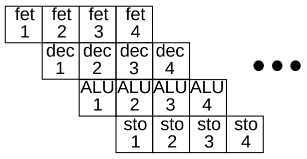
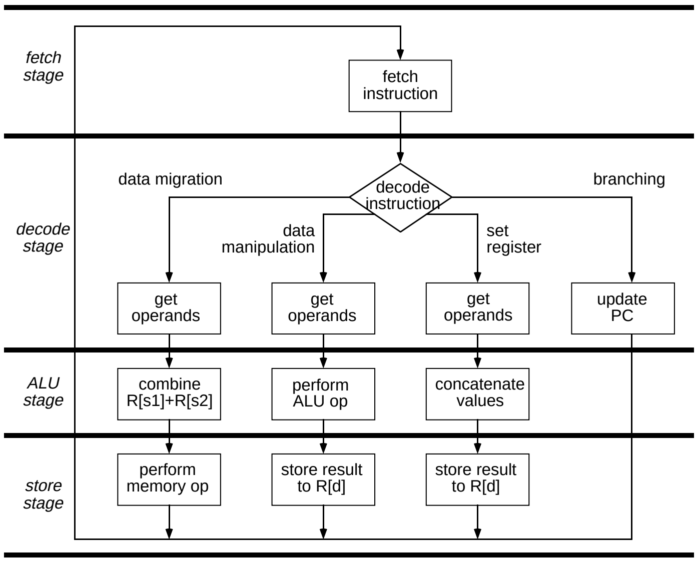
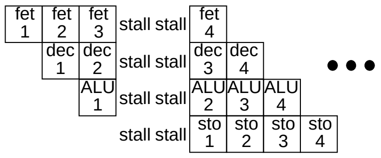
Pipelined Instruction Decode / Execute Loop (Review)





Pipeline Stalls

- If the needed instruction is not in the cache, the fetch stage for that instruction may take much longer than the other stages (more than a single clock cycle)



  - When this occurs, the pipeline *stalls* — there is no activity until that stage can continue

- In LOAD/STORE machines, the pipeline will generally consider a memory-based operand access as “completed” once we start that memory access
  - But — what if the next instruction needs a value before the memory access for that value has been completed?

Data Hazards

- Two instructions are said to have a *data dependency* between them if the second uses the result of the first
- A problem in the pipeline due to a data dependency is called a *data hazard*
- Two common data hazards:
  - *Write / read data hazard* — one instruction writes to a register, and a later instruction uses (reads) that result

    ADD R2, R3, R4

    ADD R1, R2, R6

    - Also useful if first instruction is a LOAD
  - *Write / write data hazard* — one instruction writes to a register, and a later instruction writes to that same register

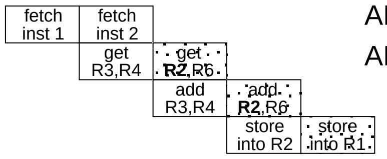
    ADD R2, R3, R4

    ADD R2, R5, R6

    - Later instructions should get the last value

Handling Data Hazards

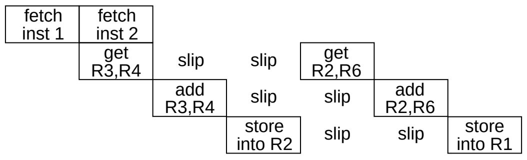
- *Write / read data hazard* example:



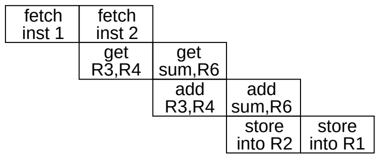
ADD R2, R3, R4

ADD R1, R2, R6

- Can be avoided with *register interlocks*



- Can also be avoided with *data forwarding*



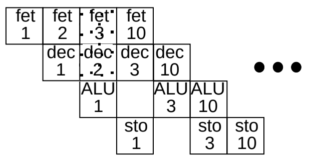
Handling Data Hazards (cont.)

- Register interlocks
  - An instruction gets *blocked* until all its source registers are loaded with the appropriate values by earlier instructions
  - A “valid / invalid” bit is associated with each register
    - During decode stage, destination register is set to invalid (it will change)
    - Decode stage blocks until all its source (and destination) registers are valid
    - Store stage sets destination register to valid
- Data forwarding
  - Output of ALU is connected directly to ALU input buses
  - Result of an ALU operation is now available immediately to later instructions (i.e., even before it gets stored in its destination register)

Handling Branch Hazards

- Consider this example:

```
1)         ADD   …
2)         BRA   test
3)         SUB   …
           …
10) test:  MPY   …
```



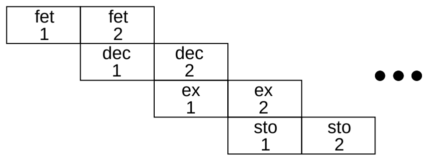
  - Instruction 2 doesn’t update the PC until the decode stage (and then does nothing in the ALU and store stages
  - What happens to instruction 3, which has already been fetched?
- One answer — *nullify* any instructions following a branch instructions that have entered the pipeline
- Another answer — always execute the one instruction always following the branch (the instruction said to be in the *branch delay slot*)

Superpipelined & Superscalar Machines(cont.)

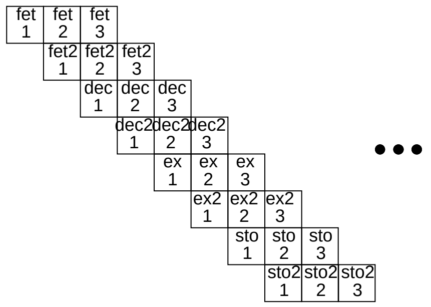
- Superpipelined machine
  - Increase the number of pipeline stages even further
  - Execution time of single instruction remains the same, but pipeline throughput increases
- Superscalar machine
  - Perform several instructions in parallel
  - CPU may contain:
    - Branch unit, to do decoding and process branch instructions
    - Integer unit, to perform integer arithmetic
    - Floating-point unit, to perform floating-point arithmetic
    - Integer and floating-point units work independently, signal branch unit when done (like memory does)

Superpipelined & Superscalar Machines

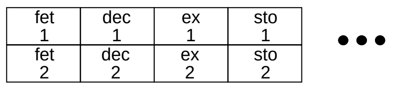
- Normal pipelined machine:



- Superpipelined machine:



- Superscalar machine:



Homework #6 — Due 12/7/98

1. How do branches cause problems when wide memories are used?

2. How do the level 1 cache, level 2 cache, and main memory compare in size and access time?

3. Suppose the operations that need to be performed by one stage in a pipeline take longer than those in the other stages. Does this affect the pipeline? Explain.

4. Pipeline stalls and slips are pretty similar. When does each occur, and how is the pipeline affected by each?

5. Explain the difference between data parallelism and control parallelism.

(This is the last question on Homework #7)