MEMORY ORGANIZATION

- Memory Hierarchy
- Main Memory
- Cache Memory
- Virtual Memory
Memory Hierarchy is to obtain the highest possible access speed while minimizing the total cost of the memory system.
Characteristics

- Location
- Capacity
- Unit of transfer
- Access method
- Performance
- Physical type
- Physical characteristics
- Organisation
<table>
<thead>
<tr>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
</tr>
<tr>
<td>Internal</td>
</tr>
<tr>
<td>External</td>
</tr>
</tbody>
</table>
• **Word size**
  – The natural unit of organisation

• **Number of words**
  – or Bytes
Unit of Transfer

- **Internal**
  - Usually governed by data bus width

- **External**
  - Usually a block which is much larger than a word

- **Addressable unit**
  - Smallest location which can be uniquely addressed
  - Word internally
  - Cluster on M$ disks
Access Methods (1)

• **Sequential**
  – Start at the beginning and read through in order
  – Access time depends on location of data and previous location
  – e.g. tape

• **Direct**
  – Individual blocks have unique address
  – Access is by jumping to vicinity plus sequential search
  – Access time depends on location and previous location
  – e.g. disk
Access Methods (2)

- **Random**
  - Individual addresses identify locations exactly
  - Access time is independent of location or previous access
  - e.g. RAM

- **Associative**
  - Data is located by a comparison with contents of a portion of the store
  - Access time is independent of location or previous access
  - e.g. cache
Memory Hierarchy

- **Registers**
  - In CPU

- **Internal or Main memory**
  - May include one or more levels of cache
  - “RAM”

- **External memory**
  - Backing store
Memory Hierarchy

- **Access Times**
  - 1ns → 2ns
  - 3ns → 10ns
  - 25ns → 50ns
  - 30ns → 90ns
  - 5ms → 20ms
  - 100ms → 5s *
  - 10s → 3m *

- **More Costly**
  - Registers
  - Level 1 Cache
  - Level 2 Cache
  - Main Memory
  - Fixed Rigid Disk
  - Optical Disk (Jukeboxes)
  - Magnetic Tape (Robotic Libraries)

- **Less Costly**
  - System
  - Online
  - Near Line
  - Offline

* If volume is mounted.
Performance

• **Access time**
  – Time between presenting the address and getting the valid data

• **Memory Cycle time**
  – Time may be required for the memory to “recover” before next access
  – Cycle time is access + recovery

• **Transfer Rate**
  – Rate at which data can be moved
Physical Types

- Semiconductor
  - RAM
- Magnetic
  - Disk & Tape
- Optical
  - CD & DVD
- Others
  - Bubble
  - Hologram
## Physical Characteristics

- Decay
- Volatility
- Erasable
- Power consumption
Organisation

- Physical arrangement of bits into words
- Not always obvious
- e.g. interleaved
## Semiconductor Memory Types

<table>
<thead>
<tr>
<th>Memory Type</th>
<th>Category</th>
<th>Erasure</th>
<th>Write Mechanism</th>
<th>Volatility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random-access memory (RAM)</td>
<td>Read-write memory</td>
<td>Electrically, byte-level</td>
<td>Electrically</td>
<td>Volatile</td>
</tr>
<tr>
<td>Read-only memory (ROM)</td>
<td>Read-only memory</td>
<td>Not possible</td>
<td>Masks</td>
<td></td>
</tr>
<tr>
<td>Programmable ROM (PROM)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Erasable PROM (EPROM)</td>
<td>Read-mostly memory</td>
<td>UV light, chip-level</td>
<td>Electrically</td>
<td>Nonvolatile</td>
</tr>
<tr>
<td>Electrically Erasable PROM (EEPROM)</td>
<td></td>
<td>Electrically, byte-level</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flash memory</td>
<td></td>
<td>Electrically, block-level</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Semiconductor Memory

• RAM
  – Misnamed as all semiconductor memory is random access
  – Read/Write
  – Volatile
  – Temporary storage
  – Static or dynamic
Read Only Memory (ROM)

• Permanent storage
  – Nonvolatile
• Microprogramming (see later)
• Library subroutines
• Systems programs (BIOS)
• Function tables
Types of ROM

• **Written during manufacture**
  – Very expensive for small runs

• **Programmable (once)**
  – PROM
  – Needs special equipment to program

• **Read “mostly”**
  – Erasable Programmable (EPROM)
    » Erased by UV
  – Electrically Erasable (EEPROM)
    » Takes much longer to write than read
  – Flash memory
    » Erase whole memory electrically
RAM and ROM Chips

Typical RAM chip

<table>
<thead>
<tr>
<th>CS1</th>
<th>CS2</th>
<th>RD</th>
<th>WR</th>
<th>Memory function</th>
<th>State of data bus</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>Inhibit</td>
<td>High-impedence</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>Inhibit</td>
<td>High-impedence</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Inhibit</td>
<td>High-impedence</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Write</td>
<td>Input data to RAM</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>Read</td>
<td>Output data from RAM</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>Inhibit</td>
<td>High-impedence</td>
</tr>
</tbody>
</table>

Typical ROM chip
• Sequential circuits all depend upon the presence of memory.
  – A flip-flop can store one bit of information.
  – A register can store a single “word,” typically 32-64 bits.

• Random access memory, or RAM, allows us to store even larger amounts of data. Today we’ll see:
  – The basic interface to memory.
  – How you can implement static RAM chips hierarchically.

• This is the last piece we need to put together a computer!
Introduction to RAM

- **Random-access memory**, or RAM, provides large quantities of temporary storage in a computer system.
- Remember the basic capabilities of a memory:
  - It should be able to store a value.
  - You should be able to read the value that was saved.
  - You should be able to change the stored value.
- A RAM is similar, except that it can store *many* values.
  - An *address* will specify which memory value we’re interested in.
  - Each value can be a multiple-bit *word* (e.g., 32 bits).
- We’ll refine the memory properties as follows:

A RAM should be able to:
- Store many words, one per address
- Read the word that was saved at a particular address
- Change the word that’s saved at a particular address
You can think of computer memory as being one big array of data.
- The address serves as an array index.
- Each address refers to one word of data.

You can read or modify the data at any given memory address, just like you can read or modify the contents of an array at any given index.

If you’ve worked with pointers in C or C++, then you’ve already worked with memory addresses.
• This block diagram introduces the main interface to RAM.
  – A Chip Select, CS, enables or disables the RAM.
  – ADRS specifies the address or location to read from or write to.
  – WR selects between reading from or writing to the memory.
    ‣ To read from memory, WR should be set to 0. OUT will be the n-bit value stored at ADRS.
    ‣ To write to memory, we set WR = 1. DATA is the n-bit value to save in memory.
• This interface makes it easy to combine RAMs together, as we’ll see.
Memory sizes

- We refer to this as a $2^k \times n$ memory.
  - There are $k$ address lines, which can specify one of $2^k$ addresses.
  - Each address contains an $n$-bit word.

- For example, a $2^{24} \times 16$ RAM contains $2^{24} = 16M$ words, each 16 bits long.
  - The RAM would need 24 address lines.
  - The total storage capacity is $2^{24} \times 16 = 2^{28}$ bits.
### Typical memory sizes

#### Some typical memory capacities:
- PCs usually come with 512MB – 2GB RAM.
- PDAs have 16-64MB of memory.
- Digital cameras and MP3 players can have 32MB-8GB or more of onboard storage.

#### Many operating systems implement virtual memory, which makes the memory seem larger than it really is.
- Most systems allow up to 32-bit addresses. This works out to $2^{32}$, or about four billion, different possible addresses.
- With a data size of one byte, the result is apparently a 4GB memory!
- The operating system uses hard disk space as a substitute for “real” memory.

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td></td>
</tr>
<tr>
<td>00000002</td>
<td></td>
</tr>
<tr>
<td>FFFFFFFD</td>
<td></td>
</tr>
<tr>
<td>FFFFFFFE</td>
<td></td>
</tr>
<tr>
<td>FFFFFFFF</td>
<td></td>
</tr>
</tbody>
</table>
Reading RAM

- To *read* from this RAM, the controlling circuit must:
  - Enable the chip by ensuring $CS = 1$.
  - Select the read operation, by setting $WR = 0$.
  - Send the desired address to the ADRS input.
  - The contents of that address appear on OUT after a little while.

- Notice that the DATA input is unused for read operations.
Writing RAM

• To write to this RAM, you need to:
  – Enable the chip by setting CS = 1.
  – Select the write operation, by setting WR = 1.
  – Send the desired address to the ADRS input.
  – Send the word to store to the DATA input.

• The output OUT is not needed for memory write operations.
MAIN MEMORY

RAM and ROM Chips
Typical RAM chip

<table>
<thead>
<tr>
<th>CS1</th>
<th>CS2</th>
<th>RD</th>
<th>WR</th>
<th>Memory function</th>
<th>State of data bus</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>Inhibit</td>
<td>High-impedence</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>Inhibit</td>
<td>High-impedence</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Inhibit</td>
<td>High-impedence</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Write</td>
<td>Input data to RAM</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>Read</td>
<td>Output data from RAM</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>Inhibit</td>
<td>High-impedence</td>
</tr>
</tbody>
</table>

Typical ROM chip

<table>
<thead>
<tr>
<th>CS1</th>
<th>CS2</th>
<th>AD 9</th>
<th>Memory function</th>
<th>State of data bus</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>512</td>
<td>ROM</td>
<td>High-impedence</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>512</td>
<td>ROM</td>
<td>High-impedence</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>512</td>
<td>ROM</td>
<td>High-impedence</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>512</td>
<td>ROM</td>
<td>High-impedence</td>
</tr>
</tbody>
</table>
Memory Address Map

Address space assignment to each memory chip

Example: 512 bytes RAM and 512 bytes ROM

<table>
<thead>
<tr>
<th>Component</th>
<th>Hexa address</th>
<th>Address bus</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM 1</td>
<td>0000 - 007F</td>
<td>0 0 0 x x x x x x x x</td>
</tr>
<tr>
<td>RAM 2</td>
<td>0080 - 00FF</td>
<td>0 0 1 x x x x x x x x</td>
</tr>
<tr>
<td>RAM 3</td>
<td>0100 - 017F</td>
<td>0 1 0 x x x x x x x x</td>
</tr>
<tr>
<td>RAM 4</td>
<td>0180 - 01FF</td>
<td>0 1 1 x x x x x x x x</td>
</tr>
<tr>
<td>ROM</td>
<td>0200 - 03FF</td>
<td>1 x x x x x x x x x x</td>
</tr>
</tbody>
</table>

Memory Connection to CPU

- RAM and ROM chips are connected to a CPU through the data and address buses
- The low-order lines in the address bus select the byte within the chips and other lines in the address bus select a particular chip through its chip select inputs
CONNECTION OF MEMORY TO CPU

CPU

Address bus
16-11  10  9  8  7-1  RD  WR

Decoder
3  2  1  0

CS1
CS2
RD
WR
AD7

128 x 8
RAM 1

CS1
CS2
RD
WR
AD7

128 x 8
RAM 2

CS1
CS2
RD
WR
AD7

128 x 8
RAM 3

CS1
CS2
RD
WR
AD7

128 x 8
RAM 4

CS1
CS2
RD
WR
AD7

512 x 8
ROM

Main Memory
Making Larger Memories

Once we have decided on the memory chip configuration, it is straightforward to determine the number of chips and the organization of the memory unit. Let us assume that we are using $D \times W$ chips to build an $M \times N$ memory. Of course, we want to make sure that $D \leq M$ and $W \leq N$.

Number of chips required $= \frac{M \times N}{D \times W}$,

Number of rows $= \frac{M}{D}$,

Number of columns $= \frac{N}{W}$.
Bigger RAMs from smaller RAMs

- We can use small RAMs as building blocks for making larger memories.
- As an example, suppose we have some 64K x 8 RAMs to start with:
  - 64K = $2^6 \times 2^{10} = 2^{16}$, so there are 16 address lines.
  - There are 8 data lines.
Making a larger memory

- We can put ?? 64K x 8 chips together to make a 256K x 8 memory.
- For 256K words, we need ?? address lines.
Making a larger memory

- We can put four 64K x 8 chips together to make a 256K x 8 memory.
- For 256K words, we need 18 address lines.
  - The two most significant address lines go to the decoder, which selects one of the four 64K x 8 RAM chips.
  - The other 16 address lines are shared by the 64K x 8 chips.
- The 64K x 8 chips also share WR and DATA inputs.
- This assumes the 64K x 8 chips have three-state outputs.
Analyzing the 256K x 8 RAM

- There are 256K words of memory, spread out among the four smaller 64K x 8 RAM chips.
- When the two most significant bits of the address are 00, the bottom RAM chip is selected. It holds data for the first 64K addresses.
- The next chip up is enabled when the address starts with 01. It holds data for the second 64K addresses.
- The third chip up holds data for the next 64K addresses.
- The final chip contains the data of the final 64K addresses.
Address ranges

<table>
<thead>
<tr>
<th>11 1111 1111 1111 1111 (0x3ffff)</th>
<th>to</th>
</tr>
</thead>
<tbody>
<tr>
<td>11 0000 0000 0000 0000 (0x30000)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>10 1111 1111 1111 1111 (0x2ffff)</th>
<th>to</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 0000 0000 0000 0000 (0x20000)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>01 1111 1111 1111 1111 (0x1ffff)</th>
<th>to</th>
</tr>
</thead>
<tbody>
<tr>
<td>01 0000 0000 0000 0000 (0x10000)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>00 1111 1111 1111 1111 (0x0ffff)</th>
<th>to</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0000 0000 0000 0000 (0x00000)</td>
<td></td>
</tr>
</tbody>
</table>
• Using the CS lines, we can make larger memories from smaller ones by tying all address, data, and R/W lines in parallel, and using the decoded higher order address bits to control CS.

• Using the 4-Word by 1-Bit memory from before, we construct a 16-Word by 1-Bit memory. ⇒
Making a wider memory

- You can also combine smaller chips to make wider memories, with the same number of addresses but more bits per word.
- How do we create a 64K x 16 RAM from two 64K x 8 chips?
Making a wider memory

- You can also combine smaller chips to make wider memories, with the same number of addresses but more bits per word.
- Here is a 64K x 16 RAM, created from two 64K x 8 chips.
  - The left chip contains the most significant 8 bits of the data.
  - The right chip contains the lower 8 bits of the data.
Making Wider Memories

- To construct wider memories from narrow ones, we tie the address and control lines in parallel and keep the data lines separate.

- For example, to make a 4-word by 4-bit memory from 4, 4-word by 1-bit memories

- Note: Both 16x1 and 4x4 memories take 4-chips and hold 16 bits of data.
12-1. a. How many $128 \times 8$ RAM chips are needed to provide a memory capacity of 2048 bytes?
   
   b. How many lines of the address bus must be used to access 2048 bytes of memory? How many of these lines will be common to all chips?
   
   c. How many lines must be decoded for chip select? Specify the size of the decoders.

\[
\frac{2048}{128} = 16 \text{ chips}
\]

(b) \(2048 = 2^{11}\) / \(128 = 2^7\)

11 lines to address 2048 bytes
7 lines to address each chip

4 lines to decoder for selecting 16 chips

(C) 4x16 decoder
12-2. A computer uses RAM chips of $1024 \times 1$ capacity.

a. How many chips are needed, and how should their address lines be connected to provide a memory capacity of 1024 bytes?

b. How many chips are needed to provide a memory capacity of 16K bytes? Explain in words how the chips are to be connected to the address bus.

12-2

(a) 8 chips are needed with address lines connected in parallel,
(b) $16 \times 8 = 128$ chips. Use 14 address lines ($16k = 2^{14}$)
10 lines specify the chip address
4 lines are decoded into 16 chip-select inputs.
12-3. A ROM chip of $1024 \times 8$ bits has four select inputs and operates from a 5-volt power supply. How many pins are needed for the IC package? Draw a block diagram and label all input and output terminals in the ROM.

12-3

10 pins for inputs, 4 for chip-select, 8 for outputs, 2 for power.
Total of 24 pins.
Extend the memory system of Fig. 12-4 to 4096 bytes of RAM and 4096 bytes of ROM. List the memory-address map and indicate what size decoders are needed.

\[ \frac{4096}{128} = 32 \text{ RAM chips}; \quad \frac{4096}{512} = 8 \text{ ROM chips.} \]

\[ 4096 = 2^{12} \quad \text{There are 12 common address lines + 1 line to select between RAM and ROM.} \]

<table>
<thead>
<tr>
<th>Component</th>
<th>Address</th>
<th>16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td>0000-0FFF</td>
<td>0 0 0 0 \xrightarrow{\text{3x8 decoder}} xxx xxxxxx</td>
</tr>
<tr>
<td>ROM</td>
<td>1000-1FFF</td>
<td>0 0 0 1 \xleftarrow{\text{5x32 decoder}} xx xxxxxx xxxxxx</td>
</tr>
</tbody>
</table>

to CS2 \xrightarrow{\text{decoder}}
A computer employs RAM chips of $256 \times 8$ and ROM chips of $1024 \times 8$. The computer system needs 2K bytes of RAM, 4K bytes of ROM, and four interface units, each with four registers. A memory-mapped I/O configuration is used. The two highest-order bits of the address bus are assigned 00 for RAM, 01 for ROM, and 10 for interface registers.

a. How many RAM and ROM chips are needed?
b. Draw a memory-address map for the system.
c. Give the address range in hexadecimal for RAM, ROM, and interface.

\[
\begin{array}{c|c|c}
\text{Component} & \text{Address} & \text{Address Range} \\
\hline
\text{RAM} & 0000-07FF & 0000-07FF \\
\text{ROM} & 4000-4FFF & 0100-0FFF \\
\text{Interface} & 8000-800F & 1000-0FFF \\
\end{array}
\]
12-6. An 8-bit computer has a 16-bit address bus. The first 15 lines of the address are used to select a bank of 32K bytes of memory. The high-order bit of the address is used to select a register which receives the contents of the data bus. Explain how this configuration can be used to extend the memory capacity of the system to eight banks of 32K bytes each, for a total of 256K bytes of memory.

The processor selects the external register with an address 8000 hexadecimal. Each bank of 32K bytes are selected by addresses 0000–7FFF. The processor loads an 8-bit number into the register with a single 1 and 7 0's. Each output of the register selects one of the 8 banks of 32K bytes through a chip-select input. A memory bank can be changed by changing the number in the register.
12-7. A magnetic disk system has the following parameters:

\[ T_s = \text{average time to position the magnetic head over a track} \]

\[ R = \text{rotation speed of disk in revolutions per second} \]

\[ N_t = \text{number of bits per track} \]

\[ N_s = \text{number of bits per sector} \]

Calculate the average time \( T_a \) that it will take to read one sector.

\[
T_a = T_s + \frac{1}{2}R + \frac{N_s}{N_t} \times \frac{1}{R}
\]
12-8. What is the transfer rate of an eight-track magnetic tape whose speed is 120 inches per second and whose density is 1600 bits per inch?

An eight-track tape reads 8 bits (one character) at the same time. Transfer rate $= 1600 \times 120 = 192,000$ characters/s

12-9
• Cache memory has the fastest access time after registers.

<table>
<thead>
<tr>
<th>Memory Type</th>
<th>Access Time</th>
<th>Cost /MB</th>
<th>Typical Amount Used</th>
<th>Typical Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>1ns</td>
<td>High</td>
<td>1KB</td>
<td>–</td>
</tr>
<tr>
<td>Cache</td>
<td>5-20 ns</td>
<td>$100</td>
<td>1MB</td>
<td>$100</td>
</tr>
<tr>
<td>Main memory</td>
<td>60-80ns</td>
<td>$1.10</td>
<td>64 MB</td>
<td>$70</td>
</tr>
<tr>
<td>Disk memory</td>
<td>10 ms</td>
<td>$0.05</td>
<td>4 GB</td>
<td>$200</td>
</tr>
</tbody>
</table>
Why is cache memory needed?

- When a program references a memory location, it is likely to reference that same memory location again soon.
- A memory location that is near a recently referenced location is more likely to be referenced than a memory location that is farther away.
Why is cache memory needed

- A small but fast cache memory, in which the contents of the most commonly accessed locations are maintained, can be placed between the CPU and the main memory.
- When a program executes, the cache memory is searched first.
Why is cache memory fast?

- Faster electronics used
- A cache memory has fewer locations than a main memory, which reduces the access time
- The cache is placed both physically closer and logically closer to the CPU than the main memory
Why is cache memory fast

- This cacheless computer usually needs a few bus cycles to synchronize the CPU with the bus.
- A cache memory can be positioned closer to the CPU.
PERFORMANCE OF CACHE

Memory Access

All the memory accesses are directed first to Cache
If the word is in Cache; Access cache to provide it to CPU
If the word is not in Cache; Bring a block (or a line) including that word to replace a block now in Cache

- How can we know if the word that is required is there?
- If a new block is to replace one of the old blocks, which one should we choose?

Performance of Cache Memory System

Hit Ratio - % of memory accesses satisfied by Cache memory system
T_e: Effective memory access time in Cache memory system
T_c: Cache access time
T_m: Main memory access time

\[ T_e = h T_c + (1 - h) T_m \]

Example: \( T_c = 0.4 \mu s, T_m = 1.2 \mu s, h = 0.85 \)
\[ T_e = 0.85 \times 0.4 + (1 - 0.85) \times 1.2 = 0.52 \mu s \]
Data Reading from Memory with Cache

1. Begin
2. Processor outputs memory address A
3. Find memory block B that contains address A
4. Use mapping function
5. Is block B in cache?
   - Yes: Supply the word at address A to processor
   - No: Initiate access to memory block B
6. Assign cache line C for memory block B
   - Use placement policy
   - Is cache line C free?
     - Yes: Replace a cache line to make room for block B
     - No: Supply the word at address A to processor
     - Yes: Load memory block B into the cache line
Cache Read Operation

(b) Read miss
Cache Write Operation

(a) Write hit
Cache Write Operation

- **CPU**
- **Cache**
- **Address and data buffers**
- **Main memory**

**Address bus**

**Data bus**

**System bus**

Buffers enabled

(b) Write miss
Example

- The access time of a cache memory is 50ns and that of main memory is 500 ns. It is estimated that 80% of the main memory requests are for read and the remaining are for write. The hit ratio for read access only is 0.9 and a write through policy is used
  - Determine the average access time considering only read cycles
  - What is the average time if write requests are also considered

\[
T = h_t c + (1-h) t_m
= 0.9 \times 50 + 0.1 \times 500
= 45 + 50 = 95 \text{ ns}
\]

\[
T_{r/w} = \text{read prob} \times t_{\text{read}} + (1 - \text{read prob}) \times t_{\text{write}}
\]

Read request prob = 0.8
Write request prob = 0.2
\[t_{\text{read}} = 95\text{ns}\]
\[t_{\text{write}} = 500\text{ns} \text{ (because both memories are updated at the same time)}\]

\[
t_{r/w} = 0.8 \times 95 + 0.2 \times 500
= 76 + 100 = 176 \text{ ns}
\]
Cache Performance

- Cache read and write policies

Cache Read
- Data is in the cache
  - Forward to CPU.
- Data is not in the cache
  - Load Through: Forward the word as cache line is filled,
    -or-
    - Fill cache line and forward word.

Cache Write
- Data is in the cache
  - Write Through: Write data to both cache and main memory,
    -or-
    - Write Back: Write data to cache only. Defer main memory write until block is flushed.
- Data is not in the cache
  - Write Allocate: Bring line into cache, then update it,
    -or-
    - Write No-Allocate: Update main memory only.
CACHE WRITE

Write Through

When writing into memory

If Hit, both Cache and memory is written in parallel
If Miss, Memory is written
For a read miss, missing block may be overloaded onto a cache block

Memory is always updated
-> Important when CPU and DMA I/O are both executing

Slow, due to the memory access time

Write-Back (Copy-Back)

When writing into memory

If Hit, only Cache is written
If Miss, missing block is brought to Cache and write into Cache
For a read miss, candidate block must be written back to the memory

Memory is not up-to-date, i.e., the same item in Cache and memory may have different value
Write Back Policy

Write hit operation in a write-back cache.
REPLACEMENT POLICIES

- **LRU**
  - the cache block that has been recently used the least is selected for replacement

- **LFU**
  - replaces the block in the set that has been referenced the least number of times.
  - Implementation of LFU requires a counter associated with each cache line.

- **FIFO**
  - replaces the block that has been in the set for the longest time.
  - FIFO policy can be implemented using a circular buffer.

- **RANDOM**
  - random selection of a cache block for replacement is done based on the output of the random number generator at the time of replacement.
  - This technique is simple and does not require much additional overhead

- **OPTIMAL**
  - The optimal replacement policy is not practical, but is used for comparison purposes to determine how effective other replacement policies are to the best possible.
  - The optimal replacement policy is determined only after a program has already executed, and so it is of little help to a running program
Many different block replacement policies are available

**LRU (Least Recently Used)** is most easy to implement

Cache word = (tag 0, data 0, \( U_0 \));(tag 1, data 1, \( U_1 \)), \( U_i = 0 \) or \( 1 \) (binary)

Implementation of LRU in the Set Associative Mapping with set size = 2

**Modifications**

Initially all \( U_0 = U_1 = 1 \)

When Hit to (tag 0, data 0, \( U_0 \)), \( U_1 \leftarrow 1 \) (least recently used)
(When Hit to (tag 1, data 1, \( U_1 \)), \( U_0 \leftarrow 1 \) (least recently used))

When Miss, find the least recently used one (\( U_i = 1 \))

If \( U_0 = 1 \), and \( U_1 = 0 \), then replace (tag 0, data 0)

\[
M[\text{tag 0, INDEX}] \leftarrow \text{Cache[INDEX]}(\text{data 0})
\]

\[
\text{Cache[INDEX]}(\text{tag 0, data 0, U0}) \leftarrow (\text{TAG}, M[\text{TAG,INDEX}], 0); \ U_1 \leftarrow 1
\]

If \( U_0 = 0 \), and \( U_1 = 1 \), then replace (tag 1, data 1)

Similar to above; \( U_0 \leftarrow 1 \)

If \( U_0 = U_1 = 0 \), this condition does not exist

If \( U_0 = U_1 = 1 \), Both of them are candidates,

Take arbitrary selection
The mapping function determines how memory blocks are mapped to cache lines. This function sometimes is also referred to as the hashing function.

This address conversion is done by giving special significance to the bits in the main memory address. We first divide the bits into distinct groups we call fields. Depending on the mapping scheme, we may have two or three fields. How we use these fields depends on the particular mapping scheme being used.

Main memory and cache are both divided into the same size blocks (the size of these blocks varies). When a memory address is generated, cache is searched first to see if the required word exists there. When the requested word is not found in cache, the entire main memory block in which the word resides is loaded into cache.

1. Direct Mapping: Specifies a single cache line for each memory block;
2. Set-Associative Mapping: Specifies a set of cache lines for each memory block;
3. Associative Mapping: No restrictions; any cache line can be used for any memory block.
Cache mapping

Commonly used methods:
• Associative Mapped Cache
• Direct-Mapped Cache
• Set-Associative Mapped Cache
Direct Mapping

- Direct mapped cache assigns cache mappings using a modular approach. Because there are more main memory blocks than there are cache blocks, it should be clear that main memory blocks compete for cache locations. Direct mapping maps block X of main memory to block Y of cache, mod N, where N is the total number of blocks in cache. For example, if cache contains 10 blocks, then main memory block 0 maps to cache block 0, main memory block 1 maps to cache block 1, . . . , main memory block 9 maps to cache block 9, and main memory block 10 maps to cache block 0.

- We are using a block size of 4 bytes so, no. of cache lines is 16/4=4.
  - E.g. memory block 11 is mapped to cache line 3 (11 mod 4)
Figure 17.8 A direct mapping cache example: The main memory and cache are assumed to be 64 bytes and 16 bytes, respectively. The mapping of memory blocks to cache lines is shown by appropriate shading. For example, main memory blocks 0, 4, 8, and 12 are mapped to cache line 0.
Mapping Function

(a) Address partition

<table>
<thead>
<tr>
<th>Valid bit</th>
<th>Cache tag</th>
<th>Cache data</th>
<th>Cache line</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>???</td>
<td>???</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>Valid tag</td>
<td>4-bytes of valid cache data</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>Valid tag</td>
<td>4-bytes of valid cache data</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>Valid tag</td>
<td>4-bytes of valid cache data</td>
<td>0</td>
</tr>
</tbody>
</table>

(b) Cache memory details
• Cache implementations maintain three pieces of information:
  – Cache Data: These are the actual data copied from the mapped memory block.
  – Cache Tag: Since more than one memory block is mapped to a cache line, we need to store the tag field for each cache line in order to know the address of the actual block stored. The size of the tag field $t$ is equal to $\log_2 N$, where $N$ is the number blocks mapped to each cache line.
  – Valid Bit: We also need to store information on whether a cache line contains a valid block. This information is necessary so that we do not treat garbage as valid data. A single bit is sufficient to maintain this information. We use 1 to represent a valid cache line and 0 for an invalid cache line.

• To find if a memory block is in the cache, use the modulo function to find the cache line for the block. Then check if the corresponding valid bit is 1. If so, compare the tag value stored at the cache line with the higher-order $t$ bits of the memory block address. If there is a match, the required block is in the cache; otherwise, there is a cache miss.
Direct Mapping

• Least significant \( b \) bits are used as the byte offset into a block. If the block size is \( B \) bytes, the number of bits for the byte offset is given by

\[
b = \log_2 B.
\]

• The next \( c \) bits are used to identify a cache line. If the number of cache lines is \( C \), we can get \( c \) as

\[
c = \log_2 C.
\]

• The remaining higher-order \( t \) bits are used as the cache tag. As we show next, cache memory would have to store this tag information for each cache line in order to identify the memory block stored in the cache line.
**Example**

**Example 1** Consider, for example, the case of a main memory consisting of 4K blocks, a cache memory consisting of 128 blocks, and a block size of 16 words.

- $B = \text{Block Size} = 16 \text{ words}$
- $C = \text{Cache Size} = 128 \text{ Blocks}$
- $M = \text{Main Memory Size} = 4K \text{ Blocks}$
- $\text{Main Memory Address length} = \log_2 (4K \times 16) = 16 \text{ bits}$

- Byte Offset $b = \log_2 B = \log_2 16 = \log_2 2^4 = 4 \text{ bits}$
- Cache Line # $c = \log_2 C = \log_2 128 = \log_2 2^7 = 7 \text{ bits}$
- Tag Bits $t = \text{Remaining Higher order bits} = 16 - (4 + 7) = 5 \text{ bits}$
- OR
- Tag bits $t = \log_2 (M/C) = \log_2 (2^2 \times 2^{10} / 2^7) = 5 \text{ bits}$
– **Example**: Suppose that the reference pattern of a program is such that it accesses the following sequence of blocks: 0, 4, 0, 8, 0, 8, 0, 4, 0, 4, 0, 4. Find the hit ratio with a direct-mapped cache of four cache lines.

> The three blocks accessed by the program—0, 4, and 8—are all mapped to cache line 0. When block 0 is accessed for the first time, it is placed in cache line 0. When the program accesses block 4 next, it replaces block 0 in cache line 0. You can continue this process and see that every block access leads to a cache miss. Thus the hit ratio is zero.

– This example describes the worst-case behavior, where each access results in a miss. This scenario where a cache line is replaced before it is referenced again is called cache thrashing.
Example: Suppose the sequence of main memory blocks accessed is 0, 7, 9, 10, 0, 7, 9, 10, 0, 7, 9, 10. Find the hit ratio in a direct-mapped cache with a cache size of four cache lines.

This reference pattern shows the best behavior. There will only be four cache misses. After these four blocks have been loaded into the cache, the remaining eight references can obtain data from the cache. Thus the hit ratio is 8/12 or about 67%. For this example, this is the maximum hit ratio one can obtain.
Fully Associative Mapping

- An incoming main memory block can be placed in any available cache block.
- This mapping does not impose any restriction on the placement of blocks in cache memory.
- This offers maximum flexibility
With associative mapping, we divide the address into two components:

- The least significant bits are used for byte offset as in the direct mapping scheme, if Block size is $B$ bytes;

$$b = \log_2 B.$$ 

- The remaining bits are used as the tag field.

(a) Address partition

(b) Cache memory details
Example

Example: Compute the above three parameters for a memory system having the following specification: size of the main memory is 4K blocks, size of the cache is 128 blocks, and the block size is 16 words. Assume that the system uses associative mapping.

- \( C = \) Cache Size = 128 Blocks
- \( M = \) Main Memory Size = 4K Blocks
- Main Memory Address length = \( \log_2 (4K \times 16) = 12 \) bits

- Byte Offset \( b = \log_2 B = \log_2 16 = \log_2 2^4 = 4 \) bits
- Tag Bits \( t = \) Remaining Higher order bits = \( 16 - 4 = 12 \) bits
• Example: Consider the reference pattern of Previous Example that accesses the sequence of blocks 0, 4, 0, 8, 0, 4, 0, 4, 0, 4, 0, 4. Assuming that the cache uses associative mapping, find the hit ratio with a cache size of four cache lines.

- In fully associative mapping, the incoming block can take any free cache line, we use a FIFO allocation. Block 0 is placed in cache line 0, block 4 is placed in the next available cache line, and so on. Thus, after three misses to load the blocks 0, 4, and 8 initially, the remaining nine references can be read from the cache. This gives us a hit ratio of 75%. This is in contrast to the hit ratio of 0% obtained with direct mapping. Because these three misses are compulsory misses, this is the highest hit ratio we can get for this reference pattern.
Disadvantages

• The major drawback is the location of a block in the cache.
• Since a block can be in any cache line, we have to search all tag fields in parallel to locate a block. This means we need hardware to do $2^c$ comparisons, where $c$ is the number of cache lines.
Set Associative Mapping

- Set-associative mapping is a compromise between direct and associative mapping. It divides the cache lines into disjoint sets. Mapping of a memory block is done as in direct mapping, except that it maps the block to a set of cache lines rather than to a single cache line. The block can be placed in any cache line within the assigned set as in the associative mapping. Set-associative mapping reduces the search complexity to the number of cache lines within a set. Typically, small set sizes—2, 4, or 8— are used.
Figure 17.13 Set-associative mapping function details: The main memory is 64 bytes and the cache is 16 bytes in size. Cache line size is 4 bytes. With these parameters, the set-associative function maps all even-numbered blocks to set 0 (shown as white) and all odd-numbered blocks to set 1 (shown as gray).
Set-associative mapping partitions the address into three components:

- The least significant bits $b$ specify the byte offset as in the other two mapping functions.
- The next $s$ bits identify a set. The number of bits for the $s$ field is given by $s = \log_2 S$ where $S$ is the number of sets.
- The remaining higher-order bits are used as the tag, as in the other two mappings.
Example

Compute the above three parameters (Word, Set, and Tag) for a memory system having the following specification: size of the main memory is 4K blocks, size of the cache is 128 blocks, and the block size is 16 words. Assume that the system uses set-associative mapping with four blocks per set.

\[ S = \frac{128}{4} = 32 \text{ sets.} \]

- B = Block Size = 16 words
- C = Cache Size = 128 Blocks
- M = Main Memory Size = 4K Blocks
- Main Memory Address length = \( \log_2 (4K \times 16) = 12 \text{ bits} \)
- Byte Offset \( b = \log_2 B = \log_2 16 = \log_2 2^4 = 4 \text{ bits} \)
- Set Field \( s = \log_2 S = \log_2 32 = \log_2 2^5 = 5 \text{ bits} \)
- Tag Bits \( t = \text{Remaining Higher order bits} = 16 - (4 + 5) = 7 \text{ bits} \)
Let us consider the reference pattern that accesses the sequence of blocks 0, 4, 0, 8, 0, 8, 0, 4, 0, 4, 0, 4.

Assuming that the set size is 2 AND cache size of four cache lines.

- Since the cache size is four lines and the set size is 2, we have two sets. Therefore, all blocks with an even number are mapped to set 0, and odd numbered blocks to set 1. In the example, all three blocks—0, 4, and 8—are mapped to set 0. The first reference to block 8 presents an interesting situation. This even-numbered block is mapped to set 0. However, set 0 is full (with blocks 0 and 4). Now we need to make a decision as to which block should be replaced. This is determined by the replacement policy in effect. We use LRU. Between blocks 0 and 4, block 0 has been accessed more recently. Thus, we replace block 4 with block 8. A similar situation arises when block 4 is accessed later. Using set-associative mapping gives us 8/12 67% as the hit ratio. As you can see, this hit ratio is better than that of the direct-mapped cache but lower than that of the fully associative-mapped cache.

<table>
<thead>
<tr>
<th>Block accessed</th>
<th>Hit or miss</th>
<th>Set 0</th>
<th>Set 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Cache line 0</td>
<td>Cache line 1</td>
</tr>
<tr>
<td>0</td>
<td>Miss</td>
<td>Block 0</td>
<td>???</td>
</tr>
<tr>
<td>4</td>
<td>Miss</td>
<td>Block 0</td>
<td>Block 4</td>
</tr>
<tr>
<td>0</td>
<td>Hit</td>
<td>Block 0</td>
<td>Block 4</td>
</tr>
<tr>
<td>8</td>
<td>Miss</td>
<td>Block 0</td>
<td>Block 8</td>
</tr>
<tr>
<td>0</td>
<td>Hit</td>
<td>Block 0</td>
<td>Block 8</td>
</tr>
<tr>
<td>8</td>
<td>Hit</td>
<td>Block 0</td>
<td>Block 8</td>
</tr>
<tr>
<td>0</td>
<td>Hit</td>
<td>Block 0</td>
<td>Block 8</td>
</tr>
<tr>
<td>4</td>
<td>Hit</td>
<td>Block 0</td>
<td>Block 8</td>
</tr>
<tr>
<td>0</td>
<td>Miss</td>
<td>Block 0</td>
<td>Block 4</td>
</tr>
<tr>
<td>4</td>
<td>Hit</td>
<td>Block 0</td>
<td>Block 4</td>
</tr>
<tr>
<td>0</td>
<td>Hit</td>
<td>Block 0</td>
<td>Block 4</td>
</tr>
<tr>
<td>4</td>
<td>Hit</td>
<td>Block 0</td>
<td>Block 4</td>
</tr>
</tbody>
</table>

HIT RATIO
Virtual Memory

- Virtual memory was developed to give programs a larger memory space than the system’s main memory. The appearance of larger address space is realized by using much slower disk storage.

- The concepts and the principal implementation techniques are very similar to the cache systems discussed in the last chapter. In implementing virtual memory, the main memory and disk are divided into fixed size pages. A mapping table called a page table does the mapping of pages between the disk and main memory.
• In a memory hierarchy system, programs and data are first stored in auxiliary memory. Portions of a program or data are brought into main memory as they are needed by the CPU.

• Virtual memory permits the user to construct programs as though a large memory space were available, equal to the totality of auxiliary memory.

• Each address that is referenced by the CPU goes through an address mapping from the so-called virtual address to a physical address in main memory.

• Virtual memory is used to give programmers the illusion that they have a very large memory at their disposal, even though the computer actually has a relatively small main memory.

• A virtual memory system provides a mechanism for translating program-generated addresses into correct main memory locations. This is done dynamically, while programs are being executed in the CPU. The translation or mapping is handled automatically by the hardware by means of a mapping table.
Thus the address space is the set of addresses generated by programs as they reference instructions and data;

- the memory space consists of the actual main memory locations directly addressable for processing.

- In most computers the address and memory spaces are identical.

- The address space is allowed to be larger than the memory space in computers with virtual memory.
Example

Main Memory Capacity = 4K words $\Rightarrow$ needs 12 bits to specify a physical address

Auxiliary Memory = 1024K words $\Rightarrow$ can support 20 bit addresses.
ADDRESS MAPPING

![Diagram of address mapping process](image)

32-bit virtual address

Virtual page number

20-bit virtual page number

Page translation

12-bit physical page number

Physical page number

24-bit physical address
MEMORY MAPPING TABLE

Virtual address

Virtual address register (20 bits)

Memory mapping table

Main memory address register (12 bits)

Main memory

Memory table buffer register

Main memory buffer register
Pages and Blocks

- Physical memory is broken down into groups of equal size called BLOCKS.
- The groups of address space of same size are called PAGES.
- A page refers to the organization of address space, while a block refers to the organization of memory space.
  - In previous example address space is divided into 1024 pages and main memory is divided into 4 blocks.
- The programs are also split into pages.
- Portions of programs are moved from auxiliary memory to main memory in records equal to the size of a page.
**Present bit:**

0: Page is not in physical memory

1: Page is in physical memory

<table>
<thead>
<tr>
<th>Page #</th>
<th>Present bit</th>
<th>Disk address</th>
<th>Page frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>01001011100</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>11101110010</td>
<td>xx</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>10110010111</td>
<td>01</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>00001001111</td>
<td>xx</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>01011100101</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>10100111001</td>
<td>xx</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>00110101100</td>
<td>xx</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>01010001011</td>
<td>10</td>
</tr>
</tbody>
</table>
Memory Table in a paged system

Virtual address

Page no.  Line number
1 0 1  0 1 0 1 0 1 0 1 0 0 1 1

Table address

Presence bit

Main memory

Block 0
Block 1
Block 2
Block 3

Memory page table

MBR
EXAMPLE

Page Offset

Virtual address

Page table

Physical address
Page Fault and Page Replacement

• The VM system decides
  – Which page in the main memory ought to be removed to make room for a new page
  – When a new page is to be transferred from auxiliary memory to main memory
  – Where the page is to be placed in the main memory.

• When a program starts execution, one or more pages are transferred into main memory and the page table is set to indicate their position. The program is executed from main memory until it attempts to reference a page that is still in auxiliary memory. This condition is called PAGE FAULT.

• When *page fault* occurs, the execution of the present program is suspended until the required page is brought into main memory.
Page Replacement Policies

• First In First Out (FIFO) – selects for replacement the page that has been in memory the longest time.
  – Easy to implement.
  – Under certain circumstances, pages are removed and loaded into main memory too frequently.

• Least Recently Used (LRU) –
  – Uses ageing registers to calculate age of pages.
    » At certain intervals these registers are incremented
    » When a page is referenced, its counter is set to zero.
FIFO

- When a page must be replaced, the oldest page is chosen

reference string
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

page frames

```
<table>
<thead>
<tr>
<th>7</th>
<th>7</th>
<th>7</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>2</th>
<th>2</th>
<th>4</th>
<th>4</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>7</th>
<th>7</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
```

FIFO

- When a page must be replaced, the oldest page is chosen.

- In all our examples, the reference string is 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.
- 3 frame (9 page faults)
- 4 frame (10 page faults)

- Notice that the number of faults for 4 frames is greater than the number of faults for 3 frames!! This unexpected result is known as Belady’s anomaly.
<table>
<thead>
<tr>
<th>Page #</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>1</th>
<th>2</th>
<th>5</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Page #</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>-------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>2</td>
<td>2</td>
<td></td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
FIFO Illustrating Belady’s Anomaly
Optimal Page-Replacement Algorithm

• Replace page that will not be used for longest period of time

• This is a design to guarantee the lowest page-fault rate for a fixed number of frames
Optimal Page-Replacement Algorithm

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

page frames

7 7 7 2 2 2 2 2 2 2 7
0 0 0 0 0 4 0 0 0 0
1 1 1 3 3 3 3 3 3 3
Optimal Page-Replacement Algorithm

4 frames example

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

6 page faults
• Unfortunately, the optimal page-replacement is difficult to implement, because it requires future knowledge of the reference string
Least-recently-used (LRU) algorithm

- LRU replacement associates with each page the time of that page’s last use.
- When a page must be replaced, LRU chooses the page that has not been used for the longest period of time.
Least-recently-used (LRU) algorithm

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

page frames

7 7 7 2 2 4 4 4 0 1 1 1
0 0 0 0 0 3 3 3 3 3 0 0
1 1 1 1 2 2 2 2 2 2 2 7
Least-recently-used (LRU) algorithm

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
Least-recently-used (LRU) algorithm

• The major problem is how to implement LRU replacement:
  1. Usage Time: whenever a reference to a page is made, the content of the clock register are copied to the time-of-use filed in the page table entry for the page. We replace the page with the smallest time value
  2. Modified Stack: Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the most recently used page is always at the top of the stack
Stack implementation

Reference string:

<table>
<thead>
<tr>
<th>4</th>
<th>7</th>
<th>0</th>
<th>7</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>1</th>
<th>2</th>
<th>7</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
</table>

Stack before a:

- 4
- 7
- 0
- 1
- 2

Stack after b:

- 7
- 2
- 1
- 0
- 4

a and b are arrows pointing upward.
Counting Based Page Replacement

- Least Frequently used (LFU) page-replacement algorithm
- Most frequently used (MFU) page-replacement algorithm
- When there is a tie, use FIFO
<table>
<thead>
<tr>
<th>REF. String</th>
<th>7</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>0</th>
<th>3</th>
<th>0</th>
<th>4</th>
<th>2</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
12-20.

A virtual memory has a page size of 1K words. There are eight pages and four blocks. The associative memory page table contains the following entries:

<table>
<thead>
<tr>
<th>Page</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

Make a list of all virtual addresses (in decimal) that will cause a page fault if used by the CPU.

The pages that are not in main memory are:

<table>
<thead>
<tr>
<th>Page</th>
<th>Address</th>
<th>address that will cause fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2K</td>
<td>2048 - 3071</td>
</tr>
<tr>
<td>3</td>
<td>3K</td>
<td>3072 - 4095</td>
</tr>
<tr>
<td>5</td>
<td>5K</td>
<td>5120 - 6143</td>
</tr>
<tr>
<td>7</td>
<td>7K</td>
<td>7168 - 8191</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A virtual memory system has an address space of 8K words, a memory space of 4K words, and page and block sizes of 1K words (see Fig. 12-18). The following page reference changes occur during a given time interval. (Only page changes are listed. If the same page is referenced again, it is not listed twice.)

4 2 0 1 2 6 1 4 0 1 0 2 3 5 7

Determine the four pages that are resident in main memory after each page reference change if the replacement algorithm used is (a) FIFO; (b) LRU.
<table>
<thead>
<tr>
<th>Page reference</th>
<th>(a) Pages in main memory</th>
<th>Contents of FIFO</th>
<th>(b) Pages in memory</th>
<th>LRU Most recently used</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initial</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0124 0124 0126 0146</td>
<td>4201</td>
<td>0124 0124 0126 0146</td>
<td>4201 4012 0126</td>
</tr>
<tr>
<td>6</td>
<td>0126 0126 0146 0146</td>
<td>2016</td>
<td>0126 0126 0146</td>
<td>2016 2614 6140</td>
</tr>
<tr>
<td>1</td>
<td>0146 0146 0146 1246</td>
<td>0164</td>
<td>0146 0146 0146</td>
<td>0164 6401 6410</td>
</tr>
<tr>
<td>4</td>
<td>0146 0146 1246 2346</td>
<td>0164</td>
<td>0146 0146 0124</td>
<td>0164 4102 1023</td>
</tr>
<tr>
<td>0</td>
<td>0146 1246 2346 2345</td>
<td>1642</td>
<td>0146 0123 0235</td>
<td>1642 4102 1023</td>
</tr>
<tr>
<td>1</td>
<td>2346 2345 2357 2357</td>
<td>6423</td>
<td>0235 0235 0235</td>
<td>6423 2357 2357</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
In static RAM, a form of flip-flop holds each bit of memory. A flip-flop for a memory cell takes 4 or 6 transistors along with some wiring, but never has to be refreshed. This makes static RAM significantly faster than dynamic RAM. However, because it has more parts, a static memory cell takes a lot more space on a chip than a dynamic memory cell. Therefore you get less memory per chip, and that makes static RAM a lot more expensive.

- Provides a fast access time at the expense of lower bit densities
- For this reason registers and cache subsystems are fabricated using SRAM technology
- Static RAM is considerably more expensive than Dynamic RAM
- However, since it doesn’t need to be refreshed, its power consumption is much lower than DRAM
- Also, the absence of the refresh circuitry makes it easier to interface to
- The simplicity of the memory circuitry compensates for the more costly technology
Dynamic RAM is the most common type of memory in use today. Inside a dynamic RAM chip, each memory cell holds one bit of information and is made up of two parts: a transistor and a capacitor. The capacitor holds the bit of information -- a 0 or a 1. The transistor acts as a switch that lets the control circuitry on the memory chip read the capacitor or change its state.

A capacitor is like a small bucket that is able to store electrons. To store a 1 in the memory cell, the bucket is filled with electrons. To store a 0, it is emptied. The problem with the capacitor's bucket is that it has a leak. In a matter of a few milliseconds a full bucket becomes empty. Therefore, for dynamic memory to work, either the CPU or the memory controller has to come along and recharge all of the capacitors holding a 1 before they discharge. To do this, the memory controller reads the memory and then writes it right back. This refresh operation happens automatically thousands of times per second.

This refresh operation is where dynamic RAM gets its name. Dynamic RAM has to be dynamically refreshed all of the time or it forgets what it is holding. The downside of all of this refreshing is that it takes time and slows down the memory.
The bulk of a processor’s main memory is comprised of dynamic RAM.

In contrast to SRAM, DRAM uses a single transistor and capacitor to store a bit.

DRAM requires that the address applied to the device be asserted in a row address (RAS) and a column address (CAS).

The requirement of RAS and CAS of course kills the access time, but it allows for higher memory densities.

RAS and CAS use the same pins, with each being asserted during either the RAS or the CAS phase of the address.

There are two metrics used to describe DRAM’s performance:
- Access time is defined as the time between assertion of RAS to the availability of data.
- Cycle time is defined as the minimum time before a next access can be granted.

Cycle times establish throughput of the system.

Of course the charge leaks slowly from the storage capacitor in DRAM and it needs to be refreshed continually.

During the refresh phase, all accesses are held-off, which happens once every 1 – 100 ms and slightly impacts the throughput.
2D MEMORY ORGANIZATION

2D organization of a memory chip of size 16 × 4
2D MEMORY ORGANIZATION

- The cells are organized in the form of a two-dimensional array with rows and columns.
- Each row refers to word line. For 4-bit per word memory, 4 cells are interconnected to a word line. Each column in the array refers to a bit line.
- The Memory Address Register (MAR) holds the address of the location where read/write operation is executed.
  - Here in this example, MAR has 4 bits.
- The content of MAR is decoded by an address decoder on the chip to activate each word line.
- The cells in each column are connected to a sense/write circuit by two bit lines. Two bit lines are complement to each other.
- The sense/write circuits are activated by the chip select (CS) lines. The sense/write circuits are connected to the data lines of the chip.
- During a read operation, these circuits sense or read the information stored in the cells selected by a word line and transmit this information to the data lines.
- During a write operation, the sense/write circuits receive or write input information from the data lines and store it in the selected cells.
3D MEMORY ORGANIZATION

\[ x + y = n \]
\[ X = 2^x \]
\[ Y = 2^y \]
3D MEMORY ORGANIZATION

- For an n-bit MAR in 2D organization the word lines are linearly selected and hence the number of decoder gates is $2^n$.
- By contrast, in 3D organization, the number of decoder gates reduces to $2.2^{n/2}$ for $x = y = n/2$. Such saving in the circuit cost has motivated the designers to design 3D organized memory cell array.
- The n-bit address is divided into two parts having x and y number of bits. For a square array, each half is decoded and $2^{n/2}$ X and Y drive lines are fed into each array of bit plane.
- For b-bit word memory, there are b number of planes each referring to a bit.
- Corresponding to each bit plane there is a sense/write circuit. The read/write operations in 3D organization is same as to 2D with the modifications that a cell in a bit plane is selected by activating X and Y drive lines simultaneously, and bit information passed through the selected cell in a bit plane.
- Thus each cell in the array needs 3 terminals – X, Y and bit line connected to sense/write circuit.
- More the number of terminals (wires) through a cell, larger the cell size and consequently switching speed is less. Also, the design of the overall circuit becomes very complex.
2½D MEMORY ORGANIZATION

2.5D memory organization
To cope up the above difficulties experienced in 3D organization, the design of 2.5D memory organization has evolved which combines the function of bit lines and Y drive lines.

In 2.5D organization there exists a segment, corresponding to bit plane of 3D organization.

The content of MAR is divided into two parts—x and y number of bits.

The number of segments S is equal to $2^y$.

$X = 2^x$ drive lines are fed into the cell array and y number of bits decode one bit line out of S lines fed into a segment of the array. In total, there are $S_b$ number of bit lines for a b bit per word memory.

Thus for any given address in the MAR, the column decoder decodes b out of $S_b$ bit lines by using the y bits of the MAR while a particular word line is activated by using the x bits.

Thus only the b number of bits in the array are accessed by enabling the word line and b number of bit lines simultaneously.

Though 2.5D organized memory may need lesser chip decoding logic, it suffers from one drawback. With high density chips, a simple failure, such as external pin connection opening or a failure on one bit can render the entire chip inoperative.