LSU EE 4755 Homework 4 Due: 18 November 2026

Formatted 19:02, 11 November 2026

For instructions visit `https://www.ece.lsu.edu/koppel/v/proc.html`. *For the complete Verilog for this assignment without visiting the lab follow* `https://www.ece.lsu.edu/koppel/v/2026/hw04.v.html`.

Student Expectations

To solve this assignment students are expected to avail themselves of references provided in class and on the Web site, such as for Verilog programming and synthesis examples, and to seek out any additional help and resources that might be needed. (Of course this doesn't mean asking someone else to solve it for you.) It is each student's duty to himself or herself to resolve frustrations and roadblocks quickly. (If you get stuck *just ask for help!*)

This assignment cannot be solved by blindly pasting together parts of past assignments. Solving the assignment is a multi-step learning process that takes effort, but one that also provides the satisfaction of progress and of developing skills and understanding.

Collaboration Rules

Each student is expected to complete his or her own assignment. It is okay to work with other students and to ask questions in order to get ideas on how to solve the problems or how to overcome some obstacle (be it a question of Verilog syntax, interpreting error messages, how a part of the problem might be solved, etc.) It is also acceptable to seek out digital design resources for help on Verilog, digital design, the IEEE 754 floating-point format, etc. It is okay to make use of AI LLM tools such as Claude and Copilot to generate sample Verilog code. (Do not assume LLM output is correct. Treat LLM output the same way one might treat legal advice given by a lawyer character in a movie: it may sound impressive, but it can range from sage advice to utter nonsense.)

After availing oneself to these resources **each student is expected to be able to complete the assignment alone**. Test questions will be based on homework questions and **the assumed time needed to complete the question will be for a student who had solved—not memorized the solution of—the homework assignment on which it was based**.

Students are expected to write code that follows the conventions presented in class and that avoids features that are considered deprecated as far as this course is concerned. Code not following these conventions will be assumed to have been copied or LLM-generated.

Problem 0: Following instructions at `https://www.ece.lsu.edu/koppel/v/proc.html`, set up your class account (if necessary), copy the assignment, and run the Verilog simulator on the unmodified homework file, `hw04.v`. Do this early enough so that minor problems (*e.g.*, password doesn't work) are minor problems.

Homework Overview

In this assignment a ripple addition is to be performed over several cycles. (Note: One should not really build an adder this way. The point of the assignment is to understand sequential and pipelined implementation. A ripple adder is used because it is familiar.) Let w denote the number of bits in the values to add and n denote the desired cycles of latency. Module `add_seq` is to add $\lceil w/n \rceil$ bits per cycle over n cycles. So the cost of the adder in `add_seq` should be about $\frac{1}{n}$ the cost of a w-bit ripple adder. Module `add_pipe` is to use n ripple $\lceil \frac{w}{n} \rceil$-bit adders. The intended benefit is higher throughput rather than lower cost.

The alert student might remember that the `add_accum` module presented in class makes use of a pipelined adder. Indeed it does, and that module is included in the assignment as reference. The pipelined adder for this assignment is to make use of BFA instantiations, so the `add_accum` pipelined adder would not be a correct solution.

Testbench

To compile your code and run the testbench press F9 in an Emacs buffer in **a properly set up account**. The testbench will test both adder modules instantiated at two widths and several numbers of stages/steps. The first pairs of numbers used to test the adders comes from the array `manual_pairs`. Feel free to add your own test pairs: Search for the array `manual_pairs` and add your own.

Appearing below is are excerpts from the testbench output for a correctly solved assignment:

```
Compilation started at Tue Nov 10 18:06:54

xrun -sv -batch -exit hw04.v
TOOL: xrun(64) 26.03-s005: Started on Nov 10, 2026 at 18:06:54 CST
[snip]
Greetings from add_seq instantiated at w=8, n=2.
Greetings from add_seq instantiated at w=16, n=2.
[snip]
Starting tests for sequential w=8,n=2.
    Cyc --a  --b  S          Cyc-In   Sum         Debug Val
Tr    5  45   12  S  Mod Out    3 -->  xx        000045
Tr    6  06   0d  _  Mod Out    4 -->  70        000006
Tr    7  07   8d  _  Mod Out    5 -->  57 <--   000007
Cr    7 ---  ---    Cor Out    5 -->  57 <-- This sum found.
Tr    8  08   65  _  Mod Out    6 -->  57        000008
Tr    9  09   12  _  Mod Out    7 -->  57        000009
```

The lines starting `Greetings` are printed by debug code in each module. This can be modified to view the contents of parameters and other constants.

The lines starting `Tr` show module inputs and output. The `Cyc` column indicates the cycle number, `a`, `b`, and `S`, should the values at the `a`, `b`, and `start` (sequential only) inputs. The `Sum` column shows the value at the `sum` output. The values of `a`, `b`, and `Sum` are shown in hexadecimal.

The column headed `Debug Val` shows the value of object `watch_value`. This can be set to help you debug the module. In the unmodified assignment it is set to `a`.

Remember that the output of a module should be the sum of a pair of inputs *that appeared several cycles ago*. The `Cyc-In` column shows when those inputs should have arrived. For example, consider the line starting `Tr 7`. The sum value is `57`, reflecting the sum of module inputs arriving at cycle `5` (inputs 45_{16} and 12_{16}).

The line starting `Cr` shows what the correct output should be. (The `Cr` lines are not shown for every cycle.) When the module output is correct the `Cr` line ends with `This sum found.` If the module output is incorrect the line ends with `This sum Not Found.`. Note that the testbench only checks for the correct value at the cycle it was expected. If the correct value appears earlier or later the testbench treats it as wrong. But the trace shows module inputs before and after the correct result is expected, so you can tell if your module is early or late by looking at the trace.

In the sample below the `add_seq` provides the correct sum, but late:

```
    Cyc --a  --b  S          Cyc-In   Sum         Debug Val
Tr    5  45   12  S  Mod Out    3 -->  xx        000045
Tr    6  06   0d  _  Mod Out    4 -->  xx        000006
Tr    7  07   8d  _  Mod Out    5 -->  xx <--   000007
Cr    7 ---  ---     Cor Out    5 -->  57 <-- This sum Not Found.
Tr    8  08   65  _  Mod Out    6 -->  57        000008
Tr    9  09   12  _  Mod Out    7 -->  57        000009
```

The `Cr` line indicates that the testbench expected the `sum` output to be 57 at cycle 7, but the `Tr 7` line shows a `sum` of `xx`, which is wrong. The correct output appears in cycle 8, but that's late and so not counted as correct. Early would not be correct early.

Some testbench output for a correctly solved trace module appears below:

```
Starting tests for pipelined w=16,n=2.
    Cyc ---a  ---b  S          Cyc-In   Sum         Debug Val
Tr    5 0045  0012  S  Mod Out    3 --> d0c5        000045
Tr    6 0006  0e07  _  Mod Out    4 --> 94cb        000006
Tr    7 0007  4cee  _  Mod Out    5 --> 0057 <--   000007
Cr    7 ----  ----     Cor Out    5 --> 0057 <-- This sum found.
Tr    8 0008  61e5  _  Mod Out    6 --> 0e0d        000008
Tr    9 0009  ebc3  _  Mod Out    7 --> 4cf5        000009
```

The trace lines above are similar to the trace lines for a sequential module, but there is an important difference that's not apparent above: the correctness of the pipeline output is checked every cycle, but a `Cr` line is only showed for certain inputs. So in the tally of correct outputs the cycle 7 line increments the number of correct outputs, but not the cycle 5,7,8, and 9 lines, even though those outputs are correct. Similarly, wrong outputs are only tallied for the `Cr` lines.

The output below is for an incorrect `add_pipe` solution:

```
Starting tests for pipelined w=8,n=2.
    Cyc --a  --b  S          Cyc-In   Sum         Debug Val
Tr   95  a5   a6  S  Mod Out   93 -->  9c        0000a5
Tr   96  60   83  _  Mod Out   94 -->  56 Err    000060
Tr   97  61   fa  _  Mod Out   95 -->  5b Err   000061
Cr   97 ---  ---     Cor Out   95 -->  4b <-- This sum Not Found.
Tr   98  62   a1  _  Mod Out   96 -->  e3        000062
Tr   99  63   e9  _  Mod Out   97 -->  5b        000063
```

Here the output a cycle 97 is wrong, and that counts as an error. The output at cycle 96 is also wrong but that error is not tallied.

In an unmodified assignment file both `add_seq` and `add_pipe` compute the correct answer, but too soon.

At the end of the testbench output is a tally of errors. The sample below is for an unmodified assignment, in which both `add_seq` and `add_pipe` compute the correct answer, but too soon:

```
xmsim: *W,RNQUIE: Simulation is complete.
xcelium> exit
```

```
Total sequential w= 8, n= 2: Errors: 995 sum.
Total sequential w=16, n= 2: Errors: 1000 sum.
Total sequential w=16, n= 4: Errors: 1000 sum.
Total sequential w=16, n= 3: Errors: 1000 sum.
Total pipelined w= 8, n= 2: Errors: 996 sum.
Total pipelined w=16, n= 2: Errors: 1000 sum.
Total pipelined w=16, n= 4: Errors: 1000 sum.
Total pipelined w=16, n= 3: Errors: 1000 sum.
Total sequential: Errors: 3995 sum.
Total pipelined: Errors: 3996 sum.
Grand Total Errors:  7991 sum.
TOOL: xrun(64) 26.03-s005: Exiting on Nov 11, 2026 at 17:39:56 CST  (total: 00:00:00)
```

Helpful References

Two items already presented in class and one to be presented soon use pipelines. Files with these items are in the assignment package. File `multi-step.v` has the pipelined multi-step module. This is easy to understand so look at it first. File `add-accum-sol.v` is the add-accumulate module done in class. Look for `add_pipe` in that file which though pipelined does not use BFA modules and so is not a solution to Problem 2. Finally file `mult-pipe.v` has several pipelined multiplier implementations.

Synthesis Script

Modules for this assignment should be synthesized using the command `genus -f syn.tcl`. The script synthesizes modules `add_seq` (Problem 1) and `add_pipe` (Problem 2), as well as single-cycle comparison modules `add_comb`, `add_proc`, and `add_cla`.

All modules are synthesized at $w = 8$ and $w = 16$. Modules `add_seq` and `add_pipe` are synthesized at $n = 2$ and $n = 4$ steps/stages.

All modules are synthesized with an easy 100 ns delay target, so the synthesis program can focus on reducing cost. Some modules are also synthesized with a 0.1 ns delay target, prioritizing speed.

Column `Area` shows area, which for us is cost. Column `Delay Actual` is delay of the synthesized module, which is the longest path from input to output. Note that `Delay` refers to the clock period. The `Delay` can be used to compute latency and throughput of the modules, the computation for `add_pipe` is different than for `add_seq`, etc.

Column `Delay Target` is the maximum acceptable delay we chose. The synthesis program will reduce delay until this delay target is reached, then it switches to optimizing to reduce cost. Column `Synth Time` is how long the synthesis program itself took to run. That matters for impatient engineers and whoever is paying for compute time, but it does not in any way describe the synthesized hardware.

The output below is from a solved assignment.

Module Name	Area	Delay Actual	Delay Target	Synth Time
add_pipe_w8_n_stages2	15830	1.97	100.0 ns	2 s
add_seq_w8_n_steps2	22678	2.21	100.0 ns	3 s
add_pipe_w8_n_stages4	27858	1.11	100.0 ns	3 s
add_seq_w8_n_steps4	25196	1.68	100.0 ns	3 s
add_pipe_w16_n_stages2	31437	3.70	100.0 ns	5 s

add_seq_w16_n_steps2	41997	4.19	100.0 ns	5 s
add_pipe_w16_n_stages4	54566	1.97	100.0 ns	4 s
add_seq_w16_n_steps4	44908	2.44	100.0 ns	4 s
add_pipe_w8_n_stages2	25321	0.90	0.1 ns	9 s
add_seq_w8_n_steps2	38433	1.02	0.1 ns	22 s
add_pipe_w8_n_stages4	34622	0.72	0.1 ns	9 s
add_seq_w8_n_steps4	34122	0.91	0.1 ns	11 s
add_pipe_w16_n_stages2	62034	1.12	0.1 ns	23 s
add_seq_w16_n_steps2	82496	1.23	0.1 ns	28 s
add_pipe_w16_n_stages4	84611	0.92	0.1 ns	20 s
add_seq_w16_n_steps4	64151	1.04	0.1 ns	22 s
add_comb_w8	9817	3.26	100.0 ns	6 s
add_proc_w8	9817	3.26	100.0 ns	1 s
add_cla_w8	9817	3.26	100.0 ns	1 s
add_comb_w16	19872	6.72	100.0 ns	7 s
add_proc_w16	19872	6.72	100.0 ns	3 s
add_cla_w16	19872	6.72	100.0 ns	2 s
add_comb_w8	34471	0.65	0.1 ns	15 s
add_proc_w8	34471	0.65	0.1 ns	14 s
add_cla_w8	31565	0.68	0.1 ns	17 s
add_comb_w16	60675	1.01	0.1 ns	26 s
add_proc_w16	48209	1.00	0.1 ns	18 s
add_cla_w16	55115	1.02	0.1 ns	20 s

Problem 1: *Note: Problem 2 might be easier than Problem 1, so feel free to solve that first.* Module `add_seq` has two w-bit inputs, `a` and `b`, and a w-bit output `sumr`, where w is the value of parameter `w` (the fonts are different). There is also inputs `start` and `clk`, both one bit and parameter `n_steps`. When completed the module will compute the sum of the values at `a` and `b` over n cycles, in each cycle $\lceil \frac{w}{n} \rceil$ of the sum are to be computed, using the same hardware each cycle. A sum computation should start in a cycle in which `start=1`. Input `start` will be 1 for just one cycle and then go to zero at least until the computation finishes. See the testbench output for samples of this timing.

Inputs `a`, `b`, and `start` are available at the beginning of the clock cycle and don't change during the clock cycle. Output `sumr` is expected to be available at the beginning of a clock cycle and to not change during the cycle.

- ☐ Compute the sum over n cycles.
- ☐ There should be hardware only for operating on $\lceil \frac{w}{n} \rceil$ bits, not w bits nor multiple copies of $\lceil \frac{w}{n} \rceil$-bit adders.
- ☐ The module can use procedural code. Don't instantiate other modules.
- ☐ Try to save cost by sharing storage for `a`, `b`, and the sum, so less than $3w$ bits of register storage are needed.
- ☐ The module must be synthesizable.
- ☐ As always, pay attention to cost and performance.
- ☐ Code clearly and follow conventions used in class.

Follow other guidelines and restrictions in the checklist in the file in the Problem 1 area.

There's another problem on the next page!

Problem 2: Module `add_pipe` has two w-bit inputs, `a` and `b`, and a w-bit output `sumr`, where w is the value of parameter `w` (the fonts are different). There is also a `clk` input and parameter `n_stages`. When completed the module will compute the sum of the values at `a` and `b` in pipelined fashion over n cycles, in each cycle $\lceil \frac{w}{n} \rceil$ of the sum are to be computed, using different hardware (of course) in each stage.

Compute the sum by instantiating `bfa` modules. In the unmodified assignment `bfa` modules are used to compute the entire w-bit sum in one cycle.

As with `add_seq`, inputs `a` and `b` are available at the beginning of a cycle and hold steady. Output `sumr` should be available at the beginning of a cycle and should not change.

- ☐ Compute the sum in an n-stage pipeline.
- ☐ Each stage should compute $\lceil \frac{w}{n} \rceil$ bits.
- ☐ The addition must be performed by `bfa` instantiations.
- ☐ Try to save cost by sharing storage for `a`, `b`, and the sum, so less than $3wn$ bits of register storage are needed.
- ☐ The module must be synthesizable.
- ☐ As always, pay attention to cost and performance.
- ☐ Code clearly and follow conventions used in class.