Last Time: CS 152 Course Introduction

- Single-cycle CPU project: 3 weeks
- Pipelined CPU: 4 weeks
- Final Project: 5 weeks, 200 hr/student
- Teams of 4-5 students

IBM Power 5 “die photo”: a die is an unpackaged part

CS 152 L2: Single Cycle Datapaths
Today: Single Cycle Datapath Design

The book presentation of single cycle processors is sufficient to do Lab 2.

This lecture is not.

This lecture is a gentle introduction, to prepare you to read the book ...
Single cycle data paths: Assumptions

Processor uses synchronous logic design (a “clock”).

All state elements act like positive edge-triggered flip flops.

<table>
<thead>
<tr>
<th>f</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MHz</td>
<td>1 μs</td>
</tr>
<tr>
<td>10 MHz</td>
<td>100 ns</td>
</tr>
<tr>
<td>100 MHz</td>
<td>10 ns</td>
</tr>
<tr>
<td>1 GHz</td>
<td>1 ns</td>
</tr>
</tbody>
</table>
Review: Edge-Triggered D Flip Flops

Value of D is sampled on positive clock edge. Q outputs sampled value for rest of cycle.
**Review: Edge-Triggering in Verilog**

Value of D is sampled on positive clock edge. Q outputs sampled value for rest of cycle.

```verilog
module ff(D, Q, CLK);
  input D, CLK;
  output Q;
  always @ (CLK)
    Q <= D;
endmodule
```

Module code has two bugs.

Where?
Review: Edge-Triggered D Flip Flops

Value of D is sampled on positive clock edge. Q outputs sampled value for rest of cycle.

module ff(D, Q, CLK);

input D, CLK;
output Q;
reg Q;

always @ (posedge CLK)
Q <= D;

endmodule
Administrivia: Upcoming deadlines ...

Friday: “Teams meet the TAs”, 12-1, 119 Cory. For 61(c) students, 150 Lab Lecture 1”, 1-2 PM, 125 Cory.

Monday: Lab 1 final report due, 11:59 PM, via the submit program.

Thursday: Lab 2 preliminary design document due to TAs via email, 11:59 PM.
Single cycle data paths: Definition

All instructions execute in a single cycle of the clock (positive edge to positive edge)

Advantage: a great way to learn CPUs.

Drawbacks: unrealistic hardware assumptions, slow clock period
Recall: MIPS R-format instructions

Syntax: ADD $8 $9 $10  Semantics: $8 = $9 + $10

Fetch next inst from memory: 012A4020

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
</table>

Decode fields to get: ADD $8 $9 $10

"Retrieve" register values: $9 $10

Add $9 to $10

Place this sum in $8

Prepare to fetch instruction that follows the ADD in the program.
Goal #1: An R-format single-cycle CPU

**Syntax:** ADD $8 $9 $10  **Semantics:** $8 = $9 + $10

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
</table>

Sample program:
ADD $8 $9 $10
SUB $4 $8 $3
AND $9 $8 $4
...

No branches or jumps: machine only runs straight line code.

No loads or stores: machine has no use for data memory, only instruction memory.

How registers get their initial values are not of concern to us right now.
Separate Read-Only Instruction Memory

Reads are **combinational**: Put a stable address on input, a short time later data appears on output.

Not concerned about how programs are loaded into this memory. Related to separate instruction and data caches in “real” designs.
Task #1: Straight-line Instruction Fetch

Fetching straight-line MIPS instructions requires a machine that generates this timing diagram:

Why do we increment every clock cycle? Why +4 and not +1?

PC == Program Counter, points to next instruction.
New Component: Register (for PC)

Built out of an array of flip-flops

In later examples, we will add an “enable” input: clock edge updates state only if enable is high.
New Component: A 32-bit adder (ALU)

**Combinational**: Put a \( A \) and \( B \) values on inputs, a short time later \( A + B \) appears on output.

**ALU**: Combinational part that is able to execute many functions of \( A \) and \( B \) (add, sub, and, or, ...). The “op” value selects the function.

Sometimes, extra outputs for use by control logic ...
Design: Straight-line Instruction Fetch

CS 152: State machine design in the service of an ISA

+4 in hexadecimal

CLK

Addr

Data

PC

PC + 4

PC + 8

IMem[PC]

IMem[PC + 4]

IMem[PC + 8]
Goal #1: An R-format single-cycle CPU

Syntax: ADD $8 $9 $10  Semantics: $8 = $9 + $10

Done! To continue, we need registers ...

Fetch next inst from memory: 012A4020

 Supply fields to get: ADD $8 $9 $10

“Retrieve” register values: $9 $10

Add $9 to $10

Place this sum in $8

Prepare to fetch instruction that follows the ADD in the program.
Register files: From the top down

Why is R0 special?

"two read ports"

Even more interesting from the bottom up...
Register File Schematic Symbol

Why do we need WE?

If we had a register file w/o WE, how could we work around it?
Syntax: ADD $8 $9 $10  Semantics: $8 = $9 + $10

What do we do with these?

Fetch next inst from memory: 012A4020

Decode fields to get: ADD $8 $9 $10

“Retrieve” register values: $9 $10

Add $9 to $10

Place this sum in $8

Prepare to fetch instruction that follows the ADD in the program.
Computing engine of the R-format CPU

Decode fields to get: ADD $8 $9 $10

```
+---------+----+----+----+----+---+---+
| opcode | rs | rt | rd | sham | funct |
+---------+----+----+----+------|------+
```

What do we do with WE?
Putting it all together ...

Is it safe to use same clock for PC and RegFile?

To rs1, rs2, ws, op decode logic ...

RegFile

rs1  rs2  rd1  
5      5     32

ws  rd2  
5  32

wd  WE

32

0x4  32  32  32

PC

32

Instr Mem

Addr  Data

32

Logic

32

32

32

ALU

op

rs1, rs2, ws, op decode logic ...
Next Steps:

* Design stand-alone machines for other major classes of instructions: branches, load/store, immediates.

* Learn how to efficiently “merge” single-function machines to make one general-purpose machine.

* Implementing control structures for the single-cycle datapath.
Memory Instructions: \texttt{LW} \ $1, 30($2)

- **Instruction Fetch**: Fetch the load inst from memory
  - Instruction: \texttt{LW} \ $1, 30($2)  
    - "I-Format"
  - Decode fields to get:  \texttt{LW} \ $1, 30($2)
  - "Retrieve" register value: \( \$2 \)

- **Operand Fetch**: Compute memory address: \( 30 + \$2 \)

- **Execute**: Load memory address contents into: \( \$1 \)

- **Result Store**: Prepare to fetch instr that follows the \texttt{LW} in the program. Depending on load semantics, new \( \$1 \) is visible to that instr, or not until the following instr ("delayed loads").
Branch Instructions: BEQ $1, $2, 25

Fetch branch inst from memory

```
| opcode | rs | rt | offset |
```

“l-Format”

Decode fields to get: BEQ $1, $2, 25

“Retrieve” register values: $1, $2

Compute if we take branch: $1 == $2 ?

ALWAYS prepare to fetch instr that follows the BEQ in the program (“delayed branch”). IF we take branch, the instr we fetch AFTER that instruction is PC + 4 + 100.

PC == “Program Counter”