FPGAs: Verilog
Sequence Alignment (maybe)
Chris Rossbach
cs378 Fall 2018
11/5/2018
Outline for Today

• Questions?
• Administrivia
  • Re: Exams
  • Keep thinking about projects!
  • Website updates
• Agenda
  • FPGAs: POTPOURRI of things you need to know
  • NW

Acknowledgements/References:
• https://s3-us-west-2.amazonaws.com/cse291personalgenomics/Lectures2017/Lecture12_AlignmentVariantCalling.pptx
• https://web.stanford.edu/~jurafsky/slides/2_EditDistance.pptx
• https://moodle.med.lu.se/pluginfile.php/45044/mod_resource/content/0/sequence_alignment_2015.pptx
• http://www.cbs.dtu.dk/phdcourse/cookbooks/PairwiseAlignmentPhD2.ppt
• http://cwcserv.ucsd.edu/~billin/classes/ECE111/lectures/Lecture1.pptx
• http://www.cs.unc.edu/~montek/teaching/Comp541-Fall16/VerilogPrimer.pptx
• Evita_verilog Tutorial, www.aldec.com
• http://wwwasic-worldcomverilog/
Faux Quiz Questions

• Why/when might one prefer an FPGA over an ASIC, CPU, or GPU?
• Define CLB, BRAM, and LUT. What role do these things play in FPGA programming?
• What is the difference between blocking and non-blocking assignment in Verilog?
• What is the difference between structural and behavioral modeling?
• How is synthesizable Verilog different from un-synthesizable? Give an example of each?
• What is discrete event simulation?
Review: FPGA Design/Build Cycle

- HW design in Verilog/VHDL
- Behavioral modeling + some structural elements
- Simulate to check functionality
- Synthesis → netlist generated
- Static analysis to check timing
Verilog

• Originally: modeling language for event-driven digital logic simulator
• Later: specification language for logic synthesis
• Consequence:
  • Combines structural and behavioral modeling styles
Components of Verilog

• Concurrent, event-triggered processes (behavioral)
  • *Initial* and *Always* blocks
  • Imperative code $\rightarrow$ standard data manipulation (assign, if-then, case)
  • Processes run until triggering event (or #delay expire)

• Structure
  • Verilog program builds from modules with I/O interfaces
  • Modules may contain instances of other modules
  • Modules contain local signals, etc.
  • Module configuration is static and all run concurrently
Discrete-event Simulation

- Key idea: *only* do work when something changes
- Core data structure: *event queue*
  - Contains events labeled with the target simulated time
- Algorithmic idea:
  - Execute every event for current simulated time
  - May change system state and may schedule events in the future (or now)
  - No events left at current time $\rightarrow$ advance simulated time (next event in Q)
Two Main Data Types

• Nets represent connections between things
  • Do not hold their value
  • Take their value from a driver such as a gate or other module
  • Cannot be assigned in an initial or always block

• Regs represent data storage
  • Behave exactly like memory in a computer
  • Hold their value until explicitly assigned in an initial or always block
  • Model latches, flip-flops, etc., but do not correspond exactly
  • *Shared variables*
    • Similar known shared state issues
Four-valued Data and Logic

Nets and regs hold *four-valued* data
- 0, 1 → Umm...
- Z
  - Output for undriven tri-state (hi-Z)
  - Nothing is setting a wire’s value
- X
  - Simulator can’t decide the value
  - Initial state of registers
  - Wire driven to 0 and 1 simultaneously
  - Output of gate with Z inputs

Data representation
- Binary → 6’b100101
- Hex → 6’h25

Logical operators work on three-valued logic

<table>
<thead>
<tr>
<th>Input</th>
<th>0</th>
<th>1</th>
<th>X</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Z</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Output 0 if one input is 0
Output X if inputs are junk
Structural Modeling

• Specification
  • Netlist: gates and connections
  • Primitives/components (e.g. logic gates)
  • Connected by wires
• Easy to translate to physical circuit

module max_4_co_1 (Out, In0, In1, In2, In3, Sel0, Sel1);
  output Out;
  input In0, In1, In2, In3, Sel0, Sel1;

  wire NotSel0, NotSel1;
  wire Y0, Y1, Y2, Y3;

  net (NotSel0, Sel0);
  net (NotSel1, Sel1);
  and (Y0, In0, NotSel1, NotSel0);
  and (Y1, In1, NotSel1, Sel0);
  and (Y2, In2, Sel0, NotSel1);
  and (Y3, In3, Sel1, Sel0);
  or (Out, Y0, Y1, Y2, Y3);
endmodule
Dataflow Modeling

- Specification
  - Components (similar to logical equations)
  - Connected by wires
- Easy to translate to structure, then to physical circuit
Behavioral Modeling

- Specification
  - In terms of expected behavior
  - Closest to natural language
  - Most difficult to synthesize

- Easier for testbenches
- Easier for abstract models of circuits
  - Simulates faster
  - Provides sequencing

```verilog
classic_module: max_4_to_1 #Out, In0, In1, In2, In3, Sel1, Sel0;  output Out;  input In0, In1, In2, In3, Sel0, Sel1;  reg Out;  always #(Sel1 or Sel0 or In0 or In1 or In2 or In3) begin  case ((Sel1, Sel0))  2'00 : Out = In0;  2'01 : Out = In1;  2'10 : Out = In2;  2'11 : Out = In3;  default : Out = 1'b0;  endcase  end
endmodule
```
Signals

- Nets
  - Physical connection between hardware elements
- Registers
  - Store value even if disconnected
Nets

- wire/tri
- wand/triand
- wor/trior

- Force synthesis to insert gates
  - (e.g. AND, OR)

```
module comparatorwithwor(
  input x,
  input y,
  output z
);
wor p;
assign p = x&y;
assign p = ~x & ~y;
assign z = p;
endmodule
```

```
module comparator(
  input x,
  input y,
  output z
);
wire p,q;
assign p = x&y;
assign q = ~x & ~y;
assign z = p | q;
endmodule
```
Ports and Registered Output

Output ports can be type register
- Add reg type to declaration
- *Output holds state*
Examples of Nets and Registers

Wires and registers can be bits, vectors, and arrays

wire a; // Simple wire
tri [15:0] dbus; // 16-bit tristate bus
tri #(5,4,8) b; // Wire with delay
reg [-1:4] vec; // Six-bit register
trireg (small) q; // Wire stores a small charge
integer imem[0:1023]; // Array of 1024 integers
reg [31:0] dcache[0:63]; // A 32-bit memory
Continuous Assignment

• Another way to describe combinational function
• Convenient for logical or datapath specifications

wire [8:0] sum;
wire [7:0] a, b;
wire carryin;

assign sum = a + b + carryin;

Define bus widths

• Continuous/"blocking" assignment: sets the value of sum to be a+b+carryin
• Recomputed when a, b, or carryin changes
Behavioral Modeling
Initial and Always Blocks

- Basic components for behavioral modeling

**initial**
```
begin
  ... imperative statements ...
end
```
- Runs when simulation starts
- Terminates when control reaches the end
- Good for providing stimulus
- Not synthesizable
- Great for debugging

**always**
```
begin
  ... imperative statements ...
end
```
- Runs when simulation starts
- Restarts when control reaches the end
- Good for modeling/specifying hardware
- Synthesizable
- Workhorse of sequential logic
Initial and Always

• Run until they encounter a delay

initial begin
  #10 a = 1; b = 0;
  #10 a = 0; b = 1;
end

• or a wait for an event

always @(posedge clk) q = d;
always begin wait(i); a = 0; wait(~i); a = 1; end
Procedural Assignment

• Inside an initial or always block:

```
sum = a + b + cin;
```

• Just like in C:
  • RHS evaluated
  • assigned to LHS
  • before next statement executes

• RHS may contain wires and regs
  • Two possible sources for data

• LHS must be a reg
  • Primitives or cont. assignment may set wire values
Imperative Statements

if (select == 1) y = a;  
else y = b;

case (op)
   2’b00: y = a + b;
   2’b01: y = a – b;
   2’b10: y = a ^ b;
   default: y = ‘hxxxx;
endcase
For and While Loops

- Increasing sequence of values on an output

```verilog
reg [3:0] i, output;

for ( i = 0 ; i <= 15 ; i = i + 1 ) begin
  output = i;
  #10;
end

reg [3:0] i, output;
i = 0;
while (i <= 15) begin
  output = i;
  #10 i = i + 1;
end
```
A Flip-Flop With Always

Edge-sensitive flip-flop

reg q;

always @(posedge clk)
    q = d;

• q = d assignment
  • runs when clock rises
  • exactly the behavior you expect
Blocking vs. Nonblocking

• Verilog has two types of procedural assignment

• Fundamental problem:
  • In a synchronous system, all flip-flops sample simultaneously
  • In Verilog, always @(posedge clk) blocks run in some undefined sequence
A Shift Register

*aka Blocking vs Non-blocking assignment*

```verilog
reg d1, d2, d3, d4;

always @(posedge clk) d2 = d1;
always @(posedge clk) d3 = d2;
always @(posedge clk) d4 = d3;
```

• These run in some order, but you don’t know which
• So...*might* not work as you’d expect
Non-blocking Assignments

reg d1, d2, d3, d4;

always @(posedge clk) d2 <= d1;
always @(posedge clk) d3 <= d2;
always @(posedge clk) d4 <= d3;

Nonblocking rule:
RHS evaluated when assignment runs

LHS updated only after all events for the current instant have run

• Blocking vs. Non-blocking: misnomer
• prefer “continuous” to “blocking”
• Guideline: blocking for combinational
• Guideline: non-blocking for sequential
Non-blocking Behavior

• A sequence of nonblocking assignments don’t communicate

a = 1;
b = a;
c = b;

Blocking assignment:
a = b = c = 1

a <= 1;
b <= a;
c <= b;

Nonblocking assignment:
a = 1
b = old value of a
c = old value of b
Dirty/tricky question:
which assignment type yields a correct shift register?

```verilog
reg d1, d2, d3, d4;

always @(posedge clk) begin
    d2 = d1;
    d3 = d2;
    d4 = d3;
end
```

Should  \( \text{op} \) be = or <= ?
Implementation: Building FSMs

• Many ways to do it
• Define the next-state logic combinationally
  • define the state-holding latches explicitly
• Define the behavior in a single always @(posedge clk) block
• Define behavior per signal in different @(posedge clk) blocks
• Variations on these themes
module FSM(o, a, b, reset);
output o;
reg o;
input a, b, reset;
reg [1:0] state, nextState;
always @(a or b or state)
case (state)
  2'b00: begin
    nextState = a ? 2'b00 : 2'b01;
    o = a & b;
  end
  2'b01: begin nextState = 2'b10; o = 0; end
endcase

Combinational block must be sensitive to any change on any of its inputs (Implies state-holding elements otherwise)
FSM with Combinational Logic

module FSM(o, a, b, reset);
...

always @(posedge clk or reset)
  if (reset)
    state <= 2'b00;
  else
    state <= nextState;

Latch implied by sensitivity to the clock or reset only
FSM from Combinational Logic

always @(a or b or state)
case (state)
  2’b00: begin
    nextState = a ? 2’b00 : 2’b01;
    o = a & b;
  end
  2’b01: begin nextState = 2’b10; o = 0; end
endcase

always @(posedge clk or reset)
if (reset)
  state <= 2’b00;
else
  state <= nextState;
module FSM(o, a, b);
output o; reg o;
input a, b;
reg [1:0] state;

always @(posedge clk or reset)
if (reset) state <= 2'b00;
else case (state)
  2'b00: begin
    state <= a ? 2'b00 : 2'b01;
    o <= a & b;
  end
  2'b01: begin state <= 2'b10; o <= 0; end
endcase

Outputs are latched
Inputs only sampled at clock edges
Nonblocking assignments used throughout.
RHS refers to values calculated in previous clock cycle
Parameters

- `localparam` keyword

```plaintext
localparam state1 = 4'b0001,
                 state2 = 4'b0010,
                 state3 = 4'b0100,
                 state4 = 4'b1000;

localparam A = 2'b00,
               G = 2'b01,
               C = 2'b10,
               T = 4'b11;
```
Operations for HDL simulation/build

• Compilation/Parsing

• Elaboration
  • Binding modules to instances
  • Build hierarchy
  • Compute parameter values
  • Resolve hierarchical names
  • Establish net connectivity

• ...(simulate, place/route, etc)
Generate Block

- Dynamically generate Verilog code at elaboration time

- Usage:
  - Parameterize modules when the parameter value determines the module contents

- Can generate
  - Modules
  - User defined primitives
  - Verilog gate primitives
  - Continuous assignments
  - initial and always blocks
Generate Loop

module bitwise_xor (output [N-1:0] out, input [N-1:0] i0, i1);
  parameter N = 32; // 32-bit bus by default
  genvar j; // This variable does not exist during simulation
  generate for (j=0; j<N; j=j+1) begin:
    xor_loop //Generate the bit-wise Xor with a single loop
      xor g1 (out[j], i0[j], i1[j]);
    end
  endgenerate //end of the generate block
endmodule

/* An alternate style using always blocks:
   reg [N-1:0] out;
   generate for (j=0; j<N; j=j+1) begin: bit
     always @(i0[j] or i1[j]) out[j] = i0[j] ^ i1[j];
   end
   endgenerate
endmodule */
Generate Conditional

module multiplier (output [product_width -1:0] product, input [a0_width-1:0] a0, input [a1_width-1:0] a1);

parameter a0_width = 8;
parameter a1_width = 8;

localparam product_width = a0_width + a1_width;

generate
  if (a0_width <8) || (a1_width < 8)
    cla_multiplier #(a0_width, a1_width) m0 (product, a0, a1);
  else
    tree_multiplier #(a0_width, a1_width) m0 (product, a0, a1);
endgenerate

endmodule
module adder (output co, output [N-1:0] sum, input [N-1:0] a0, a1, input ci);

parameter N = 4;

// Parameter N that can be redefined at instantiation time.
generate
  case (N)
    1: adder_1bit adder1(c0, sum, a0, a1, ci);
    2: adder_2bit adder2(c0, sum, a0, a1, ci);
    default: adder_cla #(N) adder3(c0, sum, a0, a1, ci);
  endcase
endgenerate

endmodule
Nesting

• Generate blocks can be nested
  • Nested loops cannot use the same `genvar` variable

```vhdl
include nwcell.v;

module nwgrid #(parameter N=8) (clk, reset, enable);
  input wire clk;
  input wire reset;
  input wire enable;
  genvar i;
  genvar j;
  for (i=0; i<N; i=i+1) begin : X
    for (j=0; j<N; j=j+1) begin : Y
      wire scout_v;
      wire [N-1:0] scout;
      wire [1:0] backpath;
      if(i==0 && j==0) begin
```
Logic Synthesis

- Verilog: two use-cases
  - Model for discrete-event simulation
  - Specification for a logic synthesis system

- Logic synthesis: convert subset of Verilog language $\rightarrow$ netlist

Two stages
1. Translate source to a netlist
   - Register inference
2. Optimize netlist for speed and area
   - Most critical part of the process
   - Awesome algorithms
What Can/Can’t Be Translated

• Structural definitions
  • Everything
• Behavioral blocks
  • When they have reasonable interpretation as combinational logic, edge, or level-sensitive latches
• User-defined primitives
  • Primitives defined with truth tables
  • Some sequential UDPs can’t be translated (not latches or flip-flops)
• Initial blocks
  • Used to set up initial state or describe finite testbench stimuli
  • Don’t have obvious hardware component
• Delays
  • May be in the Verilog source, but are simply ignored
• Other obscure language features
  • In general, things dependent on discrete-event simulation semantics
  • Certain “disable” statements
  • Pure events
Agenda

**FPGAs**
What’s so good about them? What’s so bad about them?

**Cascade**
How we make the good stuff better, and the bad stuff less awful

**Live Demo**
Writing a simple program in Cascade

**Time Permitting**
How Cascade works? Verilog minutiae?
What are FPGAs good for?

- FPGAs make sense when:
  - A workload is high-performance but also predictable
  - Application requirements change relatively frequently
But programming an FPGA is HARD!

- Verilog is complicated:
  - Mix of concurrent and sequential semantics
  - Awkward type system
  - Half-baked meta-programming
  - Synthesizable vs unsynthesizable code

- Domain-specific Languages
  - Chisel, Halide, etc...

<table>
<thead>
<tr>
<th>Widely Used</th>
<th>Niche</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaScript</td>
<td>Haskell</td>
</tr>
<tr>
<td>R</td>
<td>C/C++</td>
</tr>
<tr>
<td>Verilog</td>
<td>Java</td>
</tr>
</tbody>
</table>
And compilation takes FOREVER!

• **Software Compilers:**
  • $O(\text{seconds})$
  • Reason about programs locally
  • Pre-defined $O(n^k)$ rules

• **Software Development:**
  • Compile-test-debug cycle
  • Test and deploy in the same environment

• **Hardware Compilers:**
  • $O(\text{minutes to hours})$
  • Reason about programs globally
  • NP-hard constraint satisfaction

• **Hardware Development:**
  • Debug behavior in a simulator
  • Debug timing in hardware
  • Test and deploy in different environments
• CASCADE
• Makes programming hardware feel like programming software
Design Goals

- **Interactivity**
  Modify a running program, I/O side effects visible immediately

- **Expressiveness**
  Eliminate synthesizable vs non-synthesizable distinction

- **Portability**
  Write code once, run on many platforms with little modification

- **Performance**
  Don’t pay for features you don’t use
Interactivity

• Just-in-Time Compilation
  • Code runs immediately in a simulator
  • Compilation takes place in the background
  • Control switches when compilation is done
  • Code appears to run faster over time

• Why can we do this?
  • What’s the meaning of a Verilog program?
  • What’s the meaning of any program?
Interactivity

• Just-in-Time Compilation
  • Code runs immediately in a simulator
  • Compilation takes place in the background
  • Control switches when compilation is done
  • Code appears to run faster over time

• Why can we do this?
  • What’s the meaning of a Verilog program?

```
1: procedure EVAL(e)
2:     if e is an update then
3:         perform sequential update
4:     else
5:         evaluate combinational logic
6:     end if
7:     enqueue new events
8: end procedure

1: procedure REFERENCESCHEDULER
2:     while T do
3:         if △ activated events then
4:             EVAL(any activated event)
5:         else if △ update events then
6:             activate all update events
7:         else
8:             advance time t; schedule recurring events
9:         end if
10:     end while
11: end procedure
```
Expressiveness

• Unsynthesizable Verilog in hardware
  • Display statements
  • Finish statements
  • Longer term: support for the entire unsynthesizable language subset

• Why can we do this?
  • What is Cascade doing differently compared to a traditional compiler?

```verbatim
module Main(
  input wire clk,
  input wire [3:0] pad_val,
  output wire [7:0] led_val,
  output wire [7:0] r_x,
  input wire [7:0] r_y);

reg cnt [7:0] = 1;
assign r_x = cnt;
always @(posedge clk_val)
  if (pad_val == 0)
    cnt <= r_y;
  else
    $display(cnt);
$finish;
assign led_val = cnt;
endmodule
```
module Main(
  input wire  CLK,
  input wire  RW,
  input wire [31:0] ADDR,
  input wire [31:0] IN,
  output wire [31:0] OUT,
  output wire [31:0] WAIT
);

reg [31:0] _vars[3:0];
reg [31:0] _nvars[3:0];
reg _umask = 0, _numask = 0;
reg [1:0] _tmask = 0, _ntmask = 0;
reg [31:0] _oloop = 0, _itr = 0;
wire clk_val = _vars[0];
wire [3:0] pad_val = _vars[1];
wire [7:0] led_val;
wire [7:0] cnt = _vars[2];

always @(posedge clk_val)
  if (pad_val == 0)
    _nvars[2] <= pad_val << 1;
  _numask <= _umask;
else
  _nvars[3] <= cnt;
  _ntmask <= _tmask;
assign led_val = cnt;

wire _updates = _umask ^ _numask;
wire _latch = <LATCH> ||
  (_updates & _oloop);
wire _tasks = _tmask ^ _ntmask;
wire _clear = <CLEAR>;
wire _otick = _oloop & !_tasks;

always @(posedge CLK)
  _umask <= _latch ? _numask : _umask;
  _tmask <= _clear ? _ntmask : _tmask;
  _oloop <= <OLOOP> ? IN :
    _otick ? (_oloop-1) :
      _tasks ? 0 : _oloop;
  _itr <= <OLOOP> ? 0 :
    _otick ? (_itr+1) : _itr;
  _vars[0] <= _otick ? (_vars[0]+1) :
    <SET 0> ? IN : _vars[0];
  _vars[1] <= <SET 1> ? IN : _vars[1];
  _vars[2] <= <SET 2> ? IN :

assign WAIT = _o_loop;
always @(*)
case (ADDR)
  0: OUT = clk_val;
  // cases omitted ...
endmodule
Limitations and Future Work

- **Non-Monotonic language features**
  - Code deletion
  - Genvar statements

- **Timing-sensitive applications**
  - A giga-bit ethernet switch?
  - A peripheral which expects inputs on a perfectly periodic clock?

- **FPGA Virtualization:**
  - Share one FPGA between two instances of Cascade
  - Use Cascade to transparently run one very large program on two separate FPGAs

- **Speculative Optimization:**
  - Specialize the implementation of a program to the values that it sees at runtime
  - Generate smaller / faster code
Thank You

Questions on Piazza

Bug Reports on https://github.com/vmware/cascade
AmorphOS Motivation

Bigger, faster FPGAs deployed in the cloud
- Microsoft Catapult/Azure
- Amazon F1
- FPGAs: Reconfigurable Accelerators
  - ASIC Prototyping, Video & Image Proc., DNN, Blockchain
  - Potential solution to *accelerator provisioning challenge*

Our position: FPGAs *will* be shared
- Sharing requires protection
- Abstraction layers provide compatibility
- Beneficiary: provider → consolidation
FPGA Background

- Field Programmable Gate Array (FPGA)
  - Reconfigurable interconnect → custom data paths
  - FPGAs attached as coprocessors to a CPU

- FPGA Build Cycle
  - Synthesis: HDL → Netlist (~seconds)
  - Place and Route: Netlist → Bitstream (~min--hours)
  - Reconfiguration/Partial Reconfiguration (PR)

- Production systems: No multi-tenancy

- Emerging/Research Systems use fixed slots/PR
  - Fixed-sized slots → fragmentation (50% or more)
  - Elastic resource management needed
AmorphOS Goals

• Protected Sharing/Isolation
  • Mutually distrustful applications

• Compatibility / Portability
  • HDL written to AmorphOS interfaces
  • 14 benchmarks run unchanged on Microsoft Catapult and Amazon F1

• Elasticity
  • User logic scales with resource availability
  • Sharing density scales with availability
AmorphOS Abstractions

- **Zone**: Allocatable Unit of Fabric
  - 1 Global zone
  - N dynamically sized, sub-dividable PR zones

- **Hull**: OS/Protection Layer
  - Memory Protection, I/O Mediation
  - Interfaces form a compatibility layer

- **Morphlet**: Protection Domain
  - Extends Process abstraction
  - Encapsulate user logic on PR or global zone

- **Registry**: Bitstream Cache
  - Hides latency of place-and-route (PaR)

![Diagram of AmorphOS Abstractions]
Scheduling Morphlets

• Tradeoff
  • Fixed zones + PR → fast, fragmentation
  • Global zone + PaR → eliminates fragmentation, slow

• AmorphOS: best of both worlds
  • Low Latency Mode
    • Fixed zones + PR
    • Default Morphlet bitstream
  • High Throughput Mode
    • Combine multiple Morphlets
    • Co-schedule on a global zone
Scheduling Case Study

Low-Latency Mode

| Host | DRAM | App A | Morphlet A | FPGA Fabric | App B |

High-Throughput Mode

| Host | DRAM | App A | Morphlet A | Morphlet A' | App B | Morphlet B |

Low-Latency Mode

| Host | DRAM | App A | Morphlet A | Morphlet A' | App B | Morphlet B |

High-Throughput Mode

| Host | DRAM | App A | Morphlet A | Morphlet B | App C | Morphlet C | Morphlet D |

\[ T_0 \] \[ T_1 \] \[ T_2 \] \[ T_3 \]
AmorphOS Hull

- Hardens and extends vendor Shells
  - Microsoft Catapult
  - Amazon F1
- AmorphOS Interfaces
  - Control: \textit{CntrlReg}
  - Virtual Memory: \textit{AMI}
  - Bulk Data Transfer: Simple-\textit{PCIe}
AmorphOS Hull

- Hardens and extends vendor Shells
  - Microsoft Catapult → Higher Level
  - Amazon F1 → Lower Level
- AmorphOS Interfaces
  - Control: CntrlReg
  - Virtual Memory: AMI
  - Bulk Data Transfer: Simple-PCIe
- Multiplexing of interfaces
  - Isolation/data protection
  - Scalable, 32 accelerators
    - Tree of multiplexers
Implementation & Methodology

• Catapult Prototype
  • Altera Mt. Granite Stratix V GS 2x4GB DDR3, Windows Server
  • Segment-based protection, partial reconfiguration (PR)

• Amazon F1 Prototype
  • Xilinx UltraScale+ VU9P, 4x16GB GDDR4, CentOS 7.5
  • No PR, but much more fabric than Catapult

• Workloads
  • DNNWeaver – DNN inference
  • MemDrive – Memory Bandwidth
  • Bitcoin – blockchain hashing
  • CHStone – 11 accelerators (e.g. AES, jpeg, etc)
Scalability

- **DNNWeaver:**
  - 32X density
  - 23X throughput

- **MemDrive:**
  - BW contention

- **Bitcoin:**
  - Compute-bound

**Takeaway:** Massive throughput/density improvement possible, awareness of contended resources necessary

- **F1: Xilinx UltraScale+ VU9P, 4x16GB GDDR4, CentOS 7.5**
- **Higher is better, Homogenous Morphlets**
Throughput

- **8 Bitcoin Morphlets**
- **Catapult Altera Stratix V GS 2x4GB DDR3, Windows**
- **Registry pre-populated: ctxt sw. 200ms**
- **Log Scale, Lower is better**

<table>
<thead>
<tr>
<th>Run Time (s)</th>
<th>No Sharing</th>
<th>Two PR Zones</th>
<th>AmorphOS (HT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**End-to-End Bitcoin Run Time**

- **Fixed Zones:** worse than no sharing due to down-scaling!
- **No-Sharing:** serialized

**Takeaway:** Co-scheduling on a global zone can perform better than fixed-sized slots and PR
Partitioning Policies

Global Scope
- Everything runs serially
- Single context

Global Zone
- Multiple Morphlets
- No fixed size zones
- Single-level zone scheme
- Two PR zones
- One Morphlet each

Subdivide
- Divide top-level PR zone into smaller PR zones
Partitioning Policies

- **Bitcoin Morphlets**
- **Catapult**: Altera Mt. Granite Stratix V GS 2x4GB DDR3, Windows
- **Registry pre-populated**: ctxt sw. 200ms
- **Higher is better**

**Takeaway:**
- Hierarchical PR on limited HW not worth it
- See paper for projections on F1
Related Work

• **Access to OS-managed resources**
  - Born: So [TECS ’08, Thesis ‘07]
  - Leap: Adler [FPGA ‘11]
  - CoRAM: Chung [FPGA ‘11]

• **First-class OS support**
  - HThreads: Peck [FPL’06], ReconOS: Lübbers [TECS ‘09] -- extend threading to FPGA SoCs
  - **MURAC: Hamilton [FCCM ‘14] – extend process abstraction to FPGAs**

• **Single-application Frameworks**
  - Catapult: Putnam [ISCA ‘14] / Amazon F1

• **Fixed-slot + PR**
  - OpenStack support: Chen [CF ‘14], Byma [FCCM ’14]; Fahmy [CLOUDCOM ‘15];
  - Disaggregated FPGAs: Weerasinghe [UIC-ATC-ScalCom ‘15]

• **Overlays**
  - Zuma: Brant [FCCM ‘12],
  - Hoplite: Kapre [FPL ‘15],
  - ReconOS+Zuma: [ReConfig ’14]
Conclusions & Future Work

• Compatibility Improved
  • without restricting programming model
  • Comprehensive set of stable interfaces
  • Port AmorphOS *per platform not each accelerator per platform*

• Scalability achieved *within and across accelerators*
  • AmorphOS transparently scales morphlets up/down
  • Powerful combination of slots/Partial Reconfiguration and full FPGA bitstreams

• Future work
  • Transparently scale across multiple FPGAs
  • Scale across more than just FPGAs
  • Open source AmorphOS/port to more platforms
Example alignment view

Reference genome

Aligned reads
Sequence alignment: Scoring

- Scoring matrices are used to assign scores to each comparison of a pair of characters.
- Identities and substitutions by similar amino acids are assigned positive scores.
- Mismatches, or matches that are unlikely to have been a result of evolution, are given negative scores.

<table>
<thead>
<tr>
<th>Option 1</th>
<th>Option 2</th>
<th>Option 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>- A C - G G C - G</td>
<td>- A C G G - C - G</td>
<td>- A C G - G C - G</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>K</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>C</td>
<td>Y</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>R</td>
<td>I</td>
<td>K</td>
</tr>
</tbody>
</table>

A: +5  C: +5  D: -5  E: +5  F: +5  G: +5  H: -5  I: +5  K: +5
Pairwise alignment: the problem

The number of possible pairwise alignments increases explosively with the length of the sequences:
Two protein sequences of length 100 amino acids can be aligned in approximately $10^{60}$ different ways.

Time needed to test all possibilities is same order of magnitude as the entire lifetime of the universe.
Pairwise alignment: the canonical solution

Dynamic programming
(the Needleman-Wunsch algorithm)
Alignment depicted as path in matrix

T C G C A

T C G C A

TCGCA
TC-CA

TCGCA
T-CCA
Dynamic programming: computing scores

Any given point in matrix can only be reached from three possible positions (you cannot “align backwards”).

=> Best scoring alignment ending in any given point in the matrix can be found by choosing the highest scoring of the three possibilities.
Any given point in matrix can only be reached from three possible positions (you cannot “align backwards”).

=> Best scoring alignment ending in any given point in the matrix can be found by choosing the highest scoring of the three possibilities.

<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>C</th>
<th>G</th>
<th>C</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
score(x,y) = \max \left\{ \begin{array}{l}
    score(x,y - 1) - \text{gap penalty}
  \end{array} \right\}
\]
Any given point in matrix can only be reached from three possible positions (you cannot “align backwards”).

=> Best scoring alignment ending in any given point in the matrix can be found by choosing the highest scoring of the three possibilities.

\[
\text{score}(x, y) = \max \left\{ \begin{array}{l}
\text{score}(x, y-1) - \text{gap-penalty} \\
\text{score}(x-1, y-1) + \text{substitution-score}(x, y)
\end{array} \right.
\]
Dynamic programming

Any given point in matrix can only be reached from three possible positions (you cannot “align backwards”).

=> Best scoring alignment ending in any given point in the matrix can be found by choosing the highest scoring of the three possibilities.

score(x,y) = max \begin{align*}
& \text{score(x,y-1) - gap-penalty} \\
& \text{score(x-1,y-1) + substitution-score(x,y)} \\
& \text{score(x-1,y) - gap-penalty}
\end{align*}
Dynamic programming: example

\[
a[i,j] = \max \begin{cases} a[i-1,j-1] -2 \\ a[i-1,j] + p(i,j) \\ a[i,j] -2 \end{cases}
\]

Gaps: -2
Dynamic programming: example

\[
a[i,j] = \max\begin{cases} a[i-1,j-1] - 2 \\ a[i-1,j-1] + p(i,j) \\ a[i-1,j] - 2 \end{cases}
\]

- **T**: T
- **C**: C
- **G**: G
- **A**: A

- **0**: 0
- **-2**: -2
- **-4**: -4
- **-6**: -6
- **-8**: -8
- **-10**: -10

\[\begin{array}{c}
0 \\
\uparrow \\
-2 \\
\uparrow \\
-4 \\
\uparrow \\
-6 \\
\uparrow \\
-8 \\
\uparrow \\
-10 \\
\end{array}\]
Dynamic programming: example

\[
a[i,j] = \max \begin{cases} 
    a[i, j-1] - 2 \\
    a[i-1, j-1] + p(i, j) \\
    a[i-1, j] - 2 
\end{cases}
\]
Dynamic programming: example

\[
a[i,j] = \max \begin{cases} 
  a[i,j-1] - 2 \\ 
  a[i-1,j] - p(i,j) \\ 
  a[i-1,j-1] + p(i,j) \\ 
  a[i,j-1] - 2 
\end{cases}
\]
Dynamic programming: example

\[
\begin{array}{cccccc}
0 & 0 & -2 & -4 & -6 & -8 & -10 \\
1 & 1 & -1 & -3 & -5 & -7 \\
2 & -4 & -1 & 2 & 0 & -2 & -4 \\
3 & -6 & -3 & 0 & 1 & 1 & -1 \\
4 & -8 & -5 & -2 & -1 & 0 & 2 \\
\end{array}
\]
Dynamic programming: example

```
Big MONGO HINT: What if each box is a parallel process?
```
References:

• Evita_verilog Tutorial, [www.aldec.com](http://www.aldec.com)

Review: Module definition

- Interface: port and parameter declaration
- Body: Internal part of module
- Add-ons (optional)
Delays on Primitive Instances

• Instances of primitives may include delays

buf b1(a, b); // Zero delay
buf #3 b2(c, d); // Delay of 3
buf #(4,5) b3(e, f); // Rise=4, fall=5
buf #(3:4:5) b4(g, h); // Min-typ-max
Register Inference

- The main trick

- reg does not always equal latch

- Rule: Combinational if outputs always depend exclusively on sensitivity list
- Sequential if outputs may also depend on previous values
Register Inference

• Combinational:

  reg y;
  always @(a or b or sel)
    if (sel) y = a;
    else y = b;

• Sequential:

  reg q;
  always @(d or clk)
    if (clk) q = d;
Register Inference

• A common mistake is not completely specifying a case statement
• This implies a latch:

always @(a or b)
case ({a, b})
  2’b00 : f = 0;
  2’b01 : f = 1;
  2’b10 : f = 1;
endcase

f is not assigned when {a,b} = 2b’11
Register Inference

• The solution is to always have a default case

always @(a or b)
case ({a, b})
  2’b00: f = 0;
  2’b01: f = 1;
  2’b10: f = 1;
  default: f = 0;
endcase

f is always assigned
Inferring Latches with Reset

• Latches and Flip-flops often have reset inputs
• Can be synchronous or asynchronous

• Asynchronous positive reset:

always @(posedge clk or posedge reset)
if (reset)
  q <= 0;
else q <= d;
Simulation-synthesis Mismatches

• Many possible sources of conflict

• Synthesis ignores delays (e.g., #10), but simulation behavior can be affected by them

• Simulator models X explicitly, synthesis doesn’t

• Behaviors resulting from shared-variable-like behavior of regs is not synthesized
  • always @(posedge clk) a = 1;
  • New value of a may be seen by other @(posedge clk) statements in simulation, never in synthesis
Compared to VHDL

• Verilog and VHDL are comparable languages
• VHDL has a slightly wider scope
  • System-level modeling
  • Exposes even more discrete-event machinery
• VHDL is better-behaved
  • Fewer sources of nondeterminism (e.g., no shared variables)
• VHDL is harder to simulate quickly
• VHDL has fewer built-in facilities for hardware modeling
• VHDL is a much more verbose language
  • Most examples don’t fit on slides