CS 2630
Computer Organization

What did we accomplish in 15 weeks?
Brandon Myers
University of Iowa
Why take 2630?

• The esoteric answer: **Computer** Science graduates should have an appreciation for how real **computers** work

• But really...
  • 1. It will be up to you to **design our new computer systems**...computer architects have been panicking for nearly a decade and they are *not* calming down
  • 2. Even if you vow to never, ever, EVER do anything except applications programming...at some point you will be have to **measure a system you’ve built**: performance (latency & throughput), energy usage, reliability, ... To understand how to measure/interpret/improve your system, you need to understand more of the computer
High-level language (e.g., C, Java)

Compiler

Operating system (e.g., Linux, Windows)

Instruction set architecture (e.g., MIPS)

Memory system  Processor  I/O system

Datapath & Control

Digital logic

Circuits

Devices (e.g., transistors)

Physics
You learned how to write assembly code in HW2 (usually the compiler does the work for us)

**rug**: don’t need to write assembly code for a particular architecture. Instead write portable Java/Python/C code

**bumps**: some C code isn’t portable; some programmers write snippets of assembly code when the compiler doesn’t do the best thing

```java
public void append(int data) {
    if (this.next==null) {
        this.next = new ListNode(data);
    } else {
        this.next.append(data);
    }
}
```

```assembly
lw $t0, 4($s0)
addi $t0, $t0, 10
sw $t0, 8($s0)
```
**rug:** we can write MIPS programs in a language made of human-readable characters, use pseudo instructions, refer to labels even though the machine reads binary numbers.

```assembly
lw $t0, 4($s0)
addi $t0, $t0, 10
sw $t0, 8($s0)
```

10001110000010000000000000000100
00100001000010000000000000001010
10101110000010000000000000000100

Project 1 – MiniMA the MIPS assembler
rug: linker allows us to write our programs modularly

```assembly
lw $t0, 4($s0)
addi $t0, $t0, 10
sw $t0, 8($s0)
```
PEER INSTRUCTION (actually, a survey)

<table>
<thead>
<tr>
<th>strongly agree</th>
<th>agree</th>
<th>neutral</th>
<th>disagree</th>
<th>strongly agree</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
</tr>
</tbody>
</table>

1. I understand the relationship between bits, numbers, and information.
2. I understand the stored program concept
3. I understand the role of the instruction set architecture in a computer
4. I understand why abstractions are essential for building complex systems
5. I understand why the digital abstraction is important
6. I understand why the synchronous abstraction is important
7. I understand the tradeoffs in the memory hierarchy
8. I understand how problems can be decomposed into a datapath and a control
9. I appreciate the layers of the computing stack and why they may need to change in the near future.
In Project 2-2 you played the role of the Loader by loading your hex image into the Instruction memory.

**rug:** our program has the illusion of having access to the entire address space (e.g. all $2^{32}$ bytes) of the computer.
You designed, built, and tested a processor that runs assembled MIPS programs. **Rug:** Machine code for the MIPS architecture ought to run on any MIPS processor, regardless of its design (its microarchitecture).

**Bumps:** Choices about the architecture sometimes are based on assumptions about the microarchitecture (e.g., MIPS branch delay slot).
We can build a complex system out of basic components. Synchronous abstraction allows us to not have to worry about interfaces between components. Project 2-1, HW4: you built components (like register files and finite state machines) from sequential logic.
You don’t have to build the 5-stage pipelined MIPS processor.

You don’t have to build a MIPS processor.

Instruction set architecture (e.g., MIPS)

neural network structure and weights

GPU instruction set

you don’t have to build a MIPS processor

MIPS R10000 (out-of-order superscalar)

Digital logic

Circuits
Every component is made of logic gates; you learned how to build logic circuits in HW3.
logic gate made of pMOS and nMOS transistors arranged in a CMOS configuration

It is a bit easier just to think of gates that are functions as opposed to transistors attaching output to Vsource or Vground.

CMOS ensures every gate has a pure 0 or 1 output! This idea is the digital abstraction that lets the layers above compose two electrical circuits without worrying about how they affect each other.
When building a functional digital logic circuit, no need to worry about how it is arranged on the silicon.

layout engineer’s view of a NOR gate

https://commons.wikimedia.org/wiki/File:NOR_gate_layout.png
rug: device engineers provide layout engineers with “design rules”. If they obey the required spacing between components then the transistors will work

nMOS cross section
**rug**: When operating a transistor in the saturation regimes it looks like an electrical switch between voltages GND and VDD. Part of supporting the digital abstraction.
The Creation of Adam
The programmable computer

blatantly stealing a tradition from my Computer Organization instructor

this slide set to Handel’s Hallelujah chorus
It’s not enough to just build software these days all in production not just research

computer vision processors running Google’s augmented reality platform Tango

Google TPU is built for machine learning

“holographic” processor for Microsoft’s augmented reality platform HoloLens

Sparc M7 chip is built specifically for accelerating database queries

Project Catapult: custom hardware running part of Bing search
Life beyond Logisim?

- Logisim’s main mode of input is schematic entry
- Much digital logic design uses hardware description languages (HDL) like Verilog (look up Verilog in your textbook index)
  - HDL is not much different than what you did, except it is textual instead of graphical
  - Typically have powerful compilers than make development easier than using Logisim, e.g., write a statement like

```plaintext
case (ALUCtrl) {
  0: R = X+Y
  1: R = X-Y
  ...
}
```

And you get an ALU!
Did we really build a real processor?

• Yes! You implemented much of the MIPS Instruction Set Architecture and I/O. Your Project 2-2 could run Linux (at 4KHz clock frequency) given a bootloader program and Linux compiled for MIPS.

No, I mean like *real* hardware
No, I mean like *real* hardware

If we use a *hardware compiler*, we could turn your logisim files (look inside; it’s just some XML listing a bunch of components and wires) into an FPGA design or standard-cell VLSI design

more to learn about how to deal with the details of these design flows, but you have a good starting point
Administrivia

• Final Exam
  • Friday, 3-5pm in here!
  • open notes/book, no electronics
  • reminder: practice materials on ICON announcement
The insights you brought to the course: CA Topics
What courses next?

- CS:3620 Operating Systems
- CS:3210 Programming languages and Tools (in C++)
- CS:3640 Introduction to Networks and Their Applications
- CS:3820 Programming Language Concepts
- CS:4640 Computer Security
- CS:4700 High Performance and Parallel Computing
- CS:5610: High performance computer architecture
- CS:4980 Topics in CS (Compiler Construction on raspberry pi)
- CS:4980 Topics in CS (ask for a computer architecture course!)
What’s to learn next: operating systems
Questions we didn’t get to answer fully in CS2630

Operating systems

• how do multiple programs share the computer?
  • 2-64 processors
  • 1 network interface
  • 1 memory
  • 1 keyboard, mouse, screen
  • 100’s of running programs

• how do you keep programs isolated from each other or one program from consuming all resources?

• how do you implement syscalls?

• how do you load the OS code into memory when you power on the computer?
What’s to learn next: computer architecture
What’s to learn next: computer architecture

- the role of parallelism in microarchitectures
- every implementation and its effect on performance...

\[
\frac{\text{seconds}}{\text{program}} = \frac{\text{seconds}}{\text{cycle}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{instructions}}{\text{program}}
\]

...and cost and energy
Parallelism in architectures

**pipelining**

<table>
<thead>
<tr>
<th>Instr. No.</th>
<th>Pipeline Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>2</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>3</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>4</td>
<td>IF ID EX MEM</td>
</tr>
<tr>
<td>5</td>
<td>IF ID EX</td>
</tr>
</tbody>
</table>

**Clock Cycle**

1 2 3 4 5 6 7

**super scalar**

| IF ID EX MEM WB |
| IF ID EX MEM WB |

**vector**

| IF ID EX MEM WB |
| IF ID EX MEM WB |
| IF ID EX MEM WB |
| IF ID EX MEM WB |

**dataflow**

and others... multicore, VLIW, multithreading, ...
Vector machines

SIMD: single instruction, multiple data

found in...
- early super computers
- Intel AVX
- GPUs
Superscalar machines

Replicate resources, e.g.,
- two decoders, 2-wide
  instruction cache read port: fetch two instructions at a time
- two ALUs: execute two
  instructions at a time
- more register file write ports: write back two registers in one cycle

found in... most CPUs in servers and smartphones
Dataflow machines

A processor needs to get the instruction and the input data to the same physical place at the same time (known as “dataflow locality”)

Dataflow machines have a bunch of Execution units of various kind; the data ”flows” through the operators

Challenges?
What’s to learn next: parallel computing
Metric for performance comparison: Time

My program runs in 100 seconds

If I “parallelize it” on 10 processors I saw that it runs in 12 seconds

What is the speedup?

\[
\frac{T_{\text{serial}}}{T_{\text{parallel}}} = \frac{100}{12} = 8.33X
\]
Predicting parallel running time ($T_{par}$) from serial running time

My program runs in $T_{ser} = 100$ seconds

If I “parallelize it” on 10 processors, how fast will it run (i.e., what is $T_{par}$)?

\[ T_{improved} = T_{original} \times ((1 - r) \times 1 + r \times \frac{1}{s}) \]

$r =$ fraction of program that is able to be improved
$s =$ speedup when applying the improvement

In this form, it is called **Amdahl’s law**: says your speedup is limited by how much of the program is improved (e.g., parallelize)
Amdahl’s law applied to parallelization

A sequential abstract machine model you already know

- RAM: random access memory
- just like any other computational step, accessing memory is cost of 1
One of the foundational parallel machine models: Parallel Random Access Machine (PRAM)

- All processors are attached to a shared memory

- Memory access takes 1 step

- More realistic variants of PRAM incur greater cost for “conflicting” memory accesses

- used very often for understanding the speedup limits of parallel algorithms; not very realistic
One of the foundational parallel machine models: Bulk synchronous parallel (BSP)

this abstract machine does not support as many algorithms as CTA, but it is simpler

(see blackboard notes)

https://en.wikipedia.org/wiki/Bulk_synchronous_parallel

https://en.wikipedia.org/wiki/Bulk_synchronous_parallel
The future of CS2630

• Please stay in touch!
• Tell others how awesome CS2630 is!
• Sign up to be an approved tutor!
  https://cs.uiowa.edu/resources/approved-tutors

• CS2630 moving to TILE classrooms in Fall
• Replacing some lectures with lab assignments
  • allow us to better support learning all the tools, get more time analyzing and designing programs and circuits
  • very speculatively: future opportunity for lab assistants (help students but do not grade work)