Critical path
remaining topics
control unit design (today)
ISA design tradeoffs (Monday)
advice (Wednesday)

290
& notes
 oily
(extra)

hardwired control
using a memory for
logic functions
microprogrammed
control

Lab #4 - due next Tuesday
India + FERP today

25 min. Men: ICES

platform survey for next sem.
Android, iPhone, webos, winphone

research
Encoding a control unit

Strategy one (of two): Hardwired control

- Fix the number of cycles for each of these two stages
- Drive system with a counter
- Use combinational logic to map counter value + IR to control signals
  (PC is just used as memory address for fetch, so not too complex... for now)

This approach in general is called hardwired control.
How many clock cycles do we need for fetch? How many for execute?

- Depends on two factors
  - Complexity of ISA
  - Capability of datapath

If we design for one-cycle execution of all instructions
- called single-cycle, hardwired control
- otherwise then
  - but cycle time limited by slowest instruction

Typically, we use simpler datapaths and break instruction processing into multiple steps (as you've seen with LC-3 design)

- Result is called multi-cycle, hardwired control (still using the counter - LC-3 design in book is not this type)

- Since instruction processing times vary, add a 'counter reset' signal at end of processing
- Might also add a counter 'pause' signal to accommodate waiting for memory (for example)

So we have, for a general hard wired control unit design, ...

What about the combinational logic?

- As mentioned, PC just used directly as memory address for fetch cycle(s).
- But that leaves counter bits + 16-bit IR
  \[ \Rightarrow \text{huge K-maps! ? (No)} \]
- Careful ISA encoding allows simple wiring for some datapath controls
  In LC-3, for example,  
  \[ DR \leftarrow IR[11:9] \]
  \[ SR1 \leftarrow IR[8:6] \]
  \[ SR2 \leftarrow IR[5:0] \]
By design, remaining control signals depend only on 'state' of control unit FSM (counter bits in a hardwired design), and opcode (\(R[15:12]\) in LC-3).

Let's imagine building a hardwired control for LC-3... How many bits in a counter for LC-3?

<table>
<thead>
<tr>
<th># stage</th>
<th>max # cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetch</td>
<td>3</td>
</tr>
<tr>
<td>decode</td>
<td>(implicit in this design)</td>
</tr>
<tr>
<td>process</td>
<td>5</td>
</tr>
</tbody>
</table>

Total is 8 \(\Rightarrow\) 3 bits

So: 3-bit counter + 4-bit opcode

**\(\Rightarrow\) control signals**

(40 of them, if we include our control unit's counter RST input, and exclude the register file, which we already solved)

That's a lot of 7-variable k-maps... is there an easier way?
Using a Memory for Logic Functions

Implementing logic functions with a memory (possibly a read-only memory)
- is not a bad strategy in general
- especially for few bits \(\Rightarrow\) many bits

(variables) \(\rightarrow\)

(Synthesis tools (or hard work) can, of course, produce smaller designs, using fewer gates. But memory approach is easier to modify.
- if we make a mistake, memory size/speed is unaffected
- same if we have extra space (addresses, which have been carry in our example are FSM/control unit states) - we can change memory contents to odd/modify instructions
a couple of analogues

- look up table in software
  - used for fast trig. function approximation in low-end devices (esp. graphics)
  - used for BISR counting when not supported by ISA
- etc.

- look up table (LUT) in field-programmable gate array (FPGA)
  - modern hardware prototyping tool (see programmable logic array/PLA in textbook)

- speed of hardware, flexibility of software
- increasingly common for early-generation chip designs
- will use in 298 (385 new version)
A caveat: most people would not call a ROM-based design a 'hardwired control unit,' but depends how you view it. Memory is just one way to build combinational logic circuits; design is the same.

Using a memory will make our second control unit design strategy easier to understand...

Strategy two (of two): microprogrammed control
Let's treat the control unit as a program! ROM will hold microinstructions.

Encoding the FSM is now just putting microinstructions into memory and deciding which one to execute next—called sequencing.

Notice that most of the time, there's no choice (we have only one next microinstruction), so one simple approach:

Add the "address" (6-bit state ID) of the next microinstruction to our ROM. ⇒ so 2^6 x 45-bits instead.

---

The LC-3 state machine design in the book was designed to further simplify implementations:

- Other than reg file (IR, SR1, SR2), control signals depend on FSM state only (not IR).
- Next state can depend on IR, so IR can influence processing.
- Only 2^6 addresses needed x 39 bits for control signals.
Sometimes we need two possible next states—waiting for memory ready R, for example.

Add two addresses into ROM.

26 x 51-bit ROM

Program branch control

State ID register (6 bits)

Program branch control

ADD

26 x 51-bit ROM

DATA

Control signals

Program branch control is some Boolean logic expression based on things like the memory ready signal R, branch enable signal BEN, and current state ID.
What's missing? **DECODE!**

- need 16 possible next states
  
  - solution:
    - add a mux
    - pick state IO's in a way that it easy to use

For example, \[ IR[15:12] \] is opcode

Let's encode first processing state

for each opcode as...

\[ \begin{array}{cccc}
\end{array} \]
Then we have:

- Program branch control
- State ID register (6 bits)
- Decode state
- ADDR
- 2*51-bit ROM DATA
- Next

Now we just pick state ID #5 to make the program branch control simple and fill in ROM to implement RTL and encode next state(s).

One simplification:
- if a state doesn't branch
- fill in both ROM address fields with the same value
- Program branch control is then a don't care for that state
So, for example

If's that simple!

Appendix of textbook
- has a slightly more complex version
- in part because their design includes interrupts