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 $31 \times 2^{32}$ 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
- Determine connections
- Determine control inputs to datapath blocks

... and then put everything together.

Cross-cutting issue: clocking strategy
  when do storage elements get updated?
Example: (unsigned integers)

\[
x = 0; \\
\text{for (i=1; i <= N; i++)} \\
x = x + i;
\]

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.
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!

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

Pick states, and determine operations in each state.

A systematic way: translate program using goto...

```c
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:

- $i$
  - $i \leftarrow N, i \leftarrow i - 1, i == 0$

- $x$
  - $x \leftarrow 0, x \leftarrow x + i$

- $xdone$
  - $xdone \leftarrow 0, xdone \leftarrow 1$

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

\[ \text{blue: state-holding elements that are implemented with posflops.} \]

\[ \text{red: write control} \]

\[ i \text{ can be set in two different ways...} \]
Use MUXes...
Data Dependencies

Connections for $i$.  

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

red: control
Data Dependencies

Connections for $x$. 

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

red: control
Data Dependencies

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!
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!
Important points:

• What about \texttt{umux}, \texttt{imux} in the done state?
  \implies\ \texttt{iwrite}, \texttt{xwrite} are 0, so don’t cares.

• \textbf{Note:} each signal is set in each state!
  \implies\ 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 \textit{implied flop/latch})
Control

Format: izero/xdone xwrite xmux iwrite imux

Next step: state assignment/state table
Control

Format: izero/xdone xwrite xmux iwrite imux

s0 s1

initial 00

loop 01

done 10

X/01111

0/01010

0/01010

Next: state tables, logic design
Control

After some logic synthesis...

- $x_{\text{done}} = s_0$
- $x_{\text{write}} = \overline{s_0}$
- $i_{\text{mux}} = s_0 \cdot s_1$
- $x_{\text{mux}} = \overline{s_0} \cdot s_1$
- $i_{\text{write}} = 1$
- $\text{News}_0 = \overline{s_0} \cdot s_1 \cdot i_{\text{zero}} \cdot \overline{\text{Reset}}$
- $\text{News}_1 = (\overline{s_0} \cdot \overline{s_1} + s_1 \cdot i_{\text{zero}}) \cdot \overline{\text{Reset}}$