Pipelining

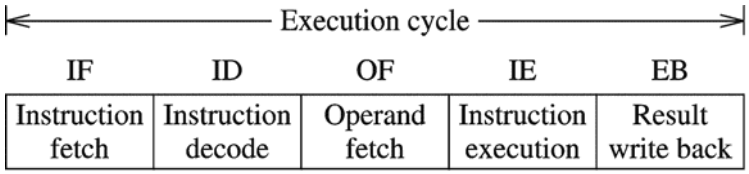
COMP375 Computer Architecture and Organization

Parallelism

- The most common method of making computers faster is to increase parallelism.
- There are many levels of parallelism
  - Macro
    - Multiple processes
    - Multiple threads
  - Micro
    - Pipelining
    - Multiple ALUs

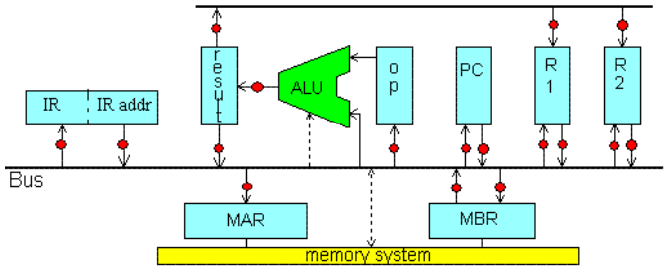
Execution Cycle

- There are several steps in the execution cycle.
- With additional internal buses, each step can be done by a separate piece of hardware.



Dual Bus Simple CPU

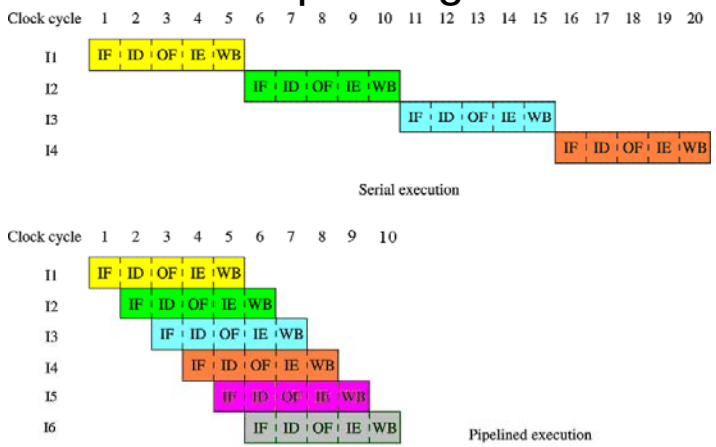
Many instructions end by copying the value in the result register back into a user register or the program counter. This could overlap with the start of the next instruction if a second bus was available.



Assembly Line

- Pipelining is like an assembly line. Each stage of the execution cycle performs its function and passes the instruction to the next cycle.
- Each stage can be working on a different instruction at the same time.
- Several instructions can be in the process of execution simultaneously.
- There is no reason to keep the ALU idle when you are fetching or storing an operand.

Pipelining



Simple Model of Execution

- Instruction sequence is determined by a simple conceptual control point.
- Each instruction is completed before the next instruction starts.
- One instruction is executed at a time.

- Pipelining seems to violate these assumptions. The challenge is to use parallelism invisibly.

Difficulties

- Pipelining works best if each instruction is independent and takes the same time to execute.
- Some instructions take more time than others. Double Precision floating point divide takes much more time than an integer add.
- The time to fetch an operand can vary depending if it is in a register, cache or RAM

For an add instruction using immediate addressing, what stage probably takes the longest time?

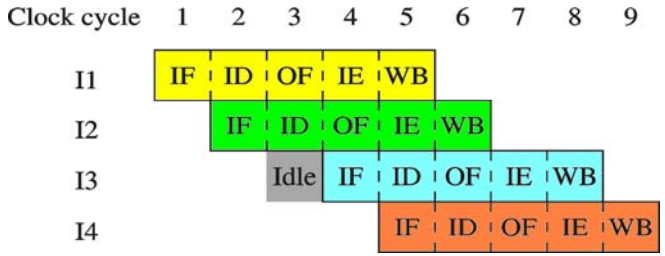
1. Instruction fetch
2. Decode
3. Operand fetch
4. Execution
5. Store results

Hazards

- A hazard is a situation that reduces the processors ability to pipeline instructions.
- **Resource** – When different instructions want to use the same CPU resource.
- **Data** – When the data used in an instruction is modified by the previous instruction.
- **Control** – When a jump is taken or anything changes the sequential flow.

Resource Hazards

- In the example below, both the operand fetch and instruction fetch stages are using the memory system.
- Hazards can cause pipeline stalls.

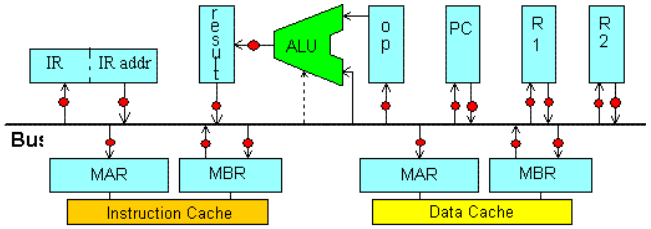


Pipeline Stall

- When a hazard prevents an instruction step from happening, the processor pauses executing that step until the hazard is resolved.
- Pipeline stalls slow the execution of an instruction, but do not prevent it from executing correctly.

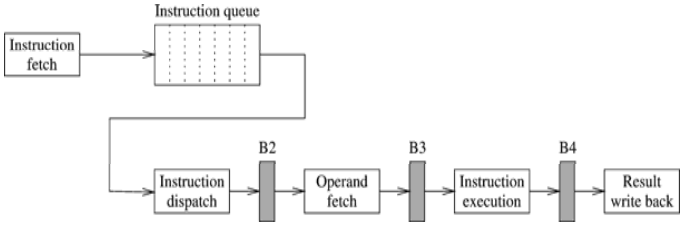
Separate Caches

- The problem of instruction fetch and operand fetch resource hazards can be reduced by having separate caches for instructions and data.



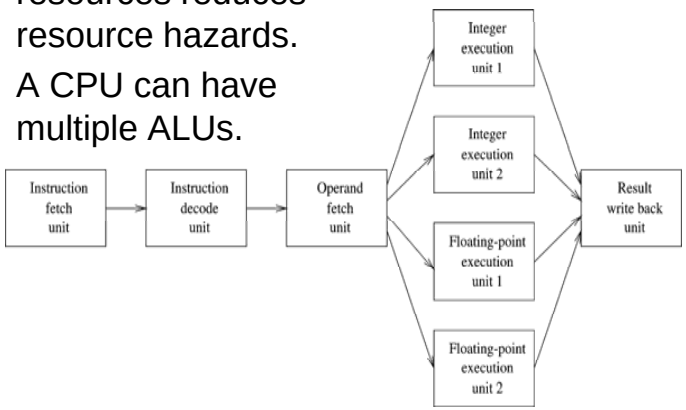
Queues

- The problem of operand and instruction fetch conflicts can be reduced by pre-fetching several instructions.
- If the operand fetch needs to use the memory, there are still instructions to start.



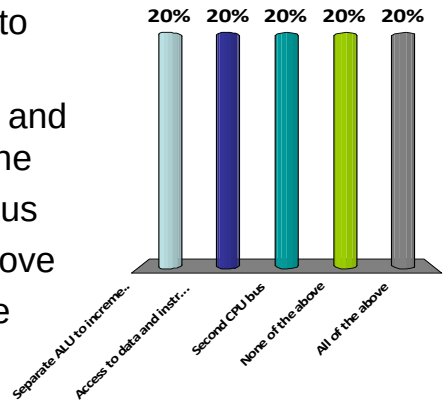
Multiple Resources

- Providing multiple resources reduces resource hazards.
- A CPU can have multiple ALUs.



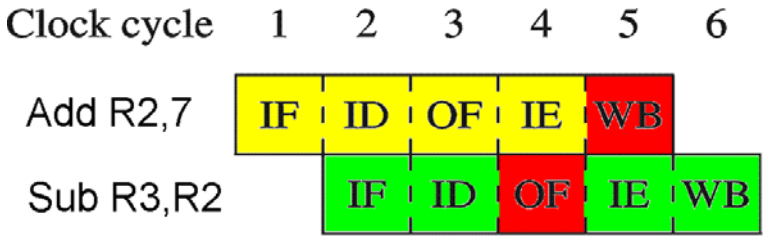
What additional resource would speed our microcode computer?

1. Separate ALU to increment PC
2. Access to data and instruction cache
3. Second CPU bus
4. None of the above
5. All of the above



Data Hazards

- The data used by one instruction may be modified by a previous instruction.
- If the previous instruction has not completed and stored the results, the next instruction will use an incorrect value.

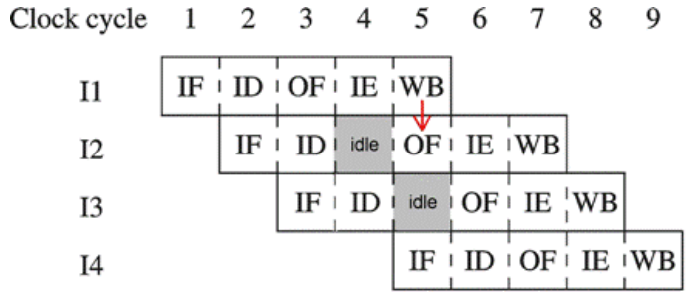


Data Hazard Resolution

- Register Forwarding – The data from a previous instruction can be used by the next instruction before or while it is being written back.
- Register locking – When a register is in use by an instruction, that register is locked to following instructions until the first instruction completes. This avoids incorrect results but introduces delays.

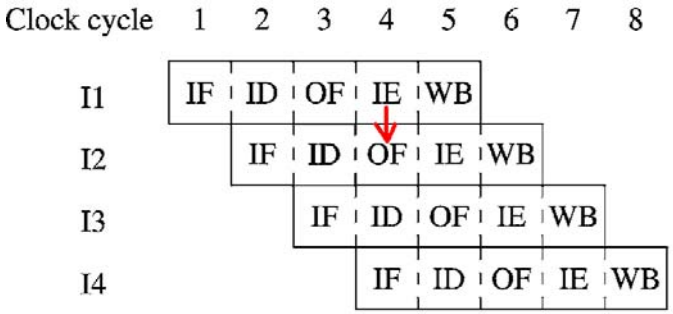
Register Forwarding

- When the result is copied from the ALU result register back to the user register, it can also be copied to the ALU operand reg.



Better Register Forwarding

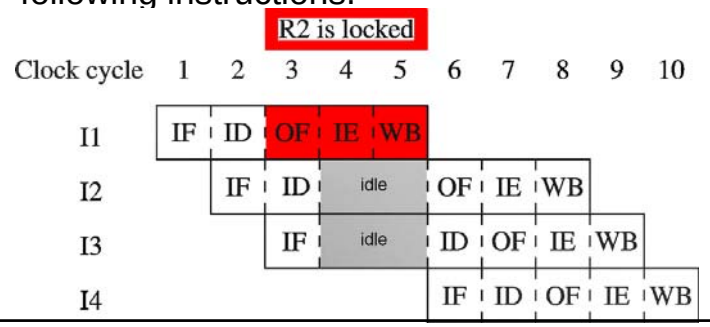
- The output of the ALU is immediately copied to the ALU operand register.



Register Locking

- When an instruction changes a register, that register is locked to following instructions.

```
Add   R2,7
Sub   R3,R2
```

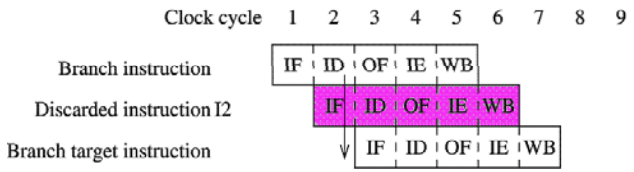


Control Hazards

- A jump or function call changes the sequential execution of instructions.
- The pipelined instruction fetch stage continually fetches sequential instructions.
- When a jump occurs, the previously fetched instructions should not be executed.
- Instructions in the pipe may have to be discarded before the write back stage.

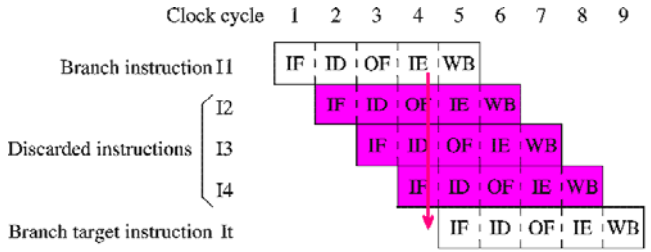
Unconditional Jump

- The CPU can detect that an instruction is an unconditional jump during the instruction decode state.
- The next instruction that has already been fetched must be discarded.



Conditional Jump

- The processor cannot determine if the jump will be taken until the execution stage.
- Three instructions in the pipe must be discarded.



Control Hazard Improvement

- The impact of control hazards can be reduced by branch prediction.
- For conditional branches or jumps, the CPU can guess which way the jump will go.
- For loops, the CPU can remember that an instruction is the end of a sequence.

Software and Hardware Solutions

- Hazards can be resolved by the hardware or the software.
- Some processors (such as the Intel Pentium) produce correct results regardless of hazards. Hazards just slow execution.
- Other processors assume that the software will avoid hazards. The compilers must prevent data hazards.
- There is a general trend towards moving the intelligence to the software.

Using A Pipeline

- Pipeline is *transparent* to programmer
- Disadvantage: programmer who does not understand pipeline can produce inefficient code
- Reason: hardware automatically *stalls* pipeline if items are not available

Software Solutions

- The compiler can add NOOP instructions or rearrange the instruction order.

```
Add   R2,X
Sub   R4,R2
Add   R3,Y
Sub   R5,R3
```

```
Add   R2,X
noop
Sub   R4,R2
Add   R3,Y
noop
Sub   R5,R3
```

```
Add   R2,X
Add   R3,Y
Sub   R4,R2
Sub   R5,R3
```

Assume a short two stage pipeline for these examples

Pipeline Efficient Code

Inefficient	Pipeline Efficient
Load R1, W	Load R1, W
Add R1, 7	Load R2, X
Store R1, W	Load R3, Y
Load R1, X	Load R4, Z
Add R1, 8	Add R1, 7
Store R1, X	Add R2, 8
Load R1, Y	Add R3, 9
Add R1, 9	Add R4, 10
Store R1, Y	Store R1, W
Load R1, Z	Store R2, X
Add R1, 10	Store R3, Y
Store R1, X	Store R4, Z

The pipeline efficient code will execute faster because

1. less control hazards
2. less data hazards
3. fewer instructions
4. they will run the same speed