Encoding Instructions

How do we represent instructions?
With bits, of course!

Instructions are encoded using a representation defined by the ISA.

The LC-3 ISA uses 16 bits to encode instructions.
That’s a big representation!
Shall we start listing instructions that we may want to include?

Let’s Use Your Skills!

But first...
(You know what’s coming, right?)
I need your help again.
I want to build an FSM to dispense soda at EOH.
Actually...
I want you to build an FSM, and call it, “Lumetta’s Soda Dispenser.”

My Plan for the Dispenser Operation

Here’s what I want:
1. Put in a quarter.
2. Pick one of four flavors: Cola, Lemon, Orange, or Grape.
3. Dispense soda for 10 clock cycles.

Let’s count states!
Count States for My Soda Dispenser

Before the user puts in a coin, we’ll have an OFF state.
Once they put in a quarter, your FSM will go to a HAVE_COIN state.
That’s two states, right?

Say the user picks Cola…
… how many states do we need to dispense Cola for 10 cycles before going back to OFF?
Another 10 states? Really? Ok.
And Orange? 10 again? I get it!
And so Grape needs … um… 10! Right.

How Many State ID Bits Do We Need?

Let’s make a table.
Help me add these...
Really? 42?
Nice.
So that’s …
OFF HAVE_COIN dispense Cola dispense Lemon dispense Orange dispense Grape
1 1 10 10 10 10
TOTAL 42

Here’s a Thought: Use 7 Bits Instead

May I make a suggestion?
Don’t try 6 bits at home.
It sounds painful.
Instead, use 7 bits:
◦ 1 bit: Do you have a coin?
◦ 2 bits: Which flavor?
◦ 4 bits: A counter for dispensing soda.
The logic will be (a lot) simpler.
Outline of Soda Dispenser Operation with a 7-Bit State ID

1. Putting in a quarter turns on the coin bit.
2. User picks a flavor when the coin bit is on.
3. Picking a flavor loads 10 into the 4-bit counter, which counts down.
4. For dispensing the soda, use a decoder:
   - flavor bits are decoded,
   - decoder enabled when counter is non-zero.

One decoder and a handful of gates, plus one extra flip-flop.

Why Does it Work Well?

Adding extra bits enables us to organize bits into groups with human meaning.
Mathematically, only relevant groups of bits affect particular outputs or next state bits.
Here “relevant” is based on the meanings we have defined for the bits!
So by making the representation easier for ourselves to understand, we also reduce the logic needed!
Don’t believe me? Try it with 6 bits.
Not impossible, but really not so fun.

Instruction Encodings Are Broken into Fields

What is the real point?
Most/all ISAs use such simplification to define instruction encodings (the representation used to encode instructions as bits).
Instruction bits are broken into fields.
One such field is an operation code, or opcode, which says what to do.
Other fields typically depend on the opcode, but they specify the operands for the operation defined by the opcode.

Let’s Look at Some Examples of LC-3 Instructions

<table>
<thead>
<tr>
<th>0 to 7</th>
<th>0 to 7</th>
<th>-32 to 31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0</td>
<td>DR</td>
<td>BaseR</td>
</tr>
<tr>
<td>LDR</td>
<td>6-bit 2’s complement offset</td>
<td></td>
</tr>
</tbody>
</table>

In words: Sign extend the offset6 field to 16 bits, then add the result to the contents of register BaseR to obtain a memory address. Read the bits at that memory address, and store them into register DR.
Here’s a Second Example: an ADD Instruction

<table>
<thead>
<tr>
<th>0 1 0 0</th>
<th>DR</th>
<th>SR1</th>
<th>0 0 0</th>
<th>SR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>opcode</td>
<td>destination register</td>
<td>source register 1</td>
<td>source register 2</td>
</tr>
</tbody>
</table>

In words: Add the contents of register SR1 to the contents of register SR2, and store the sum into register DR.

And One More, a Store to Memory

<table>
<thead>
<tr>
<th>0 1 1 1</th>
<th>SR</th>
<th>BaseR</th>
<th>-32 to 31</th>
<th>offset6</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR</td>
<td>opcode</td>
<td>source register</td>
<td>base register</td>
<td>6-bit 2's complement offset</td>
</tr>
</tbody>
</table>

M[BaseR + SEXT16(offset6)] ← SR

In words: Sign extend the offset6 field to 16 bits, then add the result to the contents of register BaseR to obtain a memory address. Store the bits from register SR to that memory address.

What Do You Need to Know?

Understand why engineers use meaningful groups of bits when defining representations.

Know the terminology that we just defined, including opcode and field.

Eventually, you should know the kinds of operations that instructions usually encode, and how such operations can be executed on a datapath (these topics are coming next).

What Don’t You Need to Know?

On the other hand, we really don’t care if you learn the LC-3 encoding, so long as you can understand and use a table explaining it (more experience will make you faster!).

You can find such a table in the back of Patt and Patel.

And another table on the Wiki under Resources / LC-3 handout. The one from the Wiki will be attached to both Midterm 3 and to the final exam.