1.1 The Machine

The machine used here is a Digital Equipment Corporation LSI 11/2 16 bit microprocessor.
A microcomputer version of the PDP II.

Discussed:

- processor
- memory
- communication devices
- architrave
- asynchronous communication devices
- disk storage devices
- mechanisms like :
  - stack
  - vectored interrupts
  - device addressing

1.2 LSI 11/2 Physical organization

The LSI 11/2 constructed from:
Printed Circuit Boards
Slotted into the *sockets* of a *backplane* see figure 1.1

These sockets are wired together to form a *Bus* (the Q-bus) consisting of:

- **power lines** attached to all board in in parallel
- **lines linking up the sockets for inter-board communication** : Signals travels to the board on one contact and away on another.

Boards :

- One board contains the 11/2 processor itself:
  at the beginning of the bus.
- other boards contain memory or device interfaces I/O devices.

Communication between boards:

- A board communicate withe another board by passing signals across the bus
- E.g., when the processor board needs to write int memory, it places **address + data** on the bus for the memory board to retrieve and store.
- memory is **logical contiguous**
  - each board contains **switches** (hardwired jumpers) that can be changed:

    It is possible to configure tow identical memory boards so that
3 one responds to **low memory addresses** and other to **high address**
  - Similarly, I/O devices interface boards contains such switches to represent tow distinct devices.
- The physical order of boards along the Q-bus determines their **priority**
- Signals enter a board one connector and leave it on another one so the board can decide whether to intercept the signal or pass it on down the bus.
- Example: tow devices waiting for service

  - The CPU sends a **grant signal** down the bus.

- It passes from board to board until it reaches a board ready for service.
- The first board with a device waiting responds to the signal without passing it on.
- Boards further down the bus wait until the CPU finishes servicing the first board and reissue the grant signal.

1.2.1 Logical Organization Of The 11/2

5 The Operating system is concerned with the logical organization of the machine, not its physical organization.

1.2.2 Registers In The LSI 11/2

6

Hier kommt Figur 2.2

1.2.3 The Address Space

- Memory divided into 8-bit quantities called **bytes** - also called **characters** -
  with the byte being the smallst addressable unite.
- Most instructions operate on tow bytes (a **word**)

7

  The operation can alwys affect the addressed byte and the next higher byte.

- Figure 2.4 :
  - 11/2 addresses 64k bytes of memory (k = 1024)
  - addresses formulated in **octal**.

Hier kommt Figure 2.3

1.2.4 Vectored Interrupts

The Le LSI 11/2 uses the **vectored interrupts scheme** for handling exception and interrupts from external devices.

Whenever an external device must communicate with the processor :

8 the device places a signal on the interrupt bus line

If the processor runs with the **interrupts enabled**

it checks the interrupt line after executing every instruction

To handle an interrupt, the processor sends an **acknowledgement** over the bus, requesting the interrupting device to return an **interrupt vector address (IVA)**

The first device with a pending request receive the acknowledgement and responds by returning its interrupt vector address v.

hen the CPU receives the vector$v$ from the Q-bus

- the processor pushes current value of **PC** and **PS** on the stack, and
- loads a new PC and PS from tow words in memory starting to location v, and
- continues execution at the new location addressed by the PC.

9

See the figure below.
Each device is assigned a unique IVA, enabling the system software to (identify/distinguish) among them.

The PC stored at v points to an **interrupt service routine** (interrupt handler) for the interrupting device.

The interrupt "acts" line a procedure call "inserted (invisibly) by the hardware in between tow instructions in user's code"

The processor executes the code in the interrupt routine returning to the place where the user's process are interrupted.

Hier Kommt eine Figure

- To make the interrupt invisible (transparent) to the running program,

  the interrupt handler must save and restore the state of the machine.
- On the 11/2: saving the state means saving R0-R6, since the hardware saves R7 (i.e., PC) and PS.
- In practice R6 (i.e., PS) need not b saved

  provided the interrupt routine pops off before returning whenever it pushed on the stack before, restoring the stack to its original position
- To prevent an interrupt routine from itself being interrupted, the stored PS (in the IVA) specifies a priority level 7, i.e,

  As soon as interrupt processing begins, the processor operates with interrupt disabled.
- Interrupt processing ends when the processor

  execute a "return from interrupt" instruction (*rti,rtt*),
- the old value of the PC and PC are restarted (before executing the interrupt) from the stack, and
- returning the SP to its original value.
- receiving the action taken by the processor when it detected the interrupt, allowing processing to continue as if nothing had

happened !

11

1.2.5 Exceptions

Exceptional condition are handled like interrupts.

Example (exceptions):

- reference to nonexistent memory
- executing of invalid instruction code
- power failure
- hardware detected error

Similarly as above :

- PC and PS are pushed onto the stack
- New values for PC and PS loaded from memory.

Exception vectors are permanently assigned to memory locations 0-27.

They, unlike interrupts, can not be disabled by changing the priority level.

1.2.6 Asynchronous Communication

- An **asynchronous communication device**, **the serial line unit**, sends and receive characters (to/from a conventional terminal).
- At least one board in any 11/2 system contains such a device, connecting the computer to a terminal.
- Cable connecting the device to terminal. consists of three wires:
  1. carries data into the computer
  2. carries data away from the computer
  3. ground voltage wire
- Signals are **serial** because they travel down the line one bit-at-a-time
- Signals are **asynchronous** because the transmitter sends a character whenever one is available
  there is no synchronization between the transmitter and receiver to control the start of a character.
- The work to convert characters into a series of pulses or to covert a series of pulses into characters is carried out by a single integrated circuit (IC) controlled by precise clock called **(UART) Universal Asynchronous Receiver and Transmitter**.
- Errors occur when a receiver can not make series out of the signals it receives.
  The receiver itself holds one character at a time.
  CPU must extract each character before subsequent characters arrive

or the receiver indicates an overrun condition and replace the earlier character with the latest one.

1.2.7 LSI 11 Asynchronous Serial Line Hardware

Conceptually:

asynchronous transmitter and receiver grouped into pairs consisting of one receiver and one transmitter to model tow-way communication.

1.2.8 Addressing a Serial Line Unit

CPU containing device interfaces watch the bus for addresses corresponding to their devices, and respond accordingly.

From the programmer's pint of view, each SLU (**Serial Line Unit** contains 4 16-bit registers, see figure 2.5

- **RCSR** (receive control and status)
- **RBUF** (receiver data buffer)
- **XCSR** (transmitter control and status)
- **XBUF** (transmitter data buffer)
- To communicate with the device, a program reads or writes the address corresponding to the device register address, just as it reads or writes to a real memory address.
- Device registers are not storage locations.
- Rather, the interface board(containing the device) intercepts the transfer and interprets it as an instruction to :
  1. transfer data between the device and CPU
  2. control the device

Examples :

1. Writing a character to the XBUF register causes the SLU to capture that character and throwing it down to the SLU.

   Once the board accepts a character, it operates independent of the CPU (processing continues while the transmitter sends the character).

17

2. CPU receives a character received over the SLU by reading from the RBUF address, see figure 2.6 below.
   ——- Hier kommt eine Figure———-

RBUF read only : write directed to that address are ignored.
Accessing RBUF has an side effect of clearing the receiver and enabling it receive the next character.

1.3 The C Run-Time Environment

Operating System are written in heigh-level language to make them easier to write, understand, debug, and move to other machines.

However, sometimes machine assembly language procedures are
18 introduced because :

machine quantities must be manipulated, e.g. for :
saving and restoring the machine's registers context-switching code, interrupt routines manipulating the process table, implementing semaphores

The **C compiler** expects each program to run in an address space as shown in figure 2.14.
HIER KOMMT EINE FIGURE
Symbols **_etext, _edata, _end** refer to global variables inserted into object code by the loader.

They are initialized to first address beyond **text, data, bss** segments respectively .
Thus running program can find out how much free memory remains between end of loaded part and the current top of the stack by taking the address of **_end**.

- **text** segment : contains text of main program and all its procedure.
- **data** segment : contains all initialized data.
- **bss** segment : contains all uninitialized data.
- **stack** segment : occupies highest part of address space and grows downward
- **heap** storage : allocated from **_end** upwards, i.e., from the bottom of the free space.

In Xinu:

- When it runs, multiple processes have stack allocated as in figure 2.15
  HIER KOMMT EINE FIGURE

20

- For each process is separate stack allocated from the highest available free space, so stack overflow in one process destroys stacks of other processes.
- Stack space returned to free space whenever process exits.

Convention for translating procedures in the C-compiler

Consider **procA :....; B(arg1, ...,argn);**

- A is calling procedure
- B the called procedure

How is the code of calling B within the procedure body of A in C's compiler ?

1. Values of actual parameters of B pushed on stack (in reverse order)
2. Address of return address in A, i.e., of the instruction follwing call to B pushed on the stack
3. Flow of control branches to B

2 + 3 curried ou t by calling

jsr pc, address-of-B

Calling procedure, A, also responsible for popping B's args from the stack after the called procedure, B, returns.

Code translating called procedure B :

The called procedure is responsible for:

- saving and restoring the machine state, i.e., the contents of the used registers
- popping the return address from the stack before returning by calling

rts pc

Hardware provides a single instruction

rts pc

that pops an address from the stack and returns to that address.

- Saving the contents of the used registers is done by calling assembly language routines **csv**.
- Restoring the contents of the used registers is done by calling assembly language routine **cret**

csv and cret :

csv - C registers save routine

cret - C registers restore routine

To save the machine registers, the compiler inserts a call to *csv*, an assmbly language routin.
csv:

- saves the register
- jumps back to called routin B

When code for B is finished, compiler inserts call to
23 assembly language routine *cret*, to restore the values of machine's registers and the **stackpointer sp** (i.e., R6) and to return control to the original caller:
cret :

- restore old values of machine registers
- restore sp
- returns control to original caller

Code for B :

```
/* csv.s - csv, cret */
/* C register save: upon entry here, procedure A has called B  */
/* and B has called csv to save registers. r5 contains return   */
/* address in B. The stack has old r5, return address in A, and */
/* arguments on it. C return: cret (below) is used to restore   */
/* regs when the called proc. exits                             */

        .glob   csv, cret
/*-------------------------------------------------------------*/
/* csv  --  C register save routine
/*-------------------------------------------------------------*/
csv:
24
        mov     r5,r0     / r0 not saved at call (C convention)
        mov     sp,r5     / r5 points to called routine's frame
        mov     r4,-(sp)  / push r4 -r2 on stack
        mov     r3,-(sp)
        mov     r2,-(sp)  /jsr pushes PC onto stack hoes to
        jsr     pc,(r0)   / address in r0 (originally in r5)

/*-------------------------------------------------------------*/
/* cret  --  C register restore routine
/*-------------------------------------------------------------*/
cret:
        mov     r5,r2     / put copy of called frame ptr in r2
        mov     -(r2),r4  / reload r4 - r2 from start of frame
        mov     -(r2),r3
```

```
        mov     -(r2),r2
        mov     r5,sp       / restore SP
        move    (sp)+,r5    / restore r5 on stack by call
25                          / to csv procedure entry
        rts     pc
```