16-dpath-uarch-notes

Datapath into Microarchitecture

Agenda

- Re-O
- 1. Leftovers
- 2. Processing and the Datapath
- 3. The Datapath, One Level Deeper
- 4. Operation as one might imagine
- 5. Operation, as Chapter 4 Develops
- 6. How can this possibly work?

Reading:

- 4.3.1
- 4.3.2
- 4.3.3

Here are the slides (https://ssl.cs.dartmouth.edu/~sws/cs51-s15/16-dpath-uarch/slides.pdf)

---

0. Re-Orienting

The layers, from the human down to the physics

The Y86 as a simplified version of the dominant X86 family

And... we started bridging the gap between the ISA and the datapath; between software and hardware!

Right now, what we're seeing is as rather direct correspondence between "machine" and "computation." What the machine is doing exactly matches what we imagine it's doing when we think about the assembly language instruction.

Architecture started with this nice correspondence. However, over the last few decades, it has successively moved away from this, sacrificing cleanliness of correspondence for increased speed.

Over the next few weeks, we'll start showing you that path.

---

1. Leftovers

"Whence the 6 in 8086?"

---

2. Processing and the Datapath

We will resume thinking about Fig 4.18 and 4.22 **together**

(In class, what I will be doing is going through the pseudocode from the book, step by step, while also showing each corresponds to operations in the datapath.)

```
rmmovl
```

```
mrmovl
```

```
pushl
```

```
jxx
```

```
call
```

```
ret
```

3. The Datapath, One Level Deeper

Fig 4.23, and stages of processing

4. Operation as one might imagine

We COULD put registers at all the right places, and then do each operation in sequence.

But how many clock cycles would that take?

5. Operation, as Chapter 4 Develops

Quoting the textbook:

- "We are left with just four hardware units that require an explicit control over their sequencing—the program counter, the condition code register, the data memory, and the register file."
- "The processor never needs to read back the state updated by an instruction in order to complete the processing of this instruction."
- "As another illustration of this principle, we can see that some instructions (the integer operations) set the condition codes, and some instructions (the jump instructions) read these condition codes, but no instruction must both set and then read the condition code."

So why don't we have.....

```
        mmovl (%eax), (%ebx)
        addl (%eax), (%ebx)
```

We have the additional wrinkle of IMem and DMem access not always being "instant...."

6. How can this possibly work