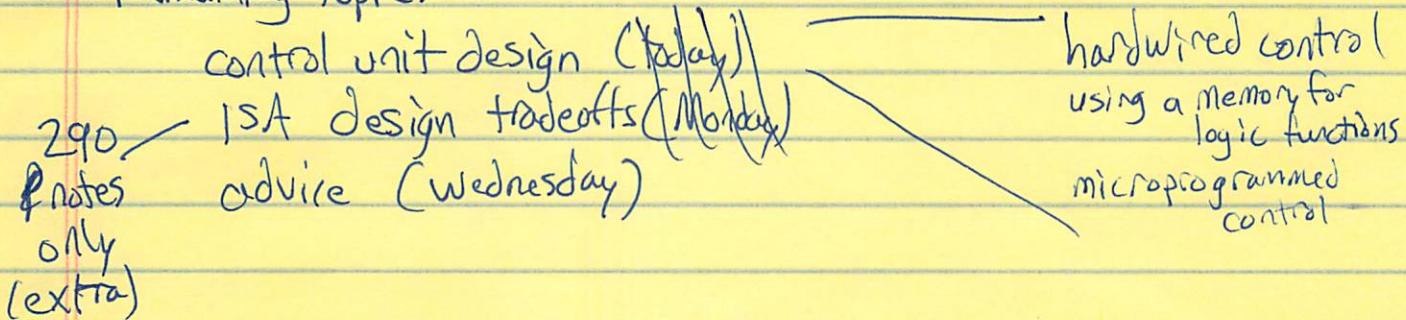


ECE199JL

7 Dec '12

L38PA

Critical path
remaining topics



Lab #4 — due next Tuesday

India + FCRP today

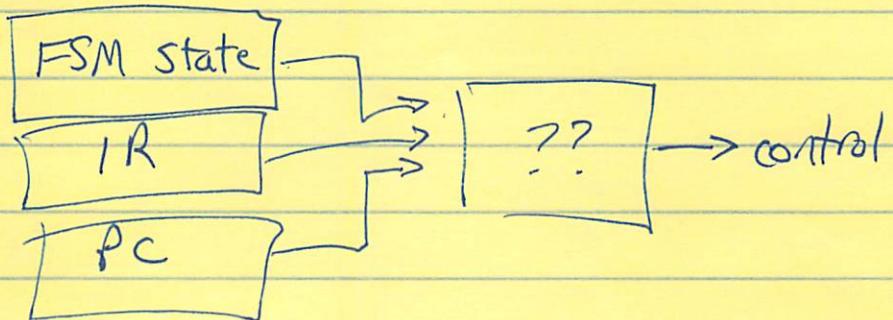
25 min Mon : ICES

platform survey for next sem.

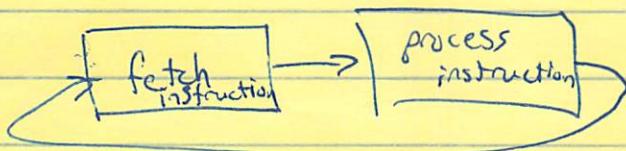
Android, iPhone, webOS, Windows Phone

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?~~
processing

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

If we design for one-cycle ~~execution~~^{processing} of all instructions

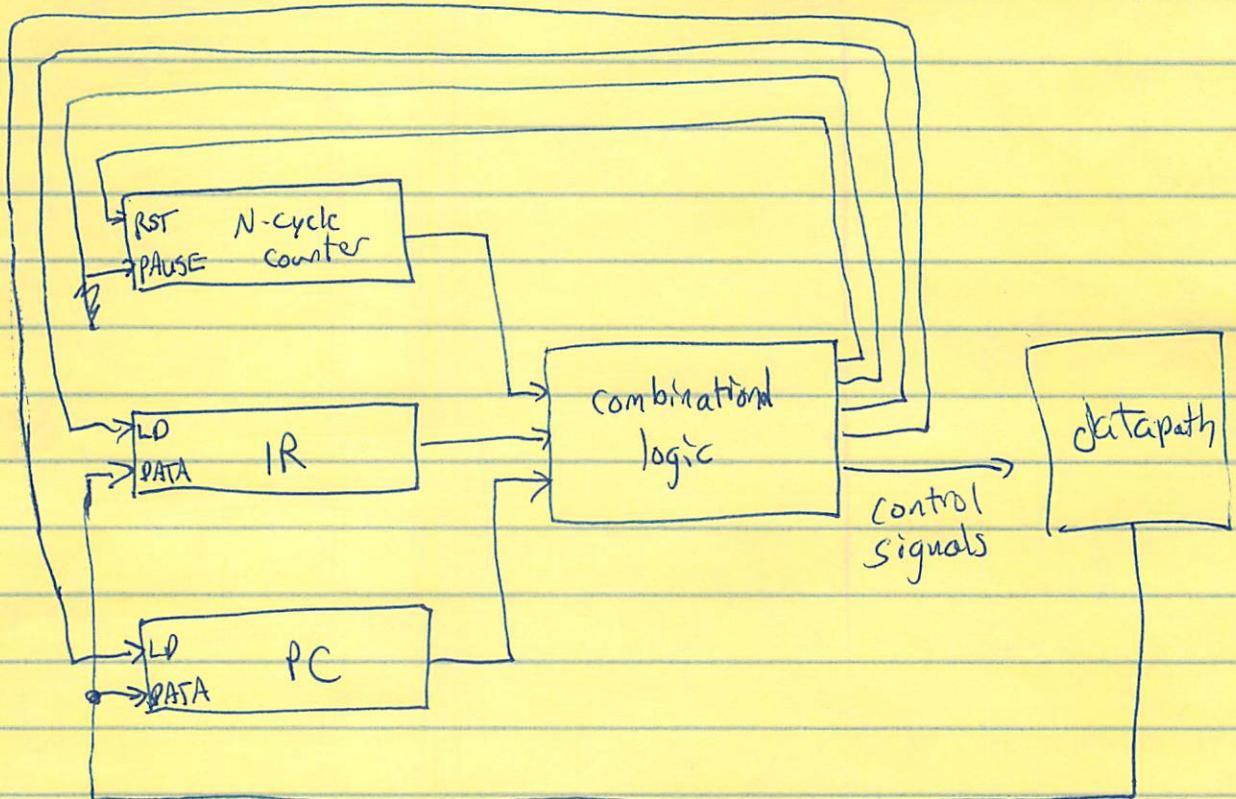
- called single-cycle, hardwired control
- ~~otherwise~~ ^{then}
- but cycle time limited by slowest instruction

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~~^{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 hardwired 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 huge K-maps!? (No)
- Careful ISA encoding allows simple wiring for some datapath controls
 In LC-3, for example,
 - $DR \leftarrow IR[11:9]$
 - $SR1 \leftarrow IR[8:6]$
 - $SR2 \leftarrow IR[5:3]$

⇒ By design,
 Remaining control signals depend only on
 'state' of control unit FSM (counter bits
 in a hardwired design), and opcode
 $(IR[15:12] \text{ in LC-3})$

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

<u># stage</u>	<u>max # cycles</u>
fetch	3
decode	(Implicit in this design)
process	5

total is $8 \rightarrow 3$ bits

So: 3-bit counter + 4-bit opcode

⇒ control signals

(40 of them, if we include
 our control unit's

counter RST input)

and exclude the register
 file, which we already solved)

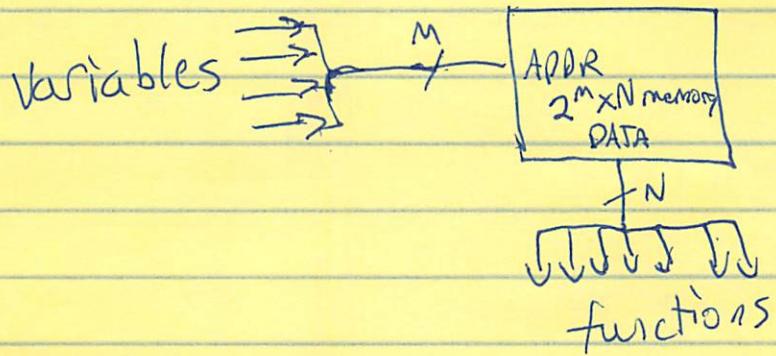
That's a lot of 7-variable k-maps... is there an
 easier way?

[based on book's
 FSM states, and
 ignoring interrupts
 + RTI since
 we haven't discussed/
 used]

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



(tradeoff)

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 ~~here cannot~~ in our example are FSM/control unit states) — we can change memory contents to add/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 bit 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

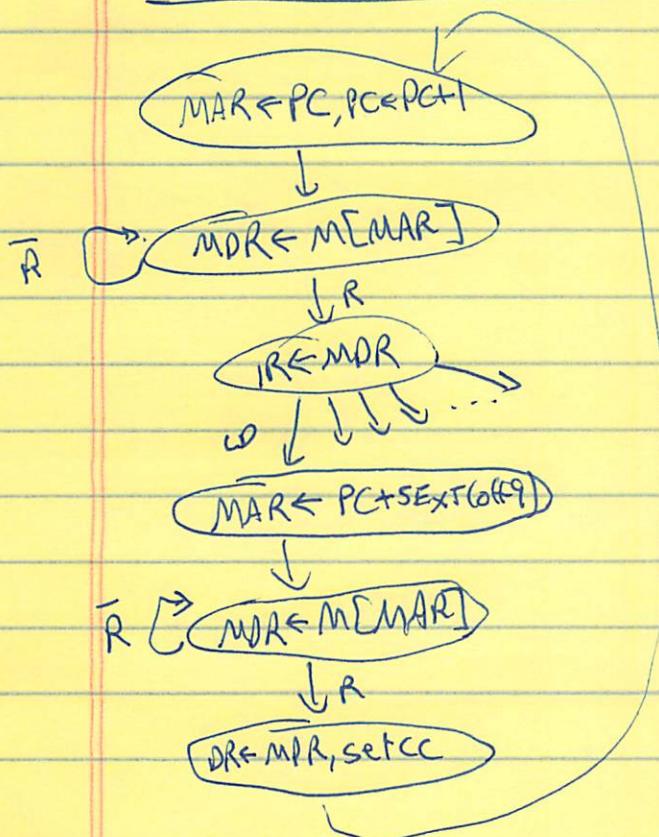
— increasingly common for early generation chip designs

— will use in 298 (385 new version)

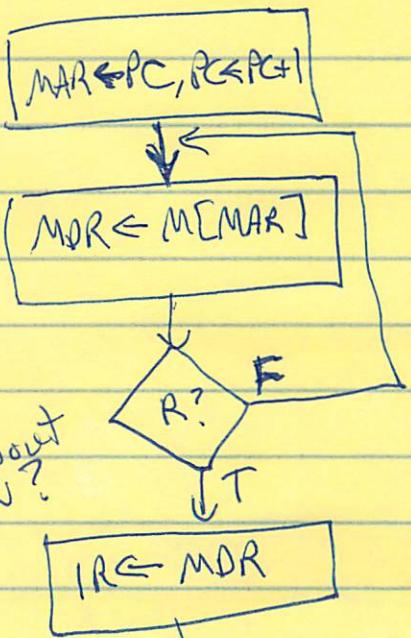
- A caveat: most people would not call a ROM-based design a hardwired control unit,
- but depends how you view it
 - memory is just one way to build combinational logic funcs.;
design is the same---

Using a memory ^{to encode our control signals} will make our second control unit design strategy easier to understand...

Strategy two (of two): microprogrammed control



Remind
you
of
anything?



How about
now?

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.

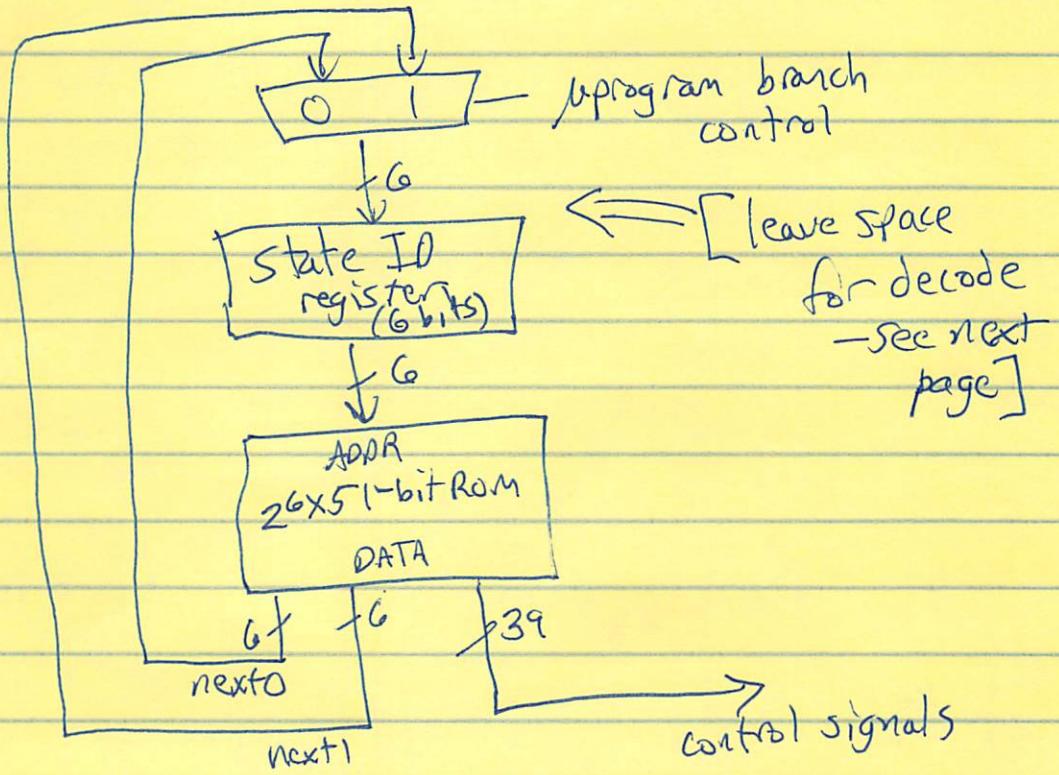
Notice that most of the time, there's no choice (we have only one next microinstruction), so one simple approach:

↙ add the "address" (6-bit state ID)
μ of the next microinstructions
to our ROM. \Rightarrow so $2^6 \times 45$ -bits instead

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

- other than reg file (DR, SR1, SR2), ~~control~~ 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 $\times 39$ bits for control signals

- Sometimes we need two possible next states
 - Waiting for memory ready R, for example
 - add two addresses into ROM
 - $2^6 \times 51$ -bit ROM



μ 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 ID's in a way
that is easy to use

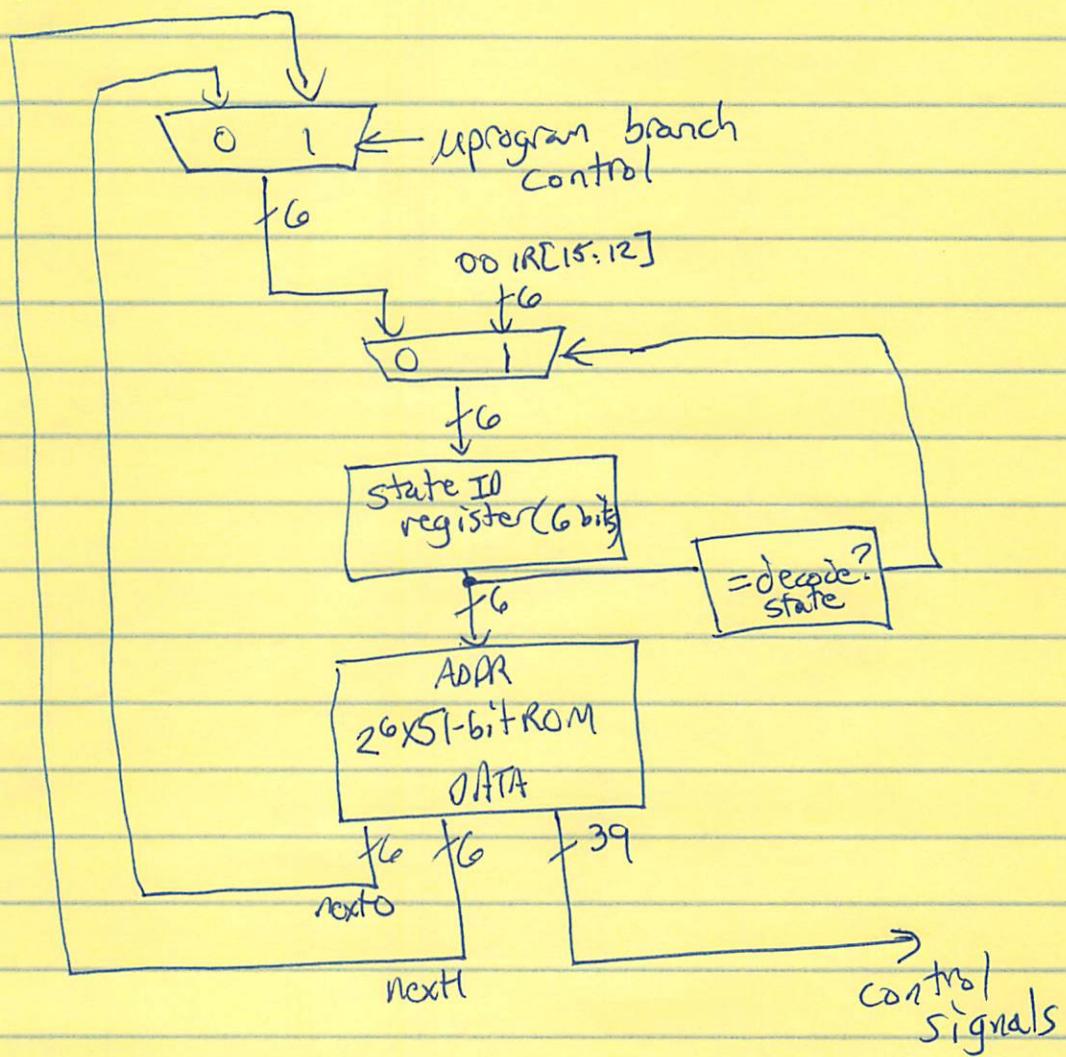
For example, $IR[15:12]$ is opcode

Let's encode first processing state

for each opcode as ...

D O $IR[5]$ $IR[4]$ $IR[3]$ $IR[2]$

Then we have ...



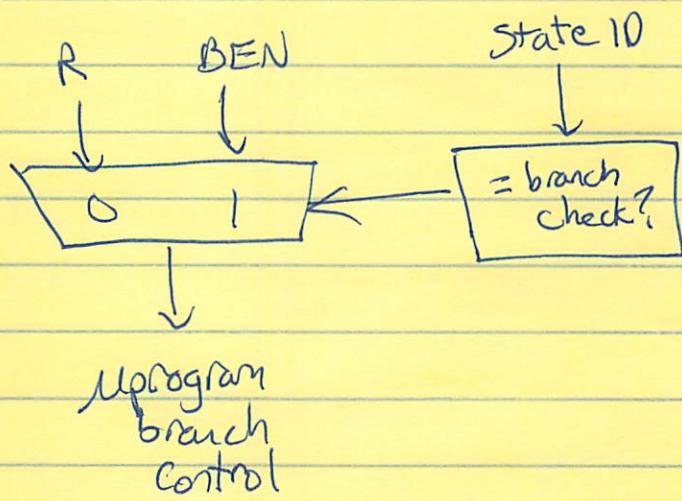
Now we just pick state ID #'s to make the μ program branch control simple and fill in ROM to implement RTL and encode next state(s).

One simplification

- if a state doesn't branch
- fill in both ROM address fields with the same value

- μ program branch control is then a don't care for that state

So, for example



It's that simple!

Appendix of textbook

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