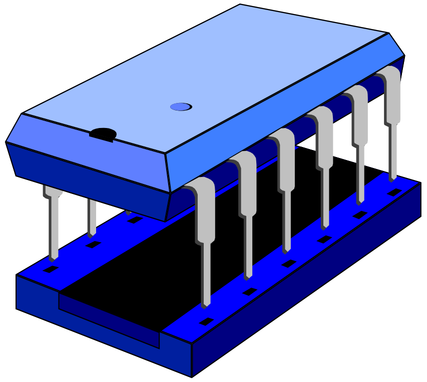
MEMORY ORGANISATION



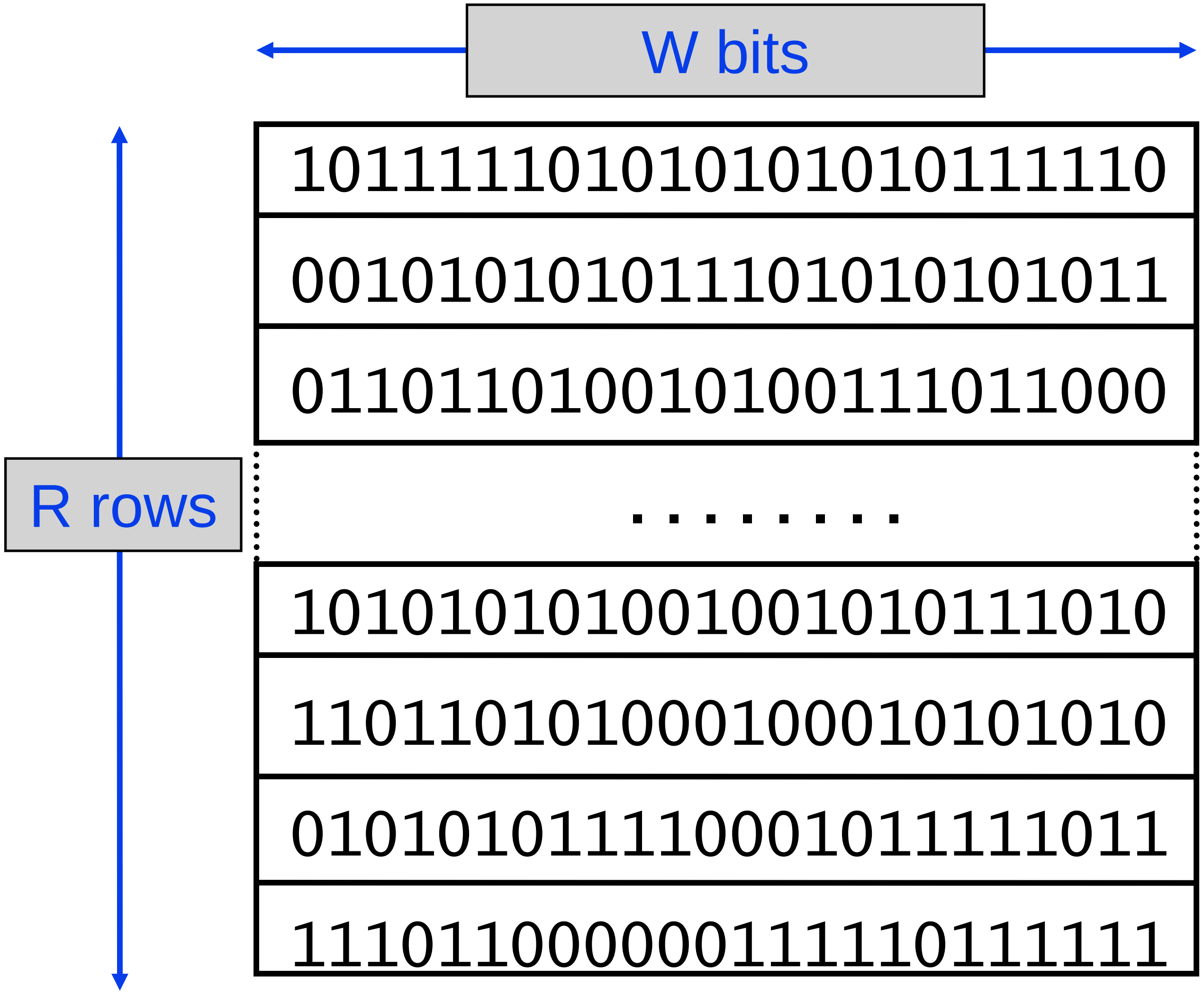
Anandha Gopalan (with thanks to **N. Dulay and E. Edwards**)

axgopala@imperial.ac.uk

Memory Organisation

- Addressing
- Byte Ordering
- Memory Modules and Chips

Main Memory (RAM)



- Each memory location is W bits long
  - Normally a byte-multiple, e.g. 16-bits, 32-bits
- Memory Size
  - R x W *bits*
- Access
  - Can Read/Write entire row or just one byte at a time

Addressing

Main Memory

0110	1101	1010	1101
0000	0000	0000	0011
0000	0000	0000	0000
1111	1111	1111	1111
0000	0000	0000	0000
1001	1010	1010	0010
0000	0000	0000	0000
1111	1111	1111	1110

- Where in memory is the 16-bit value of 3?
- We need a scheme for uniquely identifying every memory location
- **ADDRESSING**
  Identify memory locations with a positive number called the (memory) **address**

Word Addressing

Main Memory		Address	Address (binary)
0110 1101	1010 1101	0	0000
0000 0000	0000 0011	1	0001
0000 0000	0000 0000	2	0010
1111 1111	1111 1111	3	0011
0000 0000	0000 0000	4	0100
1001 1010	1010 0010	5	0101
0000 0000	0000 0000	6	0110
1111 1111	1111 1110	7	0111

Byte Addressing

Main Memory		Word Address
0110 1101	1010 1101	← 0
0000 0000	0000 0011	← 2
0000 0000	0000 0000	← 4
1111 1111	1111 1111	← 6
0000 0000	0000 0000	← 8
1001 1010	1010 0010	← 10
0000 0000	0000 0000	← 12
1111 1111	1111 1110	← 14

- With byte addressing, every byte in main memory has an address
- In this example which is byte 0 and which is byte 1?

Byte Addressing

- Two formats
  - Big Endian
    - Stores *Most Significant Byte* first
    - Motorola 6800, IBM POWER, SPARC, System/360, ARM
  - Little Endian
    - Stores *Least Significant Byte* first
    - x-86, ARM, DEC Alpha, VAX, PDP-11

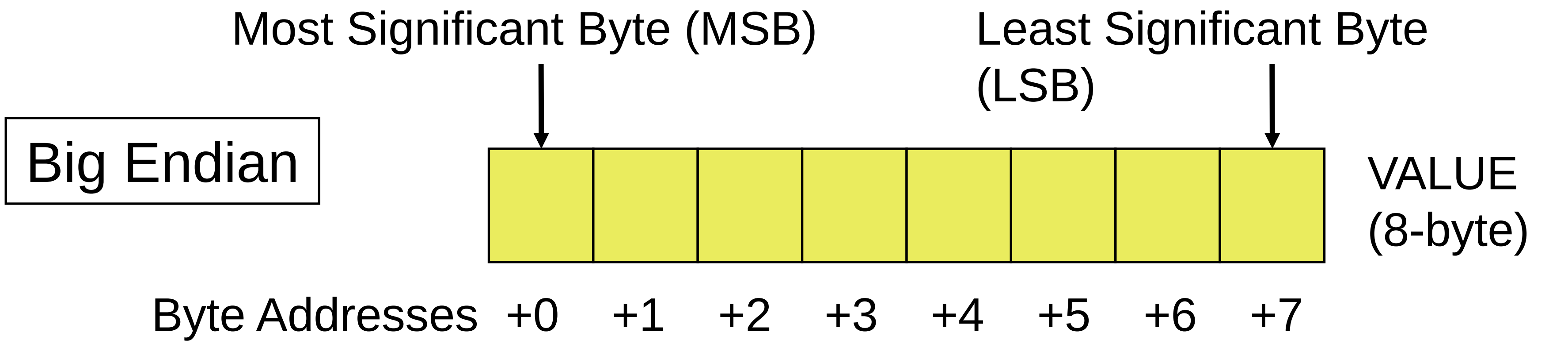
Byte Addressing (Big Endian)

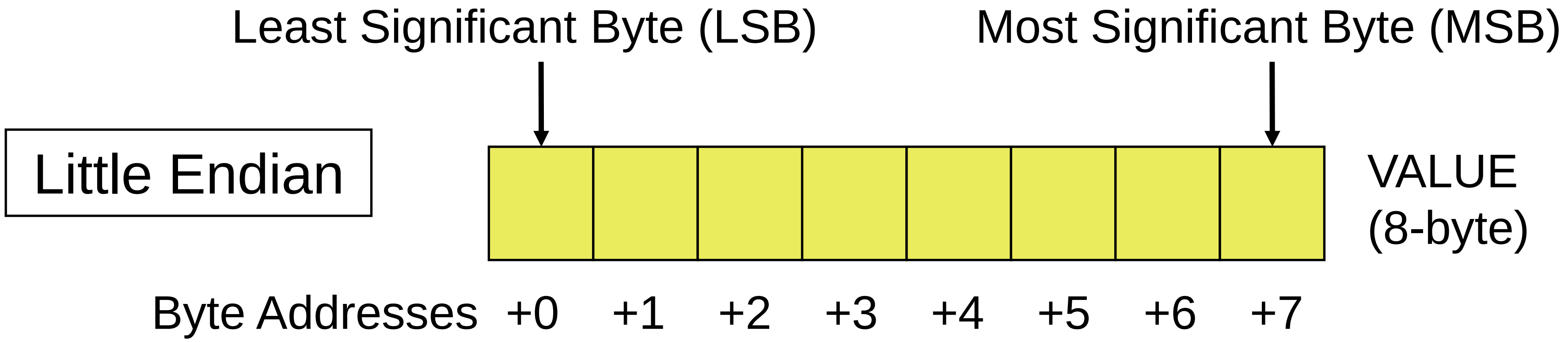
Byte Address	Main Memory		Byte Address
0 →	0110 1101	1010 1101	← 1
2 →	0000 0000	0000 0011	← 3
4 →	0000 0000	0000 0000	← 5
6 →	1111 1111	1111 1111	← 7
8 →	0000 0000	0000 0000	← 9
10 →	1001 1010	1010 0010	← 11
12 →	0000 0000	0000 0000	← 13
14 →	1111 1111	1111 1110	← 15

Byte Addressing (Little Endian)

Byte Address	Main Memory		Byte Address
1 →	0110 1101	1010 1101	← 0
3 →	0000 0000	0000 0011	← 2
5 →	0000 0000	0000 0000	← 4
7 →	1111 1111	1111 1111	← 6
9 →	0000 0000	0000 0000	← 8
11 →	1001 1010	1010 0010	← 10
13 →	0000 0000	0000 0000	← 12
15 →	1111 1111	1111 1110	← 14

Byte Ordering – *Multibyte* Data Items


Most Significant Byte (MSB)
Least Significant Byte (LSB)
Big Endian
VALUE (8-byte)
Byte Addresses +0 +1 +2 +3 +4 +5 +6 +7


Least Significant Byte (LSB)
Most Significant Byte (MSB)
Little Endian
VALUE (8-byte)
Byte Addresses +0 +1 +2 +3 +4 +5 +6 +7

Example 1: 16-bit Integer (View 1)

- 16-bit integer ‘5’ stored at memory address 24

Big Endian

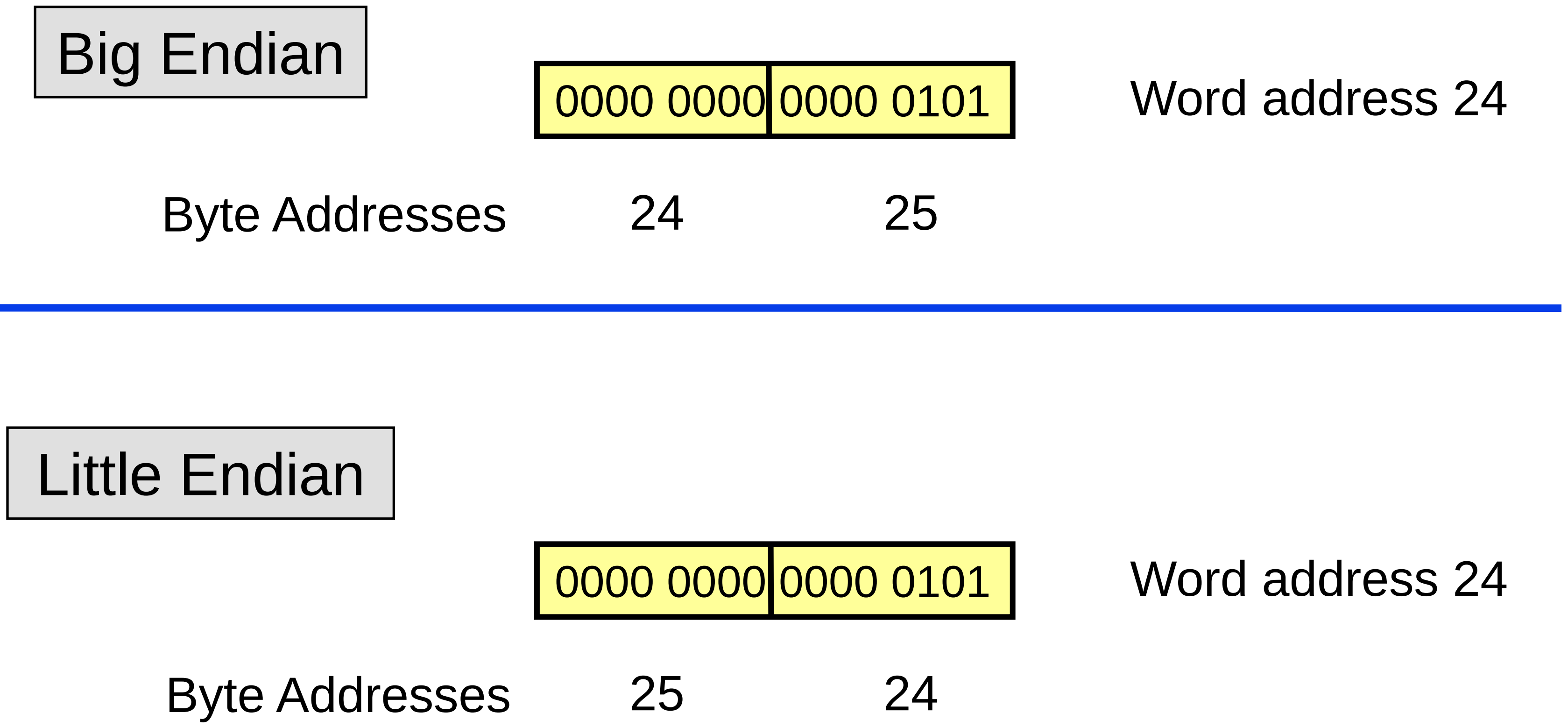
	0000 0000	0000 0101
Byte Addresses	24	25

Little Endian

	0000 0101	0000 0000
Byte Addresses	24	25

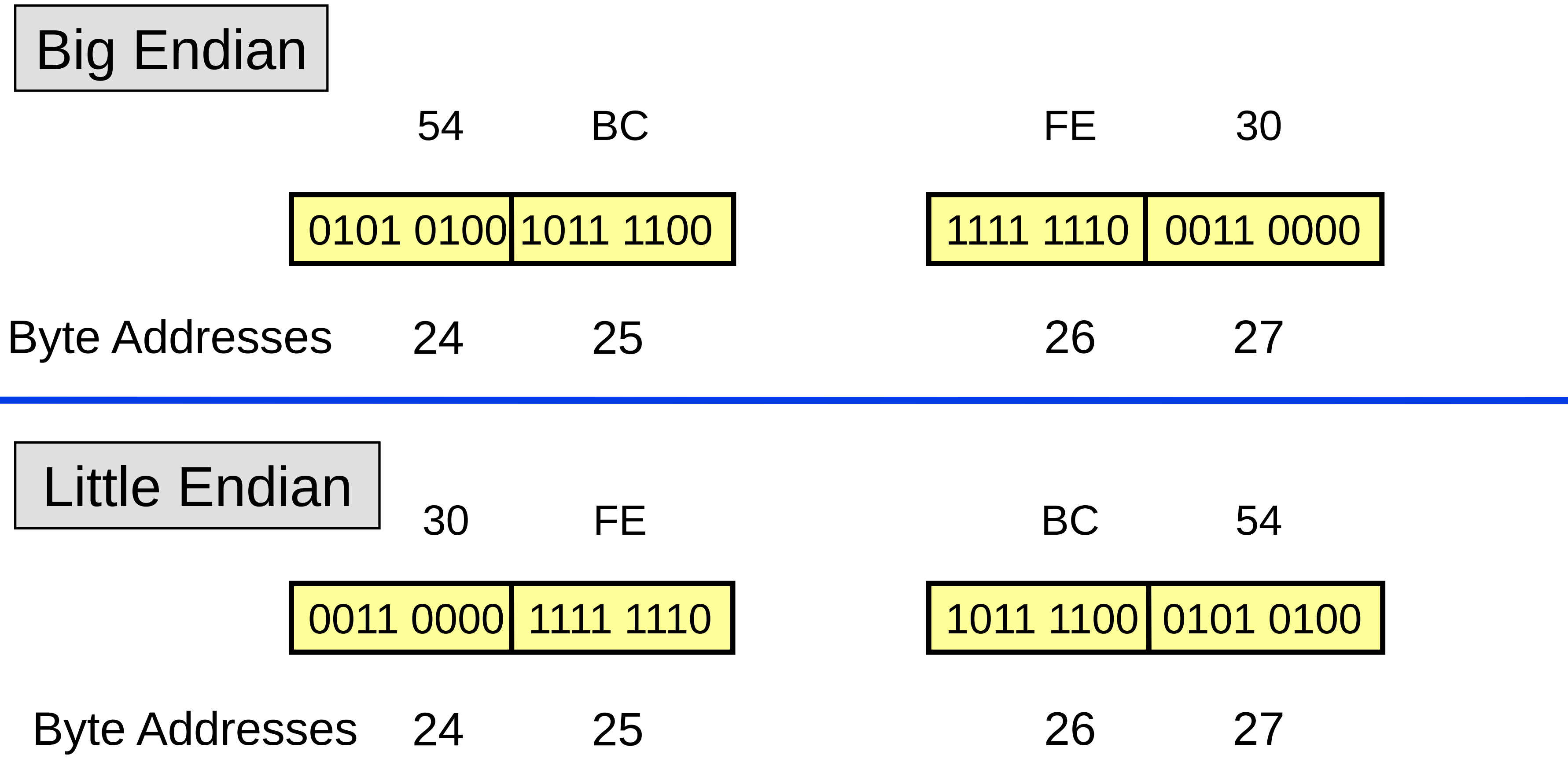
Example 1: 16-bit Integer (View 2)

- 16-bit integer ‘5’ stored at memory address 24



Example 2: 32-bit Value (View 1)

- 32-bit hex value 54 BC FE 30 stored at memory address 24



Example 2: 32-bit Value (View 2)

- 32-bit hex value 54 BC FE 30 stored at memory address 24

Big Endian

	54	BC	FE	30
	0101 0100	1011 1100	1111 1110	0011 0000
Byte Addresses	24	25	26	27

Little Endian

	54	BC	FE	30
	0101 0100	1011 1100	1111 1110	0011 0000
Byte Addresses	27	26	25	24

Example 3: ASCII String (View 1)

- String “JIM BLOGGS” stored at memory address 24
- Treat a string as an array of (ASCII) bytes
  - Each byte is considered individually

Big Endian

	J	I	M		B	L	O	G	G	S
Addresses	24	25	26	27	28	29	30	31	32	33

Little Endian

	S	G	G	O	L	B		M	I	J
Byte Addresses	24	25	26	27	28	29	30	31	32	33

Example 3: ASCII String (View 2)

- String “JIM BLOGGS” stored at memory address 24
- Treat a string as an array of (ASCII) bytes
  - Each byte is considered individually

Big Endian

	J	I	M		B	L	O	G	G	S
Addresses	24	25	26	27	28	29	30	31	32	33

Little Endian

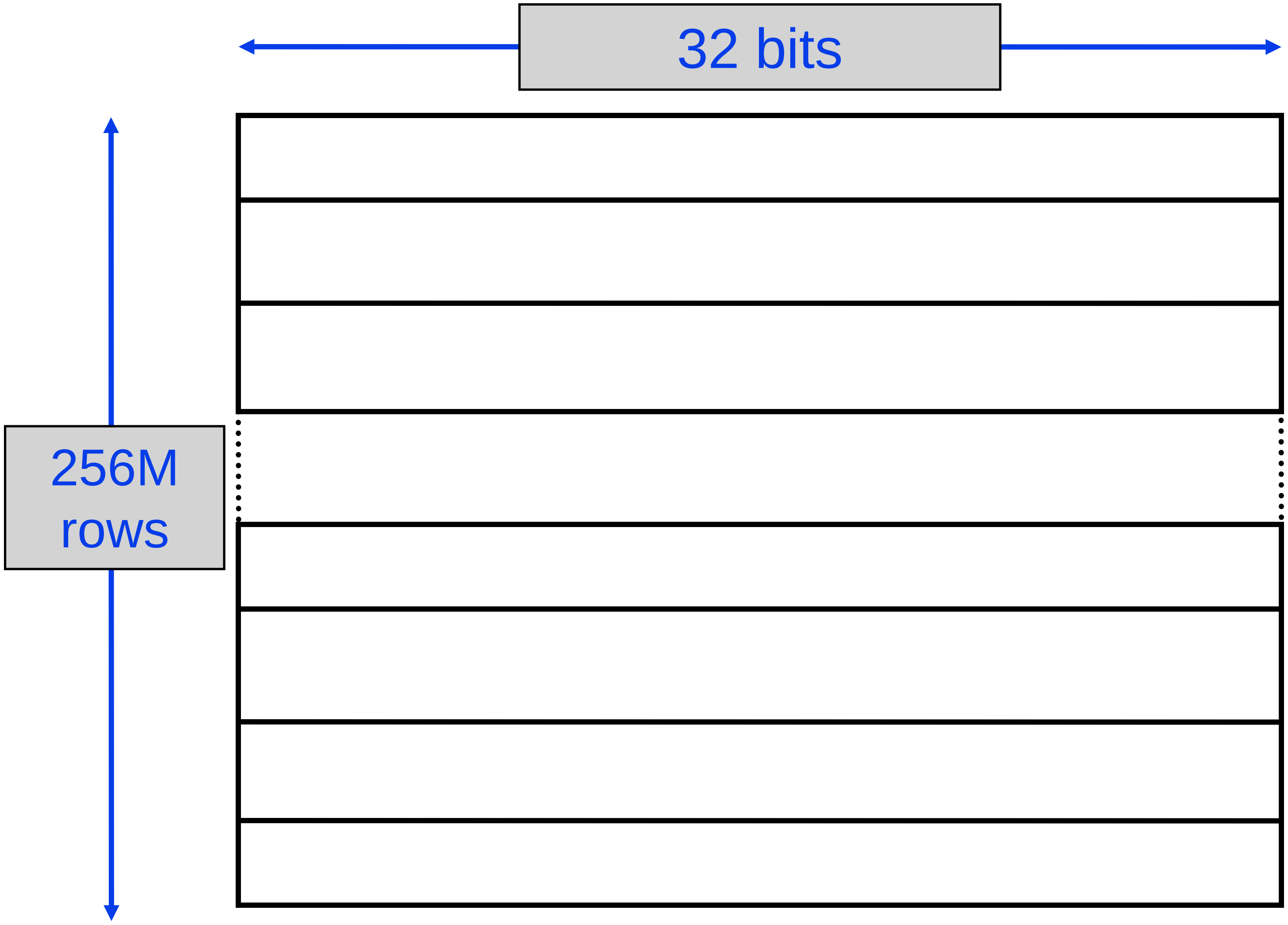
	J	I	M		B	L	O	G	G	S
Byte Addresses	33	32	31	30	29	28	27	26	25	24

Potential Problems

- How do we **transfer a multi-byte value** (e.g. a 32-bit integer) from a Big-Endian memory to a Little-Endian memory and vice-versa?
- How do we transfer an ASCII **string** value (e.g. “JIM BLOGGS”) from a Big-Endian memory to a Little-Endian memory and vice-versa?
- How do we transfer an **object** which holds both types of values above and vice-versa?
- Why is it necessary?

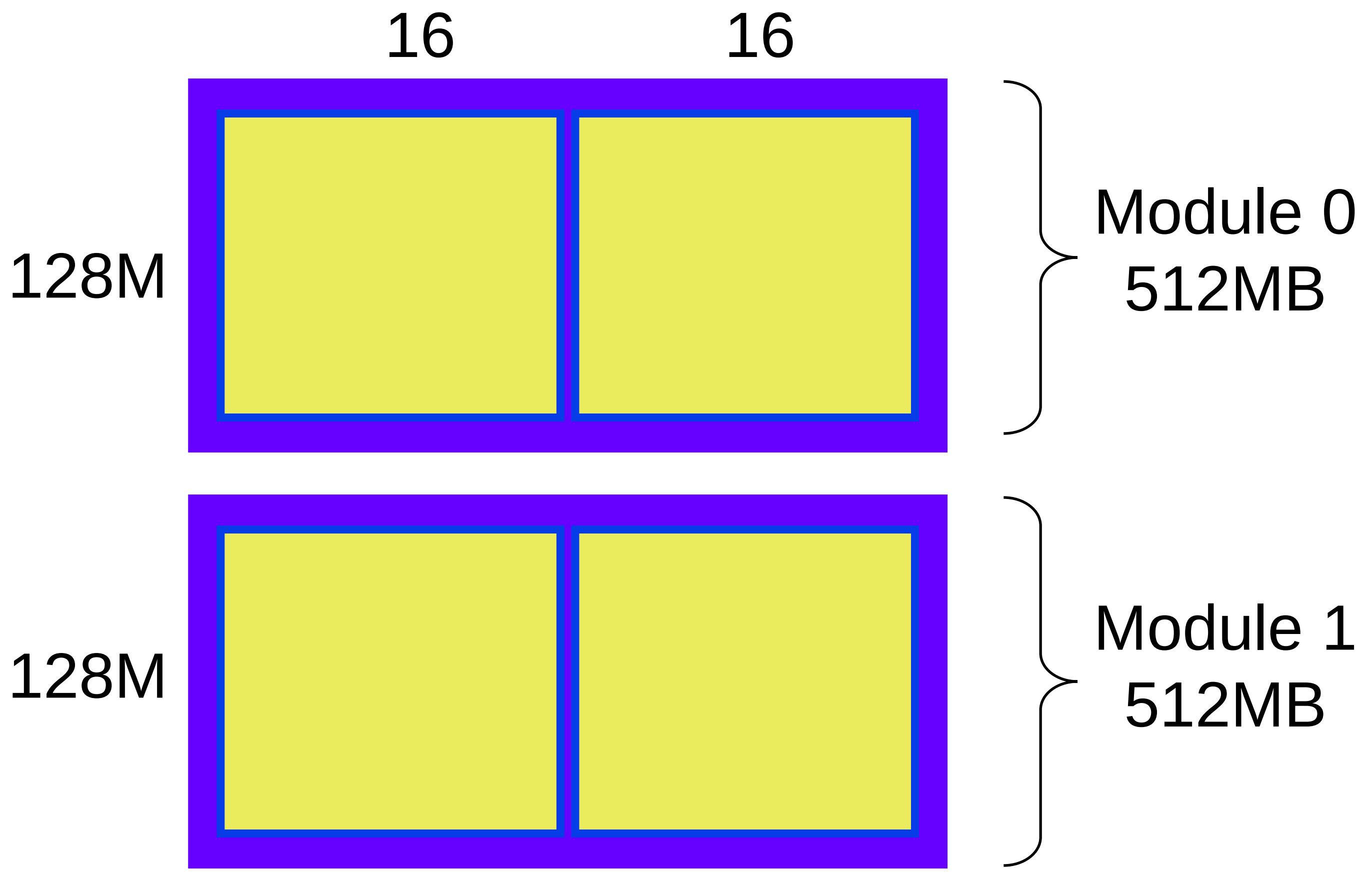
Memory Modules and Chips

1GB (256M x 32-bit) Memory



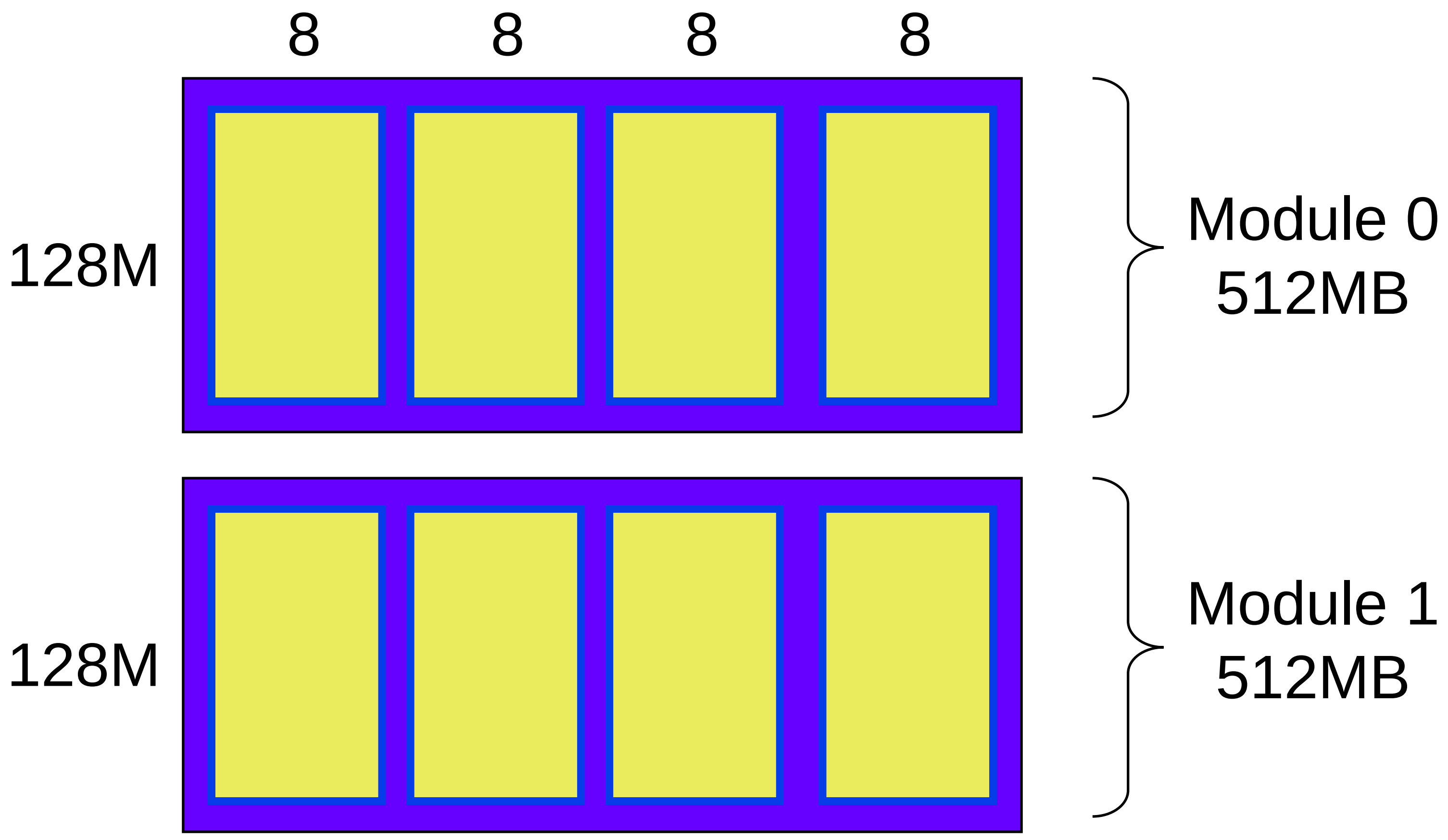
1GB (256M x 32-bit) Memory

- Two 512MB memory modules
  - Each module has two 128M x 16-bit RAM Chips



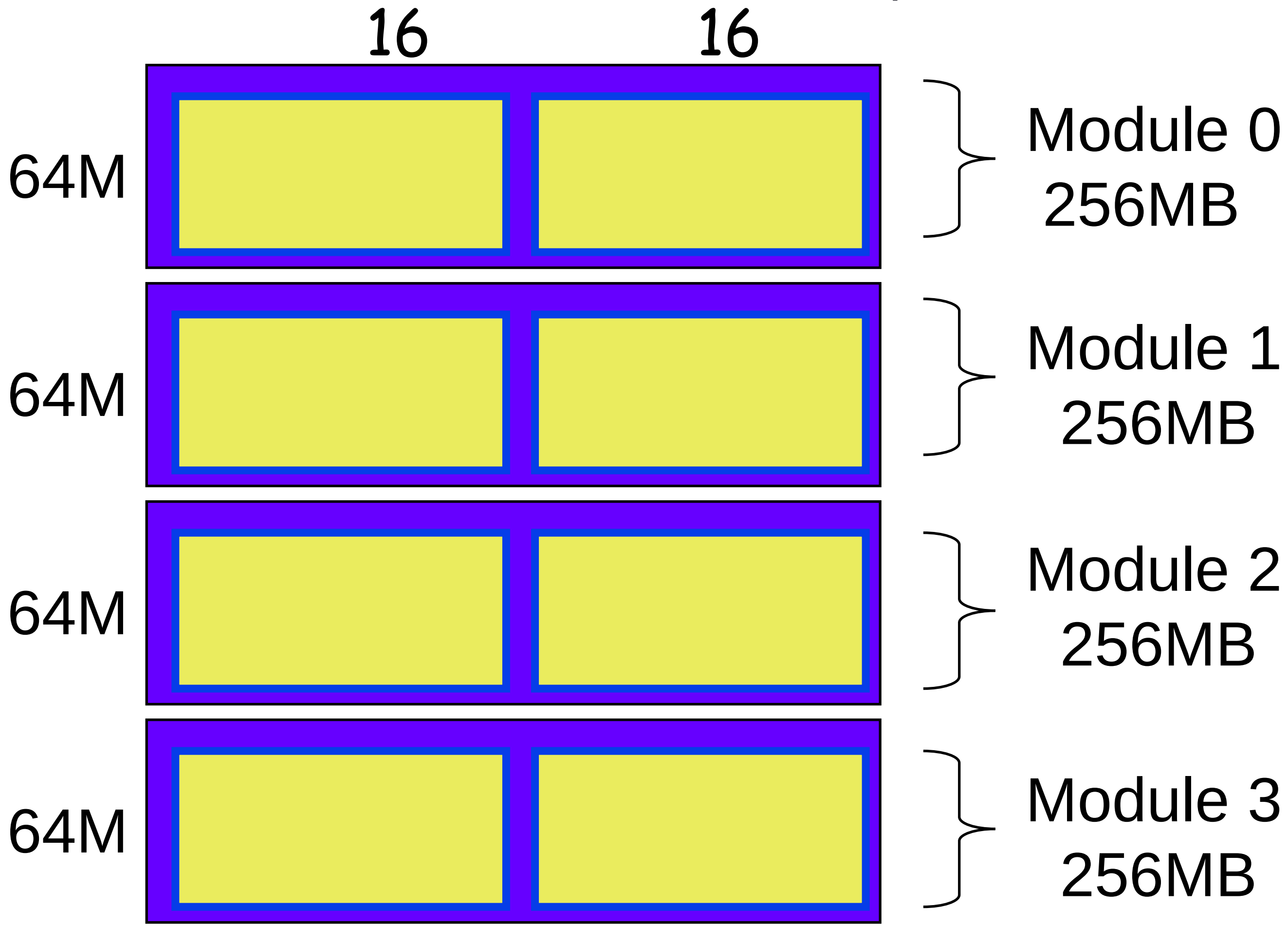
1GB (256M x 32-bit) Memory

- Two 512MB memory modules
  - Each module has four 128M x 8-bit RAM Chips



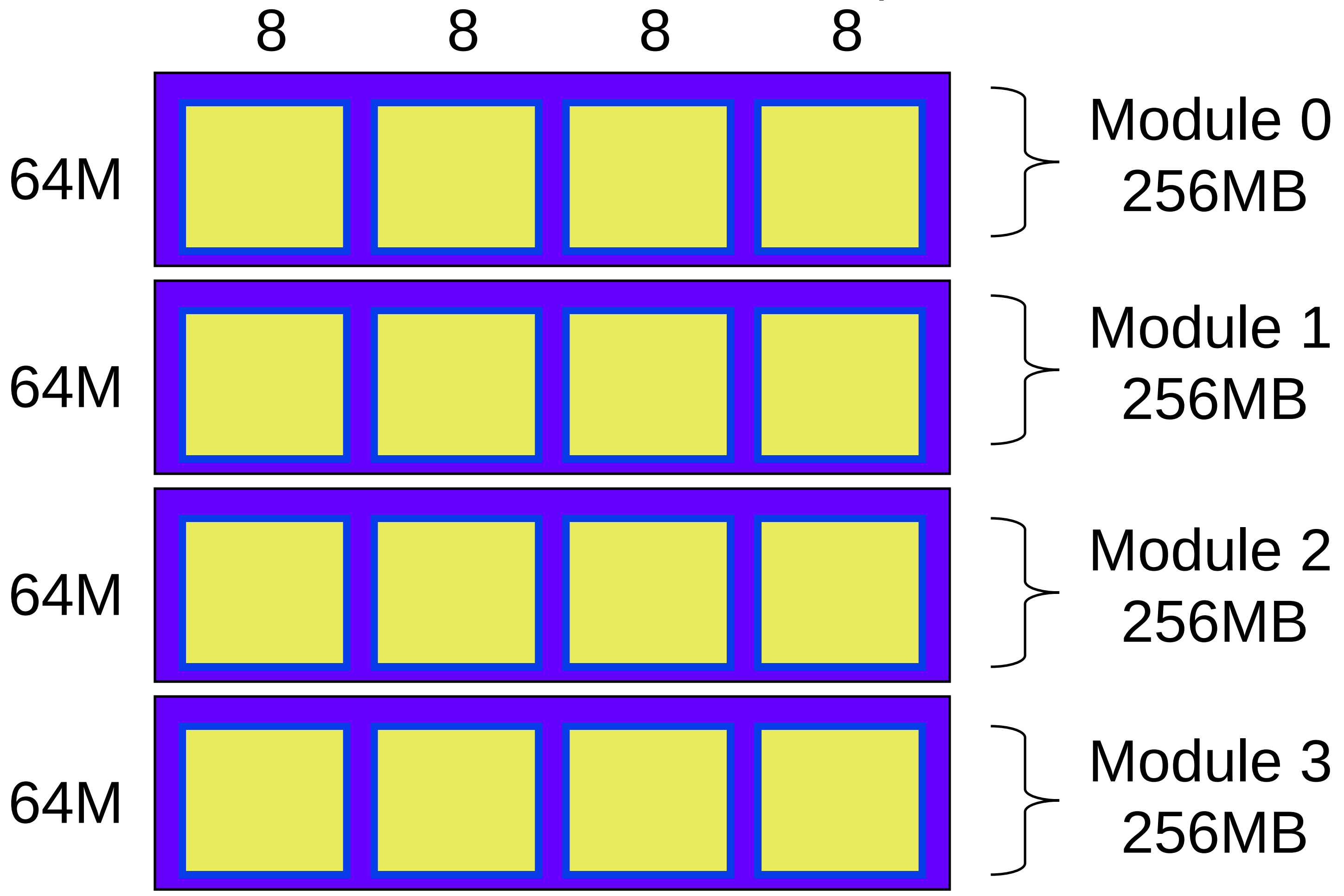
1GB (256M x 32-bit) Memory

- Four 256MB memory modules
  - Each module has two 64M x 16-bit RAM Chips



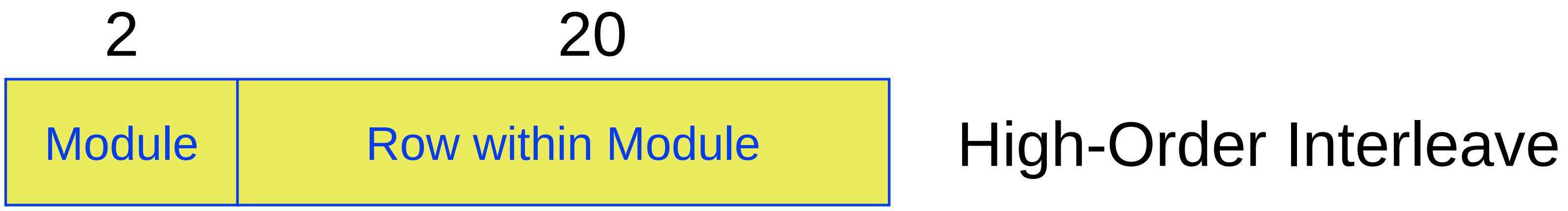
1GB (256M x 32-bit) Memory

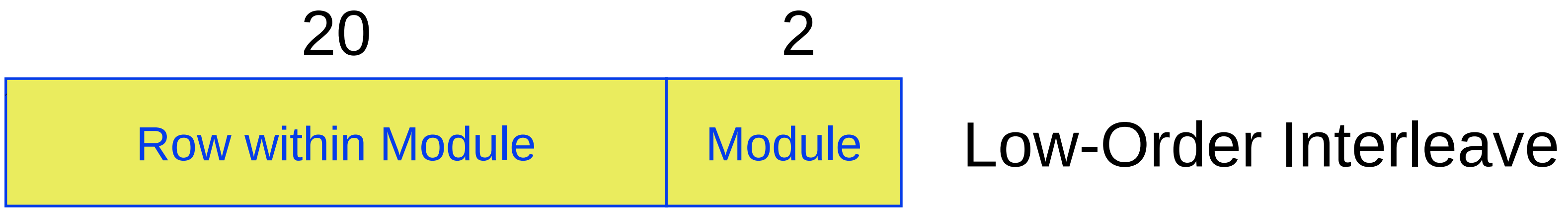
- Four 256MB memory modules
  - Each module has four 64M x 8-bit RAM Chips



Memory Interleaving

- Example:
  - Memory = 4M words, each word = 32-bits
  - Built with 4 x 1M x 32-bit memory modules
  - For 4M words we need 22 bits for an address
  - 22 bits = 2 bits (to select Modules) + 20 bits (to select row within Module)





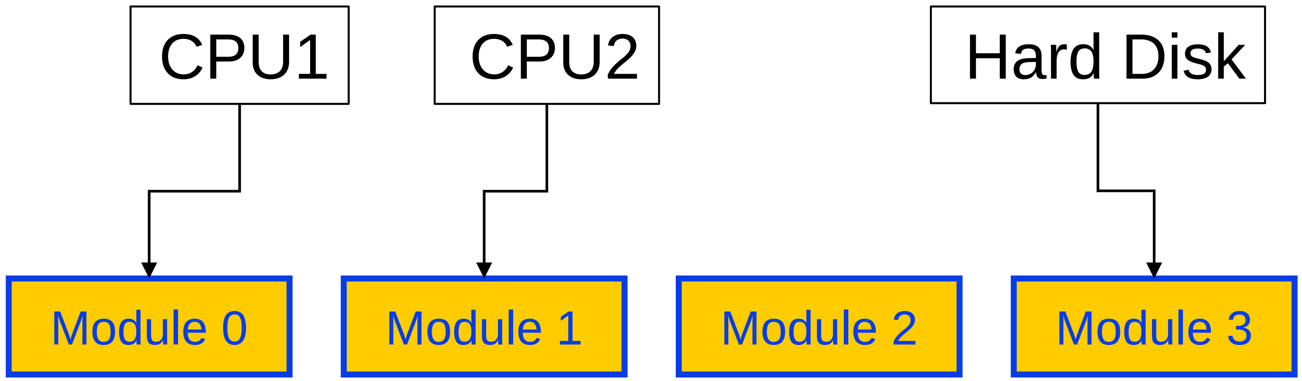
High-Order Interleave

Address Decimal	Address Binary							
0	00	0000	0000	0000	0000	0000	Module=0	Row=0
1	00	0000	0000	0000	0000	0001	Module=0	Row=1
2	00	0000	0000	0000	0000	0010	Module=0	Row=2
3	00	0000	0000	0000	0000	0011	Module=0	Row=3
4	00	0000	0000	0000	0000	0100	Module=0	Row=4
5	00	0000	0000	0000	0000	0101	Module=0	Row=5
…								
$2^{20}-1$	00	1111	1111	1111	1111	1111	Module=0	Row=$2^{20}-1$
2^{20}	01	0000	0000	0000	0000	0000	Module=1	Row=0
$2^{20}+1$	01	0000	0000	0000	0000	0001	Module=1	Row=1

High-Order Interleave

- Good if Modules can be accessed independently by different units, e.g. by the CPU and a Hard Disk (or a second CPU) AND the units use different Modules

- Parallel operation ➔ Higher Performance

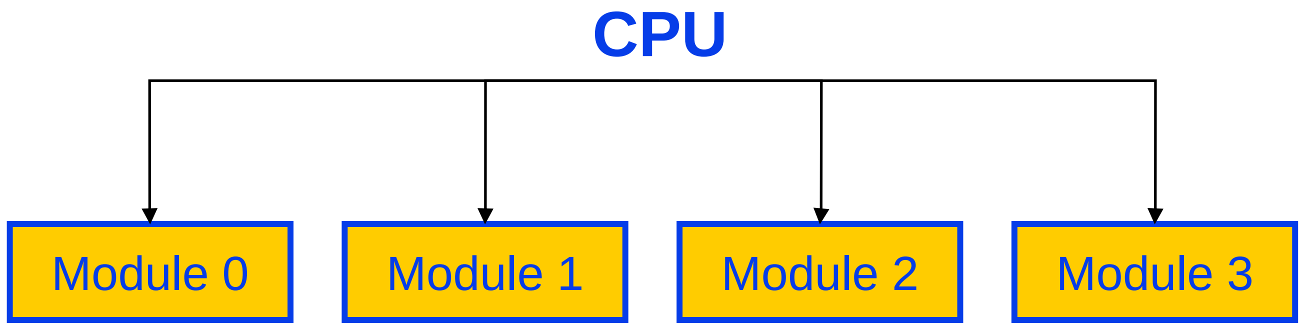


Low-Order Interleave

Address Decimal	Address Binary		
0	00 0000 0000 0000 0000 0000	Module=0	Row=0
1	00 0000 0000 0000 0000 0001	Module=1	Row=0
2	00 0000 0000 0000 0000 0010	Module=2	Row=0
3	00 0000 0000 0000 0000 0011	Module=3	Row=0
4	00 0000 0000 0000 0000 0100	Module=0	Row=1
5	00 0000 0000 0000 0000 0101	Module=1	Row=1
...			
$2^{20}-1$	00 1111 1111 1111 1111 1111	Module=3	Row=$2^{18}-1$
2^{20}	01 0000 0000 0000 0000 0000	Module=0	Row=2^{18}
$2^{20}+1$	01 0000 0000 0000 0000 0001	Module=1	Row=2^{18}
...			

Low-Order Interleave

- Good if the CPU (or other unit) can request multiple adjacent memory locations



- Since adjacent memory locations lie in different Modules an "advanced" memory system can perform the accesses in parallel
  - Such adjacent accesses often occur in practice, e.g.

    i. Elements in an array, e.g.. Array[N], Array[N+1], Array[N+2], ...
    ii. Instructions in a Programs, InstructionN, InstructionN+1,...

- In the above situations, an "advanced" CPU can pre-fetch the adjacent memory locations ➔ higher performance