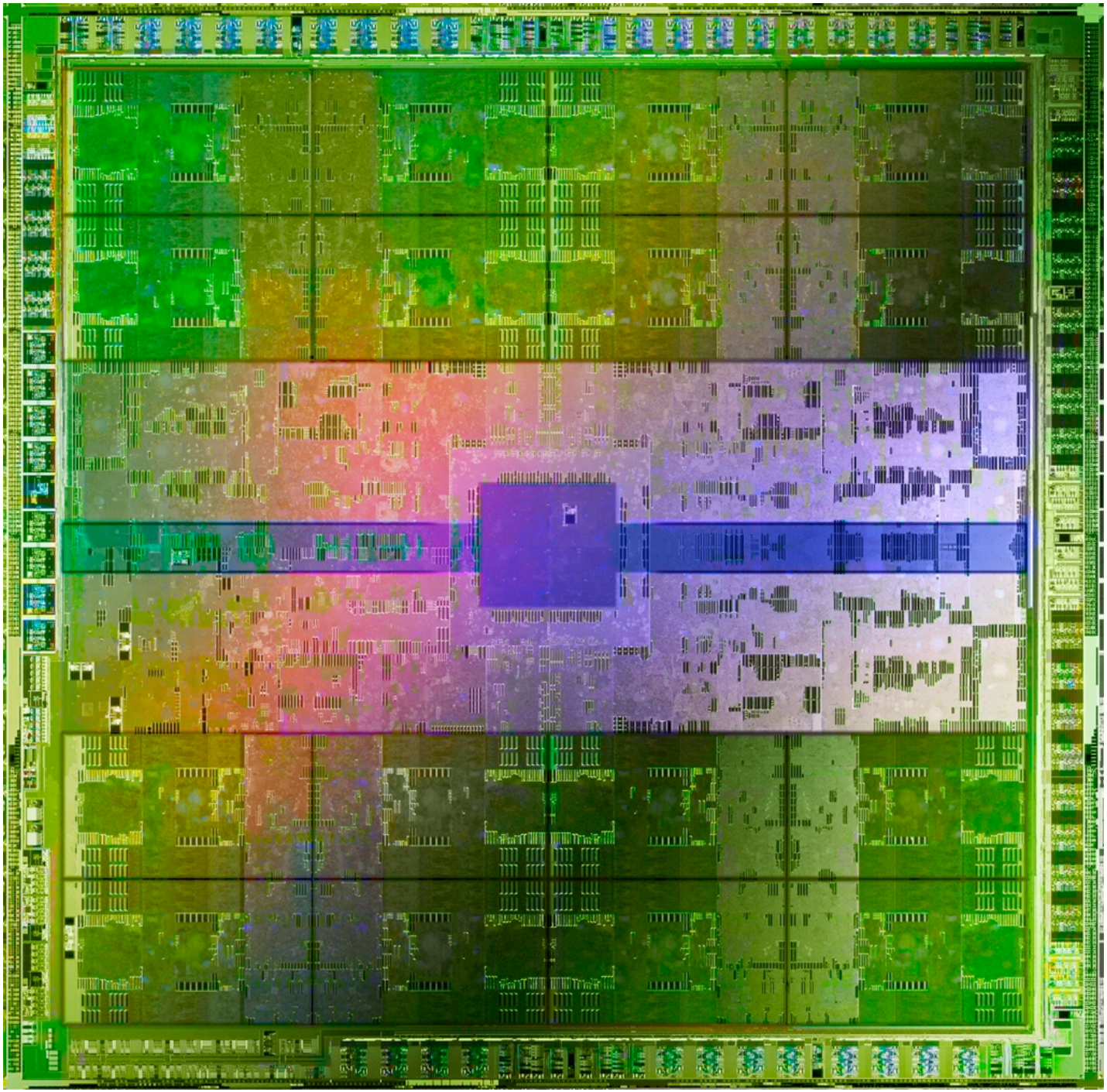
Image Courtesy: NVIDIA



CS250
Section 2

9/9/10

Yunsup Lee

Any questions on Lab 1 so far?

Paper Discussions

- We have **two** paper discussions this semester
  - September 15th: Low Power ARM, Power7 Server Processor
  - September 22nd: Scale VT Processor, Energy-Performance Exploration
- You need to turn in summaries for each paper (hard copy) before class (the day we are doing discussions)
  - Start with a brief, (1 paragraph) **summary** of the main ideas of the paper. Next address the following.
  - What are the **strengths** of the technical ideas presented (1-3 sentences)?
  - What are the **weakness** of the technical ideas presented (1-3 sentences)?
  - Rank the **writing quality** (1-10):
  - Does (or did) this paper open **new research areas**? If so, what (1-3 sentences)?
  - What specific comments would you make to the authors to **improve** the quality of their work or their presentation?

Changes to Labs and Projects

- Original plan was to use Maven, but I've decided we're going to use RISC-V
- Reasons,
  - Research is shifting to RISC-V, and your work will be more relevant
  - We're going to upgrade to RISC-V next year, and making Maven labs and tutorials this year is just more work that we know that we're going to throw away
- This means,
  - New compiler, new ISA simulator, new test cases, basically new infrastructure
  - Major revisions to the labs and tutorials (This is the reason why labs and tutorials are getting a little bit delayed)
  - New baseline RTL processor!

Upcoming dates

- 9/13/10 (10:30am) - Lab 1 due
  - Submit using SVN (src, build, writeup)
- 9/13/10 - Lab 2 & 3 out (not sure about lab 3 at this moment, but will do my best)
  - Write and Synthesize a Two-Stage/Three-Stage RISCV-v2 Processor
  - ASIC Implementation of a Three-Stage RISCV-v2 Processor with On-Chip Caches
- Did you know about tutorials?
  - Bits and Pieces of CS250's toolflow
  - Build, Run, and Write RISC-V Programs
  - Simulating Verilog RTL using Synopsys VCS
  - RTL-to-Gates Synthesis using Synopsys Design Compiler

Some other stuff ...

- SVN Server is up and running
  - https://isvn.eecs.berkeley.edu/cs250/<login>
  - Guarded with your instructional login/password
- Your source tree should look like:
  - lab1
    - trunk
      - src: commit code
      - build.manual
      - build: commit python script
        - dc-syn/icc-par/pt-pwr: commit most recent build
        - vcs-sim-*
    - branches
    - tags

Some other stuff...

- Directory structure (a proposal)
  - if you already committed to the svn server, don't do this
  - put lab harness on your home directory and /tmp, then make a symbolic link to the src directory on your home directory, and work in the /tmp directory
- Now it looks like:
  - /tmp/yunsup/vc/lab1/trunk
    - src -> /home/aa/grad/yunsup/vc/lab1/trunk/src
    - build
    - build.manual
  - ~/vc/lab1/trunk
    - src/build/build.manual
- You can commit the source code to the svn server from the lab harness on your home directory

Some other stuff ...

- Please exit the tools after using
  - Don’t keep your tools occupying a seat
  - It’s a shared resource, and you should be fair
- Labs and tutorials have version numbers
  - I’m trying my best not to change the version whenever it’s in transit
- Writeups should be in text or PDF
- Section on September 23rd moved to September 21st.
  - Same room (310 Soda)
  - Same time (5-6pm)
  - Extended office hours (3-5pm) this day (September 21st)

Scripting

- Tools generate a lot of data, and you shouldn’t go through the files by hand
- Scripting is essential
- We’re going to use Python for this class

Python list

```
#! /usr/bin/python

alist = []
alist.append(1)
alist.append(10)
alist.append(3)

print alist

alist[1] = 9

for aelement in alist:
  print aelement

def mul2(x):
  return 2*x

alist = map(mul2, alist)

for aelement in alist[:2]:
  print aelement
```

Python read file

```
#! /usr/bin/python

import sys
import string

if len(sys.argv) < 2:
  print "usage: %s <file>" % sys.argv[0]
  sys.exit(1)

f = open(file)
lines = map(string.strip, f.readlines())
f.close()

for line in lines:
  print line
```

Python line split

```
#! /usr/bin/python

import sys
import string

if len(sys.argv) < 2:
  print "usage: %s <file>" % sys.argv[0]
  sys.exit(1)

f = open(file)
lines = map(string.strip, f.readlines())
f.close()

for line in lines:
  strs = line.split()
  for str in strs:
    print str
```

Python group activity

- Let's form groups of two and write python scripts which gets the data we need out of the report files
- You can find data files in ~cs250/examples/python
- The data you need to extract is:
  - Critical Path Length
  - Post Synthesis Area
  - Post Synthesis Power

Verilog Guide Lines (1)

- Sequential Block
  - always @(posedge clk)
    - Only use non-blocking assignments (<=)
- Combinational Block
  - always @(*)
    - always have a default assignment at the top
    - Only use block assignments (=)
  - continuous assignments
    - try to use continuous assignments rather than always @(*) blocks
- Do not mix blocking and non-blocking assignments in the same always block
- Do not make assignments to the same variable from more than one always block

What's wrong?

```
always @(posedge clk)
begin
  state_reg = state_next;
  operands_reg = operands_next;
end

always @(*)
begin
  state_next <= state_reg;

  if (operands_val)
  begin
    state_next = 2'b01;
    signal_rdy = 1'b0;
  end
end

always @(*)
begin
  if (signal_val)
    signal_rdy = 1'b1;
end
```

Correct Way

```
always @(posedge clk)
begin
  state_reg <= state_next;
  operands_reg <= operands_next;
end

always @(*)
begin
  state_next = state_reg;
  signal_rdy = 1'b1;

  if (operands_val)
  begin
    state_next = 2'b01;
    signal_rdy = 1'b0;
  end
end

assign state_next
  = operands_val ? 2'b01
  : state_reg;

assign signal_rdy = ~operands_val;
```

Verilog Guide Lines (2)

- Always reset your registers, except registers in the datapath, which are normally very wide

```
always @(posedge clk)
begin
  state_reg <= state_next;
  operands_reg <= operands_next;
end

always @(posedge clk)
begin
  if (reset)
  begin
    state_reg <= 2'd0;
    operands_reg <= 32'd0;
  end
  else
  begin
    state_reg <= state_next;
    operands_reg <= operands_next;
  end
end
```

Verilog Guide Lines (3)

- Don’t have mux statements inside always @(posedge clk) - Keep your sequential block simple

```
always @(posedge clk)
begin
  if (reset)
    operands_reg <= 32’d0;
  else if (state_reg == 2’d1)
    operands_reg <= operands_A;
  else if (state_reg == 2’d2)
    operands_reg <= opernads_B;
  end
end

always @(posedge clk)
begin
  if (reset)
    operands_reg <= 32’d0;
  else
    operands_reg <= operands_next;
end

always @(*)
begin
  operands_next = operands_reg;
  if (state_reg == 2’d1)
    operands_next = operands_A;
  else if (state_reg == 2’d2)
    operands_next = operands_B;
end
```

Verilog Guide Lines (4)

- Never assign X values, except when assign default values in a case statement, and last assignment in a ternary statement

```
always @(*)
begin
  state_next = state_reg;
  signal_rdy = 1'bX;

  if (operands_val)
  begin
    state_next = 2'b01;
    signal_rdy = 1'b0;
  end
end

always @(*)
begin
  state_next = state_reg;
  signal_rdy = 1'b1;

  if (operands_val)
  begin
    state_next = 2'b01;
    signal_rdy = 1'b0;
  end
end
```

Verilog Guide Lines (5)

- Always populate all possible cases when using case statements (But turning this case statement into a ternary statement is better)

```
always @(*)
begin
  operands_next = operands_reg;

  case (state_reg)
  2'd1: operands_next = operands_A;
  2'd2: operands_next = operands_B;
  endcase
end

always @(*)
begin
  operands_next = operands_reg;

  case (state_reg)
  2'd0: operands_next = operands_reg;
  2'd1: operands_next = operands_A;
  2'd2: operands_next = operands_B;
  2'd3: operands_next = operands_reg;
  default: operands_next = 32'bx;
  endcase
end

assign operands_next
  = (state_reg == 2'd1) ? operands_A
  : (state_reg == 2'd2) ? operands_B
  : operands_reg;
```

Verilog Guide Lines (6)

- Never use casex, use casez with caution
- Never use === operator

RTL/Gate-Level Mismatch
Verilog Optimism Problem

```
always @(*)
begin
  state_next = state_reg;
  signal_rdy = 1'b1;

  if (operands_val)
  begin
    state_next = 2'b01;
    signal_rdy = 1'b0;
  end
end
```

- What happens if operands_val is an X?
  - In RTL simulation, the simulator optimistically doesn't take the branch and just falls through
  - In Gate-level simulation, the X's are propagated
- This doesn't happen when you write a ternary operator. This is the reason why you should avoid always @(*) when it's possible

RTL/Gate-Level netlist Mismatch
Verilog Optimism Problem

```
always @(*)
begin
  state_next = state_reg;
  signal_rdy = 1'b1;

  if (operands_val)
  begin
    state_next = 2'b01;
    signal_rdy = 1'b0;
  end

  `RTL_PROPGATE_X(operands_val, state_next);
  `RTL_PROPGATE_X(operands_val, signal_rdy);
end
```

- You should put a RTL propagate X at the end
- RTL_PROPGATE_X macro propagates Xs when operands_val is an X to state_next and signal_rdy
- We provide you this macro

RTL/Gate-level netlist Mismatch
Verilog Pessimism Problem

```
assign b = a & ~a;
```

- What if a was an X?
- We know that b should be 1'b0 no matter what a is.
- But, Verilog evaluates a as an X.
- This is the reason why you should reset your registers.
- It also helps formal verification if the tool knows that a register has a default value.
- What do we do about wide datapath registers?
  - In our toolflow there's a script which initializes all registers to a certain value, so that we don't start with an X.