CS223: Hardware Lab

Assignment 4(A) – Part I

Group 16

**AIM:**

To design a processor in Verilog/VHDL and simulate it using Xilinx ISE. Then download this design into FPGA kit. Processor consists of six 8-bit registers (B, C, D, E, F, G) and one Accumulator register. It should perform 4 arithmetic and 4 logical operations.

**MODULES:**

* Accumulator
* ALU
* Flag register
* Instruction register
* Memory Address Register(MAR)
* Memory Buffer Register(MBR)
* Program Counter(PC)
* RAM (Main Memory)
* Register file (consisting of 6 general purpose registers, excluding accumulator)
* Stack Pointer
* Temporary Register
* Controller

**INSTRUCTION SET:**

Each instruction consists of 8-bits of data. For each instruction, first 2 bit consists of opcode.

Format of different types of instructions:

**1. Data Movement:**

MOV dst src

Move data from dst to src.



First 2 bit consists of opcode. Opcode for this is 00. For data movement from one register to another registe, middle 3 bit consists of address for destination register and last 3 bit consists of address for source register.

For data movement from one register to memory location and vice versa. Since there are 7 registers, one 3 bit code(i.e 111) has not been assigned to a register.We can use this code to access memory location(for both dst and src).For e.g if dst is 111 and src is the address of some register. MAR is loaded with the contents of F and G registers and MBR is loaded with the contents of src register with RAM being write enabled.

e.g 00001002

This instruction moves data from register C to register B.

**2.Arithmetic and Logical operations(Single word format):**

The general structure of this instruction is : 01-op-src

Perform the operation (OPR) on data of src and accumulator and store the result in accumulator. Here, src can be register or memory location. Again, first 2-bits consists of opcode and it's opcode is 01. Next 3-bit is code for operation to be performed and last 3-bit is for the address of source register.

Again to access memory location, src is given 111. It will retrieve the address from registers F and G.

e.g 01 000 001

Suppose accumulator stores 5and register B stores 5 . Above instruction will add the contents of accumulator and src register and store the result in accumulator. Thus, accumulator will store 00001010.

**3. Immediate addressing mode:**

Opcode for this type of instruction is 10.

The third bit determines if the immediate operation is for the data movement or ALU operations. If third bit == 0 then it denotes immeditate instruction for data movement. If it is equal to 1 then it denotes for ALU operation.

In case of data movement, the 4th and 5th bit are don’t care and the last three bit of the first word denotes the destination register.

In case of ALU operation, 4th, 5th and 6th bits denotes function code:

000 – Addition

001 – Subtration

010 – Multiplication

011 – Division

100 – NOT Logical Operation

101 – AND Operation

110 – OR Operation

111 – XOR Operation

Since this is a 2-word instruction, the value of the immediate constant is specified in the next word.

**4. Conditional instruction, PUSH, POP:**

Opcode for this type of instruction is 11.

The third bit decides whether it is a JUMP instruction (3rd bit == 0) or a PUSH/POP instruction (3rd bit == 1).

In case of JUMP instruction, 4th and 5th bit decides what type of JUMP condition is to be checked.

00 – JUMP as per label.

01 – JUMP if accumulator = 0

10 – JUMP if accumulator > 0

11 – JUMP if accumulator < 0

Next three bits are don’t care.

PUSH/POP instructions are one-word instruction. 4th and 5th bit decides whether it is a PUSH or a POP instruction.

00 – PUSH

01 – POP

10 and 11 – NoOP

The next three bits decides value from which register is to be pushed or to which register the value has to be popped.

**Structure of the processor:**

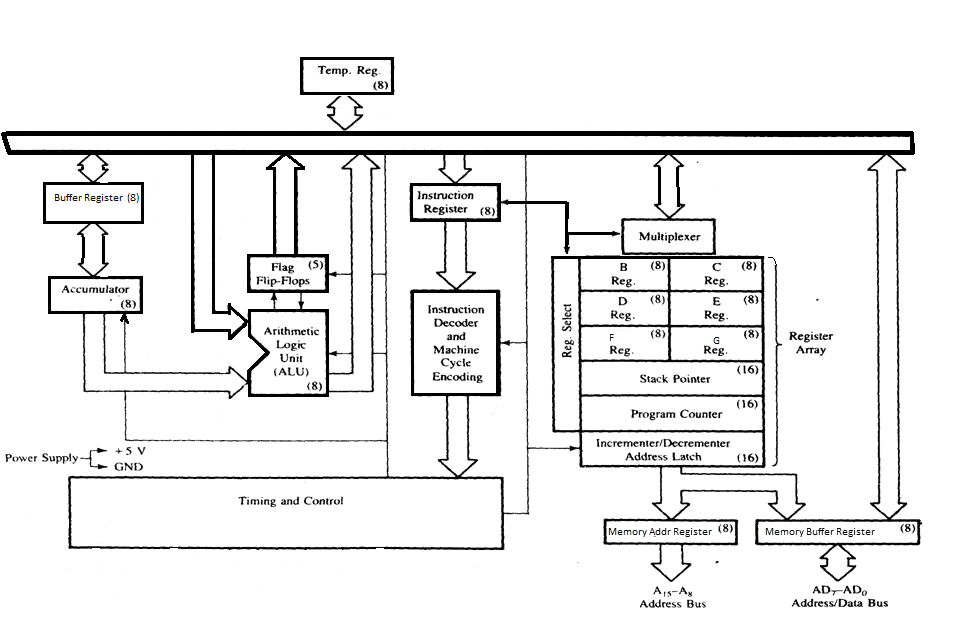


Figure 1: Architecure of Processor

**Design issues of the processor:**

1. Since we have 8-bit data bus but memory address consists of 16-bit. So, to access memory location, it takes 2 cycles.

2. Input for ALU is two 8-bit numbers and it's output is also 8-bit.This creates problems for multiplication operation as it's output can be as large as 16-bit.

3. To access memory location we use registers F and G rather than directly accessing it from RAM. This again leads to increase in time cycles and thus, decreases the performance.

4. We believe that under the constraints of our datapath and the size of the instruction we have usitilized our resources quite optimally giving lots of options to the programmer to make their code run faster.

5. To enable debugging, we are providing clock through a button. This leads to inefficient mapping in the Xilinx software. Release mode implementations of course should use the inbuilt clock.

6. We added a temporary register to the datapath after much deliberation. This addition enabled easier implementation of multi-word instructions (to be implemented in the future). Further inter-register file movements are much easier and require significantly less circuitry. This makes sense when the primary objective is to minimize area.

7. We believe that it may not be a very good idea to have complex instructions like multiplication and division in our simplistic processor. The reasons for this are twofold. First, this could increase the minimum possible clock width making the processor slower overall. Second, they consume a lot of area, while processors like this are probably likely to be used in embedded applications, such instructions may be superfluous. Further the results of such instructions are larger than 8-bits and the simplistic datapath does not have the bandwidth to include both as is done in modern architectures such as the IA-32/64 and MIPS-32/64 (this is done by splitting the results into two answers storing them in separate registers. However our data bus is too small to accommodate this and additional circuitry to circumvent this problem only increases the complexity). In its place bitwise shifting may be more appropriate.

8. We have included MAR and MBR to access memory even though they are not required and only increase the number of clock cycles required. This will help however, in case we decide to do slightly asynchronous memory access and other optimizations (like a premature load of the next instruction) in the future. We believe it is good design practice to include this feature.

9. Note that although a simple up-counter would have sufficed for the PC (and later the SP), the Xilinx software was unable to make this optimization and ended up using and adder/subtractor which consumes more resources. Manual optimization may be required here to get a more optimal design.

10. As an advanced optimization feature we could use a very small clock cycle to generate larger, variable length clock cycles so that the various operations are given the amount of time they require and no more.

11. We have adopted the principle of homogenous and simple instructions to simplify our design and produce more area and time efficient circuits. For instance the first two bits always determine the type of operation. Registers are always addressed by three bit triplets uniformly across all instructions.

12. Thanks to the sophisticated optimization algorithms applied in the Xilinx software we were able to concentrate on the design aspects while we rest assured that the best combinatorial circuits and layouts were being chosen for us under the hood. For instance, the state encoding for our FSM in the control unit was intelligently chosen to minimize associated circuitry.

**Special Consideration:**

1. All the processes and changes take place at the positive-edge triggering of the clock except two. Writing of the IR (Instruction Register) and the buffer register takes place at the negative-edge triggering of the clock.
2. CALL and RETURN are pseudo instructions. In case of RETURN instruction, the values in register B, C, D, E gets retained whereas in case of CALL instruction, all the values are retained.
3. **To prevent the learning of the format of machine code for each instruction, we have designed a translator (in python) that converts the assembly code written in a text file and generates machine code for it. This machine code can be then directly loaded into simulator.**
4. Pseudo instruction for MOD operation can be implemented.

**Structure of control unit:**

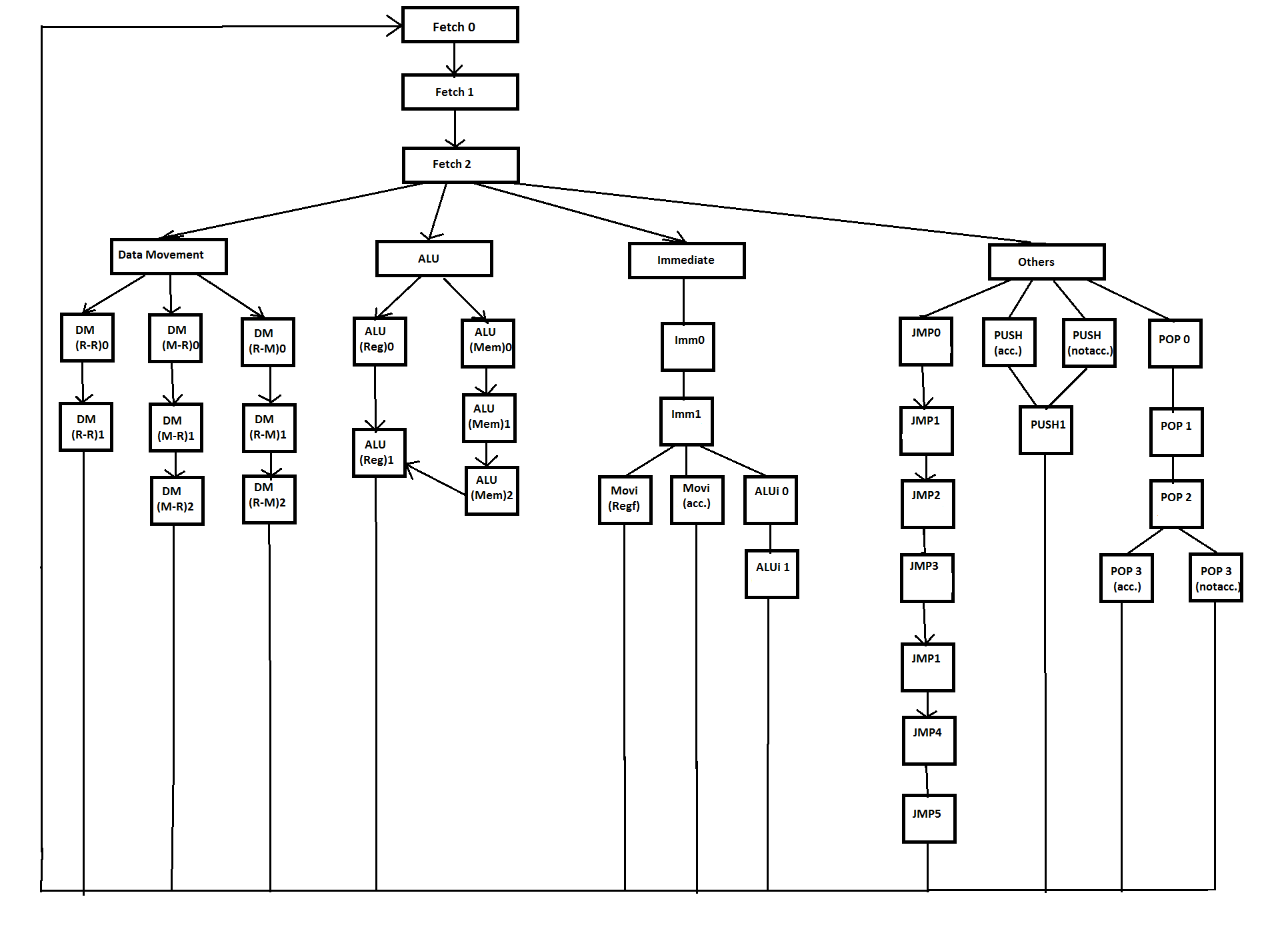


Figure 2: Graphical Depiction of Control FSM (with state labels)

**Control Signals:**

1. *acc\_en* : Accumulator Enable signal
2. *acc\_rw*: Accumulator read/write(0 for read and 1 for write) signal
3. *acc\_src*: Chooses between alu src and data-bus src
4. *alu\_op*: Chooses ALU function (by default, uses tri state buffer to send high impedance signal)
5. *ir\_rw*: Instruction register read/write(0 for read and 1 for write) signal
6. *mar\_rw*: Memory address register read/write(0 for read and 1 for write) signal
7. *mbr\_en*: Memory buffer register enable signal
8. *mbr\_rw*: Memory buffer register read/write(0 for read and 1 for write) signal
9. *mbr\_src*: Chooses between ALU src and data-bus src for MBR
10. *pc\_en*: Program counter enable signal
11. *pc\_rw*: Program counter read/write(0 for read and 1 for write)
12. *pc\_ld* : Specifies whether to load lower byte of pc or higher byte (1 loads higher byte)
13. *pc\_inc* : increments the Program counter
14. *ram\_en*: RAM enable
15. *ram\_rw*: RAM read/write(0 for read and 1 for write)
16. *regf\_en*: Register file enable
17. *regf\_rw*: Register file read/write
18. *regf\_addr* : specifies address of register to read/write. Also supports placing value of F-G pair when 111 is provided.
19. *sp\_en*: Stack pointer enable
20. *sp\_rw*: Stack pointer read/write(0 for read and 1 for write)
21. *sp\_ld* : specifies whether to load lower byte of stack pointer or higher byte (1 loads higher byte).
22. *tmp\_en*: Temporary register enable
23. *tmp\_rw*: Temporary register read/write(0 for read and 1 for write)
24. *tmp\_ld*: specifies whether to load lower byte of temporary register or higher byte (1 loads higher byte).
25. *res\_rw* : specifies when to store values of alu in result buffer

**Control steps for instructions:**

Each instruction follows the following cycle:

1. Fetch cycle: Here it fetches the data from memory. Fetch0, Fetch1, Fetch2 forms the fetch cycle as shown in structure of control unit.
2. Decode & Execute cycle: In this cycle instruction (in binary code) is decoded. The tree is split into 4 branches according to the opcode. 00 for Data movement, 01 for ALU operations, 10 for Immediate type operations and 11 for conditional operations.

**Fetch cycle:**

This cycle consists of 3 clock cycles.

1. In first clock cycle (Fetch0), value of Program counter(PC) is loaded into Memory Address register(MAR).
2. In second clock cycle(Fetch1), read the instruction from memory(RAM) as specified by the address in MAR and write the result in Memory Buffer register(MBR).
3. In third clock cycle (Fetch2), Load the value of MBR into Instruction Register(IR).
4. Value of PC is also incremented in this cycle.

**Decode and execute cycle:**

Initially the instruction is decoded from IR. Next state of control unit is defined by the opcode of the instruction. This can be seen in the structure of control unit.

**Case 1: Opcode = 00 (Data movement)**

1. Register-Register movement:
2. Initially (dm(reg->reg)0), contents of src register is stored in a Temporary register(tmp).
3. Next cycle (dm(reg->reg)1), contents of tmp register is transferred to dst register.
4. Next instruction is added as specified by PC.
5. Register-Memory movement:
6. In first clock cycle (dm(reg->mem)0), contents of src register is stored in a Temporary register(tmp).
7. In second clock cycle (dm(reg->mem)1), contents of tmp register is transferred to MBR and content of F-G pair is stored in MAR.
8. In last cycle (dm(reg->mem)2), contents of MBR is transferred to the memory(RAM) location as specified by MAR.
9. Next instruction is fetched as specified by PC.
10. Memory-Register movement:
11. Initially (dm(mem->reg)0), content of F-G pair is stored in MAR.
12. Next (dm(mem->reg)1), value stored in memory location(as specified by MAR) is fetched from RAM and stored in MBR.
13. In third clock cycle (dm(mem->reg)2), contents of MBR is transferred to dst register.
14. Next instruction is fetched as specified by PC.

**Case 2: Opcode = 01 (ALU operations)**

1. Fetching the data from registers:
2. In first clock cycle (alu(reg)0), value of operand register is transferred to data-bus and operation is performed by ALU. Other value of ALU comes from accumulator register which is connected to ALU. Operation by ALU doesn’t take whole clock cycle as it is a combinational circuit.
3. In next clock cycle (alu(reg)1), output of ALU is stored in accumulator.
4. Next instruction is fetched as specified by PC.
5. Fetching the data from memory location(as specified by F-G register pair):
6. Initially (alu(mem)0), the value of F-G pair is stored in MAR.
7. Next (alu(mem)1), read the memory location as specified by MAR and store it in MBR.
8. Now (alu(mem)2),transfer the value of MBR to data-bus and perform the ALU operation.
9. At last (alu(reg)1), store the output of ALU in accumulator register.
10. Next instruction is fetched as specified by PC.

**Case 3: Opcode = 10 (Immediate instructions)**

1. In first clock cycle (Imm0), value of PC is transferred to MAR.
2. Next clock cycle (Imm1), reads the memory location as specified by MAR and stores it in MBR.
3. A) If destination register is not an accumulator, then contents of MBR is transferred to dst register (Movi (Regf)). This is because control signals for accumulator are different from control signals for register file.

B) If destination register is accumulator, then contents of accumulator are transferred to accumulator.

C) If third bit of instruction is 1 then it is an ALUi operation. Next three bits are code for operation.

1. In this clock cycle (ALUi 0), value of MBR is transferred to data-bus which is again transferred to ALU for evaluation. Output is again transferred to buffer register.

2. Next clock cycle (ALUi 1), value of Buffer register is transferred to accumulator.

1. Next instruction is fetched as specified by PC.

**Case 4: Opcode = 11 (Other operations)**

Instruction code for opcode = 11 is specified in Intruction set section.

1. Case 1: 3rd bit = 0 (Jump instruction) Value of flag is taken in fetch cycle itself. Thus, all different types of jump instructions (J,JZ, JP, JN) will follow the same FSM.
2. First clock cycle (JMP 0), value of PC is transferred to MAR.
3. In second clock cycle (JMP 1), reads the memory location as specified by MAR and stores it in MBR. PC is incremented.
4. Next clock cycle (JMP 2), value of MBR is stored in tmp (temporary) register. This is done because current value of MBR will be used in subsequent clock cycles, but value of MBR will change.
5. In fourth clock cycle (JMP 3), value of PC is stored in MAR.
6. In this clock cycle (JMP 1), reading of memory is done and stored in MBR(over-writing previous value).
7. In sixth clock cycle (JMP 4), value of MBR is stored in lower byte of PC.
8. In last clock cycle (JMP 5), value stored in tmp register is transferred to higher byte of PC.
9. Next instruction is fetched as specified by PC.
10. Case 2: 3rd bit = 1 (PUSH and POP instruction)
11. PUSH (11-100-src):
12. In first clock cycle (PUSH (acc.) OR PUSH (notacc.)), depending on whether the src register is accumulator or not, value of src is transferred to MBR. Simultaneously value of Stack pointer(SP) register is transferred to MAR.
13. In next clock cycle (PUSH 1), value of SP is decremented. Value of MBR is stored in memory location as specified by MAR.
14. Next instruction is fetched as specified by PC.
15. POP (11-101-dst):
16. In first clock cycle (POP 0), value of SP is decremented.
17. Next (POP 1), value of SP is transferred to MAR.
18. In third clock cycle (POP 2), reads the memory location as specified by MAR and stores it in MBR.
19. In last clock cycle(POP 3(acc.) OR POP 3(notacc.)), depending on whether dst register is accumulator or not, value of MBR is transferred to dst register.
20. Next instruction is fetched as specified by PC.