ECE 747
Digital Signal Processing
Architecture

ESL Design Methodologies

Spring 2006
W. Rhett Davis
NC State University
What is ESL Design?
ESL is supposed to boost productivity...

To Design, Implement, Verify 10M transistors

Staff Months
62.5
125
625
6250
62,500

Implementations here are often not good enough

Because implementations here are inferior/ unpredictable

Kurt Keutzer & Richard Newton

W. Rhett Davis  NC State University  ECE 747  Spring 2006  Slide 3
... but why is ESL still “on the horizon”? 

Circuit Designers vs. CAD Developers

“CAD tools aren’t any more advanced today than they were 10 years ago.”

“CAD tools would work very well if circuit designers would use them properly.”

Translation: CAD Developers Suck

Translation: Circuit Designers Suck

Can we find an answer that doesn’t insult anyone?
Circuit Designers

How Circuit Designers see CAD Developers

How Circuit Designers see themselves
CAD Developers

How CAD Developers see Circuit Designers

How CAD Developers see themselves
What is ESL Design?

- According to the ITRS Electronic System Level (ESL) Design...
  - orthogonalizes System-Level (Behavior / Algorithm) and Platform-Level (Implementation) Concerns
  - uses a higher-level of abstraction than the Register Transfer Level (RTL)
  - exploits reuse
  - expands the scope of hardware design to include software

- Requirement for a good ESL Methodology
  - Less design-time!
Ways to Reduce Design Time

- Reduce the amount of thrashing in the design process
  - Top-down Design

- Reduce the time needed to learn a hardware-description language
  - C-based Design
  - Signal Flow-Graph Design

- Reduce the number of design decisions that need to be made
  - Behavioral Synthesis / Design-Space Exploration

- Reduce the number of times design decisions must be expressed
  - Design-Flow Automation

- Reduce the amount of code that needs to be written
  - HW-SW Co-design Languages

- Reduce the amount of time spent incorporating IP
  - Interface Synthesis
  - ASIPs
A Simple Design Example

- Target Clock Rate: 25 MHz (40 ns Period)
- Delay = 30 ns
- Delay = 15 ns
Timing Constraint Violations

- Connecting blocks would violate maximum clock period
- Pipeline register alleviates problem
- Block modification required

Total Delay = 45 ns
Loop Retiming

- When adding registers to a loop, the entire loop must be retimed
- Multiple blocks might need to be modified
Problem with RTL Methodology

Thrashing: A “Chain Reaction” of Modifications

- Changing one block leads to a cascade of changes
- Design-productivity drops significantly
Solution: Top-Down Design

Set constraints at the highest level so that there is no thrashing.

New Problems: Potential Loss of Performance and No Way to Regain it!
Constraints that are too aggressive cause thrashing between levels of abstraction!
Ways to Reduce Design Time

- Reduce the amount of thrashing in the design process
  » Top-down Design

- Reduce the time needed to learn a hardware-description language
  » C-based Design
  » Signal Flow-Graph Design

- Reduce the number of design decisions that need to be made
  » Behavioral Synthesis / Design-Space Exploration

- Reduce the number of times design decisions must be expressed
  » Design-Flow Automation

- Reduce the amount of code that needs to be written
  » HW-SW Co-design Languages

- Reduce the amount of time spent incorporating IP
  » Interface Synthesis
  » ASIPs
Handel-C

- An assignment statement takes exactly one clock cycle to execute.

- Operators for Bit Manipulation
  - \( z = x \ll 2; \) // Take least significant bits
  - \( z = y \gg 2; \) // Drop least significant bits
  - \( z = x @ y; \) // Concatination
  - \( z = x[3]; \) // Bit selection
  - \( z = y[2:3]; \) // Bus selection
  - \( z = \text{width}(x); \) // Width of expression

- par syntax allows for specification of parallel execution.

- See [http://www.celoxica.com](http://www.celoxica.com) for details
Handel-C

- Example Pipeline – temp1, temp2, and x are each pipeline stages

```c
void main(void)
{
    unsigned 8 x, y;
    unsigned 5 temp1;
    unsigned 4 temp2;
    ...
    temp1 = (0@(x <- 4)) + (0@(y <- 4));
    temp2 = (x \ 4) + (y \ 4);
    x = (temp2 + (0@temp1[4])) @ temp1[3:0];
}
```

- Example of parallel hardware

```c
{
    temp1=(0@(x<-4))+(0@(y<-4));
    temp2=(x\4)+(y\4);
}
x=(temp2+(0@temp1[4]))@temp1[3:0];
```
Design With Signal Flow Graphs

- **Examples:**
  - SSAFT
  - Xilinx System Generator

- **Philosophy:**
  - Algorithm designers like Signal Flow Graphs, so use the representation that is most convenient for them.

```
Multiply / Accumulate
```

W. Rhett Davis  NC State University  ECE 747  Spring 2006  Slide 17
Problems with this approach

- Doesn’t scale well to large design problems
- Control can be very difficult to design
- Makes Verification Harder
Why is Verification Harder?

- Always difficult to sift through computer-generated code
- RTL Behavior doesn’t necessarily match the Signal-Flow graph behavior

We chose a discrete-time computation model
Ways to Reduce Design Time

- Reduce the amount of thrashing in the design process
  - Top-down Design

- Reduce the time needed to learn a hardware-description language
  - C-based Design
  - Signal Flow-Graph Design

- Reduce the number of design decisions that need to be made
  - Behavioral Synthesis / Design-Space Exploration

- Reduce the number of times design decisions must be expressed
  - Design-Flow Automation

- Reduce the amount of code that needs to be written
  - HW-SW Co-design Languages

- Reduce the amount of time spent incorporating IP
  - Interface Synthesis
  - ASIPs
What is the right level of abstraction?

- Many people would like to use a single-thread execution behavior.
- How would you code this in Verilog?

```verbatim
sum=0;
for (i=0; i<=3; i++) {
    prod=a[i]*b[i];
    sum=sum+prod;
}
```

Dot-Product
PICO: Program-In, Chip-Out

- Compile the program and try mapping it to a series of well-known architectures
- Choose the best architecture from the Pareto-Optimal Designs
- Implement that architecture on a “Non-Programmable Accelerator” NPA for a VLIW micro-processor
- NPA is an array of processing elements in a reconfigurable interconnect fabric
PICO: Program-In, Chip-Out

Source: Kathail 2002, IEEE Computer
Related Approach: Behavioral Synthesis

VHDL

```vhdl
wait until clk'event and clk='1';
a := input_port1;
wait until clk'event and clk='1';
b := input_port2;
wait until clk'event and clk='1';
c := input_port3;
wait until clk'event and clk='1';
d := input_port4;
wait until clk'event and clk='1';
real <= a*c - b*d;
imag <= a*d + b*c;
wait until clk'event and clk='1';
```

- Designers specify scheduling constraints instead of cycle-to-cycle behavior
- Behavioral Compiler schedules and retimes automatically
Behavioral Synthesis

**VHDL**

```vhdl
wait until clk'event and clk='1';
a := input_port1;
wait until clk'event and clk='1';
b := input_port2;
wait until clk'event and clk='1';
c := input_port3;
wait until clk'event and clk='1';
d := input_port4;
wait until clk'event and clk='1';
real <= a*c - b*d;
imag <= a*d + b*c;
wait until clk'event and clk='1';
```

- Designers specify scheduling constraints instead of cycle-to-cycle behavior
- Behavioral Compiler schedules and retimes automatically
Problem with Behavioral Synthesis

Many architectures can map to the same behavior

but

some behaviors have more efficient architectures than others.
Behavioral Compiler Example

- Code entire module in a single “always” loop
- Nest loops as deeply as you like
- Make sure that there is at least one “@ (posedge CLK)” statement in every loop

```verbatim
module dotprod (A, B, CLK, Z);
  input [3:0] A, B;
  input CLK;
  output [7:0] Z;

  reg [7:0] Z;

  always begin : behav
    reg [7:0] P, S;
    reg [1:0] i;
    S = 0;
    for (i=0; i <= 3; i=i+1)
      begin : loop1
        P = A * B;
        S = S + P;
        @(posedge CLK);
      end // loop1
    Z <= S;
    @(posedge CLK);
  end //behav

endmodule
```
Behavioral Compiler Synthesis Flow

analyze -schedule -format verilog dotprod.v
elaborate -schedule dotprod
create_clock CLK -p 10000
bc_time_design
schedule -io_mode superstate_fixed -effort medium
report_schedule -var -op -summ -a
compile

- analyze -schedule and elaborate -schedule tell Design Compiler to use Behavioral Synthesis
- bc_time_design checks the data-dependencies and estimates the amount of time needed for each one
- schedule allocates resources and generates control logic
**Result with 10 ns period**

<table>
<thead>
<tr>
<th>cycle</th>
<th>loop</th>
<th>A</th>
<th>B</th>
<th>r10</th>
<th>r8</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>.L0</td>
<td>.R14</td>
<td>.R14d</td>
<td></td>
<td>.o14</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>.R14a</td>
<td>.R14e</td>
<td>.o15</td>
<td>.o14a</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>.R14b</td>
<td>.R14f</td>
<td>.o15a</td>
<td>.o14b</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>.R14c</td>
<td>.R14g</td>
<td>.o15b</td>
<td>.o14c</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>.L2</td>
<td></td>
<td>.W18</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>.L1</td>
<td></td>
<td>.L1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **NOTE:** Extra cycle of latency
Result with 5 ns period

- Same Verilog code, 1 more cycle of latency
- Multi-cycle paths for multiplier
Problems with Behavioral Synthesis

- Makes verification much more difficult (no “golden” simulation vectors)
- In order to be effective, the entire chip must be in one “always” block
  - No way to add hierarchy to decrease execution time
- Little variety in architectures (always a minimal set of function units with multi-cycle paths)
- Designers don’t want to give up control
Ways to Reduce Design Time

- Reduce the amount of thrashing in the design process
  » Top-down Design

- Reduce the time needed to learn a hardware-description language
  » C-based Design
  » Signal Flow-Graph Design

- Reduce the number of design decisions that need to be made
  » Behavioral Synthesis / Design-Space Exploration

- Reduce the number of times design decisions must be expressed
  » Design-Flow Automation

- Reduce the amount of code that needs to be written
  » HW-SW Co-design Languages

- Reduce the amount of time spent incorporating IP
  » Interface Synthesis
  » ASIPs
Standard DSP-ASIC Design Flow

Optimization:
- BER vs. SNR, no. of operations
- Fixed-point types, block-level structure
- Critical-path delay, area, power (w/o knowledge of wires)
- Groupings, Floorplan

Problems:
- Three translations of design data
- Optimization more tightly constrained at each stage
- Uncontrolled looping when pipeline stalls

Prohibitively Long Design Time for Efficient Architectures
Chip-In-A-Day Flow

- Refine design data rather than translate
- Encourages iterations of layout
- Depends on fast automation
Capturing Design Decisions

Categories:
- **Function** - basic input-output behavior
- **Signal** - physical signals and types
- **Circuit** - transistors
- **Floorplan** - physical positions

Layout and performance estimates in a day
“Push-Button” Automation

- Automation similar to MAKE
  - No decisions made after button is pressed
- No translation of design data
  - No decisions expressed more than once
Example: Multiple Graphs

- Early versions inferred pads from ports
- Another copy of design was maintained to examine internal signals (decisions expressed twice)
- Later version allowed “SimOnly” ports
Simplified View of the Flow

- Dataflow graph
- Generate
- Netlist
- Floorplan
- Merge
- AutoLayout
- Route
- Layout

New Software:
- Generation of netlists from a dataflow graph
- Merging of floorplan from last iteration
- Automatic routing and performance analysis
- Automation of flow as a dependency graph (UNIX MAKE program)

Macro library
Why Simulink?

- Relatively inexpensive and widely available
- Simulink is an easier sell to algorithm developers
- Closely integrated with popular algorithm design tool Matlab

Time-Multiplexed FIR Filter
Modeling Datapath Logic

- **Discrete-Time** (cycle accurate)

- **Fixed-Point Types** (bit true)

- Completely specify function and signal decisions

- No need for RTL

### Multiply / Accumulate
Modeling Control Logic

- Extended finite state-machine editor
- Co-simulation with dataflow graph
- New Software: Stateflow-VHDL translator
- No need for RTL

Address Generator / MAC Reset

init entry: addr=0; wen=1;

[addr==15]

incr during: addr++; reset_acc=0;

restart entry: addr=0; wen=0; reset_acc=1;
Specifying Circuit Decisions

Macro choices embedded in dataflow graph
Capturing Floorplan Decisions

Parallel Pipelined FIR Filter

- Commercial physical design tools used
- Instance names in floorplan match dataflow graph
- Placements merged on each iteration
- New blocks show up as unplaced instances
Problems with this Approach

- The Design Flow is married to the Design!
  - Difficult to reuse parts of the flow without modification
  - Difficult to share the flow without violating Intellectual Property Agreements
Ways to Reduce Design Time

- Reduce the amount of thrashing in the design process
  » Top-down Design

- Reduce the time needed to learn a hardware-description language
  » C-based Design
  » Signal Flow-Graph Design

- Reduce the number of design decisions that need to be made
  » Behavioral Synthesis / Design-Space Exploration

- Reduce the number of times design decisions must be expressed
  » Design-Flow Automation

- Reduce the amount of code that needs to be written
  » HW-SW Co-design Languages

- Reduce the amount of time spent incorporating IP
  » Interface Synthesis
  » ASIPs
GEZEL

- Focuses on the interface between hardware & software.

- Uses a finite-state-machine plus datapath description (.fdl files)
  - sfg – keyword for signal-flow-graph
  - fsm – keyword for finite state machine

- See [http://rijndael.ece.vt.edu/gezel2](http://rijndael.ece.vt.edu/gezel2)

- Used on the ThumbPod Project [http://www.thumbpod.com](http://www.thumbpod.com)

- Grew out of the OCAPI design and modelling framework from IMEC [http://www.imec.be/design/ocapi](http://www.imec.be/design/ocapi)
GEZEL2 LFSR Example

```verilog
dp lfsr(in seed : ns(5);
in load : ns(1);
out b : ns(1)) {
  reg shft : ns(5);
sig shft_new : ns(5);
always {
  shft_new = (shft << 1) | (shft[1] ^ shft[4]);
  shft = load ? seed : shft_new;
  b = shft[4];
}
}

// testbench
dp lfsr_tb(out seed : ns(5);
  out load : ns(1);
in b : ns(1)) {
  sfg init { seed = 0x1F; load = 1; }
  sfg run { seed = 0; load = 0; }
  sfg show { $display("b= ", b); }
}

fsm hlfsr_tb(lfsr_tb) {
  initial s0;
  state s1;
  @s0 (init) -> s1;
  @s1 (run, show) -> s1;
}

dp top {
  sig seed : ns(5);
  sig load, b : ns(1);
  use lfsr(seed, load, b);
  use lfsr_tb(seed, load, b);
}

system S {
  top;
}
```
GEZEL2 Memory-Mapped IO

euclid.fdl

```salary
ipblock myarm {
  iptype "armsystem";
  ipparm "exec=euclid";
}

// data input
ipblock b_datain(out data : ns(32)) {
  iptype "armsystemsource";
  ipparm "core=myarm";
  ipparm "address=0x80000008";
}

// data output
ipblock b_dataout(in data : ns(32)) {
  iptype "armsystemsink";
  ipparm "core=myarm";
  ipparm "address=0x80000004";
}

dp sys {
  sig ins, din, dout : ns(32);
  use myarm;
  use encap_euclid(ins, din, dout);
  use b_datain(din);
  use b_dataout(dout);
}
```
euclid.c

```salary
int gcd(int m, int n) {
    volatile unsigned int *din =
        (volatile unsigned int *) 0x80000008;
    volatile unsigned int *dout =
        (volatile unsigned int *) 0x80000004;
    int r;
    ...
    *din = m;
    ...
    while (! (r=*dout) );
    ...
```
Ways to Reduce Design Time

- Reduce the amount of thrashing in the design process
  - Top-down Design

- Reduce the time needed to learn a hardware-description language
  - C-based Design
  - Signal Flow-Graph Design

- Reduce the number of design decisions that need to be made
  - Behavioral Synthesis / Design-Space Exploration

- Reduce the number of times design decisions must be expressed
  - Design-Flow Automation

- Reduce the amount of code that needs to be written
  - HW-SW Co-design Languages

- Reduce the amount of time spent incorporating IP
  - Interface Synthesis
  - ASIPs
Why use SystemC?

• Because software and hardware are designed in such different ways, questions arise during the design process that are difficult to answer:
  » Will key sections be able to process data fast enough?
  » Will the chosen architecture implement the designed algorithms with sufficient accuracy to reproduce the performance obtained in MATLAB?
  » Have the control and configuration interfaces been specified in enough detail to enable the software to be developed?
  » *SystemC: Methodologies and Applications* Mueller, Rosenstiel, and Ruf 2003
Tools for Estimating Performance

- Xilinx FPGAs
  - ISE WebPACK™ – free download (with registration) from http://www.xilinx.com

- TI Programmable DSPs
  - Code Composer Studio IDE™ – free 120 day evaluation from http://www.ti.com