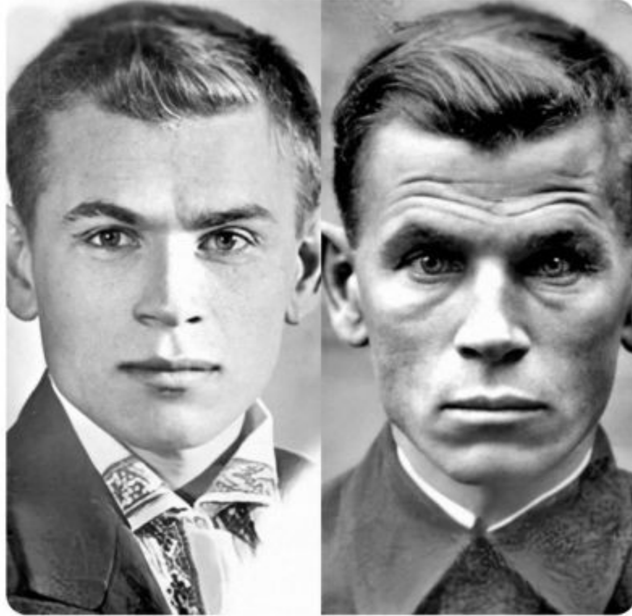
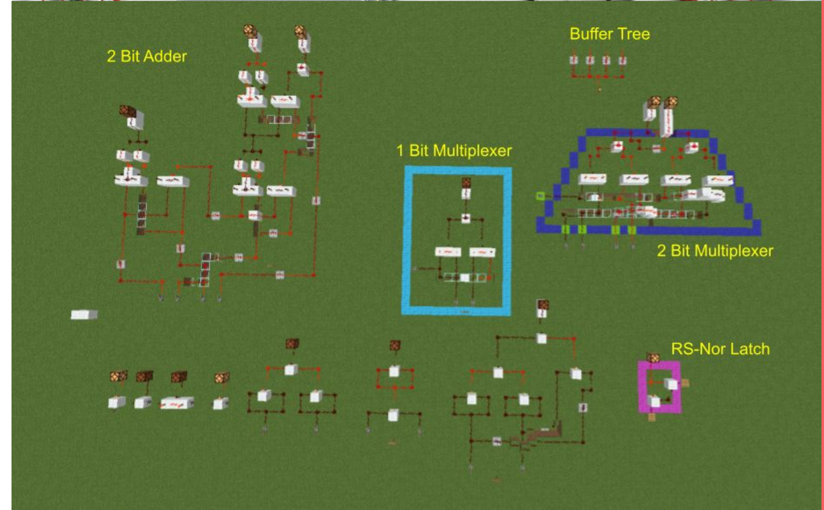
# Welcome bbaacckk to CS429H!

Week 8

# Ed meme recap:

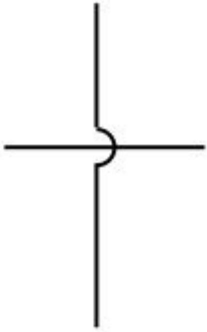

Soldier's face after p7
Before p7
After p7

decode
fetch
fetch
decode
decode
fetch
execute
execute
fetch

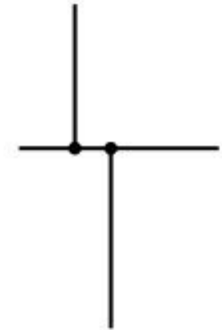

2 Bit Adder
Buffer Tree
1 Bit Multiplexer
2 Bit Multiplexer
RS-Nor Latch

Questions on lecture content?
(Please no cats today)

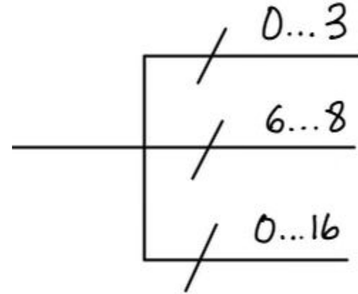
# Drawing Wires For Quiz How-To



One wire crossing over a different wire.



All of these wires have the same value.




Getting the lower order 4 bits from a wire, etc.

Quiz everyone say CHEESE!

# Poll

```
Channel *feedback = go(quiz);
Value v = receive(feedback);
```

How was the quiz?

A. easy
B. mostly fine
C. mostly fine, but not enough time
D. too hard, but finished mostly in time
E. too hard and not enough time
F. too hard regardless of time

# Stress

- 429H is not an easy class
  - Lots of new materials
  - Unfamiliar programming environments
  - Fast, often relentless pace
- Struggling in this course is normal
  - There will be times you won't know the answer of the solution
  - This is expected—we want we everyone to succeed, but the only way we can help is if you ask for it
- If you find yourself overly overwhelmed or spending more time on this class than you think you should be, please reach out to Dr. Gheith or the TAs
  - We can help out as far as the class goes
  - We can provide other resources where we are not able to help

Mental health resource available at UT

P7

# Poll

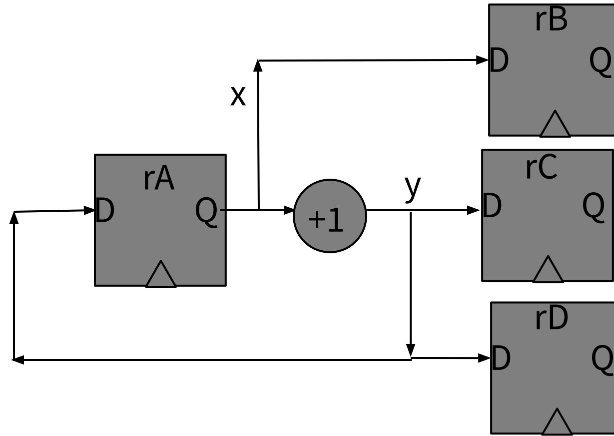
How's your status on P7?

A. What's P7?
B. I've heard of it
C. I've cloned the starter code and/or looked through it
D. I've started planning/writing code
E. I'm mostly done but might still have bugs
F. P7 any% speedrun

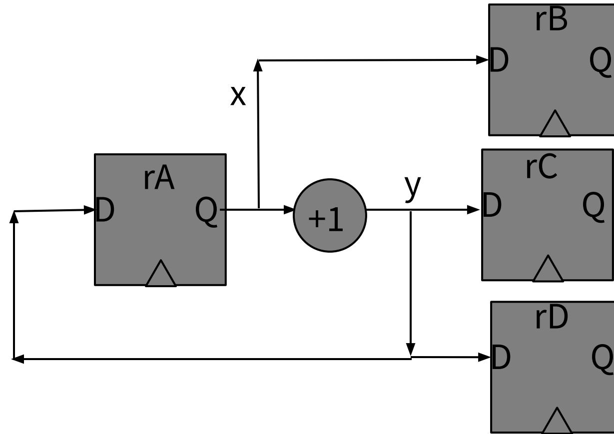
# Verilog

- Verilog is a Hardware Description Language (HDL), NOT a programming language
  - There are no if statements (well, there are, but they don’t always work like those in a PL)
  - Every line of “code” is run in parallel
  - HDL is split up into two “sections”: continual and procedural assignment

# But First... What is a wire?



# But First... What is a wire?



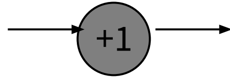

How many “inputs” does x have?
How many “outputs” does x have?

# But First... What is a wire?

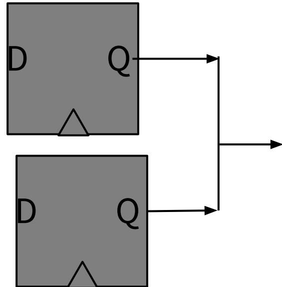
**There can only be 1 driver of a wire**

But anybody can read that wire's value!

A wire with no driver is undefined




A wire with 2 drivers is undefined



# How do we represent this circuit in Verilog?

- Verilog is split into continuous and procedural assignment blocks.
- An “always” or “initial” starts a procedural block; otherwise, you’re in a continuous block.

# Continuous Block

- where you declare wires and registers
- where you assign wires
- assignments constantly updated

```
wire x;

wire [1:0] y;

reg [1:0] rA;

assign x = rA;
```

# Procedural Block

- where you assign registers
- updated only on clock tick

```
always @(posedge clk) begin

  rB <= x;

end

// begin is a {, end is a }
```

# Style Guide

## Continuous Block

- Only use =, <= is a syntax error
- Always declare wires/regs as [x:0], not [0:x]

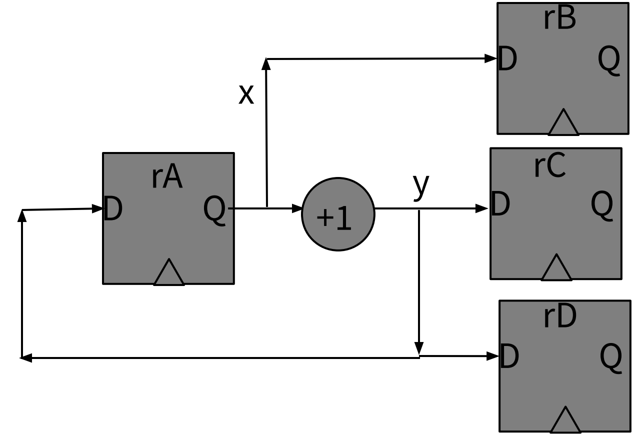
## Procedural Block

- Only use <=, NEVER USE =
- Only use always @(posedge clk), don't use negedge or other things in the @()
- Every line is run at the same time, so you can swap values like this:

```
always @(posedge clk) begin
    rA <= rB;
    rB <= rA;
end
```

# Let's Translate This Circuit!

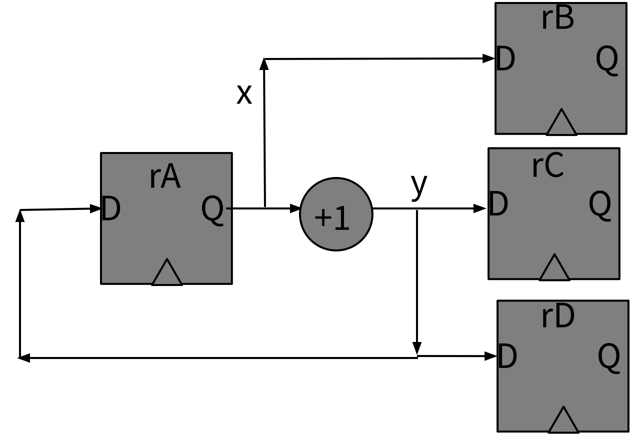
```
reg [31:0] rA;

reg [31:0] rB;

reg [31:0] rC;

reg [31:0] rD;

wire [31:0] x;

wire [31:0] y;
```



# Let's Translate This Circuit!

```
reg [31:0] rA;
reg [31:0] rB;
reg [31:0] rC;
reg [31:0] rD;
wire [31:0] x;
wire [31:0] y;
```

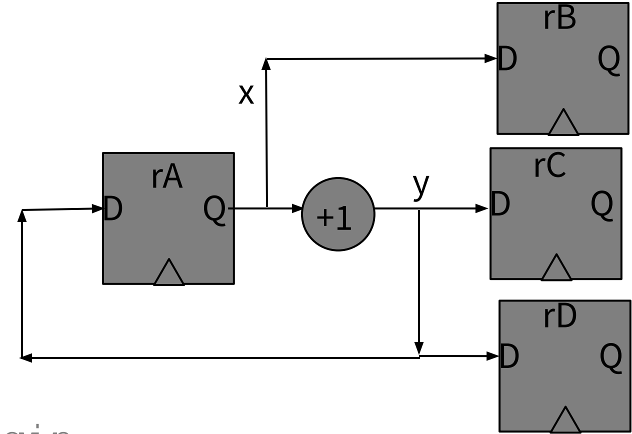
```
assign x = rA;
assign y = x + 1;
```



# Let’s Translate This Circuit!

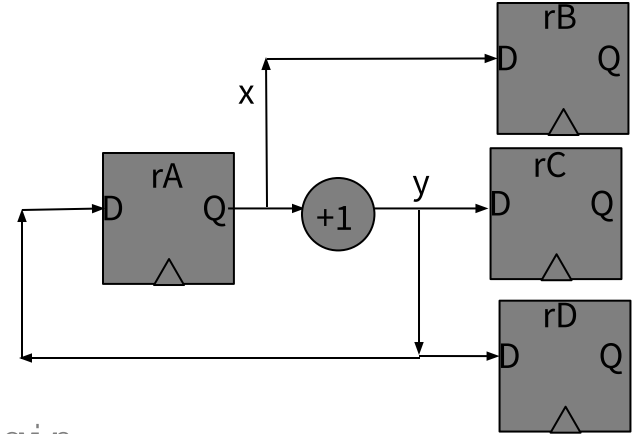
```
reg [31:0] rA;
reg [31:0] rB;
reg [31:0] rC;
reg [31:0] rD;
wire [31:0] x;
wire [31:0] y;
```

```
assign x = rA;
assign y = x + 1;

always @(posedge clk) begin
    rA <= y;
    rB <= x;
    rC <= y;
    rD <= y;
end
```



# We don't actually need wires...

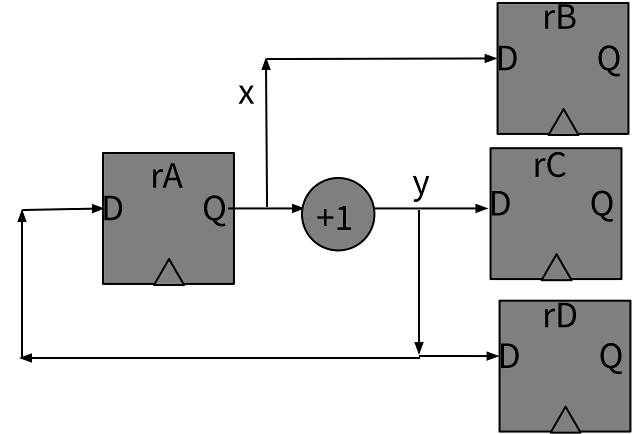
```
reg [31:0] rA;
reg [31:0] rB;
reg [31:0] rC;
reg [31:0] rD;
```




```
always @(posedge clk) begin
    rA <= rA + 1;
    rB <= rA;
    rC <= rA + 1;
    rD <= rA + 1;
end
```

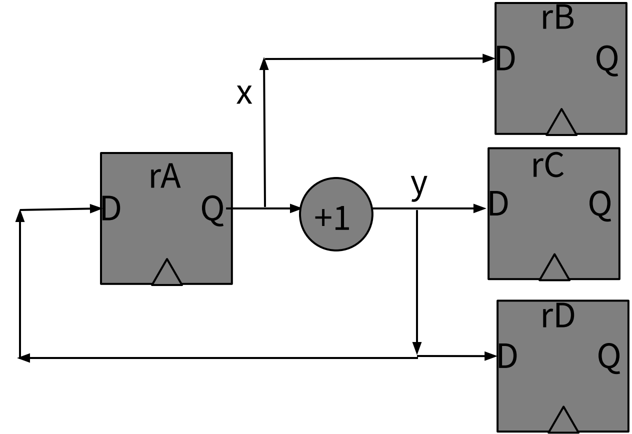
# Initial Values?

```
reg [31:0] rA;

reg [31:0] rB;

reg [31:0] rC;

reg [31:0] rD;

wire [31:0] x;

wire [31:0] y;
```



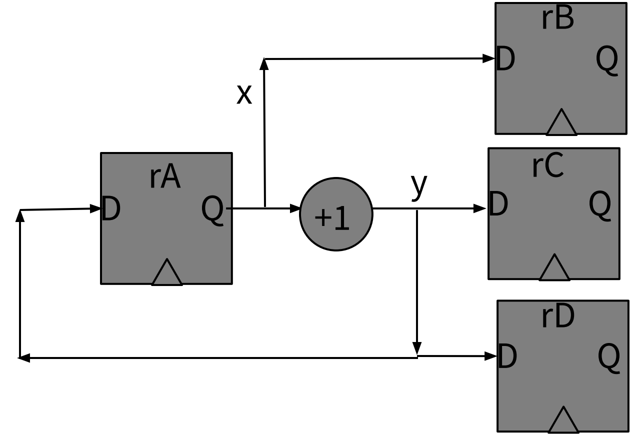
# Initial Values?

```
reg [31:0] rA = 0; // assigns 0 to rA on startup
reg [31:0] rB = 0;
reg [31:0] rC = 0;
reg [31:0] rD = 0;
wire [31:0] x = 0; // this is bad
wire [31:0] y = 0; // and equivalent to: assign y = 0
```



# Initial Values?

```
reg [31:0] rA = 0; // assigns 0 to rA on startup
reg [31:0] rB = 0;
reg [31:0] rC = 0;
reg [31:0] rD = 0;
wire [31:0] x = rA; // this is ok instead of writing a separate assign
wire [31:0] y = x + 1;
```



# What are all these X's?

```
wire [2:0] hi = {0,1}; // hi[0] = 1, hi[1] = 0,  but what is hi[2]?
```

```
initial $display("%d", hi[2]); // displays x
```

x is "undefined" and it propagates throughout the pipeline, but it doesn't mean error!

It's OK to have wires/regs undefined, as long as you don't use them to make decisions

# No IF Statements?

```
reg a, b, c;
wire x;

if (a == 1) // error: no IF statements allowed in continuous blocks
	assign x = b; // because an assignment is a permanent description of what a wire is
else
	assign x = c; // how do we fix this bug?
```

# No IF Statements?

```
reg a, b, c;
wire x;

assign x = (a == 1) ? b : c;
```

# No IF Statements?

```
reg a, b, c;
wire x;

assign x = c1 ? a :
           c2 ? b :
          c3 ? c :
          d;
```

# You can use if statements in procedural blocks

```
always @(posedge clk) begin
    if (!stall) begin
        f1_pc <= f0_pc;
    end
end
// question: what if there is a stall? what does f1_pc get set to?
```

# Or ternaries work there too

```
always @(posedge clk) begin
        f1_pc <= stall ? f1_pc : f0_pc;
end
```

# Verilog Literals

```
wire x = 1; // warning: 1 is a 32 bit value being assigned to a 1 bit wire


// solution:
wire y = 1'b1;
// general form: width'{b,d,h}number
// eg: a 64-bit 0x20 is 64'h20 or 64'd32
//     a 2-bit 3 is 2'b11 or 2'd3 or 2'h3
//     a 16-bit undefined value is 16'bX
```

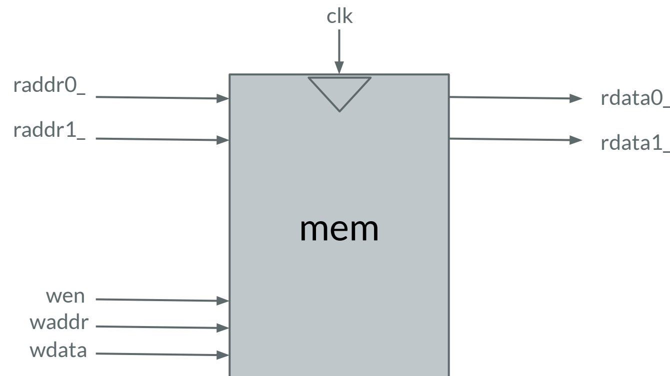
# Verilog Modules

```
module mem(input clk,
    input [15:1]raddr0_, output [15:0]rdata0_,
    input [15:1]raddr1_, output [15:0]rdata1_,
    input wen, input [15:1]waddr, input [15:0]wdata);
```

input = caller should drive wire
output = module will drive wire

It’s just a box!

The starter code instantiates it for you—you need to provide wires that will connect to raddr0_, rdata0_, etc.



# .hex files

Broken test case!

```
@0
86
80
86
50
86
c0
86
c0
86
f0
80
a0
ff
ff
```

# .hex files

Bad test case!

```
@0
8680
8650
86c0
86c0
86f0
80a0
ffff
```

# .hex files

Better test case. USE COMMENTS!

```
@0
8680 // movl r0, ‘h’
8650 // movl r0, ‘e’
86c0 // movl r0, ‘l’
86c0 // movl r0, ‘l’
86f0 // movl r0, ‘o’
80a0 // movl r0, ‘\n’
ffff // invalid
```

Part of your test grade case reflects how helpful it is to others. Hex digits are not very useful to others

# .hex files

Better test case. USE COMMENTS!

```
@0
8680 // movl r0, 'h'
8650 // movl r0, 'e'
86c0 // movl r0, 'l'
86c0 // movl r0, 'l'
86f0 // movl r0, 'o'
80a0 // movl r0, '\n'
ffff // invalid
```

Each line is a 16-bit entry in memory
a.k.a. 4 hex digits
Each instruction is also 16 bits
0x8680 = 0b1000011010000000
movl 104 = 'h' r0

Part of your test grade case reflects how helpful it is to others. Hex digits are not very useful to others

# README Correction

<test name>.raw => the raw output from running the test

<test name>.out => lines from *.raw that start with #

<test name>.cycles => number of cycles needed to run the test

<test name>.vcd => vcd file after running test

<test name>.ok => expected output

<test name>.hex => the test program

# README Correction

<test name>.raw => the raw output from running the test

<test name>.out => ~~lines from *.raw that start with #~~ lines from *.raw which are not "VCD info: dumpfile cpu.vcd opened for output"

<test name>.cycles => number of cycles needed to run the test

<test name>.vcd => vcd file after running test

<test name>.ok => expected output

<test name>.hex => the test program

# Tips

- Follow the style guide
- Follow the style guide
- Don’t $display debug
- Add -Wall to the iverilog compile options
- Don’t touch verilog functions unless you know what you’re doing
- Ignore hazards, flushing, stalling etc. to start, and slowly add those in
- Use good wire & reg naming conventions (know which things are inputs to your stage and what are outputs)
- Clearly mark and separate each stage with some consistent convention
- You can use multiple procedural blocks
- Your test case (a .hex file) MUST HAVE COMMENTS
    - It is fine to share assemblers/disassemblers, but your test case should be pretty understandable without having to use a disassembler

# Set Up X Forwarding

X Forwarding is how we can run software remotely on a lab machine while displaying the graphics to our local machine! Instructions for setting up X Forwarding posted on Ed!

Once you've set it up, you SSH into a lab machine like this:

```
ssh -X <csid>@<hostname>.cs.utexas.edu
```

And then you can run fun programs!

```
xeyes
```

```
~jocelyn/public/oneko
```

# Set Up X Forwarding

X Forwarding is how we can run software remotely on a lab machine while displaying the graphics to our local machine! Instructions for setting up X Forwarding posted on Ed!

Once you've set it up, you SSH into a lab machine like this:

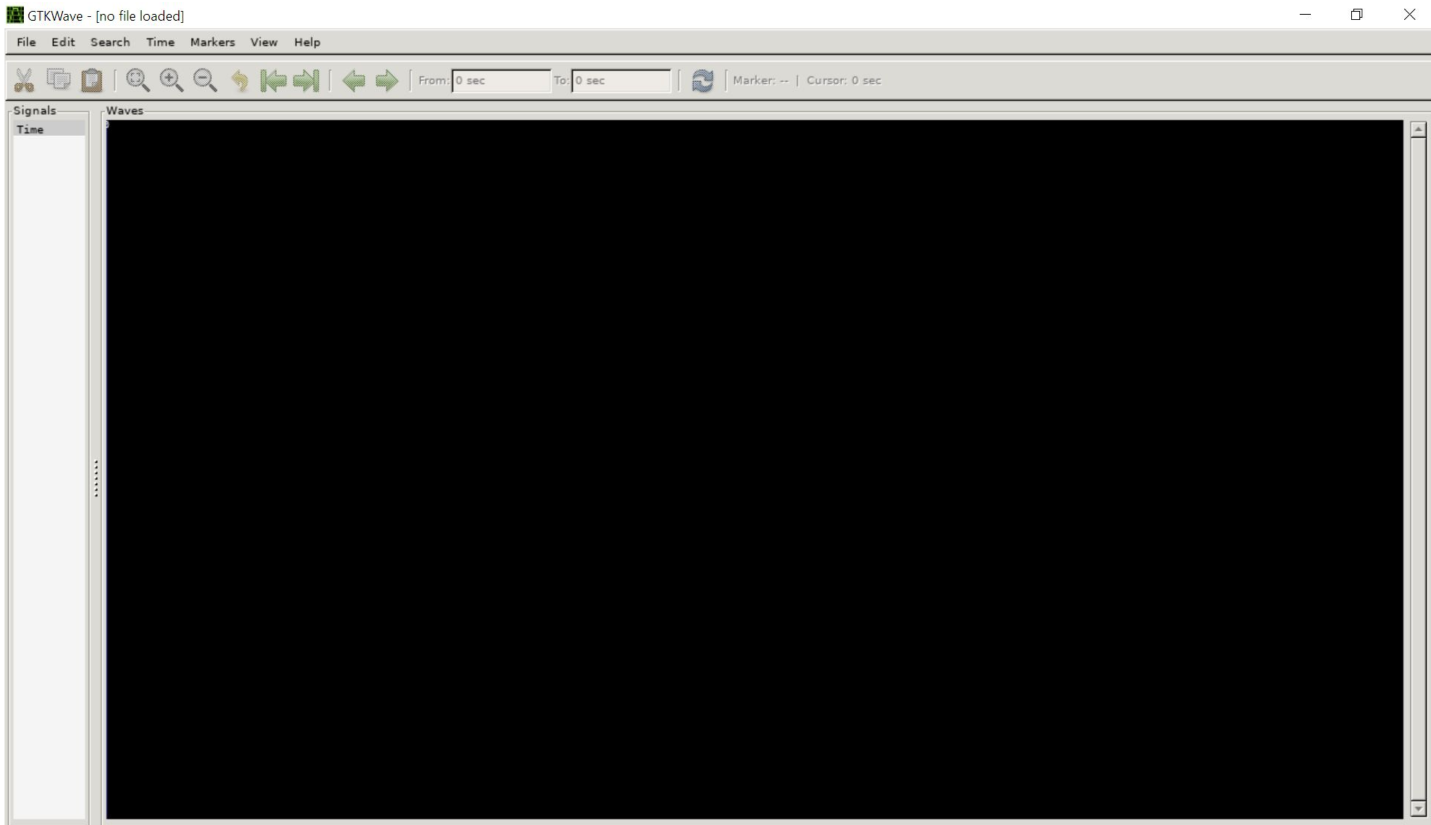
```
ssh -X <csid>@<hostname>.cs.utexas.edu
```

And then you can run fun programs!

```
xeyes
```

```
~jocelyn/public/oneko
```

```
gtkwave
```




gtkwave

[Modified] GTKWave - /u/ceden/Documents/CS429H/p7/CS429H-p7/gpu.vcd

File Edit Search Time Markers View Help

From: 0 sec To: 3515 ns Marker: 916 ns | Cursor: 2620 ps

SST

- $DUP0
- $DUP1
- $DUP2
- framebuf
- genblk2[1]
- genblk2[2]
- genblk2[3]
- genblk2[4]
- genblk2[5]
- genblk2[6]
- genblk2[7]
- genblk2[8]
- genblk2[9]
- genblk2[10]
- genblk2[11]
- genblk2[12]
- genblk2[13]
- genblk2[14]
- genblk2[15]
- genblk2[16]
- genblk2[17]

| Type | Signals |
| --- | --- |
| integer | i |

Filter:

Signals

```
Time
ready =1
clk =1
count_status[1:0] =11
object_count[31:0] =2
DEBUGfetchaddr[31:4] =0000011
DEBUGobjdata[127:0] =0000000000000E3C
id_to_addr[31:0] =0000000000000000
objaddr[31:4] =0000011
load_status[2:0] =000
processor_id[4:0] =00
object_id[31:0] =00000010
obj[127:0] =0000000000000E3C
transaddr[31:6] =xxxxxxx
current_vertex[31:5] =0000E44
first_vertex[31:5] =0000000
DEBUGtransdata[511:0] =3F8000000000000
vertex_count[31:0] =3644
in_bounds =0
wen =0
waddr[31:5] =0000E3C
inVertex[255:0] =C04000003FE66666
inVertex[255:0] =C03F76603FE66666
inVertex[255:0] =C03F76603FE66666
inVertex[255:0] =C03F53263FD544CC
DEBUGwdata0[255:0] =C04000003FE66665
DEBUGwdata1[255:0] =C03F76603FE66665
DEBUGwdata2[255:0] =C03F76603FE66665
DEBUGwdata3[255:0] =C03F53263FD544CE
current_vertex[31:5] =0001C80
first_vertex[31:5] =0000E3C
DEBUGtransdata[511:0] =BF800000A50D3132
vertex_count[31:0] =3644
in_bounds =0
vaddr[31:5] =0001C80
wen =0
waddr[31:5] =0001C78
inVertex[255:0] =xxxxxxxxxxxxxxxx
inVertex[255:0] =xxxxxxxxxxxxxxxx
inVertex[255:0] =xxxxxxxxxxxxxxxx
inVertex[255:0] =xxxxxxxxxxxxxxxx
```

Waves

```
gtkwave final_project.vcd
```

# More Advice

# some advice for debugging

- check that there are no blocking statements in your always blocks
- all always blocks should be **@ (posedge clk)**
  - this excludes the clock module
  - you **will get a 0** if you do not follow this
- if you have an if statement in an always block, are you updating the same set of registers in both the if and the else? If not, is it intentional?
- are you updating the same register in multiple locations?
- the memory and register modules **cannot** stall
- **a correct implementation that flushes on every hazard will get more correctness points than an incorrect implementation that attempts stalling**

# Writing Verilog

- Write a little bit of good code - debugging is hard so try to get it right on the first try
- Have a clear naming convention - is execute_pc the output of or input to execute?
- Reuse wires as much as reasonably possible - don't have immediate wires for each instruction variant
- Clearly separate stages - don't have intermixed code
- Recommended vscode extension

# Stage contracts

- The hard part of pipelining is the communication between stages - not the stages themselves
- Without hazards, it's quite simple - input comes in, one cycle later output is ready
- But with hazards, things are no longer clear
- When the flush wire goes high for a cycle, does that immediately invalidate the output or does it take a cycle? When fetch receives a new PC, how many cycles till the instruction is ready? When a module has to stall, exactly which stall wires does it set and for how long?
- Treat stages as independent components - they only connect through wires defined in your contract

# Debugging

- Don’t use print statements, you’ll get too much output
- DO NOT USE THE VSCODE WAVETRACE EXTENSION IT IS BUGGED (as of last year, and has not been updated since then so is still broken)
- GTKWave is vastly superior and is the only reasonable way to debug verilog
- X-forwarding works really well, just ssh with -X and launch gtkwave (use WSL on Windows 11)
- Add all the important wires for each stage, group them (press G), and optionally color code them
- Then make sure to save your layout, gtkwave will not remind you to save
- Searching for a value - select the wire you want to search, then search>pattern search 1>dropdown to “string”>plug in value>find next
- Right click on a signal, change the data format

# Verilog Resources

- https://github.com/steveicarus/iverilog
- A Verilog Primer
  - Just note to use @(posedge clk) not @(*) in your code

# Questions?

```
                          oooo$$$$$$$$$$$$oooo
                      oo$$$$$$$$$$$$$$$$$$$$$$$$o
                   oo$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$o         o$   $$ o$
   o $ oo        o$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$o       $$ $$ $$o$
oo $ $ "$      o$$$$$$$$$    $$$$$$$$$$$$$    $$$$$$$$$o       $$$o$$o$
"$$$$$$o$     o$$$$$$$$$      $$$$$$$$$$$      $$$$$$$$$$o    $$$$$$$$
  $$$$$$$    $$$$$$$$$$$      $$$$$$$$$$$      $$$$$$$$$$$$$$$$$$$$$$$
  $$$$$$$$$$$$$$$$$$$$$$$    $$$$$$$$$$$$$    $$$$$$$$$$$$$$  """$$$
   "$$$""""$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$     "$$$
    $$$   o$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$     "$$$o
   o$$"   $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$       $$$o
   $$$    $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$" "$$$$$$ooooo$$$$o
  o$$$oooo$$$$$  $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$   o$$$$$$$$$$$$$$$$$
  $$$$$$$$"$$$$   $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$     $$$$""""""""
 """"       $$$$    "$$$$$$$$$$$$$$$$$$$$$$$$$$$$"      o$$$
            "$$$o     """$$$$$$$$$$$$$$$$$$"$$"         $$$
              $$$o          "$$""$$$$$$""""           o$$$
               $$$$o                                o$$$"
                "$$$$o      o$$$$$$o"$$$$o        o$$$$
                  "$$$$$oo     ""$$$$o$$$$$o   o$$$$""
                     ""$$$$$oooo  "$$$o$$$$$$$$$"""
                        ""$$$$$$$oo $$$$$$$$$$
                                """"$$$$$$$$$$$
                                    $$$$$$$$$$$$
                                     $$$$$$$$$$$"
                                      "$$$""""
```