Chapter 14 - Processor Structure and Function

Luis Tarrataca
luis.tarrataca@gmail.com

CEFET-RJ

Table of Contents I

Table of Contents I

Table of Contents II

Processor Organization

Remember this?

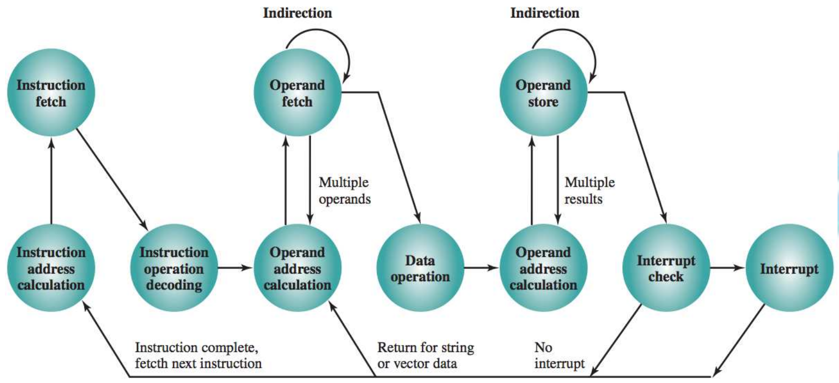

Indirection
Indirection
Instruction fetch
Operand fetch
Operand store
Multiple operands
Multiple results
Instruction address calculation
Instruction operation decoding
Operand address calculation
Data operation
Operand address calculation
Interrupt check
Interrupt
Instruction complete, fetcth next instruction
Return for string or vector data
No interrupt

Figure: Instruction cycle state diagram (Source: (Stallings, 2015))

Requirements placed on the processor:

- **Fetch instruction:** reads an instruction from memory;
- **Interpret instruction:** determines what action to perform;
- **Fetch data:** if necessary read data from memory or an I/O module.
- **Process data:** If necessary perform arithmetic / logical operation on data.
- **Write data:** If necessary write data to memory or an I/O module.

To do these things the processor needs to:

- store some data temporarily
- remember the location of the next instruction;

while an instruction is being executed:

- In other words, the processor needs a small internal memory.

Guess what this memory is called? Any idea?

To do these things the processor needs to:

- store some data temporarily
- remember the location of the next instruction;

while an instruction is being executed:

- In other words, the processor needs a small internal memory.

Guess what this memory is called? Any idea?

- Registers =)

We also need other components:

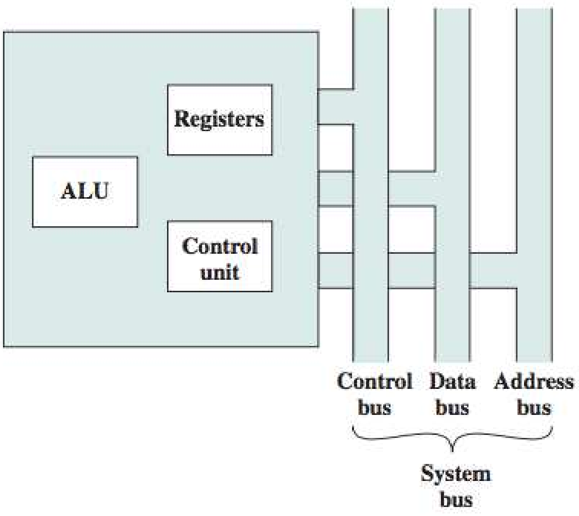


Figure: CPU with the system BUS (Source: (Stallings, 2015))

We will talk about it in Chapter 19, but:

What do you think the control unit does? Any ideas?

Major components of the processor:

- **Arithmetic and Logic Unit (ALU):**
  - Performs computation or processing of data
- **Control Unit:**
  - Moves data and instructions in and out of the processor;
  - Also controls the operation of the ALU;
- **Registers:**
  - Used as internal memory;
- **System Bus:**
  - Acting as a pathway between processor, memory and I/O module(s);

A more detailed view:

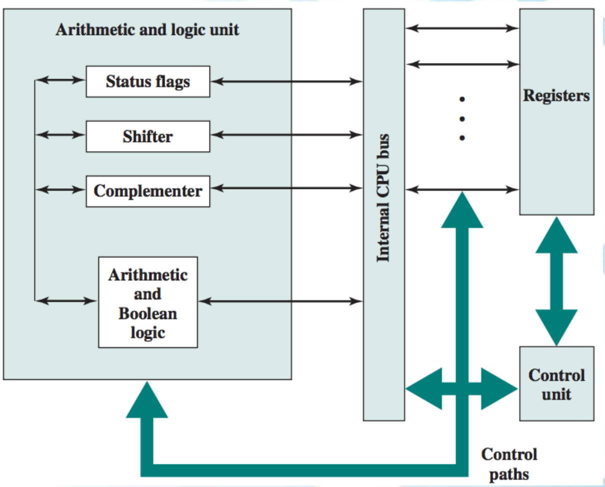


Figure: Internal structure of the CPU(Source: (Stallings, 2015))

Besides the usual elements the previous figure also includes:

- Internal CPU bus:
    - Needed to transfer data between the various registers and the ALU;
- Logic control paths;
    - Needed to specify which operations to perform;

Register Organization

Registers in the processor perform two roles:

- **User-visible registers:**
    - Used as internal memory by the assembly language programmer;
- **Control and status registers:**
    - Used to control the operation of the processor;
    - Used to check the status of the processor / ALU;

Lets have a look at each one of these =)

User-visible Registers

May be referenced by the programmer, categorized into:

- General purpose
- Data
- Address
- Condition codes

What do you think each one of these does? Any ideas?

General-purpose registers

Can be assigned to a variety of functions by the programmer:

- Memory reference & backup;
- Register reference & backup;
- Data reference & backup;
- These are the ones you use in the laboratory =)

Data registers

May be used only to hold data and cannot hold addresses:

- Must be able to hold values of most data types;
- Some machines allow two contiguous registers to be used:
  - For holding double-length values.

Address Registers

Used to hold addresses, *e.g.:*

- Stack Pointer
- Program Counter
- Index Registers

Must be at least long enough to hold the largest address.

Condition Codes

Hold condition codes (a.k.a. **flags**):

- Flags are bits set by processor as the result of operations
- *E.g.:* an arithmetic operation may produce:
    - a positive result;
    - a negative result;
    - a zero result;
    - an overflow result.

Condition code bits are collected into one or more control registers:

In some machines:

- Interruption results in all user-visible registers being saved;
- These are then restored on return;
- Allows each subroutine to use the user-visible registers independently;

On other machines:

- responsibility of the programmer to:
    - save the contents of user- visible registers prior to a subroutine call;

Regarding the previous slide:

What are the advantages / disadvantages of using one method instead of the other? Any ideas?

Guess what is the method used by the P3 simulator? =P

Control and Status Registers

Employed to control the operation of the processor:

- Mostly not visible to the user;

Do you know any registers of this type? Any ideas?

Control and Status Registers

Do you know any registers of this type? Any ideas?

- **Program counter (PC):** Contains instruction address to be fetched;
- **Instruction Register (IR):** Contains last instruction fetched;
- **Memory address register (MAR):** Contains memory location address;
- **Memory buffer register (MBR):** Contains:
  - a word of data to be written to memory;
  - a word of data read from memory.

In general terms:

- Processor updates **PC** after each instruction fetch;
- A branch or skip instruction will also modify the contents of the **PC**;
- The fetched instruction is loaded into an **IR**
- Data are exchanged with memory using the **MAR** and **MBR**, *e.g.:*
  - MAR connects directly to the address bus
  - MBR connects directly to the data bus

The four registers just mentioned are used for:

- Data movement between processor and memory;
- Within the processor, data must be presented to the ALU for processing:
    - ALU may have direct access to the MBR and user-visible registers;
    - Alternatively:
        - There may be additional buffering registers within ALU;
        - These registers serve as input and output registers for the ALU;
        - These registers exchange data with the MBR and user-visible registers.

Many processors include a program status word **(PSW)** register:

- Contains condition codes plus other status information
- Common fields or flags include the following:
  - Sign: Sign bit of the result of the last arithmetic operation;
  - Zero: when the result is 0;
  - Carry: Set if an operation resulted in a carry/borrow bit;
  - Equal: Set if a logical compare result is equality.
  - Overflow: Used to indicate arithmetic overflow.
  - Interrupt Enable/Disable: Used to enable or disable interrupts.

Instruction Cycle

Lets go back to this:

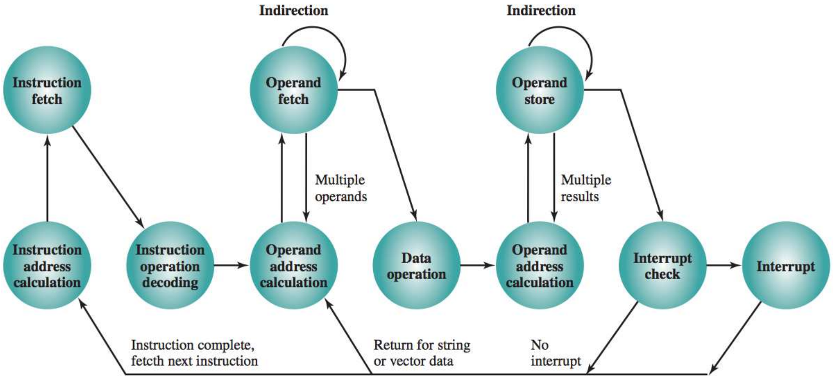

Indirection
Indirection
Instruction fetch
Operand fetch
Operand store
Multiple operands
Multiple results
Instruction address calculation
Instruction operation decoding
Operand address calculation
Data operation
Operand address calculation
Interrupt check
Interrupt
Instruction complete, fetch next instruction
Return for string or vector data
No interrupt

Figure: Instruction cycle state diagram (Source: (Stallings, 2015))

Now that we know more about the inner workings of:

- CPU;
- Registers;
- Bus

What is the information flow during the **fetch** cycle?

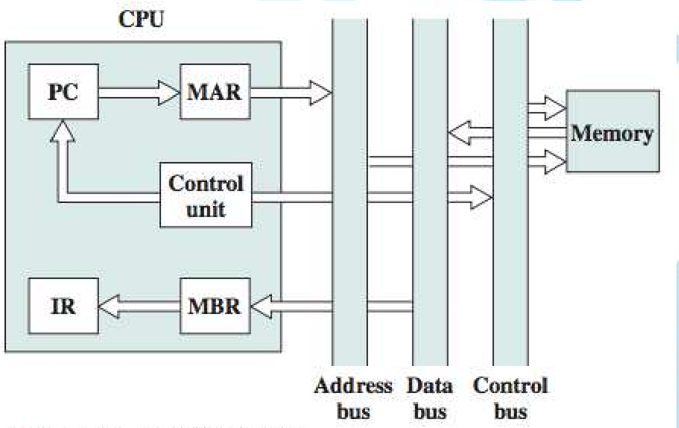
What is the information flow during the **execute** cycle?

What is the information flow during the **interruption** cycle?

Lets start with the first question:

What is the information flow during the **fetch** cycle? Any ideas?

The flow of data during the instruction **fetch** cycle:



MBR = Memory buffer register
MAR = Memory address register
IR = Instruction register
PC = Program counter

Figure: Data Flow: FetchCycle(Source: (Stallings, 2015))

The flow of data during the instruction **fetch** cycle:

1. **PC** contains the address of the next instruction to be fetched;
2. Address is moved to the **MAR** and placed on the address bus;
3. Control unit requests a memory read;
4. Result is:
    - placed on the data bus;
    - copied into the **MBR**;
    - then moved to the **IR**.
5. Meanwhile, the **PC** is incremented by 1;

Once the **fetch cycle** is over, control unit examines **IR**:

1. to determine if it contains an operand specifier using **indirect addressing**;
2. If so, an indirect cycle is performed:

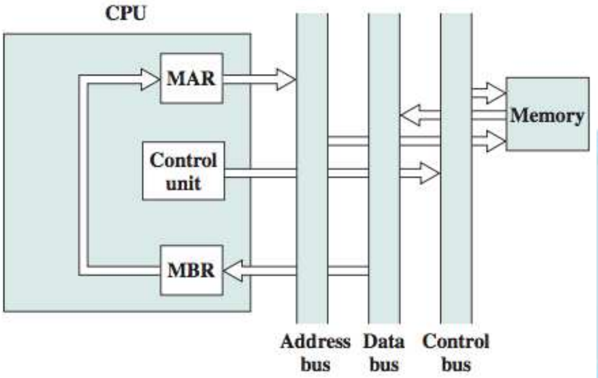


Figure: Data flow: indirect cycle (Source: (Stallings, 2015))

The indirect addressing cycle:

- Bits of the **MBR** containing the address are transferred to the **MAR**;
- Control unit then requests a memory read:
    - to get the desired address of the operand into the **MBR**.

What is the information flow during the **execute** cycle? Any ideas?

The **execute cycle** takes many forms:

- Depending on the operation to be performed...
- May involve:
  - transferring data among registers
  - read or write from memory or I/O
  - and/or the invocation of the ALU.

What is the information flow during the **interruption** cycle? Any ideas?

Then comes the **interrupt cycle:**

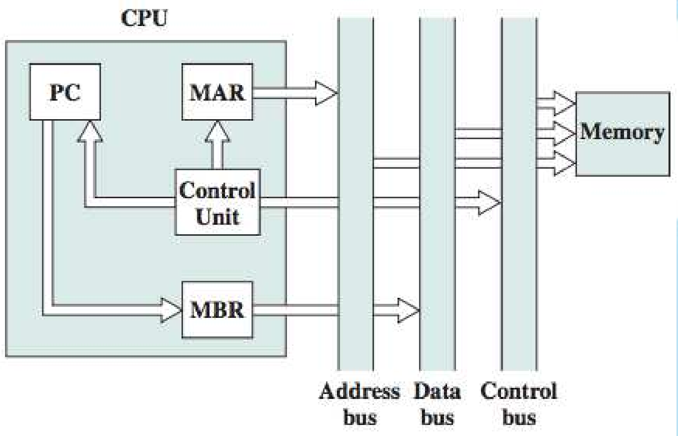


Figure: DataFlow: InterruptCycle (Source: (Stallings, 2015))

The **interruption cycle:**

- Contents of the **PC** must be saved;
- Thus the contents of the **PC** are
    - Transferred to the **MBR** to be written into memory.
    - Special memory location is loaded into the **MAR**:
        - *E.g.:* stack pointer **(SP)**
- **PC** is loaded with the address of the interrupt routine.

Now that we have seen how:

- Processor organization and function relate to the instruction cycle:

Lets have a look at how to improve performance

- Always a fun topic =)

Now that we have seen how:

- Processor organization and function relate to the instruction cycle:

Lets have a look at how to improve performance

- Always a fun topic =)

What are some of the techniques to increase processor performance?

Some examples:

- Increase Frequency (Hz). Why?

Some examples:

- Increase Frequency (Hz). Why?
  - Faster number of clock ticks per unit of time.

Some examples:

- Increase Frequency (Hz). Why?
    - Faster number of clock ticks per unit of time.
- Cache-levels. Why?

Some examples:

- Increase Frequency (Hz). Why?
  - Faster number of clock ticks per unit of time.
- Cache-levels. Why?
  - Reduce number of read / writes from high latency memory.

Some examples:

- Increase Frequency (Hz). Why?
    - Faster number of clock ticks per unit of time.
- Cache-levels. Why?
    - Reduce number of read / writes from high latency memory.
- Multi-core architecture. Why?

Some examples:

- Increase Frequency (Hz). Why?
    - Faster number of clock ticks per unit of time.
- Cache-levels. Why?
    - Reduce number of read / writes from high latency memory.
- Multi-core architecture. Why?
    - Parallelize instruction set;

Some examples:

- Increase Frequency (Hz). Why?
    - Faster number of clock ticks per unit of time.
- Cache-levels. Why?
    - Reduce number of read / writes from high latency memory.
- Multi-core architecture. Why?
    - Parallelize instruction set;
- Physical size of the processor. Why?

Some examples:

- Increase Frequency (Hz). Why?
    - Faster number of clock ticks per unit of time.
- Cache-levels. Why?
    - Reduce number of read / writes from high latency memory.
- Multi-core architecture. Why?
    - Parallelize instruction set;
- Physical size of the processor. Why?
    - Electrical signals travel shorter distances;

Some examples:

- Increase Frequency (Hz). Why?
    - Faster number of clock ticks per unit of time.
- Cache-levels. Why?
    - Reduce number of read / writes from high latency memory.
- Multi-core architecture. Why?
    - Parallelize instruction set;
- Physical size of the processor. Why?
    - Electrical signals travel shorter distances;
    - Transistor switch time decreases.

Instruction Pipelining

Rest of the presentation focus will be on **pipeline** strategies!

- Another method to improve performance =)
- Relates directly to:
  - Instruction cycle;
  - Processor organization / function;

First, what is a pipeline? Any ideas?

Pipelining

Similar to an **assembly line** in a manufacturing plant.

- Product goes through various stages of production;
- Products at various stages can be worked on simultaneously;

Figure: (Source: Wikipedia)

Equivalent concept in computation: **pipelining**

- New inputs are accepted at one end...
- ...before previously accepted inputs appear as outputs at the other end.

In practice, what does this mean? Any ideas?

Instruction Stages

How can we apply the concept of pipelining to computer instructions?

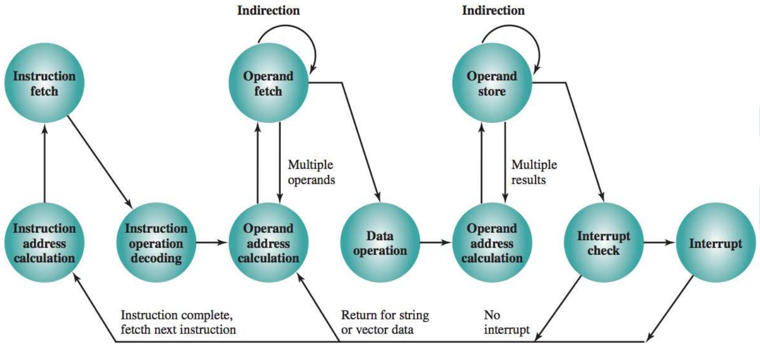

Indirection
Indirection
Instruction fetch
Operand fetch
Operand store
Multiple operands
Multiple results
Instruction address calculation
Instruction operation decoding
Operand address calculation
Data operation
Operand address calculation
Interrupt check
Interrupt
Instruction complete, fetcth next instruction
Return for string or vector data
No interrupt

Figure: Instruction Cycle State Diagram (Source: (Stallings, 2015))

Several Pipelining opportunities exist. Any ideas?

- After **fetching** data from memory for one instruction:

Several Pipelining opportunities exist. Any ideas?

- After **fetching** data from memory for one instruction:
  - Begin fetching memory data for another instruction;

Several Pipelining opportunities exist. Any ideas?

- After **fetching** data from memory for one instruction:
  - Begin fetching memory data for another instruction;
- After **writing** data from memory for one instruction:

Several Pipelining opportunities exist. Any ideas?

- After **fetching** data from memory for one instruction:
  - Begin fetching memory data for another instruction;
- After **writing** data from memory for one instruction:
  - Begin fetching memory data for another instruction;

Several Pipelining opportunities exist. Any ideas?

- After **fetching** data from memory for one instruction:
  - Begin fetching memory data for another instruction;
- After **writing** data from memory for one instruction:
  - Begin fetching memory data for another instruction;
- After **executing** operators from memory for one instruction:

Several Pipelining opportunities exist. Any ideas?

- After **fetching** data from memory for one instruction:
  - Begin fetching memory data for another instruction;
- After **writing** data from memory for one instruction:
  - Begin fetching memory data for another instruction;
- After **executing** operators from memory for one instruction:
  - Begin executing operators for another instruction

Several Pipelining opportunities exist. Any ideas?

- After **fetching** data from memory for one instruction:
  - Begin fetching memory data for another instruction;
- After **writing** data from memory for one instruction:
  - Begin fetching memory data for another instruction;
- After **executing** operators from memory for one instruction:
  - Begin executing operators for another instruction
- and so on...

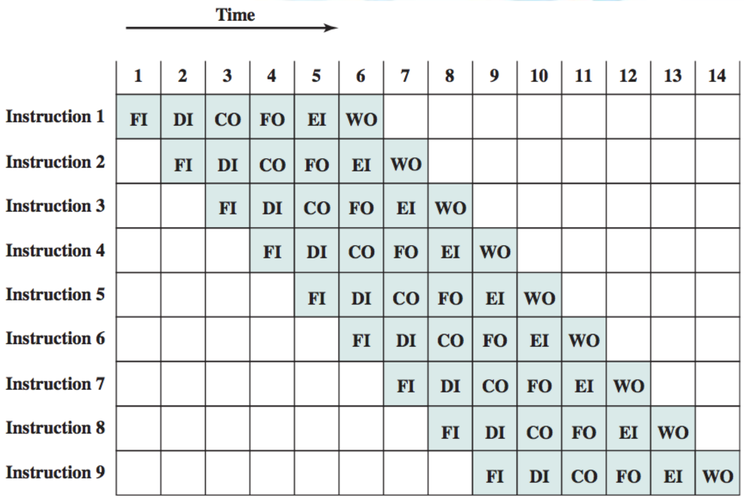
Consider the following instruction stages:

- **Fetch instruction (FI):** Read the next instruction into a buffer;
- **Decode instruction (DI):** Determine the opcode;
- **Calculate operands (CO):** Calculate the address of each operand.
- **Fetch operands (FO):** Fetch each operand from memory;
- **Execute instruction (EI):** Perform the indicated operation;
- **Write operand (WO):** Store the result in memory.

With this decomposition:

- Various stages will be of nearly equal duration.
- Rest of the slides assume equal duration.

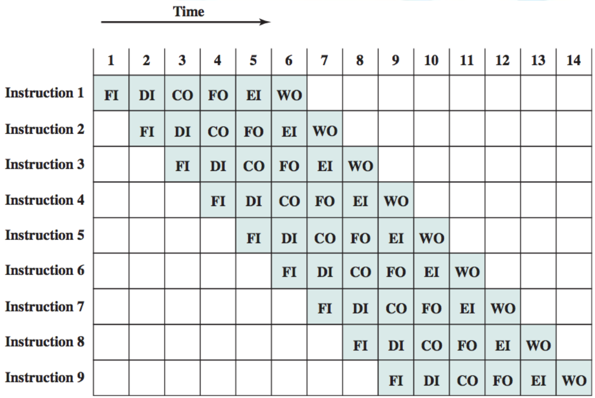
Equal duration assumption allows for the following pipeline:



	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO	EI	WO					
Instruction 5					FI	DI	CO	FO	EI	WO				
Instruction 6						FI	DI	CO	FO	EI	WO			
Instruction 7							FI	DI	CO	FO	EI	WO		
Instruction 8								FI	DI	CO	FO	EI	WO	
Instruction 9									FI	DI	CO	FO	EI	WO

Figure: Timing Diagram for a 6-stage instruction Pipeline Operation (Source: (Stallings, 2015))

What is the total number of time units required **without** the pipeline?

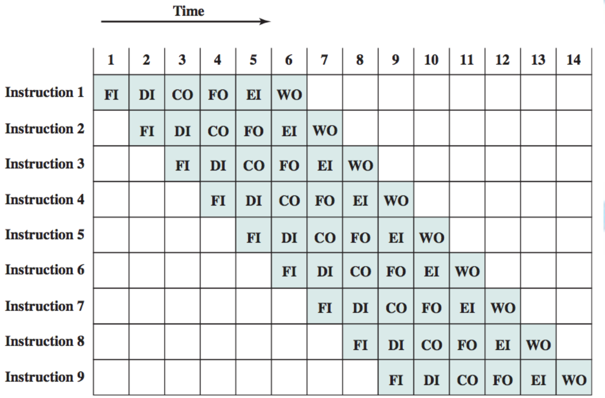


Time

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO	EI	WO					
Instruction 5					FI	DI	CO	FO	EI	WO				
Instruction 6						FI	DI	CO	FO	EI	WO			
Instruction 7							FI	DI	CO	FO	EI	WO		
Instruction 8								FI	DI	CO	FO	EI	WO	
Instruction 9									FI	DI	CO	FO	EI	WO

Figure: Timing Diagram for a 6-stage instruction Pipeline Operation (Source: (Stallings, 2015))

What is the total number of time units required **without** the pipeline?

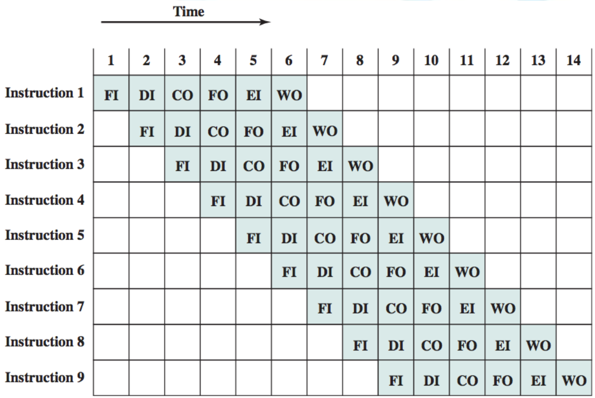


	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO	EI	WO					
Instruction 5					FI	DI	CO	FO	EI	WO				
Instruction 6						FI	DI	CO	FO	EI	WO			
Instruction 7							FI	DI	CO	FO	EI	WO		
Instruction 8								FI	DI	CO	FO	EI	WO	
Instruction 9									FI	DI	CO	FO	EI	WO

Figure: Timing Diagram for a 6-stage instruction Pipeline Operation (Source: (Stallings, 2015))

- 9 instructions, each with 6 time units implies 54 time units;

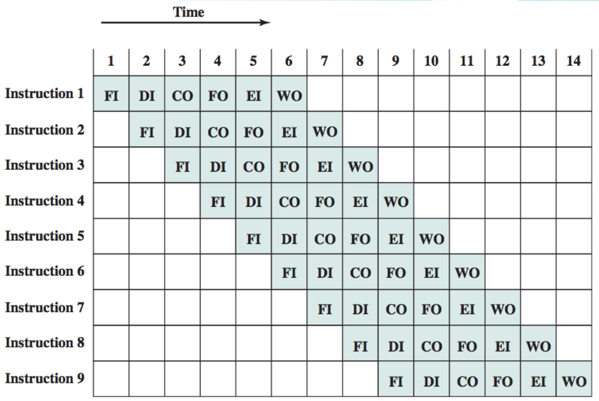
What is the total number of time units required **with** the pipeline?



	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO	EI	WO					
Instruction 5					FI	DI	CO	FO	EI	WO				
Instruction 6						FI	DI	CO	FO	EI	WO			
Instruction 7							FI	DI	CO	FO	EI	WO		
Instruction 8								FI	DI	CO	FO	EI	WO	
Instruction 9									FI	DI	CO	FO	EI	WO

Figure: Timing Diagram for a 6-stage instruction Pipeline Operation (Source: (Stallings, 2015))

What is the total number of time units required **with** the pipeline?


Time

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO	EI	WO					
Instruction 5					FI	DI	CO	FO	EI	WO				
Instruction 6						FI	DI	CO	FO	EI	WO			
Instruction 7							FI	DI	CO	FO	EI	WO		
Instruction 8								FI	DI	CO	FO	EI	WO	
Instruction 9									FI	DI	CO	FO	EI	WO

Figure: Timing Diagram for a 6-stage instruction Pipeline Operation (Source: (Stallings, 2015))

- 14 time units;

What was the total number of time units required before the pipeline?

- 9 instructions, each with 6 time units implies 54 time units;

What is the total number of time units required after the pipeline?

- 9^{th} instruction will start executing at time unit 9 and will last 6 time units;
- Total time: $9 + 6 - 1 = 14$
- Performance ratio: $54/14 = 4.15$ faster

Previous example yielded a 4.15 performance speedup:

Do we always obtain such a meaningful speedup? Any ideas?

Previous example yielded a 4.15 performance speedup:

Do we always obtain such a meaningful speedup? Any ideas?

- No... =´(
- Lets see why...

Simplifications

Pipeline example assumed several things (1/3):

- Each stage lasts an equal amount of time:

Simplifications

Pipeline example assumed several things (1/3):

- Each stage lasts an equal amount of time:
  - **Simplification:** not so in practice;

Simplifications

Pipeline example assumed several things (1/3):

- Each stage lasts an equal amount of time:
  - **Simplification:** not so in practice;
- Each operation always goes through the six stages:

Simplifications

Pipeline example assumed several things (1/3):

- Each stage lasts an equal amount of time:
  - **Simplification:** not so in practice;
- Each operation always goes through the six stages:
  - **Simplification:** not so in practice;

Pipeline example assumed several things (2/3):

- No memory-bus conflicts. What does this mean?

Pipeline example assumed several things (2/3):

- No memory-bus conflicts. What does this mean?
  - FI, FO and WO stage involve a memory access;

Pipeline example assumed several things (2/3):

- No memory-bus conflicts. What does this mean?
    - FI, FO and WO stage involve a memory access;
    - Diagram implies that all these accesses can occur simultaneously;

Pipeline example assumed several things (2/3):

- No memory-bus conflicts. What does this mean?
    - FI, FO and WO stage involve a memory access;
    - Diagram implies that all these accesses can occur simultaneously;
    - Most memory systems will not permit that.

Pipeline example assumed several things (2/3):

- No memory-bus conflicts. What does this mean?
  - FI, FO and WO stage involve a memory access;
  - Diagram implies that all these accesses can occur simultaneously;
  - Most memory systems will not permit that.
  - However it may still be possible to do. How?

Pipeline example assumed several things (2/3):

- No memory-bus conflicts. What does this mean?
  - FI, FO and WO stage involve a memory access;
  - Diagram implies that all these accesses can occur simultaneously;
  - Most memory systems will not permit that.
  - However it may still be possible to do. How?
    - Desired value may be in cache;

Pipeline example assumed several things (2/3):

- No memory-bus conflicts. What does this mean?
  - FI, FO and WO stage involve a memory access;
  - Diagram implies that all these accesses can occur simultaneously;
  - Most memory systems will not permit that.
  - However it may still be possible to do. How?
    - Desired value may be in cache;
    - FO and WO stages may not be performed (NULL values).

Pipeline example assumed several things (3/3):

- No memory-data conflicts. What does this mean?

Pipeline example assumed several things (3/3):

- No memory-data conflicts. What does this mean?
    - Several instructions can act on the same region of memory;

Pipeline example assumed several things (3/3):

- No memory-data conflicts. What does this mean?
  - Several instructions can act on the same region of memory;
  - Up to the compiler and OS to detect and avoid these cases.

Pipeline example assumed several things (3/3):

- No memory-data conflicts. What does this mean?
    - Several instructions can act on the same region of memory;
    - Up to the compiler and OS to detect and avoid these cases.
- Most of the time memory conflicts will not slow down the pipeline.

Possible pipeline disruptions

Lets have a look at events that may **disrupt** the pipeline:

Can you see any type of events that can disrupt the pipeline?

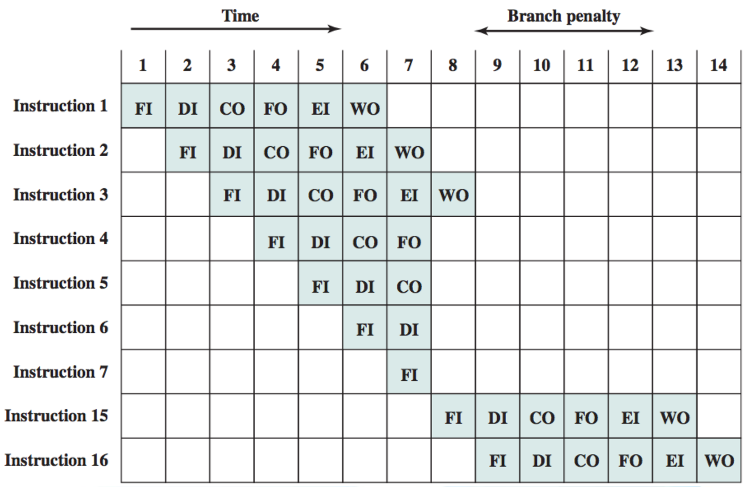
Any ideas?

What happens if we have a conditional instruction and jump to another instruction?

What happens if we have a conditional instruction and jump to another instruction?

- Conditional branch instruction can invalidate several instruction fetches!

- **Conditional branch instruction** can invalidate several instruction fetches!


Time
Branch penalty

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO							
Instruction 5					FI	DI	CO							
Instruction 6						FI	DI							
Instruction 7							FI							
Instruction 15								FI	DI	CO	FO	EI	WO	
Instruction 16									FI	DI	CO	FO	EI	WO

Figure: Effect of a Conditional Branch on Instruction Pipeline Operation. Instruction 3 is a conditional branch to instruction 15 (Source: (Stallings, 2015))

- There is no way for the pipeline to determine the conditional branch:
  - Pipeline continues to load the next instructions as if no branching will occur;
  - If no jump happens then we get the full benefit of the pipeline;
  - Otherwise, we need to reload the pipeline with the subsequent instructions.

Can you think of any other mechanism that can disrupt the pipeline?

Can you think of any other mechanism that can disrupt the pipeline?

- The need to process hardware interruptions.

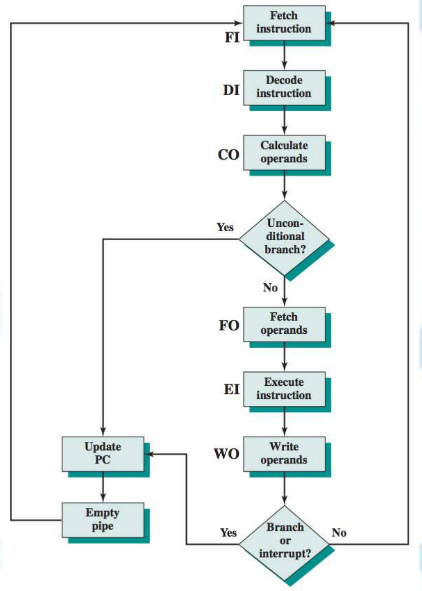


Figure: Six-Stage CPU Instruction Pipeline (Source: (Stallings, 2015))

Design Issues

Instruction pipelining is a powerful technique for enhancing performance

- Requires careful design for optimum results with reasonable complexity.
- Elements to consider:
    - Overhead of guaranteeing that the pipeline functions properly:
        - moving data from buffer to buffer;
        - preparing the system to transition to the next stage(s);
    - Control logic required increases enormously with the number of stages;

Pipeline Performance

Lets consider other issues:

How much time is required to move a set of instructions one stage through the pipeline? Any ideas?

Pipeline Performance

Cycle time τ of an instruction pipeline is the time needed to:

- Advance a set of instructions one stage through the pipeline:

$$\tau = \max_i[\tau_i] + d = \tau_m + d, \quad 1 \leq i \leq k$$

where:

- τ_i = time delay of the circuitry in the i^{th} stage of the pipeline
- τ_m = maximum stage delay;
- k = number of stages in the instruction pipeline;
- d = delay needed to advance signals/data from one stage to the next.

Pipeline Performance

Now that we know what the cycle time τ is:

How much time is required to execute n instructions in a pipeline with k stages? Any ideas?

Let $T_{k,n}$ be:

- Time required for a pipeline with k stages to execute n instructions

$$T_{k,n} = [k + (n - 1)]\tau$$

Explanation:

- k cycles stages to complete the execution of the first instruction;
- Remaining $n - 1$ instructions require $n - 1$ cycle stages.
- All these cycle stage take time τ

Now that we know how much time is required for a pipeline:

How does a system with several stages compares with one without a pipeline? Any ideas?

Now that we know how much time is required for a pipeline:

How does a system with several stages compares with one without a pipeline? Any ideas?

What is the instruction cycle time for a non-pipeline processor? Any ideas?

What is the instruction cycle time for a non-pipeline processor?

Assume for the **non-pipelined processor** the following:

- Instruction cycle time is $k \times t$:
  - Each instruction contains k stages of t time;
- Let T_{wp} be the time without pipeline:
  - For n instructions the total time is $n \times k \times t$
  - $T_{wp} = nkt$

Comparison of T_{wp} and $T_{k,n}$:

- $T_{k,n}$
- T_{wp}
- Speedup
  $S_k = \frac{T_{wp}}{T_{k,n}} = \frac{nk}{k+(n-1)}$

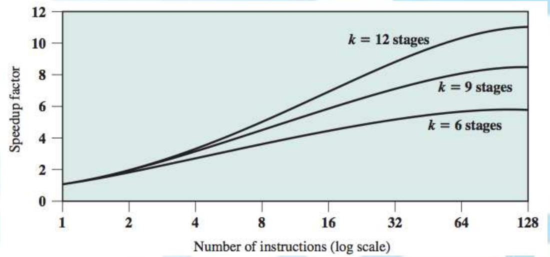


Figure: Number of instructions (log scale). (Source: (Stallings, 2015))

What are the main conclusions that you can draw from this comparison?

What are the main conclusions that you can draw from this comparison?

- Increase number of pipeline stages: increase speedup potential
- However:
    - Speedup increases costs;
    - Delays between stages increases;
    - Stages to flush also increase in case of a disruption;

Pipeline Hazards

Besides disruption pipelines are also susceptible to hazards:

What are pipeline hazards? Any ideas?

Pipeline Hazards

What are pipeline hazards?

- Hazards do not permit continued pipeline execution;

Pipeline Hazards

Occur when conditions do not permit continued pipeline execution (1/3)

- **Resource Hazards:**

What are resource hazards? Any ideas

Pipeline Hazards

Occur when conditions do not permit continued pipeline execution (1/3)

- **Resource Hazards:**

  What are resource hazards? Any ideas

  - when two (or more) instructions in the pipeline need the same resource;

Pipeline Hazards

Occur when conditions do not permit continued pipeline execution (1/3)

- **Resource Hazards:**

What are resource hazards? Any ideas

  - when two (or more) instructions in the pipeline need the same resource;
  - Resource examples: bus, memory, cache, etc...

Pipeline Hazards

Occur when conditions do not permit continued pipeline execution (2/3)

- **Data Hazards:**

What are data hazards? Any ideas

Pipeline Hazards

Occur when conditions do not permit continued pipeline execution (2/3)

- **Data Hazards:**

What are data hazards? Any ideas

  - when there is a conflict in the access of an operand location;

Pipeline Hazards

Occur when conditions do not permit continued pipeline execution (3/3)

- **Control Hazards:**

What are control hazards? Any ideas

Pipeline Hazards

Occur when conditions do not permit continued pipeline execution (3/3)

- **Control Hazards:**

> What are control hazards? Any ideas

- when a wrong decision is made on a branch prediction;

Pipeline Hazards

Occur when conditions do not permit continued pipeline execution (3/3)

- **Control Hazards:**

  What are control hazards? Any ideas

  - when a wrong decision is made on a branch prediction;
  - Instructions must be discarded.

Pipeline Hazards

Occur when conditions do not permit continued pipeline execution, *e.g.:*

- **Resource Hazards:**
  - when two (or more) instructions in the pipeline need the same resource;
  - Resource examples: bus, memory, cache, etc...
- **Data Hazards:**
  - when there is a conflict in the access of an operand location;
- **Control Hazards:**
  - when a wrong decision is made on a branch prediction;
  - Instructions must be discarded.

Resource Hazards

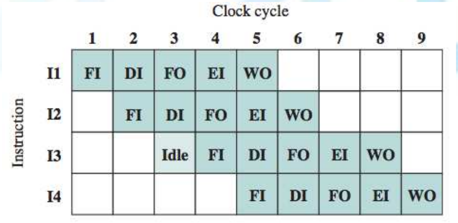
When two (or more) instructions in the pipeline need the same resource:

What can be done to solve this issue? Any ideas?

Resource Hazards

When two (or more) instructions in the pipeline need the same resource:

- Instructions must be executed in serial for a portion of the pipeline.
- Example: **IF**s, **FO**s and **WO**s must be performed one at a time.



Clock cycle

Instruction	1	2	3	4	5	6	7	8	9
I1	FI	DI	FO	EI	WO				
I2		FI	DI	FO	EI	WO			
I3			Idle	FI	DI	FO	EI	WO	
I4					FI	DI	FO	EI	WO

Figure: Example of Resource Hazard. (Source: (Stallings, 2015))

- We need to **idle** the instruction causing the hazard.

Data Hazards

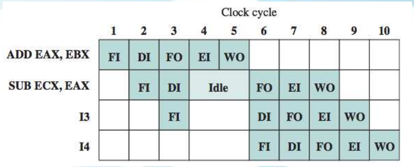
Occurs when there is a conflict in the access of an operand location:

- Two sequential instructions access the same memory / register:
  - No pipeline $\longrightarrow$ no problem:
    - If the two instructions are executed in strict sequence, no problem occurs.
  - Pipeline $\longrightarrow$ maybe a problem:
    - Depending on the way the operand is updated;
    - Lets have a look at this update problem;

Data Hazards

```
ADD  EAX,    EBX  /*  EAX  =  EAX  +  EBX
SUB  ECX,    EAX  /*  ECX  =  ECX  −  EAX
```

Figure: Example of data hazard. (Source: (Stallings, 2015))



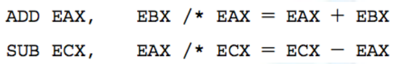
	Clock cycle 1	2	3	4	5	6	7	8	9	10
ADD EAX, EBX	FI	DI	FO	EI	WO					
SUB ECX, EAX		FI	DI	Idle		FO	EI	WO		
I3			FI			DI	FO	EI	WO	
I4						FI	DI	FO	EI	WO

Figure: The corresponding pipeline for the data hazard. (Source: (Stallings, 2015))

- ADD instruction does not update register EAX until the end of stage 5;
- SUB instruction needs that value at **FO** (stage 3, clock cycle 4);

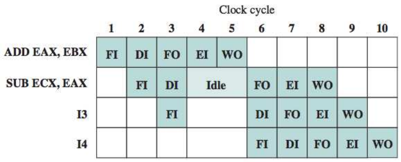
What can be done to solve this problem? Any ideas?

Data Hazards



```
ADD  EAX,    EBX /* EAX = EAX + EBX
SUB  ECX,    EAX /* ECX = ECX − EAX
```

Figure: Example of data hazard. (Source: (Stallings, 2015))



	Clock cycle 1	2	3	4	5	6	7	8	9	10
ADD EAX, EBX	FI	DI	FO	EI	WO					
SUB ECX, EAX		FI	DI	Idle		FO	EI	WO		
I3			FI			DI	FO	EI	WO	
I4						FI	DI	FO	EI	WO

Figure: The corresponding pipeline for the data hazard. (Source: (Stallings, 2015))

- ADD instruction does not update register EAX until the end of stage 5;
- SUB instruction needs that value at **FO** (stage 3, clock cycle 4);
- **Solution:** Pipeline must **idle** for two clocks cycles.

Now that we know more about data hazards:

Are all data hazards equal? Any ideas?

Now that we know more about data hazards:

Are all data hazards equal? Any ideas?

There are three types of data hazards:

- **Read after write (RAW)**
- **Write after read (WAR)**
- **Write after write (WAW)**

What do you think each one of these means? Any ideas?

Read after write (RAW)

1. Instruction modifies a register or memory location
2. Succeeding instruction reads the data in that memory or register location.
3. Hazard occurs if:
    - Data read takes place before the write operation is complete.

Write after read (WAR)

1. Instruction reads a register or memory location;
2. Succeeding instruction writes to the location;
3. Hazard occurs if:
    - Write operation completes before the read operation is complete;

Write after write (WAW)

1. Two instructions both write to the same location;
2. Hazard occurs if:
   - Write operations take place in the reverse order of the intended sequence.

Control Hazards

Control hazards, a.k.a. a branch hazard, occur when:

- Pipeline makes wrong decision on a branch prediction:
  - Instructions must be discarded...
  - Wasted work...

So what can we do to mitigate control hazards? Any ideas?

So what can we do to mitigate control hazards? Any ideas?

Myriad of strategies exist *e.g.:*

- Multiple streams;
- Prefetch branch target;
- Loop buffer;
- Branch prediction;
- Delayed branch;

Lets have a look at these =)

Multiple streams

- Makes use of multiple pipelines:
  - One pipeline loads the jump sequence;
  - Another pipeline loads the non-jump sequence;
- Brute-force approach:
  - Additional branch instructions may enter each pipeline;
  - Search space grows exponentially quick =(
- Despite these drawbacks: can still improve performance.

Prefetch Branch Target

- When a conditional branch is recognized:
  - Target is prefetched, in addition to the instruction following the branch.
    - If the branch is taken: target has already been prefetched;
    - Otherwise: instruction following the branch was also fetched.

Loop Buffer

- As always in hardware: when performance is an issue use a cache ;)
- High-speed memory maintained by the **IF** stage of the pipeline
    - Containing the *n* most recently fetched instructions;
- If a branch is to be taken:
    - Pipeline hardware checks if target is in cache;
        - If so: next instruction is fetched from the buffer;
        - No need to fetch instruction from main memory;
- Well suited to dealing with loops, or iterations. Why?:
    - Temporal and spatial locality is ideally suited for cache systems.

Branch Prediction

Several possible techniques (1/3):

- **Predict never taken** - assume that the branch will not be taken and continue to fetch instructions in sequence.
  - $p(\text{working}) < 50\%$ (Source: (Lilja, 1988))
- **Predict always taken** - assume that the branch will be taken and always fetch from the branch target.
  - $p(\text{working}) > 50\%$ (Source: (Lilja, 1988))

Several possible techniques (2/3):

- **Predict by opcode** - Some opcodes are more likely than others to lead to branch targets.
  - Processor assumes that:
    - Branch will be taken for certain opcodes;
    - Branch will not be taken for other opcodes;
  - $p(\text{working}) > 75\%$ (Source: (Lilja, 1988))

Several possible techniques (3/3):

- **Branch Taken / not taken switch**
  - Idea: use a single bit to reflect that last thing that happened (JMP or $\neg$JMP);
  - Very limiting...
- **Branch history table.**
  - Idea: counters for each branching instruction:
    - Real time decision based on history;
    - If $\geq 50\%$ time branch jumps then load branch targets;
    - Otherwise continue sequentially.

Overclocking

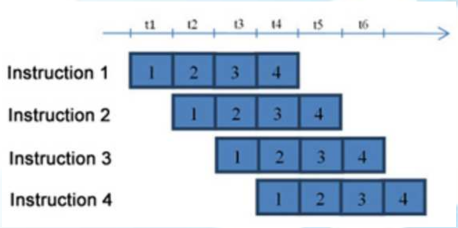
Lets discuss an additional performance technique:

- Always a fun topic =)

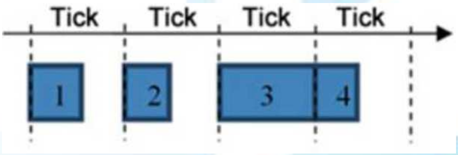
In reality we are going to discuss two topics:

- Overclocking;
- Reasons why CPU frequency ceased to grow.

- Lets look again at the instruction pipeline



- This figure is a little bit deceptive, a better representation is:



- Some stages of the pipeline execute faster than the clock speed;
- Limited by the stage that lasts the longest time, *i,e.:* τ,
- There is also a safety margin in terms of clock speed and the longest stage:
    - To deal with operating conditions outside of a manufacturer's control:
        - Ambient temperature;
        - Fluctuations in operating voltage;
        - and some others...

Overclock idea:

- Abdicate of this safety margin and increase the clock speed
- Time for each stage will diminish and we will have a faster processor.

Will we be able to diminish this time a lot? Any ideas?

Overclock idea (1/2):

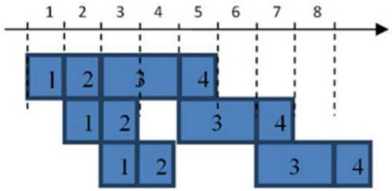
- Abdicate of this safety margin and increase the clock speed
- Time for each stage will diminish and we will have a faster processor.

Will we be able to diminish this time a lot? Any ideas?

Not always... Why? Any ideas?

Overclock idea (2/2):

Not always... Why? Any ideas?



- In the figure above the clock speed is too fast for stage 3...
- Increase frequency too much:
    - Some pipeline stages will not have time to end...
    - Performance will stop improving...

Besides pipeline problems can you think of any other problems with overclocking?

Besides pipeline problems can you think of any other problems with overclocking?

- Increasing the frequency is done by increasing the voltage to the system.
  - $f \propto V$
  - $P \propto V^3$
    - Changes in voltage lead to a cubic increase of power!!!!
    - And what is the physical manifestation of power? Heat dissipation.

- Elevated heat degrades the useful-life of the silicon used in chips;
- If heat is not properly managed:
  - Only a matter of time until the processor goes kaputz;
  - Even with built-in sensors...:
    - Video 1
    - Video 2
- Thus the need for good refrigeration systems.

- Heat: reason why CPU frequency has stopped growing;
- Yet we still have had performance gains. Why?
  - Parallel computing FTW =)
- One final question?

Why do logical processors emit heat?

- This is the same reason why the CPU frequency has stopped growing some years ago;
- Yet we still have had performance gains. Why?
  - Parallel computing FTW =)
- One final question?

Why do logical processors emit heat?

- Landauer's principle - irreversible computation leads to heat dissipation as a direct logical consequence of the underlying reversibility of physics!

Where to focus your study

After this class you should be able to:

- Explain what a pipeline is and the underlying mechanics.
- Understand how pipelines can improve performance.
- Understand the issues influencing pipeline performance;
- Understand how to tackle these issues.

Less important to know how these solutions were implemented:

- details of specific pipelines from the x86 processor family.

Your focus should always be on the building blocks for developing a solution =)

References I

Lilja, D. J. (1988).

Reducing the branch penalty in pipelined processors.

Computer, 21(7):47--55.

Stallings, W. (2015).

Computer Organization and Architecture.

Pearson Education.