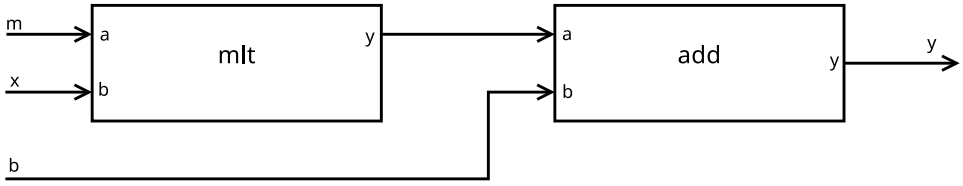
Introduction (1 of 3)

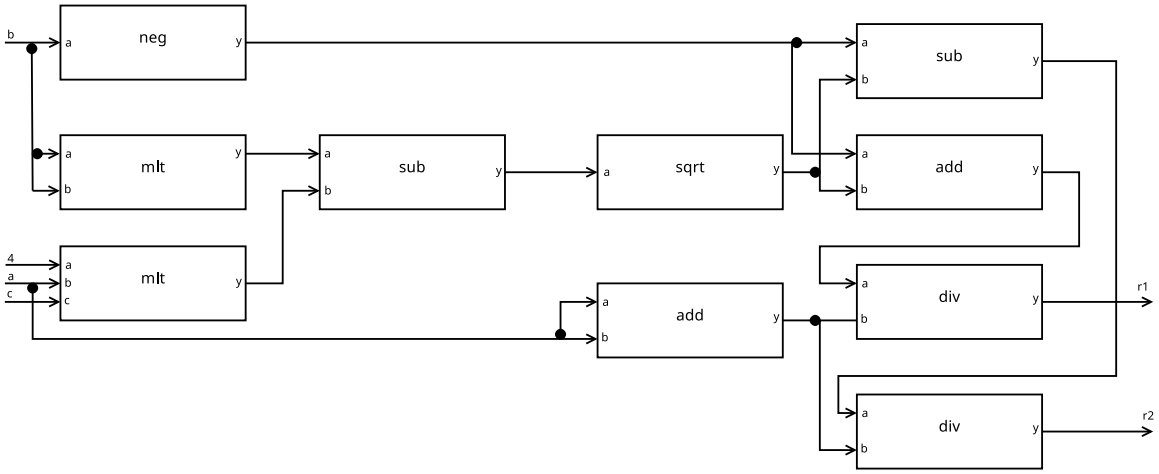
- Verilog was designed by Prabhu Goel, Phil Moorby, and Chi-Lai Huang, in 1984. Its name derives from "verification" and "logic."
- Verilog is one of our representatives of the *declarative* language paradigm. In particular, it is a *dataflow* language.
- A dataflow program can be seen as a graph: a collection of computation nodes, connected by communication paths.
- Some paths provide input data to the program. The program's output data becomes available on other paths.
- Computation happens in parallel, as does communication.

Introduction (2 of 3)

- This is an example dataflow graph, for the slope-intercept formula $y = mx + b$:



- This is an example dataflow graph, for the quadratic formula $(r_1, r_2) = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$:



Introduction (3 of 3)

- In Verilog, a computation node is called a *module*. A program providing input data to, and gathering output data from, one or more connected modules, is called a *testbench*. A module can instantiate submodules, to any depth.
- Verilog is a *hardware description language* (HDL). A module can describe a digital-logic system comprising wires, gates, flip-flops, and registers. Its logic can be combinational (e.g., unclocked) and/or sequential (e.g., clocked).

Translation (1 of 3)

- A source program can be translated in different ways. Some formats can be "executed" via simulation on a regular computer. Other formats can be written (aka, programmed) into the non-volatile *configuration flash memory* (CFM) of a *programmable logic device* (PLD).
- Examples of such integrated circuits (ICs) are the *complex programmable logic device* (CPLD) and the *field-programmable gate array* (FPGA).
- CPLD programming is said to "configure" the device's logic. The device then behaves as the program describes.
- Translating a source program into a device-specific programmable format is also called "synthesis" and "fitting."

Translation (2 of 3)

- We will focus on simulation, but I'll demonstrate synthesis and device programming, later.
- Our translator is named `iverilog`, the Icarus Verilog compiler. Its simulator is named `vvp`, the Verilog virtual processor.
- Verilog is in the public domain, and has been standardized. Yet, many dialects have evolved. Their differences and incompatibilities, can be confusing.
- Verilog looks (dangerously) like C, endowed with the elegance of FORTRAN.

Translation (3 of 3)

- Another confusing aspect is that some language constructs cannot be synthesized to hardware. They are only for simulation, and cannot be realized as logic components.
- For example, a loop only "happens" during translation and simulation. During translation, a loop can repetitively synthesize logic, allowing compact source code. During simulation, a loop can behave like a typical loop, simplifying a testbench.

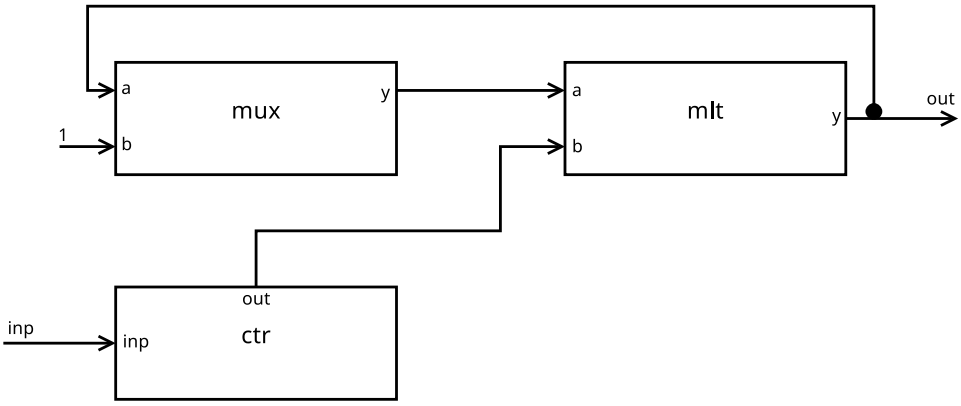
Example: Factorials (1 of 6)

- Suppose we want to compute factorials: $y = \prod_{i=1}^{x} i$, better known as $y = x!$.
- We can do so iteratively or recursively:

  pub/etc/fact/fact-itr.c

  pub/etc/fact/fact-rec.c

  Of course, there are other solutions.
- This is a simplified dataflow graph:



- This graph is quite different than those for the slope-intercept and quadratic formulae! It has a feedback path, for repetition, analogous to the `while` loop in the iterative C function.

Example: Factorials (2 of 6)

- A feedback path forces us to address realities we have been ignoring: time, control, and status.
- To hope that a node (e.g., `mlt`) "knows" when data arrives at its inputs is wishful thinking. A node must be told when new data is available, at *each* of its inputs, so it can compute its outputs.
- Opportunities for concurrency increase if a node allows inputs to arrive independently and asynchronously.
- With this in mind, we can add, to each input *port*, a one-bit *input-enable* (`IE`) control line.
- Then, we can add, to each output port, a one-bit *output-enable* (`OE`) status line. A node's `OE` line(s) often drive `IE` line(s) of neighboring nodes.

Example: Factorials (3 of 6)

- When should a node sample/inspect an `IE` line? We can add a one-bit clock (`CLK`), and route its output to every node's `CLK` input port. This is analogous to a regular CPU's global system clock.
- Putting all of these ideas together, I drew this picture of the detailed graph:

  `pub/la2/fact/fact-drawn.pdf`

  I am showing you the hand-drawn version, because you'll want to design your solution to the homework assignment with a similar picture. Don't try to just write the Verilog source code for module interconnections!
- Instead, follow these steps (next slide).

Example: Development Steps

1. Draw an initial graph, using modules you have, and those you will need to create.
2. Then, *debug your graph*. Repeat.
3. When you think your graph is bug-free, write the top-level module (e.g., `Fact.v`), and any new ones it needs (e.g., `Mlt.v`).
4. For each new module, write a testbench (e.g., `test.v`), which instantiates and exercises the module. Use `$monitor()` and `$display()` to debug it.
5. Finally, write a testbench (e.g., `test.v`), which instantiates and exercises your top-level module.

Example: Factorials (4 of 6)

- We'll come back to the top-level module, `Fact`, after we consider the lower-level modules: `Mlt`, `Mux`, and `Ctr`.
- This is the multiplier, and a testbench:

  pub/la2/mlt/Mlt.pdf

  pub/la2/mlt/test.v

  pub/la2/mlt/Mlt.v

  The `initial` block uses delays (e.g., `#30`) to cause the simulator to sequence through the statements. Hardware synthesis does not support this.
- This is the multiplexer, and a testbench:

  pub/la2/mux/Mux.pdf

  pub/la2/mux/test.v

  pub/la2/mux/Mux.v

  One of the inputs is hard-wired to one, as we saw in the detailed graph.

Example: Factorials (5 of 6)

- The counter is a bit more complex:
  - pub/la2/ctr/Ctr.pdf
  - pub/la2/ctr/test.v
  - pub/la2/ctr/Ctr.v
- It can count up or down, from a first value (`INP`) to a last value (`STOP`). For factorials, it counts down, from `INP` to one. The testbench counts up, from one to nine.
- It indicates when it is starting a new count (`INIT`), or finished (`DONE`).
- It counts as the count-enable input port (`CE`) changes from true to false.
- At each count, the output-enable output port (`OE`) is true for only one `CLK` cycle. Its duration can be increased via the three-bit `CLKS` parameter.
- The module implements a state machine, but the values of `IE` and `DONE` can "override" the current state.

Example: Factorials (6 of 6)

- Now, let's return to the top-level module.
- Intel's Quartus hardware-development tool (see below) drew this detailed graph, corresponding to my hand-drawn picture:

  `pub/la2/fact/Fact.pdf`

  `pub/la2/fact/test.v`

  `pub/la2/fact/Fact.v`
- The testbench instantiates the top-level module, `Factorial`. Then, the testbench stimulates `Factorial`'s inputs, and displays results from its outputs.
- `Factorial` instantiates three lower-level modules, `wire`-ing them together, according to our detailed graph. You should compare this module to our graph, very carefully.

Segue to Homework

- The Verilog Homework assignment asks you to transform this factorial generator into a Fibonacci generator.
- *handout/discuss homework*

Synthesis (1 of 5)

- Now that you are working on the Verilog homework assignment, eventually simulating generation of Fibonacci numbers, I want to show you how the factorial simulation can be synthesized to a particular semiconductor device (i.e., hardware).
- I'm using an Altera EPM570 MAX V chip, a *complex programmable logic device* (CPLD) (it costs $10).
- The chip is soldered onto a circuit board, called an "Altera MAX V CPLD Development System – UnoProLogic," from Earth People Technology, Inc., which I call "EPT" (it cost $24).
- EPT's external I/O buses are connected to those of a "piggy-backed" Arduino Uno, which I call "ARD" (it cost $6).

Synthesis (2 of 5)

- ARD and EPT are USB-connected via a Raspberry Pi with WiFi ($25). Pictures:

  pub/la2/ard_fact/pix/devsys.pdf

  pub/la2/ard_fact/pix/apart.pdf

  pub/la2/ard_fact/pix/pi-apart.pdf

  pub/la2/ard_fact/pix/together.pdf

  This ARD/EPT "shield" arrangement allows ARD to write/read 8-bit data to/from EPT, providing EPT with input data, and retrieving its output.
- BTW: This material is presentational, and hopefully inspirational. You will *not* be asked to work with hardware or hardware-development tools.

Synthesis (3 of 5)

To get from Verilog source code to semiconductors, we will:

1. Replace our old testbench `fact/test.v`, with a new one, which has ports and a new module for the EPM570's hardware resources (e.g., IC pins).
2. Switch to a new Verilog toolchain, which is specialized for CPLDs like our EPM570. We can still use Icarus Verilog for simulation. For synthesis, we can use Intel's Quartus IDE (Intel has a free "lite" version.)
3. Use Quartus device-programming tools to program EPT, over its USB channel.
4. Use Arduino software-development tools to program ARD, over its USB channel. An Arduino program is called a "sketch".

Synthesis (4 of 5)

With those preliminaries completed, we can do the following, on any computer:

1. Connect the computer to ARD, with a USB cable.
2. Run a terminal emulator on the computer, using ARD's serial device.
3. Use the computer's keyboard and screen to interact with ARD's sketch, which runs perpetually.
4. What we type on the keyboard is written to EPT by ARD, over their piggy-backed I/O buses. What we see on the screen is read from EPT by ARD, over those buses.

Synthesis (5 of 5)

- Our new testbench has ports that interface `Fact.v` with EPM570 pins. The IC has 100 pins. For example, pin 12 is soldered to a circuit-board trace connected to EPT's 66Mhz clock, which we've been calling `CLK`.
- The testbench instantiates a new module, which uses these ports/pins to implement an ARD/EPT 8-bit I/O interface, over our `Factorial` module's `INP` and `OUT` ports:

  `pub/la2/ard_fact/ard_fact.v`