G51CSA Handout: Addressing the System

Steven R. Bagley

Introduction

In lecture 23, we looked at how the CPU talks to the rest of the system over its *address* and *data* buses. In particular, we saw how all the chips in a system (CPU, RAM, ROM etc.) are connected to the same data and address bus and that the system uses standard digital logic to **decode** the address bus to enable (or *select*) the correct chip for a specific address. We built up the logic circuits using the *nand2tetris* simulator, but in this handout we will use written logic equations.

The important thing to remember is that the address bus is simply a set of electric signals that contain the binary number representing the address to be accessed. In these examples, we are using the 6502 CPU which has a 16-bit address bus (with signals, or *lines* as they are known, labelled `A15` to `A0` — with `A0` being the LSB). Although, the lines are carrying a number, we can treat them individually as just another logic value and combine them together using logic gates. Therefore, all we need to do is to find a logic circuit (or equation) that combines some or all of the address lines together such that the output is true whenever the address refers to a specific chip. However, it is also beneficial if we can keep the circuit as simple as possible (i.e. using the least amount of logic) so we sometimes allow things to be additional addresses. This is fine providing that everything is *uniquely* addressable.

Conventions

In this handout, we will use the following conventions to refer to AND, OR and NOT gates:

- AND — $A \cdot B$
- OR — $A + B$
- NOT — $\overline{A}$

1. BBC Micro

This system we looked at in the lecture was the venerable BBC Micro. The address space for the 'Beeb' is as follows (we'll consider I/O in a second):

Address Range (hex)	Contains	Select Signal
`0X0000—0x7FFF`	RAM	`RAMCS`
`0x8000—0xBFFF`	User ROM	`UROMCS`
`0xC000—0xFFFF`	OS ROM	`OSROMCS`

The trick is to spot what address lines stay the same, in the binary version of the address, for any valid address in a paritcular range. Often we can spot this by looking at a few choice addresses — the first, last, and those one before and after the last are good choices to start. In this example, for RAM we get:

```
          First RAM: 0000 0000 0000 0000
           Last RAM: 0111 1111 1111 1111
      First not RAM: 1000 0000 0000 0000
```

All the valid RAM addresses have `A15`, the MSB (most significant bit), set to zero and all other addresses have it set to one. So for the `RAMCS` signal to select the RAM chip, we just need to invert `A15` using a NOT gate, e.g.

$$\texttt{RAMCS}=\overline{A15}$$

It also follows then that both **UROMCS** and **OSROMCS** *must* both require **A15** to be set to one.

We can now start to work out a logic equation for **UROMCS** in the same manner. However, we can make things easier for ourselves by spotting that it's actually only the top nibble (4-bits are called a nibble, since it's smaller than a byte…) that matters. This isn't always the case, sometimes it's the top byte, or even the whole address bus. In this case, it is nice since each hex digit represents one nibble.

Therefore, for **UROMCS**, we need to develop a circuit that is true when the top nibble conains either **0x8**, **0x9**, **0xA**, or **0xB** and is false, when it contains any other value. The easiest starting point for this is to look at the binary versions:

```
1000
1001
1010
1011
```

We have two options here, we could just match each of the binary patterns that are valid and or them together, like this:

$$\texttt{UROMCS}=A15\cdot\overline{A14}\cdot\overline{A13}\cdot\overline{A12}\ +\ A15\cdot\overline{A14}\cdot\overline{A13}\cdot A12\ +\ A15\cdot\overline{A14}\cdot A13\cdot\overline{A12}\ +\ A15\cdot\overline{A14}\cdot A13\cdot A12$$

And then use the rules of boolean algebra to simplify the equation (in the same way, we can simplify mathematics). Or we can spot, that each address starts **10xx**. Either way, we can reduce the expression to the much simpler:

$$\texttt{UROMCS}=A15\cdot\overline{A14}$$

Exactly the same procedure, can be applied for **OSROMCS** and you'll find (the steps are left as an exercise to the reader) that the result is:

$$\texttt{OSROMCS}=A15\cdot A14$$

1.1. I/O

As it stands, there's no space left to map I/O devices into the address space. The BBC Micro, actually maps three I/O ranges (known, bizarrely, as **FRED**, **JIM** and **SHEILA**!) into the following addresses:

Address Range (hex)	Contains	Select Signal
0XFC00—0xFCFF	FRED	FREDCS
0xFD00—0xFDFF	JIM	JIMCS
0xFE00—0xFEFF	SHEIA	SHEILACS

Developing select signals for each range is straight-forward, although in this case we need to consider 8-bits worth of the address bus. So for **FREDCS**, we need to match **0xFC** on the top eight address lines, like this:

$$\texttt{FREDCS}=A15\cdot A14\cdot A13\cdot A12\cdot A11\cdot A10\cdot\overline{A9}\cdot\overline{A8}$$

For **JIMCS**:

$$\texttt{JIMCS}=A15\cdot A14\cdot A13\cdot A12\cdot A11\cdot A10\cdot\overline{A9}\cdot A8$$

And **SHEILACS**:

$$\texttt{SHEILACS}=A15\cdot A14\cdot A13\cdot A12\cdot A11\cdot A10\cdot A9\cdot\overline{A8}$$

But this leaves, a problem some addresses are now matched by both **OSROMCS** and one of the I/O lines—since **A15** and **A14** will be true for any I/O address in FRED, JIM or SHEILA. The solution is to redefine **OSROMCS** so that it is not true when it is an I/O area as well. We can do

this by anding it with the inverted I/O select line:

$$\texttt{OSROMCS=}A15 \cdot A14 \cdot \overline{FREDCS} \cdot \overline{JIMCS} \cdot \overline{SHEILACS}$$

Alternatively, this can be written as:

$$\texttt{OSROMCS=}A15 \cdot A14 \cdot \overline{(FREDCS + JIMCS + SHEILACS)}$$

Both are equivalent due to De Morgan's rule, which one is implemented will often depend on which logic gates are free in the circuit. In either case, the ROM chip will only be selected if the address is in the ROM range, *and* is not in an I/O range.

2. Atari VCS

The Atari VCS was an early games console that also used a 6502 CPU. It's address space was mapped as follows:

Address Range (hex)	Contains	Select Signal
0X0000—0x002C	VIDEO I/O	VIDCS
0x0100—0x01FF	RAM	RAMCS
0x2010—0x2012	I/O	IOCS
0xF000—0xFFFF	Cartridge ROM	ROMCS

This address space is much more spartan, but the same rules as before can be applied. However, we can vastly reduce the logic required if allow 'shadows' or 'mirrror' to be present. This is fine as long as everything is uniquely decodeable. It is (again) left as an exercise to the reader to work out the exact derivations, but it is possible to decode the address space using the following equations:

$$\texttt{VIDCS=}\overline{A15} \cdot \overline{A14} \cdot \overline{A13} \cdot \overline{A12} \cdot \overline{A11} \cdot \overline{A10} \cdot \overline{A9} \cdot \overline{A8}$$
$$\texttt{RAMCS=}\overline{A15} \cdot \overline{A14} \cdot \overline{A13} \cdot \overline{A12} \cdot \overline{A11} \cdot \overline{A10} \cdot \overline{A9} \cdot A8$$
$$\texttt{IOCS=}\overline{A15} \cdot \overline{A14} \cdot A13 \cdot \overline{A12}$$
$$\texttt{ROMCS=}A15 \cdot A14 \cdot A13 \cdot A12$$

However, it is possible to use the rules of Boolean algebra to simplify this even further by factoring out common sub-equations and reusing them.