

COM 410 – Slides 3

The X86-64 Processor Part II

(Review: program state, registers, irmove, etc..)

Today:

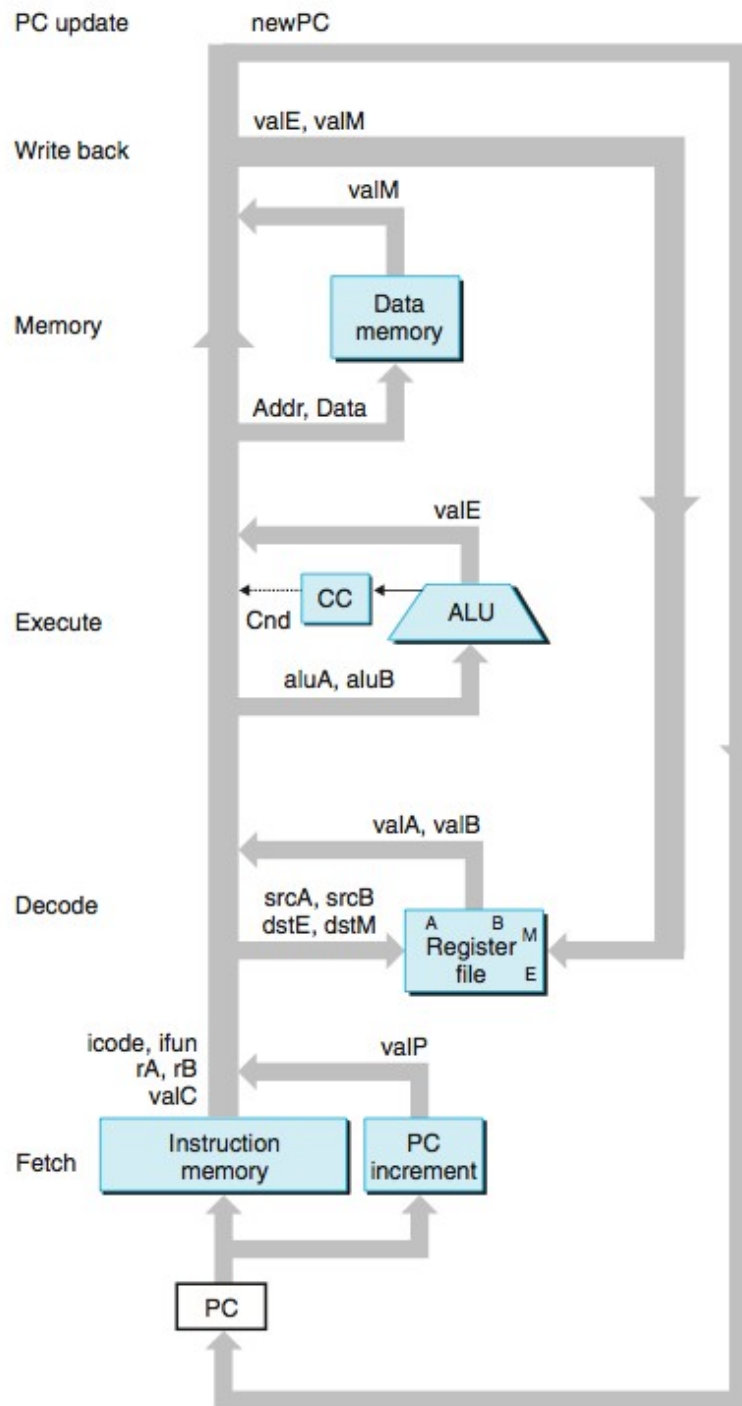
- * Logic Design and Hardware Control Language (HCL)

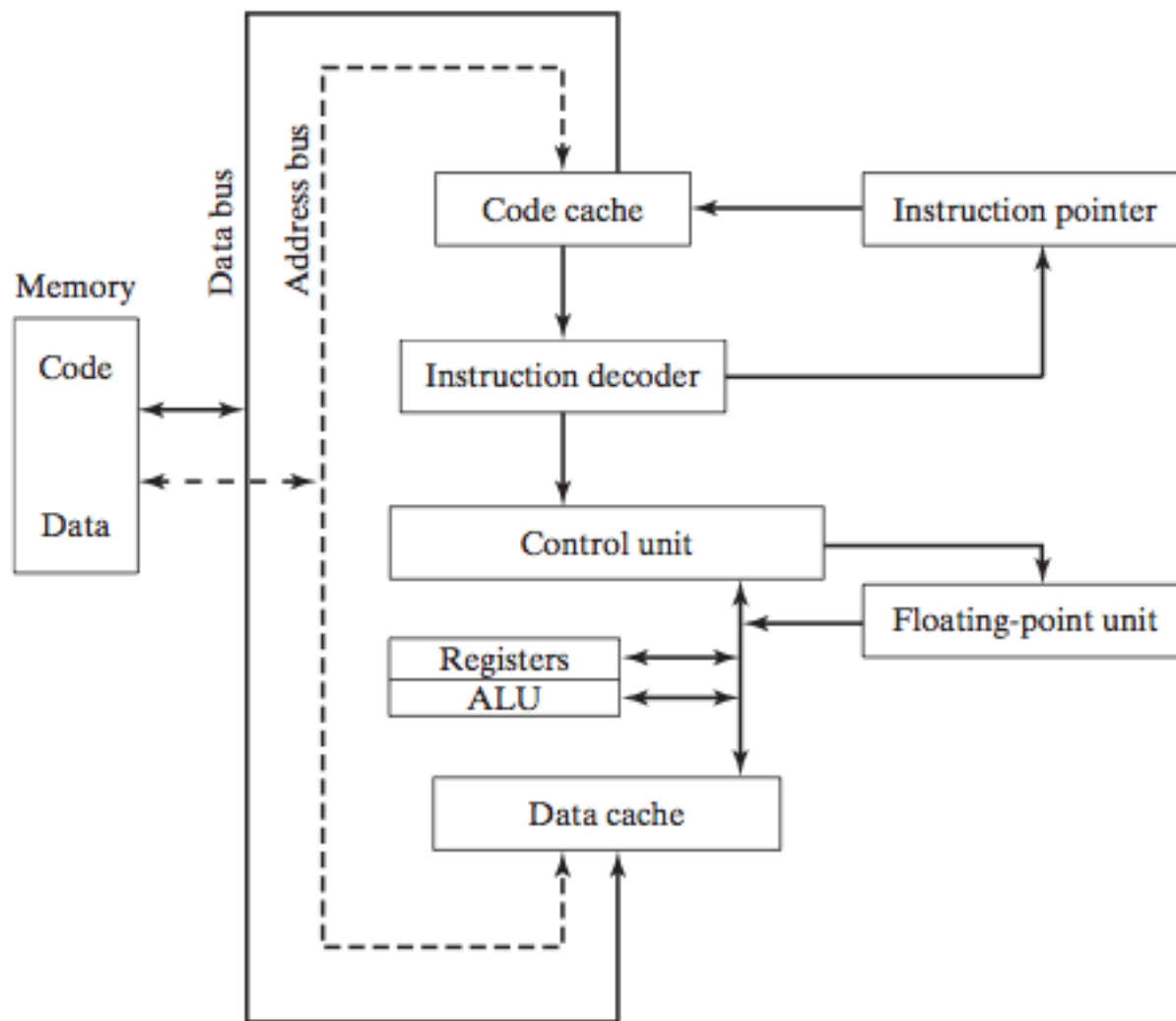
Sequential X86-64 Implementations

General Principles of Pipelining

Pipelined X86-64 Implementations

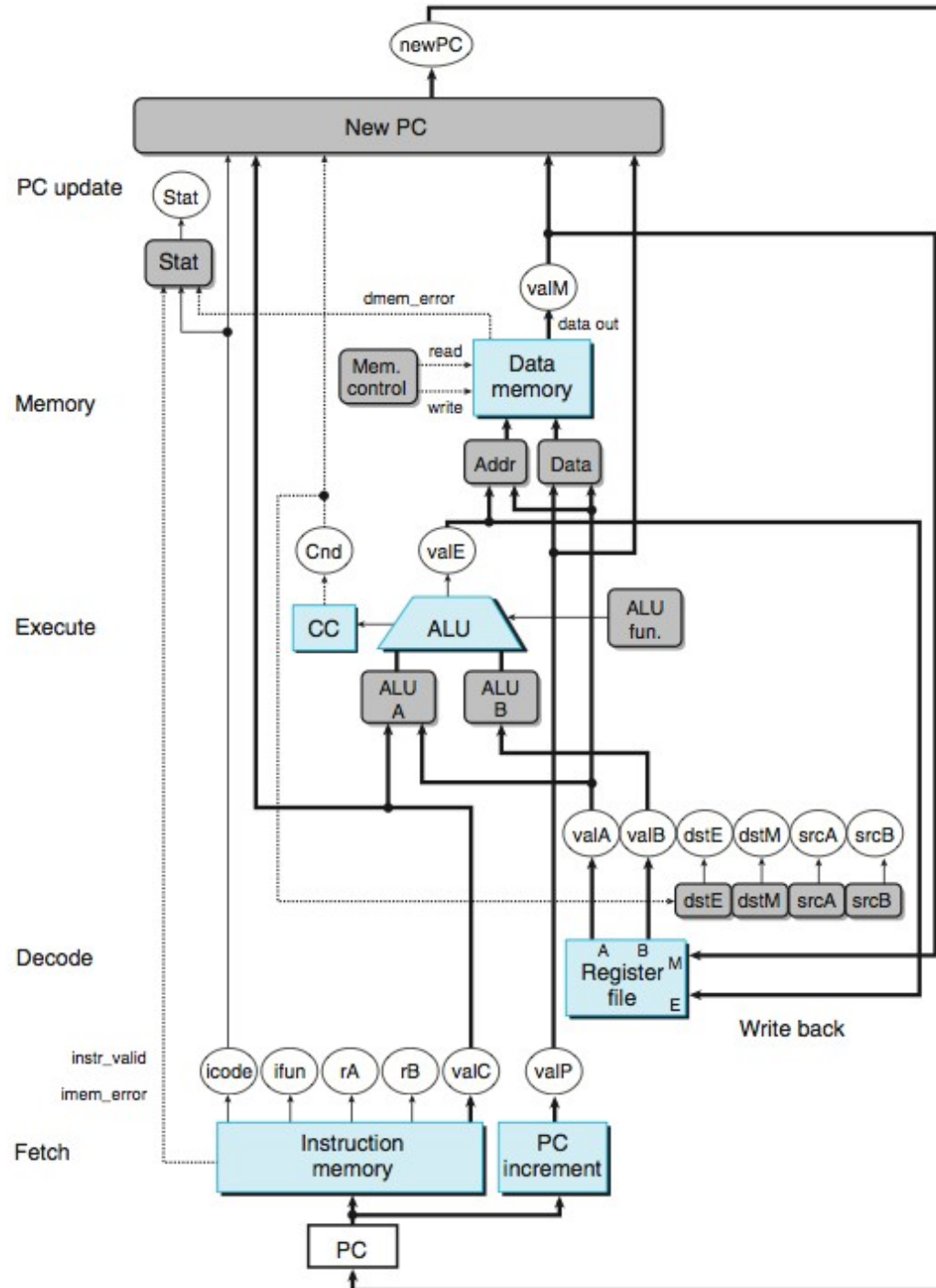
- * Postponed until after Chapters 2 & 3, if needed



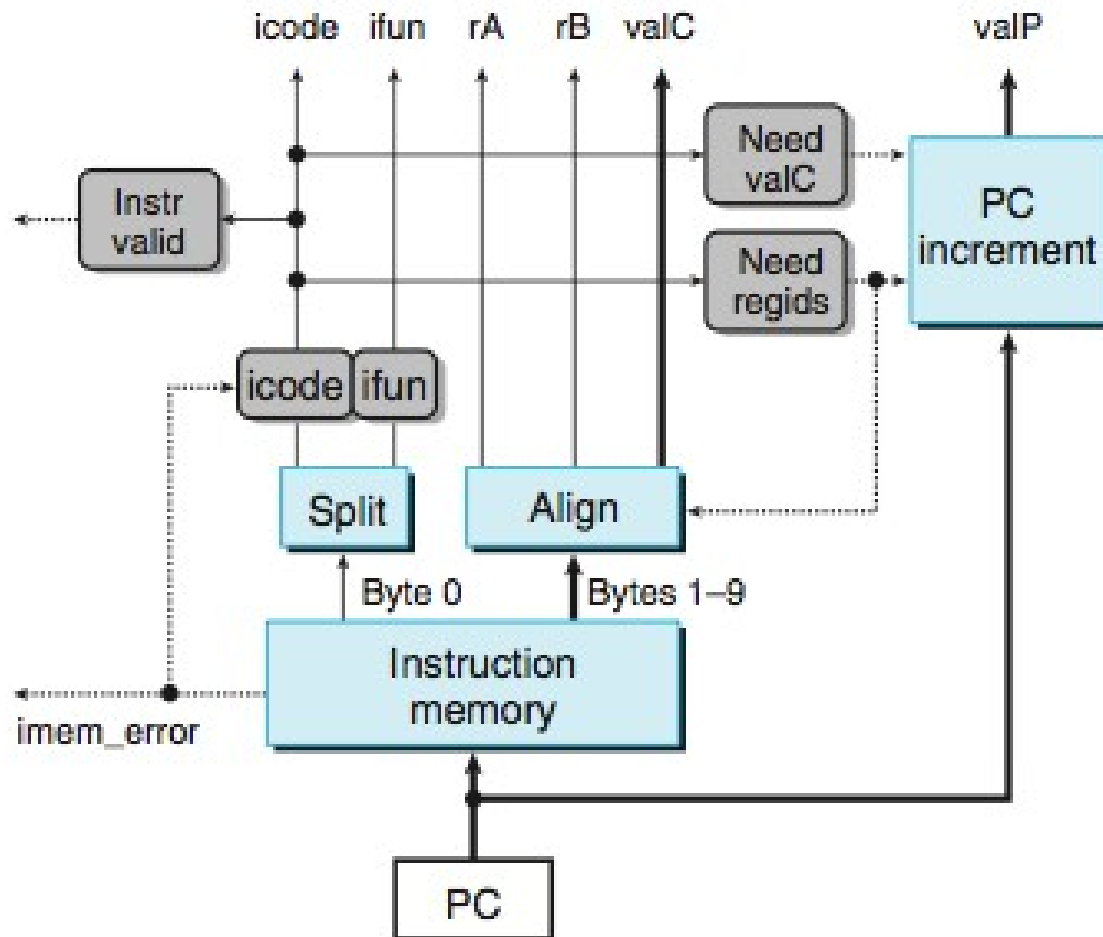


Stages of Sequential (SEQ) Processing Cycle:

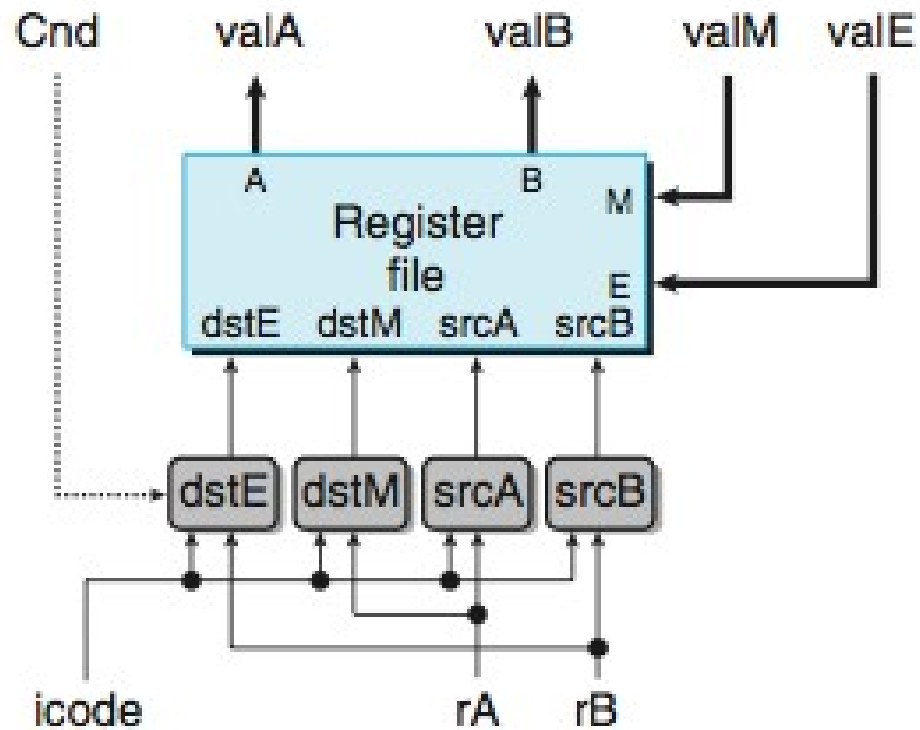
- 1) CPU **fetches** the instruction, increments instruction pointer (program counter)
- 2) CPU **decodes** the instruction, if operands involved, they are fetched
- 3) CPU **executes** the instruction, updates status flags such as zero, carry, and overflow
- 4) CPU **stores** the result to memory



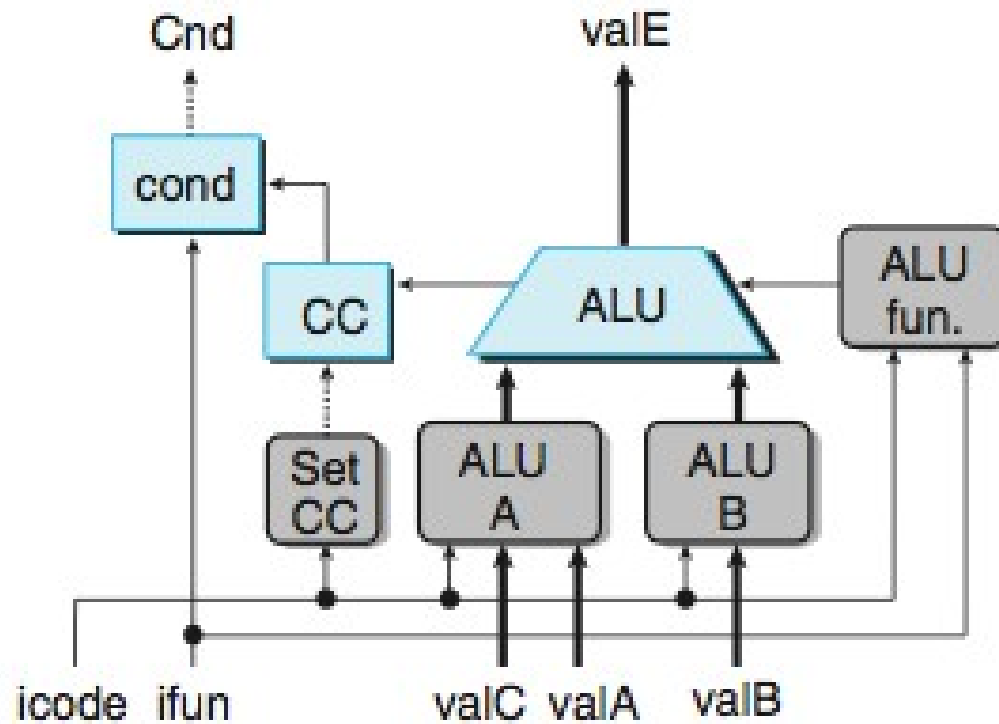
Fetch Stage



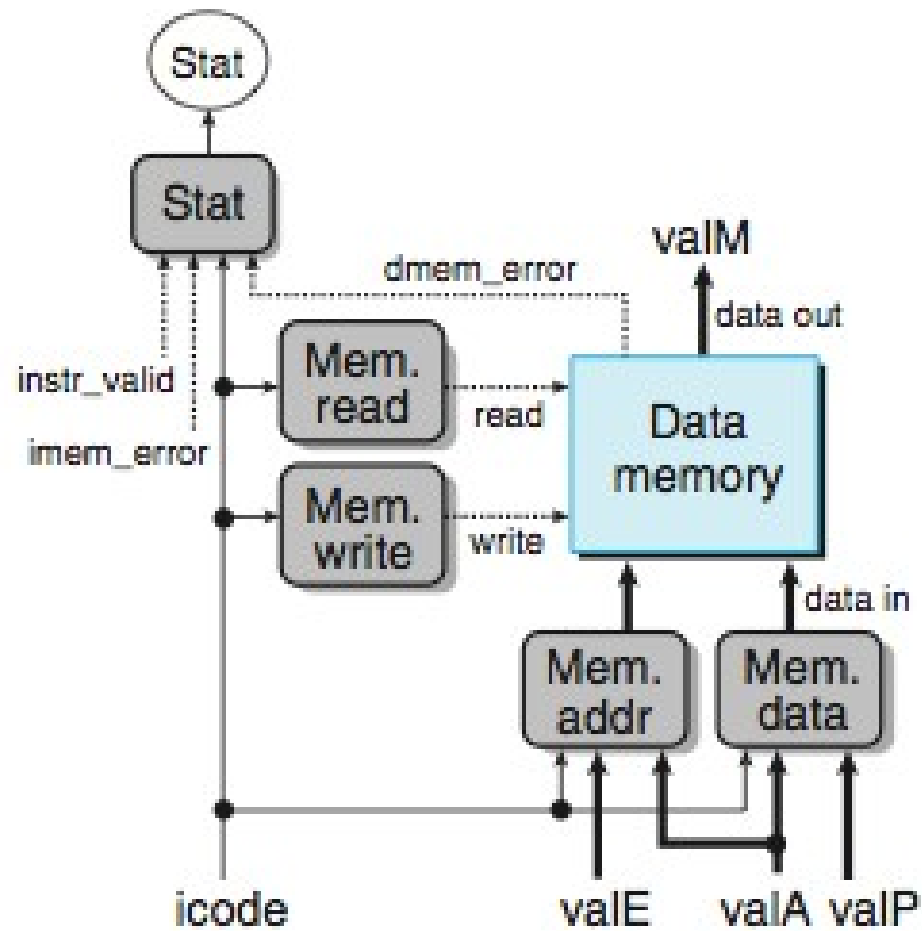
Decode Stage



Execute Stage



Memory Stage



How Does a CPU work?

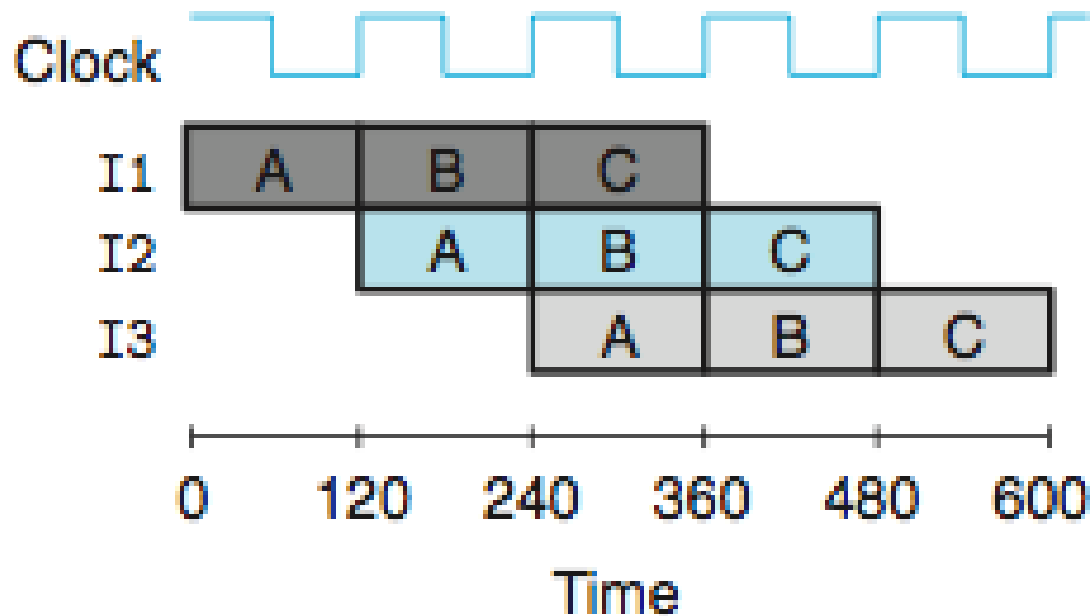
https://www.youtube.com/watch?v=cNN_tXABUA

<http://visual6502.org/JSSim/index.html>

Pipelining

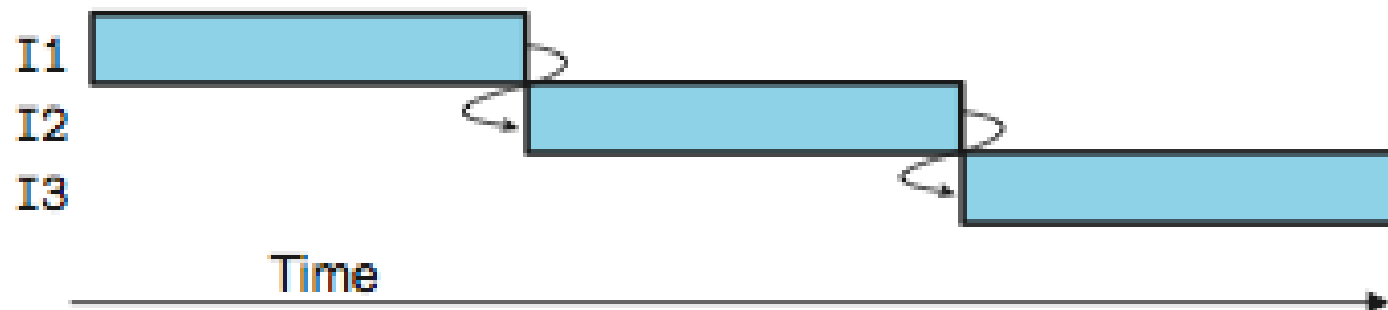
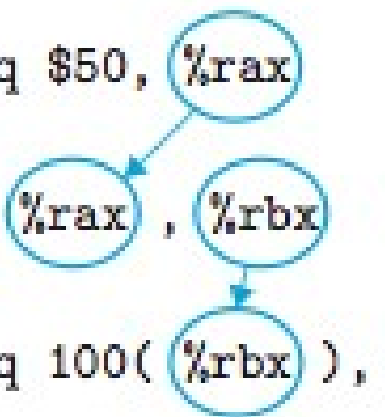
The problem with sequential processing (SEQ) is that it is too slow. The clock must run slowly enough so that signals can propagate through all of the stages within a single cycle. This style of implementation does not make very good use of our hardware units, since each unit is only active for a fraction of the total clock cycle. We will see that we can achieve much better performance by introducing pipelining.

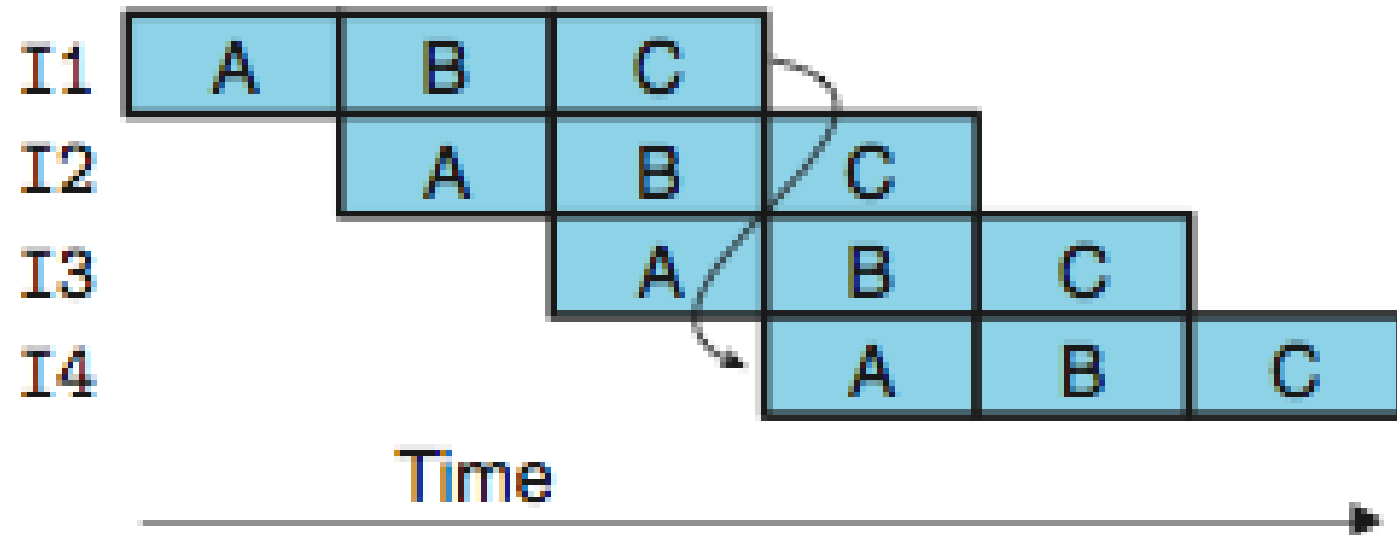
A 3-stage pipeline is 2.7x as fast

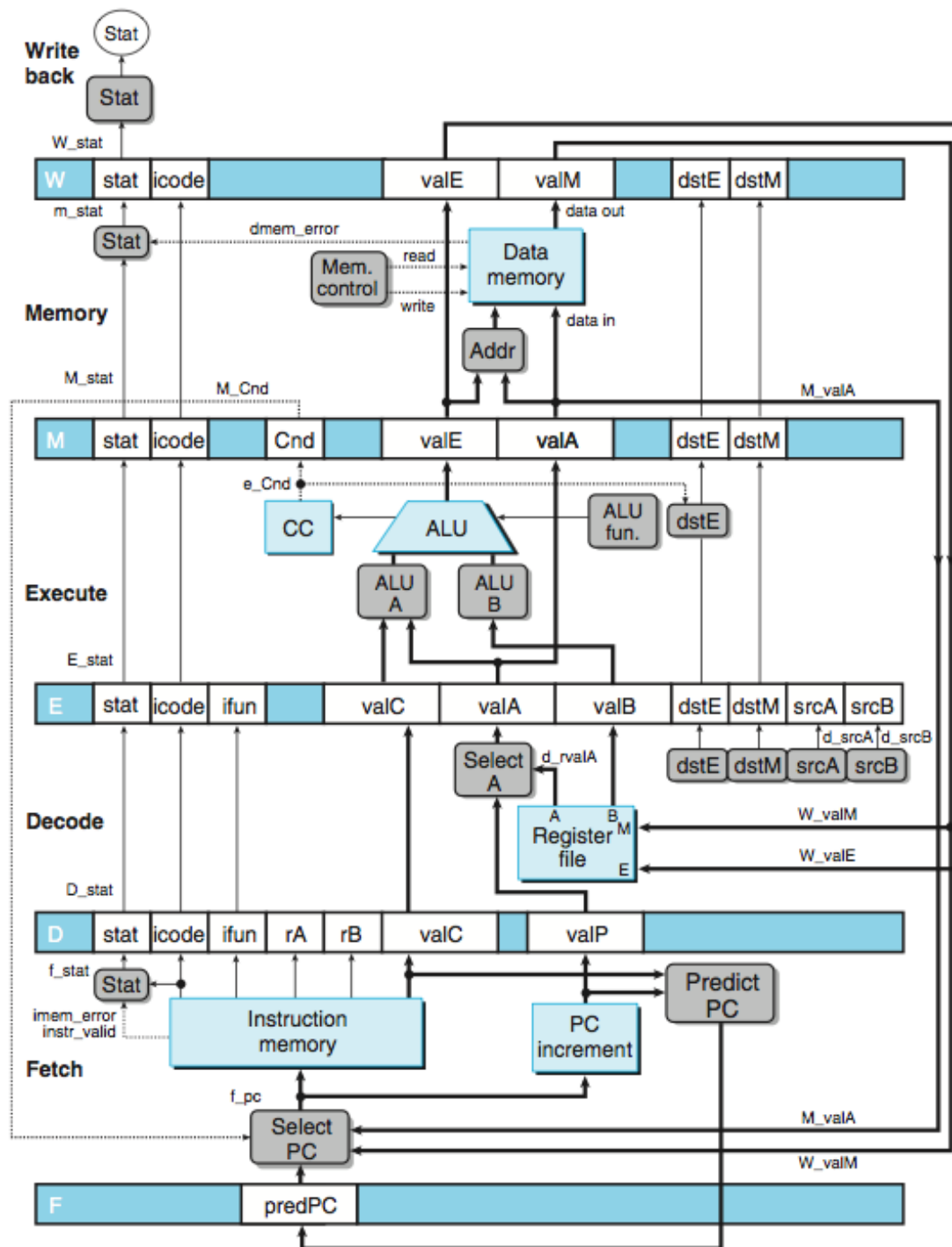


Pipelining a system with feedback

```
1  irmovq $50, %rax
2  addq  %rax, %rbx
3  mrmovq 100( %rbx ), %rdx
```



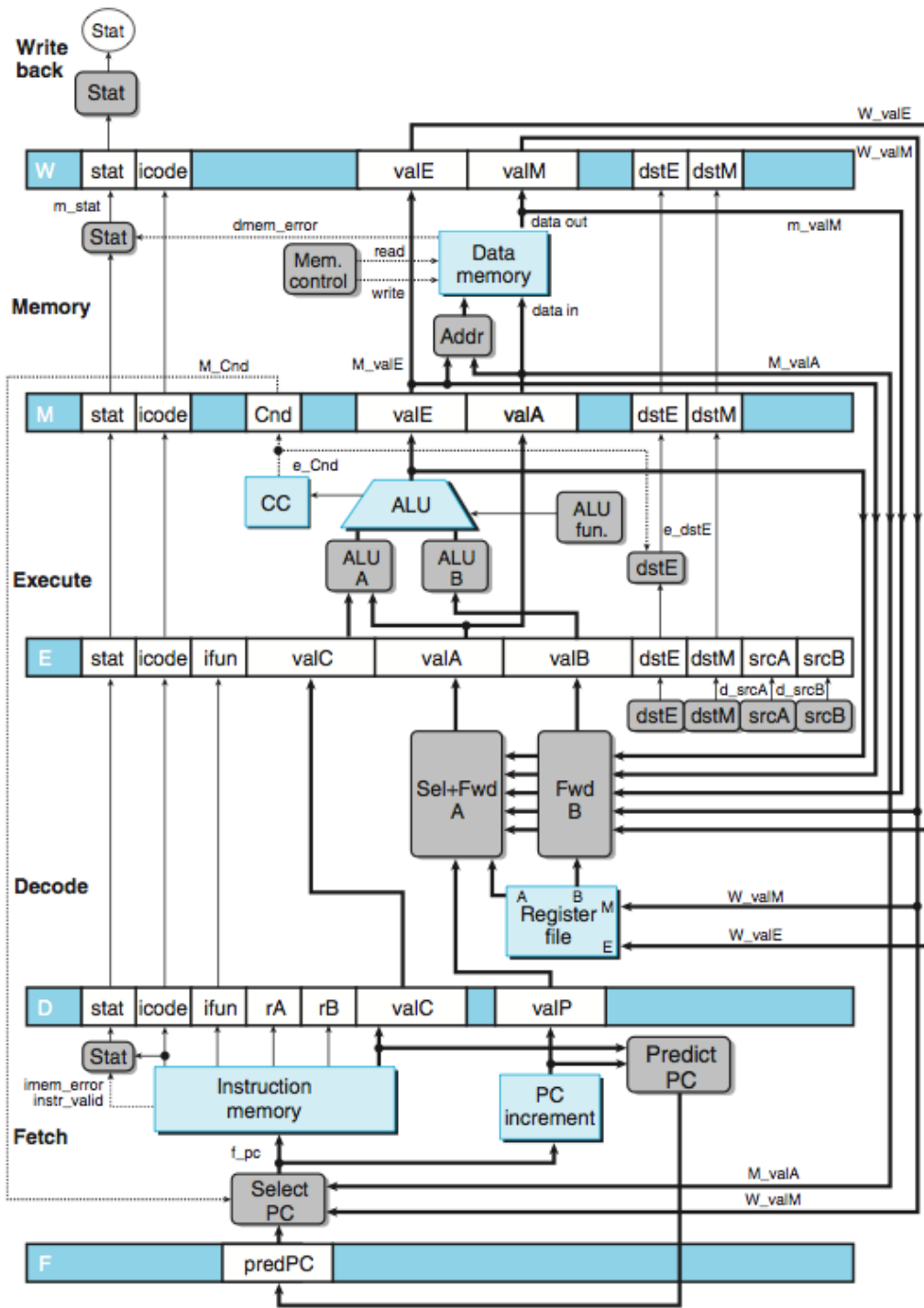




introducing pipelining into a system with feedback can lead to problems when there are dependencies between successive instructions. We must resolve this issue before we can complete our design. These dependencies can take two forms: (1) data dependencies, where the results computed by one instruction are used as the data for a following instruction, and (2) control dependencies, where one instruction determines the location of the following instruction, such as when executing a jump, call, or return. When such dependencies have the potential to cause an erroneous computation by the pipeline, they are called *hazards*.

One very general technique for avoiding hazards involves *stalling*, where the processor holds back one or more instructions in the pipeline until the hazard condition no longer holds. Our processor can avoid data hazards by holding back an instruction in the decode stage until the instructions generating its source operands have passed through the write-back stage.

Rather than stalling until a write has completed, the pipe can pass the value that is about to be written to pipeline register as the source operand. This technique of passing a result value directly from one pipeline stage to an earlier one is commonly known as *data forwarding* (or simply *forwarding*, and sometimes *bypassing*)



This use of a stall to handle a load/use hazard is called a *load interlock*. Load interlocks combined with forwarding suffice to handle all possible forms of *data hazards*.

Control hazards arise when the processor cannot reliably determine the address of the next instruction based on the current instruction in the fetch stage. Control hazards can only occur in our pipelined processor for ret and jump instructions. Moreover, the latter case only causes difficulties when the direction of a conditional jump is mis-predicted.