2 Memory and I/O Modules

2.1 Memory Taxonomy & Characteristics

Physical Types of Memory

- Semiconductor（半导体介质）: RAM & ROM;
- Magnetic（磁介质）: Disk & Tape;
- Optical（光介质）: CD & DVD.

Location

- The memory in CPU: *registers*.
- The internal memory: *cache*, *main memory*.
- The external memory: *disks, CD and DVD*.

Capacity

- *Word size*: The natural unit of organization (this 'word' is defined in memory field).
- Number of words

> [*Example*]
>
> Memory A have a word size of 8-bit and have 2M words.
>
> Memory B have a word size of 1-bit and have 16M words.
>
> Memory A and Memory B have the same capacity.

Unit of Transfer

- Internal: usually a word, governed by the data bus width;
- External: usually a block which is much larger than a word.

> [*Example*] The CPU can calculate one addition every cycle, but a memory transfer takes two cycles. That is,
>
> Memory: 0.5 words/cycle
>
> CPU: 2 words/cycle (calculation)
>
> With the memory interface width to be 4 words, the CPU can be kept with 100% utilization.

Addressable unit

- Smallest location which can be uniquely addressed.
- Normally a byte for internal memory;
- *Cluster of disks*.

Access Methods

- **Sequential**: Access start at the beginning and read through in order. Access time depends on location of data and previous location. (*tape*) (**NO address**)
- **Direct**: Individual blocks have unique address. Access is by jumping to vicinity plus sequential search. Access time depends on current location and destination location. (*disk*) (**HAVE address**)
- **Random**: Individual addresses identify locations exactly. Access time is independent of location or previous access. (*ROM*, *RAM*) (**HAVE address**)

- **Associative**: Data is located based on a portion of its contents rather than its address. Access time is independent of location or previous access. (*cache*) (**HAVE address**, but using contents to find location)

Performance Metrics

- 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. It

  > [*Example*] When we access the data in *DRAM*, we destroy the data, so we have to restore ("recover") the data in *DRAM*.

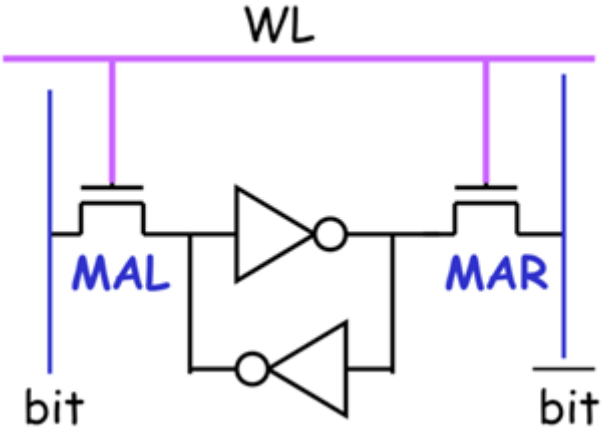
- Transfer rate: rate at which data can be moved. (unit: transfer per second)
- Transfer bandwidth: equals to transfer rate * transfer unit size (unit: bytes per second)

> [*Example*] A memory transfer takes two cycles and each transfer has 4 bytes. Clock frequency is 1GHz.
>
> $$A.T. = 2 \text{ cycles} = 2 \text{ ns}$$
> $$T.R. = 0.5 \text{ T/cycle} = 0.5 \text{ GT/s}$$
> $$T.B. = 0.5 \text{ GT/s} \times 4 \text{ B/T} = 2 \text{ GB/s}$$

Memory Basics

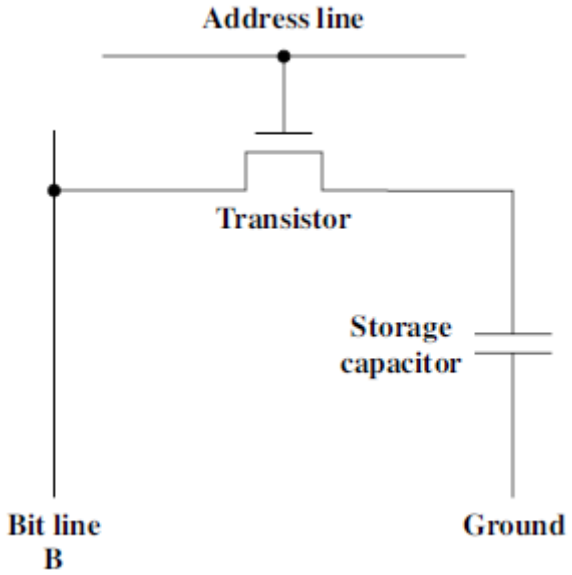
- **RAM**: Random Access Memory.
  - Read/Write;
  - Temporary storage: when the power is gone, the data will be lost;
- **ROM**: Read-Only Memory.
  - The writing process of this kind of memory is much harder than reading, often requiring some external help.
  - No capabilities for "online" memory write operations.
  - Both RAM and ROM are made by *semiconductors*.
- Permanent storage (Nonvilatile: need no power);
  - Normal applications of ROM (firmware): BIOS, function tables, etc.
  - Types of ROM:
    - Written during manufacture;
    - Programmable (only once) (*PROM*);
    - Read mostly (*EPROM*, *EEPROM*, *Flash memory*).
- Volatility of Memory（易失性）
  - Volatile memory loses data over time or when power is removed. (*RAM*)
  - Non-volatile memory stores date even when power is removed. (*ROM*)
  - **Static**: holds data as long as power is applied. (*SRAM*)
  - **Dynamic**: will lose data unless refreshed periodically. (*DRAM*)

Static RAM



- The cycle of inverters keep the data running.
- As long as the power is on, the data is stored.
- When we want to read the data, open MAL and MAR, and read the data through them; when we want to write the data, open MAL and MAR, and write the data through them (need some time).
- 6T-SRAM (6-transistors, MAL, MAR, and 2 transistors in each inverter).

Dynamic RAM



- Simpler construction;
- Need refresh circuits; slower;
- Consume less transistor (only 1 transistor), less expensive;
- Need 'recovery' part (after reading operation, the data will be lost).

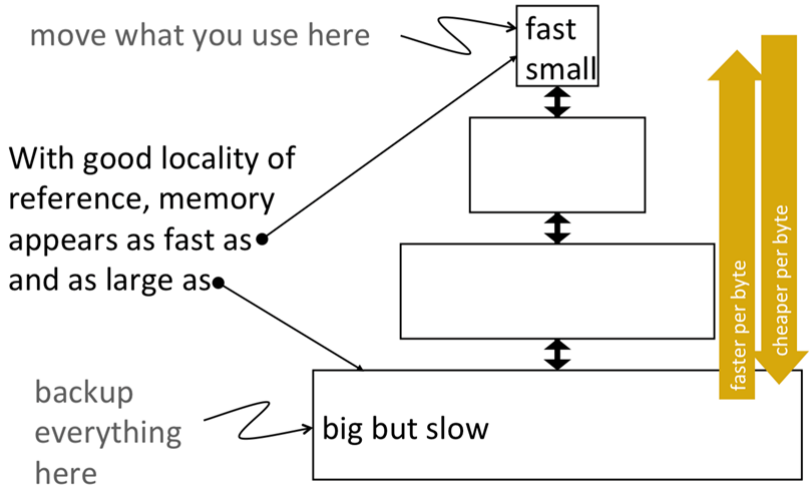
2.2 Memory Hierarchy in Computer System

Memory Hierarchy (Registers, L1 Cache, L2 Cache, Main memory, Disk, Optical, Tape).

Why Memory Hierarchy?

- Bigger (capacity) is slower.
- Faster is more expensive.

Idea Behind Memory Hierarchy

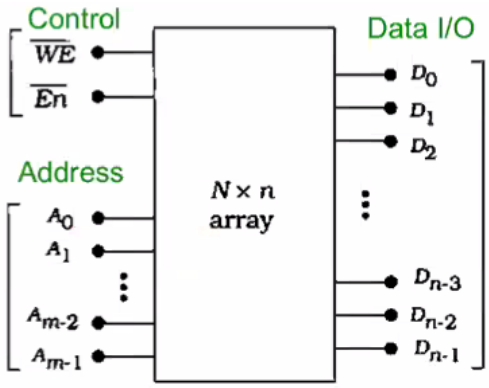


2.3 RAM Organization

Each RAM has a memory chip.

Memory Chip

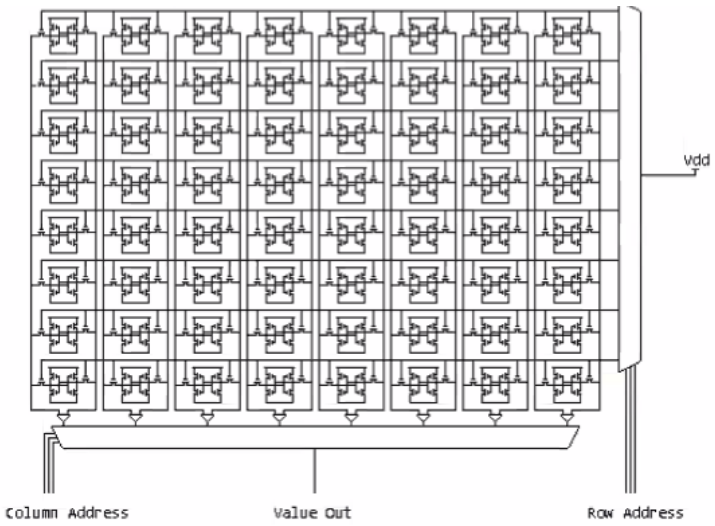
- $N \times n$ memory chip. (n is the *chip word* width, N is the number of n-bit words)
- Chip Input/Output:



  - Data (*in and out*): $D_{n-1}, D_{n-2}, \ldots, D_0$;
  - Address: $A_{m-1}, A_{m-2}, \ldots, A_0$, usually $m = \log_2 N$ (or less).
  - Control:
    - WE: write enable (assert low). (*WE = 1*, read; *WE = 0*: write)
    - En: block enable (assert low).

> [*Example*]: $4M \times 4$ RAM (16-Mbit RAM chip)
>
> - $2^{22} = 4M$ words;
> - Each word has 4 bits;
> - 22 address lines (reduced to 11 if *multiplexed*);
>   - That means, two transfers to get the full address.（每次传输一半的地址）
>   - 22 address lines is the maximum requirement.
> - 4 data lines.

- Physical view of memory chip: $N_R \times N_C$ array of 1-bit cells, where N_R is number of rows and N_C is number of columns.

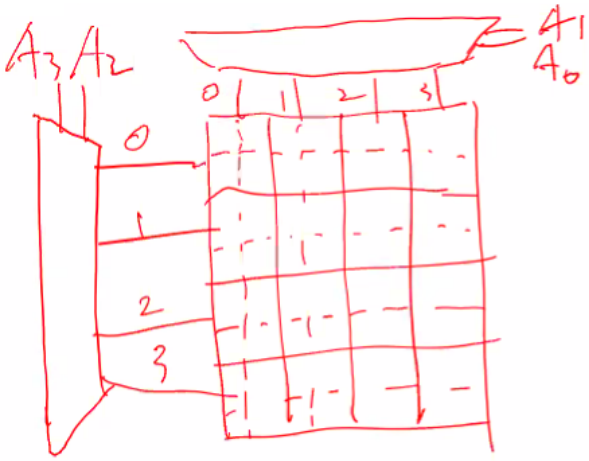


N_R and N_C should be power of 2.

It needs two decoders: column decoder and row decoder.

How to Access Memory Array

> [*Example*] A 16×1 chip. Its physical view is a 4×4 array.
>
> Address lines: A_3, A_2, A_1, A_0, Data lines: D_0.
>
> Row Address: A_3, A_2; Column Address: A_1, A_0. (*usually regard higher part as row address and lower part as column address*)
>
> The decoders are all the n **to** 2^n **decoder**, which means n input of $0/1$ and 2^n output of $0, 1, \ldots, 2^n - 1$. (Like binary form of decimal).



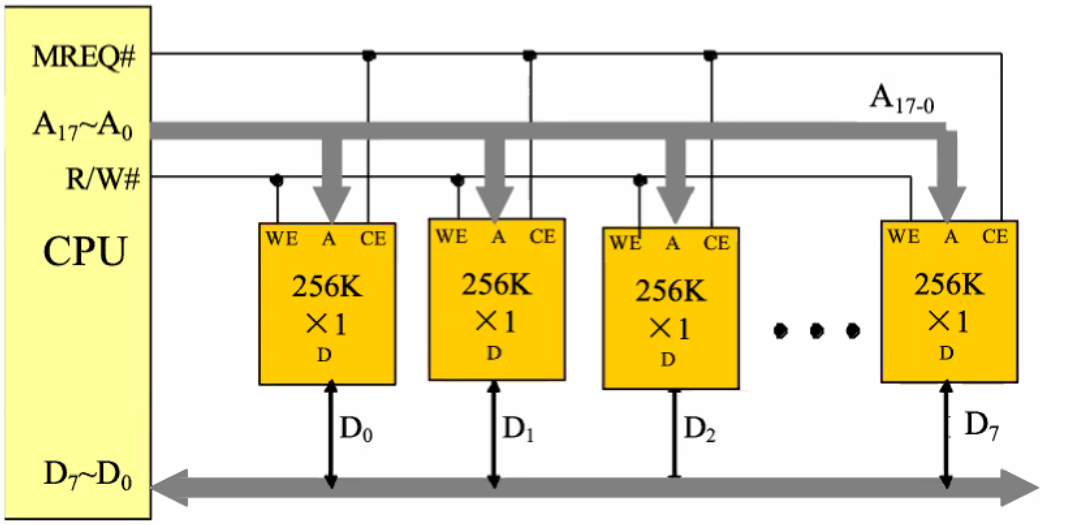
> - For each row, the word line is connected.
> - For each column, the bit line is connected.

Explanation Each array only has one bit. If n-bit is needed, then organize n array in *parallel* order *within the chip*. The address connection will be the same. Each array only gives an output to a data line, and the combination of n arrays is the true output.

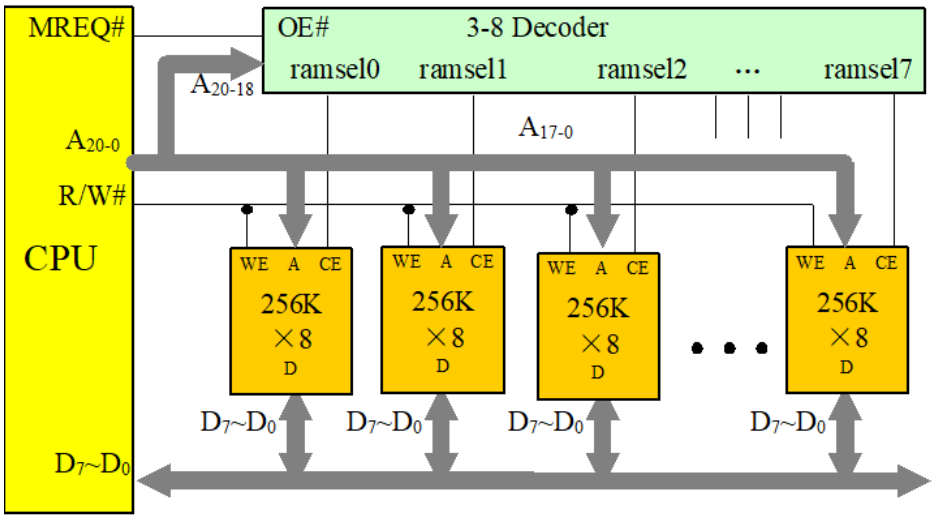
- Multiple cells form a memory array.
- Multiple arrays form a memory chip.

2.4 Memory Module Extension

Bit Extension

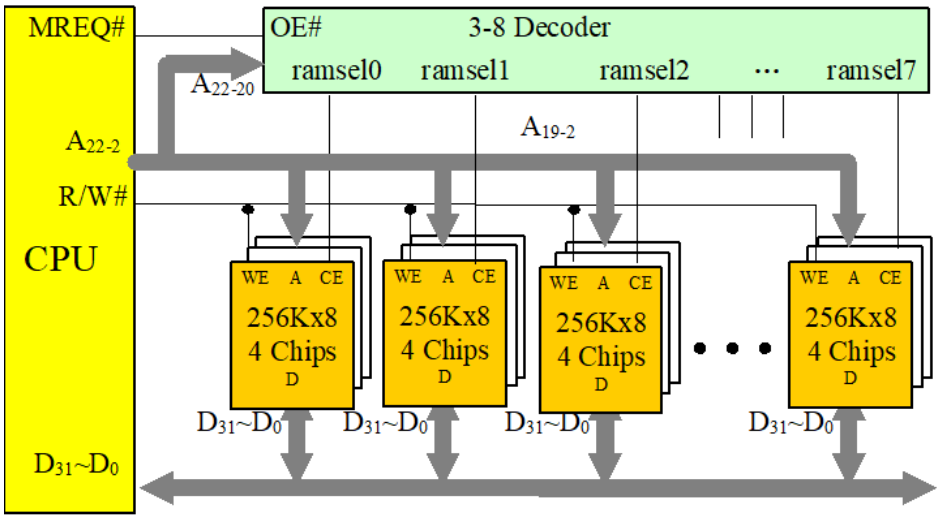
> [*Example*] You have 256K×1-bit RAM chips. How can you build a memory module of 256 KB (with word size of 8 bits) and how to connect this module with a computer system?
>
> 
>
>
> Just use the method we introduced before.

Word Extension

> [*Example*] You have 256K×8-bit RAM chips. How can you build a memory module of 2M×8-bit and how to connect this module with a computer system?
>
> 
>
>
> Add a 3 to 2^3 decoder and use 8 chips.

Word and Bit Extension

> [*Example*]
>
> Now you have 256K×8-bit RAM chips. How can you build a memory module of 2M×32-bit and how to connect this module with a computer system if the addressable unit is byte?



We remove A_0 and A_1 because the data bus width is 32 bits, and the addressable unit is usually byte (8 bits). So the last two digits of A is meaningless to memory, since we have to transfer 32 bits to the CPU (because of data bus width).

How many digits do we have to remove?

$$\max(\log_2(\text{dest bits}) - \log_2(\text{addr unit}), 0)$$

How many bit extension?

$$\log_2(\text{dest bits}) - \log_2(\text{src bits})$$

Addressable Unit default value: 8 bits.

When to use extension?

- CPU wants: $N_{CPU} \times n_{CPU}$;
- Memory chip has: $N_{mem} \times n_{mem}$;
- Word extension: $N_{CPU} > N_{mem}$:
  - connect data bus of CPU and all memory chips directly.
  - require a extra decoder.
- Bit extension: $n_{CPU} > n_{mem}$:
  - Connect address bus of CPU and all memory chips directly
    - May discard some address lines if $n_{CPU} > 8$ bits.
  - Assemble memory chips' data buses to connect with CPU.

2.5 I/O Modules

There are a wide variety of peripherals（外设）

- Different operation logic.
  - Impractical for CPU to control all kinds of devices;
- Speak different 'languages': the devices deliver different amount of data (serial / parallel), the devices have different speed, and the devices have different format.
  - Impratical for CPU to understand.
- Slower than CPU and RAM
  - Impratical to directly connect devices with highspeed system bus.

External Devices

- Human readable: Screen, Printer, Keyboard;
- Machine readable: Monitoring and control;
- Communication: Modem（调制解调器）, Network Interface Card (NIC).

I/O Module has an interface to CPU and memory and an interface to one or more peripherals. It's like a *bridge*, an *interpreter*, a *buffer*, ...

I/O Module Function

- Control & Timing
- Communication (CPU & Device)
- Data Buffering
- Error Detection

> [*Example*] (**I/O Steps**) Transfer the data from externel device to processor (CPU).
>
> - CPU checks I/O module for device status;
> - I/O module returns the device status;
> - If the device is ready, CPU requests data transfer by means of a command to the I/O module;
> - I/O module gets a unit of data from device;
> - I/O module transfers the data to CPU;
> - Variations for output, DMA, etc.

I/O Module Design Decisions

- Hide / Reveal device properties to CPU;
- Support multiple or single device
- Control device functions or leave to CPU.

Input Output Techniques

- Programmed（直接编程型）
- Interrupt driven（中断驱动型）
- Direct Memory Access (DMA)（直接内存访问）

Programmed I/O

- CPU executes a program that gives it direct control of I/O operation.
    - CPU requests I/O operation;
    - I/O module performs operation;
    - I/O module sets status bits;
    - CPU checks status bits periodically;
    - Once the I/O module is ready, then CPU deals with the event.
- I/O module does not imform CPU directly, or interrupt CPU;
- CPU may wait or come back later (for example, with the help of time-sharing operating system).
- CPU issues **address**: to indentify module.
- CPU issues different **command**: control, test (check), read/write.
- The CPU will keep waiting. Waste time resources.

> [*Example*] (Programmed I/O)

```c
# include <stdio.h>
int main() {
    int x, y, z;
    scanf("%d%d", &x, &y);
    z = x + y;
    printf("%d\n", z);
    return 0;
}
```

> When there are no numbers entered in, the CPU will keep waiting.

Interrupt Driven I/O

- I/O Module interrupts when ready.
    - CPU requests I/O operation
    - I/O module performs operation while CPU does other work;
    - I/O module informs CPU when it is ready by interrrupting CPU
    - CPU deals with the event.
- **Interrupt**: New event needs CPU to handle first but CPU needs to go back to privous work after that. It needs an **interrupt handler** to handle the interrupt.
- CPU issues read command and do other work.
- CPU check for interrupt at the end of each instruction cycle.
- If interrupted:
    - save context (registers);
    - process interrupt;
    - recovered from the saved context.
- Design Issues (only need to know)
    - How can CPU know which module is issuing the interrupt?
        - Connect a *dedicated line* for each module, but limits the number of devices that we can use;
        - Software poll（查询）: All devices share one common *Interrupt Request* line to interrupt CPU. Once get an interrupt, CPU will asks each module in turn to identify the device.
        - Hardware poll: All the device share one common *Interrupt Request* line to interrupt CPU. *Interrupt Acknowledge* signal is sent down a *daisy chain*. (Hardware circuits)
        - Bus Master: module must claim the bus before raising interrupt. (*PCI & SCSI bus protocols*)
        - Interrupt controller.
    - How to locate the corresponding handler program when interrupted?
        - General handler program: CPU enters this handler every time it gets interrupted, and looks for the module responsible and gets the address of the corresponding handler program. (*software complexity, because it must search for the module*)
        - Interrupt vectors（中断向量表）: pointers are used to link the handler programs, such pointers are interrupt vectors. We can use the pointer to find the handler program. (*hardware complexity, because it must transfer the pointer to CPU*, *high-performance*)
    - How to deal with multiple interrupts?
        - Assign *priority* for interrupts: high-priority interrupts first.
        - Nesting of interrupts: high-priority interrupts can further interrupt low-priority interrupts.

- Both *programed I/O* and *interrupt driven I/O* need involvement of CPU.

> [*Example*] (Interrupt Driven I/O)
>
> ```
> # include <signal.h>
> void keyboard_handler(int signal) {
>     // read the keyboard symbols.
> }
> void main() {
>     // first setup the keyboard interrupt handler.
>     handler_init();
>     while(1) {
>         // do something else here.
>     }
> }
> ```
>
> The event we are monitoring will signal the CPU to call the function `keyboard_handler()` automatically. We can free CPU from the waiting process. (like *function calls* but has differences.)

Direct Memory Access I/O

- DMA needs additional module (hardware on bus), and *DMA controller* will do the I/O work for CPU.
    - CPU tell DMA controller: read/write, device address, starting address of memory block for data, amount of data to be transferred, ...
    - CPU carries on with other work;
    - DMA controller deals with transfer;
    - DMA controller sends interrupt when finished.
    - **Notes**: The *interrupt* is different from the *interrupt driven I/O*: here is the sign that the process is finished and the last one is the sign of the device is ready.
- In an instruction cycle, the processor may be suspended due to DMA operation. (DMA can have breakpoints after each stage, but interrupt breakpoint can only appear after a certain stage)
- DMA Configurations (refer to the slides): It's a trade-off between simplicity and performance.

> [*Example*] (Direct Memory Access I/O)
>
> ```
> # include <dma.h>
> void main() {
>  int buffer[1024];
>  // first tsetup the dma
>  dma_init(buffer);
>  while(1) {
>      // do something else here.
>  }
> }
> ```
>
> The DMA will free CPU from waiting for the I/O event and waiting for the data. The transfer is done by hardware circuits automatically.

We increase the design complexity to improve performance.

Summary of Different I/O Techniques

Interaction with I/O device	Programmed I/O	Interrupt-driven I/O	Direct Memory Access
Waiting for the deivce	Software (CPU)	Hardware	Hardware
Transfer the device data to memory	Software (CPU)	Software	Hardware