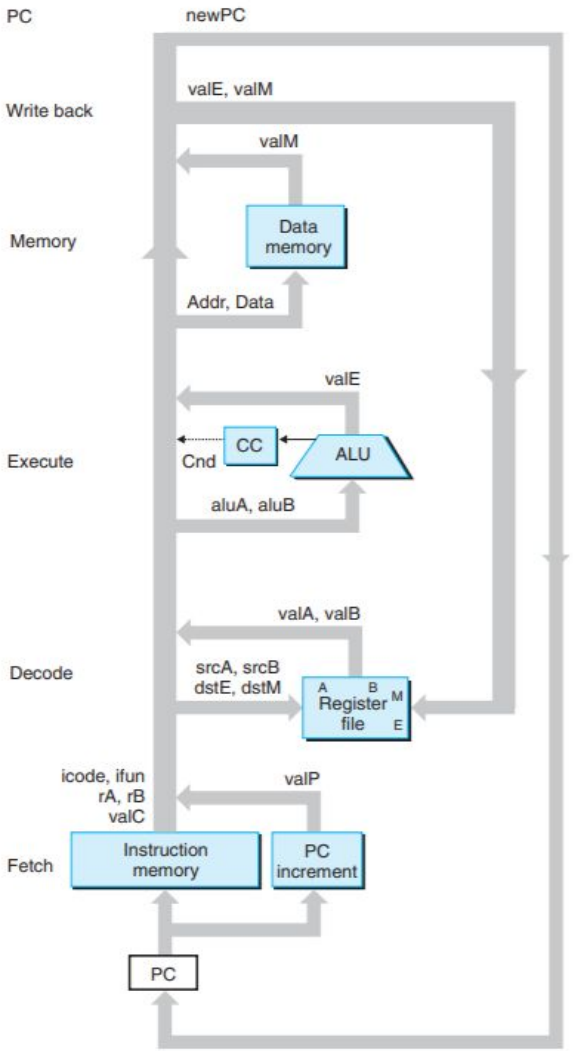
Tutorial 6

Slides by Prateek Alat

Steps for Instruction Execution



1. **Fetch** -
   a. Read the bytes of an instruction from memory.
   b. May fetch a register specifier byte, giving **rA** and **rB**.
   c. Address of the next instruction **valP** is calculated as [**PC** + length of fetched instruction].
2. **Decode** - Read upto 2 operands (denoted as **valA** and **valB**) from the register file or a constant word **valC** depending on the instruction interpreted.
3. **Execute** - ALU performs the operation specified (depending on value of **ifun**). The operation's resulting value is denoted as **valE**.
4. **Memory** - Data may be read-from or written-to memory. We refer to the value read as **valM**.
5. **Write Back** - The 2 results (**valE**, **valM**) are written back to the register file.
6. **PC Update** - The **PC** is set to the address of the next instruction (i.e. **valP**).

Example Implementations

Stage	`OPl` rA, rB	`rrmovl` rA, rB	`irmovl` V, rB
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC + 1] valP ← PC + 2	icode:ifun ← M_1[PC] rA:rB ← M_1[PC + 1] valP ← PC + 2	icode:ifun ← M_1[PC] rA:rB ← M_1[PC + 1] valC ← M_4[PC + 2] valP ← PC + 6
Decode	valA ← R[rA] valB ← R[rB]	valA ← R[rA]	
Execute	valE ← valB OP valA Set CC	valE ← 0 + valA	valE ← 0 + valC
Memory			
Write back	R[rB] ← valE	R[rB] ← valE	R[rB] ← valE
PC update	PC ← valP	PC ← valP	PC ← valP

Figure 4.18 **Computations in sequential implementation of Y86 instructions** `OPl`, `rrmovl`, **and** `irmovl`. These instructions compute a value and store the result in a register. The notation icode : ifun indicates the two components of the instruction byte, while rA : rB indicates the two components of the register specifier byte. The notation $M_1[x]$ indicates accessing (either reading or writing) 1 byte at memory location x, while $M_4[x]$ indicates accessing 4 bytes.

Sequential implementations of **rrmovl** and **irmovl**

Stage	`pushl` rA	`popl` rA
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC + 1]	icode:ifun ← M_1[PC] rA:rB ← M_1[PC + 1]
	valP ← PC + 2	valP ← PC + 2
Decode	valA ← R[rA] valB ← R[%esp]	valA ← R[%esp] valB ← R[%esp]
Execute	valE ← valB + (−4)	valE ← valB + 4
Memory	M_4[valE] ← valA	valM ← M_4[valA]
Write back	R[%esp] ← valE	R[%esp] ← valE R[rA] ← valM
PC update	PC ← valP	PC ← valP

Figure 4.20 **Computations in sequential implementation of Y86 instructions `pushl` and `popl`.** These instructions push and pop the stack.

pushl and **popl** additionally involve **valM**.

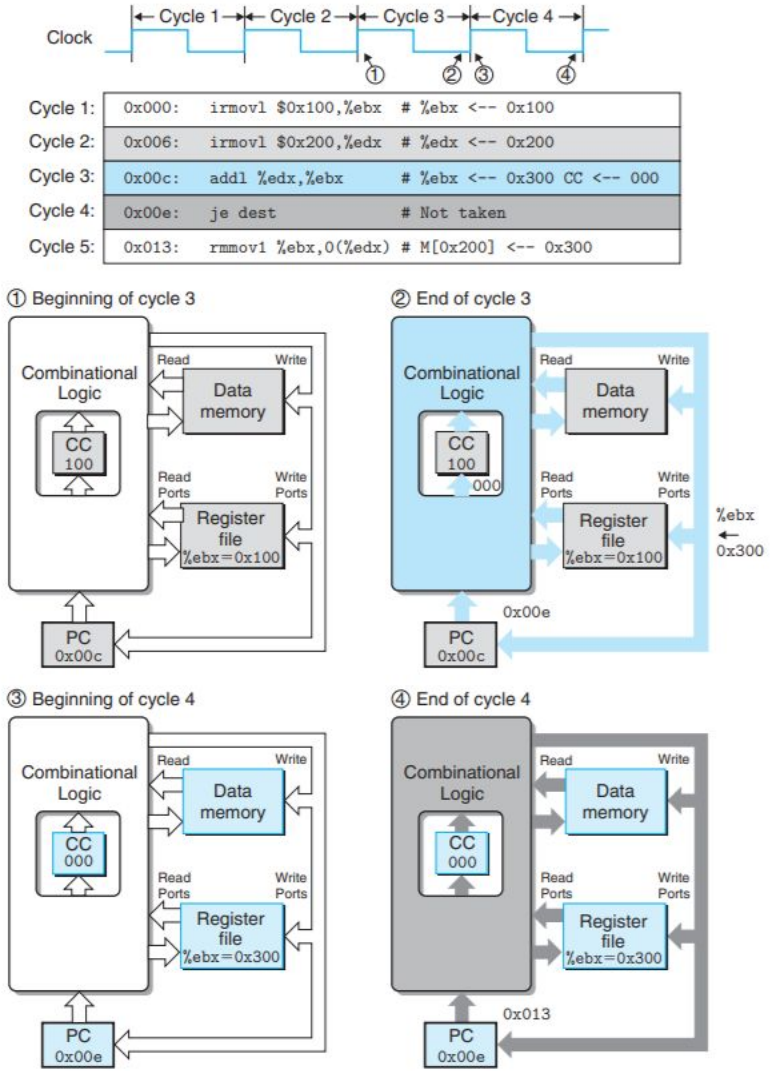
Note that "*the processor never needs to read back the state updated by an instruction in order to complete the processing of this instruction*." (pg. **380**)

The above property ensures that each instruction requires only **1 clock cycle** to execute, since there is no dependency on the previous state.

If there was a dependency on a previous value, then we'd have to wait for another clock cycle since the registers update only once per clock cycle.

For implementations of most other instructions, go to page **366** of the textbook.

SEQ Timing



At the rising edge of a clock cycle, the memory and registers are updated with the values computed by the previous instruction.

During a clock cycle, the combinational logic executes the **logical part** of the current instruction (memory is untouched).

By the end of a clock cycle, the combinational logic must have already finished executing.

The following analysis begins at the end of the 2nd instruction. Registers and combinational logic have been color-coded to identify the instruction which they are currently associated with.

1. **Registers** store instruction 2's values.
2. The **combinational logic** has finished executing instruction 3. The **registers** haven't been modified yet.
3. The **registers** have been modified to store instruction 3's values.
4. The **combinational logic** finished executing instruction 4. **Registers** still store values of instruction 3.