Addressing the System

Steven R. Bagley

Introduction

- Looked at what is inside a computer
- Digital Logic
- ARM Assembly
- Today we come full circle…
- How do we put all the bits together to build a full computer?

Or…

How do you build computer-controlled Christmas tree lights?

Tree Lights

- Wanted to build a set of Raspberry PI controlled Christmas tree lights…
- Easily done, just need to write to address `0x20200000`
- Can easily write C code to do this
- Must declare the pointer as `volatile`

volatile tells it that the contents can change/be read outside of the C program

Addressing

- How does the RPi know that `0x20200000` is the GPIO pins, and not RAM?
- How does any computer know which addresses map to RAM, ROM or I/O?
- Down to our old friend digital logic…

6502

- Going to consider things in terms of the 6502 CPU, but equally applicable to others
- 8-bit data bus
- 16-bit address bus
- Gives a 64K *address space*

Although modern CPUs use very different ways of talking to the outside world
SoCs (such as the RPi) tend to do this all internally

Address Space

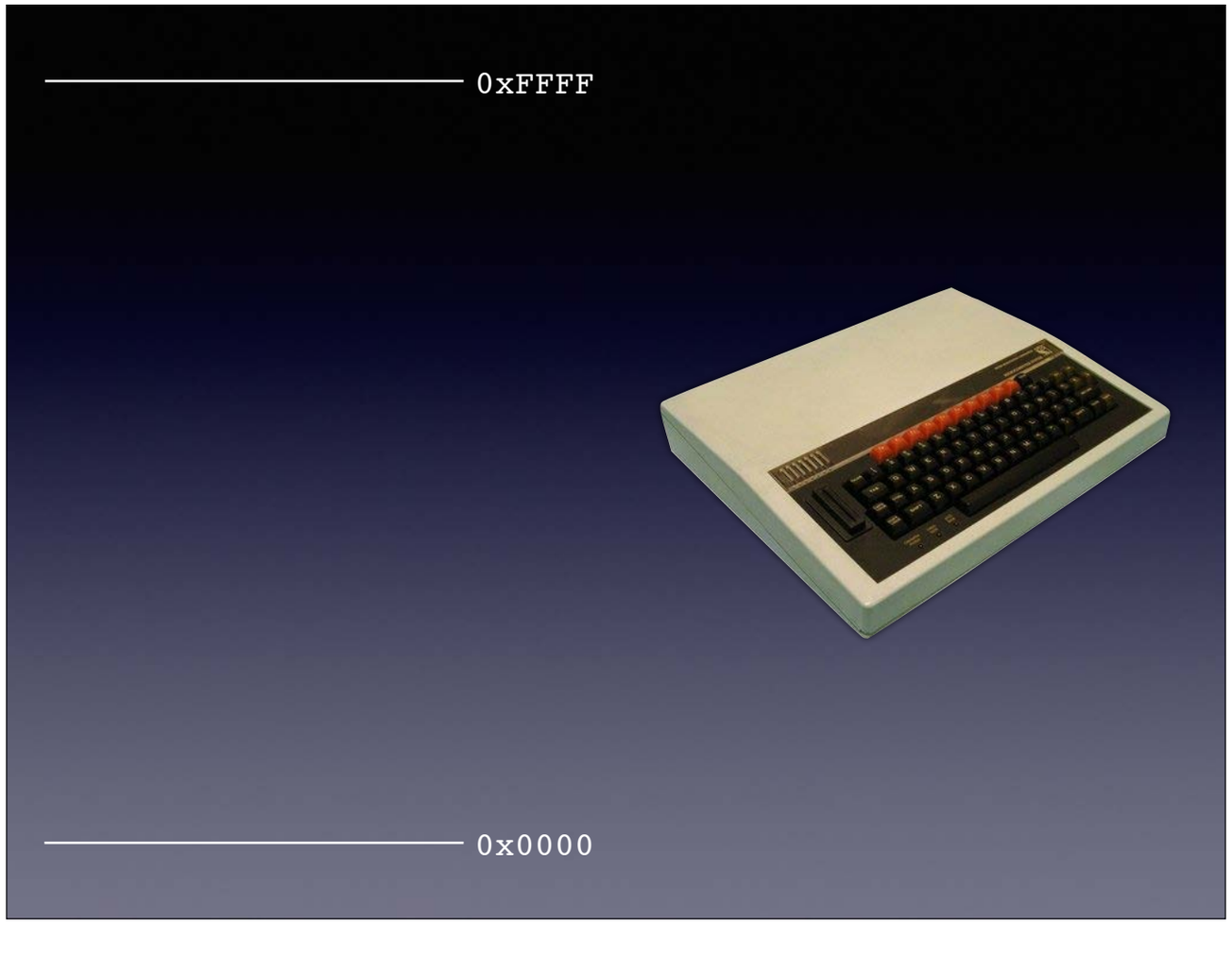
- When designing the system up we divide this address space up as we please between:
  - RAM
  - ROM
  - I/O
- Don’t necessarily have to use all of it…

Address Space

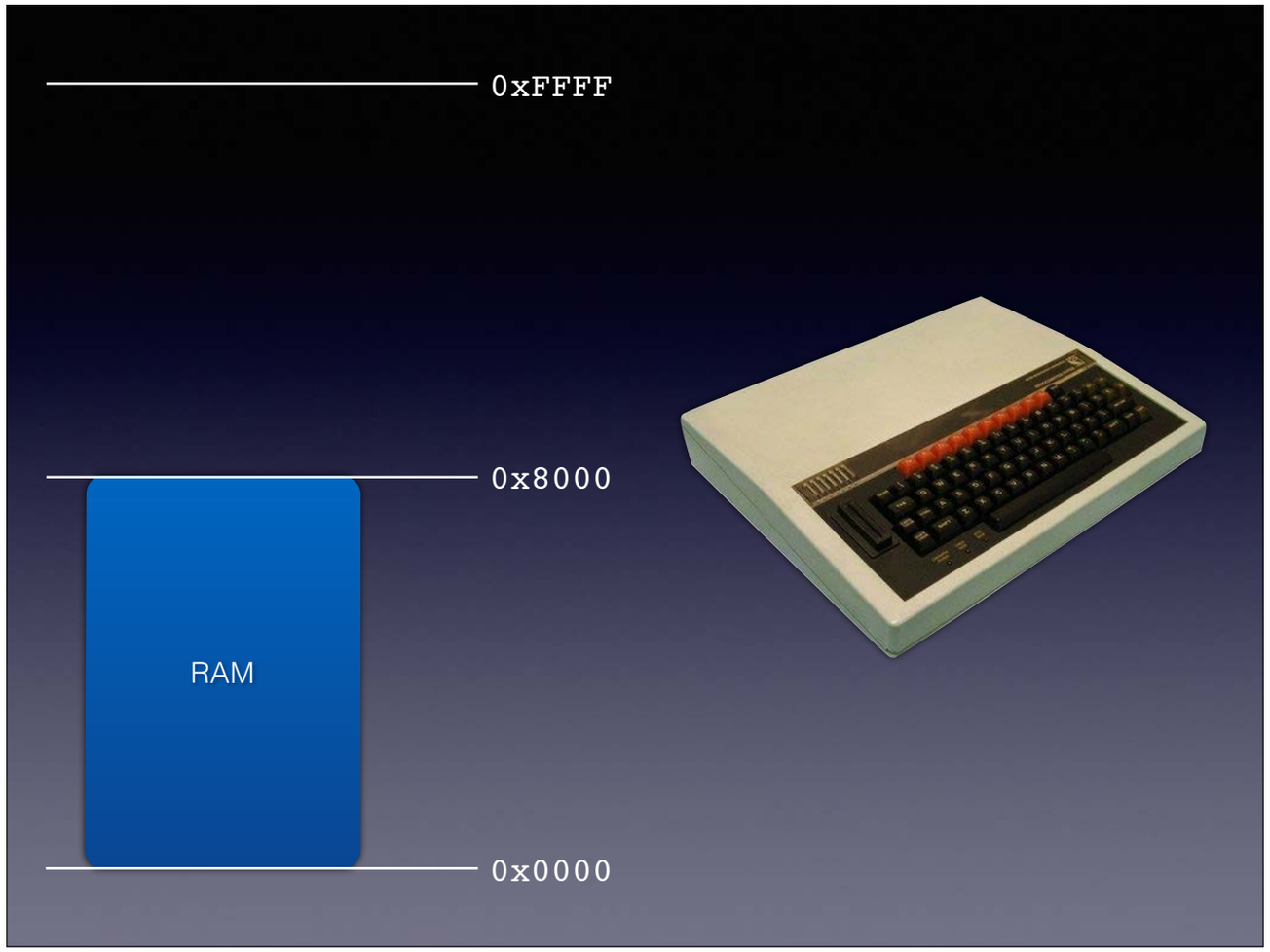
- Can effectively partition it as we want
- Often done to make it easy to decode
  - Using as little logic as possible
  - Also peculiarities of CPU might have an effect

Cost, gate delay

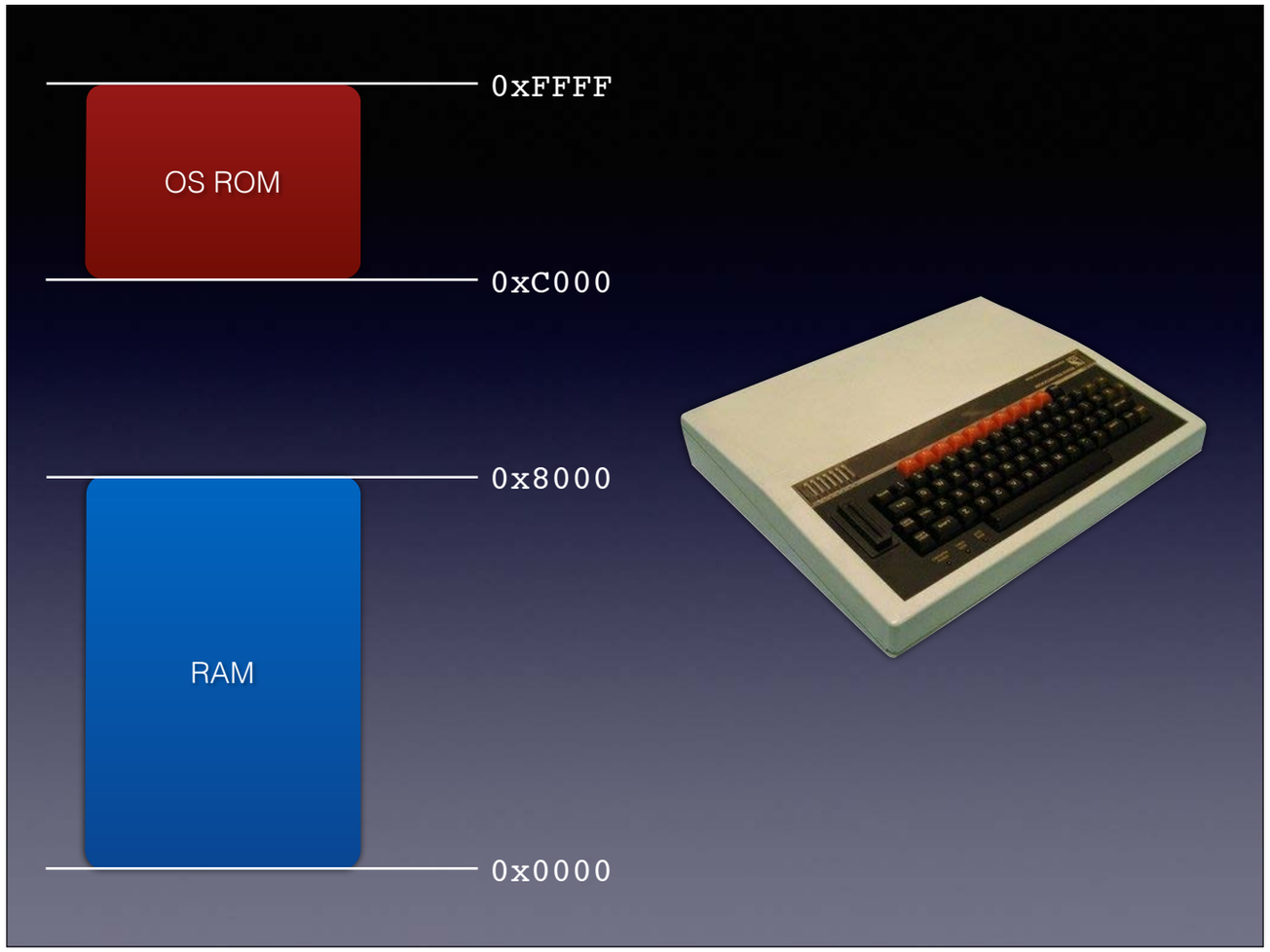
CPU requires certain addresses to contain certain things



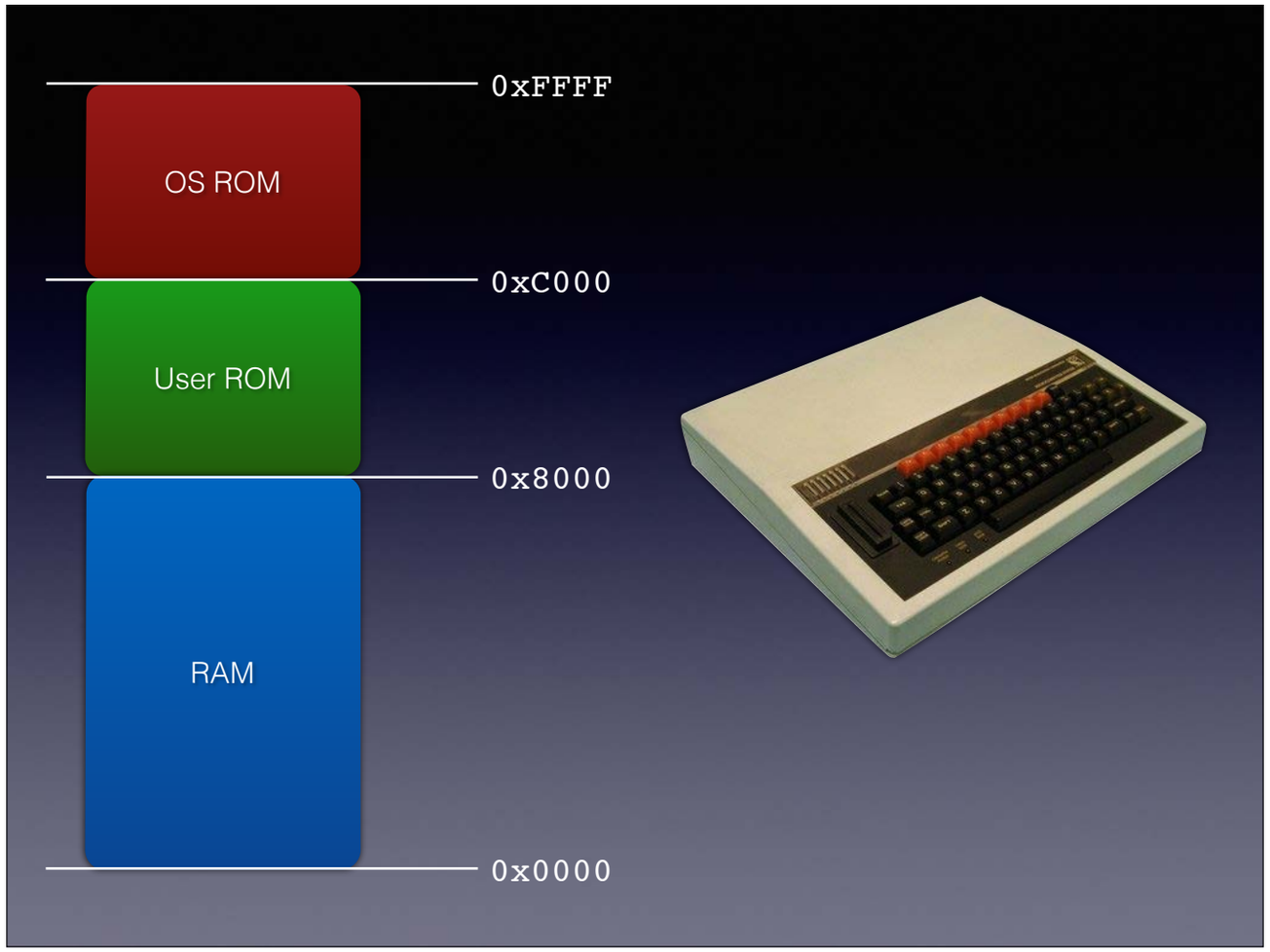
BBC Micro memory space…
6502 has very quick access to RAM at 0x0000-0x00FF and expects stack at 0x0100-0x01ff so will put 32K RAM at 0x0000-0x7fff
8K OS ROM to end at 0xFFFF so that reset vector at 0xFFFC is taken care of
User ROM in the gap



BBC Micro memory space…
6502 has very quick access to RAM at 0x0000-0x00FF and expects stack at 0x0100-0x01ff so will put 32K RAM at 0x0000-0x7fff
8K OS ROM to end at 0xFFFF so that reset vector at 0xFFFC is taken care of
User ROM in the gap



BBC Micro memory space…
6502 has very quick access to RAM at 0x0000-0x00FF and expects stack at 0x0100-0x01ff so will put 32K RAM at 0x0000-0x7fff
8K OS ROM to end at 0xFFFF so that reset vector at 0xFFFC is taken care of
User ROM in the gap

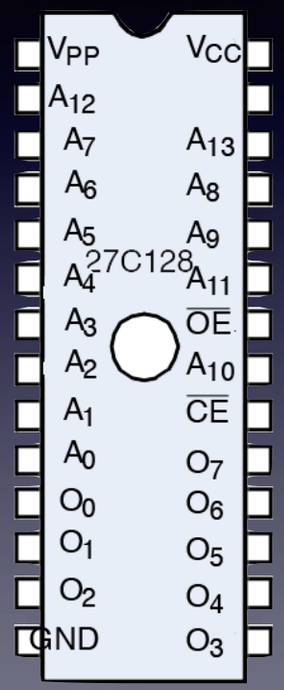


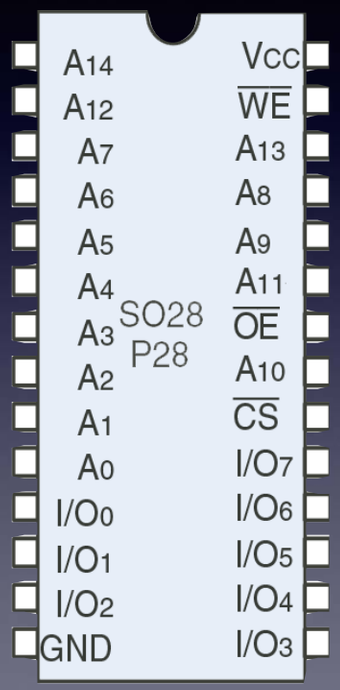
BBC Micro memory space…
6502 has very quick access to RAM at 0x0000-0x00FF and expects stack at 0x0100-0x01ff so will put 32K RAM at 0x0000-0x7fff
8K OS ROM to end at 0xFFFF so that reset vector at 0xFFFC is taken care of
User ROM in the gap

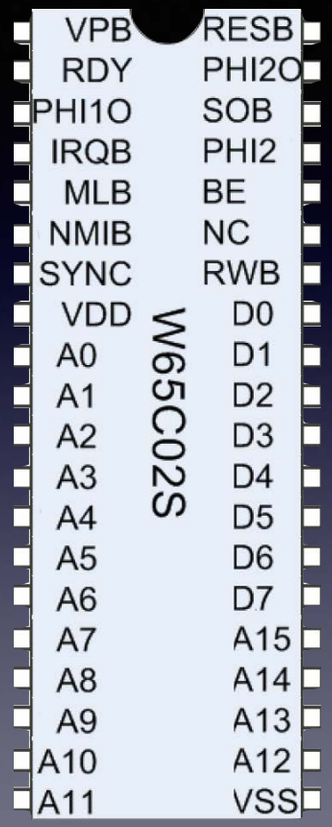
Alignment

- ROM, RAM chips all have a series address lines
- So it makes sense to ensure things are aligned to powers of 2 address
- Means we can just connect the common address lines straight to the chip
- No need to add offsets or anything…
- Means we only have to consider a subset of address lines to decode which chip

Chips

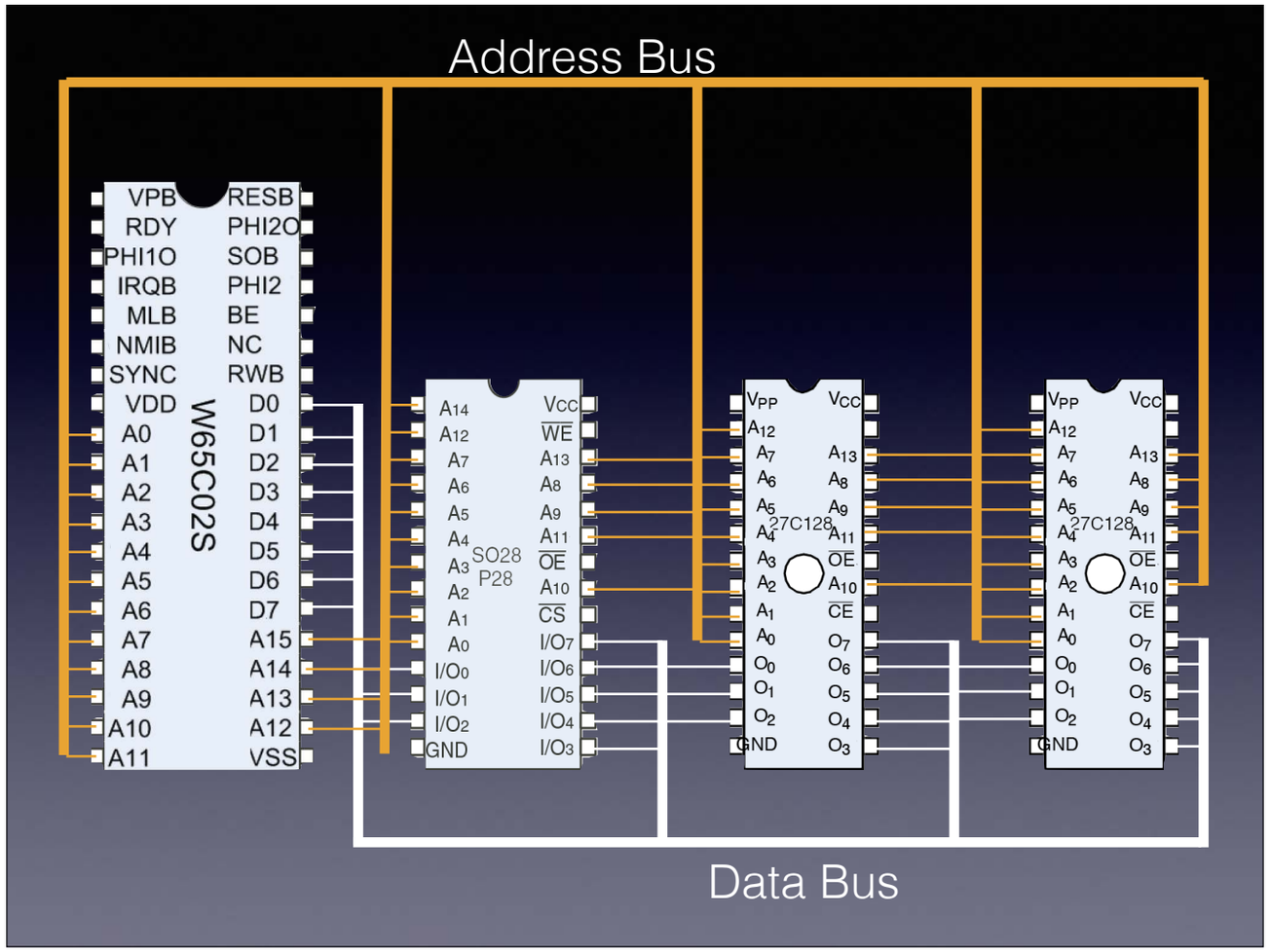

VPP
A12
A7
A6
A5
A4
A3
A2
A1
A0
O0
O1
O2
GND
VCC
A13
A8
A9
A11
OE
A10
CE
O7
O6
O5
O4
O3
27C128


A14
A12
A7
A6
A5
A4
A3
A2
A1
A0
I/O0
I/O1
I/O2
GND
VCC
WE
A13
A8
A9
A11
OE
A10
CS
I/O7
I/O6
I/O5
I/O4
I/O3
SO28
P28


VPB
RDY
PHI1O
IRQB
MLB
NMIB
SYNC
VDD
A0
A1
A2
A3
A4
A5
A6
A7
A8
A9
A10
A11
RESB
PHI2O
SOB
PHI2
BE
NC
RWB
D0
D1
D2
D3
D4
D5
D6
D7
A15
A14
A13
A12
VSS
W65C02S

Chip Select

- How do we connect everything up?
- How do we make the CPU talk to the right chip at the right time?


Address Bus
VPB
RDY
PHI1O
IRQB
MLB
NMIB
SYNC
VDD
A0
A1
A2
A3
A4
A5
A6
A7
A8
A9
A10
A11
RESB
PHI2O
SOB
PHI2
BE
NC
RWB
D0
D1
D2
D3
D4
D5
D6
D7
A15
A14
A13
A12
VSS
W65C02S
SO28
P28
A14
A12
A7
A6
A5
A4
A3
A2
A1
A0
I/O0
I/O1
I/O2
GND
Vcc
WE
A13
A8
A9
A11
OE
A10
CS
I/O7
I/O6
I/O5
I/O4
I/O3
27C128
VPP
Vcc
A12
A7
A6
A5
A4
A3
A2
A1
A0
O0
O1
O2
GND
A13
A8
A9
A11
OE
A10
CE
O7
O6
O5
O4
O3
Data Bus

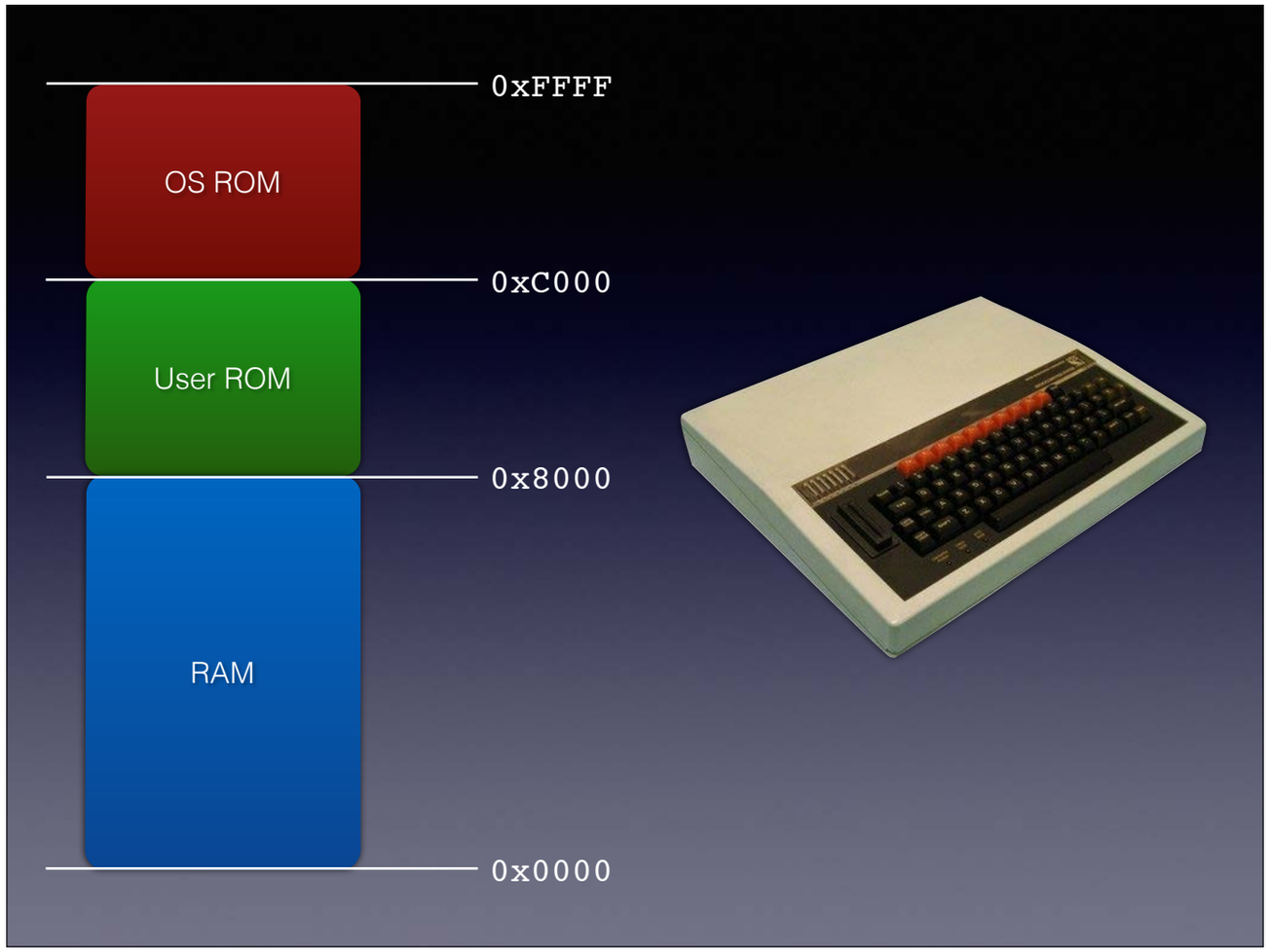
Chip Select

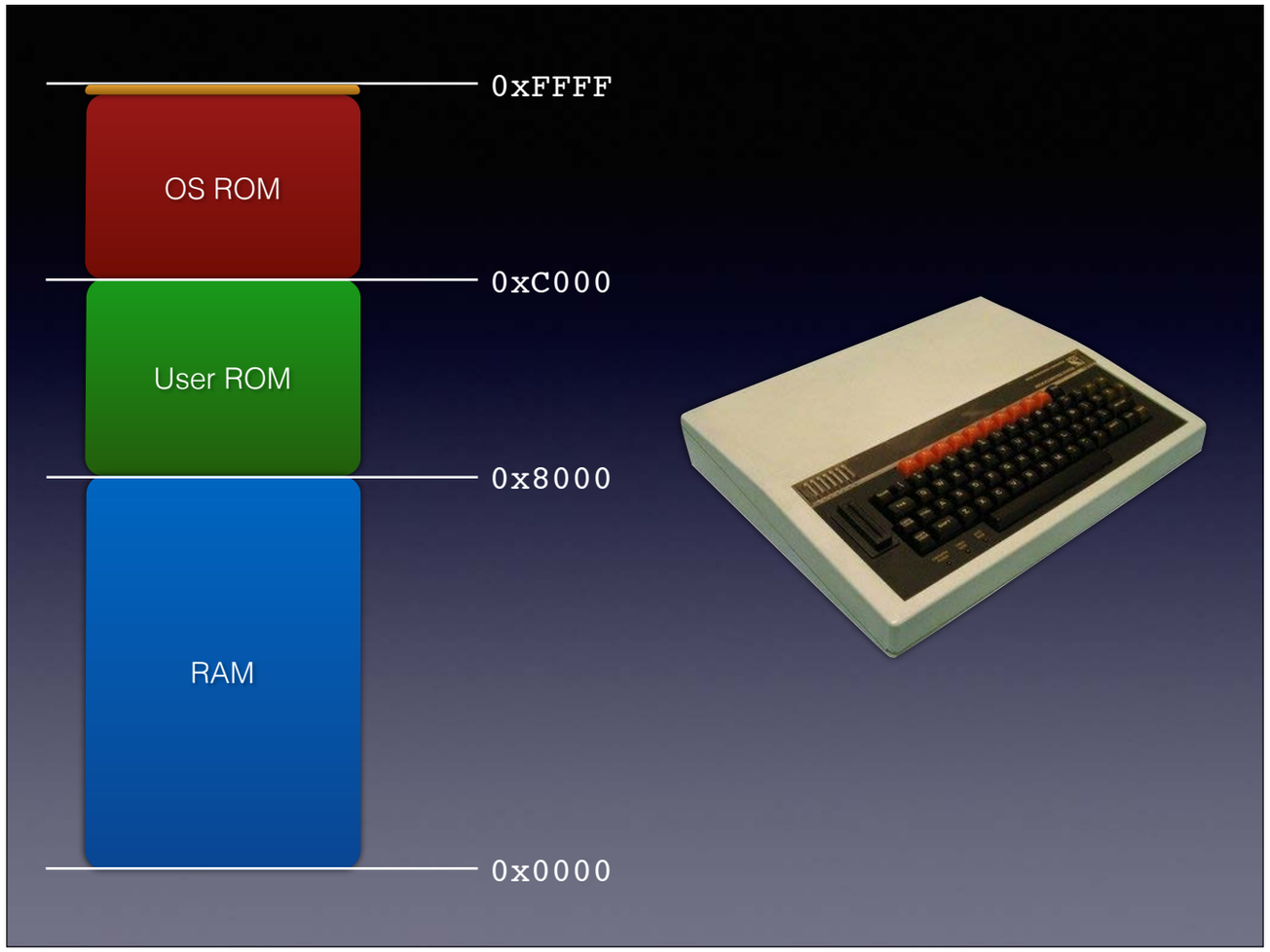
- Each chip has a *chip select* pin (`cs`) which selects whether the chip is talking to the data bus or not
- When it's not talking it is in a *high-impedance* state
  - Effectively equivalent to if it wasn't present
- By driving the `cs` line with logic we can select the right chip

Chip Select

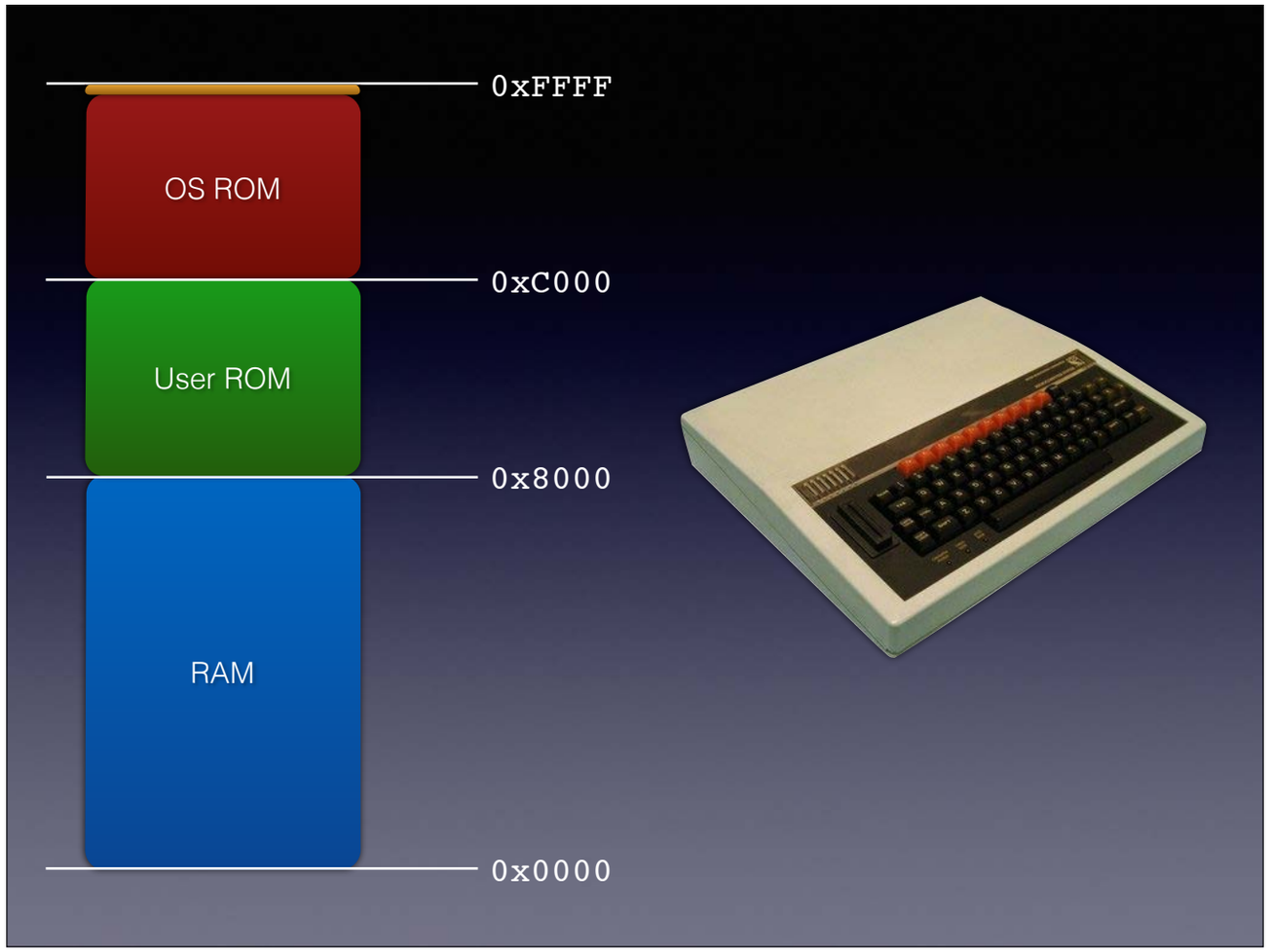
- For each chip, we need to generate a logic signal based on the address bus
- That is true, only if the address is in the right range
- Need to consider the binary value of the addresses and create logic equations based on them

Demo on paper


0xFFFF
OS ROM
0xC000
User ROM
0x8000
RAM
0x0000



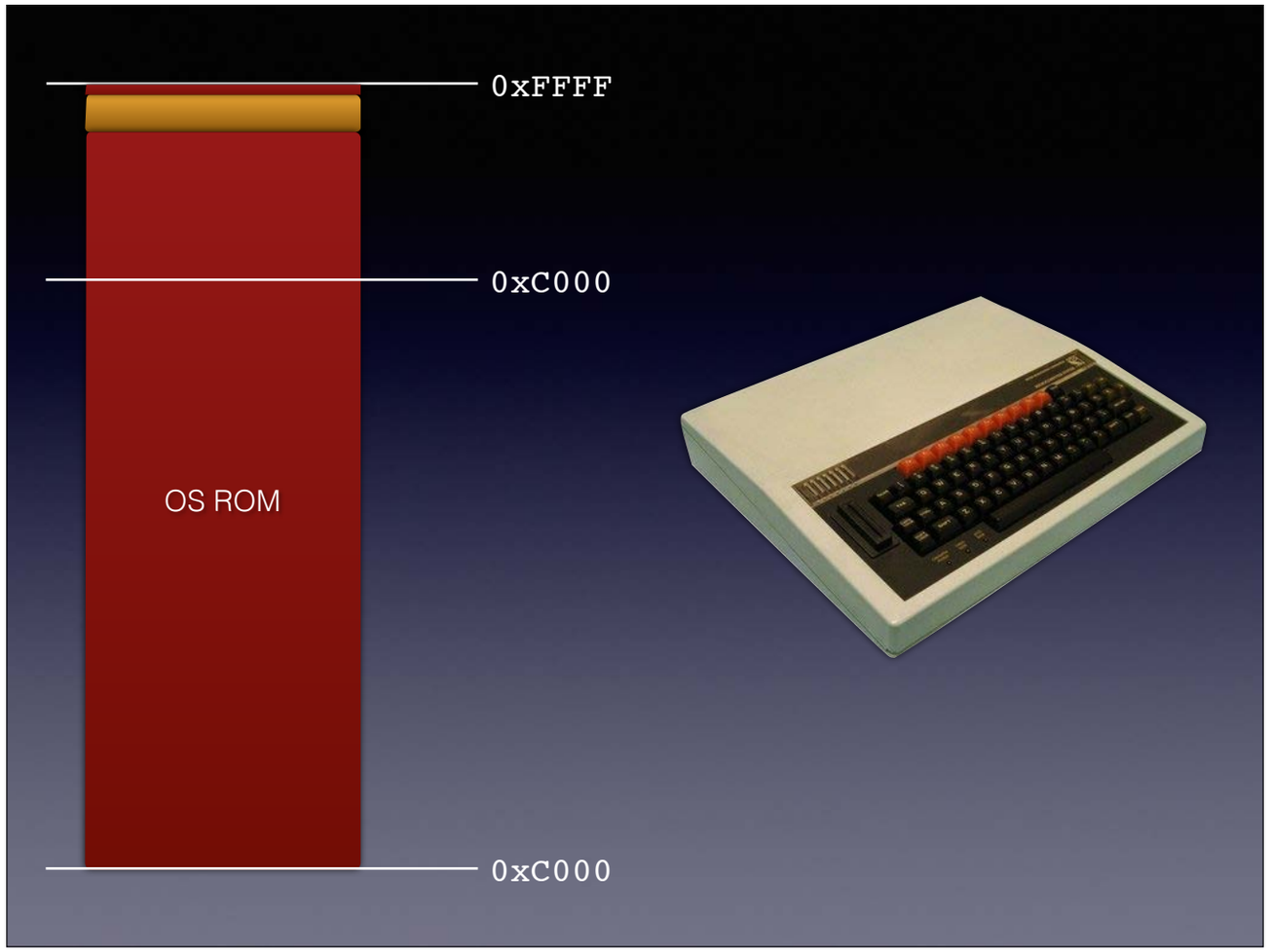
What about IO?


0xFFFF
OS ROM
0xC000
User ROM
0x8000
RAM
0x0000

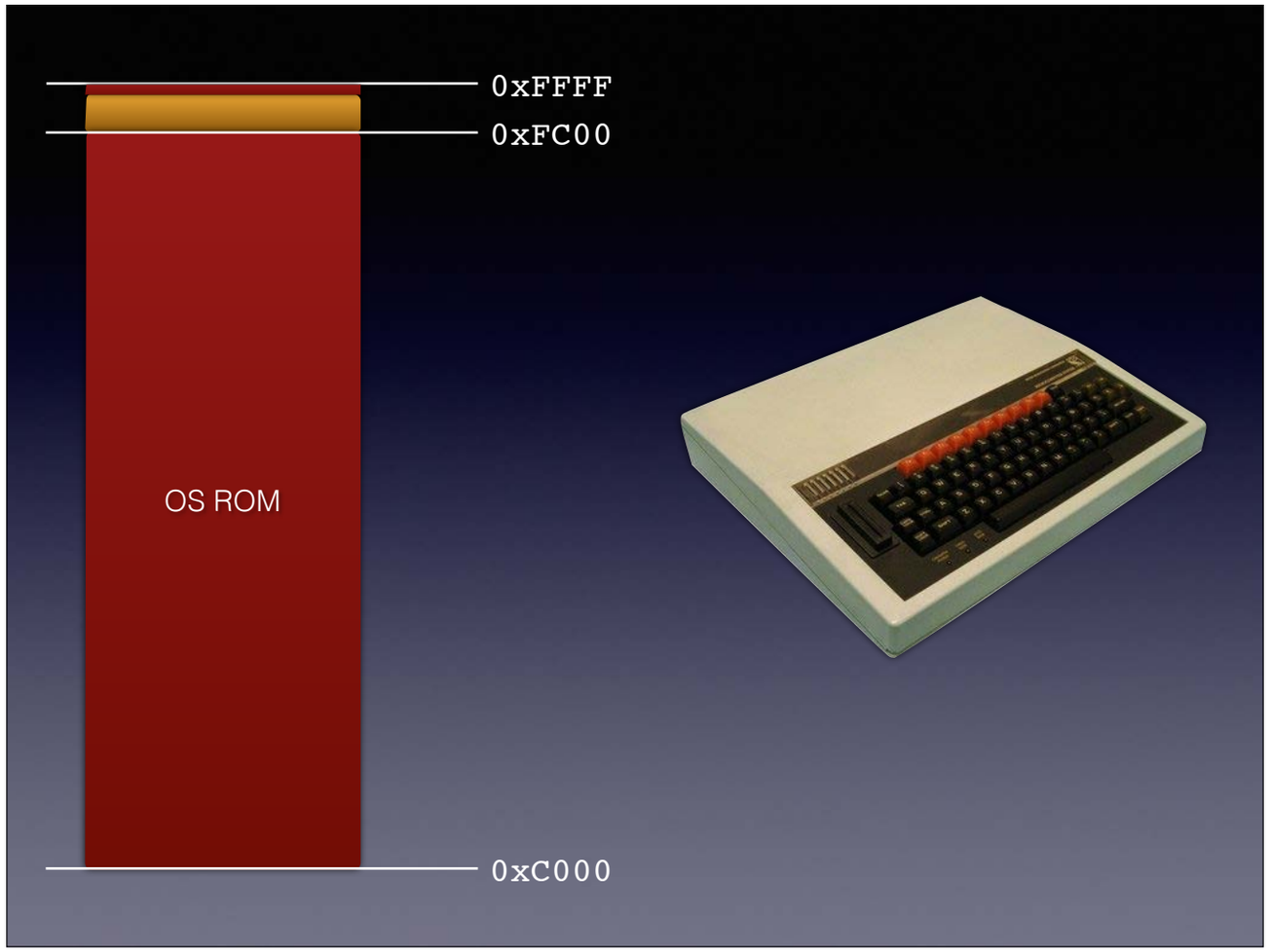
0xFFFF

OS ROM

0xC000



Treated in the same way — decode the relevant addresses and feed to the relevant chips



Treated in the same way — decode the relevant addresses and feed to the relevant chips

Address Space

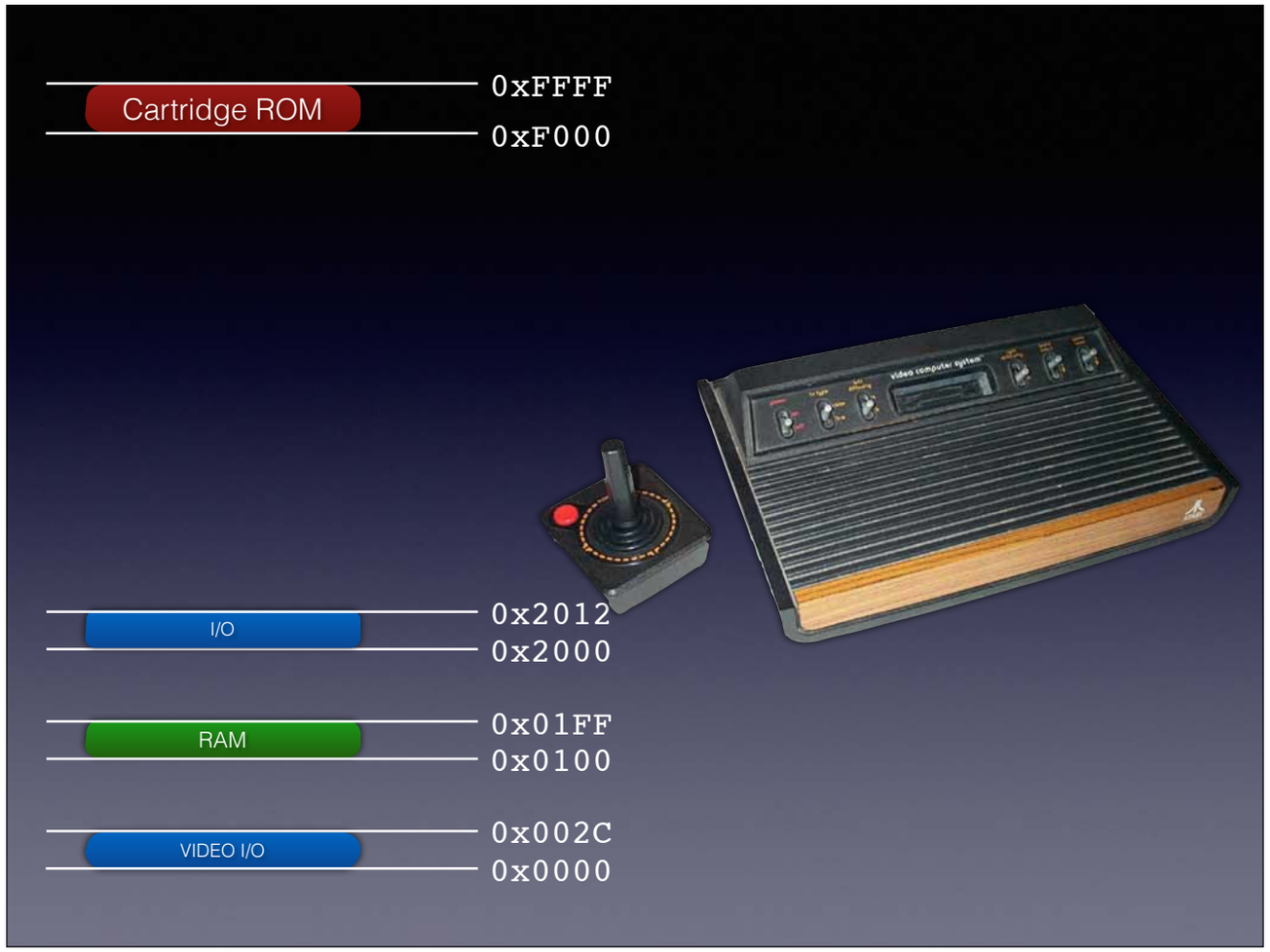
- Don’t need to fill the address space — can have unused gaps
- Also don’t have to *completely* decode the addresses
- Only have to *uniquely* decode each part…

Shadows

- Leads to shadows or mirrors of RAM/ROM in the address space
- This is fine but as a programmer it's not a good idea to use them
- Later hardware revisions may change the decoding (so your program would stop working)

Simple Logic

- Want to build the simplest, cheapest logic
- Using the least number of gates
  - Reduces cost
  - Reduces gate delay
- Optimize using boolean algebra rules
- Only decode as much as you have to


0xFFFF
Cartridge ROM
0xF000
0x2012
I/O
0x2000
0x01FF
RAM
0x0100
0x002C
VIDEO I/O
0x0000