Materials

W3 Procesor - podstawowe bloki funkcjonalne i działanie.

- Processor structure and function

Omawiane zagadnienia

- A simplified view on processor structure and function
- Instruction execution cycle
- Processor instruction set (RISC and CISC)
- Interrupt processing

Processor structure and function

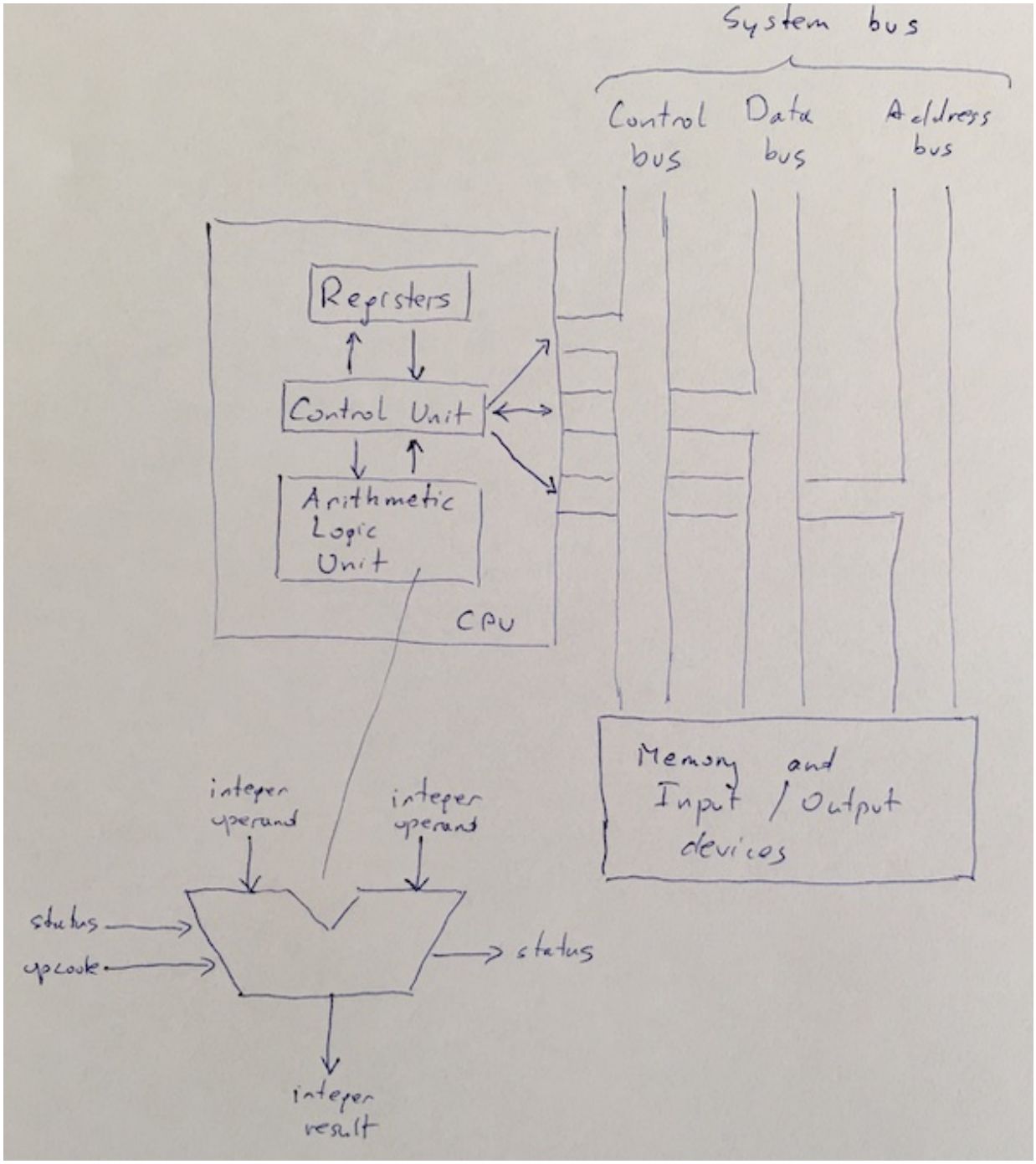
Simplified view of a processor

To understand the organization of the processor, why it has these functional blocks and not others, you should think about what you expect from it. Probaly you say that it must:

- read an instruction from memory (so called: **fetch instruction**);
- decode the instruction to determine what action it must execute (**interpret instruction**);
- if instruction require data, read them (**fetch data**);
- execute instruction on data and do what this instruction is intended to do performing some arithmetic or logical operation (**process data**);
- if instruction require, write data which are results of its execution (**write data**).

If you think a while on all of those tasks you come to a conclusion that among many other things the processor needs to store some data (understand very generally as instructions and real, pure data for them) temporarily. At least it must remember the location of the last instruction so that it can know where to get the next instruction. It needs also to store instructions and data temporarily to be able to execute them. In other words, the processor needs a small internal memory.

On the figure below I present a simplified view of a processor, indicating its connection to the rest of the system via the **system bus**:



The major components of the processor are:

- An **arithmetic and logic unit** (ALU) which does the actual computation or processing of data executing arithmetic and bit manipulation operation.
- A **control unit** (CU) which controls the movement of data and instructions into and out of the processor and controls the operation of the ALU. The control unit directs overall processor operations to carry out each instruction. All the components, the registers, ALU, memory, and input/output devices respond to commands initiated by the control unit.
- A **registers** which is a set of internal processor memory, storage locations where processor keeps data (understood as both instructions and pure data) it require to perform the tasks you pointed at the beginning of this part. ALU in fact operates only on

data in the internal processor memory.

Arithmetic logic unit

Arithmetic logic unit (ALU) performs, as it names stands, arithmetic and bit-oriented operations under the direction of the control unit. To perform an operation, the ALU requires data input values, called **operands**, and a code indicating the operation to be performed. The ALU output is the result of the operation. ALU operations may use one or more processor flags, such as the carry, negative, zero or overflow flag, as input and set the states of processor flags as outputs.

Please note that *arithmetic* in most cases means an *integer arithmetic*. ALU is not intended to perform floating-point operations. Early microprocessors were designed to operate on integers and CPU normally performs floating-point arithmetic through (relatively slow) software routines.

In 1980 Intel introduced the 8087 chip to improve floating-point performance on 8086/8088 computers such as the original IBM PC. The 8087 co-processor greatly improved floating point speed, up to 100 times faster. The 8087's architecture became part of later Intel processors (as the Floatin Point Unit, the FPU), and the 8087's instructions are still a part of today's x86 desktop computers.

If you ar curious why FPU wasn't a part of CPU from early begining of CPU history, please keep in mind that the die of the 8087 is fairly complex, with 40,000 (or 45,000) transistors [3, 4]. At the same time 8086 CPU, to which the coprocessor 8087 was an addition, contained 29,000 transistors [4]. As you can see the complexity of logic structer holding floating point number is very hight and doubles the cost of the system (you have to pay for CPU and even much more for external FPU).

You will get more informationa about FPU in next lectures.

Control unit

The control unit is a synchronous sequential digital circuit. Its function is to interpret processor instructions and manage the execution of those instructions by interacting with the other functional units within the processor and with external components, including memory and input/output devices.

Registers

For many people the ALU is the most important processor part as it does the real job (at least that's what they think) performing arithmetic and logic operations on data which are the

essence of processor existence. It's true, it's really essential but on the other hand everything it does depends and is reflected on other component: registers.

Different machines have different register organizations and use different terminology. For purposes of the following discussion I will divide registers into two categories:

- user-visible registers,
- control and status registers.

This distinction in real architectures, however, is not very strict and may vary from processor to processor so please keep your mind open.

User-visible registers

A user-visible register is one that you may reference by means of the instructions that the processor executes. You can divide them in the following categories:

- general purpose,
- data,
- address,
- condition codes.

General purpose

These registers can be assigned to a variety of functions by the programmer. Generality means that you can store there the operand for various opcodes. However very often there are some restrictions. For example, there may be dedicated registers (even among general) for floating point or stack operations.

Data registers

These registers can be used only to hold data and cannot be used in the calculation of an operand address.

Address registers

Registers you use in a particular addressing mode. Examples include the following:

- Segment pointer holds the address of the base of the memory segment. This way you may address providing displacement relative to the base.
- Index registers simplfies iterating through memory.
- Stack pointer points to the top of the stack. This allows implicit addressing; that is, `PUSH`, `POP`, and other stack instructions need not contain an explicit stack operand. The stack (also known as a call stack, execution stack, program stack, control stack, run-time stack, or machine stack) is a stack data structure that stores information about

the active subroutines of a computer program.

The main reason for stack existing is to keep track of the point to which each active subroutine should return control when it finishes executing. An active subroutine is one that has been called, but is yet to complete execution, after which control should be handed back to the point of call.

Note: how stack helps to track the subroutine return point

See: "Wstęp do informatyki"

Note: stack registers in action

In x86, the main stack register is called **stack pointer** (`SP`) and it points to current stack top. The **stack segment** register (`SS`) is usually used to store information about the memory segment that stores the call stack of currently executed program. To *push* a value to the stack, the `PUSH` instruction is used. To *pop* a value from the stack, the `POP` instruction is used.

If, for example, `SS = 0x1000` and `SP = 0xB204` then current stack top is the physical address `0x1B204` . Hold on for a while to explain this.

In x86 architecture the system uses 16-bit segment registers to derive the actual memory address (in both real and protected mode). It means that to get a linear address (in real mode) the 16-bit segment address is shifted 4 bits left and next the 16-bit offset is added to it resulting in 20-bit address.

```
SS = 0x1000
SP = 0xB204

0x01000       0001 0000 0000 0000  - (stack) segment
0x10000  0001 0000 0000 0000 0000  - segment shifted 4 bits left
0x0B204       1011 0010 0000 0100  - stack pointer

0x1B204  0001 1011 0010 0000 0100  - linear address
```

The machine instruction:

```
PUSH AX
```

push the value stored in `AX` (16-bit register) to the stack. This is done by subtracting a value of 2 (2 bytes = 16 bits) from `SP` (by default, the stack grows downward in

memory, so newer values are placed at lower memory addresses). The new value of `SP` becomes `0x0B202`. The CPU then copies the value of `AX` to the memory word whose physical address is `0x0B202`.

When similar instruction engaging `BX` register is executed, `SP` is set to `0x0B200` and `BX` is copied to `0x0B200`.

To restore the values stored at the stack, the program shall contain machine instructions like this (in reverse order):

```
POP BX
POP AX
```

First instruction, `POP BX` copies the word at `0x0B200` (which is the old value of `BX`) to `BX`, then increases `SP` by 2 so now it is `0x0B202`.

Similarly `POP AX` copies the word at `0x0B202` (which is the old value of `AX`) to `AX`, then sets `SP` to `0xB204`.

Condition codes

Condition codes (flags) are bits set by the processor hardware as the result of operations. Condition code bits are collected into one or more registers. Usually, they form part of a **control register** (described below). Generally, machine instructions allow these bits to be read by implicit reference, but the programmer cannot alter them.

For example, an arithmetic operation may produce a positive, negative, zero, or overflow result. In addition to the result itself being stored in a register or memory, a condition code is also set. The code may subsequently be tested as part of a conditional branch operation.

There are several design issues to be addressed here.

- Completely general-purpose registers vs. specialize
- Optimal number of registers
- Length of register
    - should be able to hold values of most data types,
    - must (or rather should) be at least long enough to hold the largest address

Control and status registers

There are a variety of processor registers that are employed to control the operation of the

processor. Most of these, on most machines, are not visible to the user. Some of them may be visible to machine instructions executed in a control or operating system mode.

Four registers are essential to instruction execution:

- Program counter (PC): Contains the address of an instruction to be fetched. Typically, the processor updates the PC after each instruction fetch so that the PC always points to the next instruction to be executed. A branch or skip instruction will also modify the contents of the PC.
- Instruction register (IR): Contains the fetched instruction allows the opcode and operand specifiers to be analyzed.
- Memory address register (MAR): Contains the address of a location in memory.
- Memory buffer register (MBR): Contains a word of data to be written to memory or the word most recently read. Data are exchanged with memory using the MAR and MBR. Not all processors have internal registers designated as MAR and MBR, but some equivalent buffering mechanism is needed whereby the bits to be transferred to the system bus are staged and the bits to be read from the data bus are temporarily stored.

Many processor designs include a register or set of registers, often known as the **program status word** (PSW), that contain status information. The PSW typically contains **condition codes** plus other **status information**. Common fields or flags include the following:

- Sign which contains the sign bit of the result of the last arithmetic operation.
- Zero which is set when the result of the last arithmetic operation is 0.
- Carry which is set if an operation resulted in a carry (addition) into or borrow (subtraction) out of a high-order bit.
- Equal which is set if a logical compare result is equality.
- Overflow which is used to indicate arithmetic overflow.
- Interrupt enable/disable used to enable or disable interrupts.
- Supervisor indicating whether the processor is executing in supervisor or user mode.

There may be a pointer to an interrupt vector register.

If a stack is used to implement certain functions (e.g., subroutine call), then a system stack pointer is needed.

A page table pointer is used with a virtual memory system.

Finally, registers may be used in the control of I/O operations.

Example of microprocessor register organizations

It is instructive to examine and compare the register organization of comparable systems. In this section, you will see two 16-bit CISC (see next subsection) microprocessors that were

designed at about the same time: the Motorola MC68000 and the Intel 8086.

```
8086

General registers

   111111
5432109876543210
|______AX______|  Accumulator
|__AH__||__AL__|

|______BX______|  Base
|______CX______|  Count
|______DX______|  Data

Pointers and index

|______SP______|  Stack Pointer
|______BP______|  Base Pointer
|______SI______|  Source Index
|______DI______|  Destination Index

Segment

|______CS______|  Code
|______DS______|  Data
|______SS______|  Stack
|______ES______|  Extract

Program status

|____FLAGS_____|
|______PC______|
```

```
80386, Pentium

General registers

33222222222111111111
10987654321098765432109876543210
|_____________EAX_____________|
                |______AX______|
                |__AH__||__AL__|

|_____________EBX_____________|
|_____________ECX_____________|
|_____________EDX_____________|

Pointers and index

|_____________ESP_____________|
|_____________EBP_____________|
|_____________ESI_____________|
|_____________EDI_____________|

Program status

|_____________EAX_____________|
|_____________EAX_____________|
```

```
Intel, AMD 64-bit

6666555555555444444444433333333332222222221111111111
3210987654321098765432109876543210987654321098765432109876543210
|       ||      ||      ||      ||      ||      ||      ||      |
|__64__||__56__||__48__||__40__||__32__||__24__||__16__||__8___|
|_______________________________RAX_______________________________|
|xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx|_____________EAX_____________|
|xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx|______AX______|
|xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx|__AH__||__AL__|
RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP, R8-R15
General purpose registers: EAX, EBX, ECX, EDX, ESI, EDI
EBP -- Base Pointer
ESP -- Stack pointer
```

```
Motorola MC68000

Data registers (general purpose registers)

33222222222111111111
10987654321098765432109876543210
|_____________D0_____________|
|_____________D1_____________|
|_____________D2_____________|
|_____________D3_____________|
|_____________D4_____________|
|_____________D5_____________|
|_____________D6_____________|
|_____________D7_____________|

Address registers

33222222222111111111
10987654321098765432109876543210
|_____________A0_____________|
|_____________A1_____________|
|_____________A2_____________|
|_____________A3_____________|
|_____________A4_____________|
|_____________A5_____________|
|_____________A6_____________|

|_____________A7_____________| Stack register in user mode
|_____________A7_____________| Stack register in supervisor mode

|_____________PC_____________|
                |______SR_____| Status Register
```

The x86 has six 32-bit registers suitable for temporary data storage: `EAX`, `EBX`, `ECX`, `EDX`, `ESI`, and `EDI`. In many processor architectures, **specific registers are assigned** to support functions performed by certain instructions. For example, in the x86 architecture, a single `REP MOVSD` instruction moves a block of data with a length (in words) provided in `ECX` beginning at a source address in `ESI` to a destination address in `EDI`.

The larger number of registers in RISC architectures reduces the need to access system memory because more registers are available for storage of intermediate results. This helps with performance because accessing system memory is significantly more time consuming than accessing data located in processor registers.

The Intel 80386 is a 32-bit microprocessor designed as an extension of the 8086. The 80386

uses 32-bit registers. However, to provide upward compatibility for programs written on the earlier machine, the 80386 retains the original register organization embedded in the new organization. Given this design constraint, **the architects of the 32-bit processors had limited flexibility in designing the register organization**.

Conclusion: There is no universally accepted philosophy concerning the best way to organize processor registers.

Register contents during subroutne call

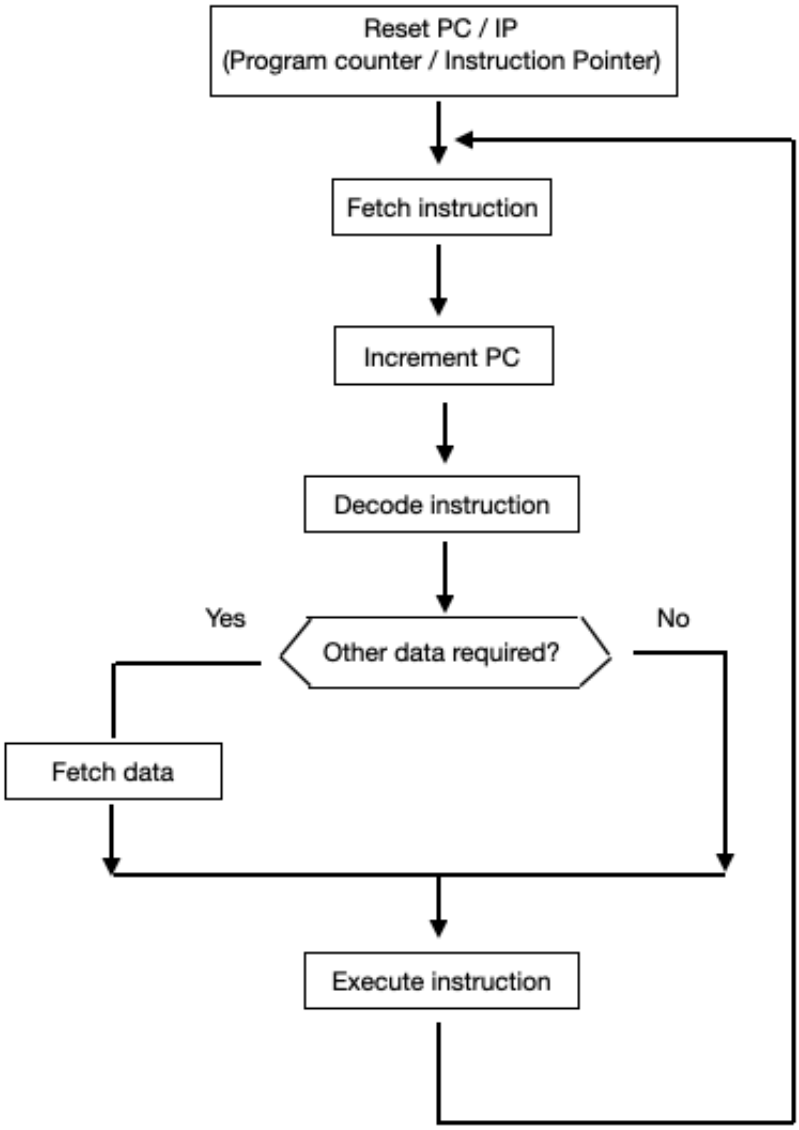
In some machines, a subroutine call will result in the automatic saving of all user-visible registers, to be restored on return. The processor performs the saving and restoring as part of the execution of call and return instructions. This allows each subroutine to use the user-visible registers independently. On other machines, it is the responsibility of the programmer to save the contents of the relevant user-visible registers prior to a subroutine call, by including instructions for this purpose in the program.

Instructions

Instruction execution cycle

When a computer system is powered on, the processor undergoes a reset process to initialize its internal components to defined values. During the reset process, it loads the **Program Counter** (**PC**) register with the memory location of the first instruction to be executed. Sofware developers who construct the lowest-level system sofware components must configure their development tools to produce a code memory image that begins execution at the address required by the processor architecture.

Once the reset has completed, the PC register contains the initial instruction location. The control unit fetches the first instruction from memory and **decodes** it. Decoding is the process the control unit performs to determine the actions required by the instruction. This way processor begins executing instructions, performing the repetitive cycle shown in figure below:



Fetch stage

The next instruction is fetched from the memory address that is currently stored in the program counter and stored into the instruction register. At the end of the fetch operation, the PC points to the next instruction that will be read at the next cycle.

This stage engage some registers mentioned earlier:

1. The CPU sends the contents of the `PC` to the `MAR` (Memory Address Register) and sends a read command on the *control bus*.
2. In response to the read command (with address equal to `PC`), the memory returns the data stored at the memory location indicated by the `PC` on the *data bus*.
3. The CPU copies the data from the data bus into its `MDR` (Memory Data Register also known as `MBR` -- Memory Buffer Register).
4. The CPU copies the data from the `MDR` to the instruction register for instruction decoding.

5. The `PC` is incremented so that it points to the next instruction. This step prepares the CPU for the next cycle.

Decode stage

During this stage, the encoded instruction presented in the instruction register is interpreted by the decoder. The decoding process allows the processor to determine what instruction is to be performed so that the CPU can tell how many operands it needs to fetch in order to perform the instruction. The opcode fetched from the memory is decoded for the next steps and moved to the appropriate registers.

Fetch operands

This stage is optional as operands are not always needed and when needed their location depends on the type of instruction and addressing. So this stage may require multiple additional steps like in case of indirect addressing or no additional action like in case of moving contents of one register to another.

Execution

In most processors, the execution of a single instruction takes place over multiple processor clock cycles. The instruction clock cycle count can vary significantly from simple instructions that require only a small number of clock cycles to complex operations that take many cycles to complete.

Any individual instruction may require multiple steps and may involve only a small portion of the circuitry present in the processor. **Most of the processor circuitry is not used in the execution of any individual instruction.** Unused components within the processor must be managed by the control unit to ensure they remain idle when not required by an executing instruction.

The CPU sends the decoded instruction as a set of control signals to the corresponding computer components. If the instruction involves arithmetic or logic, the ALU is utilized. This is the only stage of the instruction cycle that is useful from the perspective of the end-user. Everything else is overhead required to make the execute step happen.

Set of instructions

When designing a new processor architecture, it is critical to evaluate the tradeoff between the number of registers and the number and complexity of instructions available to the processor. Two approaches are possible expressed in the categorization of an architecture as **CISC** or **RISC**.

CISC (Complex Instruction Set Computer)

CISC processors are characterized as having a rich instruction set providing a variety of features, such as the ability to load operands from memory, perform an operation, and store the result to memory, all in one instruction. In a CISC processor, an instruction may take many clock cycles to execute as the processor performs all of the required subtasks. The REP MOVSD instruction mentioned previously is an example of a single instruction with a potentially lengthy execution time. CISC processors tend to have a smaller number of registers due, in part, to the die space required for the circuitry occupied by the instruction set logic. The x86 is a classic example of CISC architecture.

RISC (Reduced Instruction Set Computer)

RISC processors have a smaller number of instructions that each perform simpler tasks (compared to CISC). Performing an operation on data values stored in memory might require a pair of load instructions to load operands from memory into registers, another instruction to perform the operation, and a final instruction to store the result back to memory. **The key difference between CISC and RISC is that RISC architectures are optimized to execute individual instructions at very high speed. Even though reading memory, performing the operation, and writing the result back to memory require several more instructions in a RISC processor than in a CISC processor, the overall start-to-finish time may be comparable or even faster for the RISC processor.** Examples of RISC architectures are ARM, and RISC-V.

Instruction categories

- Memory load and store instructions
- Register-to-register data transfer instructions
- Stack instructions
- Arithmetic instructions
- Logical instructions
- Branching instructions
- Subroutine call and return instructions
- Processor fag instructions
- Interrupt-related instructions
- No operation instruction

Addressing modes

CISC processors support multiple addressing modes for instructions that require transferring data between memory and registers. RISC processors have a more limited number of addressing modes. Each processor architecture defines its collection of addressing modes based on an analysis of the anticipated memory access patterns that sofware will use on that architecture.

- Immediate addressing mode
- Direct addressing mode
- Indirect addressing mode

Interrupt

The idea

Conceptually, interrupt handling is similar to a scenario in which you are busy working on a task and your phone rings. Afer answering the call and perhaps doing some other actions as a result of this call, you resume the task that was interrupted. Humans employ several similar "interrupt" mechanisms, such as doorbells and alarm clocks, which enable us to interrupt "lower priority" activities and respond to more immediate needs.

Interrupts in computers

An interrupt (sometimes referred to as a trap) is a request for the processor to interrupt currently executing code (when permitted), so that the event can be processed in a timely manner. If the request is accepted, the processor will suspend its current activities, save its state, and execute a function called an **interrupt handler** (or an **interrupt service routine**, ISR) to deal with the event. This interruption is often temporary, allowing the software to resume normal activities after the interrupt handler finishes, as in most cases it serves well known events required for correct CPU work, although the interrupt could instead indicate a fatal error.

There are several different architectures for handling interrupts. In some, there is a single interrupt handler that must scan for the highest priority enabled interrupt. In others, there are separate interrupt handlers for separate interrupt types, separate I/O channels or devices, or both. Several interrupt causes may have the same interrupt type and thus the same interrupt handler, requiring the interrupt handler to determine the cause.

Interrupt signals may be issued in response to hardware or software events. These are classified as **hardware interrupts** or **software interrupts**, respectively.

Hardware interrupts

A hardware interrupt is a condition related to the state of the hardware that may be signaled by an external hardware device, e.g., an interrupt request (IRQ) line on a PC, or detected by devices embedded in processor logic, to communicate that the device needs attention from the operating system or, if there is no OS, from the bare metal program running on the CPU. Such external devices may be part of the computer (e.g., disk controller) or they may be

external peripherals. For example, pressing a keyboard key or moving a mouse plugged into a PS/2 port triggers hardware interrupts that cause the processor to read the keystroke or mouse position.

Hardware interrupts can arrive asynchronously with respect to the processor clock, and at any time during instruction execution. Consequently, all incoming hardware interrupt signals are conditioned by synchronizing them to the processor clock, and acted upon only at instruction execution boundaries.

The processor samples the interrupt trigger signals or interrupt register during each instruction cycle, and will process the highest priority enabled interrupt found. Regardless of the triggering method, the processor will begin interrupt processing at the next instruction boundary following a detected trigger, thus ensuring:

- The processor status is saved in a known manner. Typically the status is stored in a known location, but on some systems it is stored on a stack.
- All instructions before the one pointed to by the PC have fully executed.
- No instruction beyond the one pointed to by the PC has been executed, or any such instructions are undone before handling the interrupt.
- The execution state of the instruction pointed to by the PC is known.

Nested interrupt

An interrupt handled while processing another interrupt is refferred to as a **nested interrupt**.

Masking

To *mask* an interrupt is to disable it, so it is deferred or ignored by the processor, while to *unmask* an interrupt is to enable it.

Some interrupt signals are not affected by the interrupt mask and therefore cannot be disabled; these are called **non-maskable interrupts** (NMIs). These indicate high-priority events which cannot be ignored under any circumstances, such as the timeout signal from a watchdog timer.

Note: watchdog timer

A watchdog timer (or simply a watchdog), sometimes called a computer operating properly timer (COP timer), is an electronic or software timer that is used to detect and recover from computer malfunctions. Watchdog timers are widely used in computers to facilitate automatic correction of temporary hardware faults, and to prevent errant or malevolent software from disrupting system operation.

During normal operation, the computer regularly restarts the watchdog timer to prevent it

from elapsing, or "timing out". If, due to a hardware fault or program error, the computer fails to restart the watchdog, the timer will elapse and generate a timeout signal. The timeout signal is used to initiate corrective actions. The corrective actions typically include placing the computer and associated hardware in a safe state and invoking a computer reboot.

Software interrupts

A software interrupt is requested by the processor itself upon executing particular instructions or when certain conditions are met. Every software interrupt signal is associated with a particular interrupt handler.

A software interrupt may be intentionally caused by executing a special instruction which, by design, invokes an interrupt when executed. Such instructions function similarly to subroutine calls and are used for a variety of purposes, such as requesting operating system services and interacting with device drivers (e.g., to read or write storage media). Software interrupts may also be triggered by program execution errors or by the virtual memory system.

Typically, the operating system kernel will catch and handle such interrupts. Some interrupts are handled transparently to the program -- for example, the normal resolution of a page fault is to make the required page accessible in physical memory. But in other cases such as a segmentation fault the operating system executes a process callback.

Interrupt working cycle

Typically CPU cannot respond instantly to a low signal level on the IRQ input. Before the CPU can begin to process the interrupt, it must first complete the instruction already in progress. Next, it pushes the return address (the address of the next instruction that would have been executed afer the instruction in progress) onto the stack, followed by the processor flags register. In the next stage of interrupt processing, the processor loads the address of the IRQ handler routine from memory addresses into the PC register. Then the CPU begins executing the interrupt handler code at that address. When the interrupt handler is finished, the CPU pops the processor flags and the PC register values from the stack and resumes execution at the instruction following the instruction that was in progress when the IRQ input was driven low.

Bibliography

1. William Stallings, *Computer Organization and Architecture*, Pearson, 11th edition (2019)
2. Jim Ledin, *Modern Computer Architecture and Organization*, Packt, April 2020
3. *Ken Shirriff's blog: Inside the die of Intel's 8087 coprocessor chip, root of modern*

floating point, http://www.righto.com/2018/08/inside-die-of-intels-8087-coprocessor.html Retrieved 2023-10-21.

4. *cpu-collection.de*, http://www.cpu-collection.de/?tn=0&l0=co&l1=Intel Retrieved 2023-10-21.