Synthesis

Nachiket Kapre
nachiket@uwaterloo.ca
UNIVERSITY OF WATERLOO

Outline

- Introduction to Synthesis
- Comparing with Software Patterns
- Design practices
  - Parameters
  - Bit-level Access
  - Conditions
  - Precision
  - Registers

Simulation vs. Synthesis

- Synthesizable Verilog is a subset of the language
- Remember, Verilog can produce hardware circuits with **fixed**, **finite** structure
- Be aware of synthesis pitfalls $\rightarrow$ careless style will generate more hw than necessary, and also incorrect hw

Synthesis Process

- In software, there is no real distinction. You can, however, test for functionality in low-performance (debug) mode vs. deployment mode (fully optimized).
- Software compilers translate C code into machine code (x86, ARM, MIPS, RISC-V,...)
- Library of instruction sets available unique to each machine (x86, ARM, MIPS, RISC-V,...)

Software C compilation x86_64 vs. ARMv7

gcc -c -O3 poly.c -S

```
.cfi_startproc
imull  %edi, %esi
addl  %esi, %edx
imull  %edx, %edi
addl  %edx, %edi
movl  %edi, (%r8)
ret
.cfi_endproc
```

```
mla  r2, r0, r1, r2
ldr  r1, [sp]
mla  r3, r2, r0, r3
str  r3, [r1]
bx  lr
```

- Compile `poly.c` on Intel (left) and ARM (right) chips using gcc.
- ARM has a special fused multiply-accumulate integer instruction `MLA`, but x86 does not.
- Thus, same C code is *synthesized* differently on the two platforms with different ISAs
- Hardware compilation also dependent on library (Xilinx or Intel FPGA library, TSMC or UMC ASIC library)

Software C compilation x86_64 vs. ARMv7

```
gcc [illegible] -O3 poly.c -S
```

Separate

```
[illegible] .cfi_proc
imull  %edi, %esi
addl   %esi, %edx
imull  %edx, %edi
addl   %edx, %edi
mull   %edi, (%r8)
```

Separate

Fused

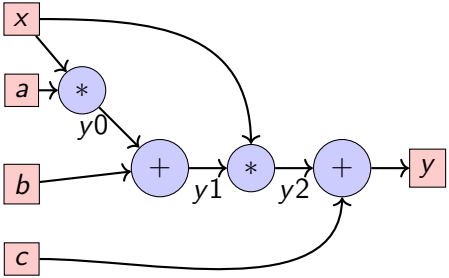
```
mla  r2, r0, r1, r2
ldr  r1, [sp]
mla  r3, r2, r0, r3
str  r3, [r1]
bx
```

Fused

- Compile `poly.c` on Intel (left) and ARM (right) chips using gcc.
- ARM has a special fused multiply-accumulate integer instruction `MLA`, but x86 does not.
- Thus, same C code is *synthesized* differently on the two platforms with different ISAs
- Hardware compilation also dependent on library (Xilinx or Intel FPGA library, TSMC or UMC ASIC library)

Hardware Compilation (Spatial Computation)

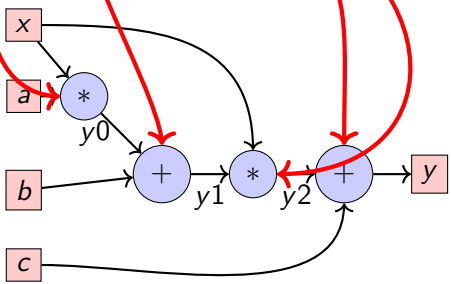
```
always @(*) begin
  y0 = a * x;
  y1 = y0 + b;
  y2 = y1 * x;
  y  = y2 + c;
end
```



- Hardware is **assembled** in space by inferring computation and connecting it through signals
  - Recall, software code is assembled in time, mapped to instructions, and connected through registers/memory
- Each concurrent statement, or always block infers a piece of hardware
- Stitch hardware together by processing the RTL file

Hardware Compilation (Spatial Computation)

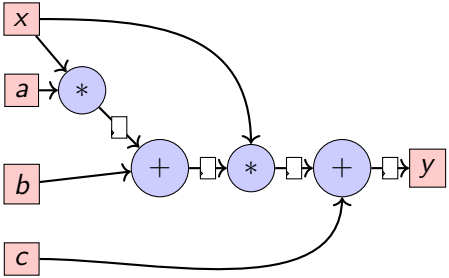
```
always @(*) begin
  y0 = a * x;
  y1 = y0 + b;
  y2 = y1 * x;
  y  = y2 + c;
end
```



- Hardware is **assembled** in space by inferring computation and connecting it through signals
  - Recall, software code is assembled in time, mapped to instructions, and connected through registers/memory
- Each concurrent statement, or always block infers a piece of hardware
- Stitch hardware together by processing the RTL file

Register-Transfer Compilation (Spatial Computation)

```
always @(posedge clk) begin
  y0 <= a * x;
  y1 <= y0 + b;
  y2 <= y1 * x;
  y  <= y2 + c;
end
```



- When `@(posedge clk)` is on the sensitivity list, we are now generating sequential logic.
- Unlike combinational logic (previous slide), sequential logic contains registers.
- The resulting clock period (frequency), and latency (number of cycles from input→ output) are affected.

Managing time in CPU computations

- In software, the compiler + CPU splits and executes a task as a series of instructions
- One instruction takes one cycle (simple model, complexities due to pipelining, multiple-issue, out-of-order, caches, external DRAM latencies)
- Programmer does not care about clock frequency of the processor or how to pack computation into instructions
- You **sacrifice** this control to simplify programming → focus on your task instead of mapping considerations

Managing time in Digital circuits

- In hardware, programmer has to explicitly manage time – both frequency of clock + pack logic into register stages
- **RTL** abstraction = Register Transfer Level
- For instance, can pack $a \cdot x^2 + b \cdot x + c$ into a single cycle $\rightarrow$ maximum frequency will be low
- Or, you can split the task into two cycles, do $y1 <= a \cdot x + b$ in first cycle and $y <= y1 \cdot x + c$ in the second cycle $\rightarrow$ can operate at $2\times$ clock
- Manually chop and pack logic into these stages. Method to this madness (See Pipelining lecture)
- Precisely control timing behavior of signals for interfacing, handshakes, external IO

Example of Pipelined RTL engineering

```
module poly(
    input wire clk,
    input wire rst,
    input wire [7:0] x,
    input wire [7:0] a,
    input wire [7:0] b,
    input wire [7:0] c,
    output reg [23:0] y
  );

  always @(posedge clk) begin
    if(rst == 1'b1) begin
      y <= 0;
    end else begin
      y <= a * x * x + b * x + c;
      //
    end
  end

endmodule
```

```
module poly (
    input wire clk,
    input wire rst,
    input wire [7:0] x,
    input wire [7:0] a,
    input wire [7:0] b,
    input wire [7:0] c,
    output reg [23:0] y
  );

  reg [23:0] y_temp;

  always @(posedge clk) begin
    if(rst == 1) begin
      y_temp <= 0;
      y <= 0;
    end else begin
      //
      y_temp <= a * x + b;
      y <= y_temp * x + c;
      //
    end
  end
```

Example of Pipelined RTL engineering

```
module poly(
    input wire clk,
    input wire rst,
    input wire [7:0] x,
    input wire [7:0] a,
    input wire [7:0] b,
    input wire [7:0] c,
    output reg [23:0] y
  );

  always @(posedge clk) begin
    if(rst == 1'b1) begin
      y <= 0;
    end else begin
      y <= a * x * x + b * x + c;
      //
    end
  end

endmodule
```

```
module poly (
    input wire clk,
    input wire rst,
    input wire [7:0] x,
    input wire [7:0] a,
    input wire [7:0] b,
    input wire [7:0] c,
    output reg [23:0] y
  );

  reg [23:0] y_temp;

  always @(posedge clk) begin
    if(rst == 1) begin
      y_temp <= 0;
      y
      y                    c;
      //
    end
  end
```

Is this correct?
Pipelining Lecture

Verilog Parameters

- Parameters allow same Verilog code to be reused in your design with different settings

```
parameter LENGTH = 8,
parameter i = 8,
```

- Design is **parameterized** so we can construct the exact hardware required by specifying values
- Parameter values must be known at **compile** time since we want fixed-sized hardware
- Analogous to `template` in C++
- Runtime parameters must be loaded/read from registers

Verilog Bit-level Access

- Signals can be handled at a bit-level in Verilog

```
a_c <= a[8*i-1:8*(i-1)];
y[24*i-1:24*(i-1)] <= y_c;
```

- You can read and/or write specific ranges within the signal array as required
- The indexing arithmetic can depend on generic parameters like `i` and must be resolved at compile time
  - Dynamic indexing is possible, and required for memories

C vs. Verilog – Handling conditional execution

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
 int a=3, b=2, c=1;
 int x=atoi(argv[1]), y;
 if(x>0) {
  y = a*x*x+b*x+c;
 } else {
  y = a*x*x-b*x+c;
 }
 return y;
}
```

- In software, compiler inserts BRANCH or JUMP instructions around the then-else code blocks.
- Compute value of `if` condition first.
- Based on value decide where to branch/jump.
- You only ever use CPU cycles to operate on the portion of code that matters.

C vs. Verilog – Handling conditional execution

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
 int a=3, b=2, c=1;
 int x=atoi(argv[1]), y;
 if(x>0) {
  y = a*x*x+b*x+c;
 } else {
  y = a*x*x-b*x+c;
 }
 return y;
}
```

```
        .cfi_startproc
        subq        $8, %rsp
        .cfi_def_cfa_offset 16
        movq        8(%rsi), %rdi
        movl        $10, %edx
        xorl        %esi, %esi
        call        strtol
        testl        %eax, %eax
        leal        (%rax,%rax,2), %edx
        jle        .L2
        addl        $2, %edx
        imull        %edx, %eax
        addl        $1, %eax
.L3:
        addq        $8, %rsp
        .cfi_remember_state
        .cfi_def_cfa_offset 8
        ret
.L2:
        .cfi_restore_state
        subl        $2, %edx
        imull        %edx, %eax
        addl        $1, %eax
        jmp        .L3
```

gcc -S -O3 c-poly-if.c

C vs. Verilog – Handling conditional execution

```
module poly_if (
 input wire signed [7:0] a,
 input wire signed [7:0] b,
 input wire signed [7:0] c,
 input wire signed [7:0] x,
 output reg signed [23:0] y
);

 always @(*) begin
  if(x>0) begin
   y <= a*x*x + b*x + c;
  end else begin
   y <= a*x*x - b*x + c;
  end
 end

endmodule
```

- In hardware, both condition branches are implemented in hardware + **multiplexer**
- Potentially wasteful as in any given instance, we only use one branch
- Potentially **faster**, as condition evaluation and branch operations are processed in parallel
- `if else` conditions used inside `always` block.
- For conditional, concurrent statements, use `assign` with `cond?then:else` selection syntax.

C vs. Verilog – Handling conditional execution

```
module poly_if (
 input wire signed [7:0] a,
 input wire signed [7:0] b,
 input wire signed [7:0] c,
 input wire signed [7:0] x,
 output wire signed [23:0] y
);

 assign y = (x>0)?
  a*x*x + b*x + c:
  a*x*x - b*x + c;
    //


endmodule
```

- In hardware, both condition branches are implemented in hardware + **multiplexer**
- Potentially wasteful as in any given instance, we only use one branch
- Potentially **faster**, as condition evaluation and branch operations are processed in parallel
- `if else` conditions used inside `always` block.
- For conditional, concurrent statements, use `assign` with `cond?then:else` selection syntax.

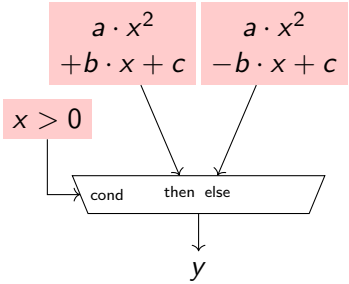
C vs. Verilog – Handling conditional execution

```
module poly_if (
 input wire signed [7:0] a,
 input wire signed [7:0] b,
 input wire signed [7:0] c,
 input wire signed [7:0] x,
 output wire signed [23:0] y
);

 assign y = (x>0)?
  a*x*x + b*x + c:
  a*x*x - b*x + c;
    //

endmo
```

cond?then:else
equivalent
to if else

- In hardware, both condition branches are implemented in hardware + **multiplexer**
- Potentially wasteful as in any given instance, we only use one branch
- Potentially **faster**, as condition evaluation and branch operations are processed in parallel
- `if else` conditions used inside `always` block.
- For conditional, concurrent statements, use `assign` with `cond?then:else` selection syntax.

C vs. Verilog – Handling conditional execution



- In hardware, both condition branches are implemented in hardware + **multiplexer**
- Potentially wasteful as in any given instance, we only use one branch
- Potentially **faster**, as condition evaluation and branch operations are processed in parallel
- `if else` conditions used inside `always` block.
- For conditional, concurrent statements, use `assign` with `cond?then:else` selection syntax.

Incomplete conditions in Verilog

```
module poly_if (
 input wire signed [7:0] a,
 input wire signed [7:0] b,
 input wire signed [7:0] c,
 input wire signed [7:0] x,
 output reg signed [23:0] y
);

 always @(*) begin
  if(x>0) begin
   y <= a*x*x + b*x + c;
  end //
 end

endmodule
```

- If a conditional statement is **incompletely** specified, Verilog will infer a feedback to the multiplexer input (a bug?)
- In combinational circuits, this will create a **latch**
  - Latches are usually bad and hard to analyze for correctness
  - Initial value problem? Timing requirement on select signal
- For sequential circuits, the feedback loop is broken at a register stage.
  - This is safer, but could be unintended

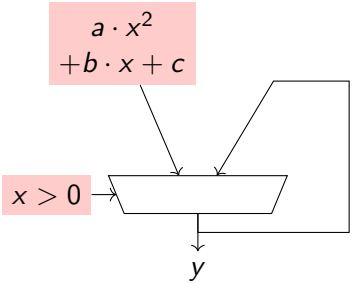
Incomplete conditions in Verilog

```
module poly_if (
 input wire signed [7:0] a,
 input wire signed [7:0] b,
 input wire signed [7:0] c,
 input wire signed [7:0] x,
 output reg signed [23:0] y
);

 always @(*) begin
  if(x>0) begin
   y <= a*x*x + b*x + c;
  end //
 end

endmodu
```

Missing Else

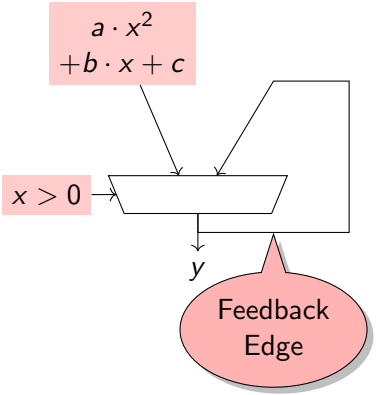
- If a conditional statement is **incompletely** specified, Verilog will infer a feedback to the multiplexer input (a bug?)
- In combinational circuits, this will create a **latch**
  - Latches are usually bad and hard to analyze for correctness
  - Initial value problem? Timing requirement on select signal
- For sequential circuits, the feedback loop is broken at a register stage.
  - This is safer, but could be unintended

Incomplete conditions in Verilog



- If a conditional statement is **incompletely** specified, Verilog will infer a feedback to the multiplexer input (a bug?)
- In combinational circuits, this will create a **latch**
  - Latches are usually bad and hard to analyze for correctness
  - Initial value problem? Timing requirement on select signal
- For sequential circuits, the feedback loop is broken at a register stage.
  - This is safer, but could be unintended

Incomplete conditions in Verilog



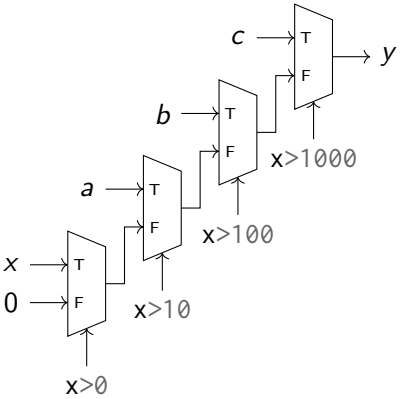
- If a conditional statement is **incompletely** specified, Verilog will infer a feedback to the multiplexer input (a bug?)
- In combinational circuits, this will create a **latch**
  - Latches are usually bad and hard to analyze for correctness
  - Initial value problem? Timing requirement on select signal
- For sequential circuits, the feedback loop is broken at a register stage.
  - This is safer, but could be unintended

Conditions

- For `if else` and `cond?then:else` blocks, we will infer priority chain of multiplexers
- For `case` block, a flat multiplexer structure will be generated
- Pick a style based on problem requirements
- Remember, in hardware all branches are implemented since condition evaluation will happen at runtime

Conditions

```
always @(a or b or c or x) begin
  if(x > 1000) // begin
    y <= c;
  end else if(x > 100) begin //
    y <= b;
  end else if(x > 10) begin
    y <= a;
  end else if(x > 0) begin
    y <= x;
  end else begin
    y <= 0; //
  end
end
```



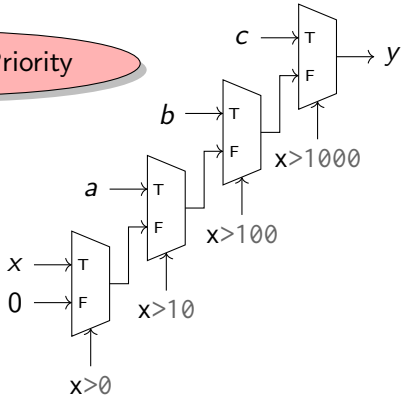
Conditions

```
always @(a or b or
  if(x > 1000) //
    y <= c;
  end else if(x > 100) begin //
    y <= b;
  end else if(x > 10) begin
    y <= a;
  end else if(x > 0) begin
    y <= x;
  end else begin
    y <= 0; //
  end
end
```

Highest Priority

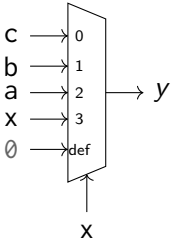
Next Highest Priority

Final Else



Conditions

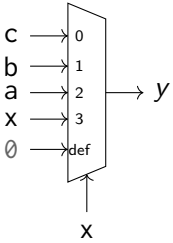
```
always @(a or b or c or x) begin
  case(x) //
  8'h00 : begin
    y <= c;
  end
  8'h01 : begin
    y <= b;
  end
  8'h02 : begin
    y <= a;
  end
  8'h03 : begin
    y <= x;
  end
  default : begin
    y <= 0;
  end
  endcase
end
```



Conditions

```
always @(a or b or
  case(x) //
  8'h00 : begin
    y <= c;
  end
  8'h01 : begin
    y <= b;
  end
  8'h02 : begin
    y <= a;
  end
  8'h03 : begin
    y <= x;
  end
  default : begin
    y <= 0;
  end
  endcase
end
```

Mutual Exclusion

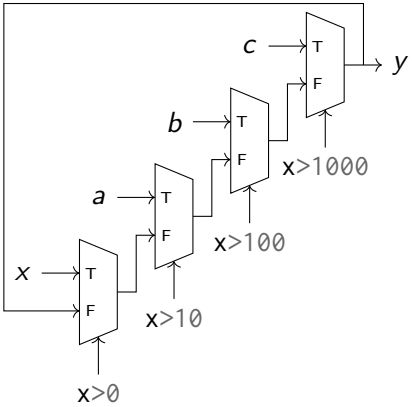


Incomplete Conditions

- If a conditional statement is **incompletely** specified, the compiler will infer a feedback to the multiplexer input (a bug?)
- In combinational circuits, this will create a **latch**
- Latches are usually bad and hard to analyze for correctness
- Initial value problem? Timing requirement on select signal
- For sequential circuits, the feedback is broken at a register stage. (This is safer, but could be unintended)

Incomplete conditions

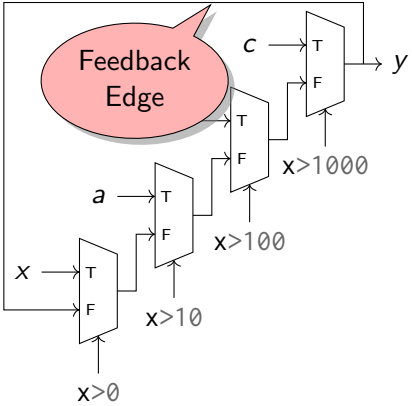
```
always @(a or b or c or x)
begin
  if(x > 1000) begin
    y <= c;
  end else if(x > 100) begin
    y <= b;
  end else if(x > 10) begin
    y <= a;
  end else if(x > 0) begin
    y <= x; //
  end
```



Incomplete conditions

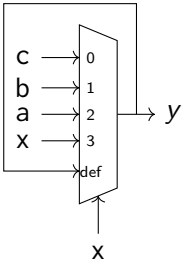
```
always @(a or b or c or x)
begin
  if(x > 1000) begin
    y <= c;
  end else if(x > 100) begin
    y <= b;
  end else if(x > 10) begin
    y <= a;
  end else if(x > 0) begin
    y <= x; //
  end
```

Missing Else



Incomplete conditions

```
always @(a or b or c or x) begin
  case(x)
  8'h00 : begin
    y <= c;
  end
  8'h01 : begin
    y <= b;
  end
  8'h02 : begin
    y <= a;
  end
  8'h03 : begin
    y <= x;
  end
  endcase
```



Reordering conditional Code

```
always @(posedge clk) begin
 if(rst) begin
  x<=1;
 end else begin
  if(x<4) begin
   x<=x+1;
  end else begin
   x<=4;
  end
 end
end
```

- A fully-specified **if else** block can be made to behave similar to one with a missing **else**
- The implication is simple →
  - Missing **else** produces a feedback edge
  - Fully-specified **if else** may not
- Default condition could be useful → just ensure correct sequential ordering!

Reordering conditional Code

```
always @(posedge clk) begin
 if(rst) begin
  x<=1;
 end else begin
  if(x<4) begin
   x<=x+1;
  end //
 end
end
```

Drop Else

- A fully-specified **if else** block can be made to behave similar to one with a missing **else**
- The implication is simple →
  - Missing **else** produces a feedback edge
  - Fully-specified **if else** may not
- Default condition could be useful → just ensure correct sequential ordering!

Reordering conditional Code

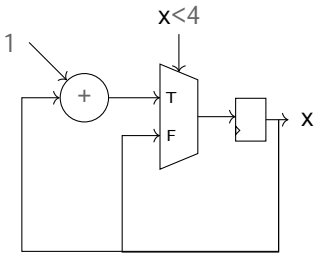
```
always @(po
 if(rst) be
  x<=1;
 end else be
  x<=4; //
  if(x<4) begin
   x<=x+1;
  end
 end
end
```

Add Default Before if

- A fully-specified **if else** block can be made to behave similar to one with a missing **else**
- The implication is simple $\rightarrow$
  - Missing **else** produces a feedback edge
  - Fully-specified **if else** may not
- Default condition could be useful $\rightarrow$ just ensure correct sequential ordering!

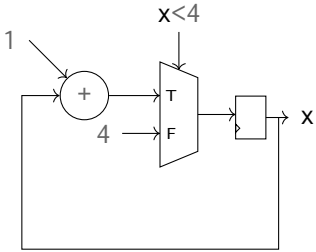
Hardware generated for missing else design

```
always @(posedge clk) begin
 if(rst) begin
  x<=1;
 end else begin
  if(x<4) begin
   x<=x+1;
  end //
 end
end
```



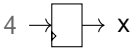
Hardware generated for default assignment

```
always @(posedge clk) begin
 if(rst) begin
  x<=1;
 end else begin
  x<=4; //
  if(x<4) begin
   x<=x+1;
  end
 end
end
```



Impact of a small error in code!

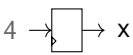
```
always @(posedge clk) begin
 if(rst) begin
  x<=1;
 end else begin
  if(x<4) begin
   x<=x+1;
  end
  x<=4; //
 end
end
```



Impact of a small error in code!

```
always @(posedge clk) begin
 if(rst) begin
  x<=1;
 end else begin
  if(x<4) begin
   x<=x+1;
  end
  x<=4; //
 end
end
```

Wrong place for assignment



Everything dissolved

Bits and Precision

- Ability to perform bit-level operations is crucial for hardware design
- Sometimes want configurable accuracy, support for binary operations
  - Logic operations for crypto, interface controls
  - Arithmetic operations with exact number of bits for algorithms
- C code/software generally makes this tricky
  - Bithacks → `https://graphics.stanford.edu/~seander/bithacks.html`
  - MPFR library → `http://www.mpfr.org/`

Bits and Precision

```
module poly(
  input wire [7:0] x,
  input wire [7:0] a,
  input wire [7:0] b,
  input wire [7:0] c,
  output wire [7:0] y //
 );

 assign y = a * x * x + b * x + c;

endmodule
```

- Verilog is not strongly-typed like VHDL
- Precision errors are not flagged at compile time!
- Need to remember certain rules:
  - **Addition—Subtraction**: 1 extra bit than the largest operand precision
  - **Multiplication**: Sum of precision of two inputs
  - **Division, Square Root, etc**???
- Sign bit handling $\rightarrow$ unsigned and signed types.
  - $0 \rightarrow (2^N - 1)$
  - $(-2^{N-1}) \rightarrow (2^{N-1} - 1)$

Bits and Precision

```
module poly
  input
  input
  input wire
  input wire [7:0] c,
  output wire [7:0] y //
);

 assign y = a * x * x + b * x + c;

endmodule
```

8b result
Verilog says nothing

- Verilog is not strongly-typed like VHDL
- Precision errors are not flagged at compile time!
- Need to remember certain rules:
  - **Addition—Subtraction**: 1 extra bit than the largest operand precision
  - **Multiplication**: Sum of precision of two inputs
  - **Division, Square Root, etc**???
- Sign bit handling $\rightarrow$ unsigned and signed types.
  - $0 \rightarrow (2^N - 1)$
  - $(-2^{N-1}) \rightarrow (2^{N-1} - 1)$

Bits and Precision

```
module poly(
  input wire [7:0] x,
  input wire [7:0] a,
  input wire [7:0] b,
  input wire [7:0] c,
  output wire [23:0] y //
 );

 assign y = a * x * x + b * x + c;

endmodule
```

- Verilog is not strongly-typed like VHDL
- Precision errors are not flagged at compile time!
- Need to remember certain rules:
  - **Addition—Subtraction**: 1 extra bit than the largest operand precision
  - **Multiplication**: Sum of precision of two inputs
  - **Division, Square Root, etc**???
- Sign bit handling $\rightarrow$ unsigned and signed types.
  - $0 \rightarrow (2^N - 1)$
  - $(-2^{N-1}) \rightarrow (2^{N-1} - 1)$

Bits and Precision

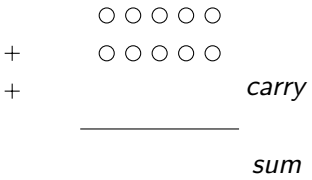
```
module poly(
  input wire
  input wire
  input wire [7
  input wire [7:0] c,
  output wire [23:0] y //
 );

 assign y = a * x * x + b * x + c;

endmodule
```

24b needed for correctness

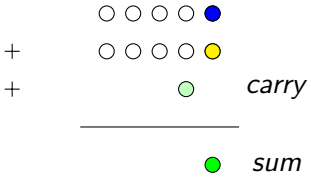
- Verilog is not strongly-typed like VHDL
- Precision errors are not flagged at compile time!
- Need to remember certain rules:
  - **Addition—Subtraction**: 1 extra bit than the largest operand precision
  - **Multiplication**: Sum of precision of two inputs
  - **Division, Square Root, etc**???
- Sign bit handling $\rightarrow$ unsigned and signed types.
  - $0 \rightarrow (2^N - 1)$
  - $(-2^{N-1}) \rightarrow (2^{N-1} - 1)$


Verilog compiles anything. Use at own risk.
WeKnowMemes

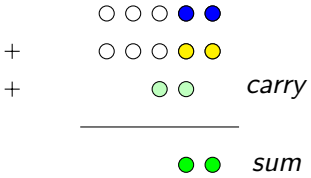
Addition — Subtraction



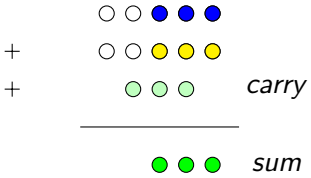
Addition — Subtraction



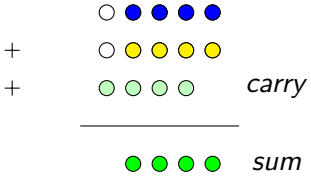
Addition — Subtraction



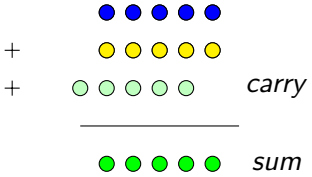
Addition — Subtraction



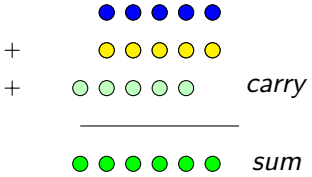
Addition — Subtraction



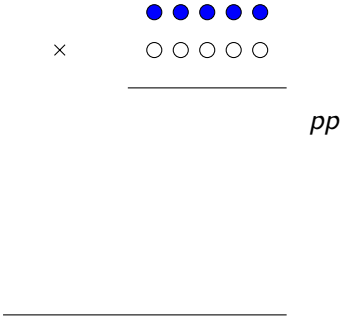
Addition — Subtraction



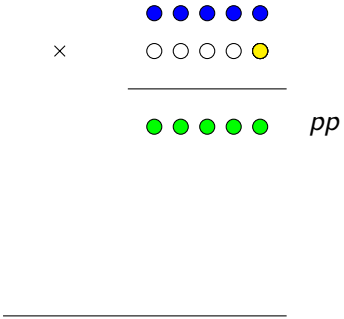
Addition — Subtraction



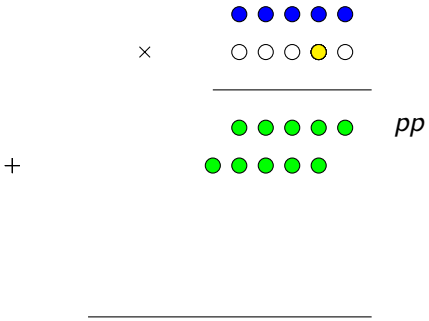
Multiplication



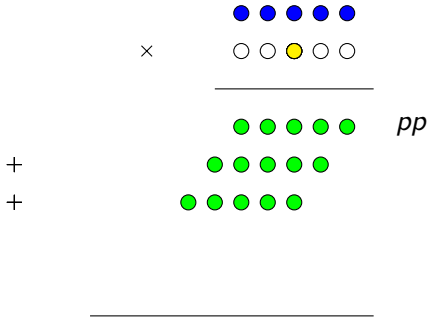
Multiplication



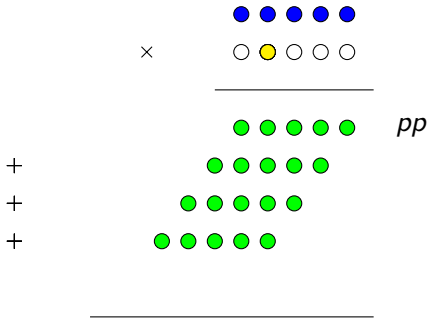
Multiplication



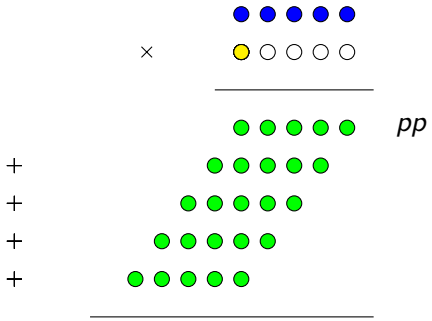
Multiplication



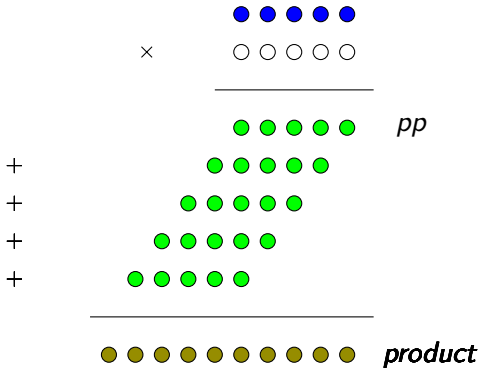
Multiplication



Multiplication



Multiplication



Registers

```
module register (
  input wire clk,
  input wire rst,
  input wire d,
  input wire ce,
  output wire q);

 reg q_int = 1; // INIT

 always @(posedge clk) begin
  if(rst) begin
   q_int <= 0; // SRVAL
  end else if(ce) begin
   q_int <= d;
  end
 end

 assign q = q_int;
endmodule
```

- Registers hold state, intermediate pipeline results
- Initializing and correct use of registers = important!
- Careless coding can cause unpredictable results in physical hardware
- **Caveat 1**: Latch inference
- **Caveat 2**: Resetting a register
  - https://www.eetimes.com/document.asp?doc_id=1278998

Understanding registers

Async Reset
on Sensitivity List

```
//
always @(posedge clk or
  posedge async_rst) begin

 if(async_rst) begin
  q<=0;
 end else begin
  q<=d;
 end
end
```

- Registers are key building blocks in RTL designs →
  - Registers hold state, data moves from D input to Q output on clock edge
- (1) Sync. vs. Async. resets →
  - Async. resets are *usually* bad as they may occur too close to clock edge → metastability.
  - Sync. resets preferred as they can be synchronized carefully
- (2) Enable signals control if Q is allowed to be updated → mux
- (3) Initial value of signal connected to register output only meaningful for FPGAs → applied at powerup

Understanding registers

Sync Reset
not on
Sensitivity List

```
//
always (@posedge clk) begin
 if(sync_rst) begin
  q<=0;
 end else begin
  q<=d;
 end
end
```

- Registers are key building blocks in RTL designs →
  - Registers hold state, data moves from D input to Q output on clock edge
- (1) Sync. vs. Async. resets →
  - Async. resets are *usually* bad as they may occur too close to clock edge → metastability.
  - Sync. resets preferred as they can be synchronized carefully
- (2) Enable signals control if Q is allowed to be updated → mux
- (3) Initial value of signal connected to register output only meaningful for FPGAs → applied at powerup

Understanding registers

```
always (@posedge clk) begin
 if(sync_rst) begin
  q<=0;
 end else begin
  if(enable) begin
   q<=d;
  end
 end
end
```

- Registers are key building blocks in RTL designs →
  - Registers hold state, data moves from D input to Q output on clock edge
- (1) Sync. vs. Async. resets →
  - Async. resets are *usually* bad as they may occur too close to clock edge → metastability.
  - Sync. resets preferred as they can be synchronized carefully
- (2) Enable signals control if Q is allowed to be updated → mux
- (3) Initial value of signal connected to register output only meaningful for FPGAs → applied at powerup

Understanding registers

```
//
reg [3:0] q = 4'b1111;

always @(posedge clk) begin
 if(sync_rst) begin
  q<=4'b00000; //
 end else begin
  if(enable) begin
   q<=d;
  end
 end
end
```

- Registers are key building blocks in RTL designs →
  - Registers hold state, data moves from D input to Q output on clock edge
- (1) Sync. vs. Async. resets →
  - Async. resets are *usually* bad as they may occur too close to clock edge → metastability.
  - Sync. resets preferred as they can be synchronized carefully
- (2) Enable signals control if Q is allowed to be updated → mux
- (3) Initial value of signal connected to register output only meaningful for FPGAs → applied at powerup

Un[illegible]gisters

Register value on powerup before clk—rst

```
//
reg [3:0] q = 4'b1111;

always @(posedge clk) begin
 if(sync_rst) begin
  q<=4'b00000; //
 end else begin
  if(enable) begin
   q<=d;
  end
 end
end
```

Register value on Sync Reset

- Registers are key building blocks in RTL designs →
  - Registers hold state, data moves from D input to Q output on clock edge
- (1) Sync. vs. Async. resets →
  - Async. resets are *usually* bad as they may occur too close to clock edge → metastability.
  - Sync. resets preferred as they can be synchronized carefully
- (2) Enable signals control if Q is allowed to be updated → mux
- (3) Initial value of signal connected to register output only meaningful for FPGAs → applied at powerup

Understanding Output of Synthesis

- Synthesis + Backend tools produce an executable circuit
  - **ASIC**: The output is sent to a foundry to manufacture chips → execution=manufacturing
  - **FPGA**: Output is a bitstream that can be loaded onto an FPGA chip at boot-time
- In software, we measure QoR (Quality of Result) in terms of speed, lines of code, bugs
- In hardware design, QoR is measured in terms of physical attributes → circuit size, frequency, power, bugs?
- Often, cannot optimize all attributes → area-time tradeoffs common

Software	**Hardware**
Software is sequential description of computation	Hardware in a concurrent, parallel description of your algorithm
Memory and code size is bounded only by DRAM+disk (stack smaller)	Hardware logic+memory capacity is fixed
Software can **defer** decision to **runtime**	Hardware must make all/most decisions are **compile** time
Recursion can be unrolled at software runtime	Hardware compiler must unroll recursion
Arrays can be allocated dynamically at runtime (heap)	Arrays sizes must be statically known at compile time
Pointers are allowed (Language supports exposing address as special type)	Verilog dynamic arrays are only for simulations (Memory addresses are `unsigned` signals)
Encapsulation allows software to expose functions from class to external world	In hardware, the interfaces are explicitly exposed as top-level signals on the `module` declaration.
Software unable to control IO ports with precise timing unless CPU has real-time properties	Hardware has cycle-exact control over IO

Wrapup

- Synthesis translates RTL code into physical hardware
- Executable software patterns do not translate to generation of digital hardware directly $\rightarrow$ focus on compile-time optimizations
- Conditional code in Verilog often exposes simulation-synthesis mismatches
- Synthesis also requires careful attention to detail like number of bits