Pipelining

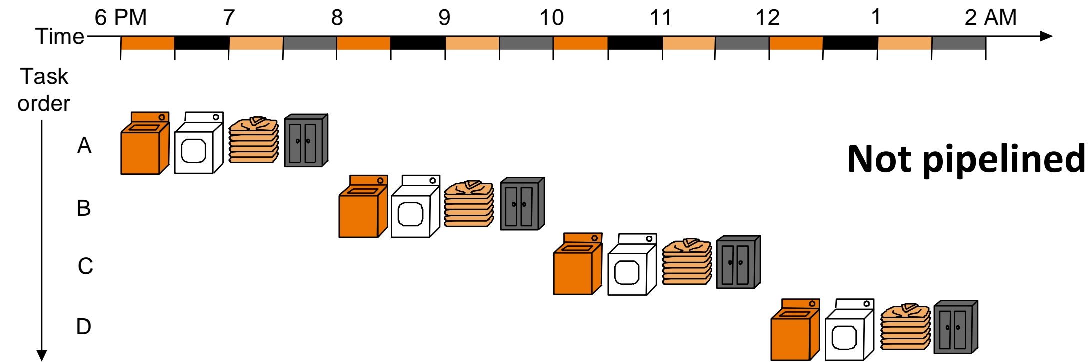
Pipelining

Pipelining is an implementation technique in which multiple instructions are overlapped in execution.
Anyone who has done a lot of laundry has intuitively used pipelining.
The *non-pipelined* approach to laundry would be as follows:
1. Place one dirty load of clothes in the washer.
2. When the washer is finished, place the wet load in the dryer.
3. When the dryer is finished, place the dry load on a table and fold.
4. When folding is finished, ask your roommate to put the clothes away.

When your roommate is done, start over with the next dirty load

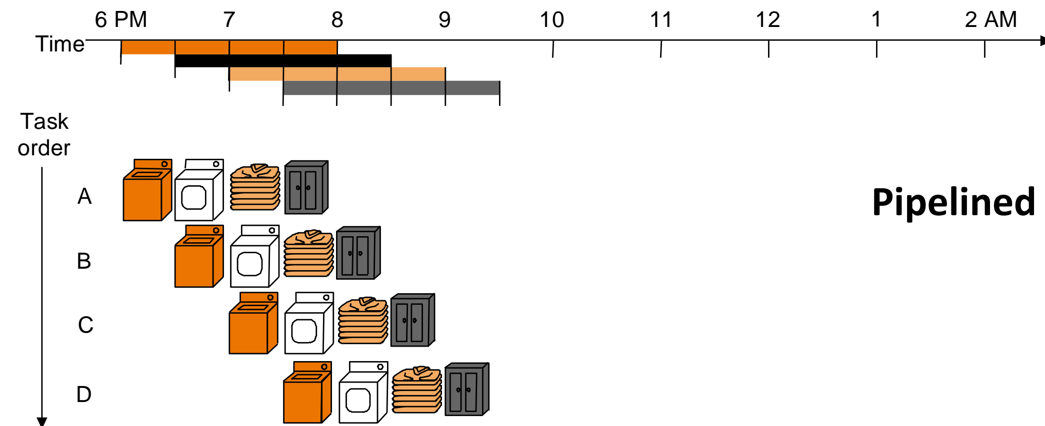
Pipelining

- Start work ASAP!! Do not waste time!



Not pipelined

Assume 30 min. each task – wash, dry, fold, store – and that separate tasks use separate hardware and so can be overlapped

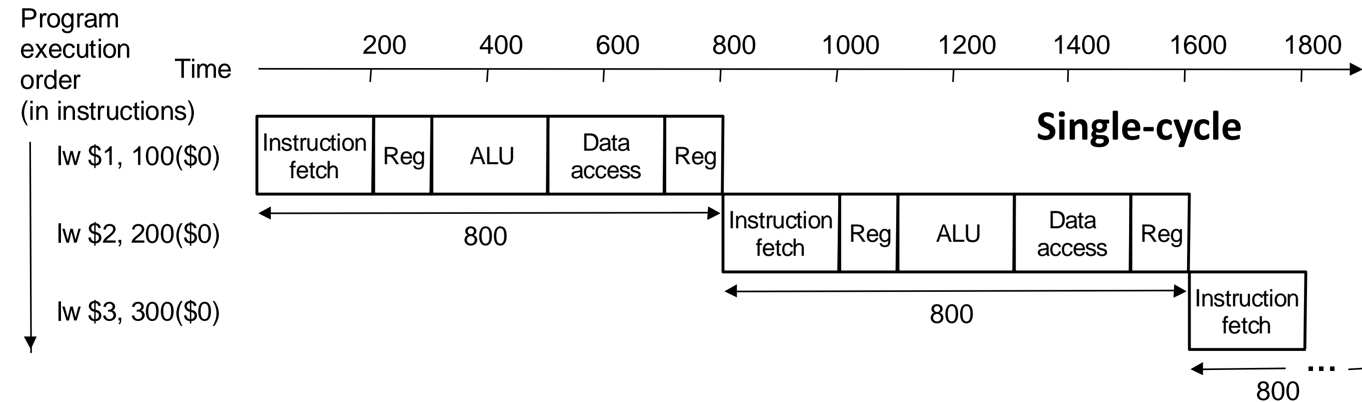


Pipelined

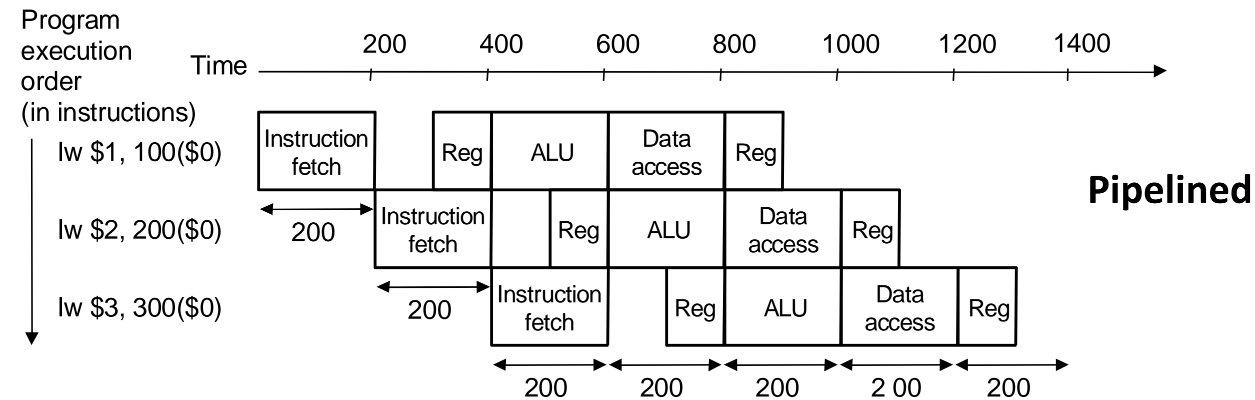
- The same principles apply to processors where we pipeline instruction-execution. MIPS instructions classically take five steps:
  1. Fetch instruction from memory.
  2. Read registers while decoding the instruction. The regular format of MIPS instructions allows reading and decoding to occur simultaneously.
  3. Execute the operation or calculate an address.
  4. Access an operand in data memory.
  5. Write the result into a register

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, AND, OR, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

Pipelined vs. Single-Cycle Instruction Execution: the Plan



Assume 200 ps for memory access, ALU operation; 100 ps for register access: therefore, single cycle clock 800 ps; pipelined clock cycle 200 ps.



Any condition that causes a stall in the pipeline operations can be called a **hazard.** There are primarily three types of hazards:

i. Data Hazards

ii. Control Hazards or instruction Hazards

iii. Structural Hazards.

i. **Data Hazards:** Data hazards arise because of the unavailability of an operand

A data hazard is any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline. As a result of which some operation has to be delayed and the pipeline stalls. Whenever there are two instructions one of which depends on the data obtained from the other.

- A=3+A
- B=A*4

For the above sequence, the second instruction needs the value of 'A' computed in the first instruction.

Thus the second instruction is said to depend on the first.

If the execution is done in a pipelined processor, it is highly likely that the interleaving of these two instructions can lead to incorrect results due to data dependency between the instructions. Thus the pipeline needs to be stalled as and when necessary to avoid errors.

ii. **Control Hazards:**

Instructions that disrupt the sequential flow of control present problems for pipelines. The effects of these instructions can't be exactly determined until late in the pipeline, so instruction fetch can't continue unless we do something special. The following types of instructions can introduce control hazards: Unconditional branches, Conditional branches, Indirect branches, Procedure calls etc.

iii. **Structural Hazards:**

This situation arises mainly when two instructions require a given hardware resource at the same time and hence for one of the instructions the pipeline needs to be stalled.

The most common case is when memory is accessed at the same time by two instructions. One instruction may need to access the memory as part of the Execute or Write back phase while other instruction is being fetched. In this case if both the instructions and data reside in the same memory. Both the instructions can’t proceed together and one of them needs to be stalled till the other is done with the memory access part. Thus in general sufficient hardware resources are needed for avoiding structural hazards.

Data Hazards and solutions

- Example: Let there be two instructions I1 and I2 such that:
  I1 : ADD R1, R2, R3
  I2 : SUB R4, R1, R2

When the above instructions are executed in a pipelined processor, then data dependency condition will occur, which means that I_2 tries to read the data before I_1 writes it, therefore, I_2 incorrectly gets the old value from I_1.

INSTRUCTION / CYCLE	1	2	3	4
I_1	IF	ID	EX	DM
I_2		IF	ID(Old value)	EX

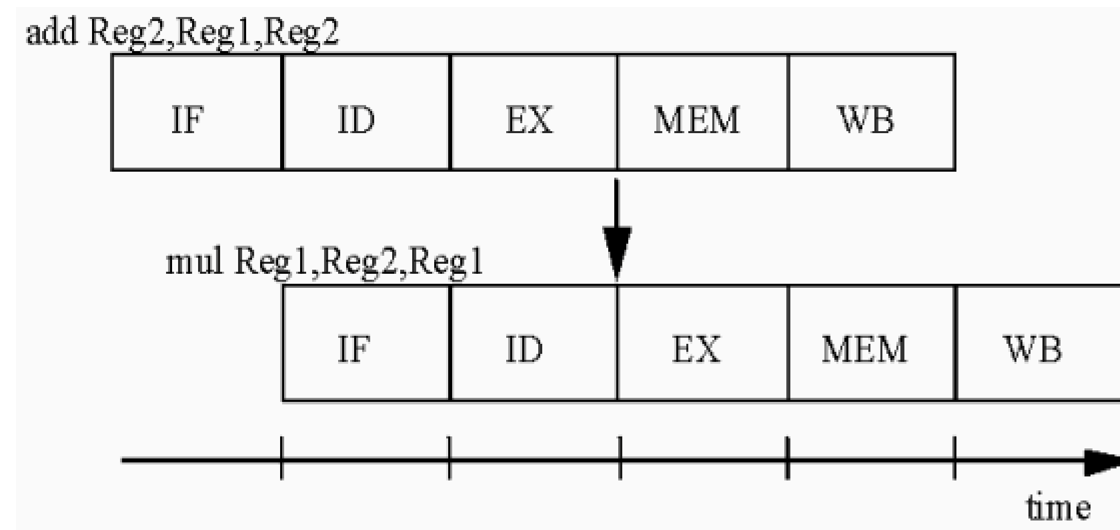
- To minimize data dependency stalls in the pipeline, **operand forwarding** is used.

Considering the same example:
I1 : ADD R1, R2, R3
I2 : SUB R4, R1, R2

Interlocking: stall pipeline for one or more cycles

INSTRUCTION / CYCLE	1	2	3	4
I_1	IF	ID	EX	DM
I_2	bubble	bubble	IF	ID

Operand Forwarding : In operand forwarding, we use the interface registers present between the stages to hold intermediate output so that dependent instruction can access new value from the interface register directly.



Control Hazards and solution

- This type of dependency occurs during the transfer of control instructions such as BRANCH, CALL, JMP, etc. On many instruction architectures, the processor will not know the target address of these instructions when it needs to insert the new instruction into the pipeline. Due to this, unwanted instructions are fed to the pipeline.
- Consider the following sequence of instructions in the program:
  100: I_1
  101: I_2 (JMP 250)
  102: I_3
  .
  .
  250: BI_1
- Expected output: I_1 -> I_2 -> BI_1

NOTE: Generally, the target address of the JMP instruction is known after ID stage only.

INSTRUCTION/ CYCLE	1	2	3	4	5	6
I_1	IF	ID	EX	MEM	WB	
I_2		IF	ID (PC:250)	EX	Mem	WB
I_3			IF	ID	EX	Mem
BI_1				IF	ID	EX

Output Sequence: I_1 -> I_2 -> I_3 -> BI_1

So, the output sequence is not equal to the expected output, that means the pipeline is not implemented correctly.

To correct the above problem we need to stop the Instruction fetch until we get target address of branch instruction. This can be implemented by introducing delay slot until we get the target address.

INSTRUCTION/ CYCLE	1	2	3	4	5	6
I_1	IF	ID	EX	MEM	WB	
I_2		IF	ID (PC:250)	EX	Mem	WB
Delay	–	–	–	–	–	–
BI_1				IF	ID	EX

- Output Sequence: I_1 -> I_2 -> Delay (Stall) -> BI_1

As the delay slot performs no operation, this output sequence is equal to the expected output sequence. But this slot introduces stall in the pipeline.

- **Solution for Control dependency** Branch Prediction is the method through which stalls due to control dependency can be eliminated. In this, at 1st stage, prediction is done about which branch will be taken. For branch prediction Branch penalty is zero.

Structural Hazards and solutions

This dependency arises due to the resource conflict in the pipeline. A resource conflict is a situation when more than one instruction tries to access the same resource in the same cycle. A resource can be a register, memory, or ALU.

Example:

INSTRUCTION / CYCLE	1	2	3	4	5
I_1	IF(Mem)	ID	EX	Mem	
I_2		IF(Mem)	ID	EX	
I_3			IF(Mem)	ID	EX
I_4				IF(Mem)	ID

In the above scenario, in cycle 4, instructions I_1 and I_4 are trying to access same resource (Memory) which introduces a resource conflict.
To avoid this problem, we have to keep the instruction on wait until the required resource (memory in our case) becomes available. This wait will introduce stalls in the pipeline as shown below:

CYCLE	1	2	3	4	5	6	7	8
I_1	IF(Mem)	ID	EX	Mem	WB			
I_2		IF(Mem)	ID	EX	Mem	WB		
I_3			IF(Mem)	ID	EX	Mem	WB	
I_4				–	–	–	IF(Mem)	

Solution for Structural Hazards

To minimize structural dependency stalls in the pipeline, we use a hardware mechanism called Renaming.
Renaming : According to renaming, we divide the memory into two independent modules used to store the instruction and data separately called Code memory(CM) and Data memory(DM) respectively. CM will contain all the instructions and DM will contain all the operands that are required for the instructions.

INSTRUCTION/ CYCLE	1	2	3	4	5	6	7
I_1	IF(CM)	ID	EX	DM	WB		
I_2		IF(CM)	ID	EX	DM	WB	
I_3			IF(CM)	ID	EX	DM	WB
I_4				IF(CM)	ID	EX	DM
I_5					IF(CM)	ID	EX
I_6						IF(CM)	ID
I_7							IF(CM)