5.10 CPU interfacing to external memory and IO

Memory technologies

The processor interface

The address map

Memory and IO interfacing

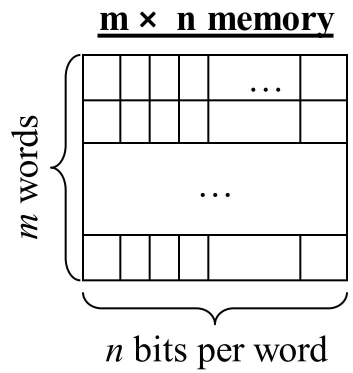
Memory (address) units recommended by IEEE 1541

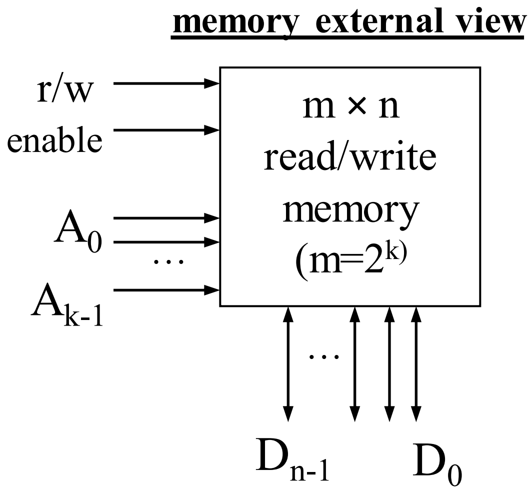
- Units for quantities used in digital electronics and computing:
  - *bit* (symbol 'b'), a binary digit;
  - *byte* (symbol 'B'), a set of adjacent bits (usually, but not necessarily, eight) operated on as a group; (In EE302 we will always use byte to mean 8 bits)
  - *octet* (symbol 'o'), a group of exactly eight bits;
- Prefixes used to indicate binary multiples of the aforesaid units:
  - *kibi* (symbol 'Ki'), 2^{10} = 1024;
    - E.g. 16 KiB = 16*1024=16384 Bytes
  - *mebi* (symbol 'Mi'), 2^{20} = 1024^2 =1048576;
  - *gibi* (symbol 'Gi'), 2^{30} = 1024^3 = 1073741824;
  - *tebi* (symbol 'Ti'), 2^{40} = 1024^4 = 1099511627776;
  - *pebi* (symbol 'Pi'), 2^{50} = 1024^5 = 1125899906842624;
  - *exbi* (symbol 'Ei'), 2^{60} = 1024^6 = 1152921504606846976;

Memory

- □ Stores data in an m x n matrix of memory cells
  - □ *m* **words** (locations), addressed using ***k***=ceiling(log$_2$(*m*)) address lines
  - □ Each address line identifies a word in memory
  - □ *n* bits per word, i.e. using *n* data lines
  - □ When a single word is addressed, all data lines are read/written in parallel
- □ Examples
  - □ 4096x8 memory implies
    - 4096 words of memory, each 8 bits wide => 8 data lines
    - 12 address lines (12=ceiling(log$_2$(4096)) and 2^{12} = 4096)
  - □ E.g. 16K x 8 memory
    - Interpret 16K as 16Ki in this context, i.e. 16384 words, each word is 8 bits wide
    - 14 address lines (log$_2$(16384)), 8 data lines

0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1





Address arithmetic 1

k address lines => m locations, where m=2^k

- □ To convert to KiB:
  KiB= m / 1024
- □ To convert to MiB:
  MiB = m / (1024 * 1024)
- □ Examples
  - □ 4 address lines => 2^4 = 16 locations
  - □ 11 address lines => 2^11 = 2048 locations = 2 KiB (if each location is 1 byte wide)

For convenience: we will only work with memory devices that use 8 bit words in this course. Therefore a memory word will always be 1 byte (or, strictly speaking, 1 octet).

m locations => k address lines, where k = ceiling(log$_2$(m))

Note: ceiling(x) is smallest integer >= x

Note: $log_2(m) = log_{10}(m) /log_{10}(2)$

- □ To obtain m if given KiB:
  m = KiB * 1024
- □ To obtain m if given MiB:
  m = MiB * 1024 *1024
- □ Examples:
  - □ 32 KiB = 32*1024 = 32768 locations
    => ceiling(log2(32768)) = ceiling(15) = 15 address lines
  - □ 32000 locations
    => ceiling(log2(32000)) = ceiling(14.97) = 15 address lines

Self test questions

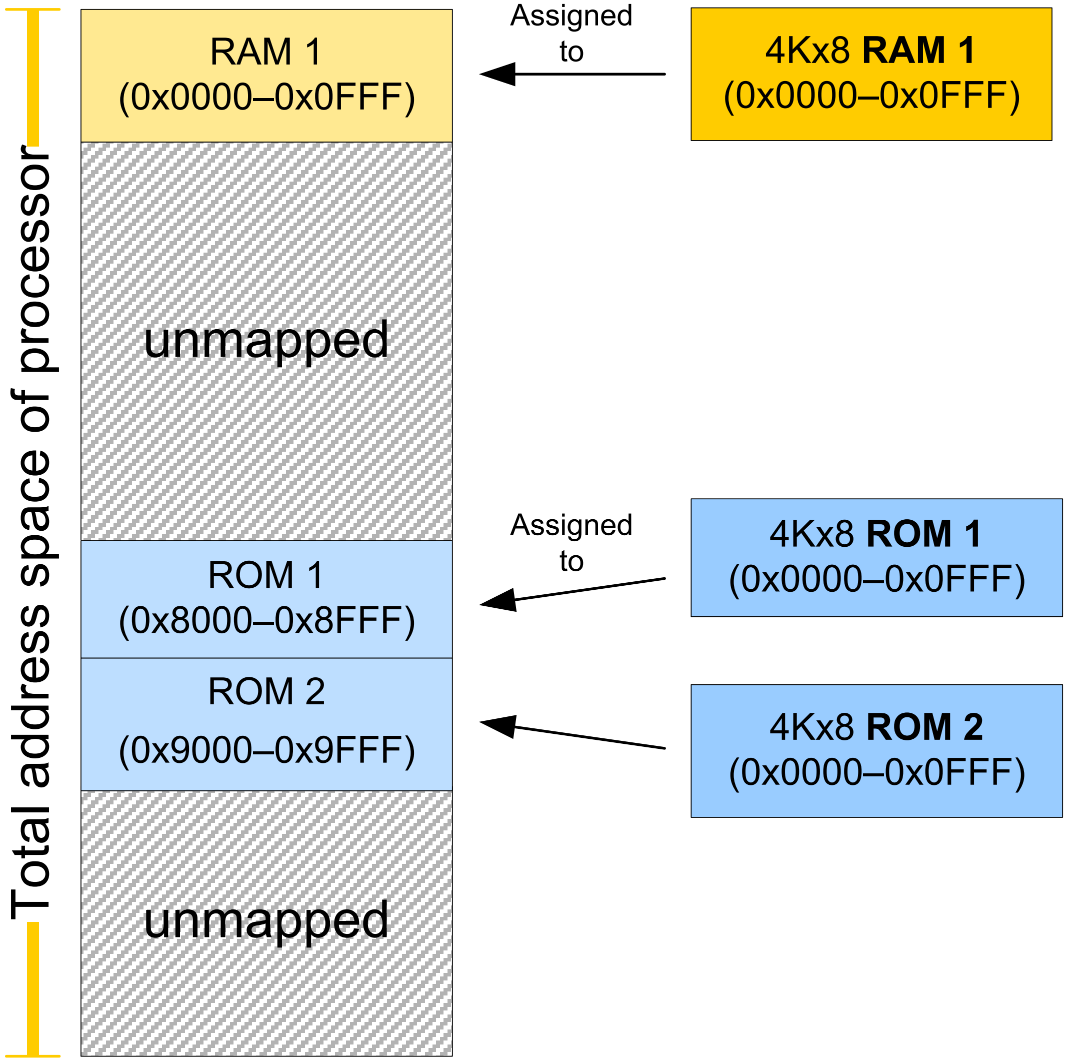
How many (a) locations, (b) KiB when using

- □ *6 address lines*
- □ *14 address lines*
- □ *21 address lines*

How many address lines used for

- □ *6 locations*
- □ *32 locations*
- □ *2KiB memory*
- □ *16KiB memory*
- □ *640KiB memory*

Address arithmetic 2: the address map



Total address space of processor

RAM 1
(0x0000–0x0FFF)

unmapped

ROM 1
(0x8000–0x8FFF)

ROM 2
(0x9000–0x9FFF)

unmapped

Assigned to

4Kx8 **RAM 1**
(0x0000–0x0FFF)

Assigned to

4Kx8 **ROM 1**
(0x0000–0x0FFF)

4Kx8 **ROM 2**
(0x0000–0x0FFF)

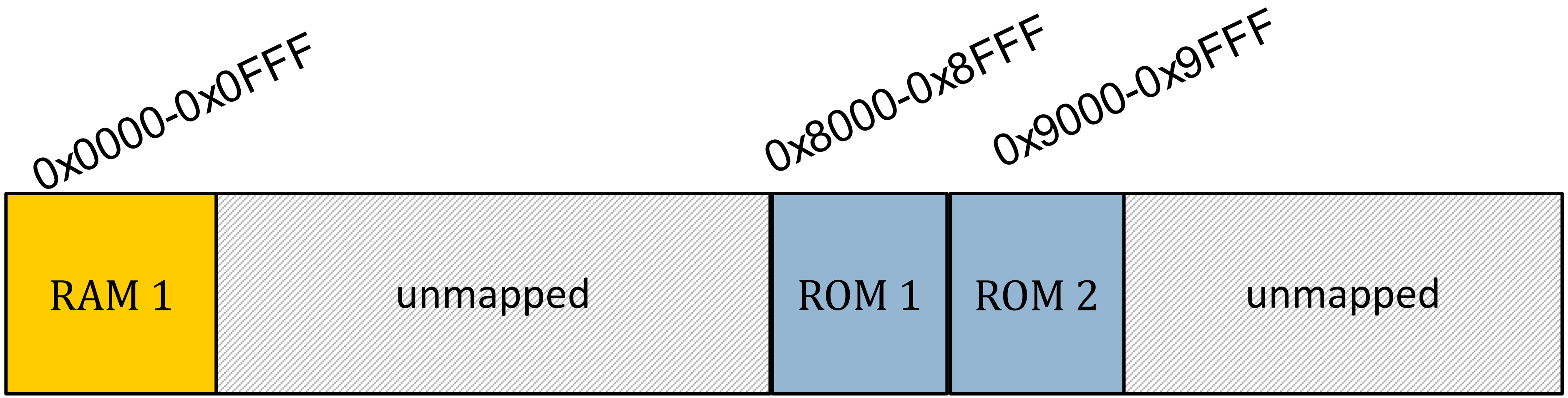
4K·B = 4096

#addr. lines, k, = ceiling($(\log_2(4096))$)
= 12 address lines

The address map is typically drawn top to bottom like this, with the minimum address (0) at the top and the maximum address at the bottom.

The max address is 2^k, where k is the number of address lines.

Contd.



It is also possible to draw the same address map left-to-right as shown (although this is less common).

Here the minimum address (0) is at the left and the maximum address is at the right.

Address arithmetic 2

- For a device containing m locations there are m addresses (i.e. each location has a unique address to identify it)
- The internal address range of a peripheral device (such as a memory device) must be mapped to some subrange of the global address space used by the processor
- If a device with m locations is mapped to the global address space starting at address START, then the full address range occupied by this device will be: **START…START+(m-1)**
- For devices mapped next to each other in memory, startAddr for device$_N$ is equal to endAddr for device$_{N-1}$ + 1

Address arithmetic 2 - Example

- consider with a processor with a 64 KiB address range divided into two 32 KiB blocks that immediately follow one another
  - Total address space: 64*1024 =65536 unique addresses
    - Address are numbered 0…(65536-1) = 0…65535
    - In hex this is 0x0000 to 0xFFFF
  - Block 1 (32 KiB): has 32 * 1024 = 32768 addresses
    - Start address: 0; end address = (0 + 32768 – 1) = 32767
    - In hex the start and end are 0x0000 and 0x7FFF respectively
  - Block 2 (32 KiB): also has 32 * 1024 = 32768 addresses
    - start address= (32767+1)=32768; last address (32768 + 32768 – 1) = 65535
    - In hex the start and end are 0x8000 and 0xFFFF respectively

Address arithmetic 3

- To convert number to hex
  - **use calculator function** (easiest if available)
  - Or know your hex tables (easy if you can remember them)
  - Or do it manually (next slide)
- Because we are usually interested in address ranges of the form START.. START+M-1 it is useful to know the hex conversions for various values of M and M-1

M	M	M-1
Dec	Hex	Hex
16	0x10	0xF
256	0x100	0xFF
1KiB	0x400	0x3FF
4KiB	0x1000	0xFFF
16KiB	0x4000	0x3FFF
64KiB	0x10000	0xFFFF
256KiB	0x40000	0x3FFFF
1MiB	0x100000	0xFFFFF
4MiB	0x400000	0x3FFFFF
16MiB	0x1000000	0xFFFFFF

Address arithmetic 4 – [for info only 2020-2021]

- General method for converting decimal to hex manually
  1. Let *d* = decimal number and *h* = hex number, initially with no digits (empty)
  2. *result, remainder = d / 16*
  3. Convert *remainder* to a single hex digit and insert as most significant digit of hex number
  4. Let *d = result* and repeat from step 2 until *result* is 0

- Example: convert 312 decimal to hex
  - d = 312, h = empty(NOTE: not zero)
  - 312 / 16 = 19 rem **8** => h = 0x**8**, d = 19
  - 19 / 16 = 1 rem **3** => h = 0x**3**8, d = 1
  - 1 / 16 = 0 rem **1** => h = 0x**1**38, d = 0, so end
  - 312 decimal is 0x138 hex

Self test questions

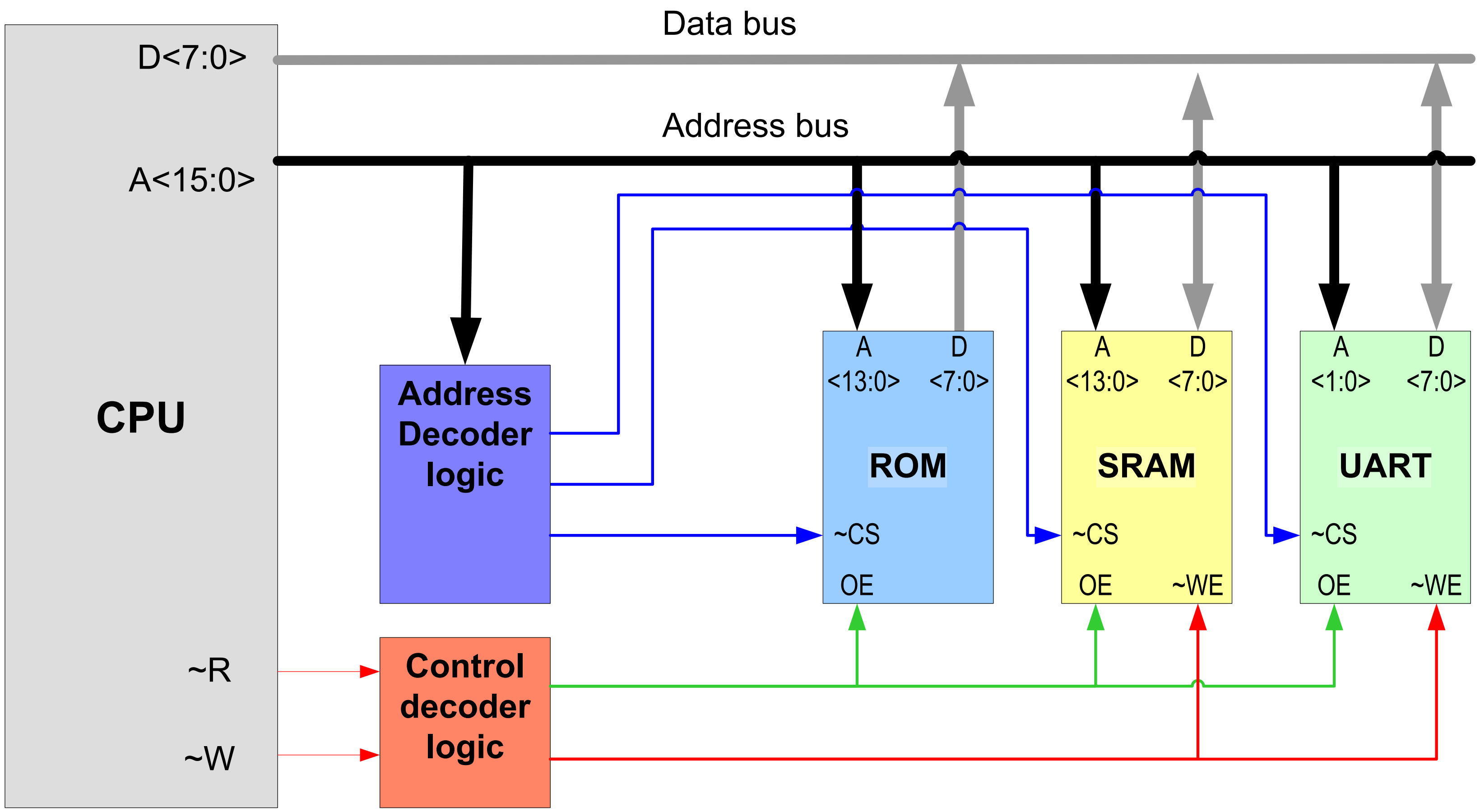
Determine hex address ranges for…

- □ *1024 Bytes, subdivided into 192 + 64 + 512 + 256*
- □ *256 Bytes, subdivided into 4 + 4 + 8 + remainder*
- □ *128KiB, subdivided into 16KiB + 16KiB + 64KiB + remainder*

Hint: use calculator

Processor interfacing

The processor interface (memory mapped IO) [For info 2021]



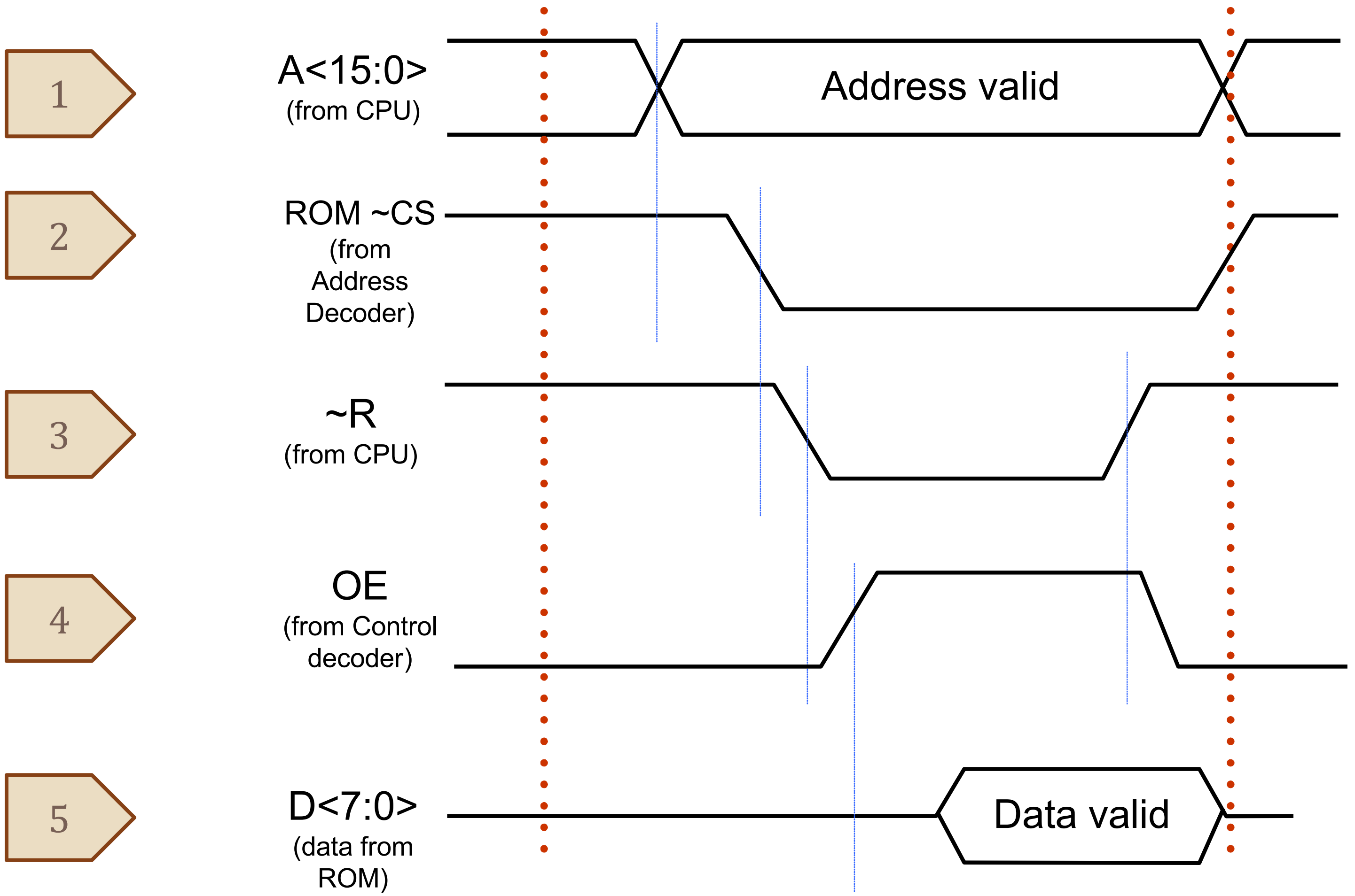
How does this work?

How does the processor interface work? [For info 2021]

For the sake of example consider how the processor/CPU reads a single byte from the ROM device

1. *The CPU outputs a valid address on the address bus. In this case it must be an address within the range that has been mapped to the ROM.*
2. *The address decoder logic decodes the address (on the address bus lines) such that at most one device is selected, in this case the ROM, by activating the appropriate chip select (CS) line.*
   - □ *Now the ROM device knows that it has been selected and it starts decoding the least significant 14 lines of the address bus to identify an internal location to access*
3. *After a short time the CPU activates its read control line (R), indicating that it wants to read a value from an external device via the data bus*
4. *The control decoder logic maps this control input to activating the output enable (OE) line.*
5. The combination of OE and CS tells the ROM to output the data value in the internal location onto the data bus lines. The internal location within the ROM is determined by the low order 14 bits of address on the address lines.

Processor interface timing [For info 2021]

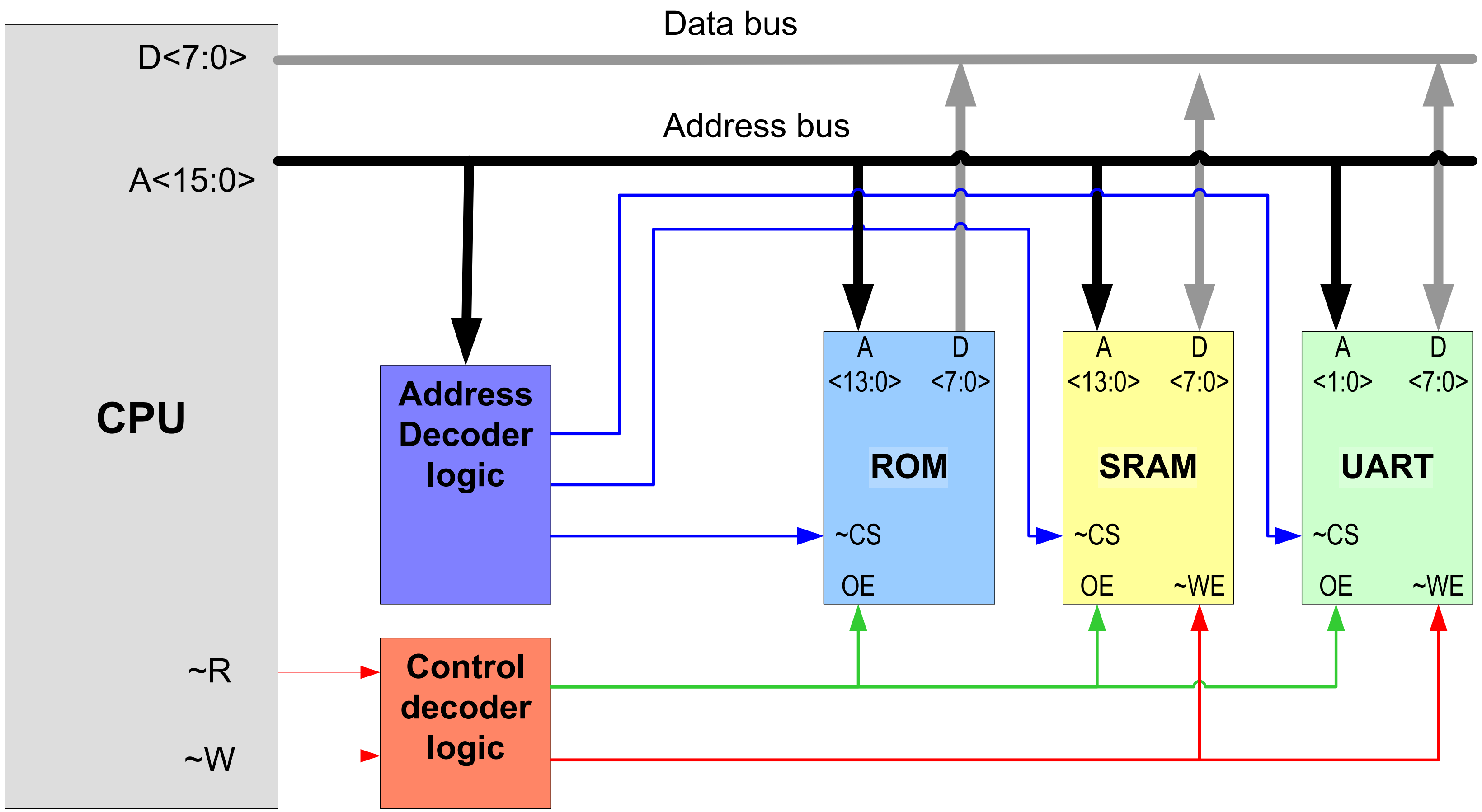


Compare with description on previous slide

Memory mapped I/O (MMIO)

- In addition RAM and ROM memory a useful computer system requires I/O to interface to the outside world
  - Such I/O takes the form of peripheral devices which must be accessed by the processor (E.g. digital port devices, ADCs, Comparators, USART serial communications, etc.)
  - Very simple I/O devices (such as a digital I/O port) may have just one internal location or register
  - More complex devices (e.g. USART) usually have several internal registers that can be addressed
- For **memory mapped I/O**, the internal locations/registers of I/O devices are mapped to sub-ranges of the processor’s address space in exactly the same way as RAM or ROM memory devices
- There is an alternative approach called **port mapped I/O** which is outside the scope of this module

The processor interface (memory mapped IO)



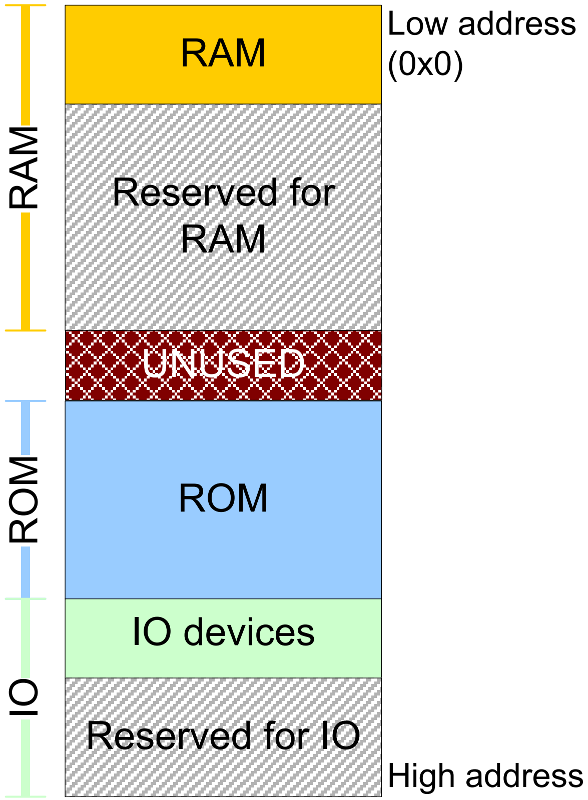
How do we design the control decoder and address decoder logic?

Processor interfacing procedure

1. Determine range of addresses to be supported
   - □ Is the address bus connected directly to processor address lines or is it augmented by additional lines (e.g. time multiplexing, or paged/banked/segmented)
2. Determine the processor control outputs
   - □ How is a read/write request signalled?
   - □ Is IO access signalled differently than memory access?
3. Is there a plan/specification for how addresses should be allocated to memory and IO devices?

Contd.

4. Draw the **address map**
   - □ Show how all possible addresses in the full address range are used (or not used)
   - □ Identify supported/reserved address ranges (for RAM, ROM, IO devices, etc.) based on the plan/spec
   - □ Indentify which part of supported/reserved range is actually used by physical devices (i.e. mapped) and which parts are currently reserved but not mapped
   - □ Indicate areas which will never be used (i.e. not reserved and not mapped)



Example address map

Contd.

5. Design the **address decoding**

- Address decoding logic is used to select which of the (many) devices connected to data bus should respond to the CPU's read or write request
  - This is achieved by activating the chip select of the correct device (as specified by the address map)
- This logic may be implemented using discrete logic (simple gates), data decoders, PROM, FPGAs, programmable address decoders, etc. based on algebraic expressions
- The address decoding input consists of all or some of the address lines coming from the CPU. There is also the possibility of augmenting the CPU address lines with additional inputs (control lines or additional latched address lines)
- The address decoder output is at most one chip select activated (if the input address is one of those which is mapped to a peripheral device)

Processor interfacing procedure

6. Connect data bus lines to device data lines
   - Usually trivial
7. Connect address bus lines to device address lines
   - Usually individual peripherals just require a subset of the address lines
   - The number of address lines required depends on the number of addressable locations inside the peripheral
     - E.g. 1024x8 memory needs 10 address lines
     - E.g. 6 register IO device needs just 3 address lines
8. Connect processor control lines (via decoder if necessary) to output enable (OE) and write enable (WE) lines of devices

What do you need to know for 2020-2021?

- □ Be able to read/interpret an address map
    - □ Given a start address and number of addresses occupied by the device, calculate last address occupied by the device (usually in hex)
    - □ Given start address and last address occupied by a device (usually in hex), calculate the number of addresses/locations occupied by the device
- □ Design the address decoding
    - □ Primarily this means calculating the algebraic expressions required to select each device based on the CPU address lines (basic and extended method – see later)
- □ Understand the circuit block diagrams showing how the CPU is connected to the address and control decoders and to the memory and IO devices

MMIO EXAMPLE 1

- Consider the following system
  - 8 bit CPU with 16 bit addresses using MMIO
  - Address plan is as follows
    - First 32 KiB reserved for ROM, but just 8 KiB installed
    - Next 16 KiB reserved for RAM, but just 8 KiB installed
    - Next 8 KiB is reserved for IO. Just one 4 register device is installed.
    - Remaining addresses are never mapped

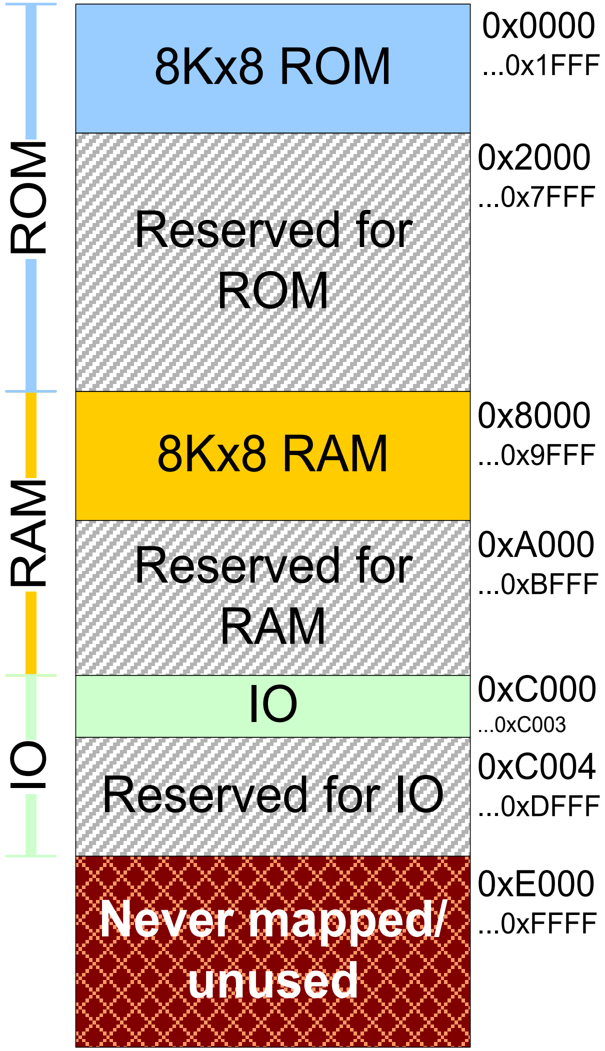
    Note: this is a somewhat improbable example!

MMIO example 1: steps 1-4

- Range of addresses
  - 16 bit addresses => 2^16 = 65536 addresses numbered (in hex) 0x0000 to 0xFFFF
- Control inputs
  - Not specified in the question so let’s assume separate read and write (R and W) lines.
  - Since we are using MMIO, there is no difference between memory and I/O device access so no extra control lines needed
- Plan for address allocation and draw the address map –next slide

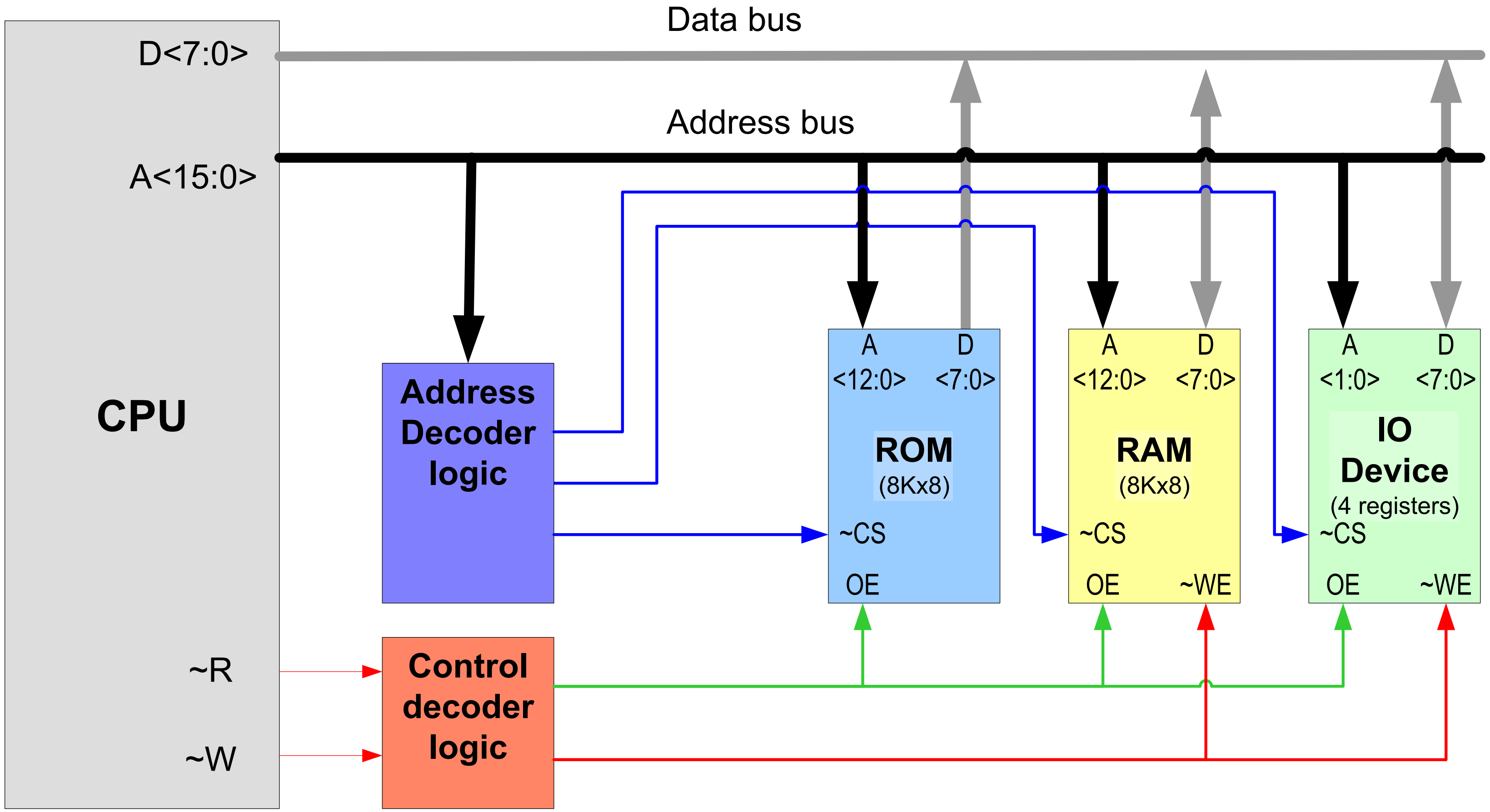
contd

- 8 bit CPU with 16 bit addresses
  - Total range: 0x0000..0xFFFF
- Memory map
  - First 32KiB ROM: 0x0000..0x7FFF
    - just 8 KiB ROM installed: 0x0000..0x1FFF
  - Next 16 KiB RAM: 0x8000..0xBFFF
    - just 8 KiB RAM installed: 0x8000..0x9FFF
  - Next 8 KiB for IO devices: 0xC000..0xDFFF
    - Just one 4 register device installed: 0xC000..0xC003
  - Remainder unused/never mapped: 0xE000..0xFFFF



MMIO example 1: steps 5-8

- Address decoding – we'll come back to this
- Connect the data bus, address bus, and control lines



Address decoding

Address decoder logic – **basic method**

- *Precondition: if you do not already have an address map, you'll need to design/sketch that first*
- Examine the range of addresses mapped to each physical device in hex (by inspecting the address map)
- We can only use the basic method (to be described next) if the next two necessary conditions are satisfied
  - the address range occupied by the device is a power of 2 number of locations
  - the start of the mapped addresses for the device is an integer multiple of the address range size
- Examples
  - An IO device with 16 internal registers must be mapped so that its start address is an integer multiple of 16 (i.e. one of 0x0, 0x10, 0x20, 0x30, etc.)
  - A memory device with 2KiB of memory must be mapped to start on an integer multiple of 2KiB (i.e. one of 0x0, 0x800 , 0x1000, 0x1800, 0x2000, etc.)

Address decoder logic – **basic method** contd

- **If the two necessary conditions for the basic methods are not satisfied then it will not be possible to use the basic method and you MUST use the extended method described later**
  - Example: a memory device with 12 KiB of memory cannot use the basic method. The address range is not a power of 2.
  - Example: an IO device with 16 registers mapped into memory starting at address 0xC008 cannot use the basic method. The start address C008 is not an integer multiple of 16, the nearest multiples of 16 would be 0xC000 or 0xC010.

Contd.

- *Assuming we are OK to proceed with the basic method…*
- For each device, we want to develop an algebraic expression which will activate the chip select of that device when the CPU attempts to access one of the addresses to which the device is mapped
    - Essentially we have a truth table where the CS line for our device is active whenever the CPU address lines refer to an address "occupied" by that device, and CS is inactive for all other addresses.
- Write the start and end address in the address map which corresponds to each physical device in <u>binary form</u>
    - Assuming we satisfy the conditions of the basic method, then low order bits which change between the start and end addresses are considered to be "don't cares" and are not included in the algebraic expression
    - High order bits which remain constant between the start and end addresses are required to chip select the device and are included as-is in the algebraic expression
    - Finalise the algebraic expression by adding control inputs (if necessary) and negating the output if the required chip select is active-low

Basic method example

(Refer back to MMIO example 1)

(1) The ROM occupies 8 KiB starting from 0x0000 to 0x1FFF:

0x0000 = **000**0 0000 0000 0000
0x1FFF = **000**1 1111 1111 1111
⇒ ~ROM_CS = ~(~A_{15}.~A_{14}.~A_{13})

ROM_CS = $\overline{(\bar{A}_{15}\bar{A}_{14}\bar{A}_{13})}$

(2) The RAM occupies 8 KiB starting from 0x8000 to 0x9FFF:

0x8000 = **100**0 0000 0000 0000
0x9FFF = **100**1 1111 1111 1111
⇒ ~RAM_CS = ~(A_{15}.~A_{14}.~A_{13})

(3) The IO device occupies 4 locations starting from 0xC000 to 0xC003:

0xC000 = **1100 0000 0000 00**00
0xC003 = **1100 0000 0000 00**11
⇒~IO_CS = ~(A_{15}.A_{14}.~A_{13}.~A_{12}.~A_{11}.~A_{10}.~A_{9}.~A_{8}.~A_{7}.~A_{6}.~A_{5}.~A_{4}.~A_{3}.~A_{2})

Basic method self test question

Q1. What is the CS expression for a ROM that occupies 4 KiB starting from 0x4000?

Q2. What is the CS expression for an IO device that has 16 registers mapped to memory starting at address 0xF000?

0xF000 = 0b 1111 0000 0000 0000

0xF00F = 0b 1111 0000 0000 1111

1111 0000 0000 xxxx

$$\overline{IO\_CS} = \overline{\left(A_{15}A_{14}A_{13}A_{12}\ \overline{A}_{11}\overline{A}_{10}\overline{A}_{9}\overline{A}_{8}\ \overline{A}_{7}\overline{A}_{6}\overline{A}_{5}\overline{A}_{4}\right)}$$

Address decoder logic – **extended method**

- □ *Precondition: if you do not already have an address map, you'll need to design/sketch that first*
- □ Examine the range of addresses mapped to each physical device (by inspecting the address map)
- □ You need to use the extended method for a device if either of the following are true:

  (a) the number of addresses in the address range is not a power of 2, i.e. log2(numDeviceAddresses) is not an integer

  (b) the number of addresses in the device does not divide evenly (no remainder) into the start of the mapped addresses for the device, i.e. remainder(startAddress/numDeviceAddresses) ≠ 0 :

Extended method contd.

- Some examples that require the extended method
  - 12 KiB device – the number of internal addresses int the device (12*1024) is not a power of 2
  - 8 KiB device starting at address 0x1000 – 8 KiB corresponds to 0x2000 addresses (8192 in decimal) so it is a power of 2, but remainder(startAddress/numAddresses)=remainder(0x1000/0x2000) = 0x1000

ASIDE: There is an alternative to the extended method in some cases for a device which is not a power of 2 size. It involves increasing the address range allocated to that device to be a power of 2 and permitting ***address mirroring*** *(wherein multiple addresses in the global address map refer to a single location within a device)—we won't use this method for now but we mention it as the concept was used in some past exam papers*

Extended method contd.

- How to apply the extended method

1. Divide the address range of the device into a number of consecutive sub-ranges which satisfy the requirements of the basic method (i.e. a power of 2 number of addresses that starts on an integer multiple of its size)

   See next slide for details of how to choose subranges.

2. For each sub-range, apply the basic method to develop an algebraic expression for the sub-range

3. Finally, use a logical-OR to combine the expressions for all the sub-ranges of a single device into one expression

Extended method contd.

- How to choose subranges

A. To start:
Set **remainingAddresses** = **numDeviceAddresses**
in the global addr map
Set **subrangeStart** = first address allocated to device

B. Find **subrangeAddresses** as the largest power of 2 which is less than or equal to **remainingAddresses** and divides evenly (no remainder) into **subrangeStart**

C. Now we have one subrange suitable for the basic method defined by the range: **subrangeStart** … **subrangeStart** + **subrangeAddresses** - 1

D. Update remainingAddresses and subrangeStart:
remainingAddresses$_{n}$ = **remainingAddresses**$_{n-1}$ – **subrangeAddresses**$_{n-1}$
subrangeStart$_{n}$ = **subrangeStart**$_{n-1}$ + **subrangeAddresses**$_{n-1}$

E. To identify remaining subranges, repeat from step B.

Extended method example

Consider a ROM occupying 12 KiB starting from 0x0000:

0x0000 = 0000 0000 0000 0000

0x2FFF = 0010 1111 1111 1111

Note that ~CS is not ~(~A_{15}.~A_{14}.~A_{12}) – check what happens to A_{12} as you count up from 0x0000 to 0x2FFF

Divide the address range into power-of-2 subranges, both of which start on an Integer multiple of subrange size

remaining = 12KiB, subrangeStart = 0, subrangeAddresses = 8KiB

0x0000 = **000**0 0000 0000 0000

0x1FFF = **000**1 1111 1111 1111

subrange1 = (~A_{15}.~A_{14}.~A_{13})

Subrange 1: 8 KiB (0x2000) starting at 0

remaining = 12KiB − 8KiB = 4KiB, subrangeStart = 0 + 0x2000 (8KiB) = 0x2000

0x2000 = **0010** 0000 0000 0000

0x2FFF = **0010** 1111 1111 1111

subrange2 = (~A_{15}.~A_{14}.A_{13}.~A_{12})

Subrange 2: 4 KiB (0x1000) starting at 0x2000

Finally combine subranges using logical-OR

~ROM_CS = ~((~A_{15}.~A_{14}.~A_{13}) + (~A_{15}.~A_{14}.A_{13}.~A_{12}))

~(subrange1 + subrange2)

Extended method self test question

Consider a RAM occupying 16 KiB starting from 0x6000:

```
0x6000 = 0110 0000 0000 0000
0x9FFF = 1001 1111 1111 1111
```

16KiB = 0x4000 addresses

Why does this device require the extended method?

rem(0x6000/0x4000) = 0x2000 ≠ 0

Divide the address range into power-of-2 subranges, which start on an Integer multiple of subrange size

remaining = 16KiB = 0x4000. start = 0x6000, subrangeSize = 0x2000

⇒ 0x6000

0x6000 + 0x2000 − 1 = 0x7FFF

subrange1 = . . .

remaining = 0x4000 − 0x2000 = 0x2000, start = 0x6000 + 0x2000 = 0x8000, subrangeSize = 0x2000

⇒ 0x8000

0x8000 + 0x2000 − 1 = 0x9FFF

subrange = . . .

Finally combine subranges using logical-OR

~RAM_CS = $\overline{(\text{subrange1} + \text{subrange2}})$

How to simplify address decoding

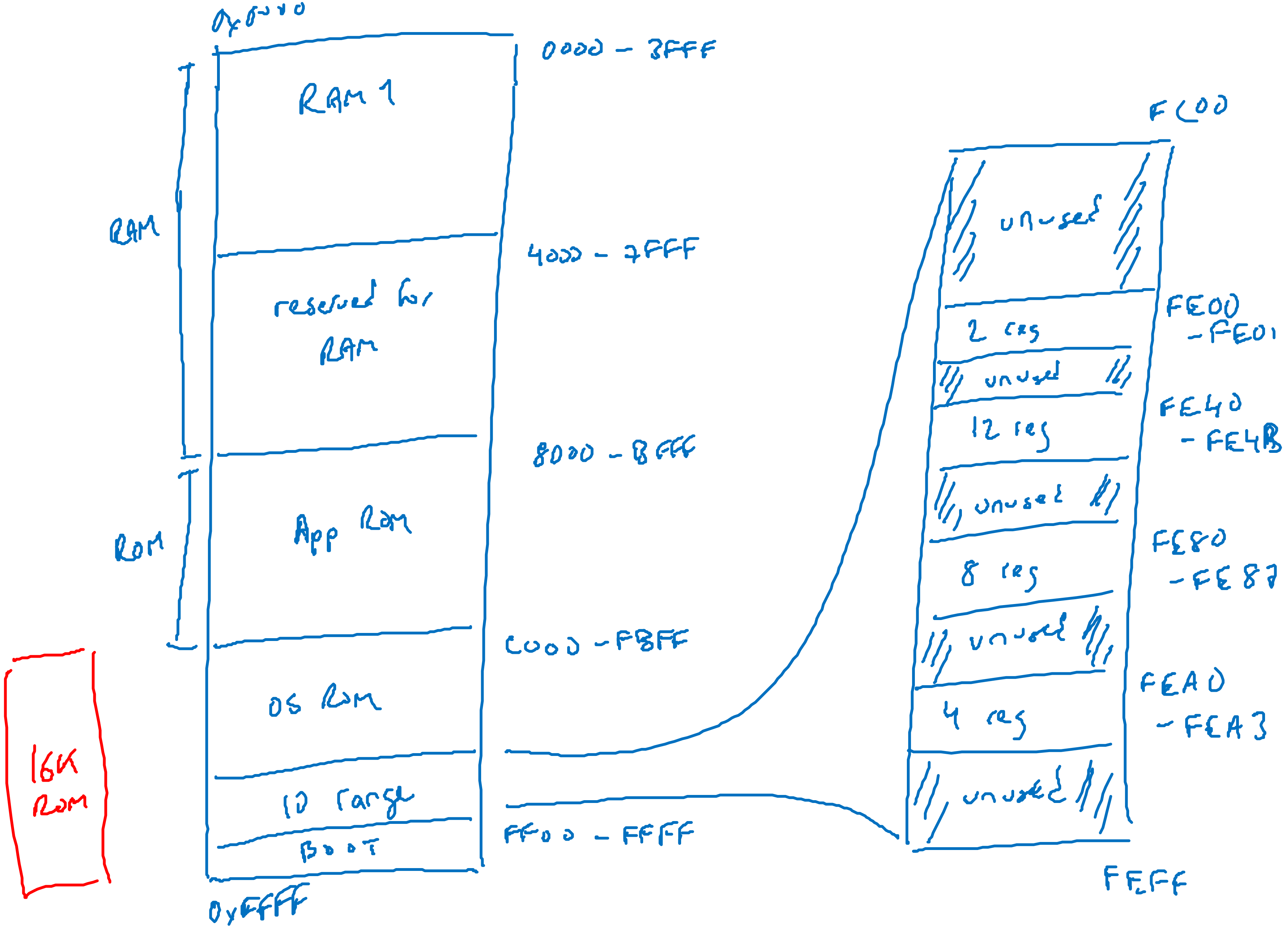
- Clearly the extended method is more time consuming or difficult to implement than the basic method, so we should try to design our system so we can use the basic method if possible
- Therefore
  - Choose devices with a power of 2 size is possible. Normally memories have a power of 2 size, but IO devices may not and that is outside our control.
  - If we have a device with a power of 2 size, always map it so that it starts from an address which is an integer multiple of its size if at all possible,
- E.g. suppose we have deviceA is size 4 and deviceB is size 16 and IO address start at 0x1000, what addresse~~d~~s should we choose?
  - If we choose deviceA at 0x1000 and deviceB at 0x1004, then we are making life hard because we'll need to use the extended method on device
  - Instead we should choose deviceA at 0x1000 and deviceB at 0x1010 or deviceB at 0x1000 and deviceA at 0x1010. This will allow us to use the basic method.

MMIO example 2: complicated example

- System specification (based loosely on 1980s BBC micro)
- CPU: 8 bit data bus, 16 bit address bus
- Address plan
  - RAM from 0x0000-0x7FFF (16 KiB installed using a single 16Kx8 RAM device)
  - Application ROM from 0x8000-0xBFFF (16KiB installed using a single 16Kx8 ROM device)
  - OS ROM from 0xC000 to 0xFBFF and Boot loader ROM from 0xFF00-0xFFFF (both ranges fully occupied by mapping addresses from a single 16Kx8 ROM device)
  - Memory mapped I/O from 0xFC00-0xFEFF
    - 2 register IO device @ 0xFE00
    - 12 register IO device (parallel port) @ 0xFE40
    - 8 register IO device @ 0xFE80
    - 4 register IO device (serial port) @ 0xFEA0

ASIDE: the gaps between the mapped address ranges of the devices could simplify the decoding logic if address mirroring was allowed (which it isn't for now)

- Sketch the address map, design the address decoding. [For 2020-2021, ignore the OS ROM and boot loader ROM when designing the decoding]


0x0000
0000 - 3FFF
RAM 1
RAM
4000 - 7FFF
reserved for RAM
8000 - BFFF
App Rom
ROM
C000 - FBFF
OS Rom
16K Rom
IO range
FF00 - FFFF
Boot
0xFFFF
FC00
unused
FE00 - FE01
2 regs
unused
FE40 - FE4B
12 regs
unused
FE80 - FE87
8 regs
unused
FEA0 - FEA3
4 regs
unused
FEFF

RAM1 (basic)

0x0000 0000 0000 0000 0000

0x3FFF 0011 1111 1111 1111

00

$\Rightarrow \overline{RAM\text{-}CS} = (\overline{A_{15}}\,\overline{A_{14}})$

2reg (basic)

0xFE00 1111 1110 0000 0000

0xFE01 1111 1110 0000 0001

$\overline{2reg\text{-}CS} = (\overline{A_{15}A_{14}A_{13}A_{12}A_{11}A_{10}A_9\overline{A_8}\,\overline{A_7}\,\overline{A_6}\,\overline{A_5}\,\overline{A_4}\,\overline{A_3}\,\overline{A_2}\,\overline{A_1}})$

12 reg   0xFE40 – 0xFE4B not a power of 2 => extended method

remaining = 12, start = FE40, subrangeSize = 8

=> FE40 = 1111 1110 0100 0000

FE47 $\underbrace{1111\ 1110\ 0100\ 0}111$

subrange1 = ...

remaining = 12 - 8 = 4, start = FE40 + 8 = FE48, subrangeSize = 4

FE48 = 1111 1110 0100 1000

FE4B $\underbrace{1111\ 1110\ 0100\ 10}11$

subrange2 = ...

$\overline{12reg\_CS} = (\overline{subrange1 + subrange2})$