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

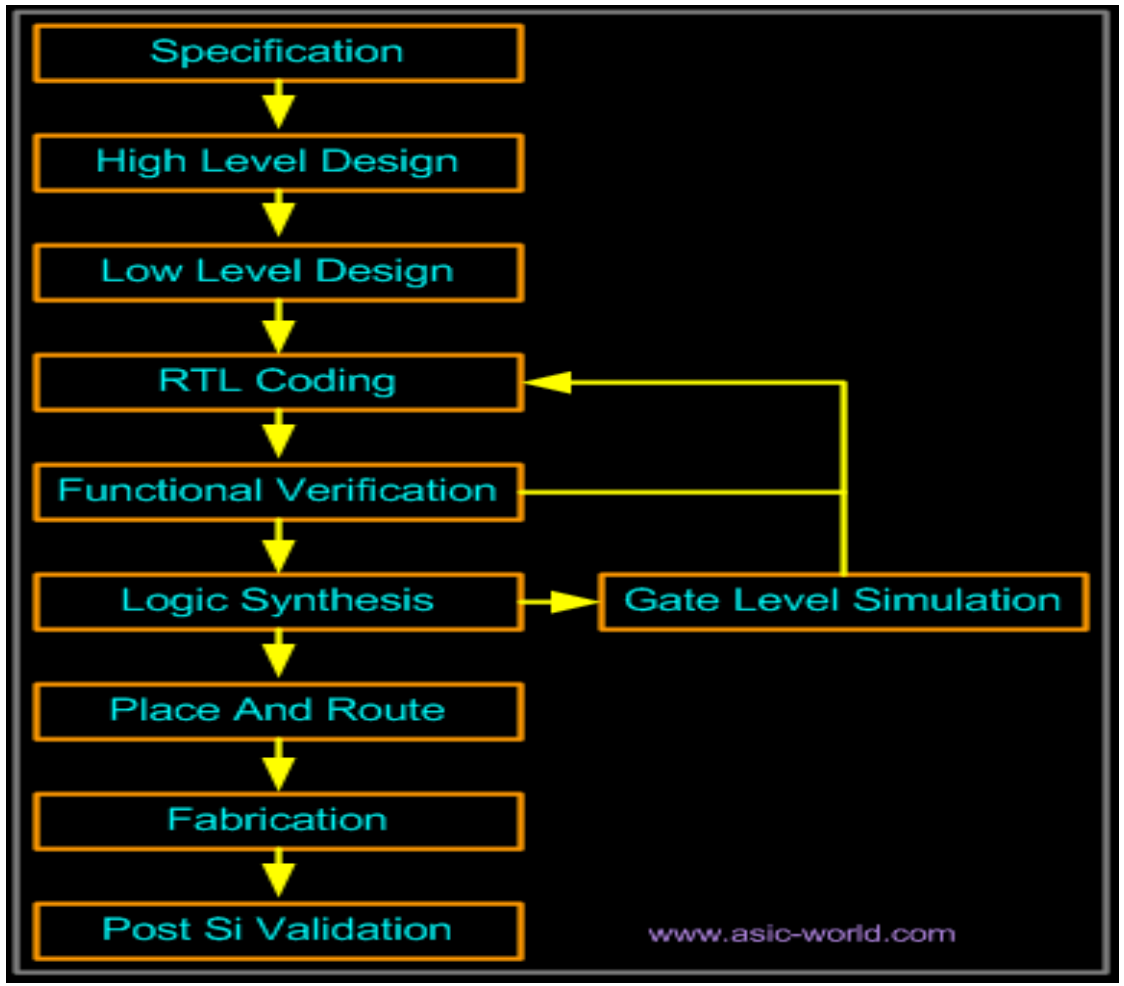
Acknowledgements/References:

- https://s3-us-west-2.amazonaws.com/cse291personalgenomics/Lectures2017/Lecture12_AlignmentVariantCalling.pptx
- https://web.stanford.edu/~jurafsky/slp3/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/~billlin/classes/ECE111/lectures/Lecture1.pptx
- http://www.cs.unc.edu/~montek/teaching/Comp541-Fall16/VerilogPrimer.pptx
- Evita_verilog Tutorial, www.aldec.com
- http://www.asic-world.com/verilog/

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 → 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 → 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

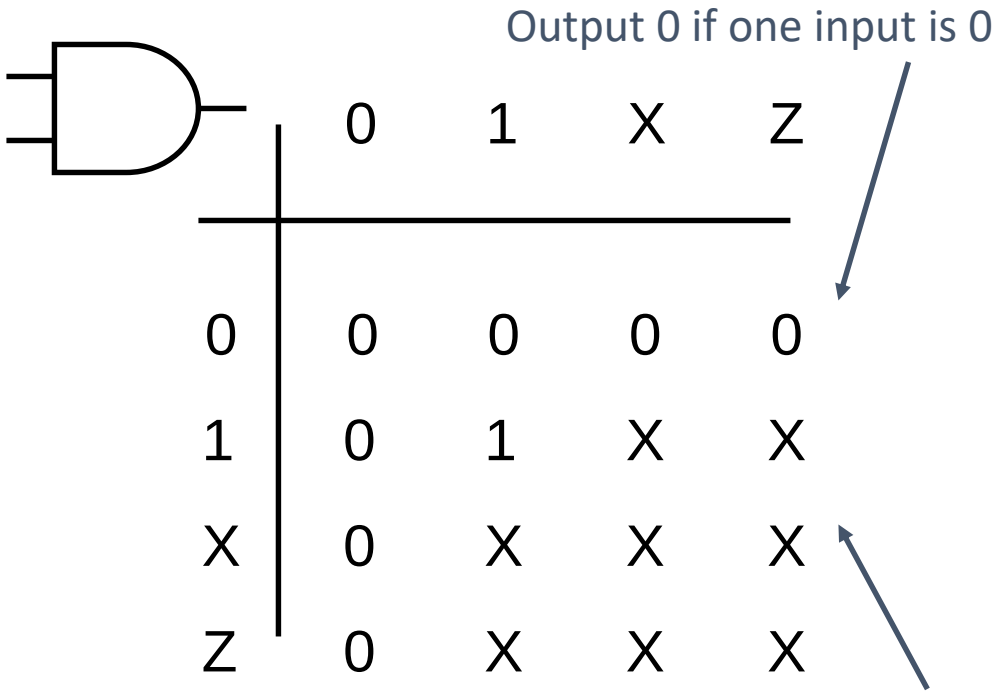
- Regs represent data storage
  - Behave exactly like memory in a computer
  - Hold their value until explicitly assigned in an *initial* or *always* block
  - Model latches, flip-flops, etc., but do not correspond exactly
  - *Shared variables*
    - Similar known shared state issues

Four-valued Data and Logic

Nets and regs hold *four-valued* data

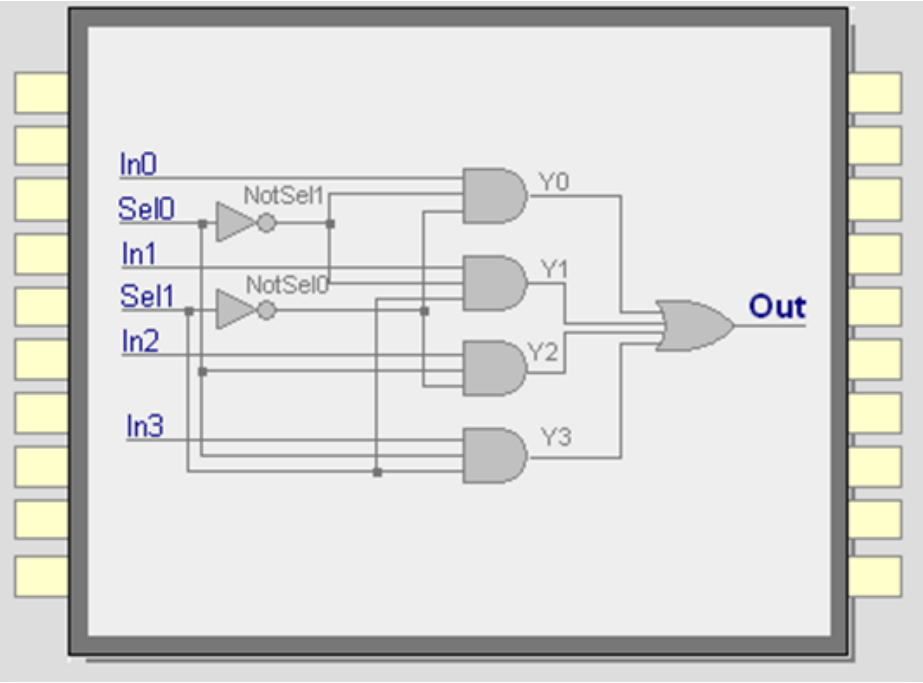
- 0, 1 → Umm…
- Z
  - Output for undriven tri-state (hi-Z)
  - Nothing is setting a wire's value
- X
  - Simulator can't decide the value
  - Initial state of registers
  - Wire driven to 0 and 1 simultaneously
  - Output of gate with Z inputs
- Data representation
  - Binary → 6'b100101
  - Hex → 6'h25

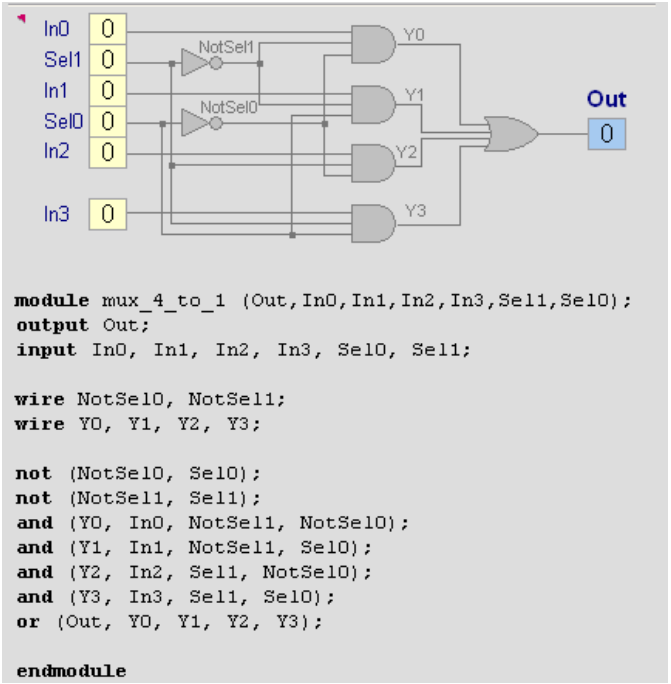
- Logical operators work on three-valued logic



Structural Modeling

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

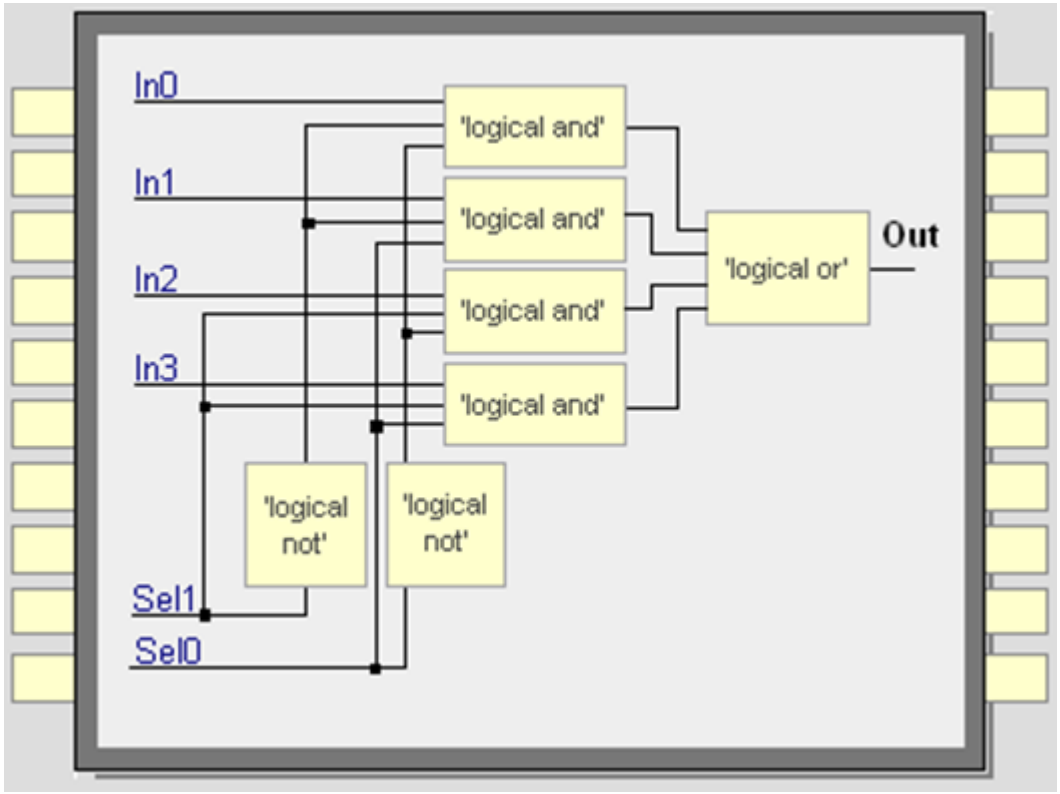


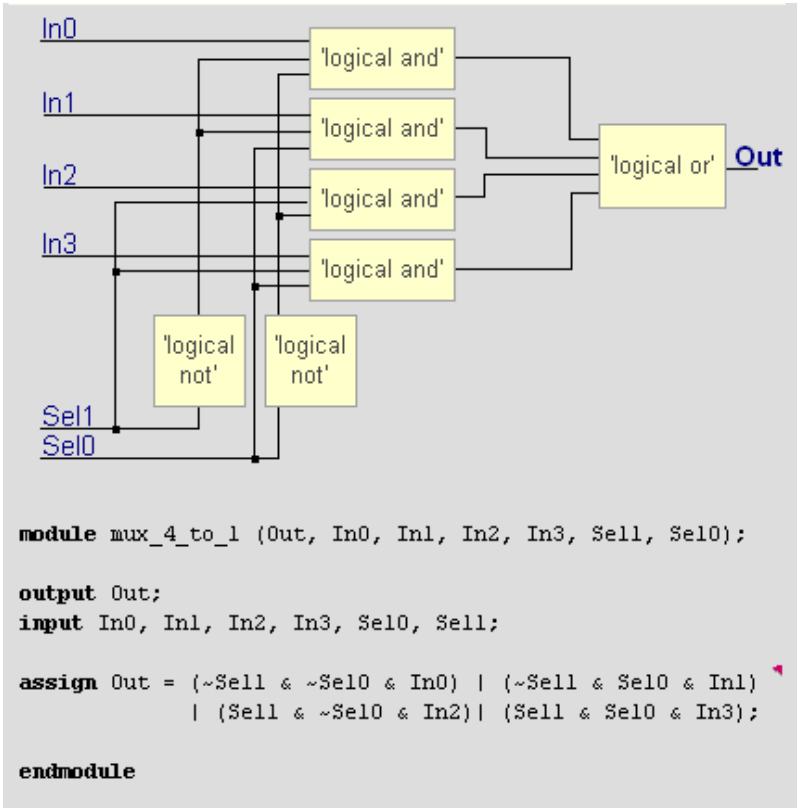



```
module mux_4_to_1 (Out,In0,In1,In2,In3,Sel1,Sel0);
output Out;
input In0, In1, In2, In3, Sel0, Sel1;

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

not (NotSel0, Sel0);
not (NotSel1, Sel1);
and (Y0, In0, NotSel1, NotSel0);
and (Y1, In1, NotSel1, Sel0);
and (Y2, In2, Sel1, NotSel0);
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

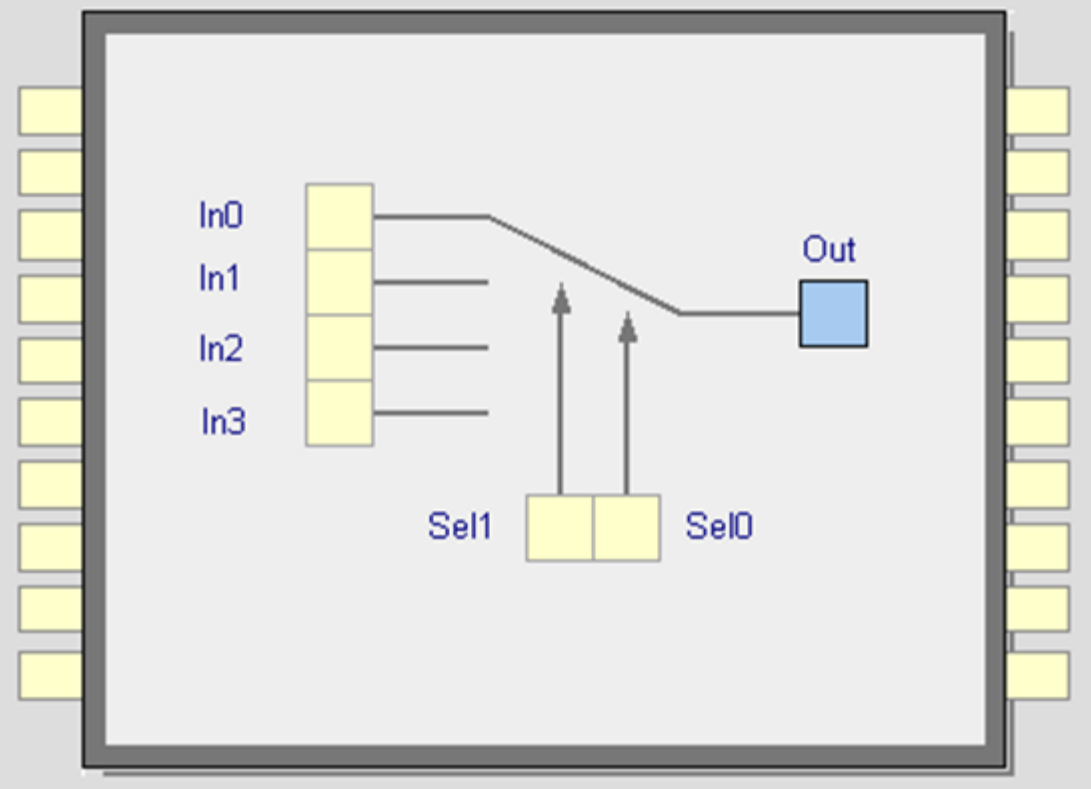




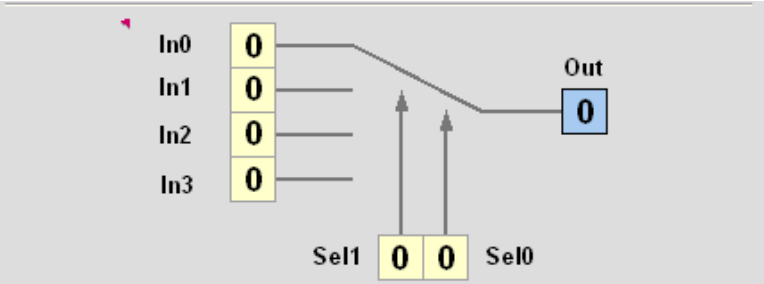

```
module mux_4_to_1 (Out, In0, In1, In2, In3, Sel1, Sel0);

output Out;
input In0, In1, In2, In3, Sel0, Sel1;

assign Out = (~Sel1 & ~Sel0 & In0) | (~Sel1 & Sel0 & In1)
             | (Sel1 & ~Sel0 & In2)| (Sel1 & Sel0 & In3);

endmodule
```

Behavioral Modeling

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



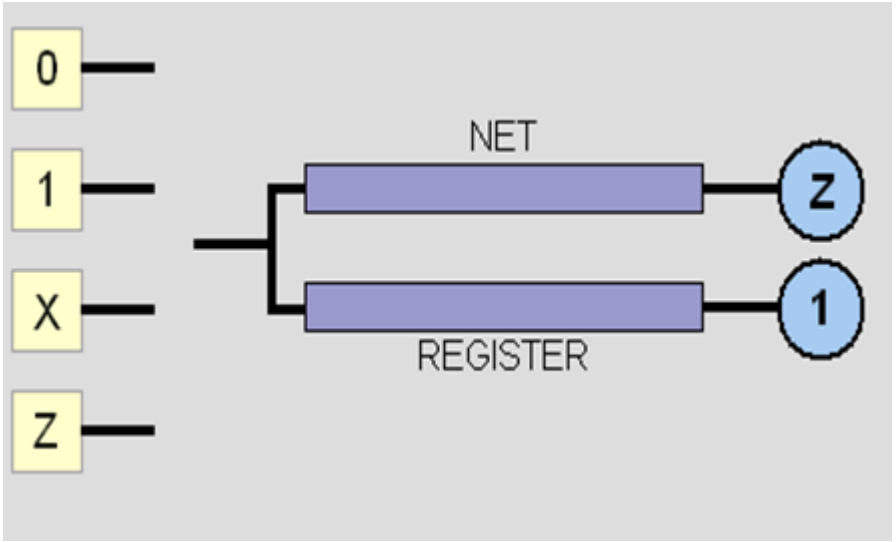
- Easier for testbenches
- Easier for abstract models of circuits
  - Simulates faster
- Provides sequencing




```
module mux_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'b00 : Out = In0;
    2'b01 : Out = In1;
    2'b10 : Out = In2;
    2'b11 : Out = In3;
    default : Out = 1'bx;
    endcase
  end

endmodule
```

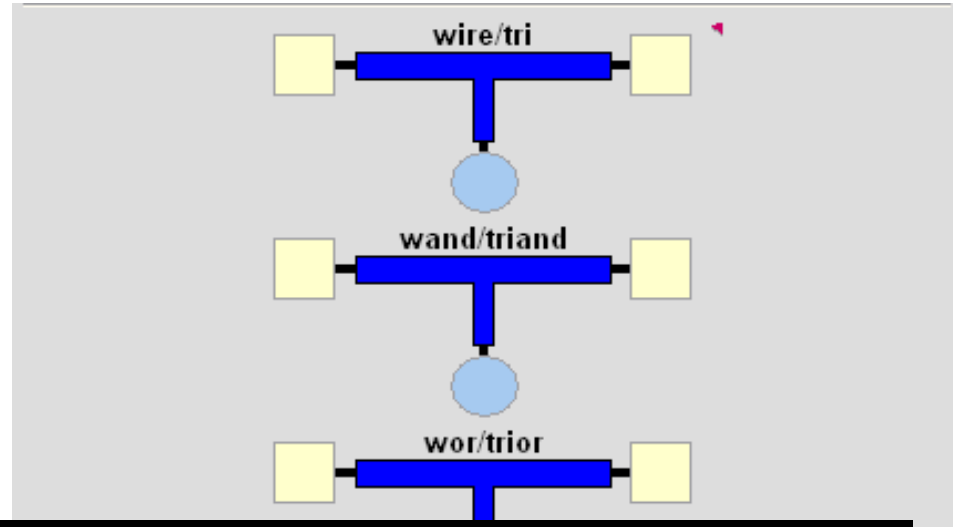
Signals

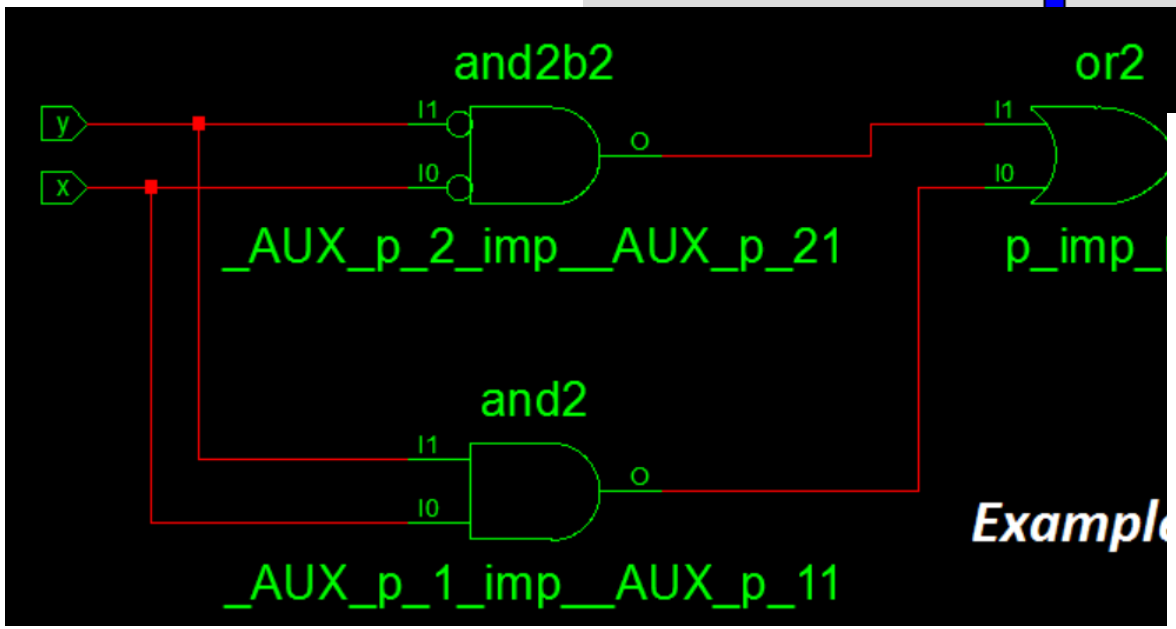
- Nets
  - Physical connection between hardware elements
- Registers
  - Store value even if disconnected



Nets

- wire/tri
- wand/triand
- wor/trior
- Force synthesis to insert gates
  - (e.g. AND, OR)



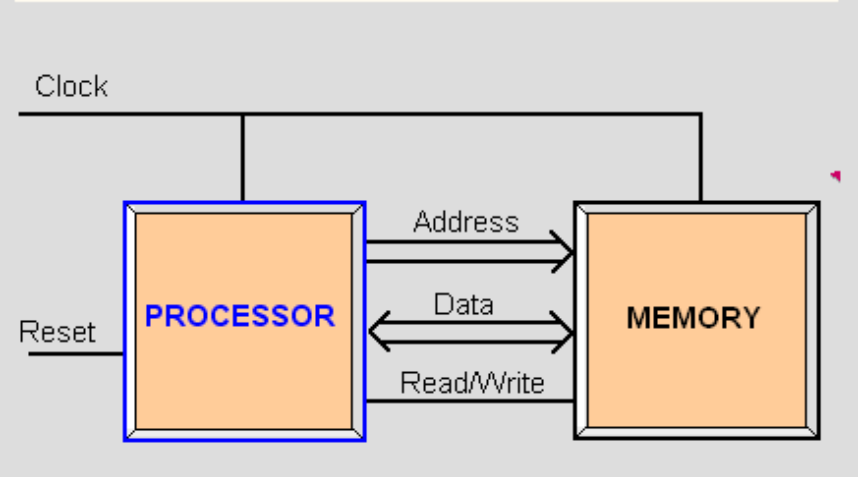



```
module comparatorwithwor(
    input x,
    input y,
    output z
    );
wor p ;

assign p = x&y;
assign p = ~x & ~y ;

assign z = p ;

endmodule
```

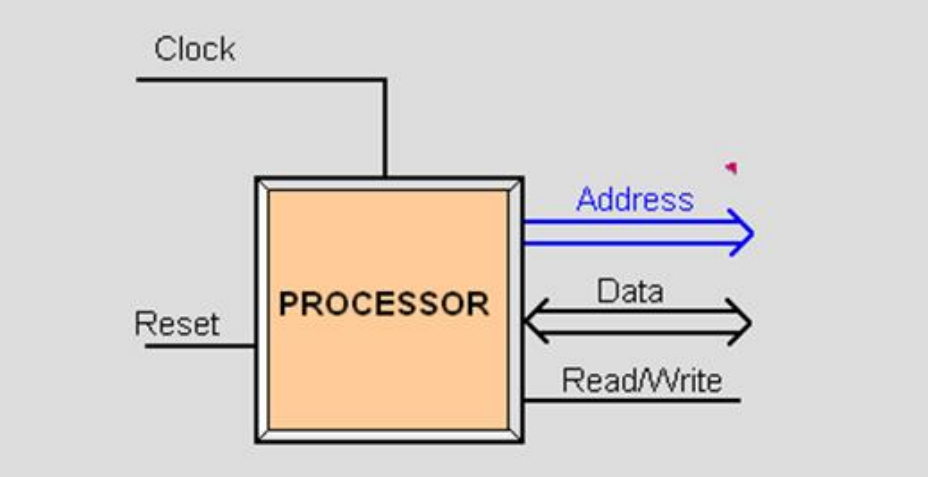
Using just wire

```
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




```
module Processor (Clock, Reset, Read_Write, Data, Address);
input Clock;
input Reset;
output Read_Write;
inout [15:0] Data;
output [19:0] Address;
. . .
endmodule
```




```
module Processor (Clock, Reset, Read_Write, Data, Address);
input Clock;
input Reset;
output Read_Write;
inout [15:0] Data;
output [19:0] Address;
. . .
reg [19:0] Address;
endmodule
```

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

Define bus widths

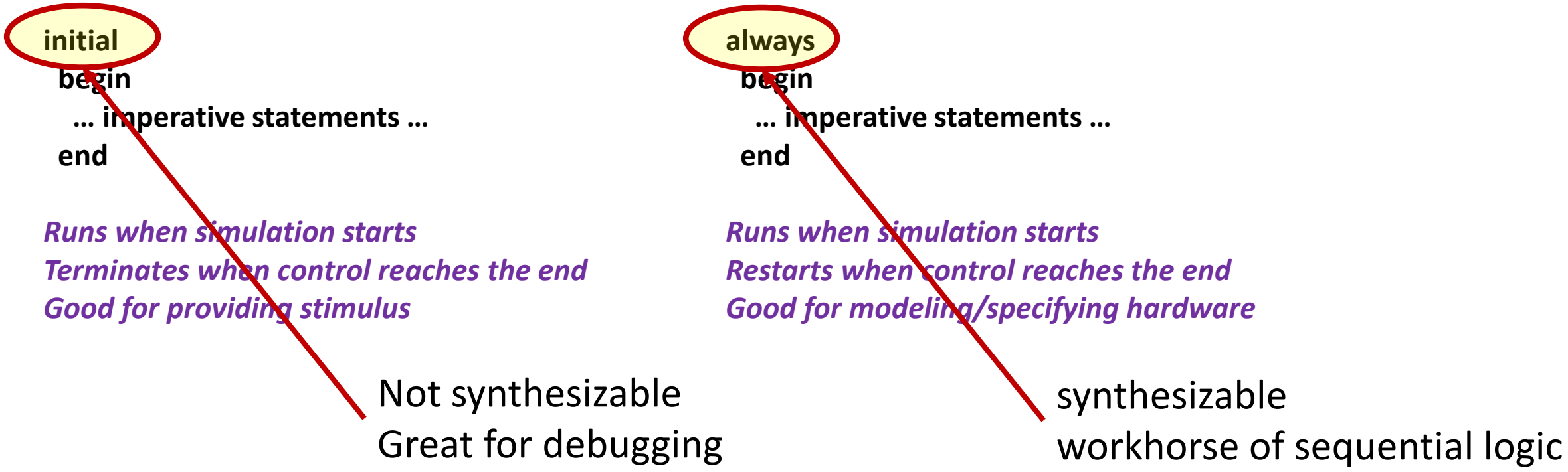
```
wire [8:0] sum;
wire [7:0] a, b;
wire carryin;

assign sum = a + b + carryin;
```

- 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 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

```
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

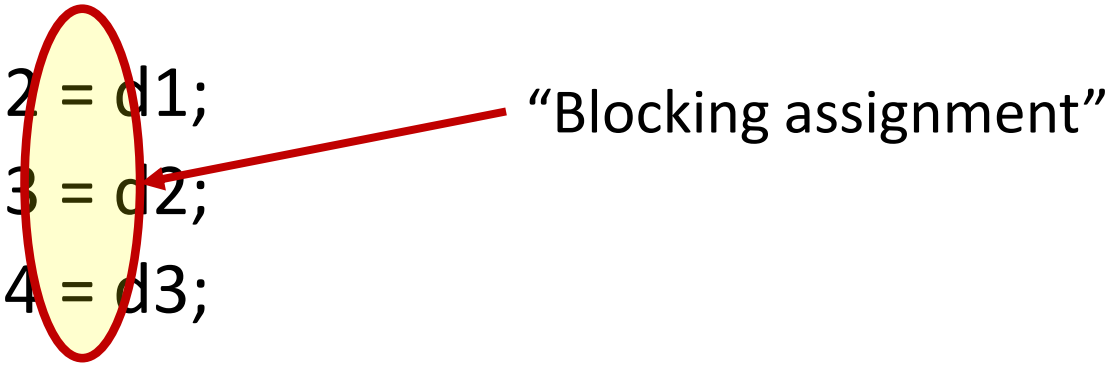
- Verilog has two types of procedural assignment

- 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

```
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?

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

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

Should ***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

FSM with Combinational Logic

```
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;
```

FSM with a Single Always Block

```
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

```
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

/* 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  */
```

Can do this with code but requires different numbers of xor modules depending on N

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
```

Generate Case

```
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

```
 8 //
 9 // Change history: 8/23/18 - Initial revision
10 //
11 ////////////////////////////////////////////////////////////////////////////////
12
13 include nwcell.v;
14
15 module nwgrid #(parameter N=8) (clk, reset, enable);
16
17    input wire clk;
18    input wire reset;
19    input wire enable;
20
21    genvar     i;
22    genvar     j;
23    for (i=0; i<N; i=i+1) begin : X
24       for (j=0; j<N; j=j+1) begin : Y
25
26      wire scout_v;
27      wire [N-1:0] scout;
28      wire [1:0]   backpath;
29
30      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 → 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

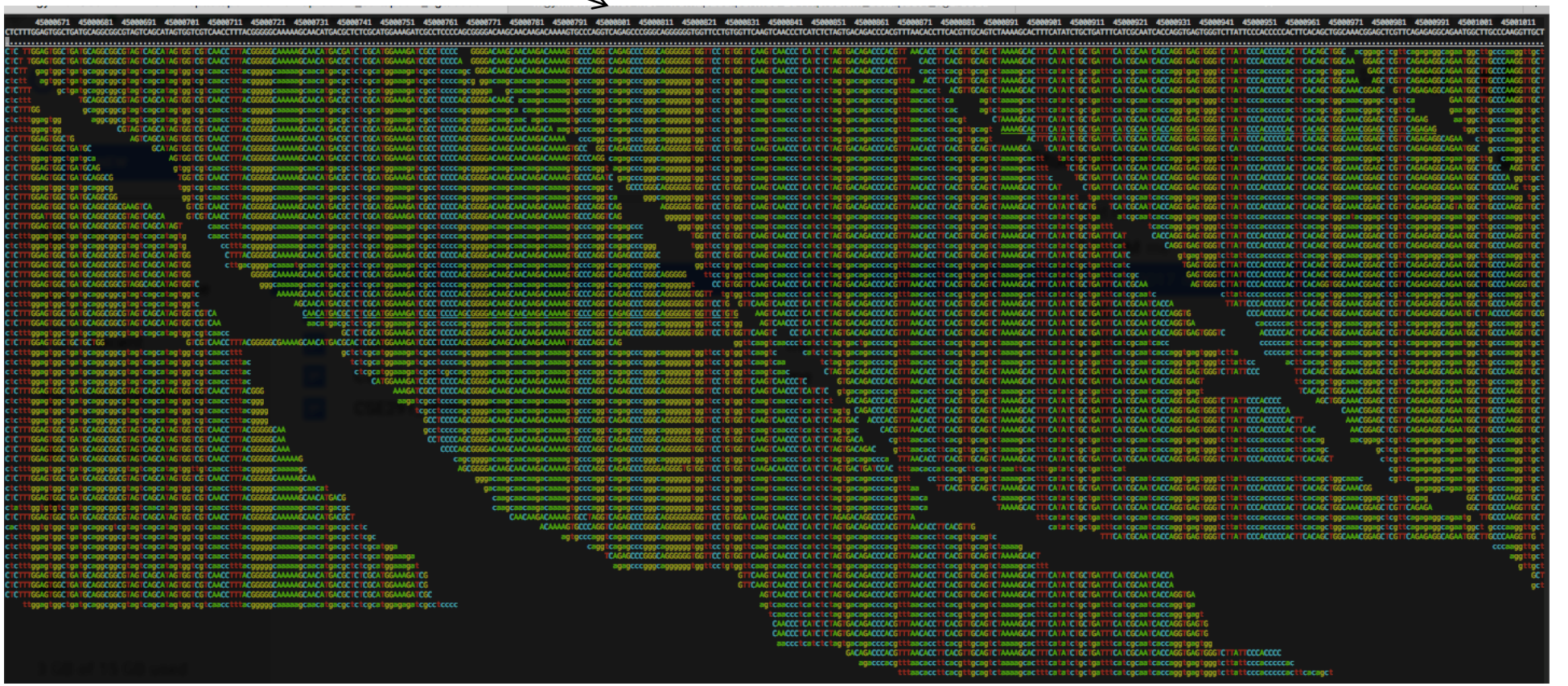
What Can/Can't Be Translated

- 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

Example alignment view

Reference genome



Aligned reads

Sequence alignment: Scoring

T A C G G G C A G	T A C G G G C A G	T A C G G G C A G
- A C - G G C - G	- A C G G - C - G	- A C G - G C - G
Option 1	Option 2	Option 3

- 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

A	C	D	E	F	G	H	I	K
A	C	Y	E	F	G	R	I	K
+5	+5	-5	+5	+5	+5	-5	+5	+5

Pairwise alignment: the problem

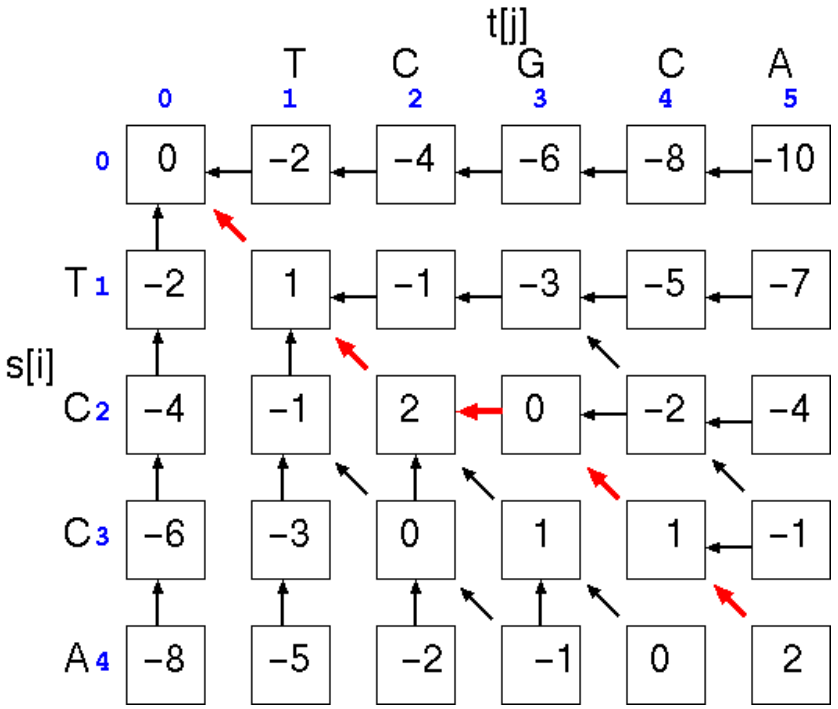
The number of possible pairwise alignments increases explosively with the length of the sequences:
Two protein sequences of length 100 amino acids can be aligned in approximately 10^{60} different ways

Time needed to test all

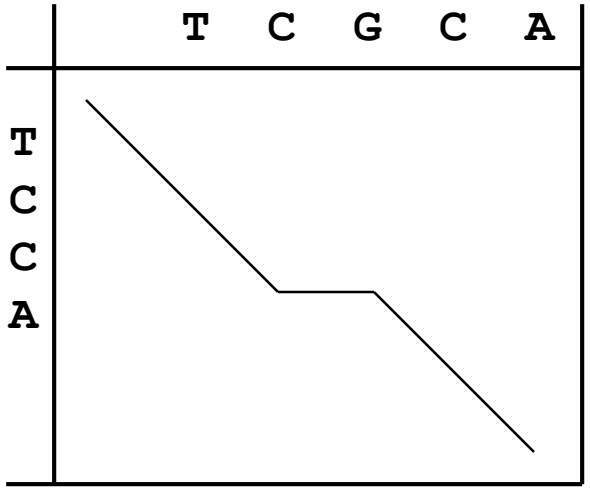
itude as the entire lifetime of the universe

Pairwise alignment: the canonical solution

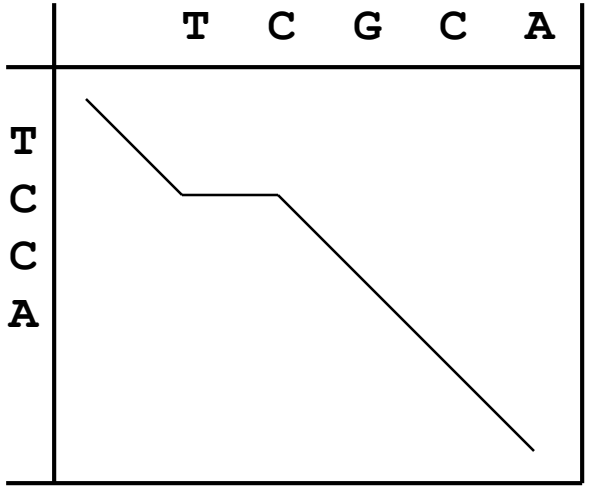
Dynamic programming
(the Needleman-Wunsch algorithm)



Alignment depicted as path in matrix

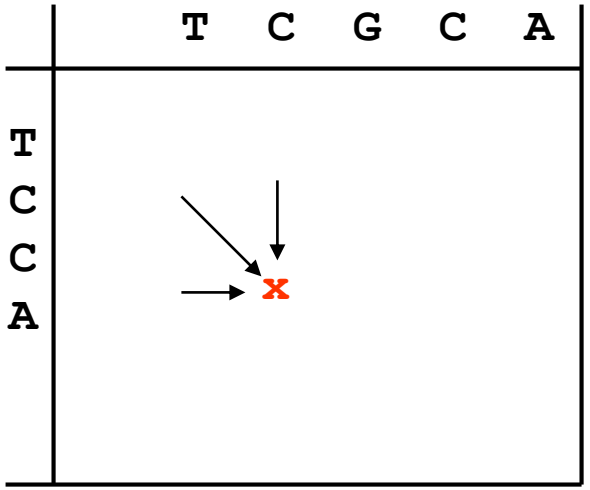


TCGCA
TC-CA



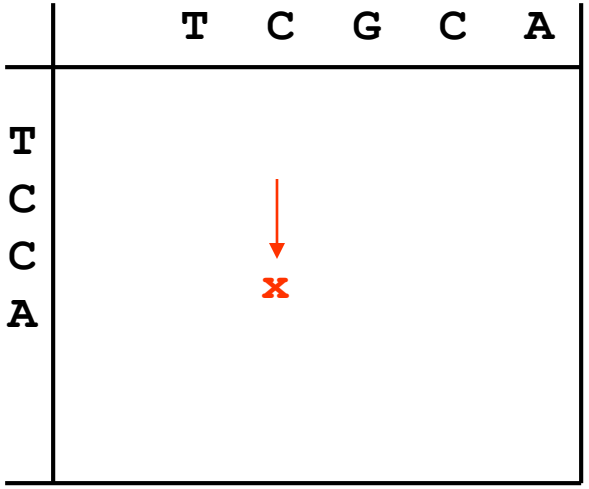
TCGCA
T-CCA

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.

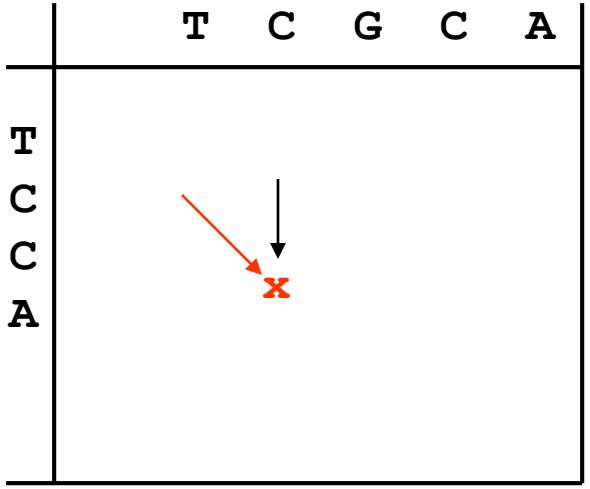
Dynamic programming



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 { score(x,y-1) - gap-penalty

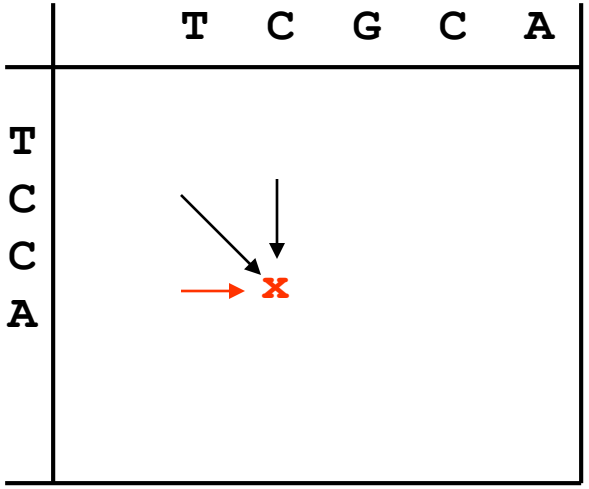
Dynamic programming



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.

$$\text{score}(x,y) = \max \begin{cases} \text{score}(x,y-1) - \text{gap-penalty} \\ \text{score}(x-1,y-1) + \text{substitution-score}(x,y) \end{cases}$$

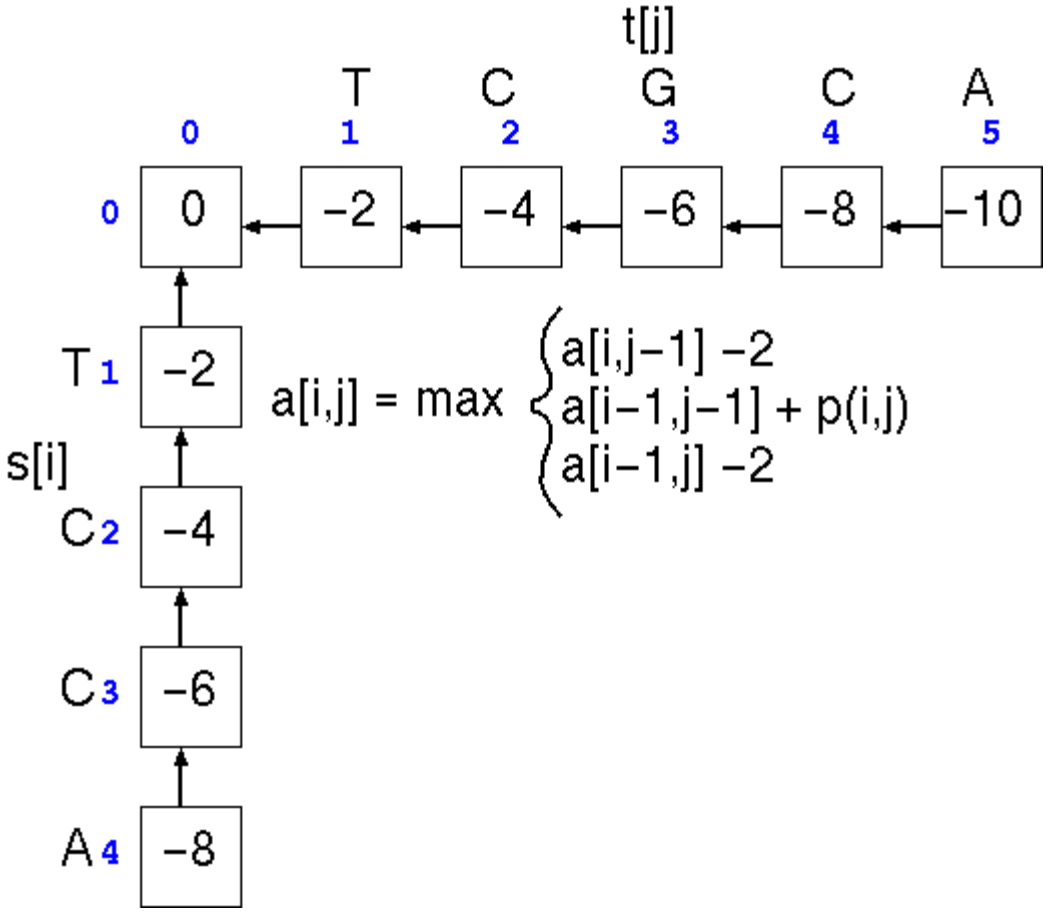
Dynamic programming



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.

$$score(x,y) = \max \begin{cases} score(x,y-1) - gap\text{-}penalty \\ score(x-1,y-1) + substitution\text{-}score(x,y) \\ score(x-1,y) - gap\text{-}penalty \end{cases}$$

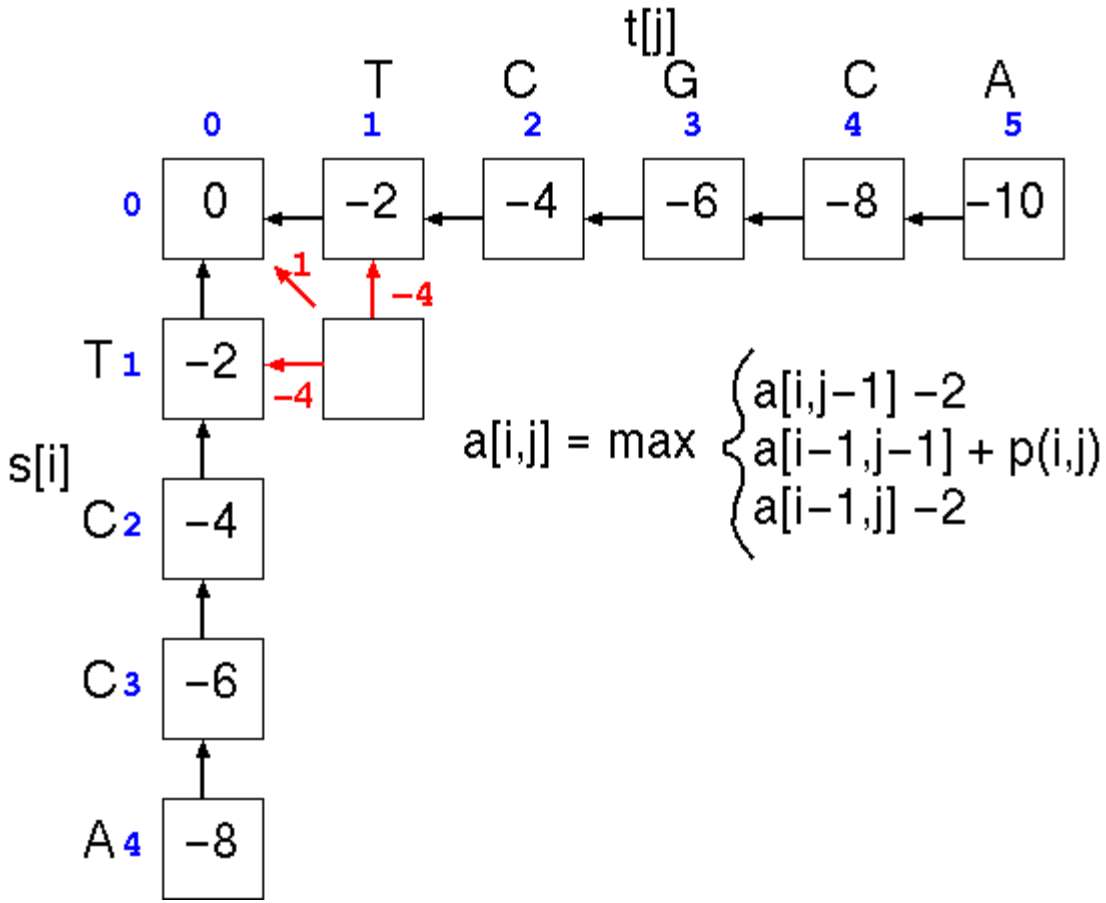
Dynamic programming: example



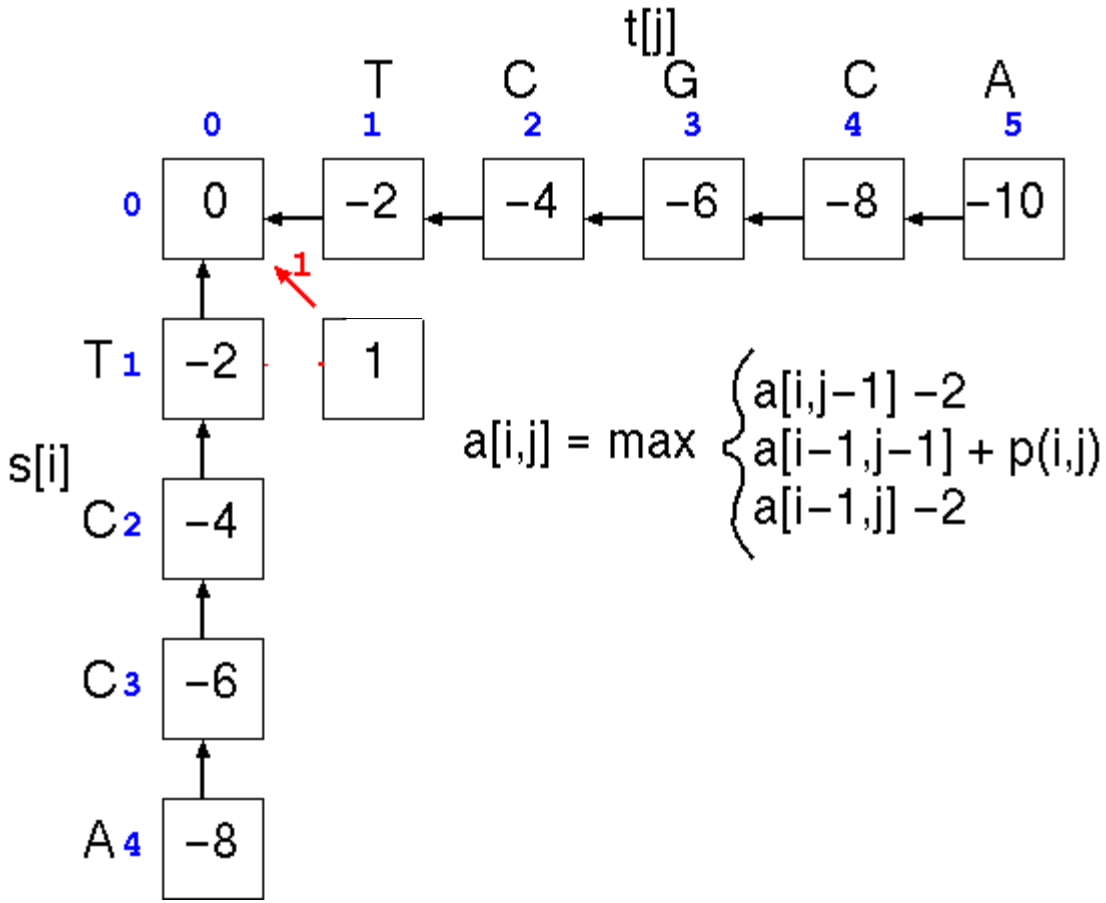
	A	C	G	T
A	1	-1	-1	-1
C	-1	1	-1	-1
G	-1	-1	1	-1
T	-1	-1	-1	1

Gaps: -2

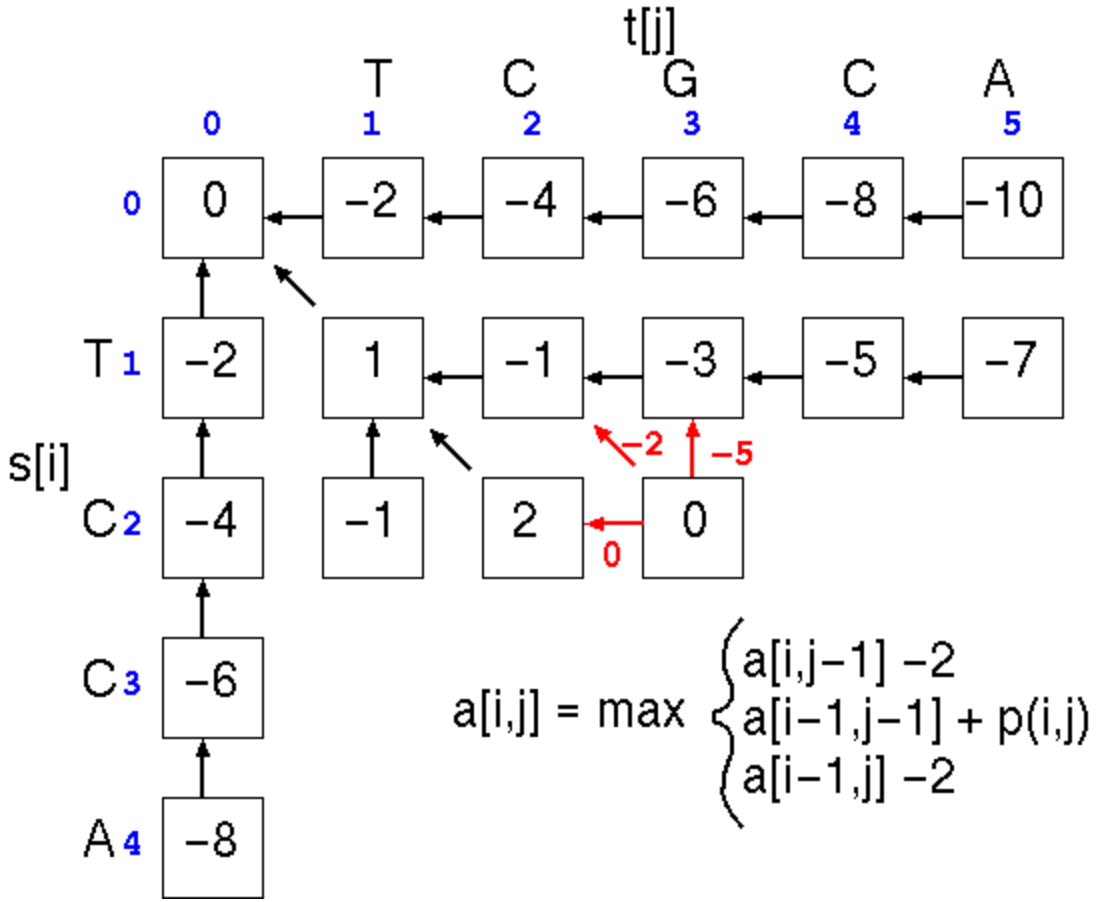
Dynamic programming: example



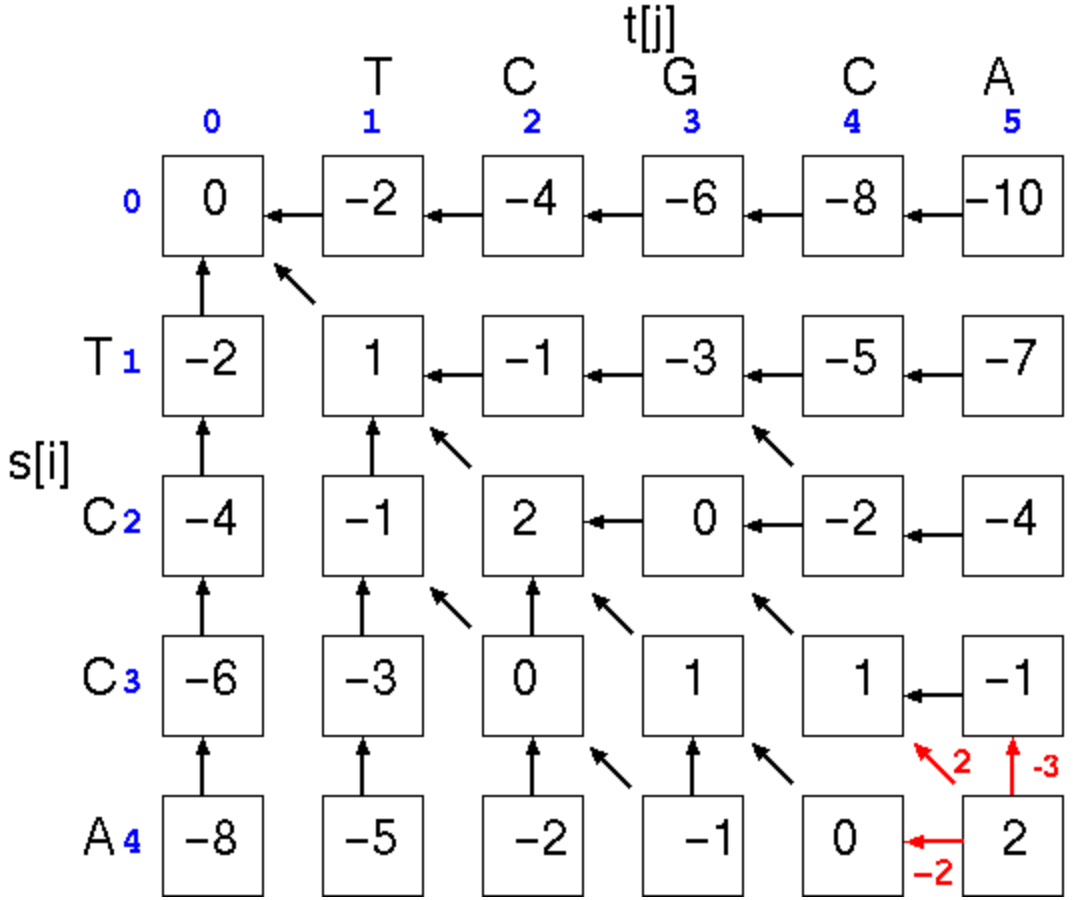
Dynamic programming: example



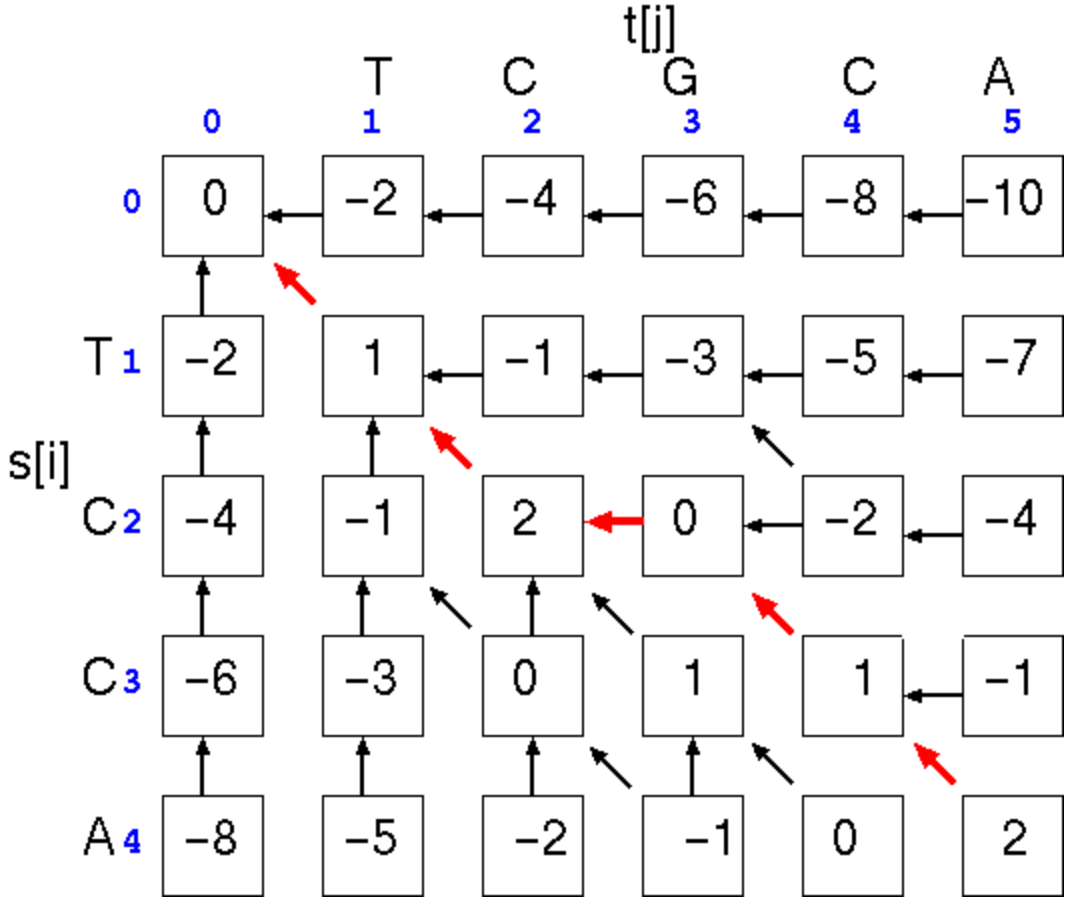
Dynamic programming: example



Dynamic programming: example



Dynamic programming: example




```
T C G C A
: :   : :
T C - C A
1+1-2+1+1 = 2
```

BIG MONGO HINT:
What if each box is a parallel process?

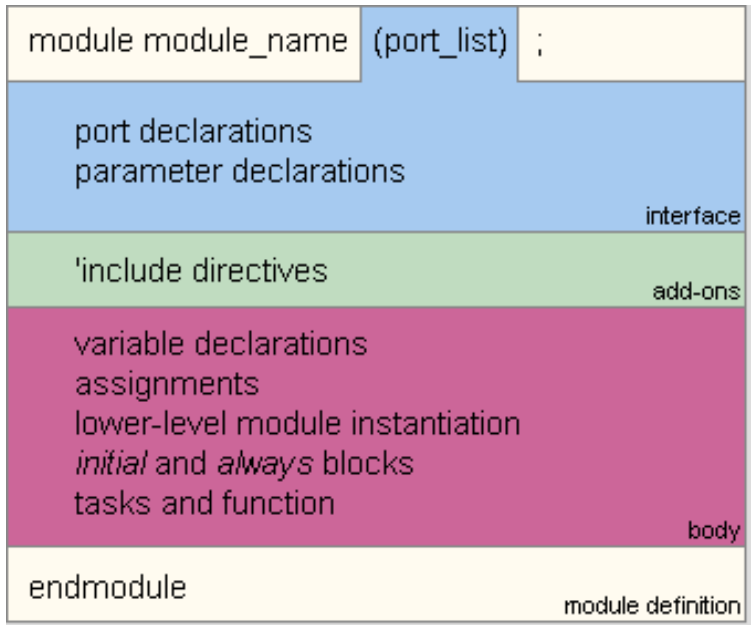
References:

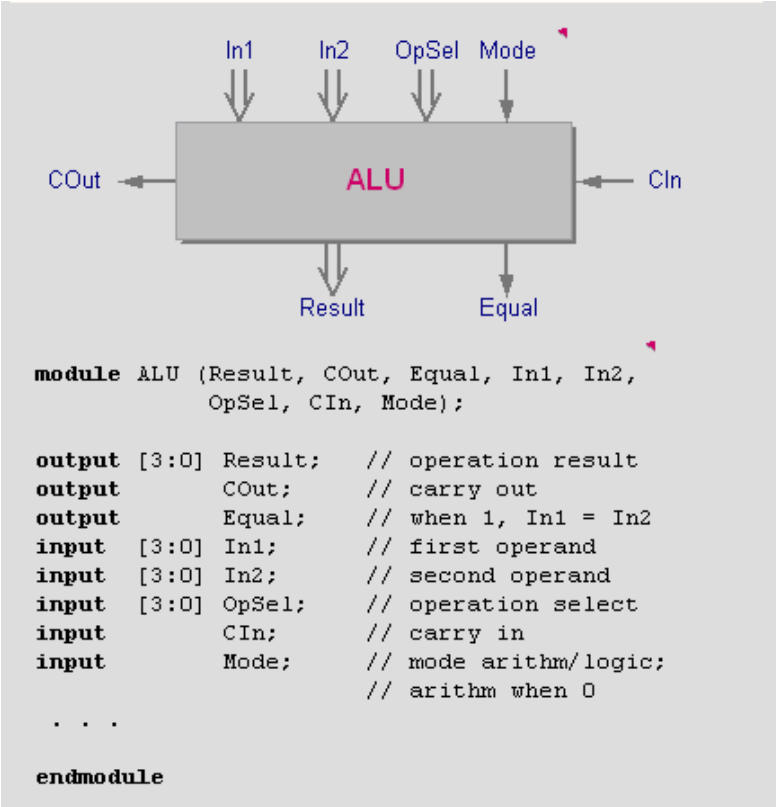
- Evita_verilog Tutorial, www.aldec.com

- http://www.asic-world.com/verilog/

Review: Module definition

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






```
module ALU (Result, COut, Equal, In1, In2,
            OpSel, CIn, Mode);

output [3:0] Result;   // operation result
output       COut;     // carry out
output       Equal;    // when 1, In1 = In2
input  [3:0] In1;      // first operand
input  [3:0] In2;      // second operand
input  [3:0] OpSel;    // operation select
input        CIn;      // carry in
input        Mode;     // mode arithm/logic;
                       // arithm when 0
 . . .

endmodule
```

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;
```

Sensitive to changes on all of the variables it reads

Y is always assigned

- Sequential:

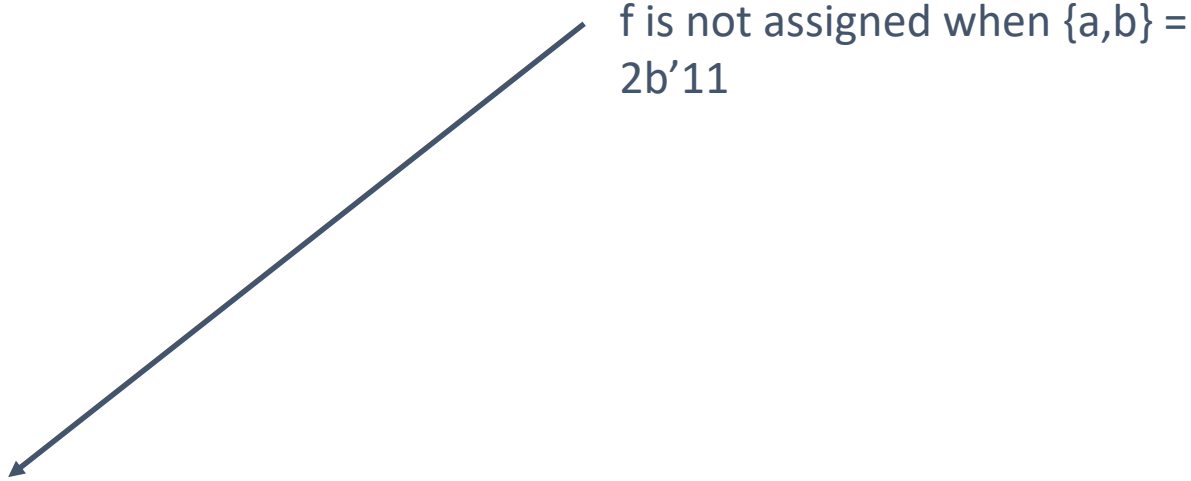
```
reg q;
always @(d or clk)
  if (clk) q = d;
```

q only assigned when clk is 1

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
```



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
```

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