A final part of the program state is a status code Stat, indicating the overall state of program execution. It will indicate either normal operation or that some sort of *exception* has occurred, such as when an instruction attempts to read from an invalid memory address. The possible status codes and the handling of exceptions is described in Section 4.1.4.

set of Y86-64 instructions is largely a subset of the x86-64 instruction set. It includes only 8-byte integer operations, has fewer addressing modes, and includes a smaller set of operations. Since we only use 8-byte data, we can refer to these as "words"

Here are some details about the Y86-64 instructions:

- The x86-64 `movq` instruction is split into four different instructions: `irmovq`, `rrmovq`, `mrmovq`, and `rmmovq`, explicitly indicating the form of the source and destination. The source is either immediate (`i`), register (`r`), or memory (`m`).

As with x86-64, we do not allow direct transfers from one memory location to another. In addition, we do not allow a transfer of immediate data to memory.

- There are four integer operation instructions, shown in Figure 4.2 as `OPq`. These are `addq`, `subq`, `andq`, and `xorq`. They operate only on register data, whereas x86-64 also allows operations on memory data. These instructions set the three condition codes `ZF`, `SF`, and `OF` (zero, sign, and overflow).

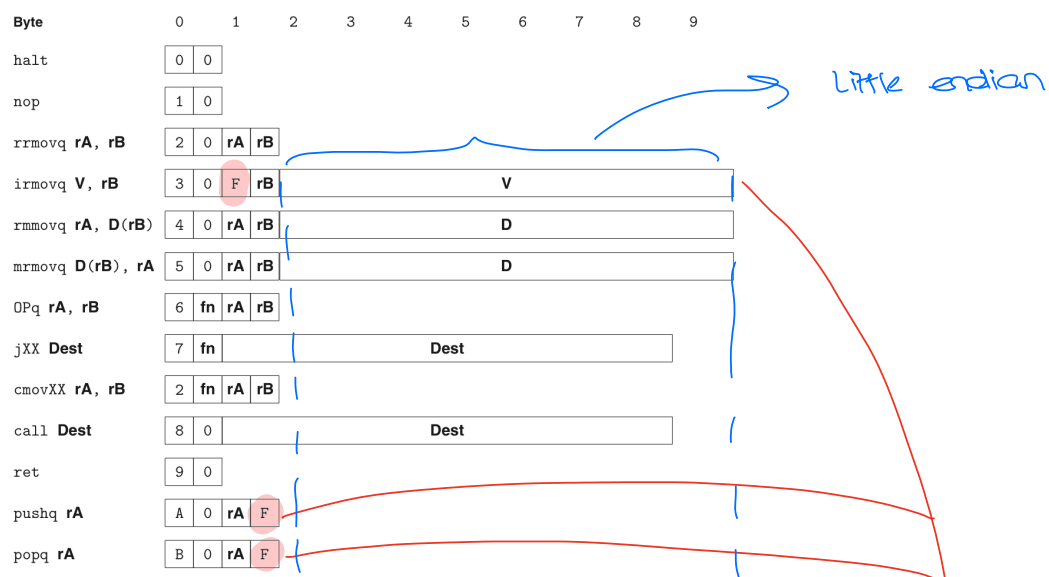

Figure 4.2 Y86-64 instruction set. Instruction encodings range between 1 and 10 bytes. An instruction consists of a 1-byte instruction specifier, possibly a 1-byte register specifier, and possibly an 8-byte constant word. Field `fn` specifies a particular integer operation (`OPq`), data movement condition (`cmovXX`), or branch condition (`jXX`). All numeric values are shown in hexadecimal.

- There are six conditional move instructions (shown in Figure 4.2 as `cmovXX`): `cmovle`, `cmovl`, `cmove`, `cmovne`, `cmovge`, and `cmovg`. These have the same format as the register–register move instruction `rrmovq`, but the destination register is updated only if the condition codes satisfy the required constraints.

- The `halt` instruction stops instruction execution. x86-64 has a comparable instruction, called `hlt`. x86-64 application programs are not permitted to use

Figure 4.2 also shows the byte-level encoding of the instructions. Each instruction requires between 1 and 10 bytes, depending on which fields are required. Every instruction has an initial byte identifying the instruction type. This byte is split into two 4-bit parts: the high-order, or *code*, part, and the low-order, or *function*, part. As can be seen in Figure 4.2, code values range from `0` to `0xB`. The function

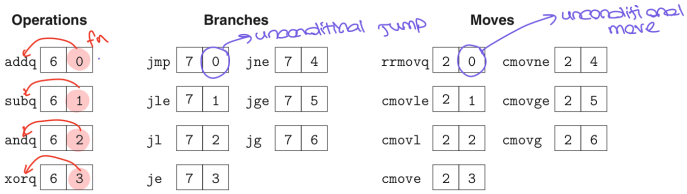

Figure 4.3 Function codes for Y86-64 instruction set. The code specifies a particular integer operation, branch condition, or data transfer condition. These instructions are shown as `OPq`, `jXX`, and `cmovXX` in Figure 4.2.

an address computation, depending on the instruction type. Instructions that have no register operands, such as branches and `call`, do not have a register specifier byte. Those that require just one register operand (`irmovq`, `pushq`, and `popq`) have the other register specifier set to value `0xF`. This convention will prove useful in our processor implementation.

- The Y86-64 code loads constants into registers (lines 2–3), since it cannot use immediate data in arithmetic instructions.

Fetch. The fetch stage reads the bytes of an instruction from memory, using the program counter (PC) as the memory address. From the instruction it extracts the two 4-bit portions of the instruction specifier byte, referred to as icode (the instruction code) and ifun (the instruction function). It possibly fetches a register specifier byte, giving one or both of the register operand specifiers rA and rB. It also possibly fetches an 8-byte constant word valC. It computes valP to be the address of the instruction following the current one in sequential order. That is, valP equals the value of the PC plus the length of the fetched instruction.

Decode. The decode stage reads up to two operands from the register file, giving values valA and/or valB. Typically, it reads the registers designated by instruction fields rA and rB, but for some instructions it reads register `%rsp`.

Execute. In the execute stage, the arithmetic/logic unit (ALU) either performs the operation specified by the instruction (according to the value of ifun), computes the effective address of a memory reference, or increments or decrements the stack pointer. We refer to the resulting value as valE. The condition codes are possibly set. For a conditional move instruction, the stage will evaluate the condition codes and move condition (given by ifun) and enable the updating of the destination register only if the condition holds. Similarly, for a jump instruction, it determines whether or not the branch should be taken.

Memory. The memory stage may write data to memory, or it may read data from memory. We refer to the value read as valM.

Write back. The write-back stage writes up to two results to the register file.

PC update. The PC is set to the address of the next instruction.

Stage	`OPq rA, rB`	`rrmovq rA, rB`	`irmovq 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_8[PC + 2] valP ← PC + 10
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 Y86-64 instructions** `OPq`, `rrmovq`, **and** `irmovq`. 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 $M_8[x]$ indicates accessing 8 bytes.

Stage	`rmmovq rA, D(rB)`	`mrmovq D(rB), rA`
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC + 1] valC ← M_8[PC + 2] valP ← PC + 10	icode:ifun ← M_1[PC] rA:rB ← M_1[PC + 1] valC ← M_8[PC + 2] valP ← PC + 10
Decode	valA ← R[rA] valB ← R[rB]	valB ← R[rB]
Execute	valE ← valB + valC	valE ← valB + valC
Memory	M_8[valE] ← valA	valM ← M_8[valE]
Write back		R[rA] ← valM
PC update	PC ← valP	PC ← valP

→ valA burada kullanılmayacağı için mrmovq'da hesaplanmıyor ama rmmovq'da hesaplanıyor.

Figure 4.19 **Computations in sequential implementation of Y86-64 instructions** `rmmovq` **and** `mrmovq`. These instructions read or write memory.

Stage	Generic `irmovq V, rB`	Specific `irmovq $128, %rsp`
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC + 1] valC ← M_8[PC + 2] valP ← PC + 10	icode:ifun ← M_1[0x016] = 3:0 rA:rB ← M_1[0x017] = f:4 valC ← M_8[0x018] = 128 valP ← 0x016 + 10 = 0x020
Decode		
Execute	valE ← 0 + valC	valE ← 0 + 128 = 128
Memory		
Write back	R[rB] ← valE	R[%rsp] ← valE = 128
PC update	PC ← valP	PC ← valP = 0x020

This instruction sets register `%rsp` to 128 and increments the PC by 10.

```
4  0x016: 30f48000000000000000 |     irmovq $128,%rsp         # Problem 4.13
5  0x020: 4043640000000000000 |     rmmovq %rsp, 100(%rbx)   # store
```

Stage	`pushq rA`	`popq rA`
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
Decode	valA ← R[rA] valB ← R[%rsp]	valA ← R[%rsp] valB ← R[%rsp]
Execute	valE ← valB + (−8)	valE ← valB + 8
Memory	M_8[valE] ← valA	valM ← M_8[valA]
Write back	R[%rsp] ← valE	R[%rsp] ← valE R[rA] ← valM
PC update	PC ← valP	PC ← valP

Figure 4.20 Computations in sequential implementation of Y86-64 instructions `pushq` **and** `popq`. These instructions push and pop the stack.

Stage	`jXX Dest`	`call Dest`	`ret`
Fetch	icode:ifun ← M_1[PC] valC ← M_8[PC + 1] valP ← PC + 9	icode:ifun ← M_1[PC] valC ← M_8[PC + 1] valP ← PC + 9	icode:ifun ← M_1[PC] valP ← PC + 1
Decode		valB ← R[%rsp]	valA ← R[%rsp] valB ← R[%rsp]
Execute	Cnd ← Cond(CC, ifun)	valE ← valB + (−8)	valE ← valB + 8
Memory		M_8[valE] ← valP	valM ← M_8[valA]
Write back		R[%rsp] ← valE	R[%rsp] ← valE
PC update	PC ← Cnd ? valC : valP	PC ← valC	PC ← valM

Figure 4.21 Computations in sequential implementation of Y86-64 instructions `jXX`, `call`, **and** `ret`. These instructions cause control transfers.

Write a program that returns 1 when little endian and 0 when big endian

```
union {
    int i;
    float f;
} a;
```

```
a.i = Tmin
print a.f ?
```

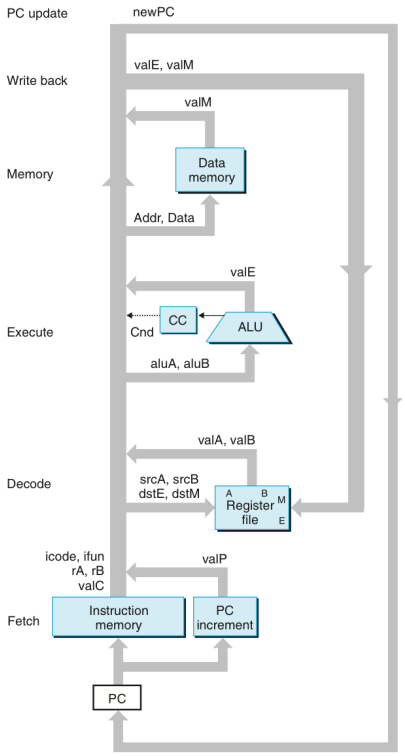

Figure 4.22 Abstract view of SEQ, a sequential implementation. The information processed during execution of an instruction follows a clockwise flow starting with an instruction fetch using the program counter (PC), shown in the lower left-hand corner of the figure.

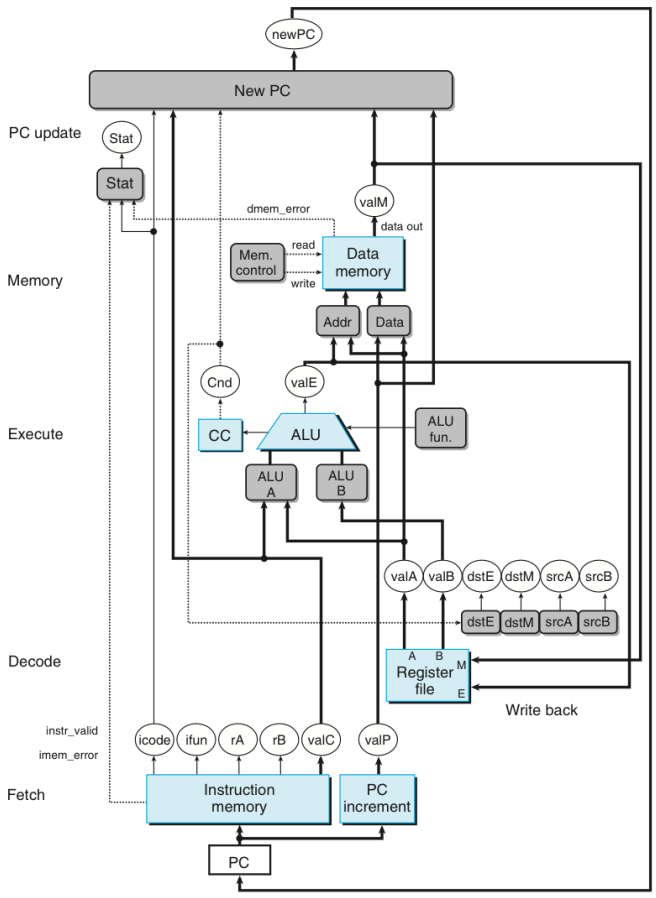

Figure 4.23 Hardware structure of SEQ, a sequential implementation. Some of the control signals, as well as the register and control word connections, are not shown.

random access memories. The program counter is loaded with a new instruction address every clock cycle. The condition code register is loaded only when an integer operation instruction is executed. The data memory is written only when an `rmmovq`, `pushq`, or `call` instruction is executed. The two write ports of the register file allow two program registers to be updated on every cycle, but we can use the special register ID `0xF` as a port address to indicate that no write should be performed for this port.

rA, rB

PRINCIPLE: No reading back

The processor never needs to read back the state updated by an instruction in order to complete the processing of this instruction. ■

begin the next clock cycle.

As another illustration of this principle, we can see that some instructions (the integer operations) set the condition codes, and some instructions (the conditional move and jump instructions) read these condition codes, but no instruction must both set and then read the condition codes. Even though the condition codes are not set until the clock rises to begin the next clock cycle, they will be updated before any instruction attempts to read them.

Figure 4.27
SEQ fetch stage. Six bytes are read from the instruction memory using the PC as the starting address. From these bytes, we generate the different instruction fields. The PC increment block computes signal valP.

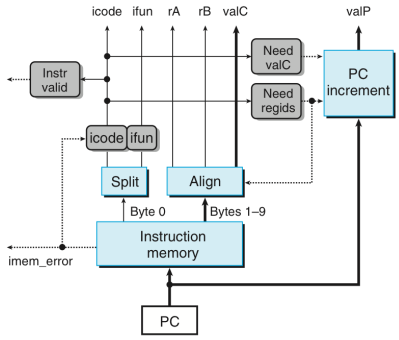

Figure 4.28
SEQ decode and write-back stage. The instruction fields are decoded to generate register identifiers for four addresses (two read and two write) used by the register file. The values read from the register file become the signals valA and valB. The two write-back values valE and valM serve as the data for the writes.

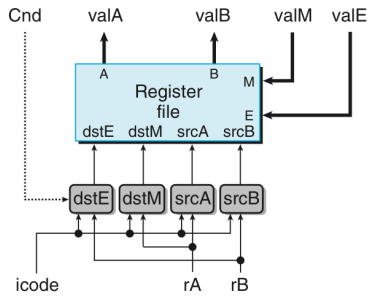

Figure 4.29
SEQ execute stage. The ALU either performs the operation for an integer operation instruction or acts as an adder. The condition code registers are set according to the ALU value. The condition code values are tested to determine whether a branch should be taken.

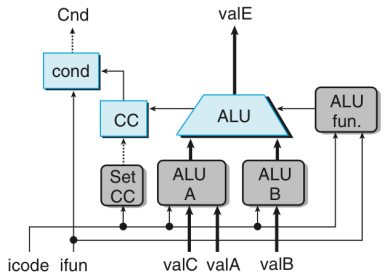

Figure 4.30
SEQ memory stage. The data memory can either write or read memory values. The value read from memory forms the signal valM.

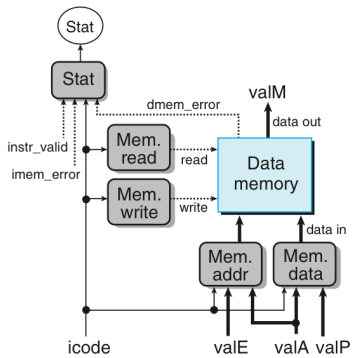

Figure 4.31
SEQ PC update stage. The next value of the PC is selected from among the signals valC, valM, and valP, depending on the instruction code and the branch flag.