9. CPU Structure

(a) Basic Structure.
(b) Data Lines.
(c) Control Lines.
(d) Data Flow during an Instruction Cycle.
(e) The Control Unit.
   (i) Hardwired CU.
   (ii) Microprogramming.
The CPU has

- **Registers** - in order to store temporarily data.
- An **ALU** - Logic and **Arithmetic Unit** - in order to carry out arithmetical and logic operations, especially addition.
- A **CU** - Control **Unit**, which controls the flow of information.

All these units will be connected by

- **Data Paths** - these connect data storage components together; addresses will be treated similarly to data.
- **Control Paths**, which carry control signals.
There are two kinds of registers:

- **User-visible** Registers.
- **Control and Status** Registers.
User-visible Registers

There are

- **General purpose registers.**
  They are used like main memory, but with faster access for frequently used data. Can be read and written by the machine language.

- **Condition codes,** often referred to as **flags.**
  Examples are overflow or result zero flags from last ALU operation. Condition codes are usually set by the ALU and (sometimes) by the CU and only read by the user.
  However some might be changed (in exceptional cases) by the machine language.
More on User-visible Registers

- Sometimes the **general purpose registers** are divided into
  - **Data registers**, which can store only data.
  - **Instruction registers**, which can store only instructions.
  This is useful in order to reduce address space for register references: Since for most operations it is clear whether they refer to data or addresses, one needs one bit less for their address.
  Sometimes **finer distinctions** (for instance floating point number registers) occur.

- **Condition Codes** are especially stored when an interrupt occurs and a jump to an interrupt handler is carried out.
Control and Status Registers

These are registers, used while carrying out the instructions in the CPU. Typical ones are (they differ very much between different processors):

- **Program Counter (PC):** Contains the address of the next instruction to be fetched.
- **Instruction Register (IR):** Contains the most recently fetched instruction.
- **Memory Address Register (MAR):** Contains the address of a location in memory.
- **Memory Buffer Register (MBR):** Contains data which is to be stored into memory or was read from memory. (When talking about main memory in the following we mean main memory accessed via the cache; a main memory read or write might actually result only in a cache read and write operation).
Some Optimizations of Control Registers

- **Memory Data Register (MDR).** Instructions can bypass the Memory buffer register and be loaded directly from memory into the instruction register. Then, if the output of main memory is data to be loaded, the information is passed on to a different register, called **MDR**, which replaces essentially the **MBR**. When storing data into main memory, which is coming from a register (not from ALU) an intermediate storage is not needed. So the **MBR can be omitted**.

- If whenever an address is passed on the address bus connected to main memory, the origin is a register in the CPU, then the **MAR** can always be bypassed, and therefore **omitted**.

  (Remark: **MAR** and **MBR** might however occur in the exam).
(b) Data Lines
Data Paths in a CPU
Remarks on the Data Flow Diagram

- The **principal design** is such that all connections are
  - either **from** some **register** to some **register**, or
  - **from the register unit output** or **data bus** of main memory **to** some **register**, or
  - **from** some **register** to the **data input** (Rdatain) of the register unit (Rdatain) or **to the system bus**.

Other connections would not be finished within one memory cycle.
• In general a CPU will not have all data paths shown.
• Some additional registers might be present.
• Some shift logic required is not shown.
• The main memory is of course not part of the CPU, only the bus system bus is connected with it. The picture given makes it easier to describe the data flow.
• The data bus is usually bidirectional. We have separated it for simplicity into “data bus (in)” and “data bus (out)”. 
• Some data paths will be permanently open:
  – From the outputs of the register unit into registers A, B.
  – From the ALU to register C.
There is a **register unit** which administers the general purpose registers. It has:

- Two **input lines** (R#1in, R#2in) for providing **register numbers** of two registers simultaneously.
- Control lines for asserting read or write to the register unit.
- Two output lines (R1out, R2out), where, if read is asserted, the **content** of the registers specified by R#1in, R#2in, will be output.

This allows to look up two arguments of an arithmetic operation with register addressing mode in one go.

- Data lines in (Rdatain), for providing **data to be stored in a register**. (The number of the register is specified in R#1in.)
• Data paths will often be split into two or more, which can operate independently. Especially the lines from the IR are separated into
  – lines corresponding to up to two register numbers
  – lines corresponding to memory addresses or immediate data.
Explanation of the Data Lines

- From the **PC** there are data paths to
  - The **address bus** of the system bus.
    * Needed to fetch the next instruction.
  - **One argument of the ALU.**
    * Needed in order to compute the next instruction:
      Add 1,2,4 or 8 in case of no branch/jump.
      Add offset in case of branch/jump.
- From **Data Bus (out)**, data paths exist to
  - **IR.**
    Used if an instruction was fetched.
  - **MDR.**
    Used if Data was fetched.
From the IR, data paths exist to

- **R#1in** and **R#2in**.
  - Used in case of addressing modes involving register numbers.

- **Rdatain**.
  - Used, if immediate data is to be stored in a register.

- The two arguments of the **ALU**.
  Used for transmitting
  - offsets in case of displacement addressing modes,
  - immediate data on which ALU operation is to be performed.

- **Data bus (in)** of main memory.
  - Used, if immediate data is to be stored in main memory.

- **Address bus** of main memory.
  - Used, in case of direct or indirect addressing mode, in order to fetch data or addresses from main memory.
From the **MDR**, data paths exist to

- **Rdatain** for the register unit.
  - Used, in order to store data fetched from memory into a register.
- the two arguments of the **ALU**.
  - Used if operations are to be performed on fetched data.
  - In case of displacement addressing modes, used for calculating the address.
- **Data bus (in)** of main memory.
  - In order to store data back to main memory.
  (In case of bidirectional system bus this means that data is sent back to the data bus).
- **Address bus** of main memory.
  - Used, in case of indirect addressing mode.
The **output of the register unit** will always be stored in register **A, B**. (Connection is permanently open).

From the **registers A, B** data paths exist to

- The two arguments of the **ALU**.
  - Used for carrying out arithmetical operations on registers.
    This is the main and most efficient use of registers.
  - Used as well for calculating addresses in case of displacement addressing modes involving registers.

- **Data bus (in)** of main memory.
  - In order to store data from a register in main memory.

- **Address bus** of main memory.
  - Used, in case of register indirect addressing mode.
• The **output of the ALU** will always be stored in register **C**. (Connection is permanently open).
• There is as well a **direct connection to the PC** in order to store the result of a calculation of the address of the next instruction in it. (All other connections from the ALU need to be buffered by C, since they don’t end directly in registers).
From register C data paths exist to

- The two arguments of the ALU.
  - Used when more arithmetical operations which require multiple cycles are be carried out.

- Rdatain of the register unit.
  - In order to store the result in a register.

- Data bus (in) of main memory.
  - In order to store the result in main memory.

- Address bus of main memory.
  - Used, if an address was calculated in an indirect addressing mode with displacement.
(c) Control Lines

![ALU Diagram]

- Flags
- 1st Argument
- 2nd Argument
- Result
- Control Signals
  (+, -, AND, OR, ...)

293
More on the ALU

- The arithmetic and logic unit is a **blind calculating engine**.
- It has as **input** usually **two arguments** and **control signals**, indicating the operation to perform, and as output the result plus some flags.
- There are additional **shift operations** and **tests** carried out in the CPU, which don’t belong to the ALU itself.
Implementation of the ALU

- Integer addition and subtraction can be implemented by adders directly, and require only one clock cycle.
- Integer multiplication can be implemented by a cascade of adders. This requires however several clock cycles.
- Floating point operations require more clock cycles.
Example: Pentium

- Integer multiplication requires 4 cycles, but the multiplication unit is pipelined, which means that it is carried out in 4 consecutive steps, after each of which the next multiplication operation can already start, giving ideally a throughput of 1 operation per cycle.
- Floating point addition requires 5 cycles and is pipelined, giving ideally a throughput of 1 operation per 2 cycles.
- Floating point division is not pipelined and requires many cycles (not specified).
Control Paths to CU

The control Unit controls the flow of data in the CPU.

- There are **control paths to the CU** from
  - the **IR**,
  - the **flags of the ALU**,  
  - **control lines from the system bus**:  
    Interrupts raised, successful/unsuccessful loading of data from main memory.  
    We will ignore the control lines from the system bus in this lecture.
Control Paths from the CU

The CU has control paths to:

- switches, which open/close data paths. (The connections from the register unit to A, B, and from ALU to C can be kept permanently open, and have no switch).
- main memory to assert read or write operation,
- the register unit to assert read or write operation,
- the ALU, to determine which arithmetic/logic operation to perform,
- the constants which are possible arguments for the ALU (which one),
- shifting units. (We will ignore them in this lecture),
- other control lines of the system bus, for dealing with interrupts, delayed memory responses etc. (We will ignore these).
States, Clock and CU

- The CU has further some internal memory, which stores the current state of the CU.
  (Every instruction requires several clock cycles, during which different operations are carried out. See for instance the steps carried out during a multiplication).
  Each such cycle corresponds to one state of the CU).
- Further a clock signals is provided to the CU.

Operation of the CU

Depending on
- the current state and
- the incoming control signals
  in one clock signal the CPU
  - determines the outgoing control signals,
  - and the next state, which is then updated.
How the CU and its states are organized will be discussed later.
Control of Data Paths

The next slide shows how the connection from one register to another can be controlled.

- Registers are built from flip flops, in this case D-latches.
- The output of one flip flop (line “Q”) is via a switch connected with the line “D” of the next flip flop.
- Other connections lead to the same flip flop.
- We assume that only one data path to the second flip flop is enabled at a time.
- If the path from the first flip flop to the input of the second one is opened and the clock signal is on, the second one stores the input from the first flip flop.
- If the control signal is off and no other control signal is on, the second flip flop preserves its current state.
Flip-flop (D-latch) 

Data Line 

Other data line 

Other control line 

Control Line 

Data Line 

D Flip-flop (D-latch) 

C 

Inputs controlled by other control lines
More on Controlling Data Paths

- Often some logical operations are inserted in between (e.g. in ALU), which take the outputs from several flip flops, operate on them (e.g. plus, AND) and have as result one input to the flip flop on the right hand side.

- Instead of having one control signal for each data line going into one unit one can use **multiplexers** which get as input several data data lines lines and control signals providing a binary number of a line and has the chosen line as output. Therefore with for instance three control lines one can control \(2^3 = 8\) data lines entering into the same unit.
Timing

Timing is sophisticated, especially since we will eventually send signals from a register to itself, with some registers in between. If there is no delay then we will get feedback loops and oscillations as in the unstable state of a flip-flop.

Solving this is the crucial step from a combinatorial circuit to a sequential computer with several states.

Here follows a first approximation to how to deal with this problem:

If the first and second D-latch are clocked by inverted clocks (ie. the clock signal of the second flip-flop is the negation of the clock signal of the first one),
then, if control is on, the second flip flop will be updated half a clock signal after the first one is updated.
Let
- \( Cl \) be the clock,
- \( \overline{Cl} \) its inversion,
- \( I1 \) a (random) input to the first D-latch,
- \( O1 \) its output (the line marked \( Q \) in the D-latch),
- \( O2 \) be the output from the second D-latch.
- and assume that the connection is open (and all others are closed).

<table>
<thead>
<tr>
<th>( Cl )</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( I1 )</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( O1 )</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( Cl )</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( O2 )</td>
<td>?</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- The input boldface 0 to the D-latch 1 is
- output from the first D-latch only when the clock is 1,
- recognized by the second D-latch when its clock is 1,
- which is half a clock cycle later.
Similarly the italic 1 arrives half a clock cycle later.
If one wants to feed back a signal from a D-latch to itself, one has to guarantee that an odd number of D-latches is in between. If one has $2k + 1$ D-latches in between, the result arrives after $k+1$ clock signals.
(d) Data Flow during an Instruction Cycle

We will look now at the flow of data during an instruction cycle. We will look at the following examples:

(i) Instruction fetching.
(ii) Instruction operation decoding.
(iii) Operand fetch: immediate addressing.
(iv) Operand fetch: direct addressing.
(v) Operand fetch: indirect addressing.
(vi) Operand fetch: register addressing.
(vii) Operand fetch: register indirect addressing.
(viii) Operand fetch: base-register displacement.
(ix) Data Operation.
(x) Operand Store.
(xi) Instruction Address Computation.
**General Outline**

For each example we will do the following:

1. We will specialize the data flow diagram of the CPU and include only the units and paths involved.
2. We will add labels to all control paths involved.
3. We will describe the instruction cycle.
4. We will give a microprogram for this specialized situation.

We first review briefly the instruction cycle (we omit the interrupt cycle).
Picture omitted for copyright reasons
(i) Instruction Fetching

Only one cycle required:

- Data path (a) from PC to the address bus is opened.
- Read from main memory (b) is asserted.
- Data path (c) from data bus (out) to the IR is opened.
  - By the first three items, at the beginning of the next cycle, *instruction with address given in the PC will be available in the IR.*
- Data path (d) from PC to argument 1 of the ALU is opened.
- Constant 1 (e) is asserted for the second output and path opened for it to the argument 2 of the ALU.
- Data path (f) from ALU to PC is opened.
- Addition (o) is asserted at the ALU.
  - By the last four items, at the beginning of next cycle *PC will be incremented by 1.*
Remarks

- The example given requires that the next instruction is (unless a jump/branch is followed) at address given by the PC plus one. If it is at PC + 2, or PC + 4 (for instance if instruction length is 2 or 4 bytes and addressable unit is byte), the constant has to be changed.

- If the instruction is longer than what can be fetched in one cycle, fetching has to be repeated. If the length is fixed, one simply repeats the above several times. If we have variable length, one loads the minimum length and then makes a decision. This might take an extra cycle: If the decision depends on the last piece of instruction fetched, one cycle is needed to decide whether an instruction has to be fetched or not. Alternatively, fetch in all cases the next instruction.
Microprogram for Instruction Fetch Cycle

We write only the control signals which play a role during this cycle. The full microprogram will have columns for all other control signals, which will all be set to 0.

The microprogram is:

<table>
<thead>
<tr>
<th>Line-#</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>o</th>
<th>Next #</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</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>

“Line #” is the line in the instruction, “Next #” is the number of the next instruction.
If one had 1 or 2 instructions to be fetched, depending on a bit $h'$ from the first instruction, and $g'$ means that the output from the data bus is moved to the second location of IR instead of the first one the micro program is:

<table>
<thead>
<tr>
<th>Line-#</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g'</th>
<th>o</th>
<th>Next #</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1'</td>
</tr>
<tr>
<td>1'</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>if $h'$ then 2' else 3'</td>
</tr>
<tr>
<td>2'</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>3'</td>
</tr>
</tbody>
</table>
(ii) Instruction Decoding

One Cycle required.

- Depending on the instruction fetch choose next microprogram line.
- The microprogram asserts no control line, just chooses a next micro program line:

<table>
<thead>
<tr>
<th>Line-#</th>
<th>Next #</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Choose next instruction depending on bits from the instruction</td>
</tr>
</tbody>
</table>
• One might carry out decoding earlier, if it takes several cycles to fetch the instruction, and decoding can be done on the basis of earlier parts of the instruction.

• In all cases one might already do an operation which is likely to be done:
  – Send fields which correspond to register numbers to $R\#1\text{in}$, $R\#2\text{in}$ and assert read from the register unit.
    (Not harmful if this is not needed; usually done).
  – Similar optimizations may be done in other states.
(iii) Operand fetch: Immediate Addressing Mode

The operand is part of the instruction. When the data is needed, a data path is opened to where the result is required. No cycle needed.
(iv) Operand Fetch: Direct Addressing
(iv) Operand Fetch: Direct Addressing

One cycle required:
- Data path (g) from the address field in the IR opened to the address bus.
- Read from main memory (b) is asserted.
- Data path (h) from data bus (out) to MDR is opened.

At the beginning of next cycle, data will be in MDR and can be used for the operation. The microprogram is (again only control lines used written down; note that (g) refers to the bits in the address field of the IR):

<table>
<thead>
<tr>
<th>Line-#</th>
<th>b</th>
<th>g</th>
<th>h</th>
<th>Next #</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Line # for second operand fetch</td>
</tr>
</tbody>
</table>
(v) Operand Fetch: Indirect Addressing
(v) Operand Fetch: Indirect Addressing

Two cycles required:

**Cycle 1**  
- Data path (g) from the address field in the IR opened to the address bus.
- Read from main memory (b) is asserted.
- Data path (h) from data bus (out) to the MDR is opened.

At the beginning of cycle 2, EA will be available in MDR.

**Cycle 2**  
- Data path (i) from MDR to the address bus is opened.
- Read (b) from main memory is asserted.
- Data path from data bus (out) to the MDR (h) is opened.

At the beginning of cycle 3, operand is in MDR.

The microprogram is:

<table>
<thead>
<tr>
<th>Line-#</th>
<th>b</th>
<th>g</th>
<th>h</th>
<th>i</th>
<th>Next #</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Line # for second operand fetch</td>
</tr>
</tbody>
</table>
(vi) Operand Fetch: Register Addressing
(vi) Operand Fetch: Register Addressing

One cycle required:

- Data path (j) from the bits in the address-field in the IR to R#1in is opened (to R#2in instead if second operand is to be fetched).
- Read from register unit (k) is asserted. (Note that connection from R1 out to A is always open).

At the beginning of next cycle, operand is in A (or B).

The microprogram is (for first argument):

<table>
<thead>
<tr>
<th>Line-#</th>
<th>j</th>
<th>k</th>
<th>Next #</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>Line # for second operand fetch</td>
</tr>
</tbody>
</table>

**Remark:** If one passes the register number during the instruction decoding to R#1in, R#2in, no cycle needed.
(vii) Operand Fetch: Register Indirect Addressing
(vii) Operand Fetch: Register Indirect Addressing

Two cycles required:

**Cycle 1**
- Data path (j) from the address field in the IR (containing the register number) to R#1in is opened.
- Read (k) from register unit is asserted. At the beginning of cycle 2, EA is available in A.

**Cycle 2**
- Data path (l) from A to the address bus is opened.
- Read (b) from main memory is asserted.
- Data path (h) from data bus (out) to the MDR is opened.

At the beginning of cycle 3, operand is in MDR. The microprogram is:

<table>
<thead>
<tr>
<th>Line-#</th>
<th>b</th>
<th>h</th>
<th>j</th>
<th>k</th>
<th>l</th>
<th>Next #</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Line # for second operand fetch</td>
</tr>
</tbody>
</table>
(viii) Operand Fetch: Base Register Displacement
(viii) Operand Fetch: Base-Register Displacement Addressing

(EA = (R#)+ A. Three cycles required:

**Cycle 1**
- Data path (j) from the bits in address field in the IR containing the register number to R#1in is opened.
- Read (k) from register unit is asserted.

**Cycle 2**
- Data path (m) from A to the first argument of ALU is opened.
- Data path (n) from the bits in the address field in the IR containing the base address to the second argument of ALU is opened.
- Add (o) asserted to ALU.

(Might be more complicated since variants of addition depending on data exist).

At the beginning of Cycle 3, EA is in C.
Cycle 3  – Data path (p) from C to the address bus is opened.
  – Read (b) from main memory is asserted.
  – Data path (h) from data bus (out) to the MDR is opened.

At the beginning of cycle 4, operand is in MDR.

The microprogram is:

<table>
<thead>
<tr>
<th>Line-#</th>
<th>b</th>
<th>h</th>
<th>j</th>
<th>k</th>
<th>m</th>
<th>n</th>
<th>o</th>
<th>p</th>
<th>Next #</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>o</td>
<td>0 9</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Line # for 2nd operand fetch</td>
</tr>
</tbody>
</table>
(ix) Data Operation

Assume that after the operand fetch, one operand is in A (output of the register unit) and the other in the MDR. Assume further that the ALU operation requires only one cycle.

One cycle required:

- Data path (p) from register A to first argument of the ALU is opened.
- Data path (q) from MDR to second argument of the ALU is opened.
- Appropriate ALU operation (r) is asserted.

Since path from ALU to C is always open, at the beginning of the next cycle the result of the operation is in C.

The microprogram is:

<table>
<thead>
<tr>
<th>Line-#</th>
<th>p</th>
<th>q</th>
<th>r</th>
<th>Next #</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>12</td>
</tr>
</tbody>
</table>
(x) Operand Store

This is done similarly to operand fetching, by first calculating the EA, if necessary using several cycles, and then moving the value to be stored (which is in C or in case of a pure store operation in the MDR, A or B) to data bus (in) of main memory or Rdatain of the Register Unit, asserting a write control signal.
(xi) Instruction Address Computation

- Case 1: no branching/jump instruction: new PC value was already calculated, jump directly to instruction fetch. No cycle required.
- Case 2: jump with relative addressing mode: Add to PC offset contained in the IR and feed back result to PC. One cycle required.
Case 3: **branching instruction:**
In case of a comparison of two registers, a data operation was carried out which had as result a result not needed and setting of a flag. Otherwise decision on the basis of flags already set.

**One or two cycle** required:
Depending on the flag which determines the condition (set by last ALU operation in case the branching instruction required comparison of two operands — in this case the operand was not to be stored and ALU operation only needed to set the flag):
- either jump directly to instruction fetch or
- carry out a similar operation as for the jump instruction.
• Arithmetical operations (including multiplication which takes several cycles) and operand store can be organized similarly.

• Branching instructions are dealt with by first computing the boolean value of the equation in the branch condition, and then, if it is true (determined by a flag), opening a data path from PC to the ALU, from the offset in IR to the other argument of the ALU and a path back to the PC.

• Current structure allows in general only one non-register addressing mode. If one wants several, one might use registers from the register unit (add new ones with new register numbers), or new extra internal registers.

• For floating point arithmetic as well additional registers needed.

• Final microprogram has lots of bits for control signals, several small case distinctions and one (with longer instruction sets a few) bigger case distinction.
**Remark:** One can avoid a complex logic in the CPU and instead have simple addressing modes and operations (for instance not even multiplication). Complex instructions can either be interpreted as small instructions in hardware (done for the Pentium) or one can translate programs written in a complex assembler language into simple instructions (translation on compiler level).
(e) The Control Unit

There are two ways of implementing the control unit.

(i) Hardwired Control Unit
(ii) Microprogramming.
Picture omitted for copyright reasons
(i) Hardwired Control Unit

Enumerate all states of the CU. Store the number of current state in a state register.

Define a circuit which, depending on

- the incoming control signals and
- the current state number,
determines

- the outgoing control signals and
- the number of the next state.

Connect now this circuit with control lines and state register as in the next slide.

Now the following simple loop is carried out:

For every clock signal

- **Enable the outgoing control signals** determined by the circuit,
- **Store next state** in the state register.
(i) Hardwired Control Unit
(ii) Microprogramming

- Write for every state of a CU a microprogram instruction which contains:
  - information of all the outgoing control signals,
  - information how to select the next state depending on the incoming control signals.

Now construct a small computer with the following components:

- It has a small memory containing all micro program lines. The content is called microprogram or firmware, since it is half between software and hardware.
- It has a logic, which for every micro program line determines the outgoing control signals and the next microprogram line.
Problems:

- Control signals require a lot of storage. Encode them in a more compact way, which can be decoded easily. (Remark: such an encoding not required in coursework 2).

- Determination of the next instruction complicated. Three main cases:
  - In most cases next line to be selected.
  - In some cases jumps or depending on incoming control signals next line or another line to be selected.
  - In few cases large case distinction depending on incoming signals into many different cases to be selected.
**Solution for problem of selecting next line:**
Introduce an **address select logic** which, depending on information encoded in the microprogram and control signals:

- either selects the current microprogram number increased by one or
- determines using
  - a hardwired circuit,
  - a branch table
  - or an address directly written into the microprogram

the next micro program number.

The condition for a branch (depending on which signals to choose the branch) might be hardwired or encoded in the microprogram.
(ii) Microprogrammed CU

Microcode Memory

Control Signals out

Mirroprogram Counter

Information about branch selection, next address, encoded into the micro program

Address Select Logic

Incoming Control Signals

Adder

1
Comparison

- **Hardwired CU:**
  - Is faster.
  - Implementation is very complex and sensitive to errors.
  - Main technique used for RISC instruction sets.

- **Microprogrammed CU:**
  - Easier to implement.
  - Easy to change if mistake occurs (not much change in the factory).
  - Can even be programmed in some cases (emulation of other architectures possible).
  - However slightly slower.
  - Main approach taken for CISC instruction sets. (Implementation of CISC processors require extremely many states).
Summary Microprogramming vs. Hardwired CU

- Hardwired CU:
  Selection of next state and control signals done by a circuit.

- Microprogramming:
  Microprogram is stored in some memory in the CU.
  Only a small selection logic for selecting the next line in the microprogram is implemented directly as a circuit.
  CU is essentially built as a small computer.