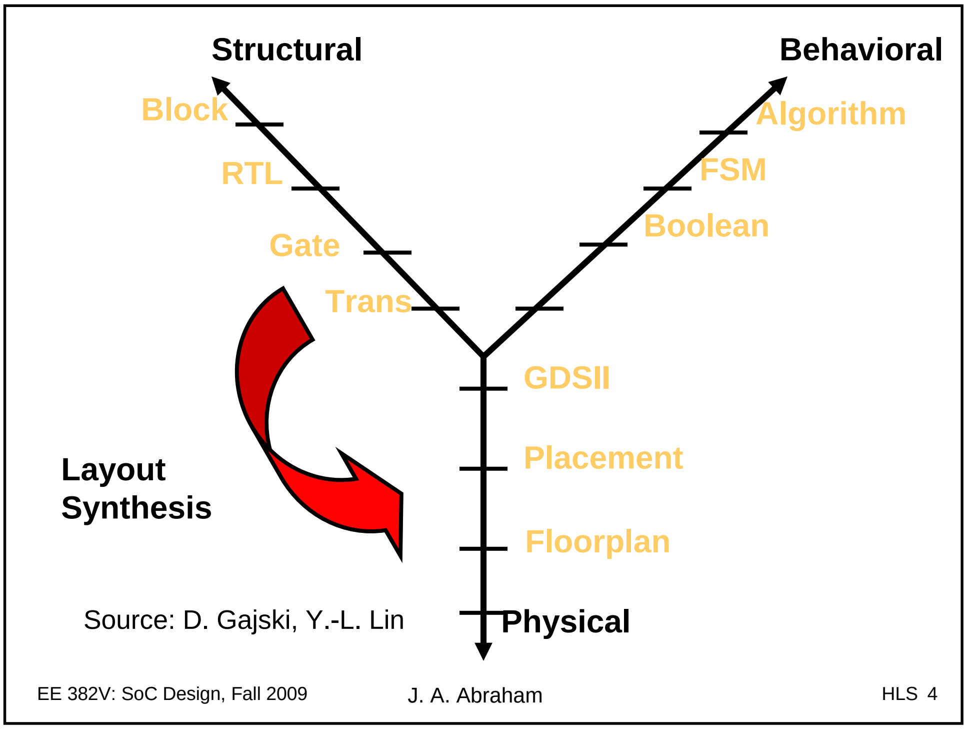
Parsing

CDFG

Synthesis

Structural RTL

Transformation



Structural

Block

RTL

Gate

Trans

Behavioral

Algorithm

FSM

Boolean

GDSII

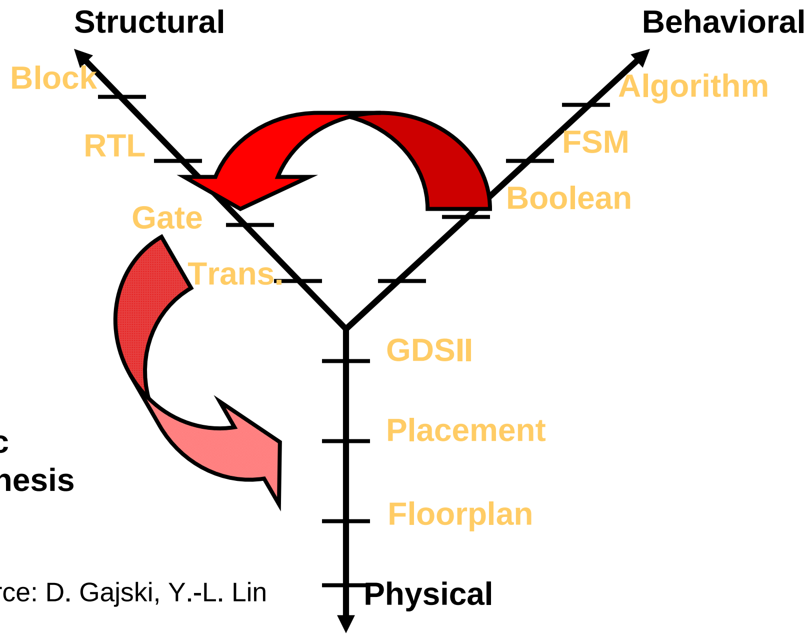
Placement

Floorplan

Physical

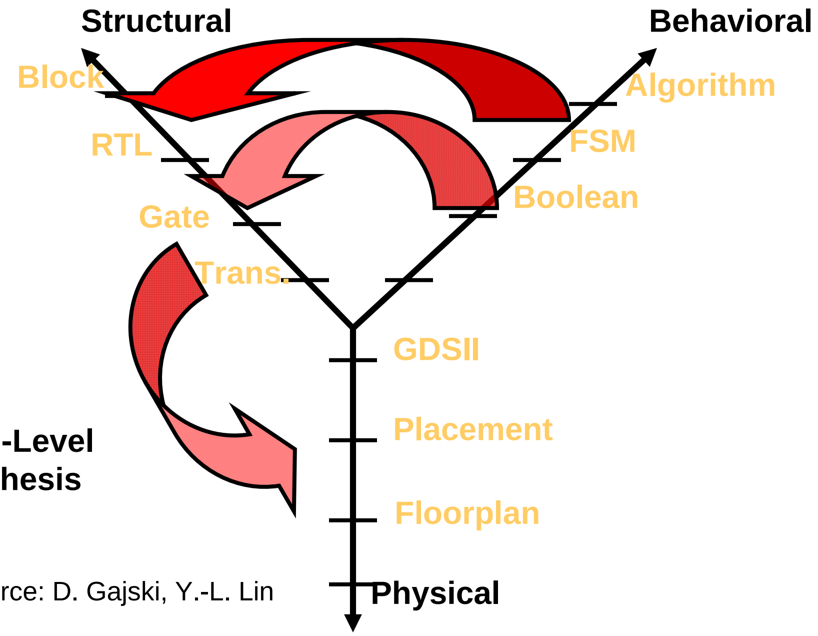
Layout Synthesis

Source: D. Gajski, Y.-L. Lin



Logic Synthesis

Source: D. Gajski, Y.-L. Lin



High-Level Synthesis

Source: D. Gajski, Y.-L. Lin

Essential Issues

- Behavioral Specification Languages
- Target Architectures
- Intermediate Representation
- Operation Scheduling
- Allocation/Binding
- Control Generation

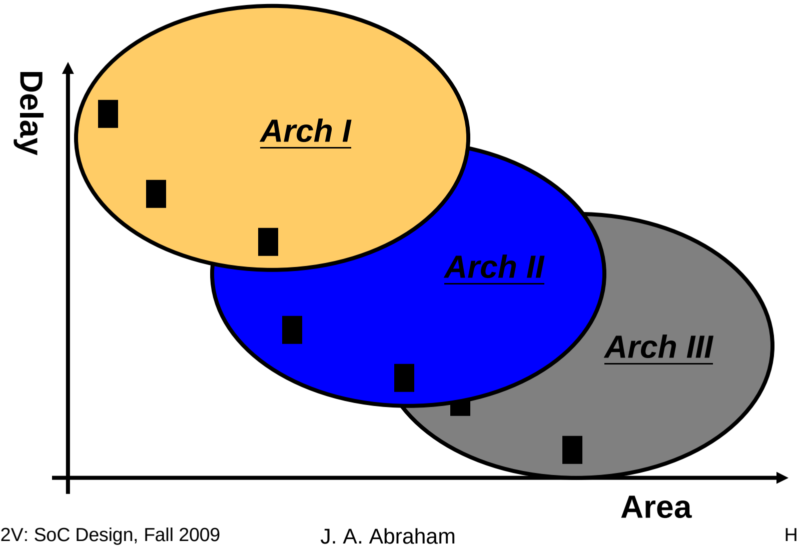
Behavioral Specification Languages

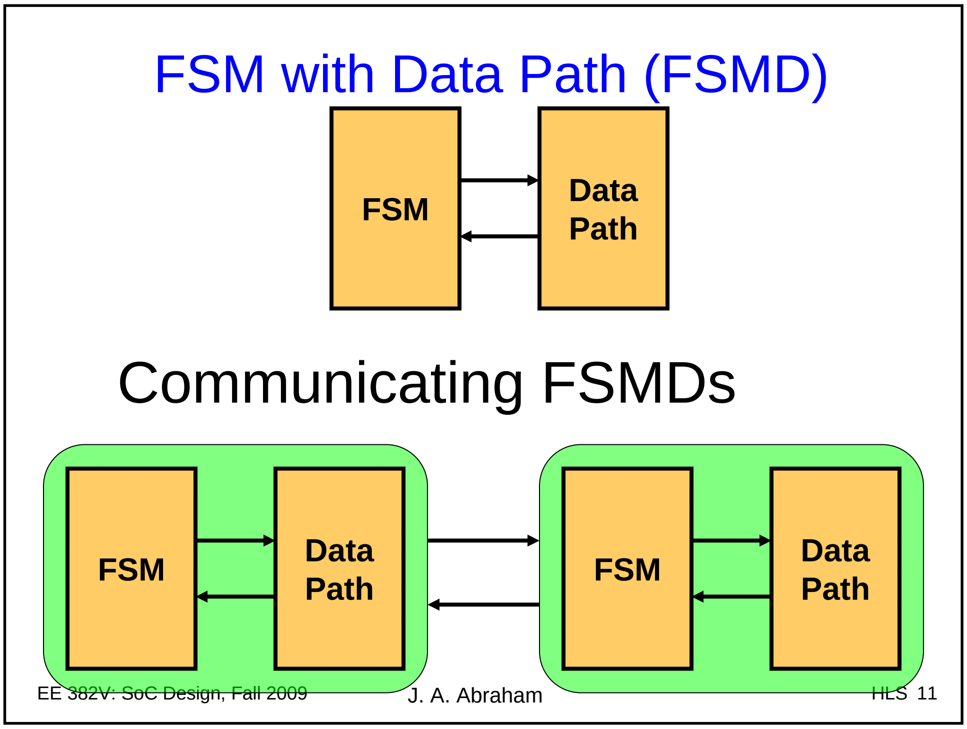
- Add hardware-specific constructs to existing languages
  - – SystemC
- Popular HDL
  - – Verilog, VHDL
- Synthesis-oriented HDL
  - – UDL/I

Target Architectures

- Bus-based
- Multiplexer-based
- Register file
- Pipelined
- RISC, VLIW
- Interface Protocol

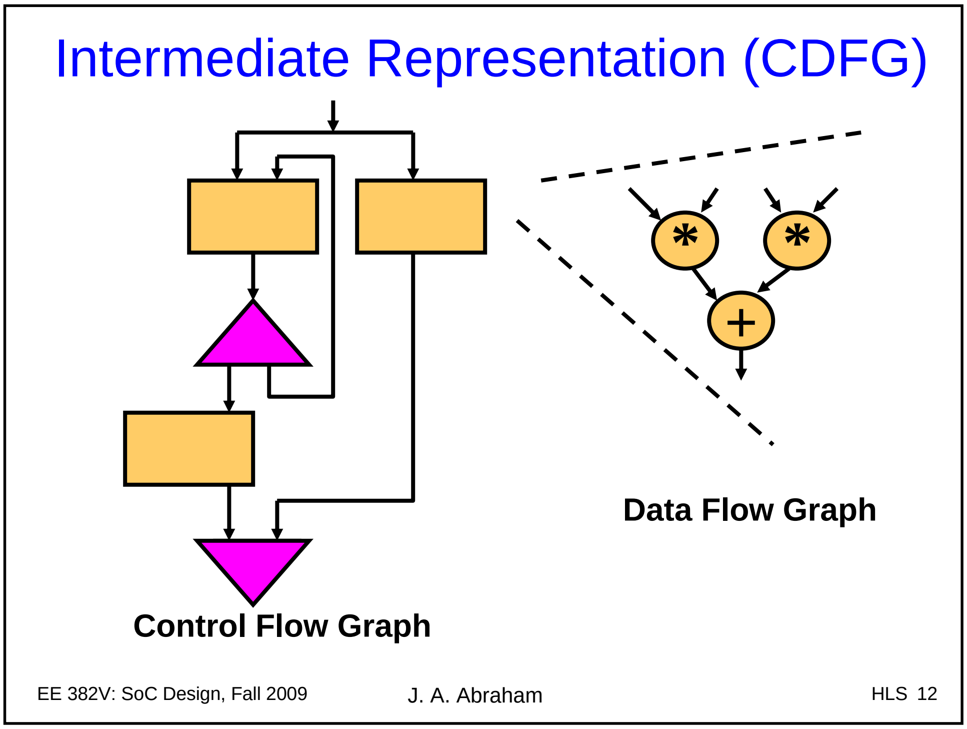
Design Space Exploration





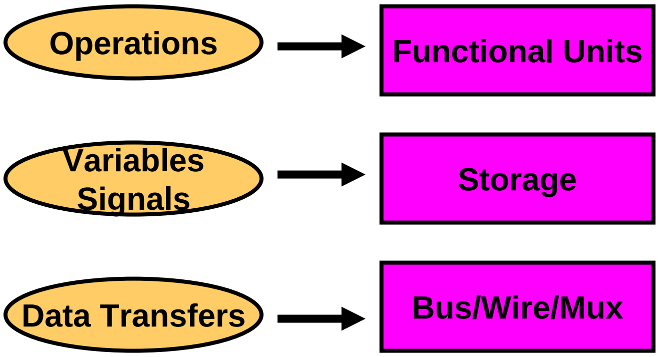
FSM with Data Path (FSMD)

Communicating FSMDs



Intermediate Representation (CDFG)

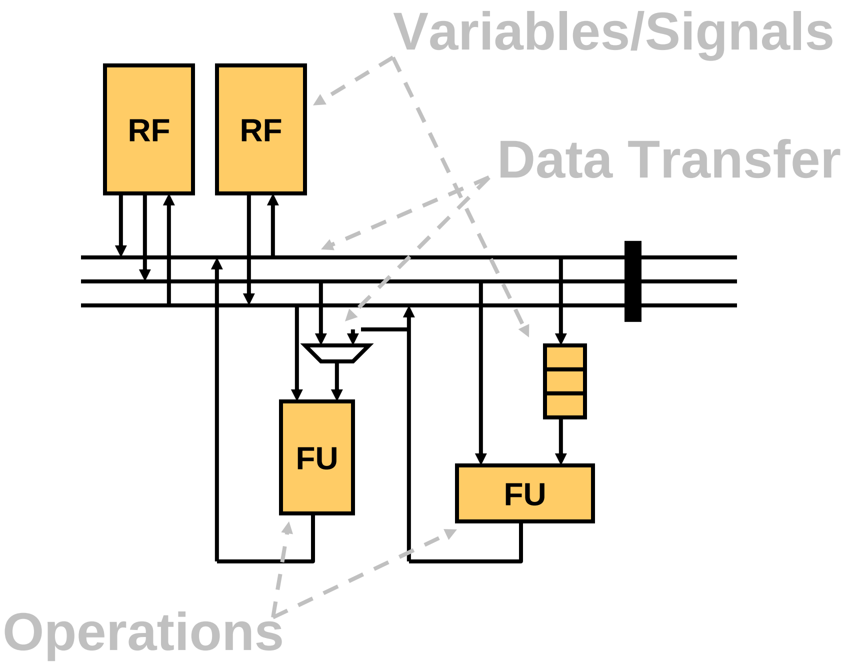
Allocation/Binding



Operations → Functional Units

Variables Signals → Storage

Data Transfers → Bus/Wire/Mux

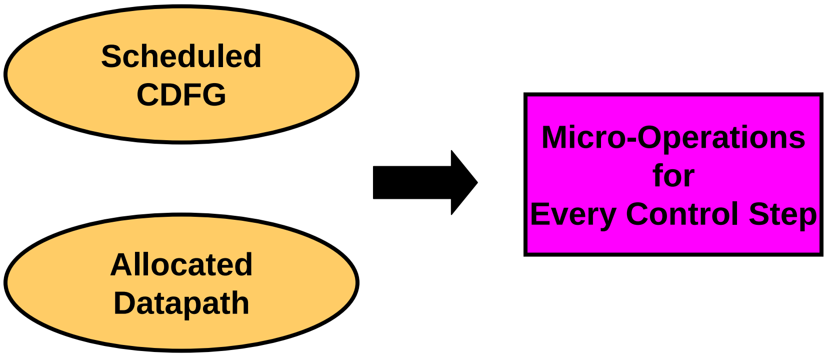


Variables/Signals

Data Transfer

Operations

Controller Specification Generation



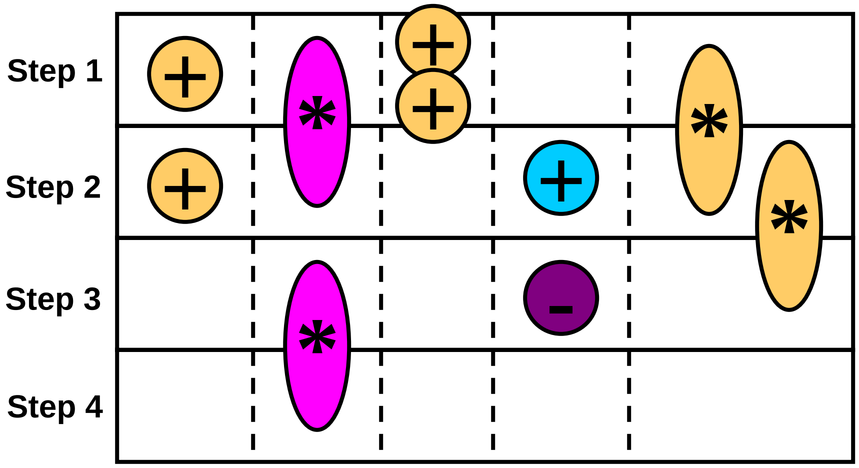
Quality Measures for High-Level Synthesis

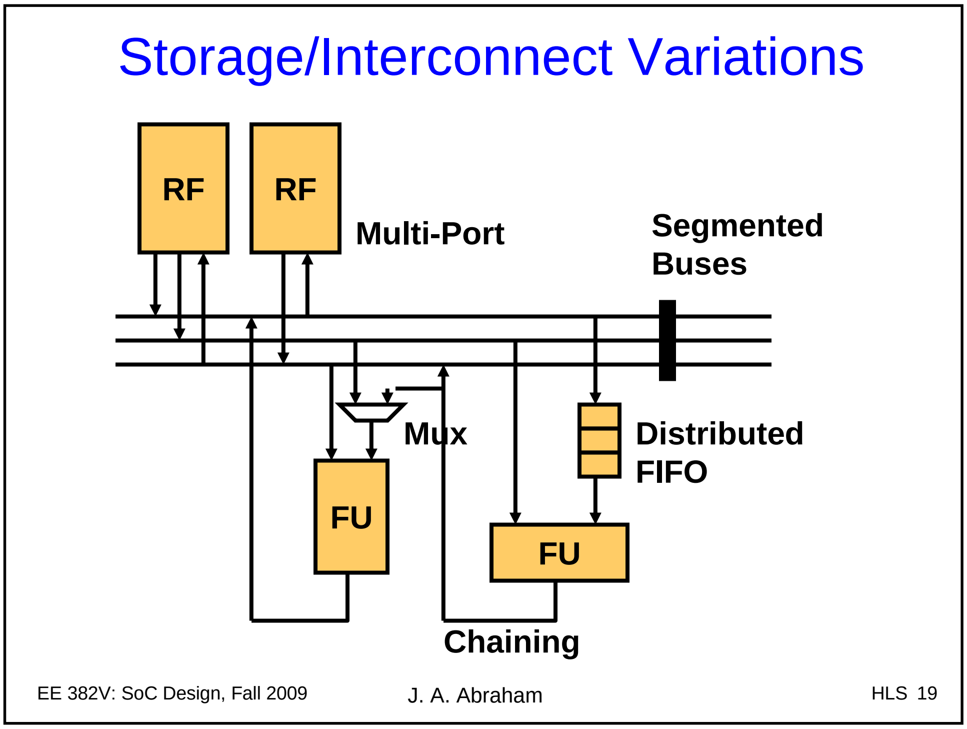
- Performance
- Area Cost
- Power Consumption
- Testability
- Reusability

Hardware Variations

- Functional Units
  - Pipelined, Multi-Cycle, Chained, Multi-Function
- Storage
  - Register, RF, Multi-Ported, RAM, ROM, FIFO, Distributed
- Interconnect
  - Bus, Segmented Bus, Mux, Protocol-Based

Functional Unit Variations





Storage/Interconnect Variations

RF

RF

Multi-Port

Segmented Buses

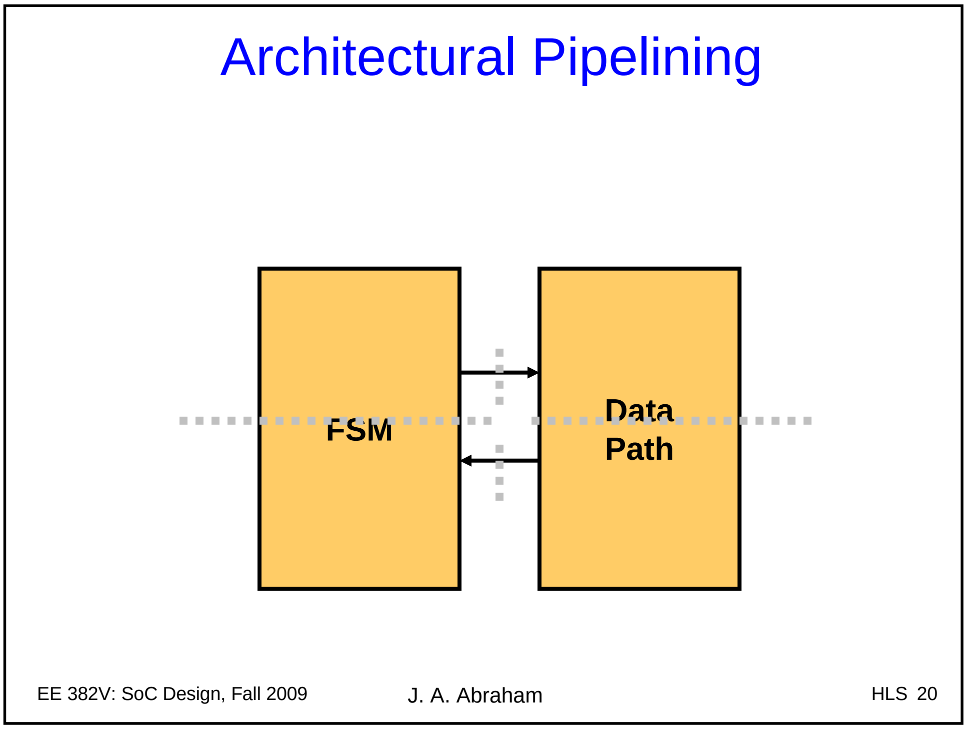
Mux

Distributed FIFO

FU

FU

Chaining

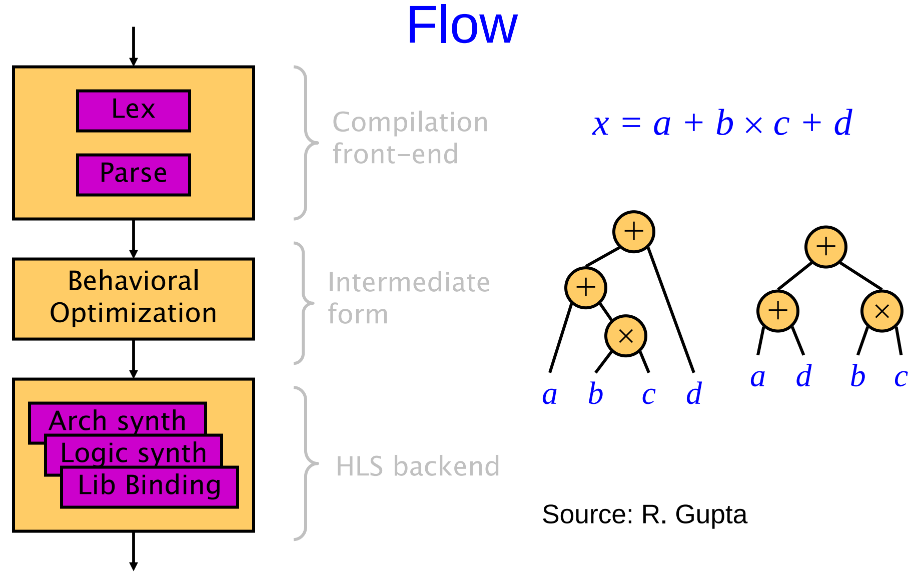


Architectural Pipelining

FSM

Data Path

High-Level Synthesis Compilation Flow



$x = a + b \times c + d$

Source: R. Gupta

Data flow graph

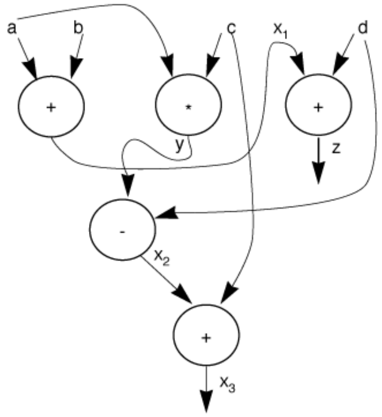
- Data flow graph (DFG) models data dependencies
- Does not require that operations be performed in a particular order
- Models operations in a basic block of a functional model—no conditionals
- Requires single-assignment form

Data flow graph construction

original code:	single-assignment form:
x <= a + b;	x1 <= a + b;
y <= a * c;	y <= a * c;
z <= x + d;	z <= x1 + d;
x <= y - d;	x2 <= y - d;
x <= x + c;	x3 <= x2 + c;

Data flow graph construction, cont'd

Data flow forms directed acyclic graph (DAG):

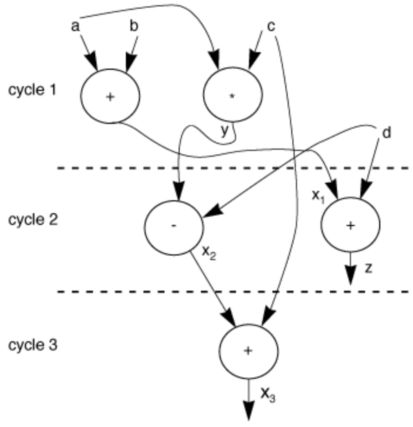


Goals of scheduling and allocation

- Preserve behavior—at end of execution, should have received all outputs, be in proper state (ignoring exact times of events)
- Utilize hardware efficiently
- Obtain acceptable performance

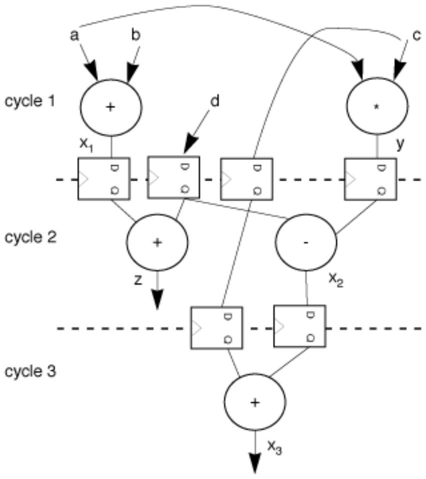
Data flow to data path-controller

One feasible schedule for last DFG:



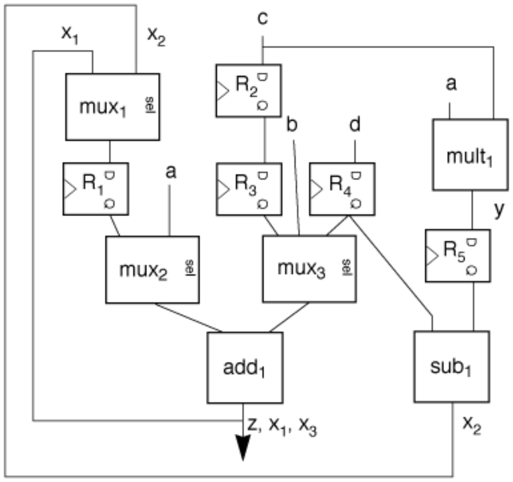
Binding values to registers

registers fall on clock cycle boundaries

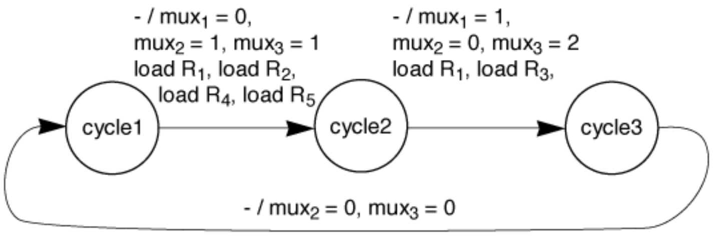


Choosing function units

muxes allow function units to be shared for several operations



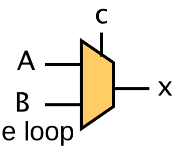
Building the sequencer



Sequencer requires three states,
even with no conditionals

Behavioral Optimization

- Techniques used in software compilation
  - Expression tree height reduction
  - Constant and variable propagation
  - Common sub-expression elimination
  - Dead-code elimination
  - Operator strength reduction
- Typical Hardware transformations
  - Conditional expansion
    - If (c) then x=A else x=B
      ➔ compute A and B in parallel, x=(C)?A:B



  - Loop expansion
    - Instead of three iterations of a loop, replicate the loop body three times

Source: R. Gupta

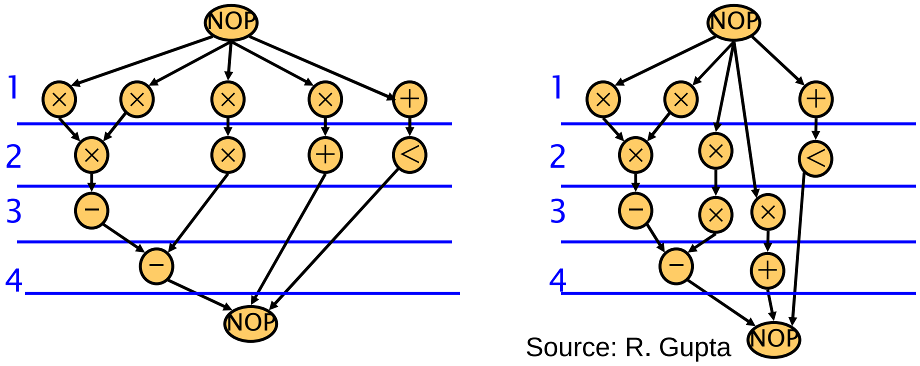
Architectual Synthesis

- Deals with “computational” behavioral descriptions
  - Behavior as sequencing graph (called dependency graph, or data flow graph DFG)
  - Hardware resources as library elements
    - Pipelined or non-pipelined
    - Resource performance in terms of execution delay
  - Constraints on operation timing
  - Constraints on hardware resource availability
  - Storage as registers, data transfer using wires
- Objective
  - Generate a synchronous, single-phase clock circuit
  - Might have multiple feasible solutions (explore tradeoff)
  - Satisfy constraints, minimize objective:
    - Maximize performance subject to area constraint
    - Minimize area subject to performance constraints

Source: R. Gupta

Synthesis in Temporal Domain

- Scheduling and binding can be done in different order or together
  - Schedule is a mapping of operations to time slots (cycles)
  - Scheduled sequencing graph is a labeled graph



Source: R. Gupta

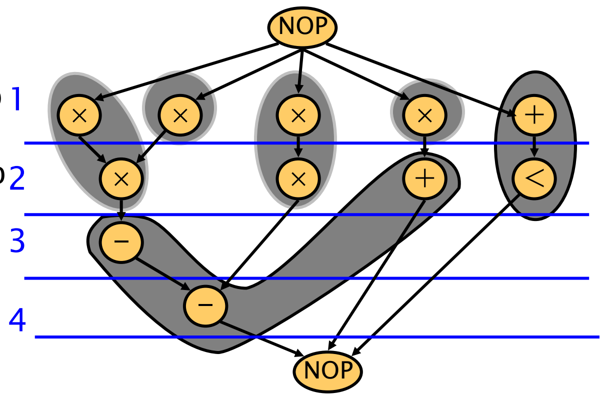
Operation Types

- For each operation, define its *type*.
- For each resource, define a resource type, and a delay (in terms of # cycles)
- T is a relation that maps an operation to a resource type that can implement it
  - $T : V \rightarrow \{1, 2, \ldots, n_{res}\}$.
- More general case:
  - A resource type may implement more than one operation type (e.g., ALU)
- Resource binding:
  - Map each operation to a resource with the same type
  - Might have multiple options

Source: R. Gupta

Schedule in Spatial Domain

- Resource sharing
  - More than one operation bound to same resource
  - Operations have to be serialized
  - Can be represented using hyperedges (define vertex partition)



Source: R. Gupta

Scheduling and Binding

- Resource constraints:
  - Number of resource instances of each type $\{a_k : k=1, 2, ..., n_{res}\}$.
- Scheduling:
  - Labeled vertices $\phi(v_3)=1$.
- Binding:
  - Hyperedges (or vertex partitions) $\beta(v_2)=adder1$.
- Cost:
  - Number of resources ≈ area
  - Registers, steering logic (Muxes, busses), wiring, control unit
- Delay:
  - Start time of the "sink" node
  - Might be affected by steering logic and schedule (control logic) – resource-dominated vs. ctrl-dominated

Architectural Optimization

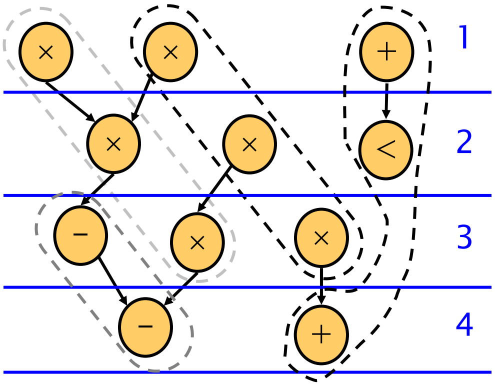
- Optimization in view of design space flexibility
- A multi-criteria optimization problem:
  - Determine schedule ϕ and binding β.
  - Under area A, latency λ and cycle time τ objectives
- Find non-dominated points in solution space
- Solution space tradeoff curves:
  - Non-linear, discontinuous
  - Area / latency / cycle time (more?)
- Evaluate (estimate) cost functions
- Unconstrained optimization problems for resource dominated circuits:
  - Min area: solve for minimal binding
  - Min latency: solve for minimum λ scheduling

Scheduling and Binding

- Cost λ and A determined by both ϕ and β.
  - Also affected by floorplan and detailed routing
- β affected by ϕ:
  - Resources cannot be shared among concurrent ops
- ϕ affected by β:
  - Resources cannot be shared among concurrent ops
  - When register and steering logic delays added to execution delays, might violate cycle time.
- Order?
  - Apply either one (scheduling, binding) first

How Is the Datapath Implemented?

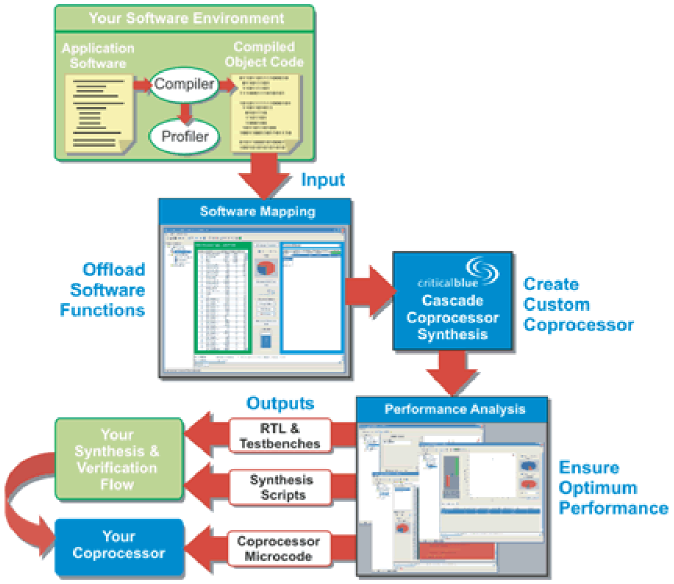
- Assuming the following schedule and binding
- Wires between modules?
- Input selection?
- How does binding/ scheduling affect congestion?
- How does binding/ scheduling affect steering logic?



Co-Processor Synthesis

- Bruce and Taylor, *Chip Design*, 2005
- Accelerators for speeding up software execution
- Exploit parallelism in the software
- Synthesize custom control logic and datapaths
- Explore candidate architectures and optimize

Design Flow for Coprocessor Synthesis



Example Algorithm

- Accelerate BCH3.c algorithm
  - www.eccpage.com/bch3.c
- Triple-error-correction encoder/decoder
  - correct transmission bit errors resulting from a “lossy” environment
  - SONET, ATM
- Algorithm: approx. 600 lines of C
  - Two primary functions: encode_bch, decode_bch

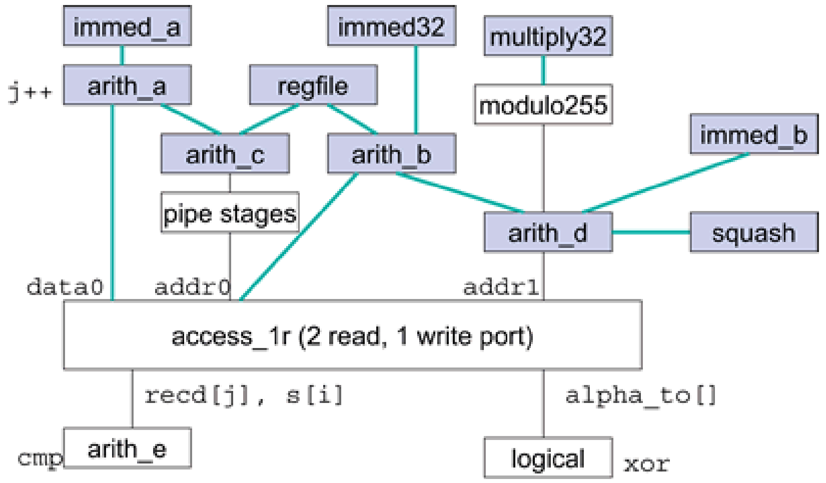
Analysis of Algorithm

- Four inner loops consume 85-95% of execution time
  - DEC1, DEC2, ENC1, and ENC2
  - less than 20 lines (› 3%) of code
- Example: DEC1 code:

```
for (j=0; j < length; j++)
 if (recd[j] != 0) s[i] ^= alpha to[(i*j)%n];
```

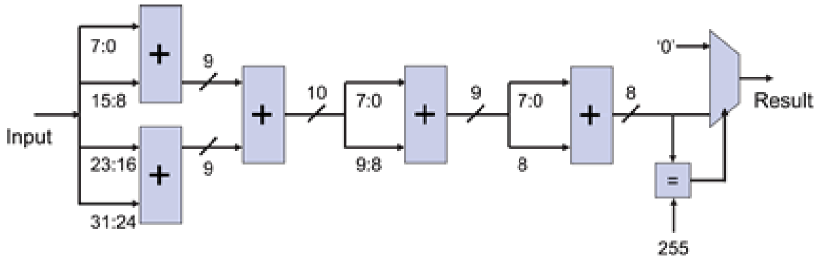
  - length varies from 64 to 1024 bits
  - this loop is nested within another loop which executes 16 times
- Total executions: 1024 to 16384

Functional Blocks for Coprocessor



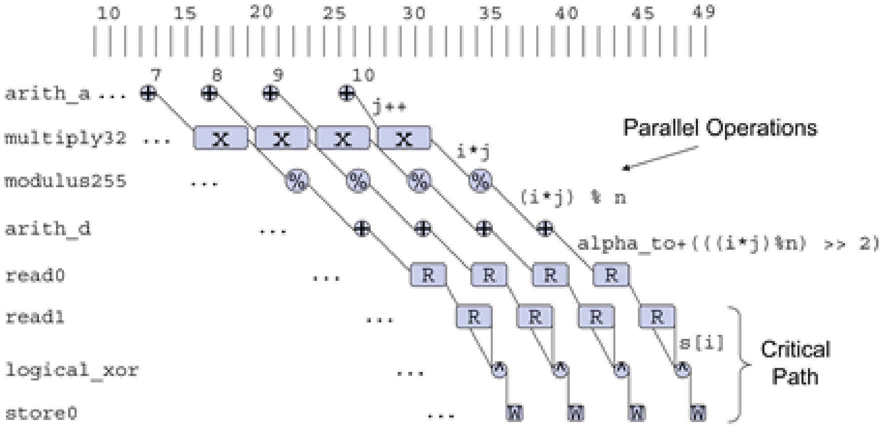
```
logical_a/result_o[0], arithmetic_c/result_o[2] =>
[S11:WRITE_WORD, 0 on access_1_0/0]
```

Coprocessor Design and Performance



Function	Execution time (cycles)		Acceleration
	ARM9	Coprocessor	
Encode	550,233	125,926	4.37
Decode	1,430,294	280,432	5.10
Total	**1,980,527**	**406,358**	**4.87**

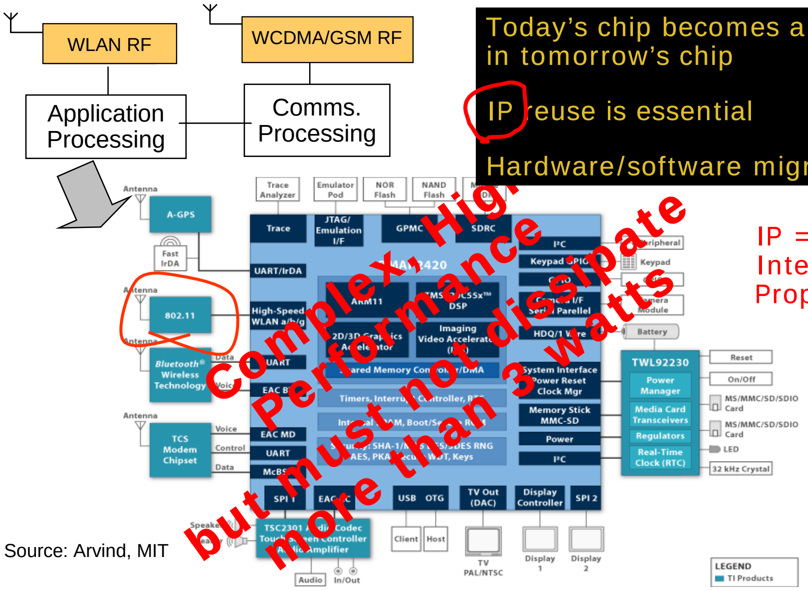
DEC1 Algorithm Execution



Adoption of High-Level Synthesis

- Automated tools for high-level synthesis are not used widely
  - Low-level structuring primitives (e.g., Behavioural Verilog still has modules)
  - Scheduling performed statically
  - Black-box approach (tools are not as smart as engineers yet)
  - Artificial separation of control/data-flow (C is not a good language for hardware description)

Current Cellphone Architecture



Complex, High Performance
but must not dissipate
more than 3 watts

Today's chip becomes a block in tomorrow's chip

IP reuse is essential

Hardware/software migration

IP = Intellectual Property

Source: Arvind, MIT

An under appreciated fact

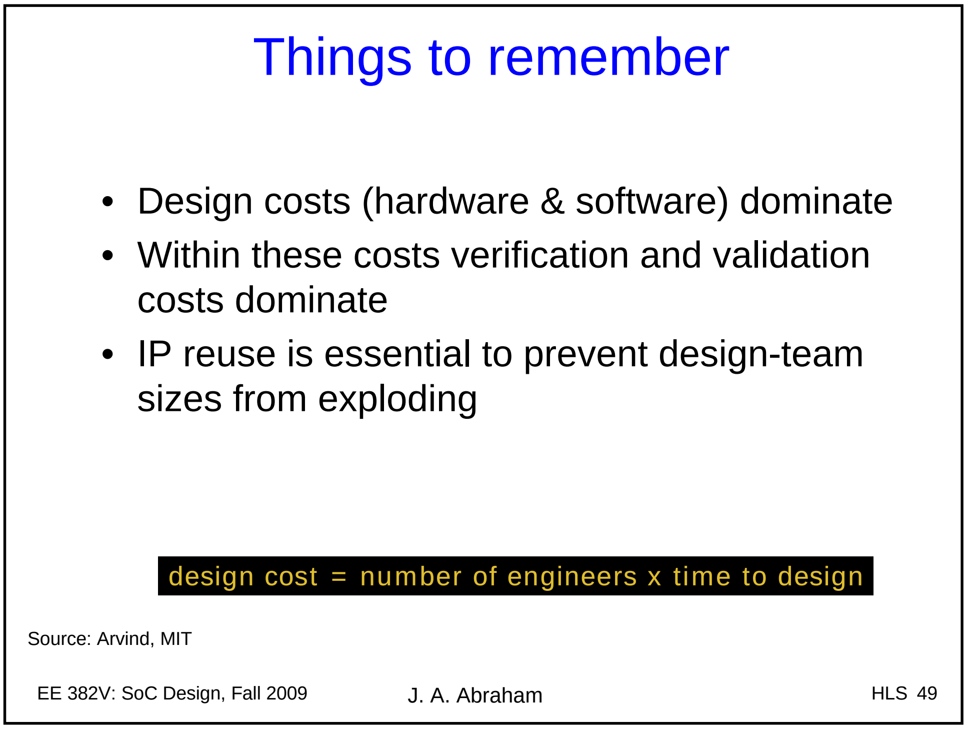
- If a functionality (e.g. H.264) is moved from a programmable device to a specialized hardware block, the power/energy savings are 100 to 1000 fold

Power savings ⇒ more specialized hardware

but our mind set

- Software is forgiving
- Hardware design is difficult, inflexible, brittle, error prone, ...

Source: Arvind, MIT

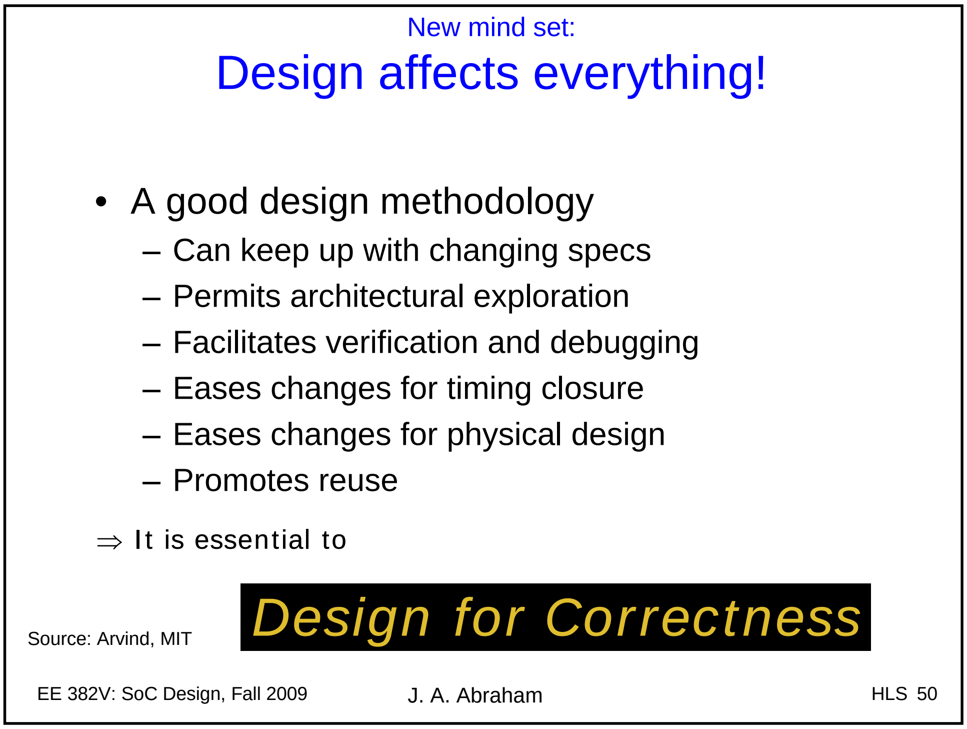


Things to remember

- Design costs (hardware & software) dominate
- Within these costs verification and validation costs dominate
- IP reuse is essential to prevent design-team sizes from exploding

design cost = number of engineers x time to design

Source: Arvind, MIT



New mind set:

Design affects everything!

- A good design methodology
  - Can keep up with changing specs
  - Permits architectural exploration
  - Facilitates verification and debugging
  - Eases changes for timing closure
  - Eases changes for physical design
  - Promotes reuse

⇒ It is essential to

Design for Correctness

Source: Arvind, MIT

Term Rewriting for High Level Synthesis

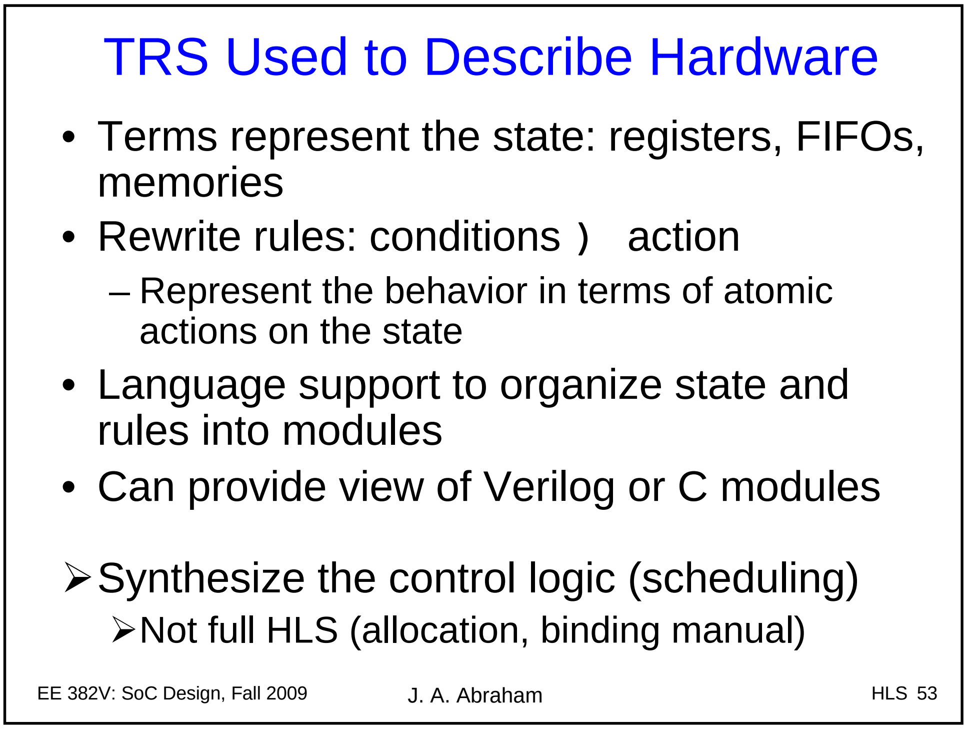
- Research at MIT (Arvind group)
- New programming language to facilitate high level synthesis
  - Object oriented
  - Rich types
  - Higher-order functions
  - Transformable
  - Borrows from Haskell
- Commercial: Bluespec

Term Rewriting Systems: Example

- Terms: GCD(x,y)
- Rewrite rules:
  - GCD(x,y)) GCD(y,x) if x > y, y ≠ 0
  - GCD(x,y)) GCD(x,y-x) if x – y, y ≠ 0
- Initial term: GCD(initX, initY)

$$\mathrm{GCD}(6, 15) \overset{R_2}{\Rightarrow} \mathrm{GCD}(6, 9) \overset{R_2}{\Rightarrow} \mathrm{GCD}(6, 3) \overset{R_1}{\Rightarrow}$$

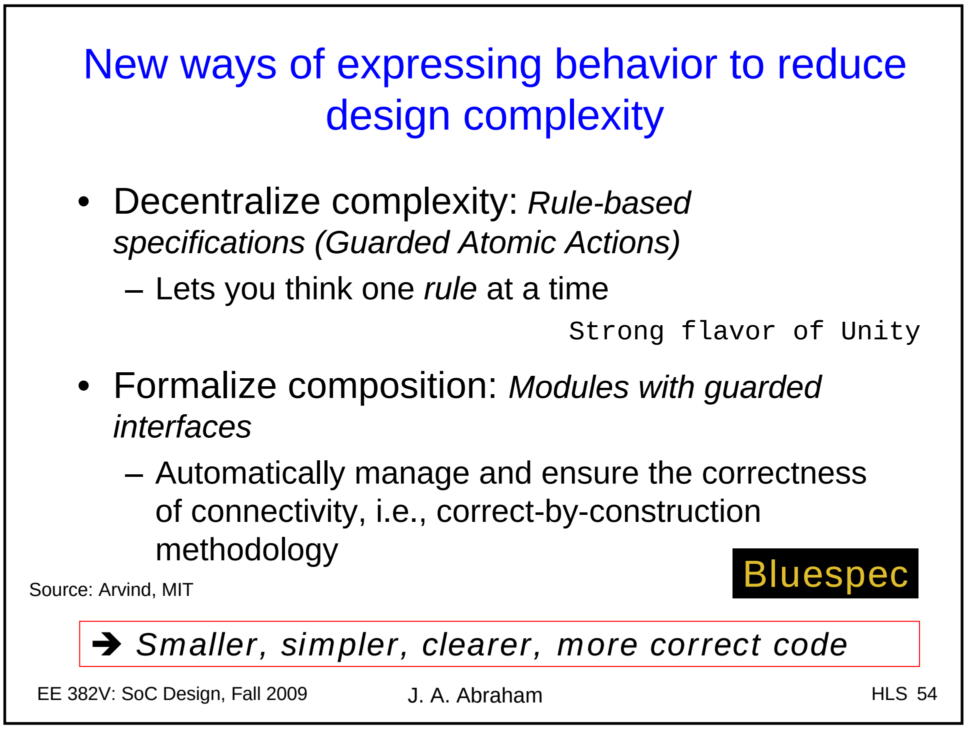
$$\mathrm{GCD}(3, 6) \overset{R_2}{\Rightarrow} \mathrm{GCD}(3, 3) \overset{R_2}{\Rightarrow} \mathrm{GCD}(3, 0)$$



TRS Used to Describe Hardware

- Terms represent the state: registers, FIFOs, memories
- Rewrite rules: conditions)   action
  - Represent the behavior in terms of atomic actions on the state
- Language support to organize state and rules into modules
- Can provide view of Verilog or C modules

➢ Synthesize the control logic (scheduling)
  ➢ Not full HLS (allocation, binding manual)



New ways of expressing behavior to reduce design complexity

- Decentralize complexity: *Rule-based specifications (Guarded Atomic Actions)*
  - Lets you think one *rule* at a time

`Strong flavor of Unity`

- Formalize composition: *Modules with guarded interfaces*
  - Automatically manage and ensure the correctness of connectivity, i.e., correct-by-construction methodology

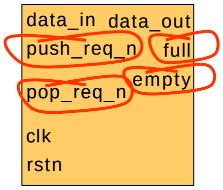
Bluespec

Source: Arvind, MIT

➔ *Smaller, simpler, clearer, more correct code*

Reusing IP Blocks

Example: Commercially available FIFO IP block



An error occurs if a push is attempted while the FIF l.

Thus, there is no conflict in a simul when the FIFO is full. A simultaneous push and pop c FIFO is empty, since there is no pop data to prefetch. How red in the FIFO.

A pop o pop_req_n is asserted (LOW), as long as the FIFO is not en eq_n causes the internal read pointer to be incremented on the nex of clk. Thus, the RAM read data must be captured on the clk following the a on of pop_req_n.

No machine verification of such informal constraints is feasible

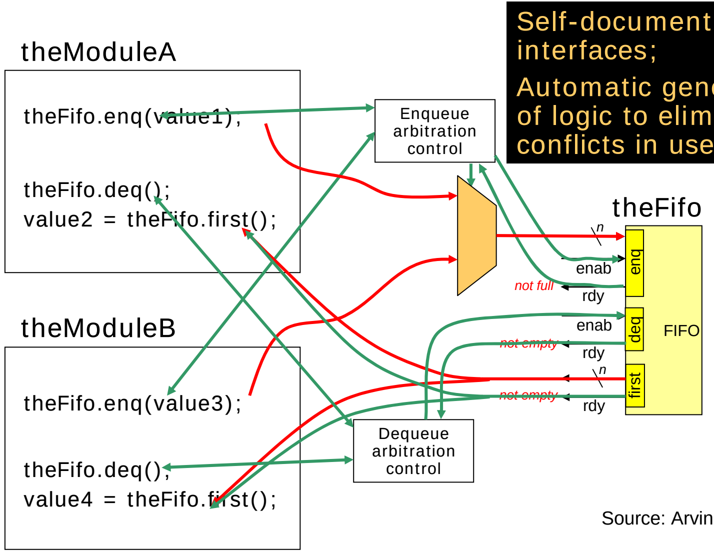
These constraints are spread over many pages of the documentation…

Source: Arvind, MIT

Bluespec promotes composition through guarded interfaces

Self-documenting interfaces;

Automatic generation of logic to eliminate conflicts in use.



theModuleA

```
theFifo.enq(value1);

theFifo.deq();
value2 = theFifo.first();
```

theModuleB

```
theFifo.enq(value3);

theFifo.deq();
value4 = theFifo.first();
```

Source: Arvind, MIT

Bluespec SystemVerilog (BSV)

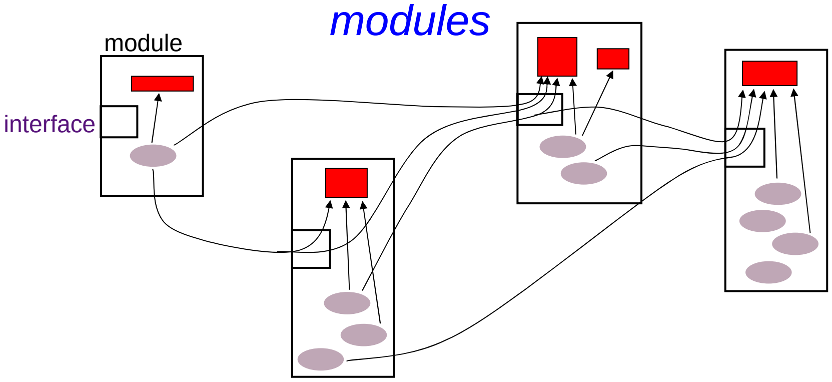
- Power to express complex static structures and constraints
  - – Checked by the compiler
- “Micro-protocols” are managed by the compiler
  - – The necessary hardware for muxing and control is generated automatically and is correct by construction
- Easier to make changes while preserving correctness
- Also available: Bluespec in SystemC (ESEPro)

➔ *Smaller, simpler, clearer, more correct code*

➔ *not just simulation, synthesis as well*

Source: Arvind, MIT

Bluespec: State and Rules organized into *modules*



All *state* (e.g., Registers, FIFOs, RAMs, ...) is explicit.
Behavior is expressed in terms of atomic actions on the state:

Rule: condition ➔ action

Rules can manipulate state in other modules only *via* their interfaces.

Source: Arvind, MIT

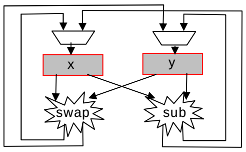
Programming with rules: A simple example

Euclid's algorithm for computing the Greatest Common Divisor (GCD):

15		6	
9		6	*subtract*
3		6	*subtract*
6		3	*swap*
3		3	*subtract*
0	*answer:*	3	*subtract*

Source: Arvind, MIT

GCD in BSV



Source: Arvind, MIT

```
module mkGCD (I_GCD);
    Reg#(int) x <- mkRegU;
    Reg#(int) y <- mkReg(0);

    rule swap ((x > y) &&  (y != 0));
        x <= y;  y <= x;
    endrule
    rule subtract ((x <= y) && (y != 0));
        y <= y – x;
    endrule

    method Action start(int a, int b) if (y==0);
       x <= a;  y <= b;
    endmethod
    method int result() if (y==0);
        return x;
    endmethod
endmodule
```

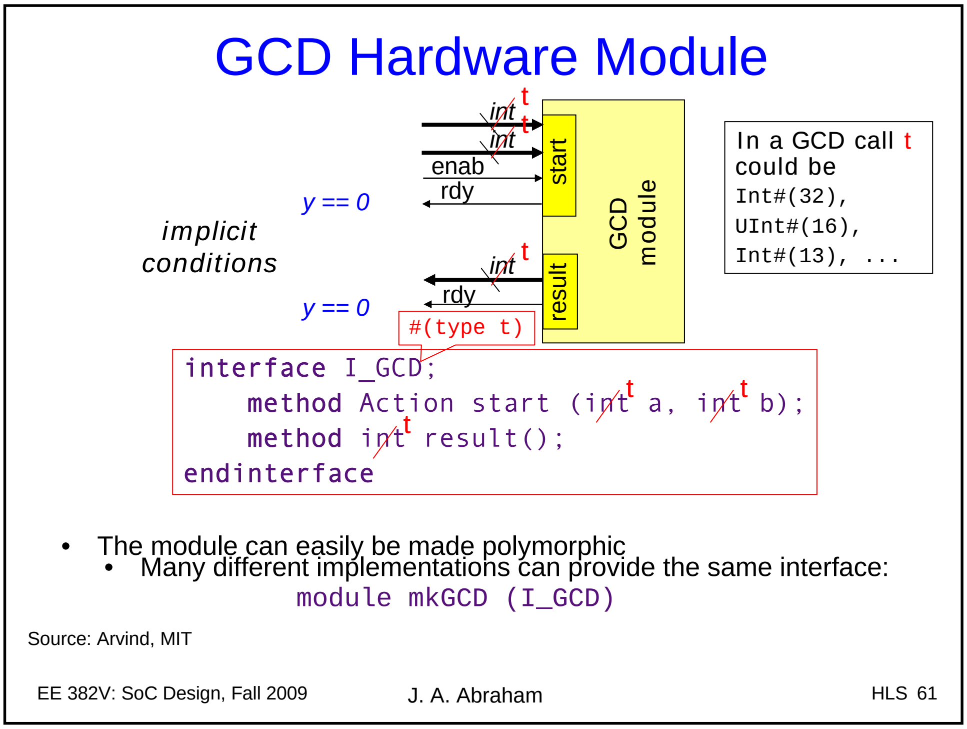
State

```
typedef int Int#(32)
```

Internal behavior

External interface

Assumes x /= 0 and y /= 0



GCD Hardware Module

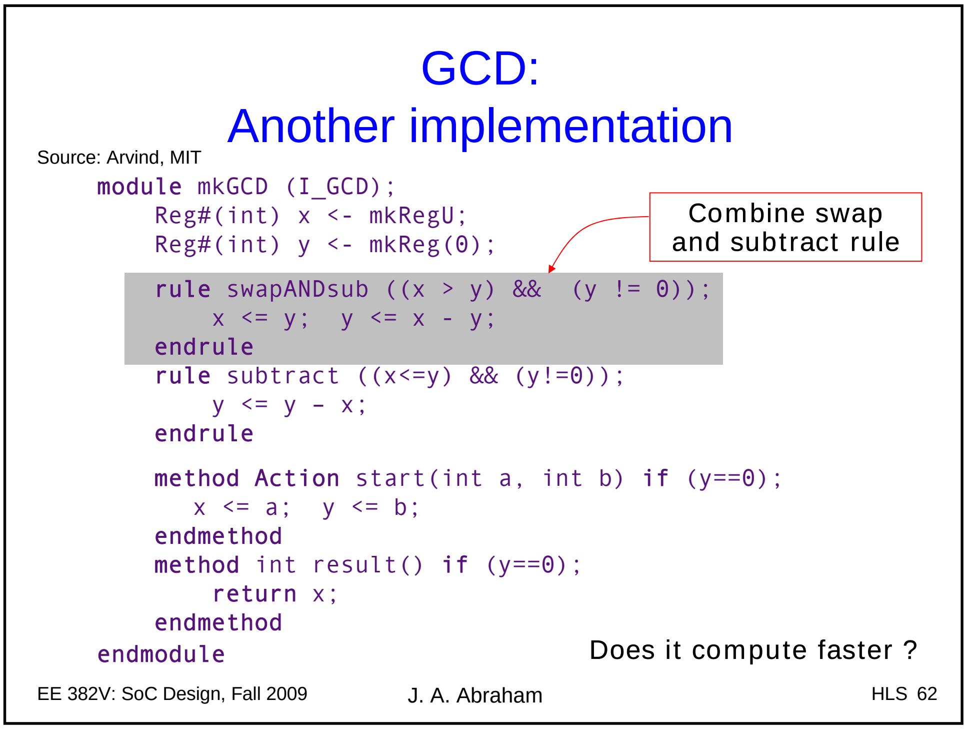
In a GCD call t could be
`Int#(32)`,
`UInt#(16)`,
`Int#(13)`, ...

`#(type t)`

```
interface I_GCD;
    method Action start (~~int~~t a, ~~int~~t b);
    method ~~int~~t result();
endinterface
```

- The module can easily be made polymorphic
  - Many different implementations can provide the same interface:
    `module mkGCD (I_GCD)`

Source: Arvind, MIT



GCD:
Another implementation

Source: Arvind, MIT

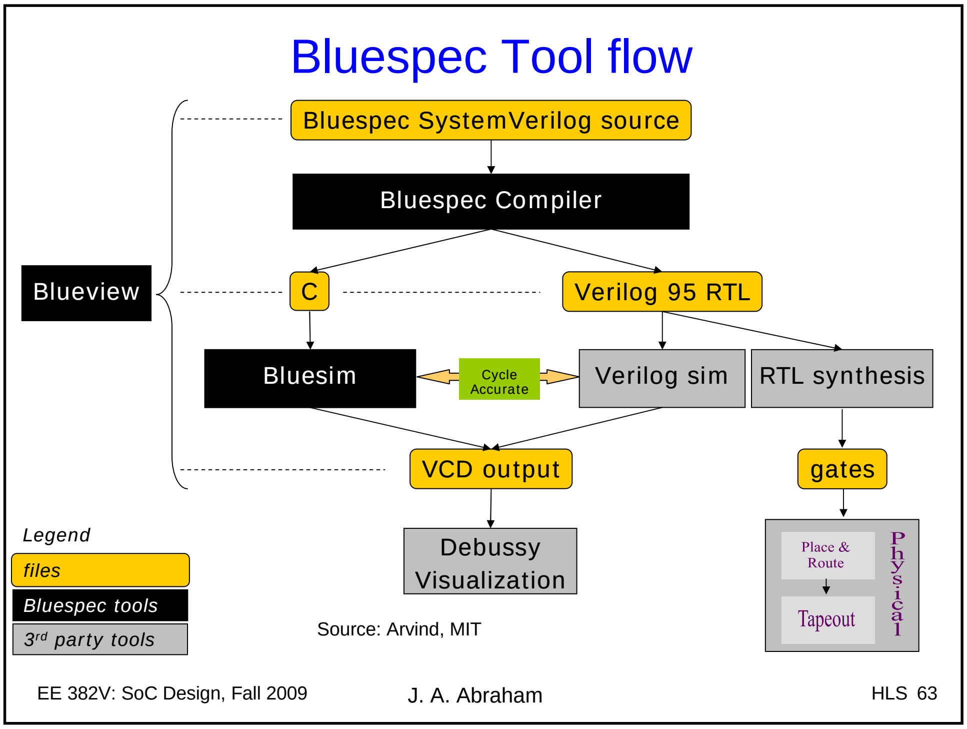
```
module mkGCD (I_GCD);
    Reg#(int) x <- mkRegU;
    Reg#(int) y <- mkReg(0);

    rule swapANDsub ((x > y) && (y != 0));
        x <= y;  y <= x - y;
    endrule
    rule subtract ((x<=y) && (y!=0));
        y <= y – x;
    endrule

    method Action start(int a, int b) if (y==0);
       x <= a;  y <= b;
    endmethod
    method int result() if (y==0);
        return x;
    endmethod
endmodule
```

Combine swap and subtract rule

Does it compute faster ?



Bluespec Tool flow

Bluespec SystemVerilog source

Bluespec Compiler

Blueview

C

Verilog 95 RTL

Bluesim

Cycle Accurate

Verilog sim

RTL synthesis

VCD output

gates

Debussy Visualization

Place & Route

Tapeout

Physical

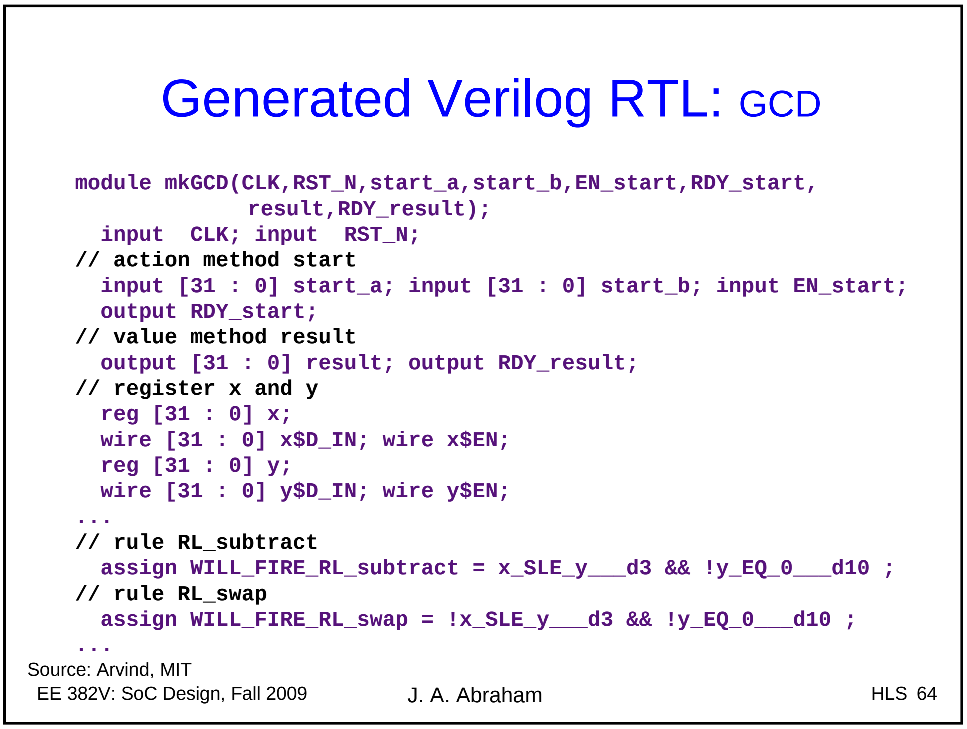
Legend

files

Bluespec tools

3rd party tools

Source: Arvind, MIT

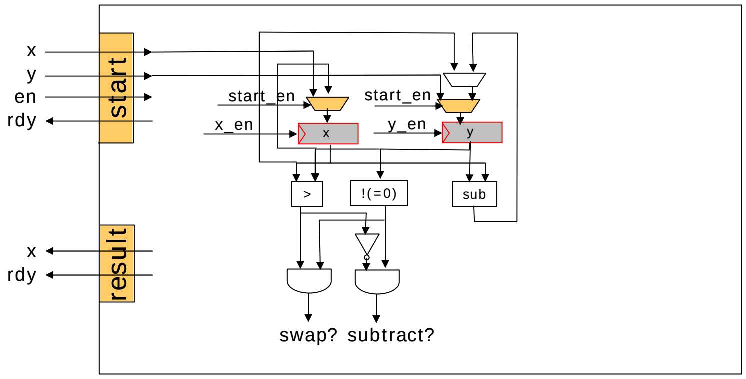


Generated Verilog RTL: GCD

```
module mkGCD(CLK,RST_N,start_a,start_b,EN_start,RDY_start,
             result,RDY_result);
  input  CLK; input  RST_N;
// action method start
  input [31 : 0] start_a; input [31 : 0] start_b; input EN_start;
  output RDY_start;
// value method result
  output [31 : 0] result; output RDY_result;
// register x and y
  reg [31 : 0] x;
  wire [31 : 0] x$D_IN; wire x$EN;
  reg [31 : 0] y;
  wire [31 : 0] y$D_IN; wire y$EN;
...
// rule RL_subtract
  assign WILL_FIRE_RL_subtract = x_SLE_y___d3 && !y_EQ_0___d10 ;
// rule RL_swap
  assign WILL_FIRE_RL_swap = !x_SLE_y___d3 && !y_EQ_0___d10 ;
...
```

Source: Arvind, MIT

Generated Hardware Module



x_en = swap? OR start_en
y_en = swap? OR subtract? OR start_en

rdy = (y==0)

Source: Arvind, MIT

GCD: Synthesis results

- Original (16 bits)
  - Clock Period: 1.6 ns
  - Area: 4240 μm^2
- Unrolled (16 bits)
  - Clock Period: 1.65ns
  - Area: 5944 μm^2
- Unrolled takes 31% fewer cycles on the testbench

Source: Arvind, MIT