THE pipelining technique

Advanced Computer Architectures

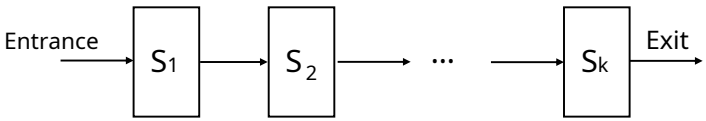
Valeria Cardellini

Pipelining: the idea

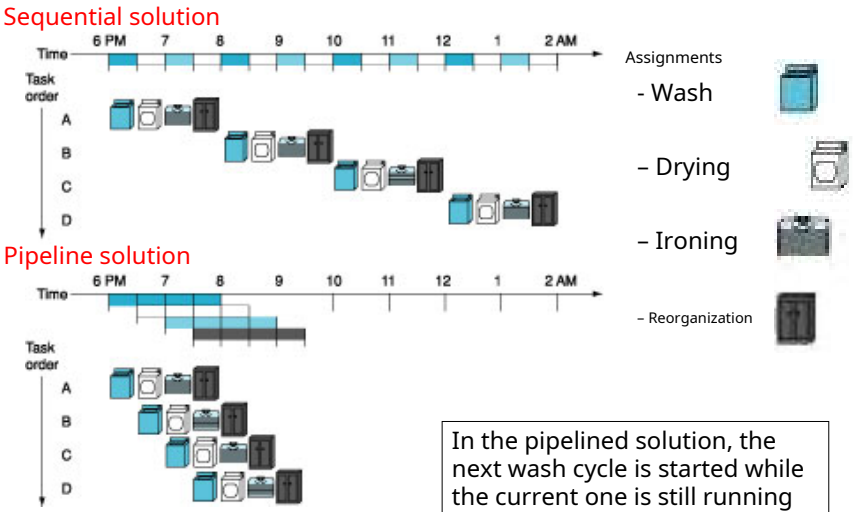
- It's a technique
  - to improve processor performance
  - based on overlap of the execution of more instructions belonging to a sequential execution flow
- Analogy with the assembly line

The pipeline: the idea

- The work done by a pipelined processor to execute an instruction is divided into steps (*pipeline stages*), which require a fraction of the time needed to execute the entire instruction
- The stages are connected in a serial manner to form the pipeline; the instructions:
  - enter from one end of the pipeline
  - are processed by the various stages according to the expected order
  - come out the other end of the pipeline



Laundry example



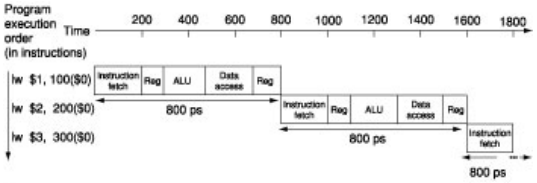
In the pipelined solution, the next wash cycle is started while the current one is still running

Comparison between single cycle and pipelines

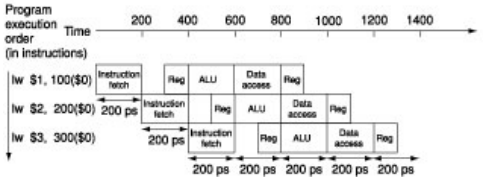
• Example of execution times of the different instruction classes

Instruction	IF	ID	EX	MEM	WB	Total
lw	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
sw	200 ps	100 ps	200 ps	200 ps		700 ps
R format	200 ps	100 ps	200 ps		100 ps	600 ps
well	200 ps	100 ps	200 ps			500 ps

Single cycle



Pipelines



To a comment up pe in pg

• The presence of the pipeline increases the number of instructions*at the same time*running

• So, by introducing pipelining into the processor, increases the***throughput***...

– Throughput: number of instructions executed per unit of time

• ... but*it is not reduced****latency***of the individual instruction

– Latency: execution time of the single instruction, from its start until its completion

– An instruction that takes 5 steps, continues to require 5 clock cycles for its execution with pipelining

Sad de ap pe pe

• The time required to advance an instruction one stage through the pipeline corresponds to one pipeline clock cycle

• Since the pipeline stages are connected sequentially, they must operate synchronously

– Clock-synchronized pipeline progress

– Clock cycle length of the pipelined processor determined by the length of the slowest stage of the pipeline

• Ex.: 200 ps for the slowest operation execution

– For some instructions, some stages are wasted cycles

• Designers' objective: to balance the length of the stadiums

• If the stages are*perfectly balanced*, it*ideal speedup*due to pipelining is equal to the number of pipeline stages

$$\text{Ideal speedup}_{\text{pipeline}} = \frac{\text{time between instructions}_{\text{no pipelines}}}{\text{time between instructions}_{\text{pipeline}}} = \text{num. pipeline stages}$$

Sad de ap pe pe ?

• But, in general, the pipeline stages are not perfectly balanced

• The introduction of pipelining therefore involves additional costs

– The time interval for completing an instruction is greater than the minimum possible value

– Lo*real speedup*will be less than the number of pipeline stages introduced

• A 5-stage pipeline typically fails to quintuple performance

Multicycle implementation

- Example: sequence of 3 lw instructions (see slide 4)
- Ideal speedup of 5, but more modest improvement
  - 3 lw instructions without pipeline: 800×3 = 2400 ps
  - 3 pipelined lw instructions: 200×3 + 800 = 1400 ps
  - So 1400 ps instead of 2400 ps (2400/1400 = 1.71)
- Difference due to the time required to fill and empty the pipeline
  - It takes 4 stages (800 ps) to fill and empty the pipeline
- *In general*: starting from the empty pipeline with *k* stages, to complete *n* instructions are needed *k* + (*n*-1) clock cycles
  - *k* loops to fill the pipeline and complete the execution of the first instruction
  - *n*-1 cycles to complete the remaining ones *n*-1 instructions

Multicycle implementation (2)

- As the number of instructions increases *n*, the ratio of total execution times on non- and pipelined machines approaches the ideal limit
  - The time to fill the pipeline becomes negligible compared to the total time to complete the instructions
  - 1000 lw instructions without pipeline: 800×1000 = 800000 ps
  - 1000 pipelined lw instructions: 200×1000 + 800 = 200800 ps
  - 200800 ps instead of 800000 ps (800000/200800 = 3.98)

Multicycle implementation (3)

- In the asymptotic case ($n \to \infty$)
  - The latency of the single lw instruction it gets worse
    - Go from 800 ps (without pipelining) to 1000 ps (with pipelining)
  - The throughput improves 4 times
    - Goes from 1 lw instruction completed every 800 ps (without pipelining) to 1 lw instruction completed every 200 ps (with pipelining)
- If we consider a 1000 ps single cycle processor (composed of 5 stages each of 200 ps) and a pipelining processor (with 5 stages of 200 ps each) in the asymptotic case
  - The latency of the single instruction remains unchanged and equal to 1000 ps
  - The throughput improves by 5 times
    - Goes from 1 instruction completed every 1000 ps (without pipelining) to 1 instruction completed every 200 ps (with pipelining)

Instruction set design for MIPS pipelining

- The MIPS instruction set design allows for a simple and efficient pipeline
  - All instructions have the same length (32 bits)
    - Easier to load the instruction in the first step and decode the instruction in the second step
  - Few instruction formats (only 3) with symmetry between formats
    - It is possible to start reading the registers in the second step, before knowing what instruction (and format) it is
  - Memory operations are limited to load/store instructions
    - You can use the third step to calculate the address
  - Alignment of operands in memory
    - Only one stage can be used to transfer data between processor and memory
  - Each MIPS instruction writes at most one result and does so towards the end of the pipeline

AND - and the structure

IF Instruction Fetch	ID Instruction Decode	FORMER EXecute	MEM MEMory access	WB Write-Back

- Logical-arithmetic instructions

IF Prel. instruct. and incr. PC	ID Record reading source	FORMER Op. ALU on read data		WB Reg. writing dest.

- Load instructions

IF Prel. instruct. and incr. PC	ID Record reading basic	FORMER Sum ALU	MEM Withdrawal given by M	WB Reg. writing dest.

- Store instructions

IF Prel. instruct. and incr. PC	ID Record reading basic e source	FORMER Sum ALU	MEM Writing given in M	

- Beq instructions

IF Prel. is tr. and incr. PC	ID Record reading source	FORMER ALU subtraction e jump address	MEM PC writing	

LAnd to

- The *critical issues* (or *conflicts* or *alee*) arise in pipelining architectures when an instruction cannot be executed in the immediately following cycle

- Three types of critical issues
  - Structural critical issues
  - Data critical issues
  - Critical issues regarding control

LAnd to (2)

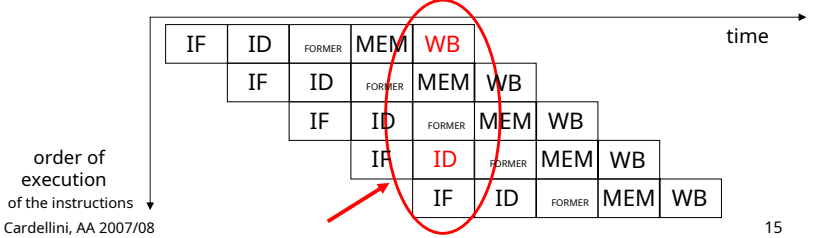
- Critical issues **structural**
  - Attempt to use the same hardware resource by different instructions in different ways in the same clock cycle
  - E.g.: if in MIPS we had a single instruction and data memory

- Critical issues **on the data**
  - Attempting to use a result before it is available
  - E.g.: instruction that depends on the result of a previous instruction that is still in the pipeline

- Critical issues **on control**
  - Attempting to make a decision about the next instruction to execute before the condition is evaluated
  - E.g.: conditional branch instructions: if you are executing beq, how do you know (in advance) which is the next instruction to start executing?

C

- In the single-cycle MIPS architecture we have no structural conflicts
- Data memory separated from instruction memory
- Register bank used in the same pipeline cycle by giving a read access by one instruction and a write access by another instruction
  - Solution to avoid type uncertainty *Read After Write*
    - Writing of the register desk in the first half of the clock cycle
    - Reading of the register desk in the second half of the clock cycle



Control hazards

- An instruction depends on the result of a previous instruction that is still in the pipeline
- Example 1:

```
add $s0, $t0, $t1
sub $t2,$s0, $t3
```

  - One of sub's source operands ($s0) is produced by add, which is still in the pipeline
  - Critical issues regarding type data *define-use*
- Example 2:

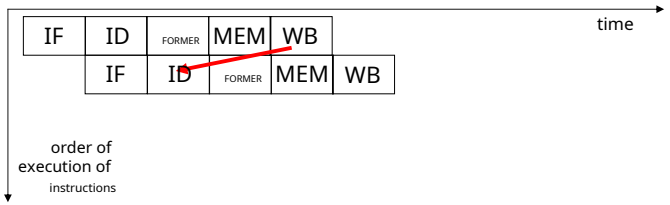
```
lw $s0, 20($t1)
sub $t2,$s0, $t3
```

  - One of sub's source operands ($s0) is produced by lw, which is still in the pipeline
  - Critical issues regarding type data *load-use*

Control hazards (2)

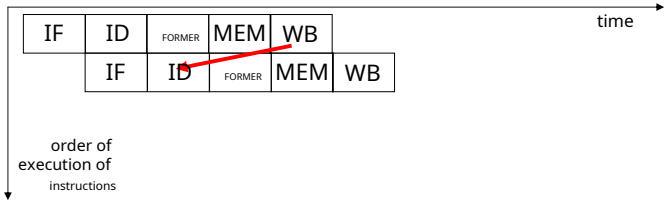
- Example 1:

```
add $ s0, $t0, $t1
sub $t2,$s0, $t3
```



- Example 2:

```
lw $s0, 20($t1)
sub $t2,$s0, $t3
```



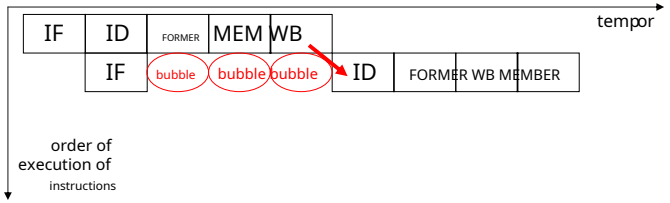
Solutions for control hazards

- Hardware solutions
  - Insertion of bubbles (*bubble*) or stalls in the pipeline
    - Dead times are inserted
    - Worses throughput
  - Propagation or overriding (*forwarding* or bypassing)
    - Propagate data forward as soon as it is available to the units that request it
- Software-type solutions
  - Insertion of nop (no operation) instructions
    - Worses throughput
  - Reorganization of instructions
    - Move "harmless" instructions so that they eliminate the criticality

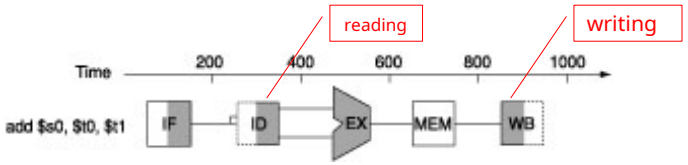
THE THE management of control hazards

- You insert bubbles into the pipeline, that is, you block the flow of instructions in the pipeline until the conflict is resolved
  - *Stall*: state the processor is in when instructions are blocked
- Example 1: must be inserted three bubbles to stop the sub statement so that the correct data can be read
  - Two bubbles if register bank optimization

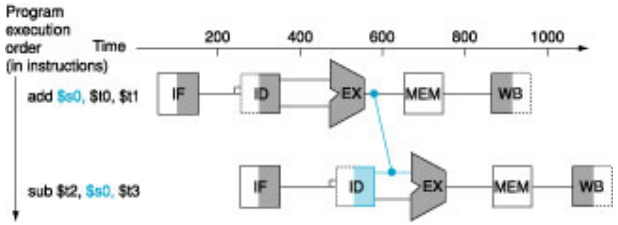
```
add $s0, $t0, $t1
sub $t2,$s0, $t3
```



Propagazione (forwarding)



- Example 1: when the ALU generates the result, it comes *right away* made available for the following instruction step via a *forward propagation*



Propagazione e LW

- Example 2:

  lw $s0, 20($t1)

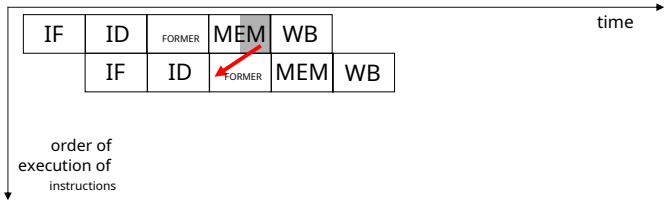
  sub $t2,$s0, $t3

  – It is a critical issue on type data *load-use*

  - The data loaded by the load instruction is not yet available when requested by a subsequent instruction

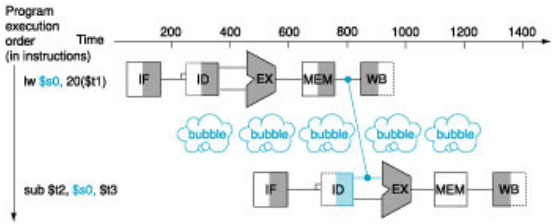
- Propagation alone is insufficient to resolve this type of critical issue

lw $s0, 20($t1)
sub $t2,$s0, $t3



Propagazione e LW (2)

- Possible solution: propagation and a stall

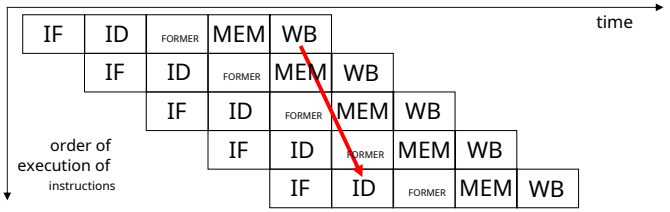


- Without register bank propagation and optimization, they would have been necessary three stalls

The nop instruction

- Example 1: the assembler must insert three nop instructions between the add and sub instructions, thus making the conflict disappear
  – The nop instruction is the software equivalent of stall

add $s0, $t0, $t1
nop
nop
nop
sub $t2,$s0, $t3



Reordering

- The assembler reorders instructions to prevent related instructions from being too close together
  - The assembler tries to insert instructions between related (conflicting) instructions *independent* from the result of the previous instructions
  - When the assembler cannot find independent instructions it must insert nop instructions
- Example:

```
lw $t1, 0($t0)                lw $t1, 0($t0)
lw $t2, 4($t0)     tidying up lw $t2, 4($t0)
add $t3,$t1,$t2               lw $t4, 8($t0)
sw $t3, 12($t0)               add $t3, $t1, $t2
lw $t4, 8($t0)                sw $t3, 12($t0)
add $t5, $t1,$t4              add $t5, $t1, $t4
sw $t5, 16($t0)               sw $t5, 16($t0)
```

critical issues

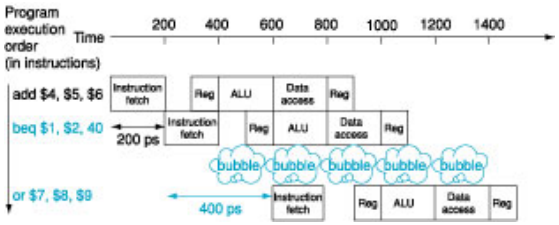
  - Propagation allows you to resolve remaining conflicts after reordering

Control hazard

- To feed the pipeline, an instruction must be inserted at every clock cycle
- However, in the MIPS processor the conditional branch decision is not made until the fourth step (MEM) of the beq instruction
- Desired jumping behavior
  - If the comparison fails, continue execution with the statement after beq
  - If the comparison is verified, do not execute the instructions after beq and jump to the specified address

Solutions to control hazard

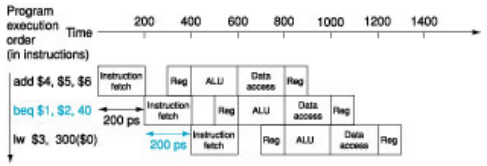
- Insertion of bubbles
  - The pipeline is blocked until the result of the beq comparison is known and we know which instruction to execute next
    - In MIPS the result of the comparison is known at the fourth step: it must be entered three stalls
- Anticipation of the comparison at the second step (ID)
  - You add extra hardware: after decoding the instruction, you can decide and modify the PC if necessary
  - However, a stall must be added before the instruction following the beq



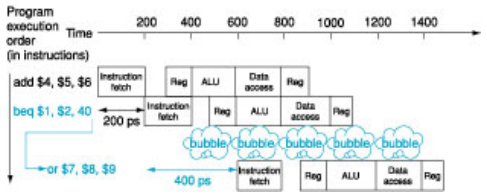
Solutions to control hazard (2)

- Jump prediction
  - Static prediction techniques
    - E.g.: it is predicted that the jump will not be performed (*untaken branch*)
  - Dynamic prediction techniques

Jump not performed



Jump performed



Soluzioni per il controllo (3)

- Delayed branch
  - In any case (regardless of the result of the comparison) the instruction that immediately follows the jump is executed (defined *branch-delay slot*)
  - Worst case
    - Entering nop
  - Best case
    - It is possible to find a pre-jump instruction that can be postponed to the jump *without* alter the flow of control (and data)
  - Example

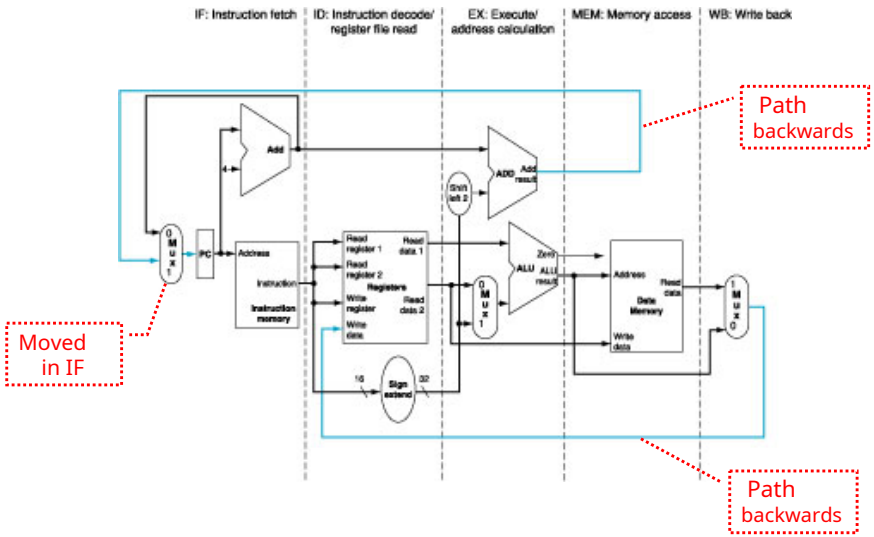
```
or $t0, $t1, $t2                    add $s0, $s1, $s2
add $s0, $s1, $s2                   sub $s3, $s4, $s5
sub $s3, $s4, $s5   delayed branch  beq $s0, $s3, Exit
beq $s0, $s3, Exit                  or $t0, $t1, $t2 xor
xor $t2, $s0, $t3 ...               $t2, $s0, $t3 ...

Exit:                               Exit:
```


Cause principali di complessità dell'unità di elaborazione

- The division of the instruction into 5 stages implies that 5 instructions are executed in each clock cycle
  - The structure of a 5-stage pipelined processor must be broken down into 5 parts (or *stages of execution*), each of which corresponds to one of the pipeline phases
- A separation between the various stages must be introduced
  - *Pipeline logs*
- Furthermore, several instructions executing at the same time may require similar hardware resources
  - Replication of hardware resources
- Let's take the diagram of the single cycle processing unit and identify the 5 stages

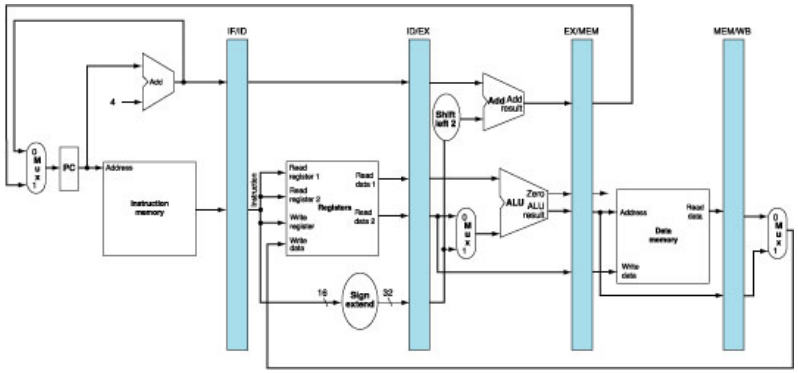
L'unità di elaborazione con pipeline



L'unità di elaborazione con pipeline

- Guiding principle:
  - Allow the reuse of components for the next instruction
- Introduction of *pipeline logs* (interstage registers)
  - At each clock cycle information proceeds from one pipeline register to the next
  - The name of the register is given by the name of the two stages it separates
    - Register IF/ID (Instruction Fetch / Instruction Decode)
    - Register ID/EX (Instruction Decode / EXecute)
    - Register EX/MEM (Execute / MEMory access)
    - Register MEM/WB (MEMory access / Write Back)
  - The PC can be considered as a pipeline register for the IF stage
- Compared to the single cycle unit, the PC multiplexer has been moved to the IF stage
  - To avoid conflicts in its writing in case of a jump instruction

L'unità di elaborazione con pipeline (2)

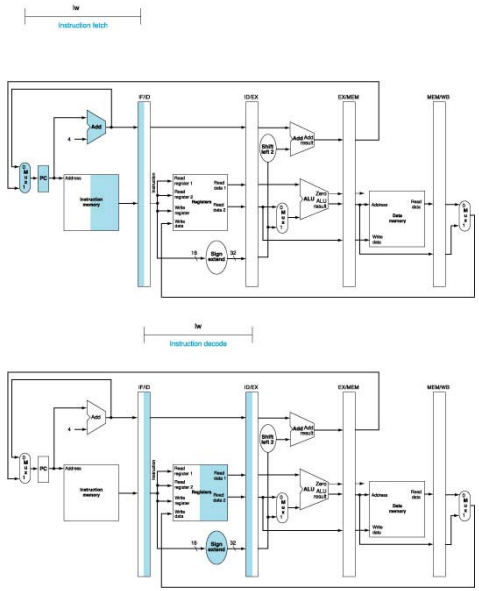


- What is the size of the pipeline registers that can be obtained from the scheme?
  - IF/ID: 64 bits (32+32)
  - ID/EX: 128 bits (32+32+32+32)
  - EX/MEM: 97 bits (32+32+32+1)
  - MEM/WB: 64 bit (32+32)

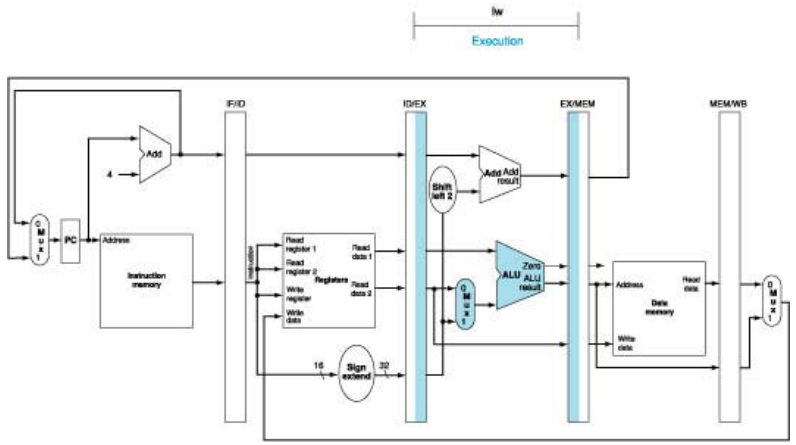
Use of the pipeline

- How is an instruction executed at various stages of the pipeline?
- Let's consider the lw statement first
  - Education withdrawal
  - Decoding the instruction and reading the registers
  - Execution (use of ALU for address calculation)
  - Reading from memory
  - Writing in the register
- We then analyze the execution of the SW instruction
- Finally, we consider the simultaneous execution of multiple instructions

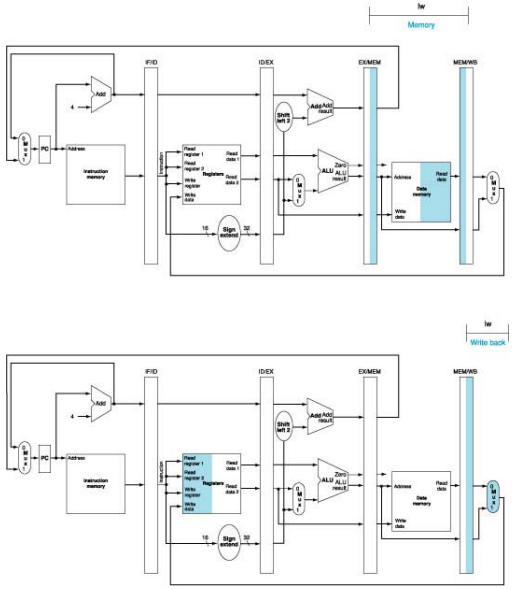
AND: Esecuzione dell'istruzione lw



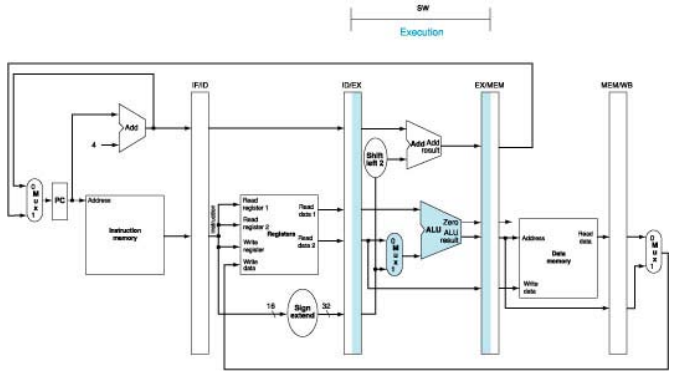
AND: Esecuzione dell'istruzione lw



AND ...

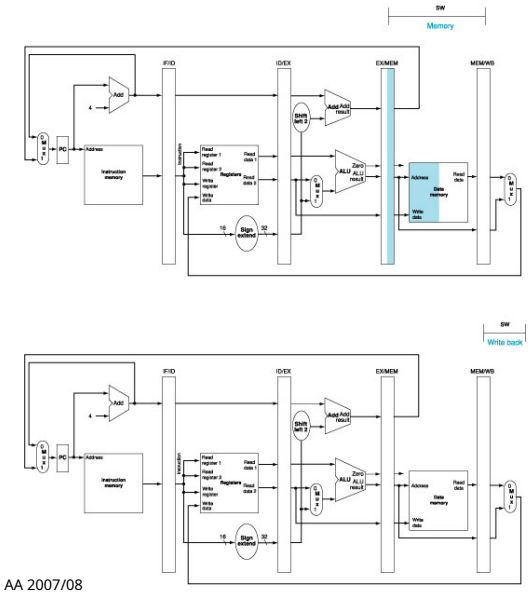


AND ...

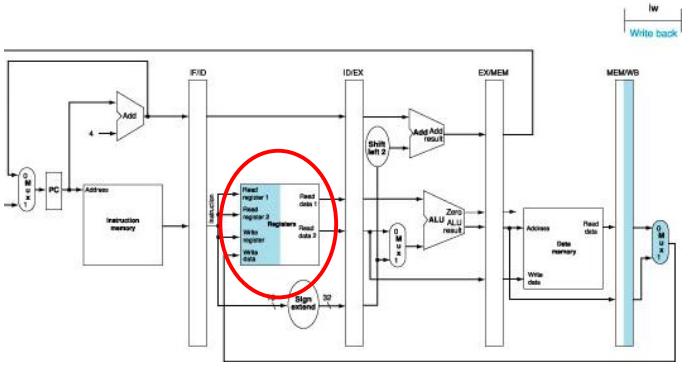


- The value of the second register is written to the ID/EX register to be able to use it in the MEM stage

AND ...

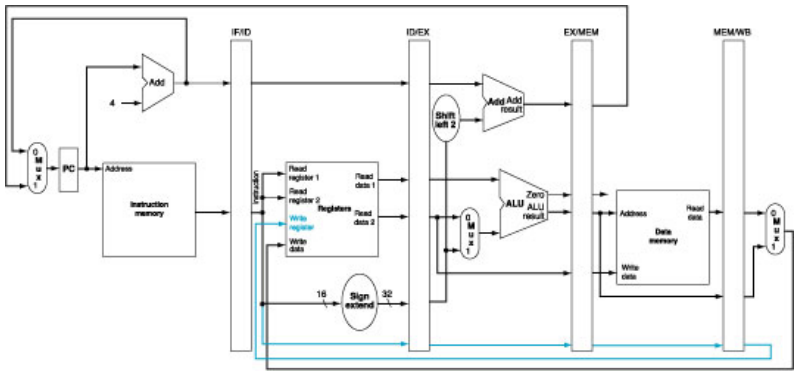


Command ...



- Which target register is written?
  - The IF/ID register contains an instruction following lw
- Solution
  - The destination register number of lw must be preserved

Solution



- The destination register number is written to the pipeline registers:
  - First in ID/EX, then in EX/MEM, finally in MEM/WB

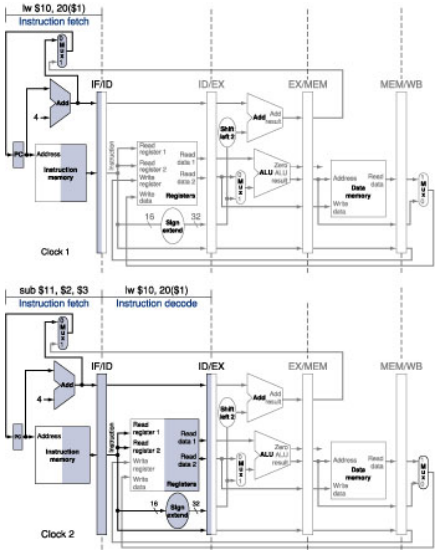
Pipelined execution: an example

- Consider the MIPS instruction sequence
  - lw $10, 20($1)
  - sub $11, $2, $3
- We analyze the execution of the sequence in the 6 necessary clock cycles

The pipelined execution: an example

Cycle 1
- lw: enter the pipeline

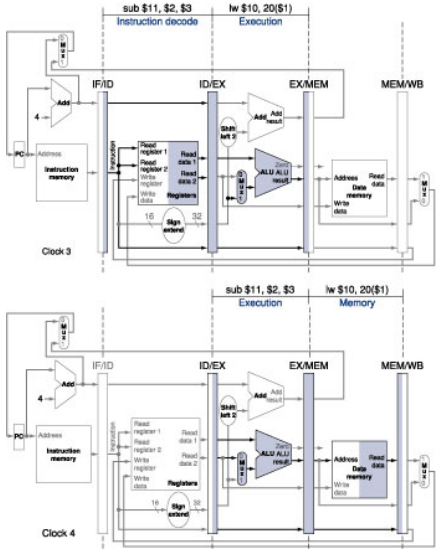
Cycle 2
- sub: enter the pipeline
- lw: enters the ID stage



The pipelined execution: an example

Cycle 3
- lw: enter the EX stage
- sub: enters the ID stage

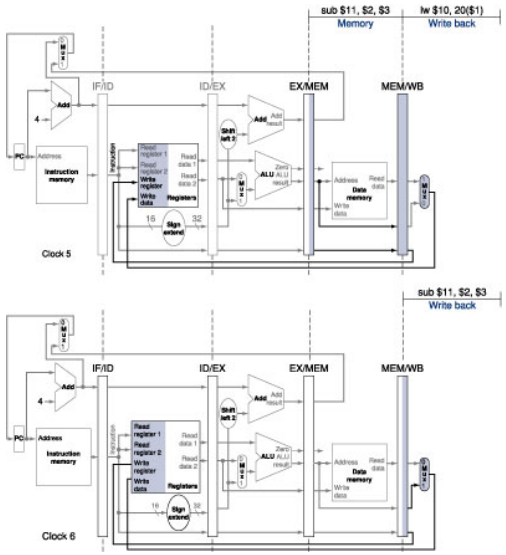
Cycle 4
- lw: enters the MEM stage and reads the addressed memory location saved in EX/MEM
- sub: enter the EX stage; the result of subtraction is written to EX/MEM at the end of the loop



The Multicycle Datapath: Adding Pipeline Registers

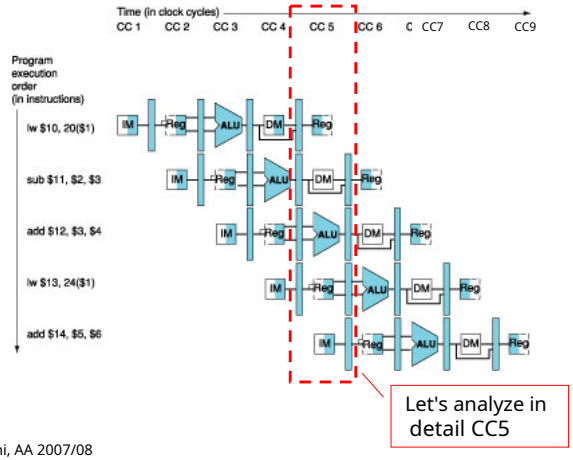
Cycle 5
- lw: ends by writing the value in MEM/WB to the register
  $10 from the bank
- sub: the subtraction result is written in MEM/WB

Cycle 6
- sub: ends by writing the value in MEM/WB to the register
  $11 from the dealer



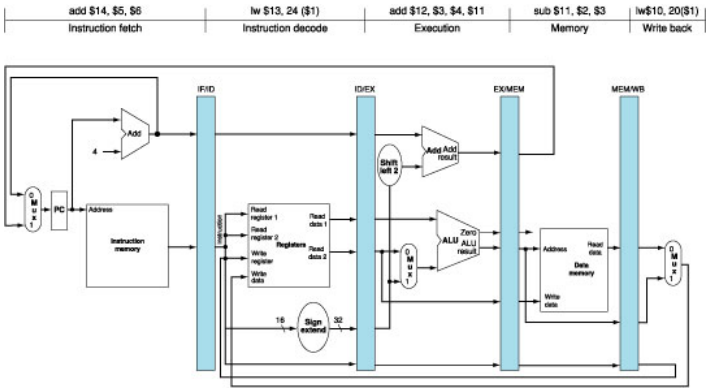
Graphically Representing Pipelines

- Pipeline diagram with multiple clock cycles
  - Provides a resource-oriented and simplified representation



The Single Cycle Pipeline Diagram

- Single clock cycle pipeline diagram
  - Provides a more detailed and vertical representation of the diagram with multiple clock cycles
  - Consider the fifth clock cycle of the pipeline in the previous slide



AND or OR?

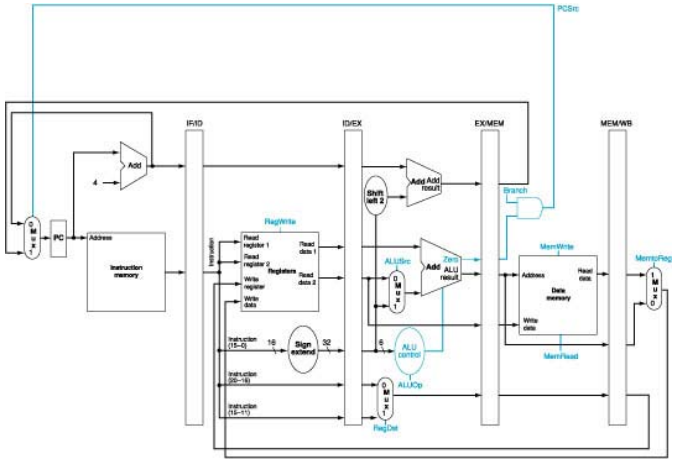
- Consider the MIPS instruction sequence
  add $4, $2, $3
  sw $5, 4($2)
- Analyze the execution of the sequence in the 6 necessary clock cycles

Control unit pipeline

- Data travels through pipeline stages
- All data belonging to an instruction must be maintained within the stage
- Information transfers only through pipeline registers

- Control information must travel with the instruction

The control signals



- No control signals are needed for writing pipeline registers

The control signals (2)

- We group control signals based on pipeline stages

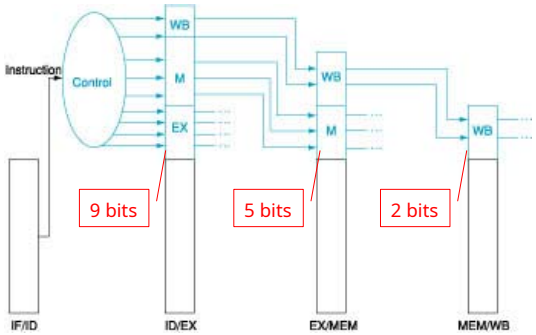
- Education withdrawal
  - Identical for all instructions
- Decoding the instruction/reading the register bank
  - Identical for all instructions
- Address execution/calculation
  - RegDst, ALUOp, ALUSrc
- Access to memory
  - Branch, MemRead,MemWrite
- Writing the result
  - MemtoReg, RegWrite

The control signals (3)

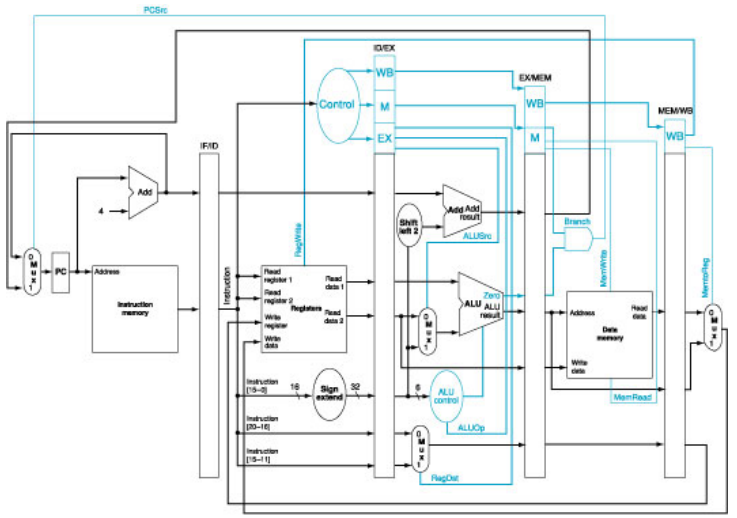
Instruction	EX control signals				Control signals MEM			Signals of WB control	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg Write	Memto Reg
type-R	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
well	X	0	1	0	1	0	0	0	X

AND Pipeline control LUT

- Pipeline registers also contain the values of control signals
  - Maximum 8 control signals (9 bits)
- The values needed for the next stage are propagated from the current pipeline register to the next



AND Pipeline control LUT (2)



AND Pipeline control: wicked or

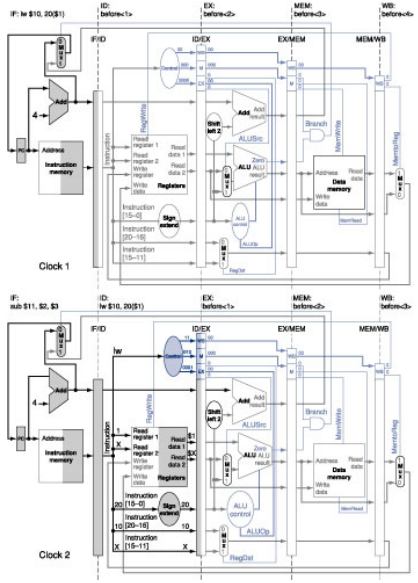
- Consider the MIPS instruction sequence

  lw $10, 20($1)
  sub $11, $2, $3
  and $12, $4, $5
  or $13, $6, $7
  add $14, $8, $9

- We analyze the execution of the sequence in the necessary 9 clock cycles

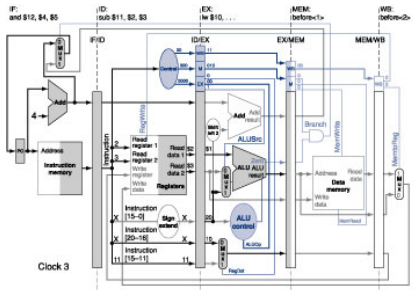
AND Pipeline control: wicked or: the clock 1 and 2

- lw: enter the pipeline
- sub: enter the pipeline
- lw: in ID/EX written $1, 20 (offset) and 10 (destination register number)

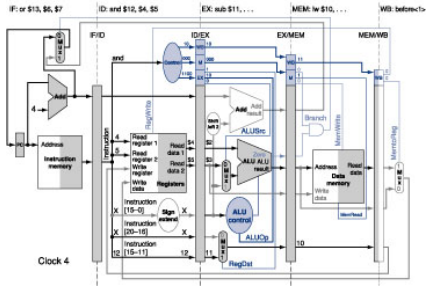


AND: wicked or: the the clock 3 and 4

- and: enter the pipeline
- sub: in ID/EX written $2, $3, and 11 (destination register number)
- lw: $1+20 and 10 written in EX/MEM

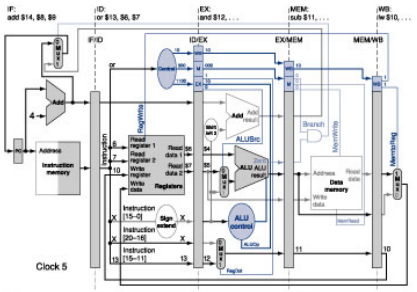


- or: enter the pipeline
- and: in ID/EX written $4, $5, and 12 (destination register number)
- sub: in EX/MEM written $2-$3 and 11
- lw: the value read from the memory and 10 are written in MEM/WB

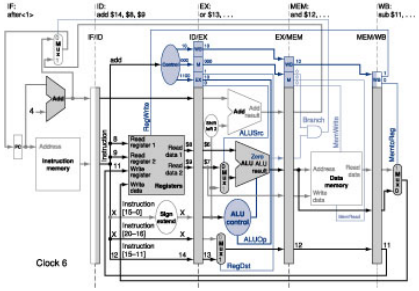


AND: wicked or: the the clock 5 and 6

- add: enter the pipeline
- or: in ID/EX written $6, $7, and 13 (destination register number)
- and: in EX/MEM written $4 AND $5 and 12
- sub: in MEM/WB written $2-$3 and 11
- lw: ends by writing $10

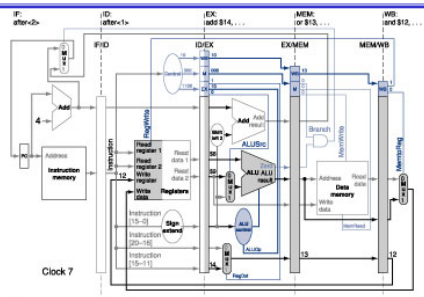


- add: $8, $9, and 14 (destination register number) written in ID/EX
- or: in EX/MEM written $6 OR $7 and 13
- and: in MEM/WB written $4 AND $5 and 12
- sub: ends by writing $11

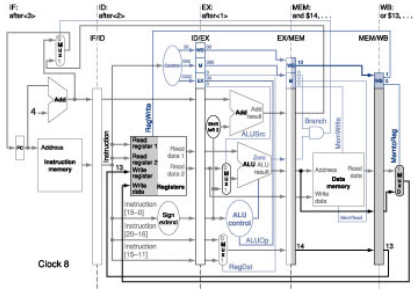


AND: wicked or: the the clock 7 and 8

- add: in EX/MEM written $8+$9 and 14
- or: in MEM/WB written $6 OR $7 and 13
- and: ends by writing $12

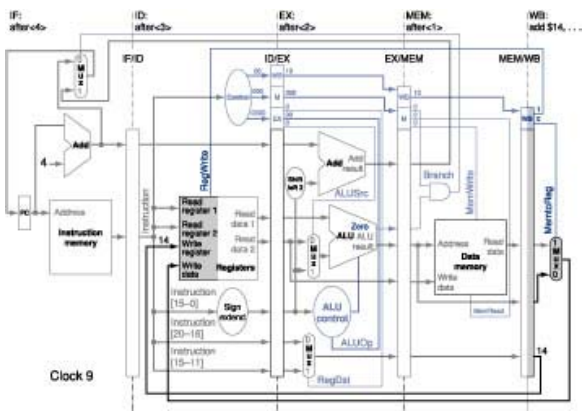


- add: in MEM/WB written $8+$9 and 14
- or: ends by writing $13



AND: wicked or: the the clock 9

- add: ends by writing $14



Pipelining

- Pipelining **increases throughput** of the processor (number of instructions completed per unit of time), but **Not** reduces the execution time (latency) of the single instruction
- Indeed, in general pipelining increases the execution time of a single instruction, due to imbalances between pipeline stages and pipeline control overhead
  - Imbalance between pipeline stages reduces performance
    - The clock cannot be less than the time needed for the slowest stage of the pipeline
  - Pipeline overhead is caused by pipeline register delays and clock skew (clock signal propagation delay on the wires)

Pipelining (2)

- The average execution time of an instruction for the unpipelined processor is:

$$\text{Average T exec.}_{\text{no pipelines}} = \text{Average CPI}_{\text{no pipelines}} \times \text{clock cycle}_{\text{no pipelines}}$$

- The speedup resulting from the introduction of pipelining is:

$$\text{Speedup}_{\text{pipeline}} = \frac{\text{Average T ex.}_{\text{no pipelines}}}{\text{Average T exec.}_{\text{pipeline}}} = \frac{\text{Average CPI}_{\text{no pipelines}}}{\text{Average CPI}_{\text{pipeline}}} \times \frac{\text{clock cycle}_{\text{no pipelines}}}{\text{clock cycle}_{\text{pipeline}}}$$

Pipelining and ideal CPI

- The ideal CPI of a pipelined processor is almost always 1; however, stalls lead to performance degradation, therefore:

$$\text{CPI}_{\text{pipeline}} = \text{ideal CPI} + \text{pipeline stall cycles per instruction} = 1 + \text{pipeline stall cycles per instruction}$$

- Pipeline stall cycles per instruction are due to:
  - Structural criticalities + data criticalities + control criticalities

Pipelining and ideal CPI (2)

- Neglecting the clock time overhead of pipelining and assuming that the pipeline stages are perfectly balanced
  - the cycle time of the two processors can be considered equal, therefore:

$$\text{Speedup}_{\text{pipeline}} = \frac{\text{CPI}_{\text{no pipelines}}}{1 + \text{pipeline stall cycles per instruction}}$$

  - Simple case: all instructions require the same number of cycles, which corresponds to the number of pipeline stages (also called *pipeline depth*)

$$\text{Speedup}_{\text{pipeline}} = \frac{\text{Pipeline depth}}{1 + \text{pipeline stall cycles per instruction}}$$

  - If there are no stalls (ideal case), pipelining increases performance by a factor equal to the depth of the pipeline