

Komponenten/Busse

Dr.-Ing. Volkmar Sieh

Department Informatik 4
Systemsoftware
Friedrich-Alexander-Universität Erlangen-Nürnberg

WS 2024/2025



Beispiel war einfach; Praxis u.U. sehr viel komplexer:

- mehrere Busse; jeweils mehrere angeschl. Geräte
- mehrere CPUs
- mehrere RAM-Module
- mehrere (verschiedene) I/O-Geräte

Zusätzliche Forderung:

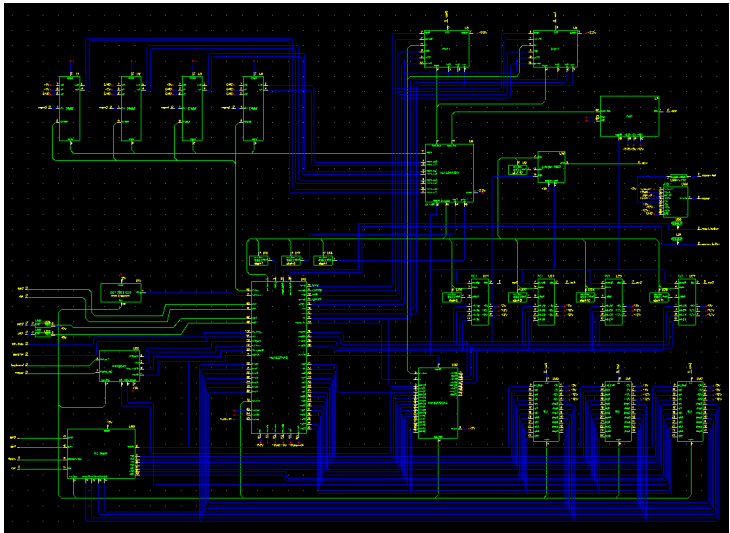
- Konfigurierbarkeit



**Einfacher „Standard“-PC:
ca. 300 Komponenten und Kabel!**



Beispiel: FAUmaschine Motherboard:



Werden

- **Komponenten** (z.B. CPU, RAM, ROM, I/O-Geräte)
- **Busse** (z.B. ISA-Bus, PCI-Bus, PCI-Express-Bus, SCSI-Bus)

getrennt voneinander modelliert, können sie wiederverwendet werden (z.B. für Multiprozessor-System mit mehreren Speicher-Modulen und mehreren Platten).



Alle Komponenten sollten keine globalen/statischen Variablen verwenden. Sonst können sie nur einmal instanziiert werden.

Jede Komponente muss ihren Speicher allozieren. Zugriff auf Zustand nur über Pointer.



Callback-Funktionen:

CPU(s) und I/O-Geräte melden sich an „ihrem“ Bus an. Sie hinterlegen Funktionen, die bei entsprechenden Bus-Zyklen aufgerufen werden sollen („Callback“-Funktionen), und übergeben Pointer auf ihren Zustandsspeicher („State“-Pointer).

CPU ruft Bus-Funktion (z.B. `inb`) auf. Bus-Funktion ruft der Reihe nach die Funktionen der angeschlossenen I/O-Geräte auf (z.B. `ser_in`, `disk_in`) und übergibt State-Pointer wieder.

Jedes aufgerufene Gerät überprüft anhand der übergebenen Adresse/Ports selbst, ob es gemeint ist.



```
struct bus_funcs {
    void (*in)(void *state, int port, uint8_t *valp);
    void (*out)(void *state, int port, uint8_t val);
    void (*load)(void *state, int addr, uint8_t *valp);
    void (*store)(void *state, int addr, uint8_t val);
    ...
};

struct bus {
    struct {
        void *state;
        struct bus_funcs *funcs;
    } member[NMEMBERS];
    unsigned int nmembers;
};
```




```
void bus_connect(  
    struct bus *bus,  
    void *state,  
    struct bus_funcs *funcs  
) {  
    bus->member[bus->nmembers].state = state;  
    bus->member[bus->nmembers].funcs = funcs;  
    bus->nmembers++;  
}
```



```
void bus_out(  
    struct bus *bus,  
    int port,  
    uint8_t val  
) {  
    unsigned int i;  
  
    for (i = 0; i < bus->nmembers; i++) {  
        (bus->member[i].func->out)(bus->member[i].state,  
                                   port, val);  
    }  
}
```



```
void bus_in(  
    struct bus *bus,  
    int port,  
    uint8_t *valp  
) {  
    unsigned int i;  
  
    *valp = 0x00;    /* Or: *valp = 0xff; */  
    for (i = 0; i < bus->nmembers; i++) {  
        (bus->member[i].func->in)(bus->member[i].state,  
                                  port, valp);  
    }  
}
```

Alle angeschlossenen Geräte werden aufgerufen. *Eines* (oder keines!) trägt den Return-Wert ein.



Aufbau des Systems durch

```
void
system_init()
{
    mainbus = bus_new();
    cpu = cpu_new(mainbus);
    ram = ram_new(mainbus, 0, 16384);
    rom = rom_new(mainbus, 16384);
    ser = serial_new(mainbus, 0);
    disk = disk_new(mainbus, 4);
}
```



Vergleich mit VHDL:

```
architecture structural of system is
  signal mainbus : bus;
begin
  cpu : cpu
    port map(mainbus => mainbus);
  ram : ram
    generic map(start=0, end=16384)
    port map(mainbus => mainbus);
  rom : rom
    generic map(start=16384)
    port map(mainbus => mainbus);
  ser : serial
    generic map(io=0)
    port map(mainbus => mainbus);
  disk : disk
    generic map(io=4)
    port map(mainbus => mainbus);
end structural;
```



Emulation des Systems durch

```
void
system_step()
{
    while (1) {
        /* bus_step(bus); */
        cpu_step(cpu);
        /* ram_step(ram); */
        /* rom_step(rom); */
        ser_step(ser);
        disk_step(disk);
    }
}
```



```
struct ser {
    /* Config */
    int port;

    /* Ports */
    struct bus *bus;

    /* State */
    int key;
    int avail;
};
```



```
void
ser_in(struct ser *ser, int port, uint8_t *valp)
{
    if (port < ser->port || ser->port + 1 < port)
        return;
    switch (port & 1) {
        case 0: ser->avail = 0; *valp = ser->key; return;
        case 1: *valp = ser->avail; return;
    }
}

void
ser_out(struct ser *ser, int port, uint8_t val)
{
    if (port != ser->port)
        return;
    host_print_char(val);
}
```




```
void
ser_step(struct ser *ser)
{
    if (host_key_avail()) {
        ser->key = host_get_key();
        ser->avail = 1;
        bus_interrupt(ser->bus, ser);
    }
}
```



```
struct ser *
ser_new(struct bus *bus, int port)
{
    static const struct bus_funcs ser_funcs = {
        .in = ser_in,
        .out = ser_out,
    };
    struct ser *ser = malloc(sizeof(struct ser));
    ser->port = port;
    ser->bus = bus;
    ser->key = 0;
    ser->avail = 0;
    bus_connect(bus, ser, &ser_funcs);
    return ser;
}
```



Komponenten und Busse entsprechen in VHDL-Entities bzw. -Signalen.

Bus-Funktionsaufrufe (z.B. `bus_in`, `bus_store`) entsprechen Bus-Zyklen echter Busse (hier IOR- bzw. MEMW-Zyklus).



Die Bus-Indirektionsstufen

CPU -> Bus -> RAM

CPU -> Bus -> I/O

kosten leider Performance.

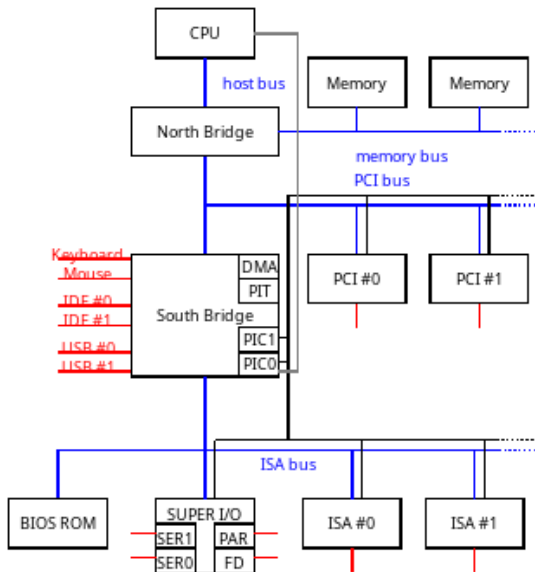
In Realität leider sogar noch längere Wege: z.B.:

CPU -> Host-Bus -> North-Bridge -> Mem-Bus -> RAM

CPU -> Host-Bus -> North-Bridge -> PCI-Bus -> South-Bridge ->
ISA-Bus -> I/O



Komponenten/Busse – Performance



- Lösung für I/O-Komponenten: Weiterreichen der Callback-Funktionen an die CPU und Cachen der Callback-Funktionen in der CPU
- Lösung für Speicher: Weiterreichen der Adresse an die CPU und direktes Nutzen des Speichers durch die CPU



- I/O-Zugriffe auf den gleichen Port gehen (fast) immer wieder an das gleiche I/O-Gerät

=> in- bzw. out-Callback-Pointer in der CPU cachen



```
void
bus_in_info(
    struct bus *bus, int port,
    void **statep,
    void (**inp)(int, uint8_t *)
) {
    unsigned int i;

    *inp = NULL;
    for (i = 0; i < bus->nmembers; i++) {
        bus->member[i].func->in_info(
            bus->member[i].state,
            port, statep, inp);
    }
}
```

Entsprechend: `bus_out_info`. Evtl. existiert noch eine Funktion zum Flushen der I/O-Info (wenn Ports sich ändern).




```
void *state[0x10000];
void (*cb[0x10000])(void *state, int p, uint8_t *valp);

void
in(int p, uint8_t *valp) {
    void (*func)(void *state, int p, uint8_t *valp);

    func = cb[p];
    if (! func) {
        bus_in_info(bus, p, &state[p], &cb[p]);
        func = cb[p];
        if (! func) return;
    }
    func(state[p], p, valp);
}
```

Entsprechend: out.

Bridges reichen bus_{in,out}_info-Aufrufe einfach weiter.



- bei Speicherzugriffen muss nichts simuliert werden
- i.a. existiert Speicher immer in Vielfachen von ganzen Pages (x86: 4 KByte)

=> direkter Zugriff der CPU auf den Speicher

Ausnahme: Zugriff auf Memory-mapped-I/O-Bereiche (z.B. Grafikkarte)!



```
void
bus_load_info(
    struct bus *bus,
    uint32_t addr,
    uint8_t **memp
) {
    unsigned int i;

    *memp = NULL;
    for (i = 0; i < bus->nmembers; i++) {
        bus->member[i].func->load_info(
            bus->member[i].state, addr, memp);
    }
}
```

Entsprechend: `bus_store_info`. Evtl. existiert noch eine Funktion zum Invalidieren der gecachten Speicheradressen (wenn physikalischer Speicher sich ändert).



```
uint8_t *mem[1 << (32 - 12)];

uint8_t load(uint32_t addr) {
    uint8_t *p;

    p = mem[addr >> 12];
    if (! p) {
        bus_load_info(bus, addr, &mem[addr >> 12]);
        p = mem[addr >> 12];
        if (! p)
            return bus_load(bus, addr);
    }
    return *(p + (addr & 0xfff));
}
```

Entsprechend: store.



Ergebnisse dieses Abschnitts:

- Man kann komplexe Systeme aufbauen,
- man kann die einzelnen Komponenten gut voneinander trennen,
- man kann die einzelnen Busse gut voneinander trennen,
- man kann die Systeme konfigurierbar machen,

ohne große Performance-Einbußen hinnehmen zu müssen.

Motto: Cachen, was cache-bar ist...!

