Processor Implementation

Overview

- Start with common datapath design concepts
- Simple processor implementation
- Move to more complex implementation
- Finally, pipelined implementation

We'll look at how modern hardware works:

- Clocking, combinational logic, etc
- Pipelining
- Dealing with complexity

  a.k.a. how do they build those 10 million transistor chips??

Designing Complex FSMs

How many states do we need for a MIPS CPU?

- MIPS has 32 32-bit registers
- Each register could be in one of 2^{32} states
- We need at least 31×2^{32} states!
  (register 0 is 0)

... so we clearly don't want to draw the FSM or write the truth-table!

Idea: exploit FSM *structure*

Datapath Concepts

Datapath:

- Part of an architecture that manipulates data
- Tends to be "regular"

Example: ALU, MUX, etc.

Control:

- Tells datapath what to do

Example: alu opcode, MUX select input, etc.

Datapath Design

Steps in designing a processor:
(or any other piece of hardware)

- Start with high-level specification
  processor: the ISA
- Identify major storage elements and signals
  processor: registers, memory, ...
- Translate specification
  determine operations on storage elements
  (RTL, register transfer language)

Datapath Design

More steps...

- Pick computation blocks

  processor: alu

- Determine connections

  data-dependencies among different blocks

  i.e., the *datapath*

- Determine control inputs to datapath blocks

... and then put everything together.

Cross-cutting issue: clocking strategy

when do storage elements get updated?

Datapath Design Example

Example: (unsigned integers)

```
x = 0;
for (i=1; i <= N; i++)
    x = x + i;
```

Here `N` is an input and we should produce the result that is stored in `x`.

First step: make the specification precise in terms of signals and clocks.

Datapath Design Example

Better specification of I/O behavior:

- Input `N` arrives on a bus that is 4 bits wide and `N` is non-zero
- Output `x` is 7 bits wide
- Output `xdone` (1 bit) is set to 1 when the `x` output holds the correct data
- `xdone` stays high for 1 cycle, after which the next `N` input is read and the computation proceeds as before
- Signals change at the positive edge of the clock

Pick Storage Elements

There is normally a choice here. For our example:

- `i`: 4 bit storage element
- `N`: 4 bit storage element
- `x`: 7 bit storage element
- `xdone`: 1 bit output signal

Another option: rewrite the loop and eliminate one of the storage elements!

```
x = 0;
for (i=N; i >= 1; i--)
    x = x + i;
```

CSL

Translate Specification

Pick states, and determine operations in each state.

A systematic way: translate program using gotos...

```
initial:
   xdone = 0; x = 0; i = N;
   goto loop;
loop:
   xdone = 0; x = x + i; i = i - 1;
   if (i == 0) goto done;
   else goto loop;
done:
   xdone = 1;
   goto initial
```

Translate Specification

Idea:

- Each label is a state
- Advance from one state to the next every cycle
- End of program fragment at each label is a `goto`

  all paths lead to a goto!

  no intervening labels

RTL: normally uses "<-" for assignment

What about Concurrency?

Everything happens in parallel in hardware...

```
loop:
   xdone = 0; x = x + i; i = i - 1;
   if (i == 0) goto done;
   else goto loop;
```

- variables are updated when the state *changes*
- the state-holding element for i holds the value of i at the beginning of the state
- computation blocks have to handle this

Computation Blocks

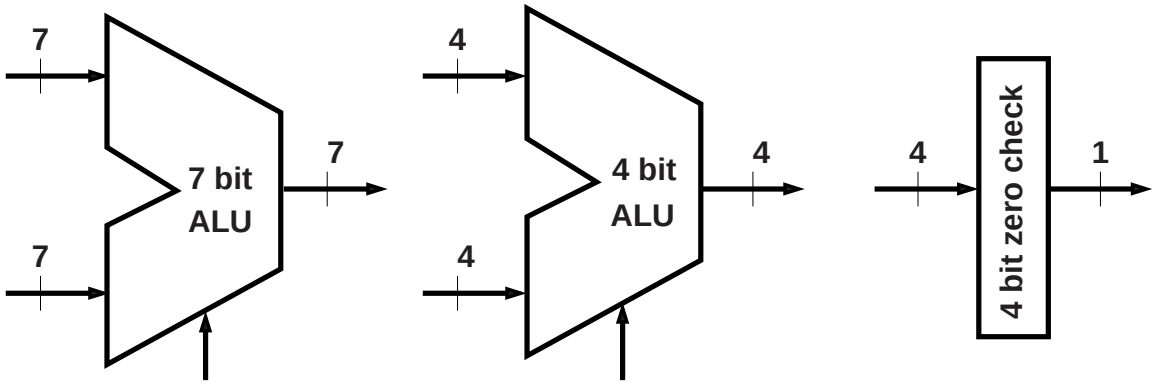
Operations on storage elements/signals:

- `i`
  - `i <- N, i <- i - 1, i == 0`
- `x`
  - `x <- 0, x <- x + i`
- `xdone`
  - `xdone <- 0, xdone <- 1`

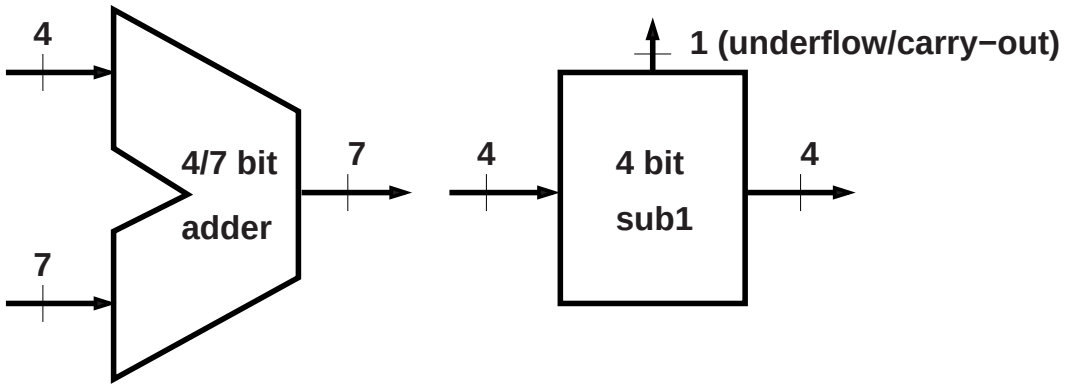
You can see why `x` and `i` are state-holding elements...

Blocks: "subtract 1," adder, compare to zero

Computation Blocks



or...



Reduce, reuse, recycle...

Data Dependencies

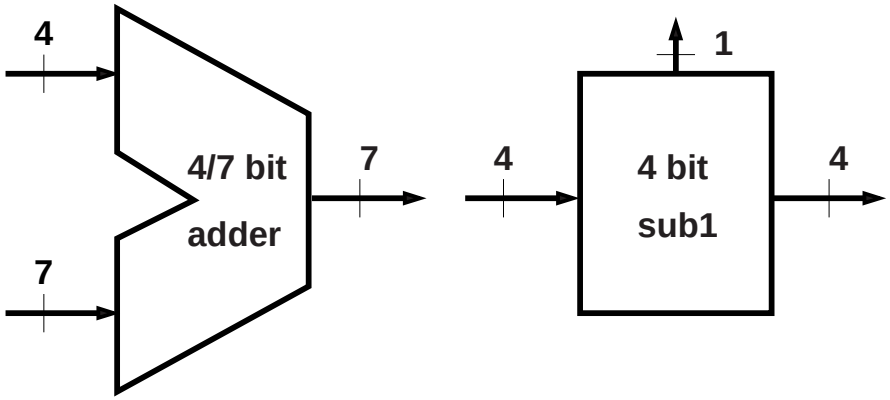
For each storage element, find dependencies.

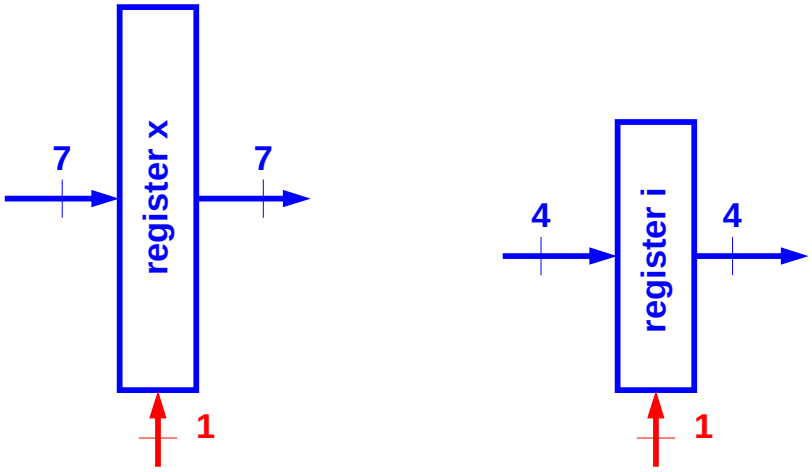
- `i`
  - set to `N`, output of `sub1` block
  - used by `sub1` block, `adder`
- `x`
  - set to `0`, output of `adder`
  - used by `adder`

(In general case, determine how computation blocks are interconnected too.)

What is the data flow?

Data Dependencies



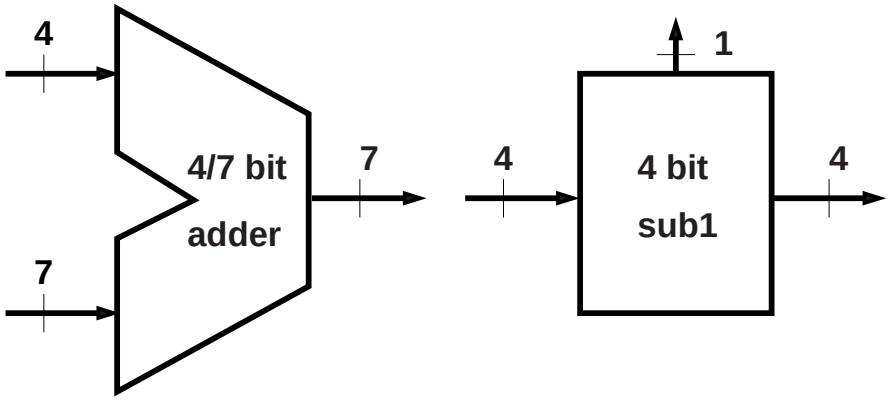


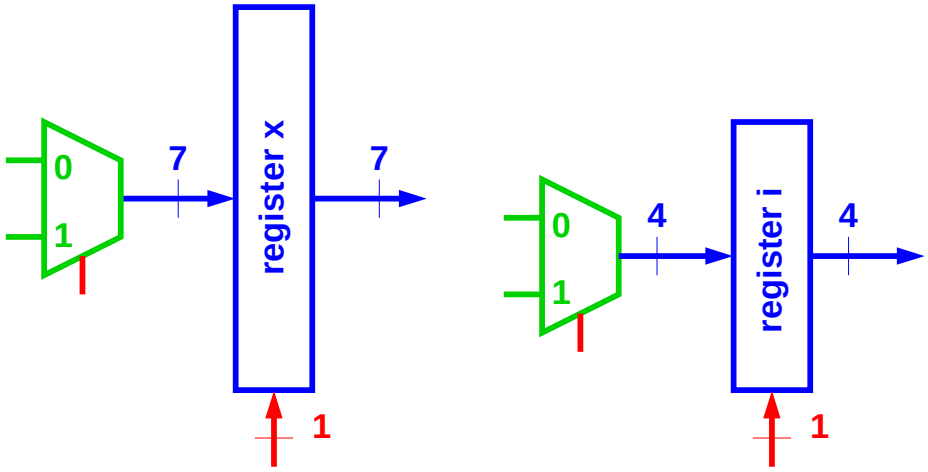
blue: state-holding elements that are implemented with posflops.

red: write control

`i` can be set in two different ways...

Data Dependencies



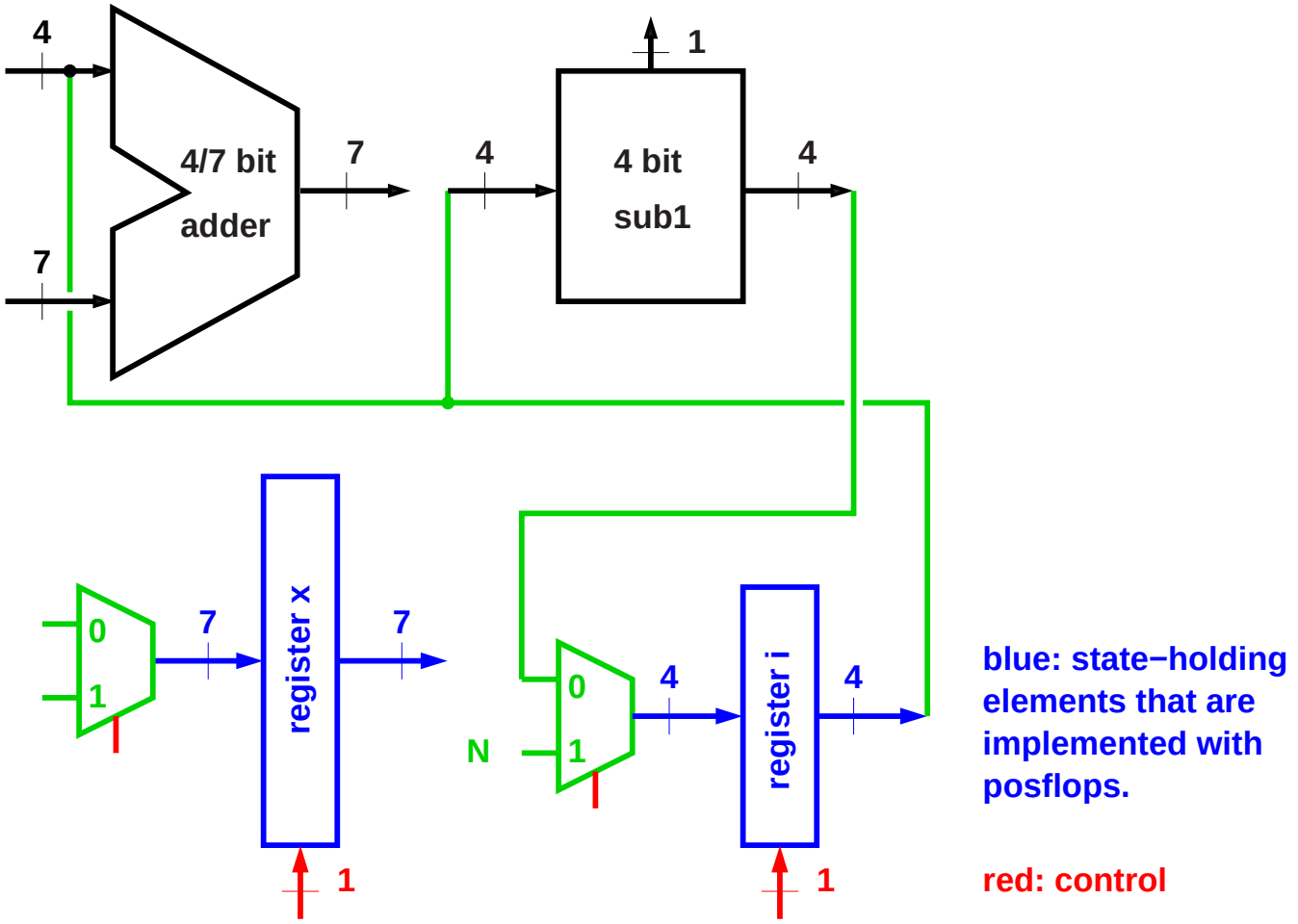


blue: state–holding elements that are implemented with posflops.

red: control

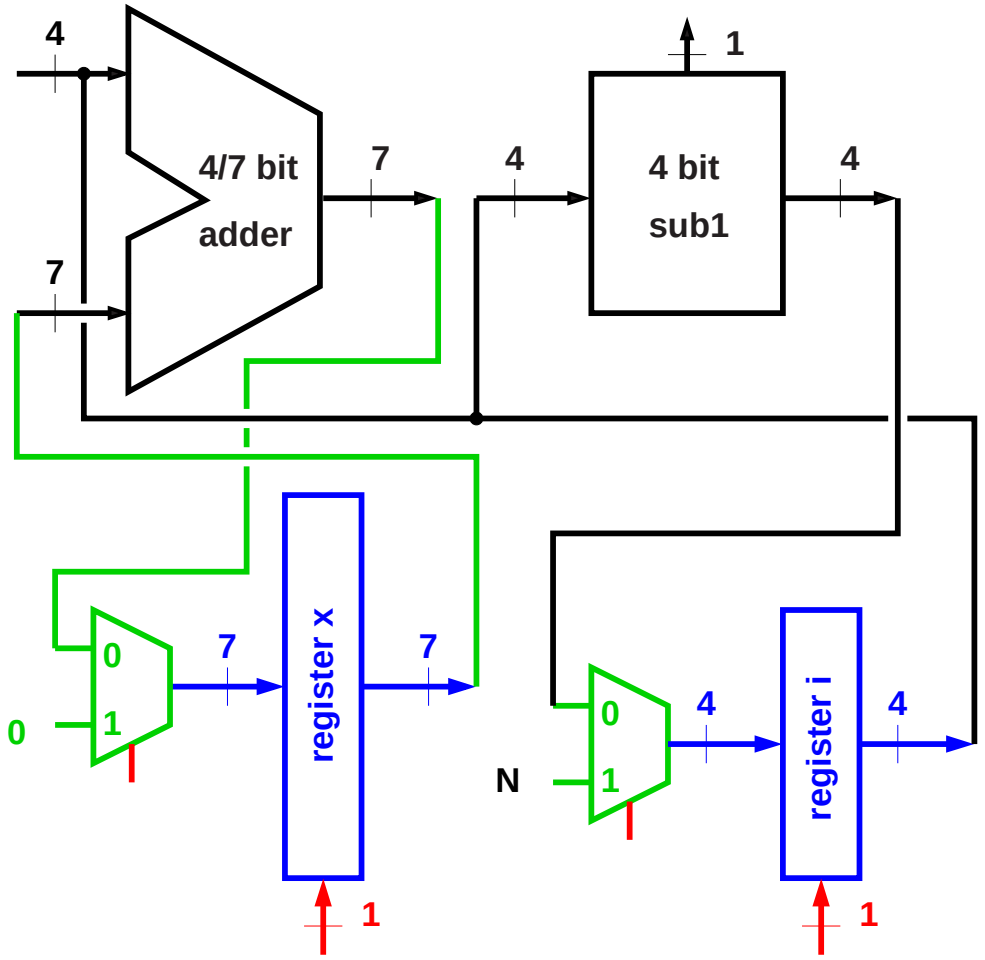
Use MUXes...

Data Dependencies



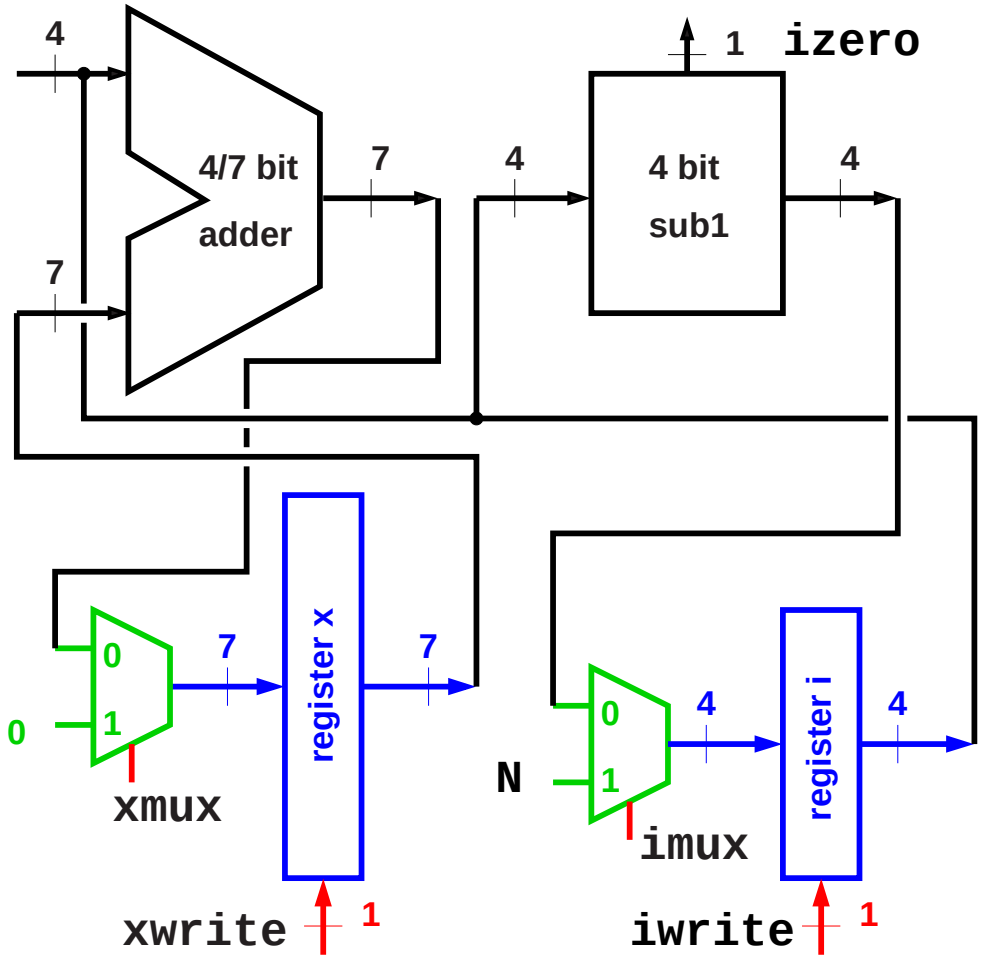
Connections for `i`.

Data Dependencies



Connections for x.

Data Dependencies



blue: state-holding elements that are implemented with posflops.

red: control

Finally: *control*

Control

```
initial:
   xdone <- 0;
   x <- 0;
   i <- N;
   goto loop;
loop:
   xdone <- 0;  x <- x + i;
   i <- i - 1;
   if (i == 0) goto done;
   else goto loop;
done:
   xdone <- 1;
   goto initial
```

Specify operations on data by using control signals!

CSL

Control

```
initial:
   xdone <- 0;
   xwrite <- 1; xmux <- 1;
   iwrite <- 1; imux <- 1;
   goto loop;
loop:
   xdone <- 0; xwrite <- 1; xmux <- 0;
   iwrite <- 1; imux <- 0;
   if (izero == 1) goto done;
   else goto loop;
done:
   xdone <- 1; xwrite <- 0; iwrite <- 0;
   goto initial
```

Specify operations on data by using control signals!

Control

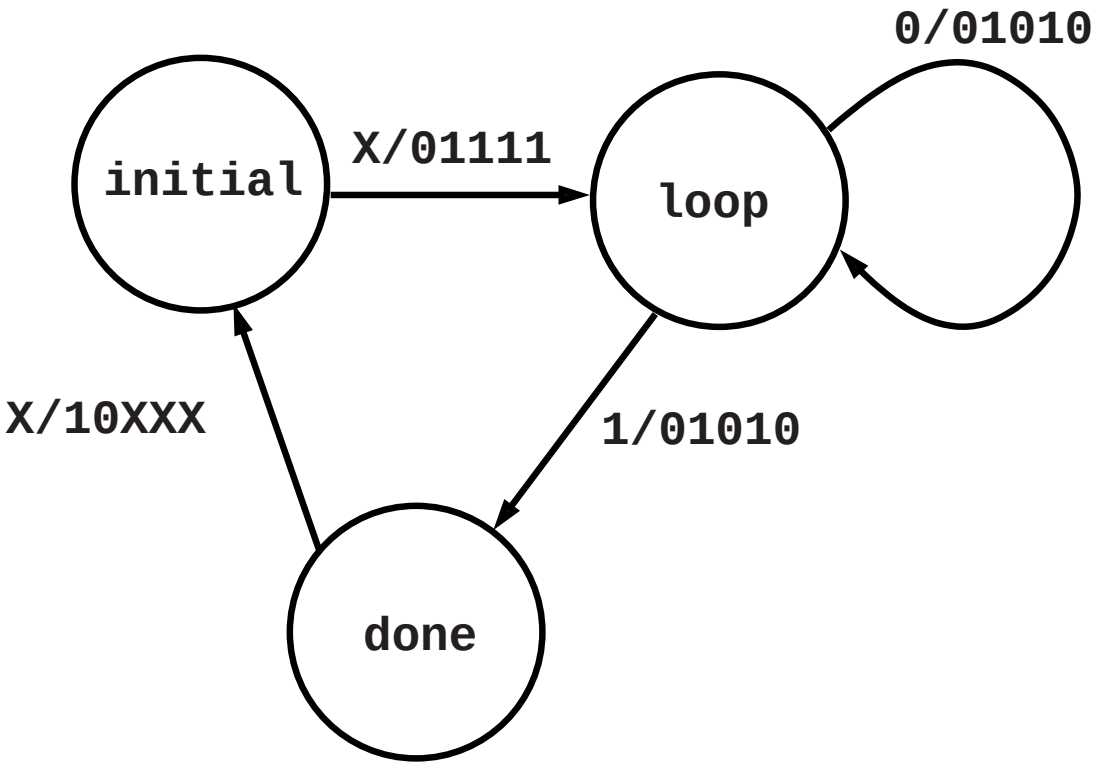
Important points:

- What about `xmux`, `imux` in the `done` state?
  $\Rightarrow$ `iwrite`, `xwrite` are `0`, so don't cares.
- Note: *each signal is set in each state!*
  $\Rightarrow$ we can generate them using combinational logic!
- If this is not the case, need a state-holding element to remember what the old value of the variable was...
  (sometimes referred to as an *implied flop/latch*)

CSL

Control

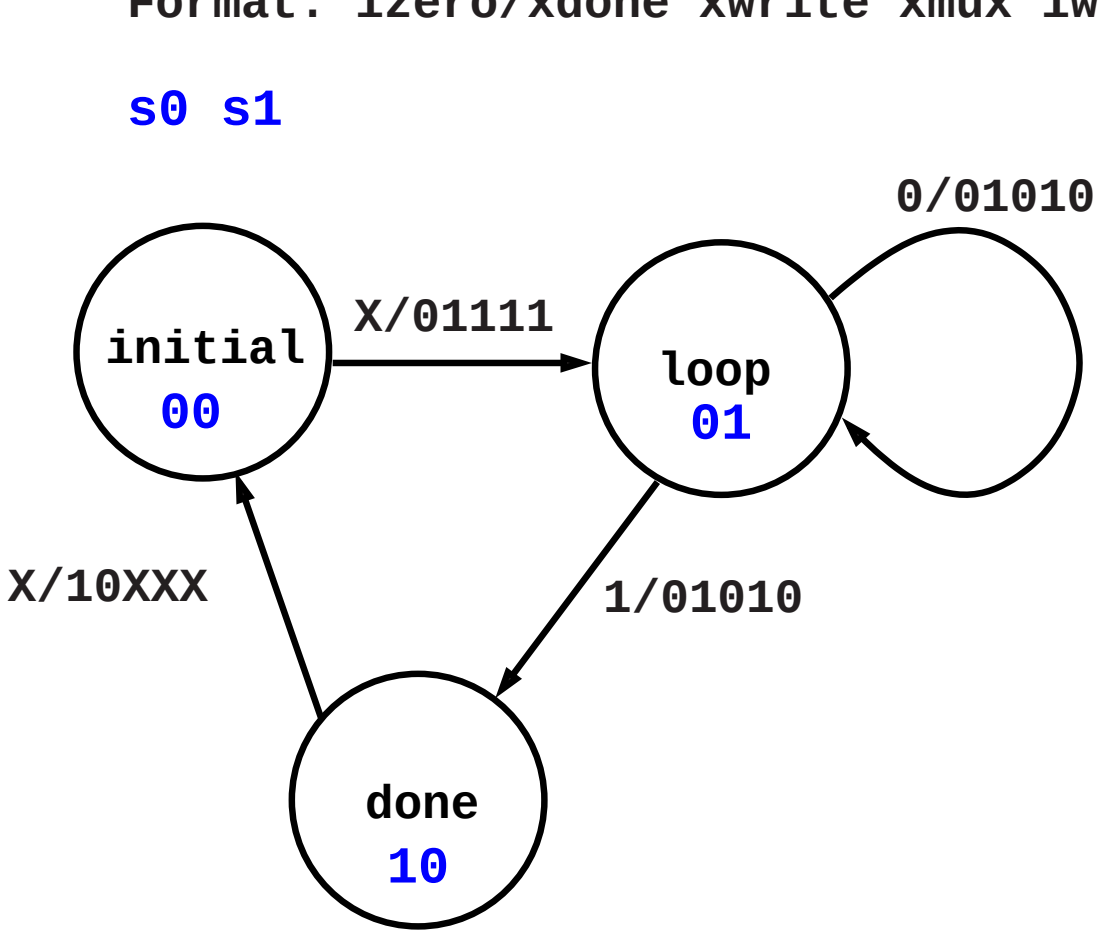
Format: izero/xdone xwrite xmux iwrite imux



Next step: state assignment/state table

Control

Format: izero/xdone xwrite xmux iwrite imux



Next: state tables, logic design

CSL

Control

After some logic synthesis...

- xdone = s0
- xwrite = $\overline{\text{s0}}$
- imux = $\overline{\text{s0}} \cdot \overline{\text{s1}}$
- xmux = $\overline{\text{s0}} \cdot \overline{\text{s1}}$
- iwrite = 1
- News0 = $\overline{\text{s0}} \cdot \text{s1} \cdot \text{izero} \cdot \overline{\text{Reset}}$
- News1 = $(\overline{\text{s0}} \cdot \overline{\text{s1}} + \text{s1} \cdot \overline{\text{izero}}) \cdot \overline{\text{Reset}}$

CSL