Course Structure

Part 1
ASIC Design Overview

Part 2
Digital CMOS Circuits

Prereq
Computer Architecture

Part 3
CAD Algorithms
Part 1: ASIC Design Overview

- **Topic 1**: Hardware Description Languages
- **Topic 2**: CMOS Devices
- **Topic 3**: CMOS Circuits
- **Topic 4**: Full-Custom Design Methodology
- **Topic 5**: Automated Design Methodologies
- **Topic 6**: Closing the Gap
- **Topic 7**: Clocking, Power Distribution, Packaging, and I/O
- **Topic 8**: Testing and Verification
Agenda

Evolution of Hardware Description Languages

Hardware Description Languages Across Stack

“High-Level” RTL with SystemVerilog

Guarded-Atomic Actions with Bluespec

System-Level Modeling with SystemC
Originally designers used manual translation and breadboards for verification

Algorithm:

```plaintext
while ( a > 0 )
    b = b * a
    a = a - 1
```

Register Transfer Level

Gate Level

Layout

Manual

Verification via Breadboard
Hardware description languages enabled gate-level verification via simulation

Algorithm

\[
\begin{align*}
\text{while ( } a > 0 \text{ )} \\
b &= b \times a \\
a &= a - 1
\end{align*}
\]

Register Transfer Level

Gate Level

Layout
Designers began to use HDLs for higher-level verification and design exploration.

Algorithm:
```
while ( a > 0 )
    b = b * a
    a = a - 1
```

Register Transfer Level

Gate Level

Layout

Manual

Manual

Manual

Verification via Simulation

Verification via Simulation

Verification via Simulation

Manual
High-level algorithmic models act as a precise and executable specification.

Algorithm:

```plaintext
while ( a > 0 )
  b = b * a
  a = a - 1
```

Verification via Simulation:

Manual

Verify
via Simulation

Layout

Manual

Verify
via Simulation

Register Transfer Level

Manual

Verify
via Simulation

Gate Level

Manual

Verify
via Simulation

Manual
Once designs were written in HDLs tools, could be used for automatic translation.

\[
\text{while ( } a > 0 \text{ )} \\
b = b * a \\
a = a - 1
\]
Hardware Verification Languages

- A separate or embedded language that is meant purely for verification as opposed to simulation or synthesis
  - Includes high-level programming features to simplify writing test benches such as object-oriented constructs and random stimulus generation
  - Includes special language constructs for writing complex assertions
  - Example HVLs: e, OpenVera, PSL, SystemVerilog Verification Subset

Example SystemVerilog assertions

- Assert that the read enable and write enable signals are never both true:
  ```verilog
  assert ! (read_en && write_en);
  ```

- Assert that priority register in round-robin arbiter is one-hot:
  ```verilog
  assert property (@(posedge clk) $onehot(priority))
  ```

- Assert that acknowledge signal is true cycle after the request signal is true:
  ```verilog
  assert property (@(posedge clk) req |-> ##[1] ack);
  ```
Agenda

Evolution of Hardware Description Languages

Hardware Description Languages Across Stack

“High-Level” RTL with SystemVerilog

Guarded-Atomic Actions with Bluespec

System-Level Modeling with SystemC
HDLs Across The Computer Engineering Stack

- **Layout Level**
- **Circuit Level**
- **Gate Level**
- **Register Transfer Level**
- **Guarded Atomic Actions**
- **System Level**
- **Algorithm**

```plaintext
while (a>0) 
  b = b * a 
  a = a - 1
```

- **Modeling for Simulation**
  - **GDSII**
  - **MATLAB/C++**

- **Modeling for Synthesis**
  - Lower-Level: More Control, Less Productive
  - Higher-Level: Less Control, More Productive
**Circuit-Level Modeling with Spice**

- **CMOS NAND gate**
  - MP1 4 1 3 3 CMOS W=28.0U L=2.0U AS=252P AD=252P
  - MP2 4 2 3 3 CMOS W=28.0U L=2.0U AS=252P AD=252P
  - MN1 4 1 5 0 CMOSN W=10.0U L=2.0U AS=90P AD=90P
  - MN2 5 2 0 0 CMOSN W=10.0U L=2.0U AS=90P AD=90P

- **Input stimulus**
  - VINA 2 0 PULSE(0 5 100ns 5ns 5ns 100n 200ns)
  - VINB 1 0 PULSE(0 5 205ns 5ns 5ns 200n 400ns)
  - VDD 3 0 DC 5.0

```plaintext
while (a>0)
b = b * a
a = a - 1
```
Gate-Level Modeling with Verilog

module mux4( input  a, b, c, d, input [1:0] sel, output out );

wire [1:0] sel_b;
not not0( sel_b[0], sel[0] );
not not1( sel_b[1], sel[1] );
wire n0, n1, n2, n3;
and and0( n0, c, sel[1] );
and and1( n1, a, sel_b[1] );
and and2( n2, d, sel[1] );
and and3( n3, b, sel_b[1] );
wire x0, x1;
nor nor0( x0, n0, n1 );
nor nor1( x1, n2, n3 );
wire y0, y1;
or or0( y0, x0, sel[0] );
or or1( y1, x1, sel_b[0] );
nand nand0( out, y0, y1 );
endmodule


### Evolution of HDLs

- **HDLs Across Stack**
- "High-Level" RTL
- Guarded-Atomic Actions
- System-Level Modeling

---

**“Low-Level” RTL Modeling with Verilog**

---

```verilog
// Combinational Logic: Operand Muxes
wire[63:0] a_mux_out = (a_mux_sel == op_load) ? {32'b0, unsigned_a} : (a_mux_sel == op_next) ? a_shift_out : 64'bx;
wire[31:0] b_mux_out = (b_mux_sel == op_load) ? unsigned_b : (b_mux_sel == op_next) ? b_shift_out : 32'bx;

reg[4:0] counter_reg;
reg sign_reg;
reg[63:0] a_reg;
reg[31:0] b_reg;
reg[63:0] result_reg;
```

### Sequential State

```verilog
always @(posedge clk) begin
    if (sign_en) begin
        sign_reg <= sign_next;
    end
    if (result_en) begin
        result_reg <= result_mux_out;
    end
    counter_reg <= counter_mux_out;
    a_reg <= a_mux_out;
    b_reg <= b_mux_out;
end
```

---

- while (a>0)
  - b = b * a
  - a = a - 1
### Simulation vs. Synthesis Mismatch

#### Algorithm
```
while (a>0)
  b = b * a
  a = a - 1
```

#### Mux with assign statement
```
wire [3:0] out = ( sel == 0 ) ? a : b;
```

What happens if the `sel` signal contains an `X`?

#### Mux with always block
```
reg [3:0] out;

always @(*)
begin
  if ( sel == 0 )
    out = a;
  else
    out = b;
end
```
Higher-Level HDLs

How can we raise the level of abstraction to increase hardware design productivity?

- “High-Level” Register-Transfer-Level Modeling with SystemVerilog
- Guarded Atomic Actions with Bluespec
- System-Level Modeling with SystemC

while (a > 0)
  b = b * a
  a = a - 1
Agenda

Evolution of Hardware Description Languages

Hardware Description Languages Across Stack

“High-Level” RTL with SystemVerilog

Guarded-Atomic Actions with Bluespec

System-Level Modeling with SystemC
SystemVerilog: Struct and Union Types

### Declare JumpInstr structure

typedef struct packed {
    logic [7:0] opcode;
    logic [23:0] addr
} JumpInstr;

### Instantiate JumpInstr structure

JumpInstr instr;
instr.opcode = c_opcode_jump;
instr.addr = addr;

### Declare AddInstr structure

typedef struct packed {
    logic [7:0] opcode;
    logic [4:0] rd;
    logic [4:0] rt;
    logic [4:0] rs;
    logic [8:0] null;
} AddInstr;

### Declare Instr union

typedef union packed {
    JumpInstr jump;
    AddInstr add;
} Instr;

### Instantiate Instr union

Instr instr;
instr.jump.opcode = c_opcode_jump;
instr.jump.addr = addr;
SystemVerilog: Tagged Union Types

// Declare Instr w/ common opcode
typedef struct packed {
  logic [23:0] addr
} JumpInstrFields;

typedef struct packed {
  logic [4:0] rd;
  logic [4:0] rt;
  logic [4:0] rs;
  logic [8:0] null;
} AddInstrFields;

typedef union packed {
  JumpInstrFields jump;
  AddInstrFields add;
} InstrFields;

typedef struct packed {
  logic [7:0] opcode;
  InstrFields fields;
} Instr;

// Declared Instr tagged union
typedef union tagged packed {
  JumpInstrFields jump;
  AddInstrFields add;
} Instr;

// Instantiate Instr tagged union
Instr instr = tagged jump { addr: addr };

// Pattern matching
case ( instr ) matches
tagged add: cs={ sel_a, y };
tagged jump: cs={ sel_b, n };
endcase
SystemVerilog: Typed Ports and Type Parameters

// Structs, unions, tagged unions can be used as ports
module InstrDecodeTable
(input Instr instr, output ControlSigs cs)

// Use structure selectors to access instruction and control signal fields
endmodule

// Type parameter allows more expressive polymorphism
module Queue#(
    parameter type ItemType)
(input clk, reset
    input enq_val, enq_rdy,
    input ItemType enq_item,
    output deq_val, deq_rdy,
    output ItemType deq_bits,
)

// Use $bits for size of item
endmodule

// Instantiate polymorphic queue
Queue#(Instr) queue(...)

// High-Level RTL
• “High-Level” RTL
• Guarded-Atomic Actions
• System-Level Modeling
SystemVerilog: Port Bundle Interfaces

```vhdl
// Declare valrdy interface

interface ValRdyIfc;
logic val;
logic rdy;
logic [31:0] msg;
endinterface

// Using an interface

module Producer (ValRdyIfc.send_ifc send_ifc)
    input clk, reset,
    ValRdyIfc.send_ifc send_ifc
endmodule

// Instantiate and use interface

ValRdyIfc channel;
Producer producer(channel);
Consumer consumer(channel);
```
SystemVerilog: Method Interfaces

// Declare valrdy method interface

interface ValRdyIfc;
  logic val;
  logic rdy;
  logic [31:0] msg;

  // ...

  function send ( input logic [31:0] msg );
  // ...
  endfunction

  function is_send_done ( output logic done );
  // ...
  endfunction

endinterface

// Using an method interface

module Producer
  ( input clk, reset,
    ValRdyIfc.send_ifc send_ifc )

  // ...

  logic done;

  always @(posedge clk )
  begin
    send_ifc.send( msg );
    send_ifc.is_send_done( done );
    if ( done )
      ...
  end

endmodule
SystemVerilog: Interfaces
Evolution of Hardware Description Languages

Hardware Description Languages Across Stack

“High-Level” RTL with SystemVerilog

Guarded-Atomic Actions with Bluespec

System-Level Modeling with SystemC
Designers Usually Use Weak Interfaces

Example: Commercially available FIFO IP block

An error occurs if a push is attempted while the FIFO is full.

Thus, there is no conflict in a simultaneous push and pop when the FIFO is full. A simultaneous push and pop cannot occur when the FIFO is empty, since there is no pop data to prefetch. However, push data is captured in the FIFO.

A pop operation occurs when \texttt{pop\_req\_n} is asserted (LOW), as long as the FIFO is not empty. Asserting \texttt{pop\_req\_n} causes the internal read pointer to be incremented on the next rising edge of \texttt{clk}. Thus, the RAM read data must be captured on the \texttt{clk} following the assertion of \texttt{pop\_req\_n}.

\textit{These constraints are spread over many pages of the documentation...}

Adapted from [Arvind’11]
Expressing Hardware with Guarded Atomic Actions in Bluespec

- **Guarded rules**
  - Hardware expressed as collection of rules that execute atomically and in a well-defined serialized sequence
  - Allows thinking of pieces of the design in isolation
  - Compiler manages scheduling of rules to increase performance

- **Guarded method interfaces**
  - Formalizes composition
  - Compiler manages connectivity (muxing and control logic)

- **Powerful type and static elaboration facilities**
  - Significant amount of compile-time static checking
  - Permits parameterization of designs at all levels
Guarded Atomic Action Execution Model

- **Semantics**
  - Actions execute in a serialized order
  - Actions execute in isolation

- **Repeatedly**
  - Select a rule to execute (highly non-deterministic)
  - Compute the new state values
  - Update the state

- **Implementation concerns**
  - But doesn’t executing one rule at a time mean our implementation will be very slow?
  - Can we schedule multiple rules concurrently without violating one-rule-at-a-time semantics?

Work through extra notes ...
State and Rules Organized into Modules

- All state is explicit (no inferred latches or flip-flops)
- Behavior is expressed in terms of guarded rules within each module that atomically update state internal to that module
- Rules can manipulate state in other modules only via their guarded method interfaces

Adapted from [Arvind'11]
def gcd( x, y ):
    while True:
        if x > y:
            x, y = y, x
        elif y != 0:
            y = y - x
        else:
            return x

1. 25 15 swap
2. 15 25 sub
3. 15 10 swap
4. 10 15 sub
5. 10 5 swap
6. 5 10 sub
7. 5 5 sub
8. 5 0 return x
### GCD Implementation in Bluespec

**Explicit State**

```plaintext
module mkGCD (I_GCD);
    Reg#(Int#(32)) x <- mkRegU;
    Reg#(Int#(32)) y <- mkReg(0);
endmodule
```

**Internal Behavior**

```plaintext
rule swap ((x > y) && (y != 0));
    x <= y; y <= x;
endrule

rule sub ((x <= y) && (y != 0));
    y <= y - x;
endrule
```

**External Interface**

```plaintext
method Action start(Int#(32) a, Int#(32) b) if (y==0);
    x <= a; y <= b;
endmethod

method Int#(32) result() if (y==0);
    return x;
endmethod
endmodule
```

Adapted from [Arvind’11]
GCD Alternative Implementation

```verilog
module mkGCD (I_GCD);
    Reg#(Int#(32)) x <- mkRegU;
    Reg#(Int#(32)) y <- mkReg(0);
rule swap
    x <= y;
endrule
rule sub ((x <= y) && (y != 0));
    y <= y - x;
endrule
method Action start(Int#(32) a, Int#(32) b) if (y==0);
    x <= a; y <= b;
endmethod
method Int#(32) result() if (y==0);
    return x;
endmethod
endmodule
```

Combined Rule

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>op</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>25</td>
<td>15</td>
</tr>
<tr>
<td>2.</td>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td>3.</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>4.</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>5.</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

Adapted from [Arvind'11]
Evolution of HDLs

HDLs Across Stack

“High-Level” RTL

• **Guarded-Atomic Actions**

• System-Level Modeling

---

**Generated GCD Hardware: Interface**

- Module can easily be made polymorphic as in SystemVerilog
- Many different implementations can provide the same interface

Adapted from [Arvind’11]
Generated GCD Hardware: Rules

\[
\begin{align*}
\text{rule} & \quad \text{swap} \quad ((x>y) \land (y\neq 0)); \\
& \quad x \leq y; \quad y \leq x; \quad \text{endrule} \\
\text{rule} & \quad \text{subtract} \quad ((x\leq y) \land (y\neq 0)); \\
& \quad y \leq y - x; \quad \text{endrule}
\end{align*}
\]

\[
\begin{align*}
\text{x\_en} &= \quad \text{swap}\? \\
\text{y\_en} &= \quad \text{swap}\? \ OR \ \text{subtract}\?
\end{align*}
\]

Adapted from [Arvind’11]
Generated GCD Hardware: Rules and Methods

\[
x_{en} = \text{swap?} \quad \text{OR} \quad \text{start}_{en}
\]
\[
y_{en} = \text{swap?} \quad \text{OR} \quad \text{subtract?} \quad \text{OR} \quad \text{start}_{en}
\]
\[
\text{rdy} = (y==0)
\]

Adapted from [Arvind’11]
A rule may be decomposed into two parts $\pi(s)$ and $\delta(s)$ such that

$$s_{\text{next}} = \text{if } \pi(s) \text{ then } \delta(s) \text{ else } s$$

$\pi(s)$ is the condition (predicate) of the rule, a.k.a. the “CAN_FIRE” signal of the rule. $\pi$ is a conjunction of explicit and implicit conditions.

$\delta(s)$ is the “state transformation” function, i.e., computes the next-state values from the current state values.

Adapted from [Arvind’11]
Compiling a Rule

\texttt{rule} r (f.first() > 0) ;
\texttt{x <= x + 1 ; f.deq ();
endrule}

\textbf{current state}

\textit{\pi} = enabling condition
\textit{\delta} = action signals & values
Combining State Updates (strawman)

\[\pi \text{'s from the rules that update } R\]

\[\delta \text{'s from the rules that update } R\]

What if more than one rule is enabled?
Combining State Updates

Scheduler ensures that at most one $\phi_i$ is true

Adapted from [Arvind’11]
Scheduling and Control Logic

Compiler synthesizes a scheduler such that at any given time $\phi$’s for only non-conflicting rules are true.

Adapted from [Arvind'11]
Agenda

Evolution of Hardware Description Languages

Hardware Description Languages Across Stack

“High-Level” RTL with SystemVerilog

Guarded-Atomic Actions with Bluespec

System-Level Modeling with SystemC
System-Level Modeling

Separate computation/storage from communication

A. Specification model
B. Component-assembly model
C. Bus-arbitration model
D. Bus-functional model
E. Cycle-accurate computation model
F. Implementation model

Adapted from [Cai'03]


**Specification Model**

- **Describes system functionality without any implementation details**
- **Computation modeled as abstract concurrent processes**
- **Communication modeled with standard software variables**

```
B1
v1 = a*a;

B2
v2 = v1 + b*b;

B3
v3 = v1 - b*b;

B4
v4 = v2 + v3;
c = sequ(v4);
```

Adapted from [Cai'03]
TLM: Component-Assembly Model

- Approximately estimate execution time of PEs using first-order models
- Explicitly capture process-to-PE mapping
- Use dedicated point-to-point channels to interconnect computation and storage PEs
- No modeling of bus or protocol details

Adapted from [Cai’03]
TLM: Bus-Arbitration Model

- Approximately estimate execution time of PEs using first-order models
- Explicitly capture channel-to-bus mapping
- Still communicate through abstract channels, but also estimate bus performance with first-order arbitration models

Adapted from [Cai’03]

Evolution of HDLs HDLs Across Stack “High-Level” RTL Guarded-Atomic Actions • System-Level Modeling •

ECE 5745 T01: Hardware Description Languages 45 / 55
TLM: Bus-Functional Model

Approximately estimate execution time of PEs using first-order models

Cycle-accurate RTL model of bus protocol

Adapted from [Cai'03]
TLM: Cycle-Accurate Computation Model

Cycle-accurate RTL model of (some) PEs

Adapters interface lower-level RTL interface to higher-level bus-arbitration model

Still communicate through abstract channels, but also estimate bus performance with first-order arbitration models

Adapted from [Cai'03]
Register-Transfer-Level Model

Cycle-accurate RTL models of both computation/storage and communication
SystemC Framework

Methodology-Specific Libraries
Master/Slave Lib, Verification Lib, Static Dataflow

Primitive Channels
Signal, Mutex, Semaphore, FIFO

Core Language
- Modules
- Ports
- Processes
- Interfaces
- Channels

Data Types
- 4-Valued Logic Types
- 4-Valued Logic Vectors
- Bits and Bit Vectors
- Fixed-Point Types
- C++ User-Defined Types

Event-Driven Simulation Kernel

C++ Language Standard
SystemC Modules, Processes, Channels

- Separate computation from communication
  - Computation: implemented with Processes in Modules
  - Communication: implemented in Channels

- Interface method calls
  - Collection of a fixed set of communication Methods is called an Interface
  - Channels implement one or more Interfaces
  - Modules can be connected via their Ports to those Channels which implement the corresponding Interface

Adapted from [Moondanos'04]
SystemC Producer-Consumer Example

```c
struct Producer : public sc_module
{
    // Ports
    sc_out< bool > clk;
    sc_out< int >  value;

    SC_HAS_PROCESS(Producer);
    Producer( sc_module_name name ) : sc_module(name)
    {
        // Declares compute as a thread
        SC_THREAD(compute);
    }

    void compute()
    {
        for ( int i = 0; i < 10; ++i ) {
            clk.write(false);   // toggle clk
            wait( 5, SC_NS );   // wait for 5 nanoseconds
            clk.write(true);    // toggle clk
            value.write(i);     // write value to channel
            wait( 5, SC_NS );   // wait for 5 nanoseconds
        }
    }
};
```
SystemC Producer-Consumer Example

```
struct Consumer : public sc_module {
    // Ports
    sc_in<bool> clk;
    sc_in<int>  value;

    SC_HAS_PROCESS(Consumer);
    Consumer( sc_module_name name ) : sc_module(name) {
        // Declares receive() as a process triggered
        // on clk value changes
        SC_METHOD(receive);
        sensitive << clk;
    }

    void receive() {
        // If clk is changing from false to true
        if ( clk.posedge() )
            std::cout << 'Received:' << ' ' << value.read() << std::endl;
    }
};
```
SystemC System

Adapted from [Moondanos'04]
Take-Away Points

▶ Hardware description languages involve a four-way tension
  ▶ Low-level languages offer more control but less productivity
  ▶ High-level languages offer less control but more productivity
  ▶ Simulation features for modeling function and test harnesses
  ▶ Synthesis features for modeling actual hardware

▶ Hardware description languages are (slowly) moving towards including higher abstractions to improve productivity such as
  ▶ Types, Interfaces (SystemVerilog)
  ▶ Guarded Atomic Rules, Guarded Method Interfaces (Bluespec)
  ▶ Transaction-Level Modeling (SystemC)

while (a>0)
b = b * a
a = a - 1
Acknowledgments

