Materials

W4 Procesor (tryby pracy, współpraca z koprocesorem matematycznym, jednostka sterująca i jej mikrokod).

Omawiane zagadnienia

- Specialized processor extensions
  - Privileged processor modes
  - Doing floating-point calculations
- Processor structure and function
  - Controlling control unit with micro-operations

Interrupts and exceptions

The distinction between the terms interrupt and exception is not precisely defined. Interrupt processing and exception processing typically rely on the same or similar processor hardware resources operating in much the same manner.

Hardware interrupts allow processors to respond promptly to requests for services from peripheral devices. A hardware interrupt notifies the processor of a need to take some action, usually involving data transfer to or from an external device.

Exceptions are similar to interrupts, with the key difference that **exceptions generally involve responding to some condition arising internal to the processor**. One example of an internally generated exception is division by zero.

It is possible for user code to intentionally generate exceptions. In fact, this is a standard method used by *unprivileged* code to request system services provided by *privileged* code in the operating system kernel and device drivers.

Exceptions include error conditions occurring during sofware execution such as division by zero, as well as non-error situations such as page faults. Unless the interrupt or exception results in the drastic response of terminating the application, an interrupt service routine or exception handler processes the event and returns control to the system scheduler, which eventually resumes execution of the interrupted thread.

Hardware interrupts and system exceptions tend to occur at random times relative to the ongoing execution of program code, while behavior related to sofware-generated exceptions is generally repeatable if the same code executes again while operating on the same data

Privileged modes

The idea of protection rings

You can think of the protection strategy employed by modern processors and operating systems as similar in significant ways to the defenses implemented in the designs of medieval castles.

The most privileged members of the castle hierarchy enjoy unrestricted access in and out of the castle and the outer wall. Less privileged members of the society may have authorization to pass inward through the outer wall but are prohibited from accessing the castle directly. The least privileged members of the local population are prohibited from entering the castle under most conditions and may have to accept limitations on what they can do on occasions when they are granted access, such as being permitted to access only specified public areas. The protections provided by this strategy can be represented as concentric rings, with the highest privilege required to access the innermost rings, while the outer ring represents the area requiring the least privilege.

This protection strategy contains three privilege levels that determine the types of access available to each individual in the medieval castle inhabitants. Ring 0 requires the highest privilege level, while ring 2 requires no special privilege.

Although, in principle, it is possible to provide multiple intermediate levels between the highest and lowest privileged rings, most modern computer architectures implement just two rings: a privileged ring, called the **kernel** or **supervisor**, and an unprivileged **user** ring.

The x86 architecture, for example, supports up to four rings, but only two of those (ring 0, the most privileged, and ring 3, the least privileged) are used by Windows and Linux.

The following figure represents the ring organization of most operating systems running on the x86 architecture:

Supervisor mode and user mode

In a two-level protection ring hierarchy, the protection level of the currently executing thread is typically represented by a bit in a register. When operating in ring 0, the supervisor mode

bit is 1, and when operating in user mode (ring 3 on x86) the supervisor mode bit is 0. **The supervisor mode bit can only be modified by code running in supervisor mode.**

The state of the supervisor mode bit determines which instructions are available for execution by the thread. Instructions that could interfere with system operation, such as the x86 `HLT` instruction, which halts processor instruction execution, are unavailable in user mode. Any attempt to execute a prohibited instruction results in a general protection fault. In user mode, access by user applications to system memory regions and the memory allocated to other users is prohibited. In supervisor mode, all instructions are available for execution and all memory regions are accessible.

System calls

All code belonging to the kernel and device drivers runs in ring 0, always. All user code runs in ring 3, always, even for users with enhanced operating system privileges such as system administrators. Code running in ring 3 is strictly controlled by the system and cannot directly do anything that involves allocating memory, opening a file, displaying information to the screen, or interacting with an I/O device. To access any of those system features, ring 3 user code must make a service request to the kernel.

The kernel service request must first pass through a gate where the type of operation being requested, as well as any associated parameters, are scrutinized and validated before the execution of the operation is allowed to proceed. The code performing the requested operation runs in supervisor mode at ring 0 and, when complete, returns control to the user mode calling routine in ring 3.

Price of system call

In early versions of Windows (prior to Windows XP), an application used the sofware interrupt mechanism with vector `2eh` to request system services. The `int 2eh` mechanism for requesting kernel services is that it is not very efficient. In fact, it takes over 1,000 processor clock cycles to get from the point at which the `int 2eh` instruction executes to the kernel code that actually begins to handle the exception. A busy system may request kernel services thousands of times per second.

To address this inefficiency, Intel implemented the `sysenter` and `sysexit` instructions in the x86 architecture beginning with the Pentium II processor in 1997. The purpose of these instructions is to accelerate the process of calling from ring 3 to ring 0, and later returning to ring 3. By using these instructions instead of `int 2eh`, entry into and exit from kernel mode speeds up by about a factor of three. [2]

Around the time Intel began producing processors with the `sysenter` and `sysexit`

instructions, AMD released the `syscall` and `sysret` instructions in their processor architectures, with the same performance objective. Unfortunately, the instructions in the Intel and AMD processor architectures are not compatible, which leads to a requirement for operating systems to differentiate between architectures when using accelerated kernel calling instructions.

Doing floating-point calculations

Recall from previous lecture that the very general model of CPU is not able to perform calculations on numbers different than integers. CPU normally performs floating-point arithmetic through slow software routines. To improve floating-point performance on 8086/8088 based computers, Intel introduced in 1980 the 8087 foating-point coprocessor with its own instruction set. The 8086 is called as the host and 8087 as coprocessor as 8087 cannot operate all by itself. Like other extensions to the basic instruction set, 8087 instructions are not strictly needed to construct working programs, but allowing common numerical tasks to be performed much faster than corresponding machine code routines can. Was it really such an innovation and improvement? Let's give the floor to one of the computer users from those times.

The overall speed-up from using an 8087 instead of software floating routines is likely a good deal better than 100x. Back in the day when I was still in college, the class had an assignment to write a FORTRAN program that did a Fourier analysis. Over the holidays, I wrote the program on a COMPAQ Portable with the standard 4.77 MHz 8088 and no math coprocessor. When I ran the program to test it, I thought I had an infinite loop somewhere after the program did not complete after five or so minutes. After hours of debugging and finding no errors, someone suggested to me to let the program run and take a long break. Much to my surprise, the program completed, successfully and with the correct results, after about half an hour.

When I went to submit the program for grading, the teaching assistants in the lab would run the program with another dataset they had. The computers in the lab were standard IBM PCs with 4.77 MHz 8088 CPUs but they also had the 8087 installed. I was fully prepared to wait 30 minutes but, being that the machines had 8087s, I had guessed the program would complete execution in five or ten minutes.

To my utter shock, the program completed in one or two seconds. The MS-DOS cursor reappeared almost immediately after the TA had pressed the Enter key to run my program.

Of course, this is just one anecdote and I was very inexperienced at writing programs at the time so it is entirely possible that the program I wrote was unusually bad. But that is one of the war stories I've accumulated over the years from developing software where a very surprising result made an indelible impression on me. [3]

Below I present execution times for selected 8087 numerical instructions and corresponding 8086 emulations [4]:

```
Comparison of 8087 and 8086 Clock Times

                            | Approximate execution time (in us)
                            +-------------+---------------------
Instruction                 | 8087        | 8086 Emulation
                            | 8 MHz clock |
----------------------------+-------------+---------------------
Add/Subtract                | 10          | 1000
Multiply (single precision) | 11.9        | 1000
Multiply (double precision) | 16.0        | 1312
Divide (single precision)   | 24.4        | 2000
Compare                     |  5.6        | 812
Load (double precision)     |  6.3        | 1062
Store (double precision)    | 13.1        | 750
Square root                 | 22.5        | 12250
Tangent                     | 56.3        | 8125
Exponentiation              | 62.5        | 10687
```

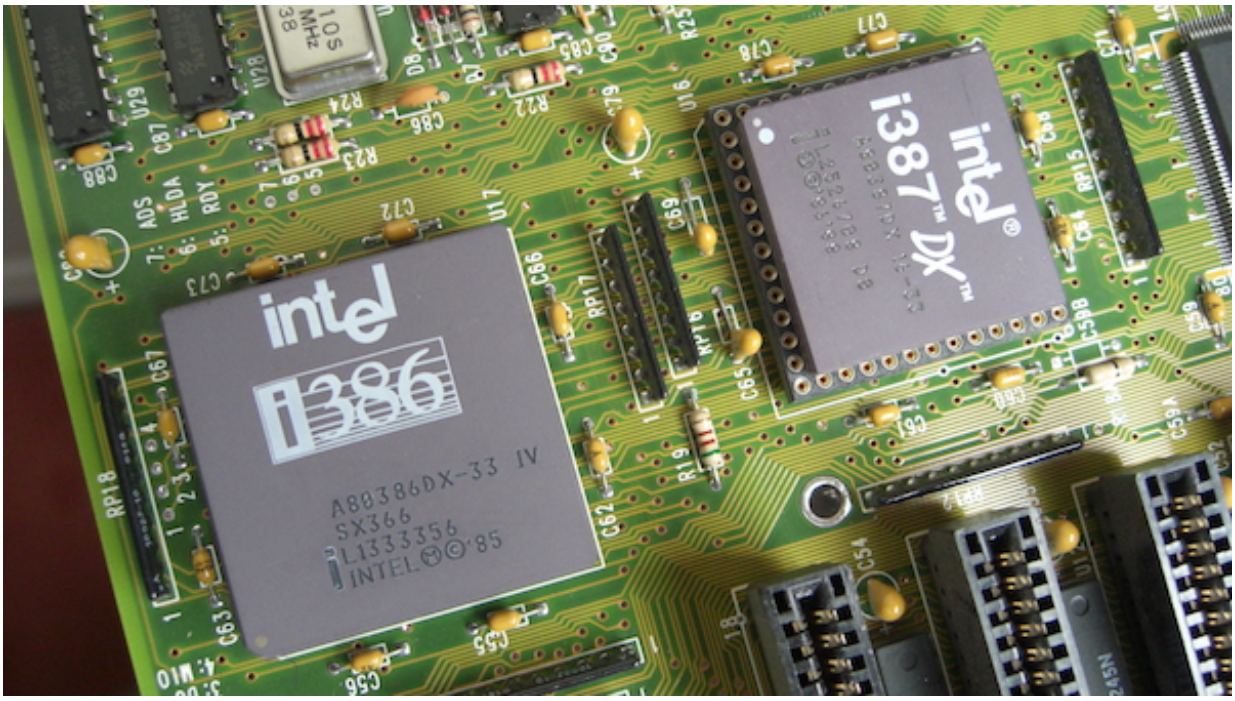

fig. Intel 80386 CPU with Intel 387 Math Coprocessor [6]

Because presence of the 8087 was optional, application programs had to be written to make use of the special floating-point instructions. At run time, sofware applications that wished to take advantage of its capabilities first tested for the availability of the 8087. When detected

direct floating-point operations could be performed safely. Otherwise adequate but slower floating-point code calculating desired functions had to be executed in software.

So in compiled program two "branches" of code were present, each executed respectively in case of presence or absence of floating-point coprocesor.

Intercepting floating point instructions by 8087 coprocessor

CPU and coprocessor coexistence wouldn't be possible without support from hardware site. The 8087 was initially conceived by Bill Pohlman, the engineering manager at Intel who oversaw the development of the 8086 chip. He took steps to be sure that the 8086 chip could support a yet-to-be-developed math chip.

Technically speaking both chips share the same address and data buses and were connected to each other with just a few control lines.

As you can see the 8086 and 8087 do not directly transfer data between themselves, and can only share data by storing it in memory.

The 8087 adds 68 new instructions to the standard 8086/8088 instruction set for performing arithmetic, trigonometric, exponential, and logarithmic functions. Most 8087 assembly mnemonics begin with `F`, such as `FADD`, `FMUL`, `FCOM` and so on, making them easily distinguishable from 8086 instructions.

Using the 8087 coprocessor, the code consists of a single stream of 8086 and 8087 instructions retrieved in the usual sequential manner by the 8086 host processor because only the processor has the knowledge where the next instruction is located.

The 8087 passively monitors the address and data buses as the host processor executes instructions and only becomes active when an 8087 opcode appears.

As the 8086 fetches instruction bytes from the memory and puts them in its queue, the 8087 also reads these instruction bytes and puts them in its own queue. The 8087 decodes each instruction that comes into its queue just like CPU does it.

Each processor decodes all instructions in the fetched instruction byte stream but executes only its own instructions. It is possible the binary encodings for all 8087 instructions begin with the bit pattern `11011`, decimal `27`, the same as the ASCII character `ESCAPE` or `ESC`, although in the higher-order bits of a byte. The coprocessor operation codes are encoded in 6 bits across 2 bytes, beginning with the escape sequence:

```
1101 1xxx | mmxx xrrr
```

The first three `x` bits are the first three bits of the floating-point opcode. The latter half three bits of the floating-point opcode is located between two `m` bits and three `r` bits. The `m` and `r` bits specify the addressing-mode information.

The control role of 8086 in 8087 working cycle

When 8087 decodes an instruction from its queue and finds that it is an 8086 instruction, the 8087 simply treats the instruction as NOP (no-operation). Likewise, when the 8086 decodes an instruction from its queue and finds that it is an 8087 instruction, the 8086 simply treats the instruction as NOP or in some cases reads a data word from the memory for the 8087.

The start of a numeric operation is accomplished when the CPU executes the ESC instruction. Exactly at the same time coprocessor realized that floating-point instruction must be executed. The instruction may or may not identify a memory operand. 8086 is not executing floating-point instruction but it helps to do so the 8087 coprocesor.

No memory reference

The CPU does distinguish between ESC instructions that reference mamory and those that do not. If the 8087 instruction has only an opcode and does not contain a memory reference (e.g. an 8087 stack operation), then 8087 will start execution and 8086 simply proceeds to the next instruction.

Simple memory reference

If the instruction refers to a memory operand, the CPU calculates the operand's address using any one of its available addressing modes, and then executes a bus cycle to performs a "dummy read" of the word at that location. This is a normal read cycle but named as a "dummy read" because CPU do not store the read operand into any CPU register or perform any operation on it. The CPU simply ignores the data it receives. The 8086 have to fetch the first word of the operand as 8087 cannot calculate the physical address. The 8087's control unit uses a "dummy read" cycle initiated by the CPU to capture and save the address that the CPU places on the bus as well as the data word when it becomes available on the local data bus.

Large operand memory reference

If data required is longer than one word then, after completion of the data read cycle driven by the CPU, the 8087 would immediately use DMA to take control of the bus and transfer the additional bytes of the operand itself. Now it is possible for 8087 to operate "independently" because the addres of the first word was computed by CPU, so the remaining words can be

fetched directly by 8087 by simply incrementing the address of the first word.

Writing results

If an 8087 instruction with a memory operand is intended to store results, the 8087 ignores the read word on the data bus and just copy the address, then request DMA and write the result, in the same way that it does reading the end of an extended operand.

In this way, the main CPU maintained general control of the bus and bus timing, while the 8087 handled all other aspects of execution of coprocessor instructions, except for brief DMA periods when the 8087 would take over the bus to read or write operands to/from its own internal registers from/to memory.

As a consequence of this design, the 8087 could only operate on operands taken either from memory or from its own registers, and any exchange of data between the 8087 and the 8086 or 8088 was only through RAM.

Executing instruction

The main CPU program continue to execute while the 8087 execute an instruction; from the perspective of the main CPU, a coprocessor instruction took only as long as the processing of the opcode and any memory operand cycle, after which the CPU begin executing the next instruction of the program. Thus, a system with an 8087 was capable of (limited) true parallel processing, performing one operation in the integer ALU of the main CPU while at the same time performing a floating-point operation in the 8087 coprocessor.

`WAIT` instruction, `BUSY` and `TEST` pin

Since the 8086 or 8088 exclusively controll the instruction flow and timing and has no direct access to the internal status of the 8087, and because the 8087 can execute only one instruction at a time, programs for the combined 8086/8087 system have to ensure that the 8087 have time to complete the last instruction issued to it before it was issued another one. This explains why parallel processing by these chips tandem is limited; there is no option to enqueue floating-point instructions.

When 8087 gets its operand, it begins processing by making the `BUSY` output high. This `BUSY` output is connected to the `TEST` input of the CPU. At this moment 8087 execute its instruction and 8086 moves ahead with its next instruction. Hence multiprocessing takes place. The `WAIT` instruction of the main CPU is provided for checking if 8087 chip is ready to serve another floating-point instruction. Most assemblers implicitly asserted a `WAIT` instruction before each instance of most floating-point coprocessor instructions. `WAIT` makes the CPU check the `TEST` pin. If the `TEST` pin is high (8087 is `BUSY`), then the CPU enters `WAIT` state. It comes out of it only when `TEST` is low (8087 has finished its execution). The same synchronization is used before the CPU accesses data that was

written by the coprocessor. A `WAIT` instruction after any coprocessor instruction that writes to memory causes the CPU to stop until the coprocessor has completed transfer of the data to memory, after which the CPU can safely access it. Waiting on `TEST` pin guarantee that 8087's operations finishes, it's safe for 8086 to get the (correct) result and new, if there is any waiting, floating-point instruction can be executed.

During execution, if 8087 needs to read or write more data from the memory, then it does so by stealing bus cycles from the CPU in the following manner. The `!RQ / !GT0` of 8087 is connected to `!RQ / !GT0` of the CPU. 8087 gives an active low request pulse. 8086 completes the current bus cycle and gives the grant pulse and enters the hold state. 8087 uses the shared system bus to perform the data transfer with the memory. 8087 gives the release pulse and returns the system bus back to the CPU.

The 80287 coprocessor

The 80287 coprocessor-CPU interface is totally different from the 8087 design. Since the 80286 implements memory protection via an MMU based on segmentation, it would be much too expensive to duplicate the whole memory protection logic on the coprocessor, which an interface solution similar to the 8087 would have required. Instead, in an 80286/80287 system, the CPU fetches and stores all opcodes and operands for the coprocessor. Information is then passed through the CPU ports F8h-ffh. (As these ports are accessible under program control, care must be taken in user programs not to accidentally perform write operations to them, as this could corrupt data in the math coprocessor.)

The 8086/8088 combination can be characterized as a cooperation of partners with equal rights, while the 80286/287 is more a master-slave relationship. This makes synchronization easier, since the complete instruction and data flow of the coprocessor goes through the CPU. Before executing most coprocessor instructions, the 80286 tests its /BUSY pin, which is tied to the 287 coprocessor and signals if the 80287 is still executing a previous coprocessor instruction or has encountered an exception. The 80286 then waits until the /BUSY signal goes to "low" before loading the next coprocessor instruction into the 80287. Therefore, a WAIT instruction before every coprocessor instruction is not required. These WAITs are permissible, but not necessary, in 80287 programs. The second form of WAIT synchronization (after the coprocessor has written a memory operand) is still necessary on 286/287 systems. [5]

Note: IEEE 754 standard compatibility

The 8087 was not entirely compliant with the initial IEEE 754 standard, which was published several years afer the introduction of the 8087. Later Intel floating-point coprocessors,

beginning with the 80387 in 1987, were fully standard-compliant.

The 80387 (387 or i387) is the first Intel coprocessor to be fully compliant with the IEEE 754-1985 standard. Released in 1987, two years after the 386 chip, the i387 includes much improved speed over Intel's previous 8087/80287 coprocessors and improved characteristics of its trigonometric functions.

Controlling control unit with micoro-operations

In previous lecture you have seen that the operation of a computer, in executing a program, consists of a sequence of instruction cycles, with one machine instruction per cycle.

Micro-operations are the functional, or atomic, operations of a processor. In this section, you will examine micro-operations to gain an understanding of how the events of any instruction cycle can be described as a sequence of such micro-operations.

You will decompose the behavior or functioning of the processor into elementary operations, called micro-operations. By reducing the operation of the processor to its most fundamental level, you are able to define exactly what it is that the control unit must cause to happen.

In the following analysis you will consider four registers:

- `MAR`, memory address register, is connected to the address lines of the system bus. It specifies the address in memory for a read or write operation.
- `MBR`, memory buffer register, is connected to the data lines of the system bus. It contains the value to be stored in memory or the last value read from memory.
- `PC`, program counter, holds the address of the next instruction to be fetched.
- `IR`, instruction register, holds the last instruction fetched.

Whenever you see `IR(address)` it means the address portion of the instruction register.

The fetch cycle

The fetch cycle, which occurs at the beginning of each instruction cycle and causes an instruction to be fetched from memory.

```
1 t1: MAR <- (PC)
2 t2: MBR <- Memory
3     PC <- (PC) + I
4 t3: IR <- (MBR)
```

1. Move the address of the next instruction to be executed to the memory address register (MAR).
2. Bring in the instruction. The desired address (in the MAR) is placed on the address bus, the control unit issues a READ command on the control bus, and the result appears on the data bus and is copied into the memory buffer register (MBR).
3. Increment the PC by the instruction length to get ready for the next instruction.
4. Move the contents of the MBR to the instruction register (IR). This frees up the MBR for use during a possible indirect cycle.

Fetch source operands (indirect addressing case)

Once an instruction is fetched, the next step is to fetch source operands.

```
1 t1: MAR <- (IR(Address))
2 t2: MBR <- Memory
3 t3: IR(Address) <- (MBR(Address))
```

1. The address field of the instruction is transferred to the MAR.
2. Fetch the address of the operand.
3. The address field of the IR is updated from the MBR, so that it now contains a direct rather than an indirect address.

The interrupt cycle

At the completion of the execute cycle, a test is made to determine whether any enabled interrupts have occurred. If so, the interrupt cycle occurs.

```
1 t1: MBR <- (PC)
2 t2: MAR <- Save_Address
3     PC <- Routine_Address
4 t3: Memory <- (MBR)
```

1. The contents of the PC are transferred to the MBR, so that they can be saved for return from the interrupt.
2. The MAR is loaded with the address at which the contents of the PC are to be saved.
3. PC is loaded with the address of the start of the interrupt-processing routine.
4. Store the MBR, which contains the old value of the PC, into memory.

The execute cycle

The fetch, indirect, and interrupt cycles are simple and predictable. Each involves a small, fixed sequence of micro-operations and, in each case, the same micro-operations are repeated each time around.

This is not true of the execute cycle. Because of the variety of opcodes, there are a number of different sequences of micro-operations that can occur. The control unit examines the opcode and generates a sequence of micro-operations based on the value of the opcode.

Consider several hypothetical examples.

Example 1

Consider an add instruction:

```
ADD R1, X
```

which adds the contents of the location `X` to register `R1`. The following sequence of micro-operations might occur:

```
1 t1: MAR <- (IR(address))
2 t2: MBR <- Memory
3 t3: R1 <- (R1) + (MBR)
```

1. The address portion of the `ADD` instruction is loaded from `IR` into the `MAR`.
2. The referenced memory location is read.
3. The contents of `R1` and `MBR` are added by the ALU.

Example 2

A common instruction is increment and skip if zero:

```
ISZ X
```

The content of location `X` is incremented by 1. If the result is 0, the next instruction is skipped. A possible sequence of micro-operations:

```
1 t1: MAR <- (IR(address))
2 t2: MBR <- Memory
3 t3: MBR <- (MBR) + 1
4 t4: Memory <- (MBR)
5     If ((MBR) = 0) then:
6       PC <- (PC) + I
```

The new feature introduced here is the conditional action. The PC is incremented if `(MBR) = 0`. This test and action can be implemented as one micro-operation. Note also that this micro-operation can be performed during the same time unit during which the

updated value in MBR is stored back to memory.

Example 3

Consider a subroutine call instruction. As an example, consider a branch-and-save-address instruction:

```
BSA X
```

The address of the instruction that follows the `BSA` instruction is saved in location `X`, and execution continues at location `X + I`. The saved address will later be used for return. This is a straightforward technique for supporting subroutine calls. The following micro-operations suffice:

```
1 t1: MAR <- (IR(address))
2     MBR <- (PC)
3 t2: PC <- (IR(address))
4     Memory <- (MBR)
5 t3: PC <- (PC) + I
```

The address in the `PC` at the start of the instruction is the address of the next instruction in sequence. This is saved at the address designated in the IR. The latter address is also incremented to provide the address of the instruction for the next instruction cycle.

A control signals example

To illustrate the functioning of the control unit, let us examine a simple example.

Consider simplified processor with a single accumulator (AC):

The data paths between elements are indicated. For clarity the control paths for signals emanating from the control unit are not shown, but the terminations of control signals are labeled `Ci` and indicated by a circle. For simplicity, the data and control paths for incrementing the PC and for loading the fixed addresses into the PC and MAR are not shown. The control unit receives inputs from the clock, the IR, and flags. With each clock cycle, the control unit reads all of its inputs and emits a set of control signals.

Control signals go to three separate destinations:

- Data paths: The control unit controls the internal flow of data. For example, on instruction fetch, the contents of the memory buffer register are transferred to the IR.

For each path to be controlled, there is a switch (indicated by a circle in the figure). A control signal from the control unit temporarily opens the gate to let data pass.
- ALU: The control unit controls the operation of the ALU by a set of control signals. These signals activate various logic circuits and gates within the ALU.
- System bus: The control unit sends control signals out onto the control lines of the system bus (e.g., memory READ).

The control unit must maintain knowledge of where it is in the instruction cycle. Using this knowledge, and by reading all of its inputs, the control unit emits a sequence of control signals that causes micro-operations to occur. It uses the clock pulses to time the sequence of events, allowing time between events for signal levels to stabilize.

Examples of the control signals that are needed for some of the micro-operation described earlier a given below:

```
Control signals for micro-operation
```

It is worth pondering the minimal nature of the control unit. The control unit is the engine that runs the entire computer. It does this based only on knowing the instructions to be executed and the nature of the results of arithmetic and logical operations (e.g., positive, overflow, etc.). It never gets to see the data being processed or the actual results produced. And it controls everything with a few control signals to points within the processor and a few control signals to the system bus.

Towards microcode and microinstructions

The control unit seems a reasonably simple device. Nevertheless, to implement a control unit as an interconnection of basic logic elements is no easy task. The design must include logic for sequencing through micro-operations, for executing micro-operations, for interpreting opcodes, and for making decisions based on ALU flags. It is difficult to design and test such a piece of hardware. Furthermore, the design is relatively inflexible. For example, it is difficult to change the design if one wishes to add a new machine instruction.

Notice that all previously mentioned actions related to executing instruction can be completed with a sequenc of a very simple, atomic steps of from the set of possible **microoperations**:

```
AI01   IR <- (MBR)
AI02   IR(Address) <- (MBR(Address))
AI03   MAR <- (IR(address))
AI04   MAR <- (PC)
AI05   MAR <- Save_Address
AI06   MBR <- (PC)
AI07   MBR <- Memory
AI08   MBR <- (MBR) + 1
AI09   Memory <- (MBR)
AI10   PC <- (IR(address))
AI11   PC <- (PC) + I
AI12   PC <- Routine_Address
AI13   R1 <- (R1) + (MBR)
```

Thus the fetch sequence:

```
1 t1: MAR <- (PC)
2 t2: MBR <- Memory
3     PC <- (PC) + I
4 t3: IR <- (MBR)
```

can be encoded as:

```
AI04
AI07
AI11
AI01
```

This way you obtain something you can call a *control unit programming language* in a form of sequence of **microinstructions** from above defined atomic actions.

Thus an alternative to fixed hardware wiring is to implement a microprogrammed control unit. A sequence of microinstruction controlling the control unit is known as a microprogram, or firmware. This latter term reflects the fact that a microprogram is midway between hardware and software. It is easier to design in firmware than hardware, but it is more difficult to write a firmware program than a software program.

Bibliography

1. William Stallings, *Computer Organization and Architecture*, Pearson, 11th edition (2019)
2. Jim Ledin, *Modern Computer Architecture and Organization*, Packt, April 2020
3. Anonymous comment in *Ken Shirriff's blog: Die analysis of the 8087 math coprocessor's*

fast bit shifter, May 31, 2020 at 8:32 AM, http://www.righto.com/2020/05/die-analysis-of-8087-math-coprocessors.html Retrieved 2023-10-21.

4. *Datasheet for the Intel 8087 Math Coprocessor*, https://pdf.datasheetcatalog.com/datasheets/2300/45014_DS.pdf Retrieved 2023-10-21.
5. ARCHIVED: How do various PC math coprocessors work?, https://kb.iu.edu/d/aann Retrieved 2023-10-21.
6. https://commons.wikimedia.org/wiki/file:80386with387.JPG Retrieved 2023-10-21.