Lecture 2
Basic Microprocessor

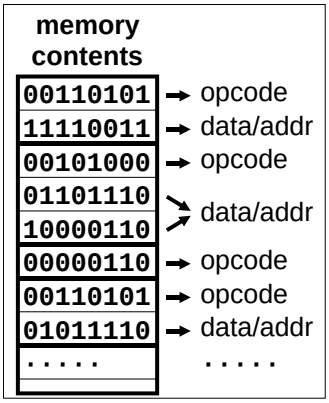
This lecture is a rapid review of microprocessor functionality and organization from a user's point of view.

The lecture contents provide an introduction to microprocessor basics for a first time user.

Microprocessor Functions

- **Execute** a series of instructions in memory.
- **Access memory** to move data between microprocessor registers and memory.
- **Calculate** or perform arithmetic and logic functions.
- **Compare:** Check for equality, less-than, greater-than conditions.
- **Branch:** Make decisions according to comparison results and call subprograms.
- **Communicate:** Send or receive data and control signals to/from peripheral devices.
- **Respond:** Take action in response to external events, such as user controls and peripheral device signals.

Machine Language

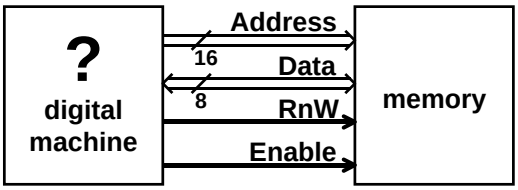


Operation code (**opcode** in short) tells microprocessor what to do. Opcode specifies the operation type (LOAD, ADD, JUMP, ...) and a method of accessing the data or address required for the operation.

After the opcode, microprocessor reads the following memory location(s) to retrieve the data or address information. An opcode together with this data or address information forms a **machine instruction**.

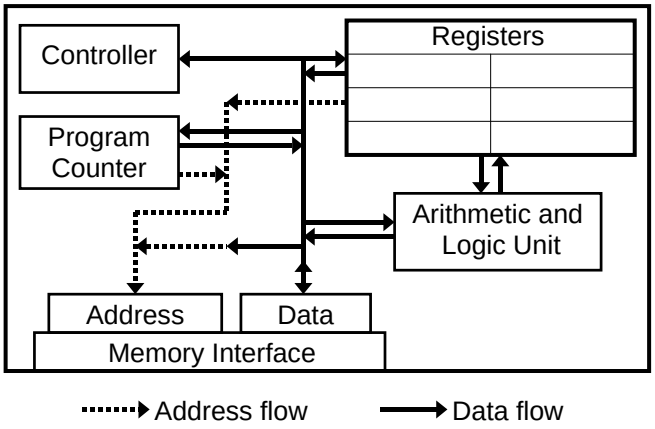
Focus on the first microprocessor function:
"Execute a series of instructions in memory**"**

You are given a memory device that stores the machine instructions. How can you build a machine that will execute these instructions?



Addr: address of the memory location to be accessed
Data: data read/written from/to the memory location
RnW: indicates data direction: 0=>write, 1=>read
Enable: controls timing of the read/write operation
(i.e. ALE=Address Latch Enable, CE=Chip Enable)

Microprocessor Organization

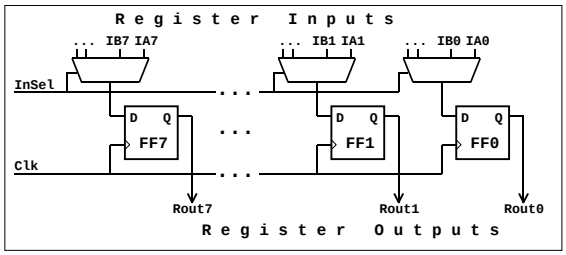


Program Counter (PC)

Program Counter is a special-purpose address register that keeps track of the memory address of the machine instruction to be executed. Following shows the typical utilization of PC to handle program flow:

- **Regular program flow:** Microprocessor incremets PC to the address of the next operation code.
- **Jump to another location:** Microprocessor loads the target address from memory into the PC.
- **Call a subprogram:** Microprocessor saves the incremented PC (address of the next operation code) at a safe memory location (usually stack) and loads the target subprogram address into the PC. PC value is restored when a "RETURN" instruction is executed at the end of the subprogram.

Register?



A register is a collection of flipflops organized with the necessary combinational logic to perform specific operations. All flipflops in a register operate in parallel. In this example, the eight flipflops (**`FF7..FF0`**) act simultaneously at the rising edge of the clock input, **`Clk`**. The register stores one of the 8-bit inputs, either **`IA7..IA0`**, or **`IB7..IB0`**, or some other input, depending on the select input(s), **`InSel`**.

Arithmetic and Logic Unit (ALU)

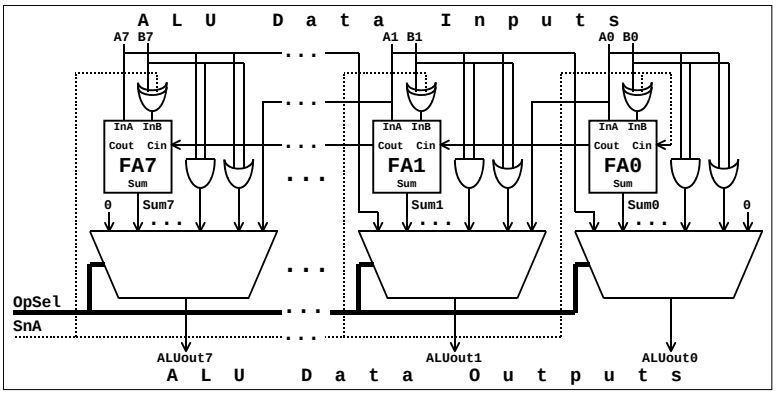
Arithmetic and Logic Unit is a block heavily loaded with combinational logic to perform operations between the microprocessor registers and/or data retrieved from memory.

- **Arithmetic operations:** Addition, subtraction, negation. Some microprocessors have multiplication too.
- **Logic operations:** Bit-wise NOT (inversion), AND, OR, EXOR.
- **Setting condition flags:** Generation of status information according to the result after every ALU operation:

  =0 : set if the result is zero
  >0 : set if the result is positive
  <0 : set if the result is negative
  Carry : set if there is a carry after the operation
  Overflow : set if there is an overflow after the operation

A collection of these and similar flag bits is usually called as "**Status Register**".

ALU?



The diagram above shows a simple ALU. The control inputs, **`OpSel`**, select operations (ADD, AND, OR, ...) between the 8-bit data inputs **`A7..A0`**, and **`B7..B0`**, or unary operations (SHIFT left/right) on **`A7..A0`**.

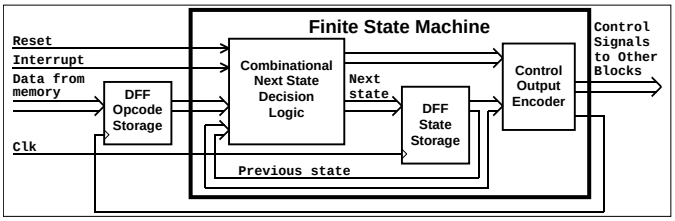
Exercise: What does **SnA** input do? How?

Controller

Controller block is a complex finite state machine that generates the necessary control signals for the chain of events required for execution of instructions.

- Select the source of address for memory access cycle.
- Select the source of data to be written into memory.
- Select the target of data read from memory or who grabs the incoming data:
  Controller itself: opcode fetch
  PC: target address of a JUMP or call
  Registers: data LOAD from memory
  ALU: operand of an ALU operation
  Address buffer: direct or indirect addressing for data
- Select ALU operation type.
- Decide for conditional branching according to ALU status flags.

Controller?



The controller is a Finite State Machine (FSM) that makes decisions depending on the previously stored state information and the current inputs. A combinational logic circuitry decides on the next state and an encoder generates the control outputs.

Execution of all machine instructions starts with opcode fetch cycles. Once the opcode is stored, the controller decides on the sequence of states that generate the necessary control signals to complete the instruction. The sequence of control outputs are usually called as **micro instructions**.

Registers

All microprocessors are equipped with a number of registers that are internal storage units. Microprocessor can access the registers much faster than reading or writing a memory location.

A register is nothing but a collection of flipflops organized with the necessary combinational logic to perform specific operations. Number of registers, their names and functionality varies from one microprocessor architecture to another. Typical register functions are:

- Temporary fast storage inside the microprocessor
- Storage for operation results (accumulator)
- Keeping status information (status register)
- Address generation (index register, stack pointer). Program counter is a register too.

Accumulator

It is the register closely associated with the ALU. Some microprocessor architectures considers accumulator as a part of the ALU, setting it aside from the other registers

Accumulator keeps the result of the last arithmetic or logic operation. This allows execution of a series of instructions without storing intermediate results back in the memory.

Status register flags are set according to the operation result or the data last loaded into the accumulator.

Some microprocessors have more than one register that can be used as accumulator.

Index Register

The purpose of index register is to use the data stored in the microprocessor as an address to access memory. In many microprocessor architectures, an index register is the only way to implement the "array" structures or the "pointer" functionality that you can see in high-level languages like C.

Microprocessor first loads the index register with the address information stored as data in the memory, and then uses the index register as the source of memory address to access the target data location.

An index register is usually equipped with counter functions that facilitate quick access to sequential memory blocks once the first address of the block is loaded into the index register. The machine instruction set includes increment and decrement operations on index registers to support the sequential memory access.

Addressing Modes

Addressing modes are the methods of accessing memory to retrieve an operand or to store the result of an operation. The access method determines what will follow the opcode in a machine instruction.

- **Immediate addressing:** Data is stored immediately after the opcode.
- **Direct addressing:** Address of data follows the opcode.
- **Indirect addressing:** Address of a pointer to data (address of address) follows the opcode. Indirect addressing through memory storage is not always efficient in overall program flow. Most of the time, we want to modify the pointer before or after accessing data, so it is better to keep it in the microprocessor This addressing mode is usually implemented through index registers (**register-indirect addressing**).

Immediate Addressing

Data is stored immediately after the opcode. This addressing mode is restricted to read-only operand data stored with the executable code.

	addr	data	comment
	1A00	3B	
	1A01	28	
	1A02	C3	
PC→	1A03	57	opcode: LOADim Acc
	1A04	33	data: 0x33 into Acc
	1A05	55	
	1A06	1A	
	1A07	21	
	1A08	38	
	1A09	0E	
	1A0A	C3	
	....	....	

1) Controller gets opcode at PC address.
2) Controller increments PC.
3) Accumulator gets data at PC address.
4) Controller increments PC.

Direct Addressing

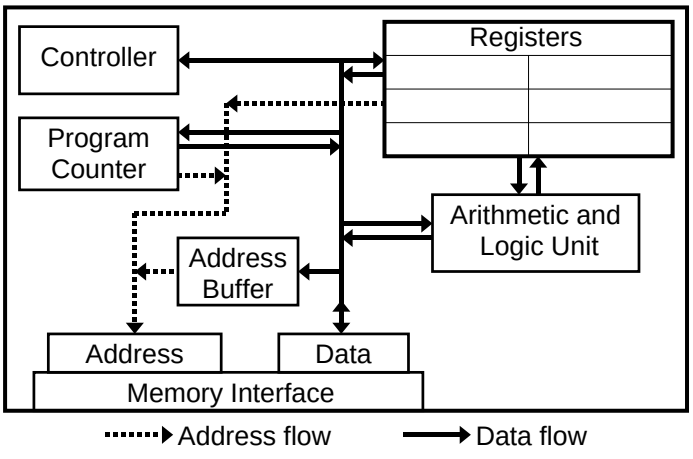
Address of data follows the opcode. This addressing mode can be used for storing data in memory as well as retrieving an operand from memory.

	addr	data	comment
	1A00	3B	
	1A01	28	
	1A02	C3	
PC→	1A03	56	opcode: LOADdc Acc
	1A04	09	addr: low-byte of addr.
	1A05	1A	addr: high-byte of addr.
	1A06	08	
	1A07	A5	
	1A08	55	
	1A09	33	data: 0x33 into Acc
	1A0A	21	
	....	....	

1) Controller gets opcode at PC address.
2) Controller increments PC.
3) Address buffer gets low addr. byte at PC address.
4) Controller increments PC.
5) Address buffer gets high addr. byte at PC address.
6) Controller increments PC.
7) Accumulator gets data at the location targeted by the address buffer.

Address Loaded as Data

In direct addressing, address of data follows the opcode. An address buffer is necessary to drive the memory address inputs with the data retrieved from memory.



Register-Indirect Addressing

Address of data is stored in an index register. An opcode that requires indirect addressing also specifies which register is to be used for addressing.

	addr	data	comment
	1A00	3B	
	1A01	28	opcode: LOADim XR
	1A02	09	im. data: low-byte of addr.
	1A03	1A	im. data: high-byte of addr.
PC→	1A04	C3	opcode: LOADid Acc, @XR
	1A05	56	opcode: INC XR
	1A06	A3	opcode: ADDid Acc, @XR
	1A07	70	
	1A08	55	
	1A09	33	data: 0x33 into Acc
	1A0A	21	data: 0x21 add to Acc
	....	....	

1) Controller gets opcode at PC address.
2) Accumulator gets data at the location (**1A09**) targeted by XR index register.
3) Controller increments PC.
4) Controller gets opcode and increments XR.
5) Controller increments PC.
6) Controller gets opcode.
7) Data from the address in XR (**1A0A**) added to Acc.
8) Controller increments PC.

Memory Access: Address Paths

- **From PC:** Opcode fetch. Loading immediate data that follow the opcode in the program. Reading address of data in direct addressing mode.
- **From Registers:** Address of data is obtained from an index register.
- **From Address Buffer:** Address of data follows the opcode in direct addressing mode. First, address of data is loaded into the address buffer while PC is the address source, then data is loaded or stored while address buffer is the address source.

Memory Access: Data Paths

- **To Controller:** Opcode fetch.
- **To/From Registers:** Loading data from memory or saving register data into memory.
- **To/From ALU:** Retrieving data for an operation or storing the result.
- **To PC:** Retrieving a new program address (through the address buffer) for a branch or call operation.
- **To Address Buffer:** Retrieving address of data for indirect addressing.
- **To Memory:** Storing contents of a register in memory.

Instruction Execution:

LOAD A, #0; *Load Register-A Immediate*

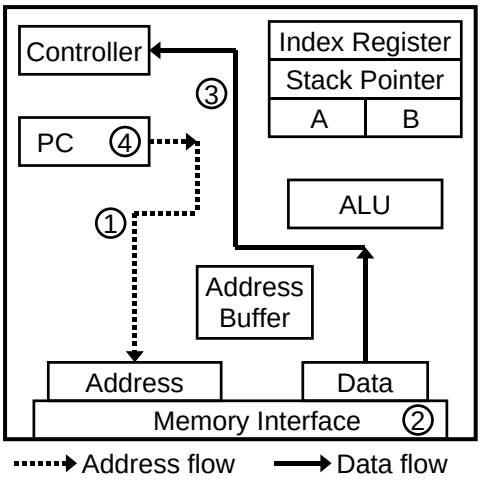
	address	data
PC >	1A00	36
	1A01	00
	1A02	76
	1A03	11
	1A04	62
	1A05	1A
	1A06	55
	1A07	A1
	1A08	20
	1A09	BB
	...	...
	1A55	33
	1A56	00
	...	...

Initial values:
PC=1A00, Opcode=??, Reg-A=??

After Cycle-1: Fetch opcode from memory
PC=1A01, Opcode=36

After Cycle-2: Get immediate data from the memory byte following opcode (PC+1)
PC=1A02, Reg-A=00

Instruction: **Load Register-A Immediate**
Cycle-1: Opcode Fetch



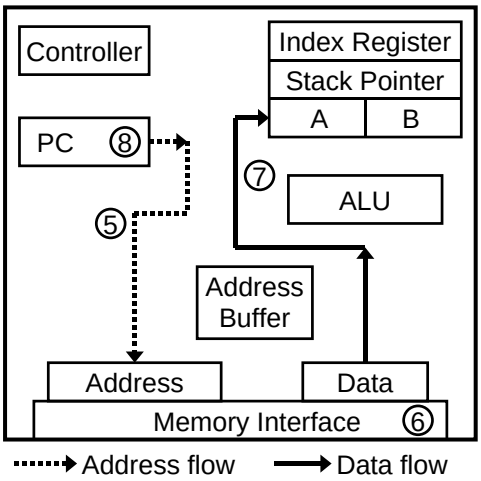
1) PC => Mem. address

2) Memory wait state

3) Get opcode:
Mem. data => controller

4) Increment PC

Instruction: **Load Register-A Immediate**
Cycle-2: Read immediate data



5) PC => memory address

6) Memory wait state

7) Get data from memory:
Memory data => Reg-A

8) Increment PC

Instruction Execution:

ADD A, #11; *Add Register-A Immediate*

	address	data
	1A00	36
	1A01	00
PC >	1A02	76
	1A03	11
	1A04	62
	1A05	1A
	1A06	55
	1A07	A1
	1A08	20
	1A09	BB
	...	...
	1A55	33
	1A56	00
	...	...

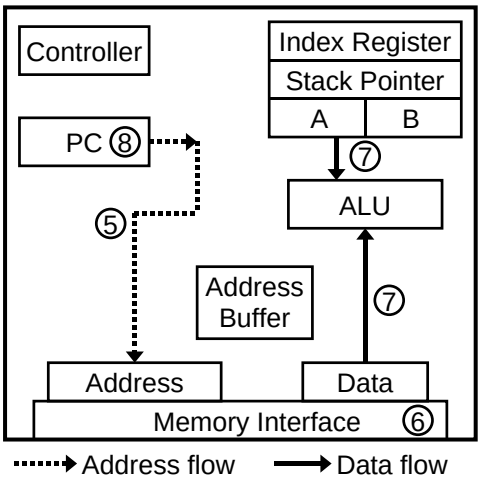
Initial values:
PC=1A02, Opcode=36, Reg-A=00

After Cycle-1: Fetch opcode from memory
PC=1A03, Opcode=76

After Cycle-2: Get immediate data from the memory byte following opcode (PC+1)
PC=1A04, ALUinA=00, ALUinB=11

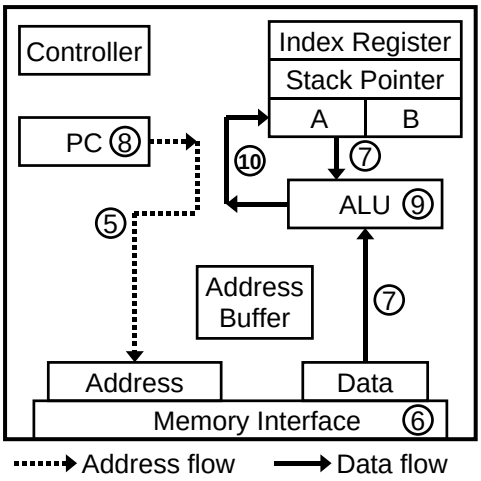
After Cycle-3: Perform addition and store the result in register-A
PC=1A04, Reg-A=11

Instruction: **Add Register-A Immediate**
Cycle-2: Get immediate data



5) PC => memory address
6) Memory wait state
7) ALU gets data:
Register-A => ALUinA
Memory data => ALUinB
8) Increment PC

Instruction: **Add Register-A Immediate**
Cycle-3: Add operand and store result



5) PC => memory address

6) Memory wait state

7) ALU gets data:
Register-A => ALUinA
Memory data => ALUinB

8) Increment PC

9) ALU performs addition operation.

10) Store the result:
ALUout => Register-A

Instruction Execution:
LOAD A,<addr>; *Load Register-A Direct*

	address	data
	1A00	36
	1A01	00
	1A02	76
	1A03	11
PC >	1A04	62
	1A05	1A
	1A06	55
	1A07	A1
	1A08	20
	1A09	BB
	...	...
	1A55	33
	1A56	00
	...	...

Initial values:
PC=1A04, Opcode=76, Addr.Buf.=????, Register-A=11

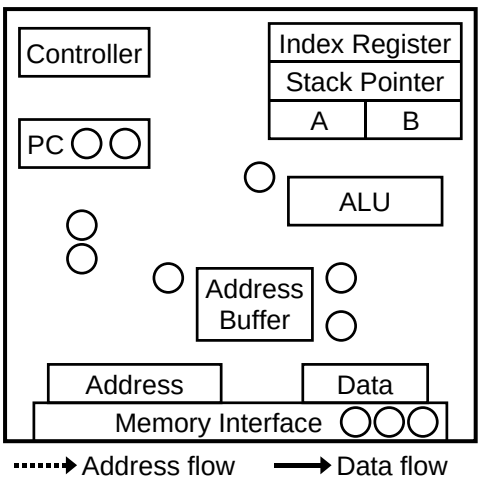
After Cycle-1: Fetch opcode from memory
PC=1A05, Opcode=62

After Cycle-2: Get high byte of the target address from the memory byte following opcode (PC+1)
PC=1A06, Addr.Buf.=1A??

After Cycle-3: Get low byte of the target address from the next memory address (PC+2)
PC=1A07, Addr.Buf.=1A55

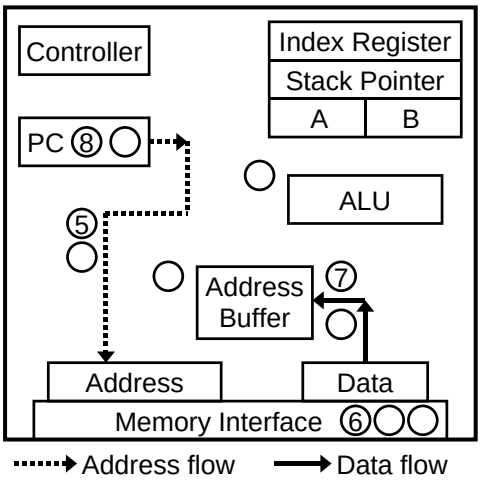
After Cycle-4: Read data into Register-A from memory address 1A55
PC=1A07, Register-A=33

Instruction: **Load Register-A Direct**
Draw data/address paths and mark sequence



5)
6)
7)
8)
9)
10)
11)
12)
13)
14)
15)

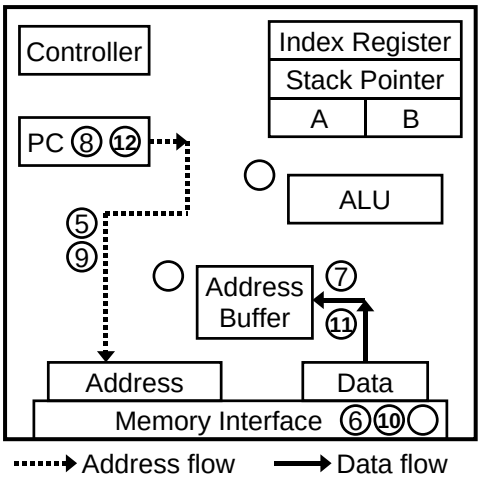
Instruction: **Load Register-A Direct**
Cycle-2: Get high byte of operand address



5) PC => memory address
6) Memory wait state
7) Get high byte of address:
Mem. data => Addr. buffer
8) Increment PC

9)
10)
11)
12)
13)
14)
15)

Instruction: **Load Register-A Direct**
Cycle-3: Get low byte of operand address

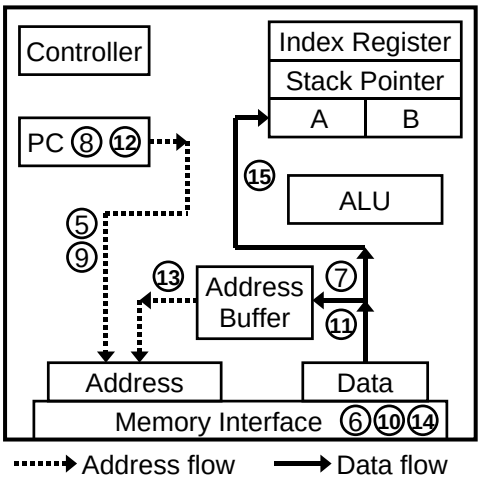


5) PC => memory address
6) Memory wait state
7) Get high byte of address:
Mem. data => Addr. buffer
8) Increment PC

9) PC => memory address
10) Memory wait state
11) Get low byte of address:
Mem. data => Addr. buffer
12) Increment PC

13)
14)
15)

Instruction: **Load Register-A Direct**
Cycle-4: Read data into Register-A



5) PC => memory address
6) Memory wait state
7) Get high byte of address:
Mem. data => Addr. buffer
8) Increment PC

9) PC => memory address
10) Memory wait state
11) Get low byte of address:
Mem. data => Addr. buffer
12) Increment PC

13) Addr. buf. => mem addr.
14) Memory wait state
15) Get operand data:
Mem. data => Register-A

Instruction Execution:

JUMP <addr>; *Jump to Absolute Address*

	address	data
	1A00	36
	1A01	00
	1A02	76
	1A03	11
	1A04	62
	1A05	1A
	1A06	55
PC >	1A07	A1
	1A08	20
	1A09	BB
	...	...
	1A55	33
	1A56	00
	...	...

Initial values:
PC=1A07, Opcode=62, Addr.Buf.=????

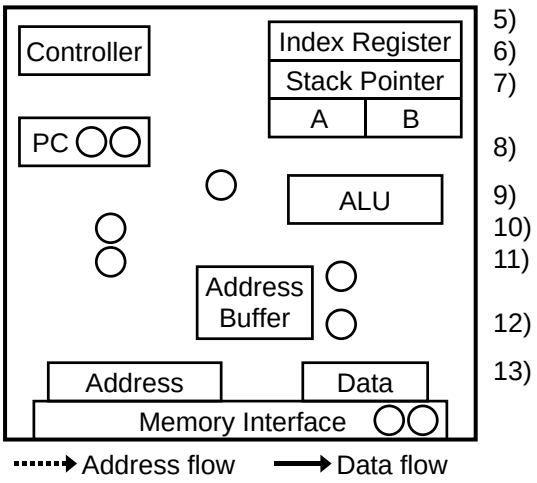
After Cycle-1: Fetch opcode from memory
PC=1A08, Opcode=A1

After Cycle-2: Get high byte of target address from the memory byte following opcode (PC+1)
PC=1A09, Addr.Buf.=20??

After Cycle-3: Get low byte of the target address from the next memory address (PC+2)
PC=1A0A, Addr.Buf.=20BB

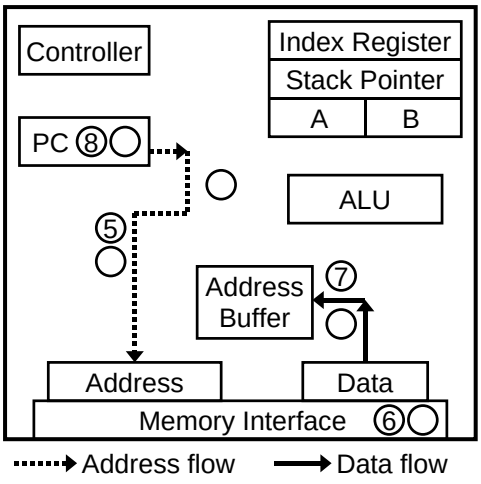
After Cycle-4: Transfer target address from address buffer to PC:
PC=20BB, Addr.Buf.=20BB

Instruction: Jump to Absolute Address
Draw data/address paths and mark sequence



5)
6)
7)
8)
9)
10)
11)
12)
13)

Instruction: Jump to Absolute Address
Cycle-2: Get high byte of target address



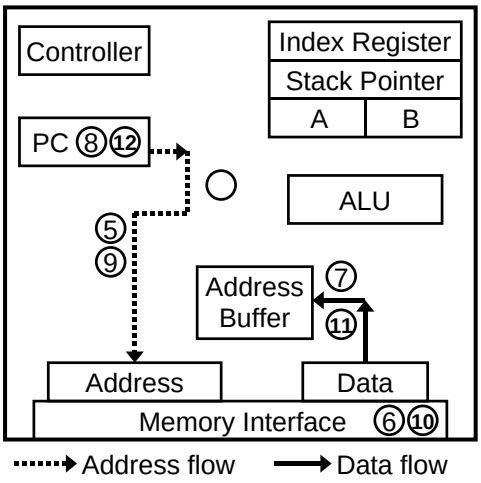
5) PC => memory address
6) Memory wait state
7) Get high byte of address:
Mem. data => Addr. buffer
8) Increment PC

9)
10)
11)

12)

13)

Instruction: Jump to Absolute Address
Cycle-3: Get low byte of target address

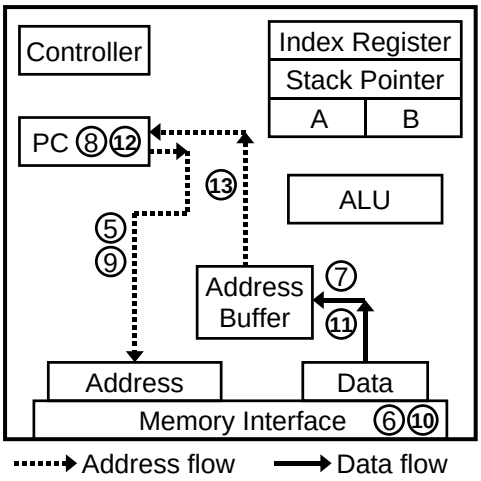


5) PC => memory address
6) Memory wait state
7) Get high byte of address:
Mem. data => Addr. buffer
8) Increment PC

9) PC => memory address
10) Memory wait state
11) Get low byte of addr.:
Mem. data => Addr. buffer
12) Increment PC

13)

Instruction: Jump to Absolute Address
Cycle-4: Transfer target address to PC



5) PC => memory address
6) Memory wait state
7) Get high byte of address:
Mem. data => Addr. buffer
8) Increment PC

9) PC => memory address
10) Memory wait state
11) Get low byte of addr.:
Mem. data => Addr. buffer
12) Increment PC

13) PC gets new address for the next instruction:
Addr. buffer => PC

Absolute, Relative Addressing

The term, "**absolute address**," refers to the hardware address supplied to the address inputs of memory device to access data.

In a typical memory organization with 16 address bits and 8 data bits (64 KBytes memory), reading an absolute address takes two memory access cycles. If the necessary data can be stored within 256 bytes of the executable code, then we can use an 8-bit relative address that saves one memory read cycle. This 1-byte **relative address** is an offset added to program counter or an index register to obtain the absolute address.

Relative addressing can also be utilized in branching instructions (i.e. JUMP relative to current PC address).

Relative Addressing Benefits

Faster execution: Absolute addressing requires two memory read cycles to retrive an address whereas relative addressing takes only one cycle. Time savings can be significant assuming that memory access is the most time-consuming cycle.

Saves memory: Relative addressing instructions use less memory.

Code relocation: A program segment or a subprogram that uses relative addressing can be moved to a different memory location without changing the code. On the other hand, all absolute address references had to be updated whenever an absolute target is relocated.

Data records: If the base address is stored in an index register, then a single subprogram can be used on different data sets simply by changing the address in the index register.

Relative Addressing Example

address	code/data	comment
0x0100		
0x0101	**JUMPrel**	jump-relative (0x0101+0x06) to the
0x0102	**0x06**	calculated absolute address 0x0107
0x0103		
0x0104	**LOADrel**	load-relative (0x0104+0x05) data from the
0x0105	**0x05**	calculated absolute address 0x0109
0x0106		
0x0107	**RETURN**	target of JUMPrel 0x06
0x0108		
0x0109	**0x33**	target of LOADrel 0x05
0x010A		

I/O With Outside World

A microprocessors can communicate with external devices through the same address and data lines used for memory access. There are two widely used addressing options:

1. There are dedicated instructions for I/O operations. Microprocessor signals external devices through an additional control line when execution I/O instructions. This signal enables addressing of other devices instead of the memory.

2. A section of the memory address space is reserved for I/O operations. An external address decoder redirects any memory access at these addresses to I/O devices. This I/O addressing method is called "**memory mapped I/O**."

In case of a microcontroller, I/O ports, channels, and other peripherals are integrated in a single device. Several **special-function-registers** (**SFR**) are used as a common method of access to these peripherals.

Handling External Events

Interrupt mechanism allows external devices to redirect program flow in a microprocessor. The controller saves the PC address and makes an automatic subprogram call to an **Interrupt Service Routine (ISR)** after an interrupt signal is received.

Address of the ISR is predetermined and it is available to the controller when the interrupt is received. ISR code communicates with the external device and performs the necessary tasks required to clear the interrupt. The regular code execution resumes after a RETURN instruction is executed at the end of the ISR.

•