**Chapter 14: Processor Structure and Function**

Table of Contents

[14.1 Processor Organization 3](#_Toc49000012)

[14.2 Register Organization 5](#_Toc49000013)

[User-Visible Registers 5](#_Toc49000014)

[Control and Status Registers 9](#_Toc49000015)

[Example Microprocessor Register Organization 12](#_Toc49000016)

[14.3 Instruction Cycle 15](#_Toc49000017)

[The Indirect Cycle 15](#_Toc49000018)

[Data Flow 16](#_Toc49000019)

[14.4 Instruction Pipelining 19](#_Toc49000020)

[Pipelining Strategy 19](#_Toc49000021)

[Pipelining Performance 26](#_Toc49000022)

[Pipeline Hazards 28](#_Toc49000023)

[Resource Hazards 28](#_Toc49000024)

[Data Hazards 29](#_Toc49000025)

[Control Hazards 31](#_Toc49000026)

[Dealing with Branches 31](#_Toc49000027)

[Multiple Streams 32](#_Toc49000028)

[Pre-Fetch Branch Target 32](#_Toc49000029)

[Loop Buffer 32](#_Toc49000030)

[Branch Prediction 34](#_Toc49000031)

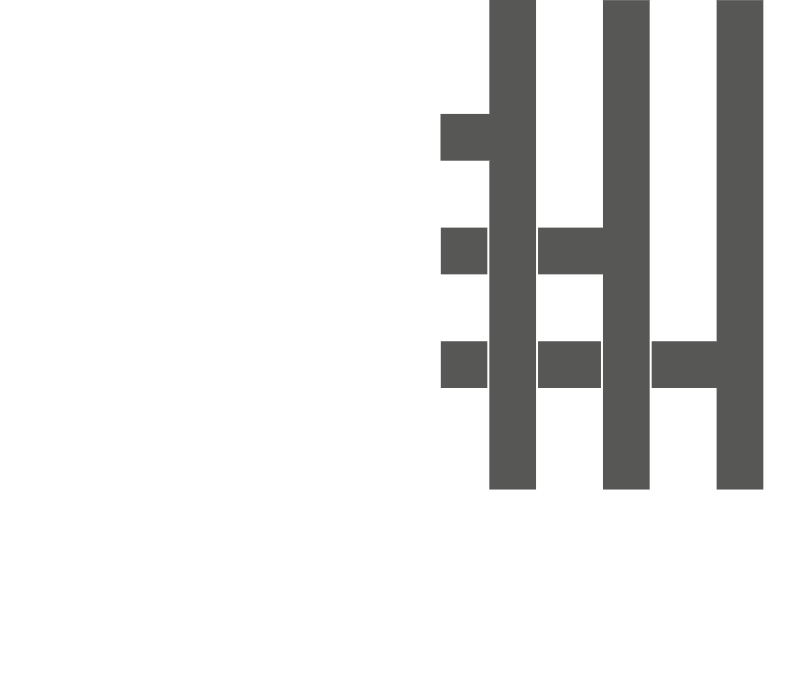
[Intel 80486 Pipelining 39](#_Toc49000032)

## 14.1 Processor Organization

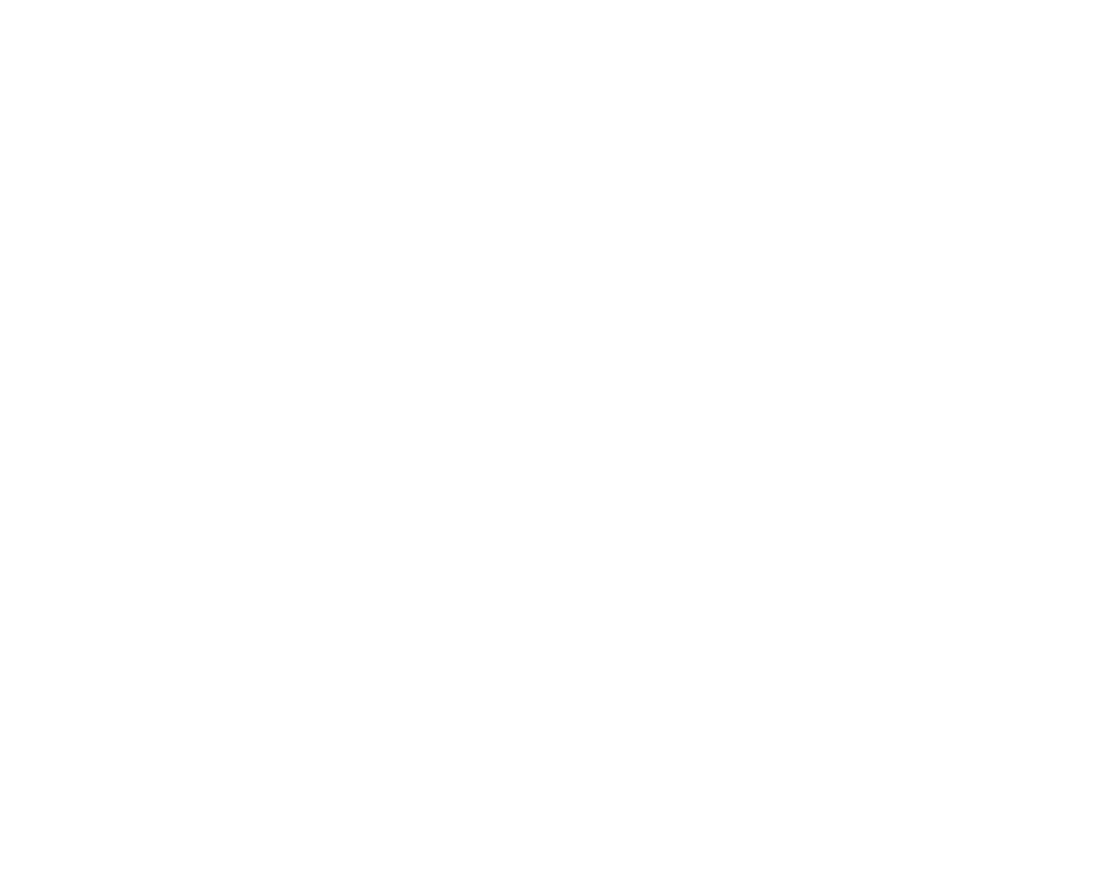
The processor has a few things to do:

* **Fetch Instructions**: Read instruction from memory (register, cache or main memory)
* **Interpret Instruction**: Decode the instruction to determine what action must be taken
* **Fetch Data**: The instruction may require data to be read from memory or an I/O module
* **Process Data**: The instruction may require arithmetic or logical operations on some data
* **Write Data**: The results of the executed instruction may need to be written to memory or to an I/O module

To do these things, it should be clear that the processor will need to store some data temporarily. It must remember the location of the last instruction so that it can find the next instruction. It must store instructions and data temporarily while and instruction is being executed. Basically, the processor needs its own memory.



The figure above is a simplified view of a processor, connecting it to the rest of the system via the system bus. The major components of the processor are the ALU and the CU. The ALU performs computation and processing of data and the CU controls movement of instructions and data into and out of the processor and also controls the ALU. A minimal internal memory, consisting of a set of storage locations, is also shown. These are called registers.



The figure above is a more detailed view of the processor. The data transfer and logic control paths are shown. The internal CPU bus transfer data between the various registers and the ALU, because the ALU actually only operates on data in the internal processor memory. Typical basic elements of the ALU are also shown. Note the similarities between the internal structure of the processor and the internal structure of the computer as a whole. Both have a small collection of components connected by data paths.

## 14.2 Register Organization

We know that a computer system has a memory hierarchy. At higher levels, memory is faster, smaller and more expensive. Within the processor, there is a set of registers that are above the main memory and cache in this hierarchy. These registers have two roles:

* **User-Visible Registers**: These allow the machine or assembly language programmer to minimize main memory references by optimizing use of registers.
* **Control and Status Registers**: These are used by the control unit to control the operation of the processer and by privileged, operating system programs to control the execution of programs.

Registers are not cleanly separated into just these two categories. For example, on some machines, the program counter is user visible, while on most it is not. For now, however, we will look at just these categories.

### User-Visible Registers

A user-visible register is one that can be referenced by means of the machine language that the processor executes. We can put these registers into the following categories:

* General-Purpose
* Data
* Address
* Condition Codes

General purpose registers can be assigned to many different functions by the programmer. Sometimes their use in the instruction set is orthogonal to the operation, meaning any general-purpose register can contain the operand for any op-code. This provides true general-purpose usage. Often though, there are limitations. For example, there may be dedicated registers for floating-point and stack operations.

In some cases, general purpose registers can be used for addressing functions (e.g. register indirect, displacement). In other cases, there is a partial or clean separation between data registers and address registers.

Data registers may be used only to hold data and cannot be used in calculations for an operand address.

Address registers may themselves be somewhat general-purpose, or they may be devoted to a particular addressing mode. For example:

* **Segment Pointers**: In a machine with segmented addressing, a segment register holds the address of the base of the segment. There may be multiple registers, such as one for the OS and one for the current process.
* **Index Registers**: These are used for indexed addressing and may be auto-indexed.
* **Stack Pointer**: If there is user-visible stack addressing, then there is typically a dedicated register that points to the top of the stack. This allows implicit address, i.e. push, pop and other stack instructions need not contain an explicit stack operand.

There are many design issues here, an important one being whether to use completely general-purpose registers or to specialize their use. With specialized registers, the register an operand specifier refers to is generally implicit in the op-code. The operand need only specify one out of a set of specialized registers instead of one out of all of them, which saves bits. On the other hand, the specialization limits the programmer’s flexibility.

Another design issue is the number of registers of each type to be provided. Again, this affects instruction set design since more registers need more operand specifier bits. As discussed, somewhere between 8 to 32 registers should be optimal. Fewer registers cause more memory references, while more does not noticeably reduce the number of memory references. However, there is a new approach that finds advantage in using hundreds of registers. This is seen in some RISC systems and is discussed in the next chapter.

Finally, there is the issue of register length. Registers that hold addresses must be long enough to hold the longest address. Data registers need to be able to hold values of most data types. Some machines allow two contiguous registers to be used as one to hold double-length values.

The final category of registers, and the one least visible to users, holds condition codes or flags. These are bits set by the processor hardware as a result of operations. For example, an arithmetic operation may give a positive, negative, zero or overflow result. In addition to the result itself, a condition code is also stored in a register or memory. This code may be tested later on as part of a conditional branch operation.

Condition code bits are collected into one or more registers. Usually, they form part of a control register. Generally, machine instructions allow these bits to be read by implicit reference, but the programmer cannot alter them.

Many processors do not use condition codes at all, instead allowing conditional branch instructions to make a comparison on the fly and act on the results without storing a condition code. The table below shows the major advantages and disadvantages of condition codes.

|  |  |
| --- | --- |
| Advantages | Disadvantages |
| 1. Because condition codes are set by normal arithmetic and data movement instructions, they should reduce the number of COMPARE and TEST instructions needed. | 1. Condition codes add complexity, both to the hardware and software. Condition code bits are often modified in different ways by different instructions, making life more difficult for both the micro programmer and compiler writer. |
| 2. Conditional instructions, such as BRANCH, are simplified relative to composite instructions, such as TEST and BRANCH. | 2. Condition codes are irregular; they are typically not part of the main data path, so they require extra hardware connections. |
| 3. Condition codes facilitate multiway branches. For example, a TEST instruction can be followed by two branches, one on less than or equal to zero and one on greater than zero. | 3. Often, condition code machines must add special non-condition-code instructions for special situations anyway, such as bit checking, loop control and atomic semaphore operations. |
| 4. Condition codes can be saved on the stack during subroutine calls, along with other register information. | 4. In a pipelined implementation, condition codes require special synchronization to avoid conflicts. |

In some machines, a subroutine call will result in all the user-visible registers being saved so that they can 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. Other machines put the responsibility on the programmer, so they need to include instructions to do this in the program.

### Control and Status Registers

Many processor registers are used to control the operation of the processor. Most of these are not usually visible to the user. Some may be visible to machine instructions executed in a control or operating system mode.

Different machines have different register organizations and terminology. A reasonably complete list of register types is given below. Four registers are essential to instruction execution:

* **Program Counter (PC)**: This contains the address of an instruction to be fetched.
* **Instruction Register (IR)**: This contains the instruction most recently fetched.
* **Memory Address Register (MAR)**: This contains the address of a location in memory.
* **Memory Buffer Register (MBR)**: This contains a word of data to be written to memory or the word most recently read from memory.

Not all processors have internal registers designated as MAR and MBR, but some buffering mechanism must be involved to stage the bits that must be transferred to the system bus and to temporarily store bits that are read from the data bus.

Typically, the processor updates the PC after each instruction fetch so that it always points to the next instruction to be executed. A branch or skip instruction also modifies the contents of the PC. The fetched instruction is loaded into an IR, where the op-code and operand specifiers are analysed. Data is exchanged with memory using the MAR and MBR. In a bus-organized system, the MAR connects directly to the address bus and the MBR connects directly to the data bus. User-visible registers in turn exchange data with the MBR.

These four registers allow movement of data between the processor and memory. Inside the processor, data must be given to the ALU for processing. The ALU could have direct access to the MBR and user-visible registers, or there could be more buffering registers at the boundary to the ALU. Any buffer registers would serve as input and output registers for the ALU, exchanging data with the MBR and user-visible registers.

Many processor designs include a register or set of registers, often called the program status word (PSW). This contains status information. The PSW usually contains condition codes and other status information. Common fields or flags are:

* **Sign**: Contains the sign bit of the result of the last arithmetic operation
* **Zero**: Set when the result is
* **Carry**: Set if an operation caused a carry into or borrow out of a higher-order bit; used for multi-word arithmetic operations
* **Equal**: Set if a local compare result is equality
* **Overflow**: Set to indicate arithmetic overflow
* **Interrupt Enable/Disable**: Use to enable or disable interrupts
* **Supervisor**: Indicates that the processor is executing in supervisor mode, as opposed to user mode; certain privileged instructions can only be executed and certain areas of memory access in supervisor mode

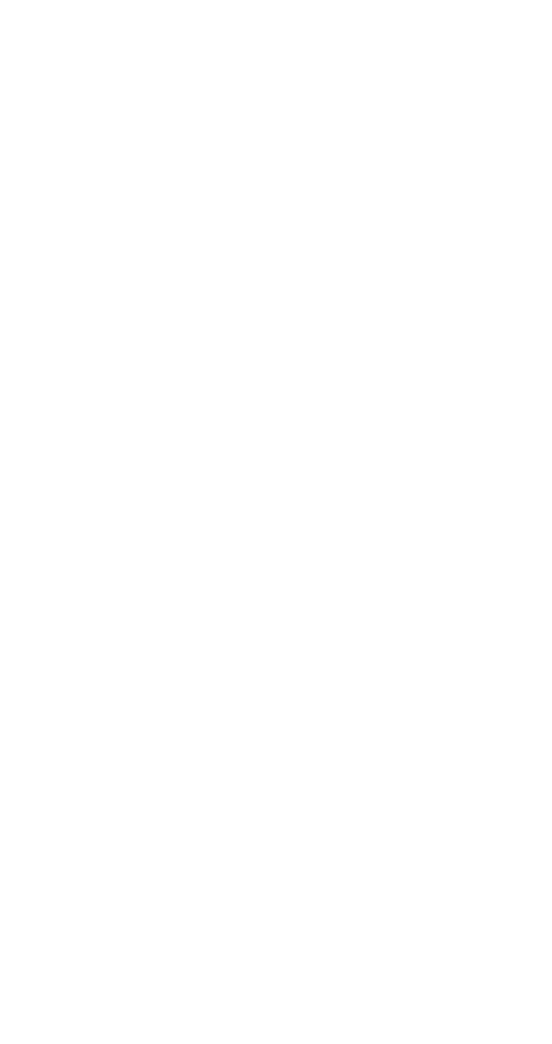
Many other registers related to status and control bits may be found in a particular processor design. There may be a pointer to a block of memory containing additional status information (e.g. process control blocks). In machines using vectored interrupts, an interrupt vector register could exist. If a stack is used to implement certain functions (e.g. a subroutine call), then a system stack pointer is needed. A page table pointer is needed for a virtual memory system. Finally, registers may be used in the control of I/O operations.

Several factors must be considered when designing the control and status register organization. One key issue is OS support. Certain types of control information are of specific use to the OS. If the processor designer has a functional understanding of the OS to be used, then the register organization can, to some extent, be tailored to the OS.

Another major design issue is the allocation of control information between registers and memory. It is common to dedicate the first (lowest) few hundred or thousand words of memory for control purposes. The designer must decide how much control information should be in registers and how much should be in memory. The usual trade-off of cost against speed will arise here.

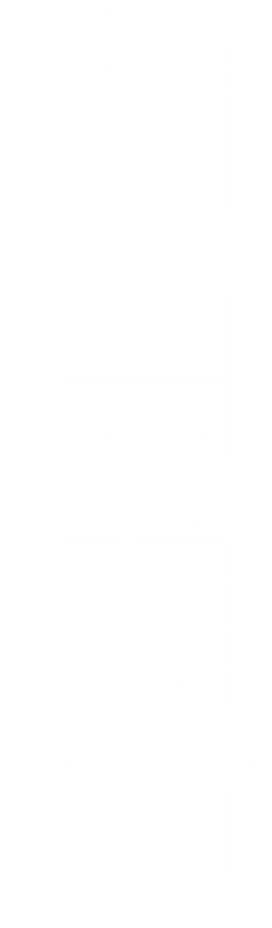
### Example Microprocessor Register Organization

Now we will look at two register organizations for two different microprocessors. Purely internal registers, like the MAR, are not shown.



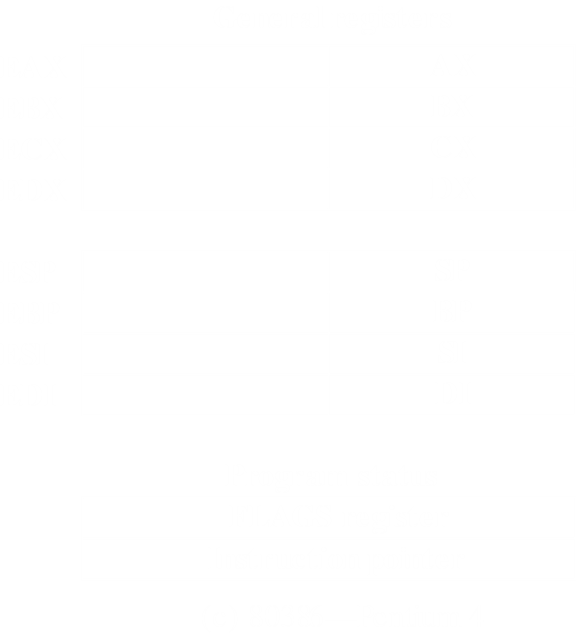
In this organization, the 32-bit registers are partitioned into 8 data registers and 9 address registers. The 8 data registers are primarily for data manipulation and also used in addressing as index registers. The width of the registers allows 8-, 16- and 32-bit data operations, depending on the op-code. The address registers contain 32-bit addresses with no segmentation. Two of these registers are also used as stack pointers, one for users and one for the OS, depending on the current execution mode. Both registers are numbered 7 since only one can be used at a time. There is also a 32-bit program counter and a 16-bit status register.

The team that designed this wanted a very regular instruction set with no special-purpose registers. A concern for code efficiency lead to the division of the registers into two functional components, saving one bit on each register specifier. This is a reasonable compromise between complete generality and code compaction.



This register takes a different approach. Every register is special purpose, although some are also usable as general-purpose. There are 4 16-bit data registers that are addressable on a byte or 16-bit basis, and 4 16-bit pointer and index registers. The data registers can be used as general-purpose in some instructions, while in others, the registers are used implicitly. For example, a multiply instruction always uses the accumulator. The 4 pointer registers are also used implicitly in a number of operations, each containing a segment offset. There are also 4 16-bit segment registers, three of which are used in a dedicated, implicit fashion to point to the segment of the current instruction (useful for branch instructions), a segment containing data and a segment containing a stack respectively. These dedicated and implicit uses provide a compact encoding at the cost of reduced flexibility. There is also an instruction pointer and a set of 1-bit status and control flags.

The point of this comparison is to show that there is no universally accepted philosophy concerning the best way to organize processor registers. As with overall instruction set design and so many other processor design issues, it is still a matter of judgement and taste.



A second instructive point about register organization is show above. This figure shows the user-visible register organization for a 32-bit microprocessor. It uses 32-bit registers. However, this processor was designed as an extension to an earlier processor. As such, it retains the original register organization embedded in the new organization, so as to provide upward compatibility for programs written on the earlier machine. Given this design constraint, the architects had limited flexibility.

## 14.3 Instruction Cycle

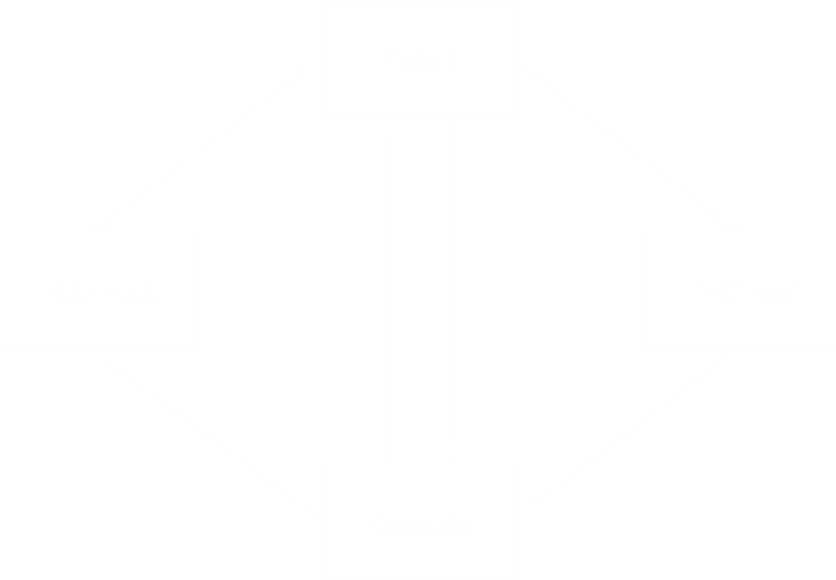
An instruction cycle has the following stages:

* **Fetch**: Read the next instruction from memory into the processor
* **Execute**: Interpret the op-code and perform the indicated operation
* **Interrupt**: If interrupts are enabled and an interrupt has occurred, save the current process state and service the interrupt

There is one more stage that we have not yet introduced, the indirect cycle.

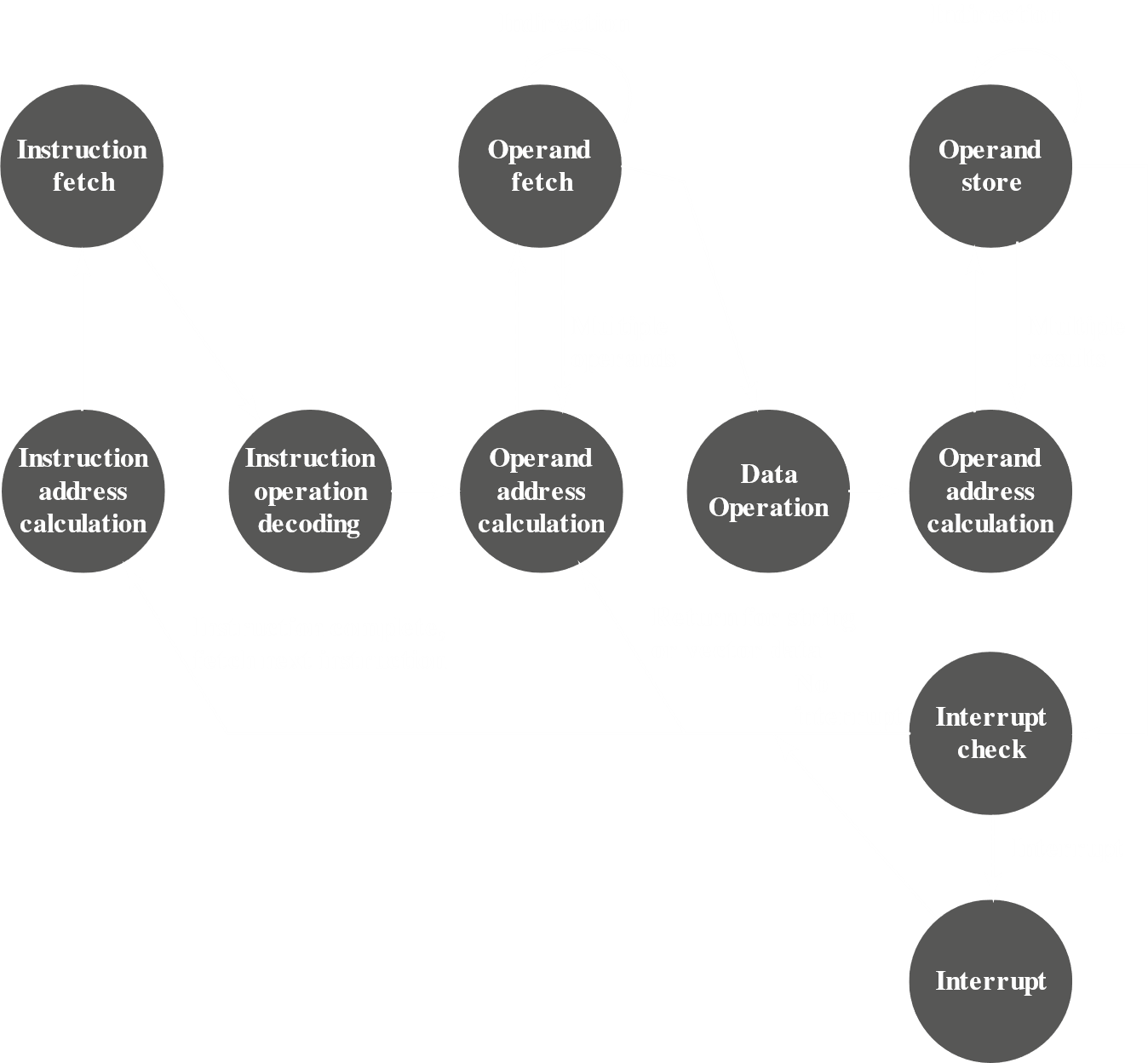
### The Indirect Cycle

We have seen that the execution of an instruction may involve one or more operands, each of which requires a memory access. If indirect addressing is used, additional memory access is also required. We can think of the fetching of indirect addresses as one more instruction stage. This results in the diagram below:



The main activity is alternating between instruction fetches and executions. When an instruction is fetched, we check if indirect addressing is used. If so, the required operands are fetched using indirect addressing. After execution, an interrupt may be processed before the next instruction fetch.

The overall instruction cycle looks like this:



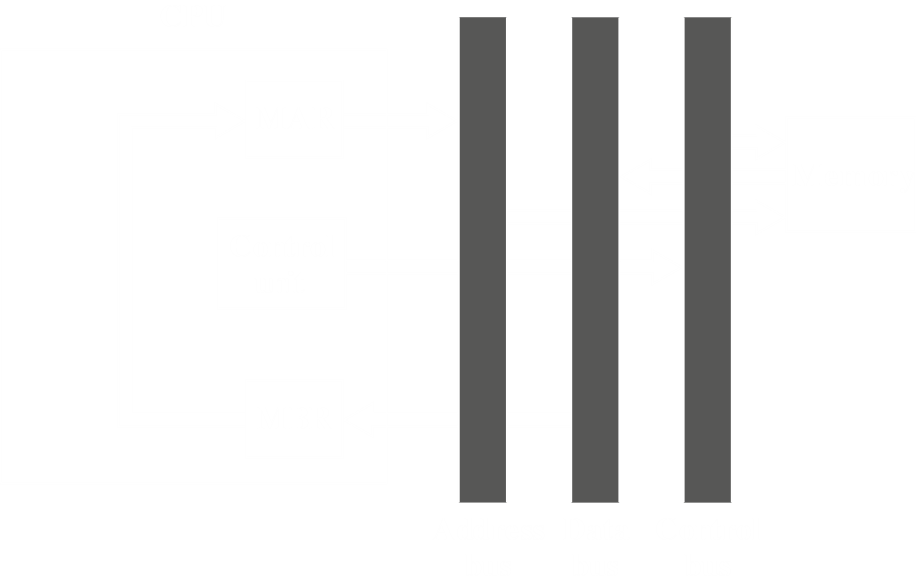
Once an instruction is fetched, its operand specifiers are identified. Each input operand in memory is fetched, which may require indirect addressing. Register-based operands do not need to be fetched. Once the op-code is executed, a similar process may be needed to store the results in main memory.

### Data Flow

The exact sequence of events during an instruction cycle depends on the design of the processor. We can however, indicate in general terms what has to happen.

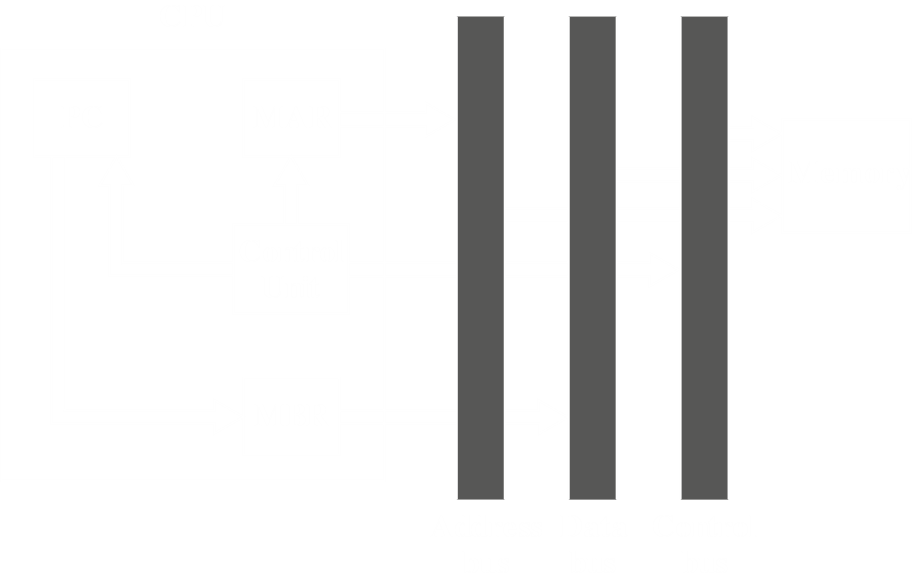


During the fetch cycle, an instruction is read from memory. The diagram above shows the flow of data during this cycle. The PC has the address of the next instruction, and this address is moved to the MAR, which places it on the address bus. The CU requests a memory read, and the result is placed on the data bus and copied to the MBR. This is then moved to the IR. The PC is incremented by 1, in preparation for the next instruction.



Once the fetch cycle is over, the CU checks the contents of the IR to see if it has an operand specifier that uses indirect addressing. If it does, an indirect cycle is performed. The data flow for this cycle is shown above. The rightmost N bits of the MBR, which contain the address reference, are transferred to the MAR. Then the CU requests a memory read, to get the desired address of the operand into the MBR.

The execute cycle has many forms, depending on which machine instruction is in the IR. This cycle may involve transferring data among registers, read or write from memory or I/O and/or the invocation of the ALU.



The data flow for the interrupt cycle is shown above. The current contents of the PC are saved so the processor can resume normal activity after the interrupt. This is done by moving the contents of the PC to the MBR, from where it is written to memory. The special memory location used for this is loaded into the MAR from the CU. It may for example, be a stack pointer. Next, the PC is loaded with the address of the interrupt routine. Thus, the next instruction cycle begins by fetching the required instruction.

## 14.4 Instruction Pipelining

Improvements in technology gives us advantages we can use to achieve greater performance. Simple things like faster circuitry help, but organizational enhancements to the processor can improve performance much more. We have seen some advantages of this, such as multiple registers instead of a single accumulator, and the use of cache memory. Another common organizational approach is called instruction pipelining.

### Pipelining Strategy

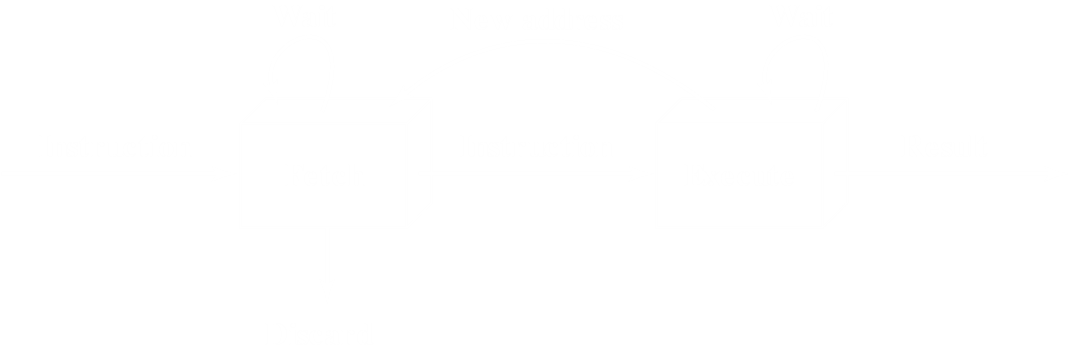
Instruction pipelining is like an assembly line in a manufacturing plant. Products go through various stages, so we can put the production process into an assembly line and work on different products at different stages of production at the same time. This is referred to as pipelining since new input is accepted at one end first, and then the inputs appear as outputs one by one on the other end.

Remember that an instruction has several stages, which provides a good opportunity for pipelining. Before getting into the complicated details, consider a simple approach where an instruction is divided into two parts, fetching and execution. Fetching an instruction requires memory access, but execution does not. As such, when one instruction is in the execution stage, we could have another instruction in the fetching stage, i.e. we could actually fetch the instruction during this time.



This pipeline has two stages. The first stage fetches an instruction and buffers it. When the second stage is free, the buffered instruction is passed to it. When the second stage is executing this instruction, the first stage takes advantage of the unused memory cycles to fetch another instruction and buffer it. This is called instruction pre-fetch or fetch overlap. Note that this approach requires more registers. In fact, pipelining in general requires more registers to store data in between stages.

Obviously, this process will speed up instruction execution as a whole. If the fetch and execute cycles were of equal duration, the instruction cycle time would be halved. However, this is unlikely to happen. To understand why, consider the more detailed view of the process given below.



Generally, the execution time will be longer than the fetch time, since it involves reading and storing operands as well as some operations. This means the fetch stage will have to wait to empty its buffer.

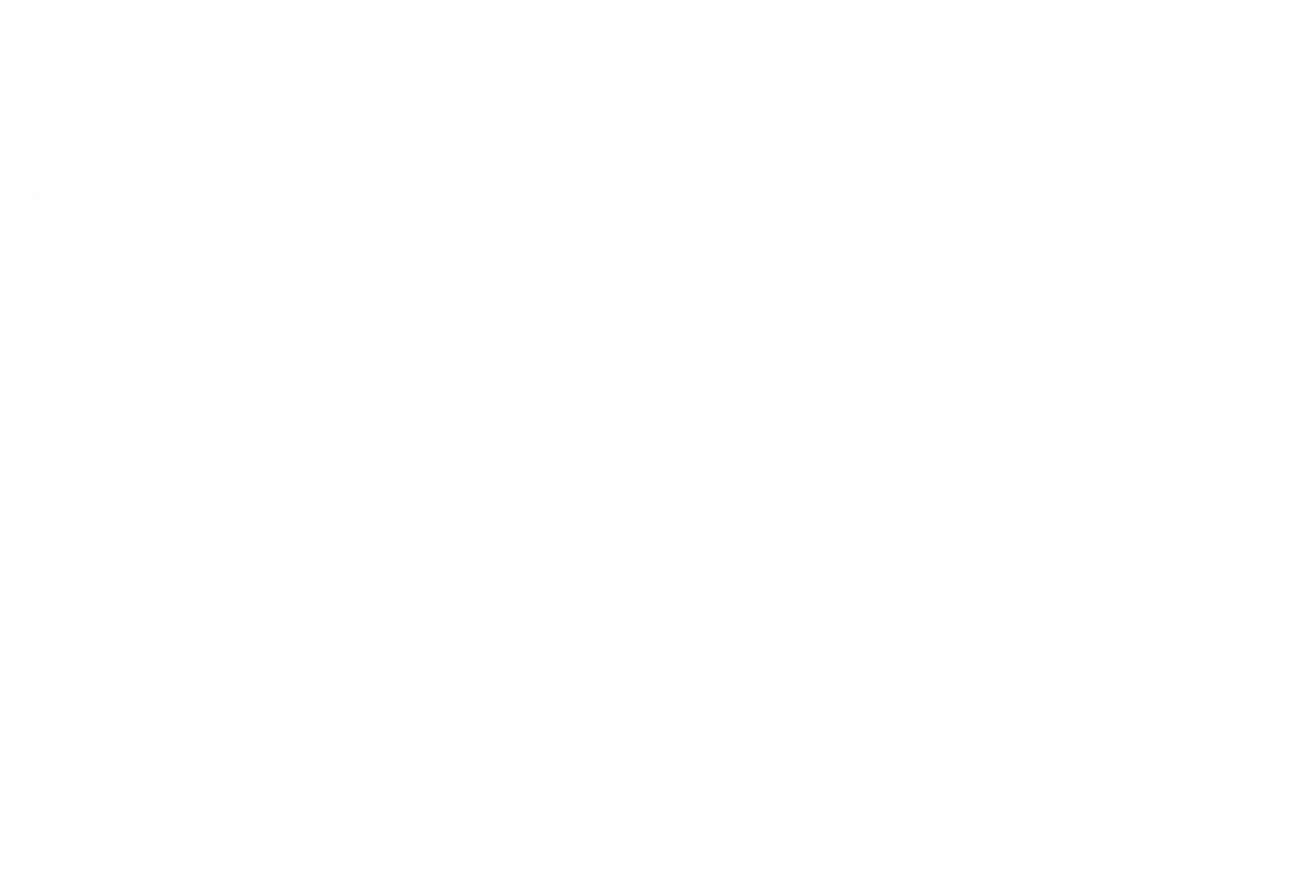
A second point is that a conditional branch would make the address of the next instruction unknown, which means the execute stage would need to inform the fetch stage about the address before it can fetch it. This can result in the execute stage having to wait for the fetch stage to get the next instruction.

The second shortcoming can be somewhat reduced. The fetch stage could simply get the next instruction after a conditional branch anyways. If the branch is not taken, then the next instruction will be needed which is good, but if the branch is taken then the fetched instruction would need to be discarded and the required one brought in.

Although there are some drawbacks, the overall effect of pipelining is still an improvement to execution speed. To achieve an even greater speed, there have to be more stages. Consider the following breakdown:

* **Fetch Instruction (FI)**: Read the next expected instruction into a buffer.
* **Decode Instruction (DI)**: Determine the opcode and the operand specifiers.
* **Calculate Operands (CO)**: Calculate the effective address of each source operand. This may involve displacement, register indirect, indirect, or other forms of address calculation.
* **Fetch Operands (FO)**: Fetch each operand from memory. Operands in registers do not need to be fetched.
* **Execute Instruction (EI)**: Perform the indicated operation and store the result, if any, in the specified destination operand location.
* **Write Operand (WO)**: Store the result in memory.

Under this breakdown, the stages are more likely to be equal. Let us assume that they are equal and look at the six-stage pipeline depiction below. We have managed to reduce 9 instructions taking a total of 54 time units to just 14 time units.

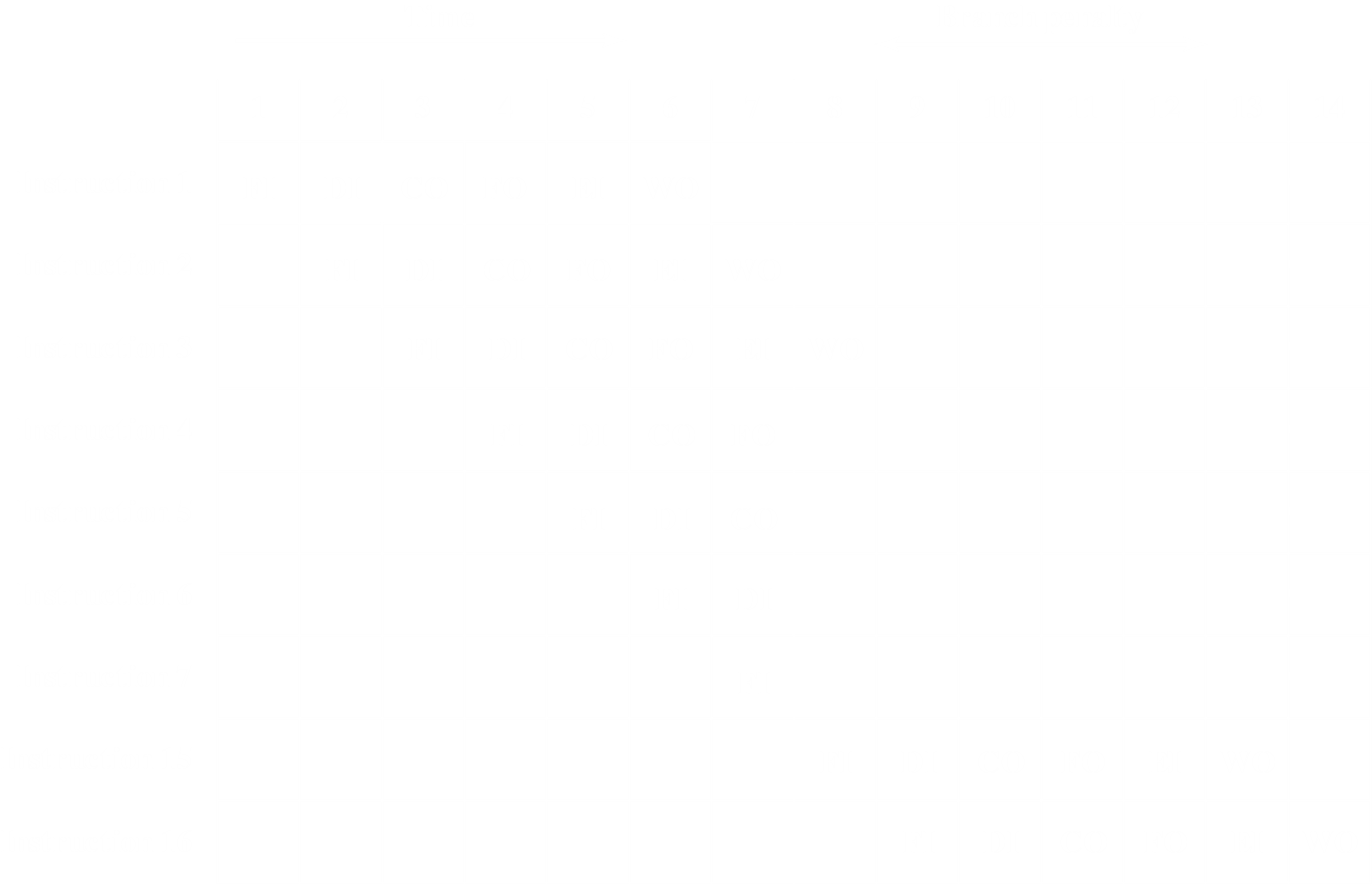


We do however, need to make several notes here. For one, the diagram assumes that every instruction has to go through all the stages, which is not true. Say we have a load instruction that does not need the WO stage. To simplify the hardware though, the timing is set up under this assumption.

We also assumed that all the stages can be performed parallelly and specifically that there would be no memory conflicts. The FI, FO and WO stages in particular require memory access, and we are assuming that these accesses can occur simultaneously. Most memory systems would not allow this. There could still be values in the cache or unneeded FO or WO stages though. This results in the pipeline not being significantly slowed down.

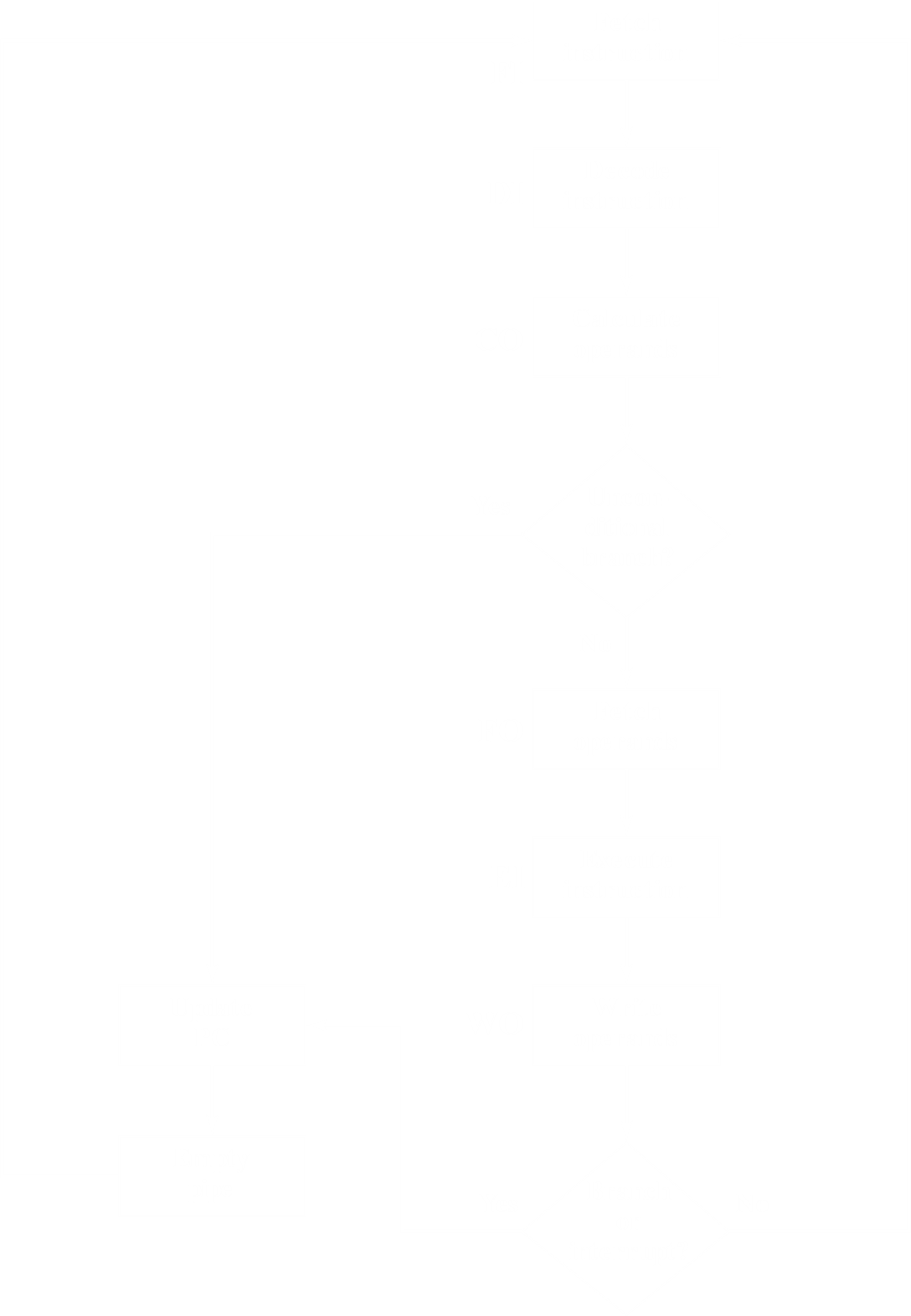
We will also most likely not have all the stages be of equal duration, and some waiting could be involved, as discussed earlier.

Having conditional branches could invalidate several instruction fetches, as could an interrupt. Consider the figure below.



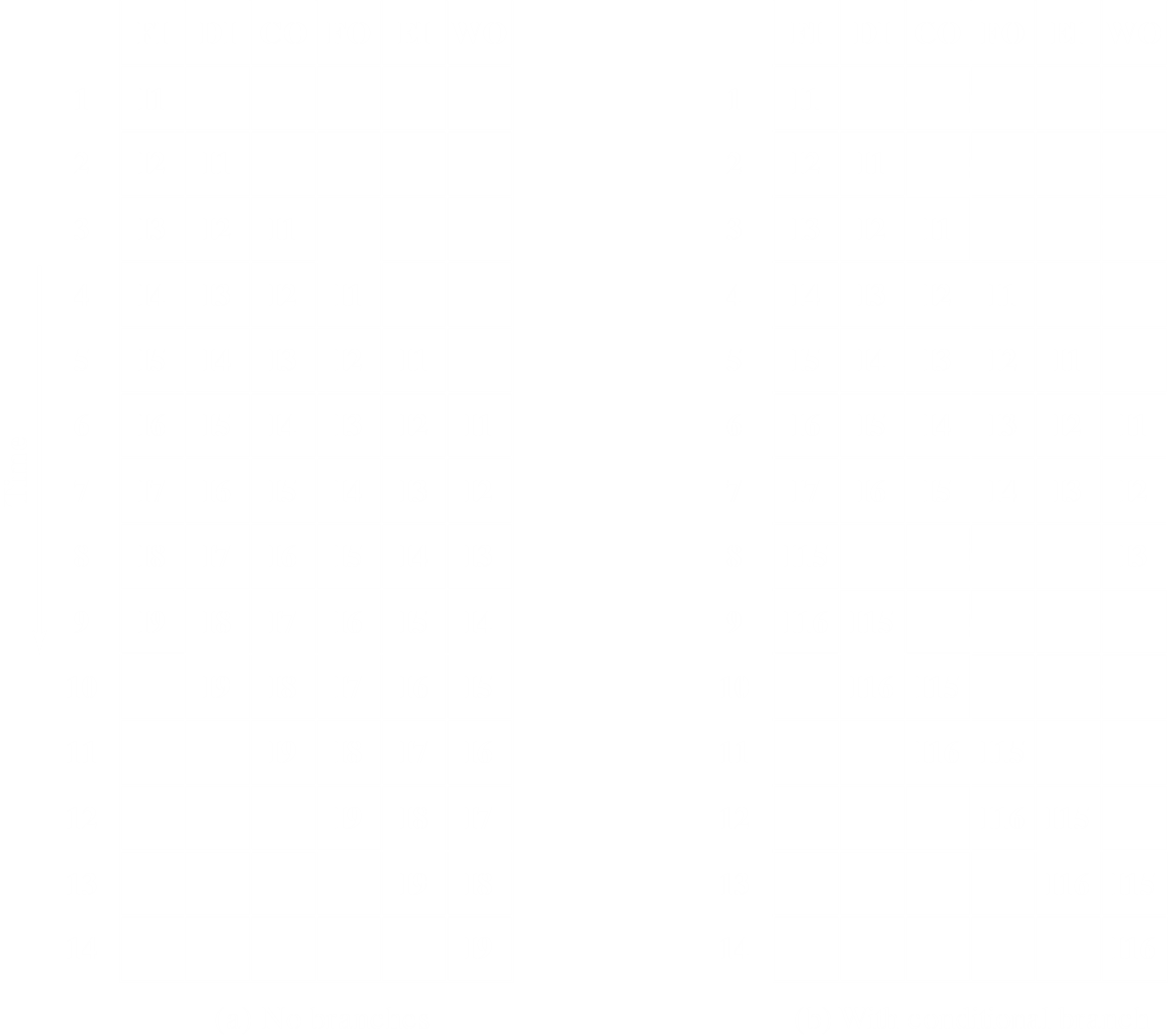
The third instruction, which had a conditional branch to the fifteenth instruction, forced the pipeline to discard the stages it had already gone over for the instructions after the third one. The pipeline did not know before the third instruction had completed executing that it would need to make this change, so it kept putting the next few instructions through the normal process and only stopped when it got to know about the branch. Then it had to fetch the fifteenth instruction. This clearly resulted in a huge loss of time.

The process for branching is shown in the diagram below:



There are more problems that did not arise in our first simplistic version of pipelining. The CO stage might depend on the contents of a register that could be altered by a previous instruction that is still in the pipeline, for example. Other such register and memory conflicts could also occur. The system needs to include logic that accounts for such conflicts.

It might be useful to see the depiction of the pipelining timing diagram vertically. Notice how on the left-hand diagram with no branching, the pipeline is full at time 6, with 6 different instructions at various stages and remains full until time 9 (since we assume instruction 19 is the last one). On the right-hand diagram, the pipeline is only full at times 6 and 7. At time 7, the third instruction causes the pipeline to be flushed, this results in only instructions 13 and 115 being present in the pipeline at time 8.



It may now seem that the more stages in a pipeline there are, the faster the execution rate. However, this is countered by two points.

Firstly, at each stage there is some overhead in moving data from buffer to buffer and performing various preparation and delivery functions. This overhead can significantly lengthen the execution time of a single function, especially when sequential instructions are dependent on each other.

Secondly, the amount of control logic required to handle memory and register dependencies and to optimize their use of the pipeline increases enormously with the number of stages. This can lead to the logic controlling the gating between stages becoming more complex than the stages themselves.

Overall, pipelining is a powerful technique for enhancing performance, but it needs to be designed carefully to reach optimum results with reasonable complexity.

### Pipelining Performance

The cycle time, is one unit of time.

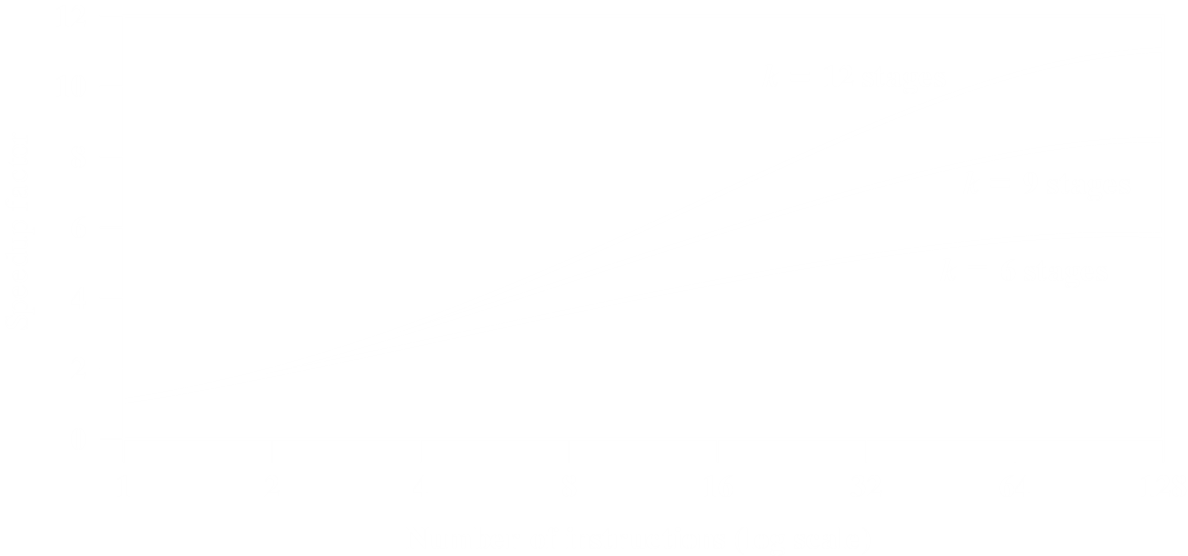
where .

Say we have instructions.

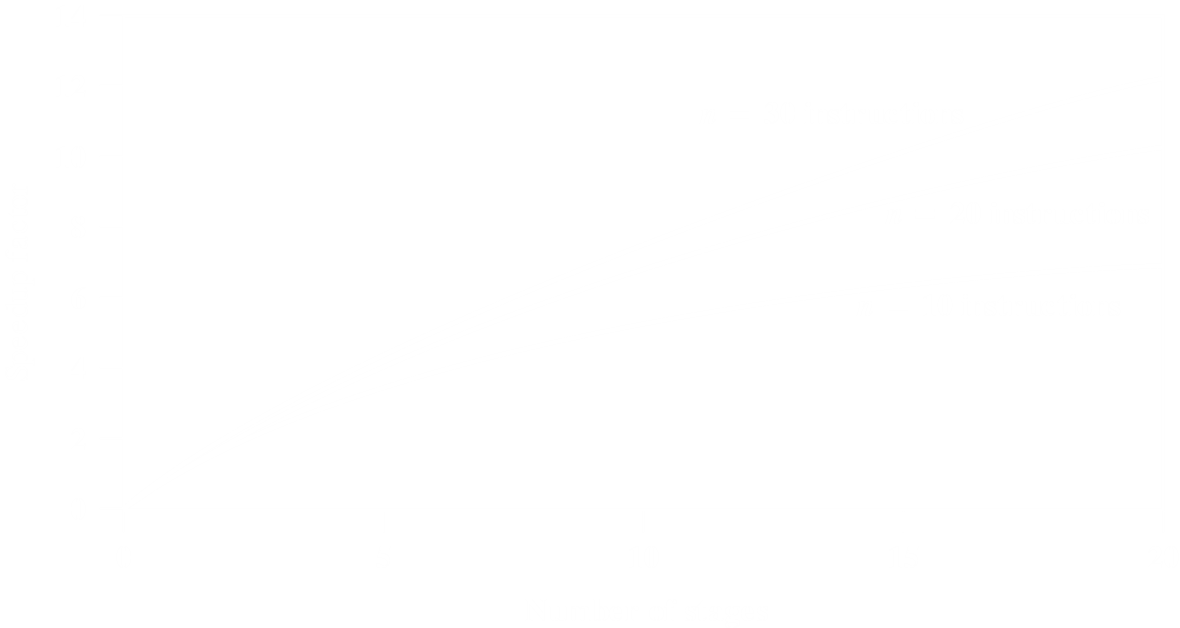
A total of is only valid when all the stages are full, but we are ignoring that in this equation. The equation can easily be verified using one of the timing diagrams we saw in the last section.

Say we have a processor with no pipelining, but the same functions, which has an instruction cycle time of . The speedup factor for using an instruction pipeline compared to not using it is

The graph below plots the speedup factor as a function of the number of instructions executed, without branches. As -fold speedup.



The graph shown next is the speedup factor as a function of the number of stages in the instruction pipeline. The speedup factor approaches the number of instructions without branches that can be fed into the pipeline, which seems to indicate that more stages would be beneficial (since we could add more instructions). However, practically, we know that the gains are not worth the cost encountered by increases in cost, between stages and the fact that we will face branches and have to flush the entire pipeline at some point.



### Pipeline Hazards

We will now examine the problems with pipeline performance in a more systematic way. A pipeline hazard occurs when some part of the pipeline must stop because conditions do not allow it to continue execution. A pipeline stall like this is also called a pipeline bubble. Hazards are of three types: resource, data and control.

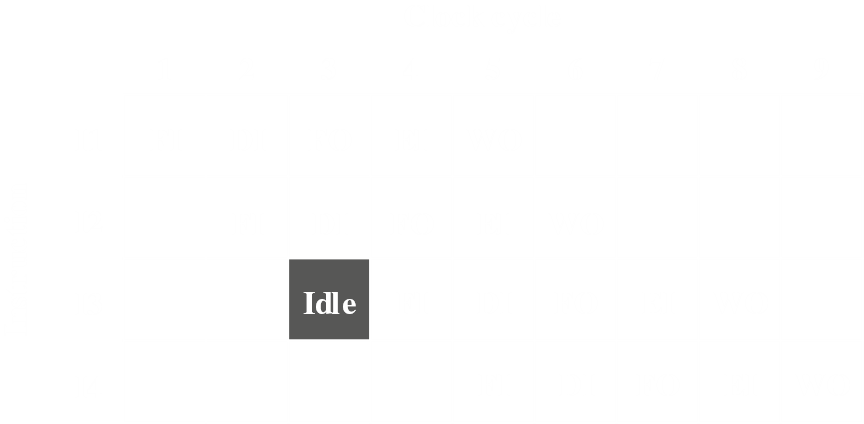
#### Resource Hazards

A resource hazard, also called a structural hazard, is when two or more instructions in the pipeline need the same resource. This causes instructions to be executed in serial for that portion instead of parallelly.

Consider this simple example in an ideal state.



Now consider that main memory has a single port and all instruction fetches and data reads and writes must be performed one at a time (ignore the cache). Thus, an operand read or write cannot be performed in parallel with an instruction fetch.



Notice how the fetch instruction for the 13th instruction had to wait one clock cycle while the FO stage of the 11th instruction was active. Note that we are assuming all other operands were present in registers (which is why there are no other conflicts like this).

Another resource conflict is when multiple instructions are ready for the execution stage, but there is only one ALU. We could solve these conflicts by having more ports into main memory and multiple ALU units.

#### Data Hazards

Data hazards occur when there is a conflict in the access of an operand location. Essentially, two instructions in a program must use the same memory location’s value. If the two instructions are sequentially executed, there is no problem. However, if they are put in a pipeline, there is the chance that the first instruction changed the value of the memory location after the second instruction had already loaded the data from it but had not processed it yet. This would give us different results than a strictly sequential execution.

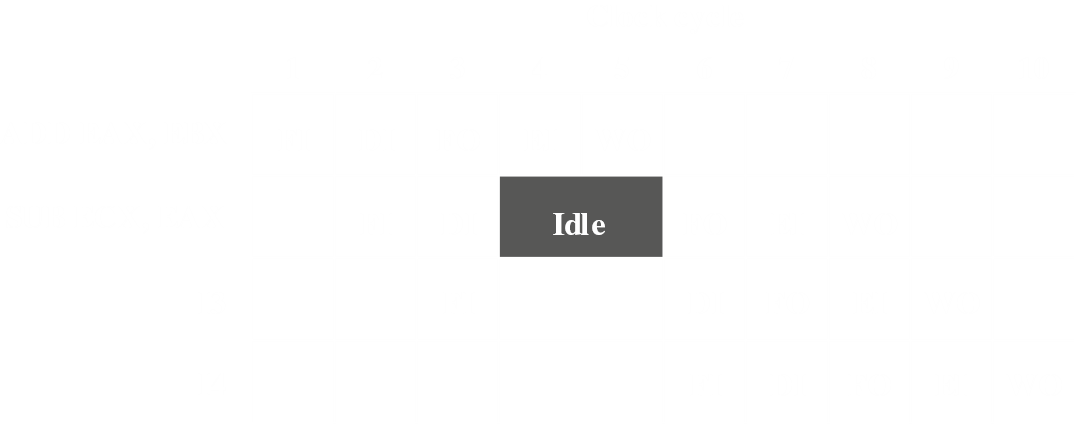
Consider these two instructions:

ADD EAX, EBX /\* EAX = EAX + EBX

SUB ECX, EAX /\* ECX = ECX – EAX

The first instruction adds the contents of two registers and puts the result in the first register. The second instruction subtracts this result from a third register and stores it in the third register.

In a pipeline, the instructions would look like this:



Notice how the second instruction had to stop after the second stage, just before it was going to fetch the operands. It had to wait for the first instruction, which was dealing the operands, to finish executing before it could fetch that operand. In the absence of special hardware and algorithms designed to avoid these situations, data hazards results in inefficiencies like this.

There are three types of data hazards:

* **Read After Write (RAW), or True Dependency**: There are two instructions, one that modifies a register and another after that that reads the data. The hazard occurs if the read is done before the write has occurred.
* **Write After Read (WAR), or Anti-Dependency**: One instruction reads from a register and another after that writes to it. The hazard occurs if the write occurs before the read has been performed.
* **Write After Write (WAW), or Output Dependency**: Two instructions, both of whom write to the same location, but the writes take place in the wrong order.

The example we saw was a RAW hazard.

#### Control Hazards

Control hazard, or branch hazards, occur when the pipeline makes a mistake on a branch prediction, which results in the instruction brought it having to be discarded. The next section discusses this and how to deal with it.

### Dealing with Branches

Assuring a steady flow of instructions to the initial stages of the pipeline is a major problem faced during designing the pipeline. We know that branch instructions are particularly difficult to deal with, since it is impossible to tell if a branch is taken or not until the instruction is actually executed.

A few approaches to dealing with conditional branches are:

* Multiple Streams
* Pre-Fetch Branch Target
* Loop Buffer
* Branch Prediction
* Delayed Branch

#### Multiple Streams

A simple pipeline suffers a penalty for a branch instruction because it must choose one of the two instructions to pre-fetch and might pre-fetch the wrong one. A brute-force approach is to pre-fetch both instructions, making use of two streams. There are two problems with this:

* With multiple pipelines, there are delays for access to the registers and to memory.
* More branch instructions might enter the pipeline before the first one is resolved, which in turn would need additional streams.

Despite the drawbacks, this strategy can still improve performance and there are machines that use this strategy.

#### Pre-Fetch Branch Target

When a conditional branch is recognized, the target of the branch is pre-fetched along with the instruction following the branch, which would have been pre-fetched anyways.

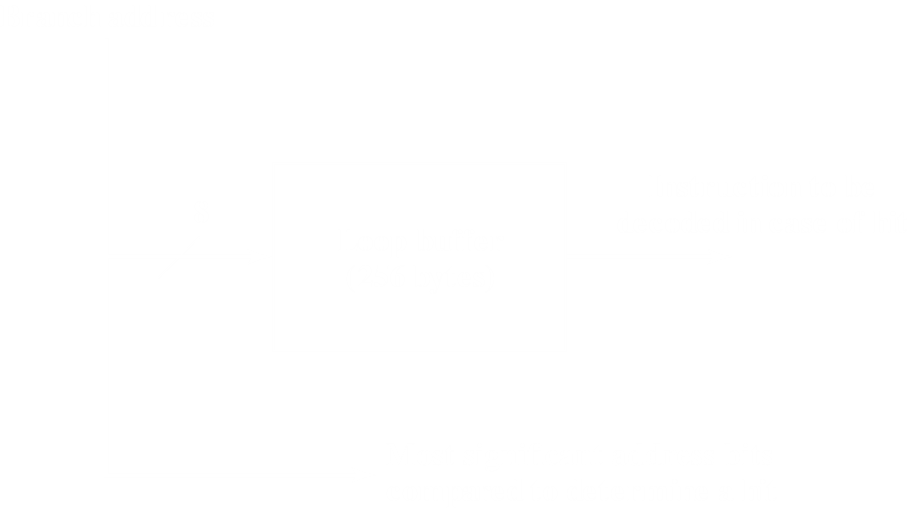
#### Loop Buffer

A loop buffer is a small, very-high-speed memory maintained by the instruction fetch stage of the pipeline. It contains the most recently fetched instructions in sequence. If a branch is to be taken, the loop buffer is checked for the branch target and fetched if it is available.

The loop buffer has three benefits:

* Some instructions that are sequentially ahead of the current instruction are available, which reduces instruction fetch time since memory access is not needed.
* If a branch occurs to a target just a few locations ahead, the target will be available in the loop. This is a common occurrence for if-then and if-then-else sequences.
* This strategy is particularly useful for loops. If the loop buffer is large enough to contain all the instructions in a loop, then they need only be fetched from memory for the first iteration. For subsequent iterations, all the required instructions will be in the buffer already.

The loop buffer is similar in principle to a cache dedicated to instructions. The differences are that it only retains instructions in sequence and it is much smaller and hence cheaper.



In the above diagram, the loop buffer is of 256 bytes and uses the least significant 8 bits to index the buffer. The remaining bits are checked for the branch target.

#### Branch Prediction

We can use a few techniques to predict whether a branch will be taken.

* Predict Never Taken
* Predict Always Taken
* Predict by Op-Code
* Taken/Not Taken Switch
* Branch History Table

The first three approaches are static and do not depend on the execution history up to the time of the conditional branch instructions. The last two are dynamic and do depend on the execution history.

The first two approaches are the simplest. They always assume either that the branch will not be taken and fetch the instructions in sequence, or that the branch will be taken and fetch the branch target.

The Predict Never Taken method is the most popular among all five methods. Studies show that branches are taken more than 50% of the time, so the Predict Always Taken method should be the best, given the cost of pre-fetching either path is the same. However, in a paged machine, pre-fetching the branch target is more likely to cause a page fault than pre-fetching the next instruction in sequence, so we need to take this performance penalty into account. An avoidance mechanism could be used to avoid the penalty.

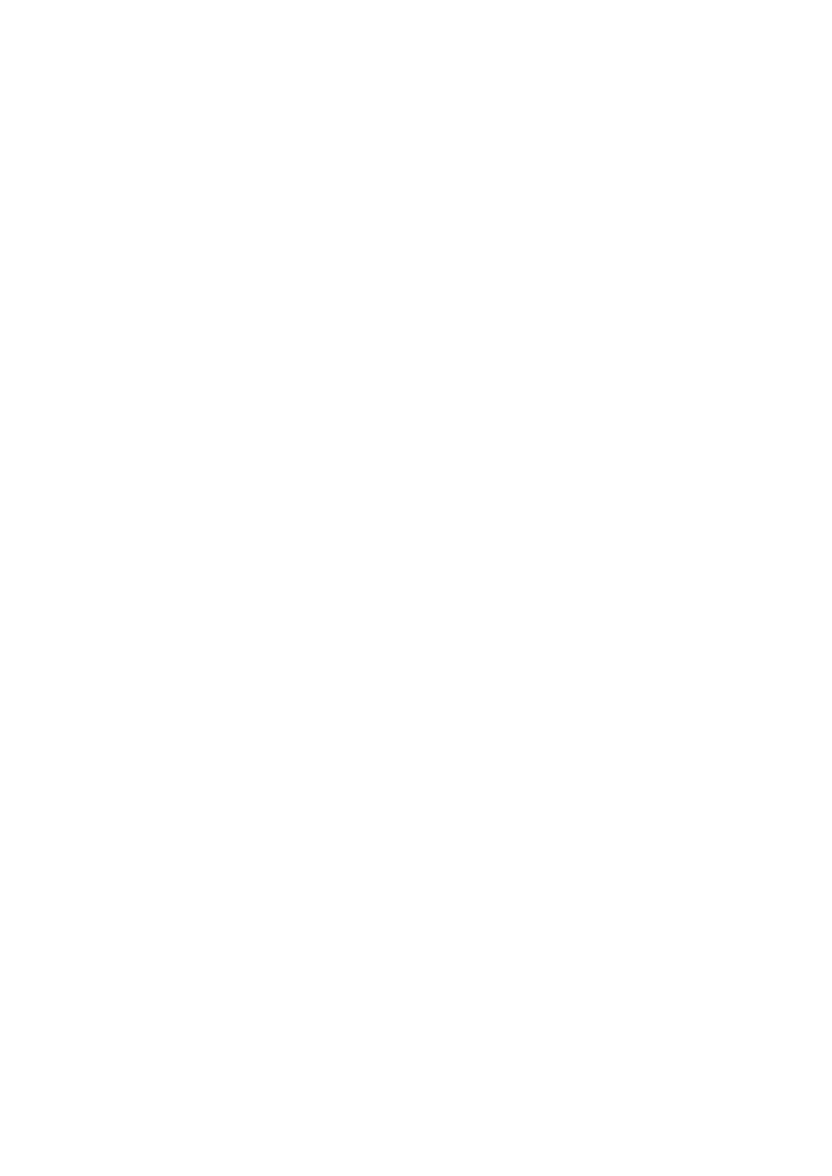
The final static approach makes the decision based on the op-code of the branch instruction. For certain op-codes, the processor assumes the branch is taken, while for others, it assumes the branch is not taken. This strategy is reported to have a success rate greater than 75%.

Dynamic branch strategies try to improve the accuracy of prediction by recording the history of conditional branch instructions in a program. For example, one or more bits can be associated with each conditional branch instruction that reflect the recent history of the instruction. These bits are referred to as a Taken/Not Taken Switch and direct the processor to make a decision the next time the instruction is encountered.

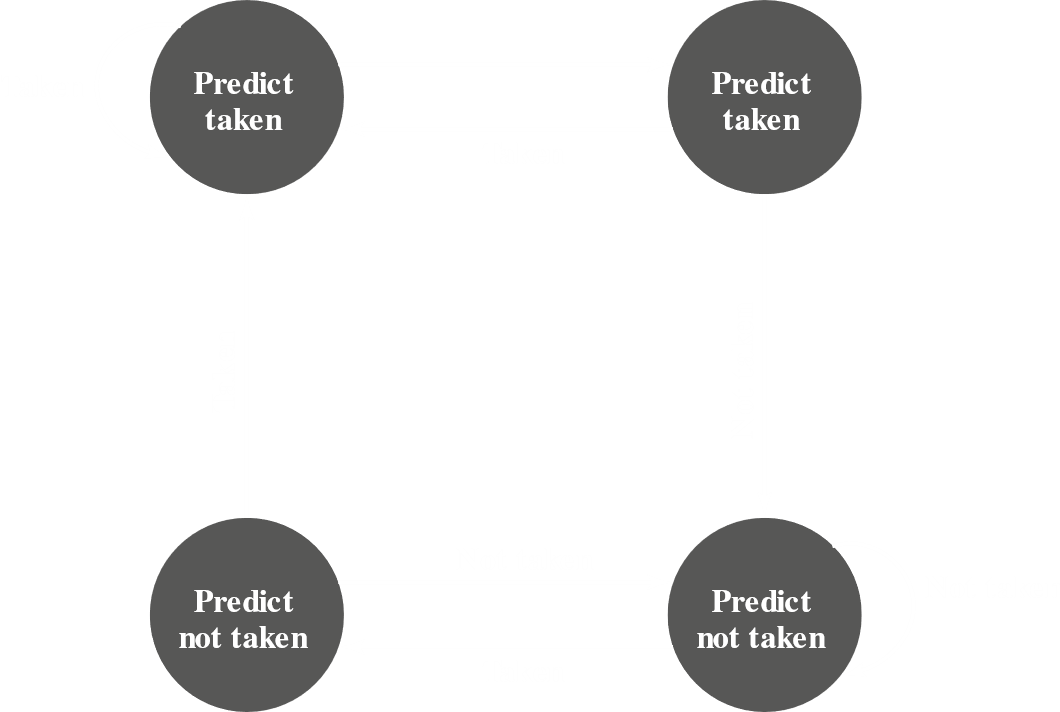
Typically, these history bits are not associated with the instruction in main memory but kept in a temporary high-speed storage. One possibility is to associate these bits with any conditional branch instruction in cache. When the instruction is replaced in cache, its history is lost. Another possibility is to keep a small table for the recently executed branch instructions with one or more history bits in each entry. The processor could access the table associatively, like a cache, or by using the low-order bits of the branch instruction’s address.

With a single bit, we can only record if the last execution of the instruction took the branch or not. With a single bit, we face a shortcoming for a conditional branch that is almost always taken, such as a loop instruction. With just one bit, we will get an error in prediction twice. Once when entering the loop, and once when exiting.

With two bits, we can record the result of the last two executions of the instruction, or we could record a state in some other fashion. The figure below shows a typical approach.

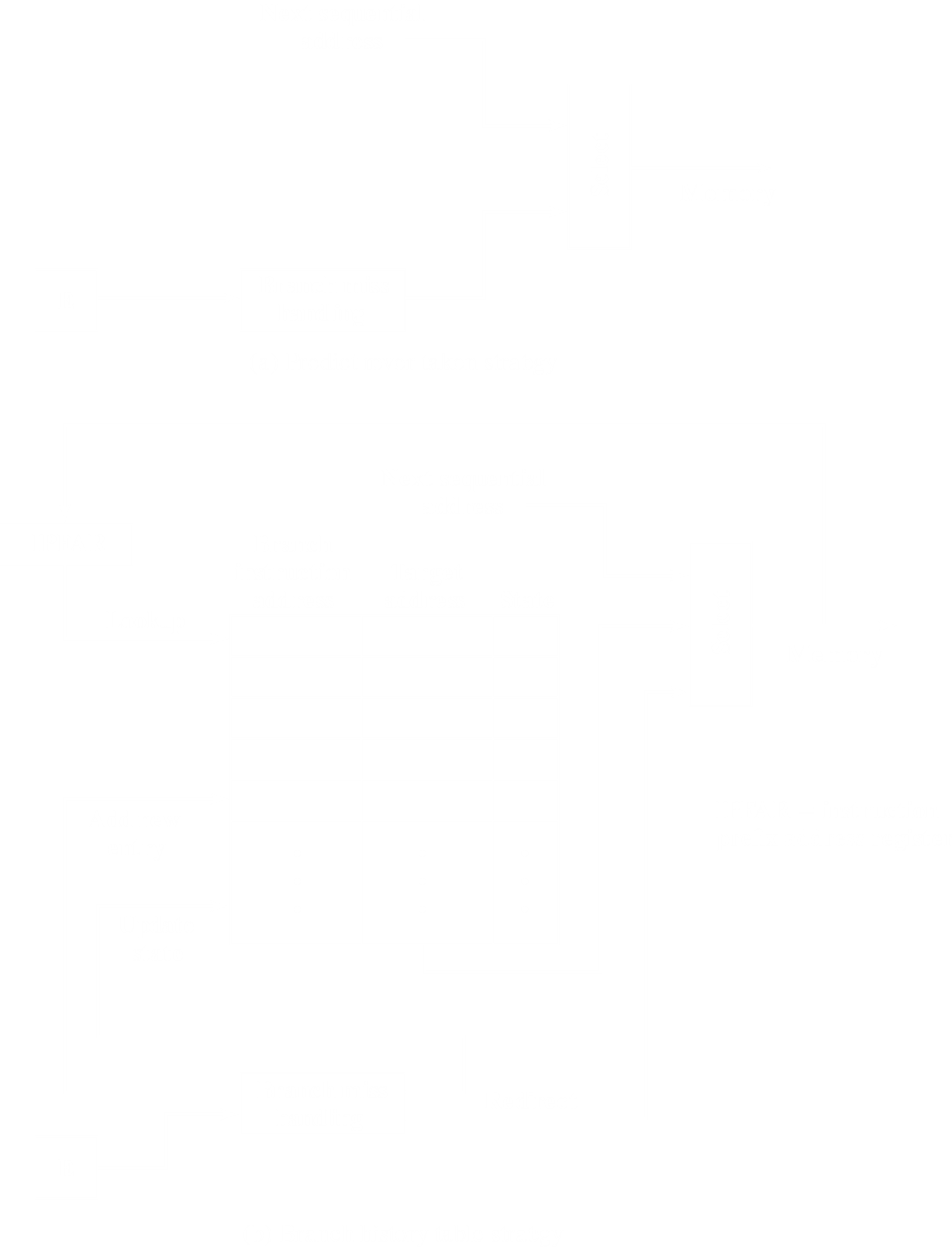


Starting at the top-left, only if we encounter two consecutive instances where the branch is not taken do we shift to the right side of the diagram and vice versa. We can describe the same thing using a state diagram.



Using history bits like this has one drawback. If the decision is taken to take a branch, the target cannot be fetched until the target address, which is an operand in the conditional branch instruction, is decoded. If the instruction fetch could be made as soon as the branch decision was taken, the process would be more efficient. We can do exactly this, by saving more information, using a branch target buffer, or branch history table.

The branch history table is a small cache memory associated with the instruction fetch stage. Each entry in the table has three elements, the address of the branch instruction, its history bits, and information about the target instruction. In most implementations, the third field is the address of the target instruction. Alternatively, it could contain the actual target instruction. The obvious trade-off is smaller table versus greater instruction fetch time.



The figure above compares the Branch History Table strategy with the Predict Never Taken strategy. With the latter method, the next sequential address is always fetched. If a branch is taken, the processor detects this and instructs that the next instruction be fetched from the target address, and the pipeline be flushed. The branch history table on the other hand, is treated like a cache. Each pre-fetch first searches the table. If it cannot find a match, the next sequential address is fetched. If a match is found, a prediction is made based on the state of the instruction. Either the next sequential address or the branch target address is used.

When a branch instruction is executed, the execute stage signals the branch history table with the result, so the state of the instruction can be updated. If the prediction was incorrect, the correct address can be used for the next fetch. When a conditional branch instruction is found that is not in the table, it is added and one of the existing entries is discarded using one of the cache replacement algorithms discussed in Chapter 4.

A refinement of the branch history table is two-level or correlation-based branch history. This approach assumes that, whereas in loop-closing branches, the past history is a good predictor of future behaviour, with complex control-flow structures, the direction of a branch is related to the direction of related branches. An if-then-else or a case structure is a good example. The global branch history, which has the history of the most recent branches, is used along with the current branch’s history.

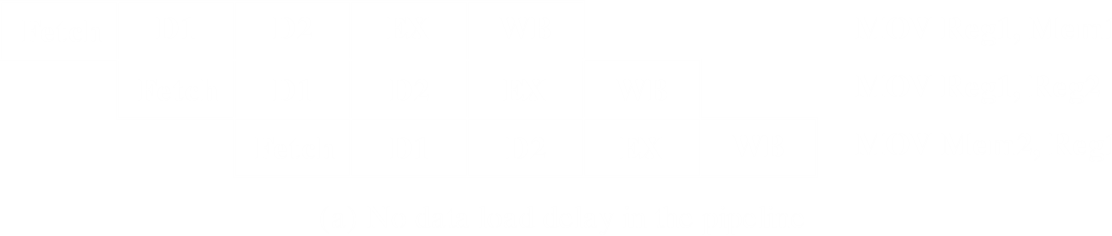
Delayed Branch causes the instructions to be rearranged so that the branch instructions occur later than actually desired. This means one or more instructions after the branch instruction that are not related to it are executed first, so as to avoid stalling the pipeline while the branch is being examined.

### Intel 80486 Pipelining

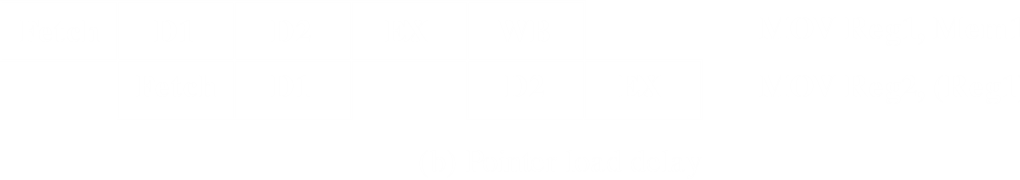
We shall now look at a real-life example of pipelining. The 80486 uses a five-stage pipeline:

* **Fetch**: Instructions are fetched from the cache or external memory and placed in one of the two 16-byte pre-fetch buffers. Since instructions vary in length, the status of the pre-fetch is different from other stages. On average, 5 instructions are fetched for every 16-byte load.
* **Decode Stage 1**: All op-code and addressing-mode information is decoded. The required information and the instruction length information is stored in the instruction, using at most the first 3 bytes. Thus, only 3 bytes are passed to the D1 stage from the pre-fetch buffers. The D1 decoder then tells the D2 stage to capture the rest of the instruction.
* **Decode Stage 2**: The D2 stage gets displace and immediate data from the instruction. It expands each op-code into control signals for the ALU and controls the computation of more complex addressing modes.
* **Execute**: This stage includes ALU operations, cache access and register updates.
* **Write Back**: This stage, if needed, updates registers and status flags modified by the execute stage. If the current instruction updates memory, the computer value is sent to the cache and to the bus-interface write buffers simultaneously.

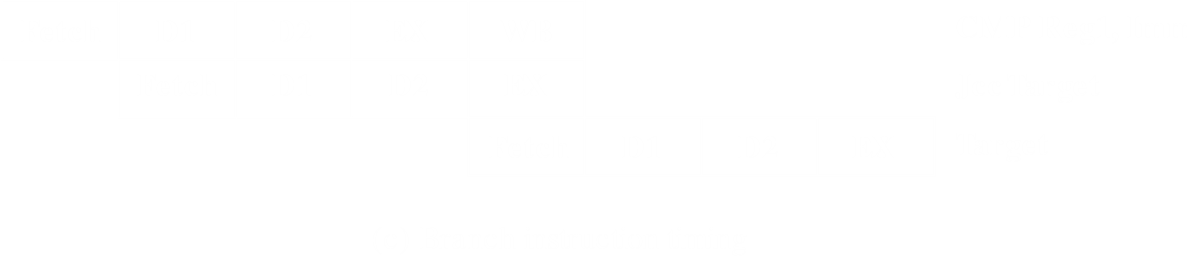
Using two decode stages, the pipeline can maintain a throughput of nearly one instruction per clock cycle. Complex instructions and conditional branches slow down this rate.



In the figure above, we see the performance for instructions with no memory access.



In this figure, a value is loaded from memory into a register and that register is then used as a base register in the next instruction. This causes a delay, as seen by the D2 stage of the second instruction having to wait for the execute stage of the first instruction to finish, since the D2 stage needs access to the register. A bypass signal allows the D2 stage of the second instruction to have access to the register while the WB stage of the first instruction is still using it, thus saving one clock cycle.



In this figure, we see what happens with a branch instruction. The compare instruction updates condition codes in the WB stage, and bypass paths make this available to the EX stage of the second instruction at the same time. In parallel, the processor runs a speculative fetch cycle to the target of the jump in the EX stage of the second instruction. A false branch condition causes this pre-fetch to be discarded and the next sequential instruction, which is already fetched and decoded, to be executed.