

• (The Microarchitecture • Level

Your PC ran into a problem and needs to restart. This BSOD (blue screen of death) was faked for the sake of this presentation.

• (The Microarchitecture • Level

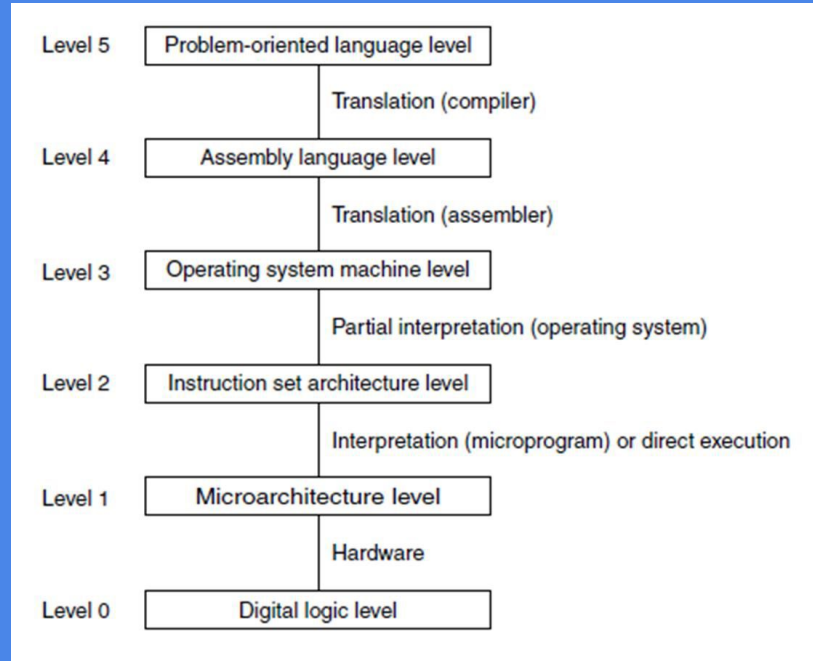
Above the digital logic level is the *microarchitecture level*. Its job is to implement the *ISA (Instruction Set Architecture)*.

• (The Microarchitecture • Level

Its design depends on the ISA being implemented, as well as the cost and performance goals of the computer.

Many modern ISAs, particularly RISC designs, have simple instructions that can usually be executed in a single clock cycle.

• (The Microarchitecture Level



A six-level computer. The support method for each level is indicated below it.

• (An Example of • Microarchitecture

Our microarchitecture will contain a *microprogram* (in *ROM*), whose job is to fetch, decode, and execute *IJVM instructions*.

• (An Example of • Microarchitecture

As a convenient model for the design of the microarchitecture we can think of it as a programming problem, where each instruction at the ISA level is a function to be called by a *master program*.

• (An Example of • Microarchitecture

The microprogram has:

a set of *variables*

called the *state* of the computer, which can be accessed by all the functions.

• (An Example of • Microarchitecture

IJVM instructions:

- Each instruction has a few fields, usually one or two, each of which has some specific purpose.
- First field of every instruction is the *opcode* (short for *operation code*)
- Many instructions have an additional field, which specifies the operand.

• (An Example of • Microarchitecture

- Model of execution, sometimes called the *fetch-decode-execute cycle*.

• (The Data Path

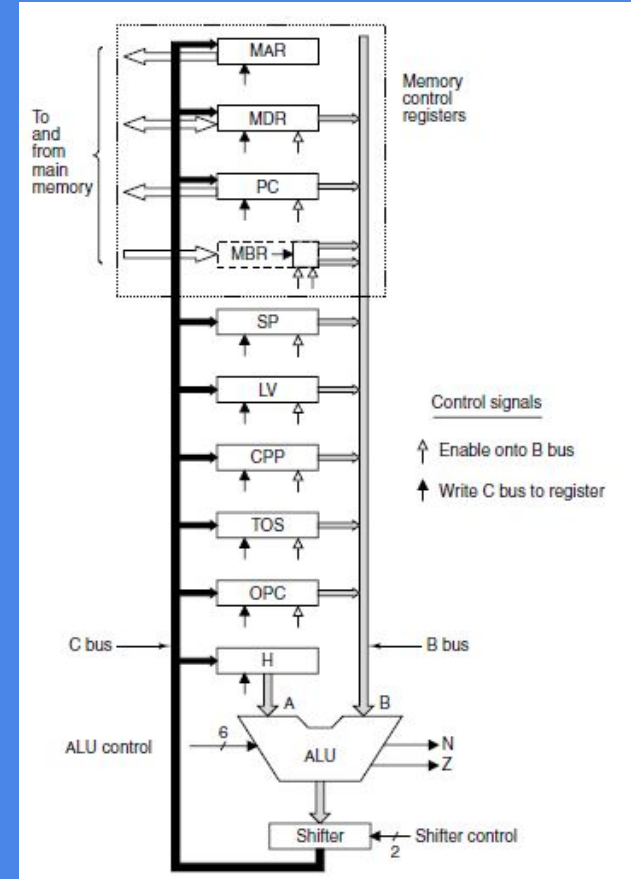
The *data path* is that part of the CPU containing the ALU, its inputs, and its outputs.

While it has been carefully optimized for interpreting IJVM programs, it is fairly similar to the data path used in most machines.

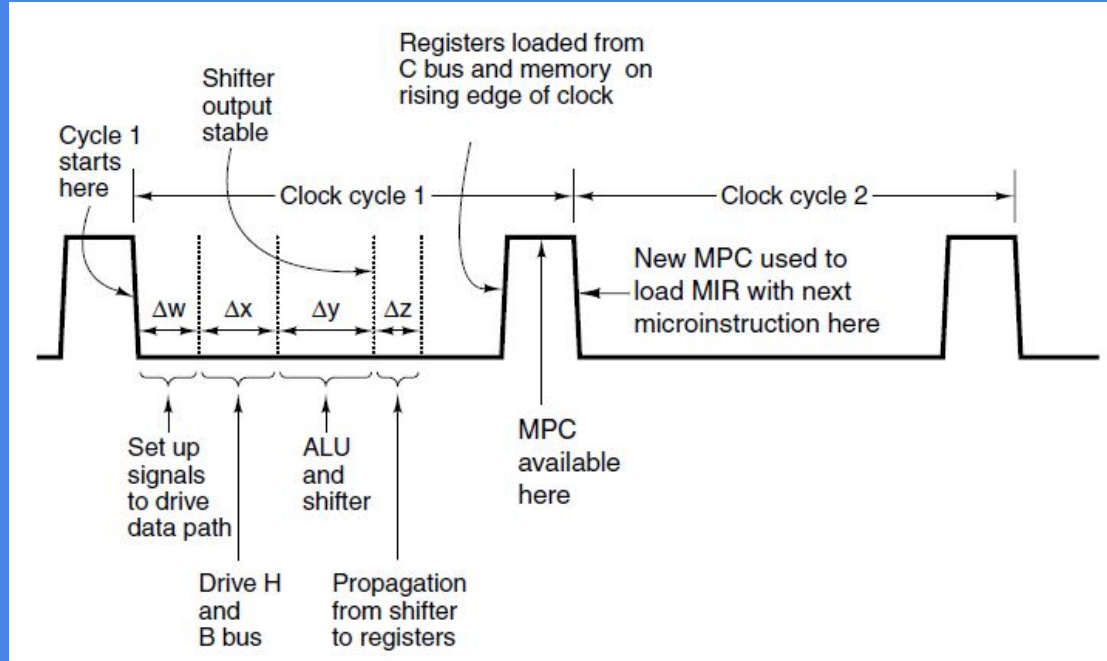
• (The Data Path

- The data path of the example microarchitecture.

In this model, the master program is a simple, endless loop that determines a function to be invoked, calls the function, then starts over.



Data Path Timing



Timing diagram of one data path cycle

• (Data Path Timing

The timing of these events is shown in the previous slide. Here a short pulse is produced at the start of each clock cycle. It can be derived from the main clock.

• (The Data Path Timing

The activities that go on during the subcycles are shown below along with the subcycle lengths (in parentheses).

- The control signals are set up (Δw).
- The registers are loaded onto the B bus (Δx).
- The ALU and shifter operate (Δy).
- The results propagate along the C bus back to the registers (Δz).

• (Memory Operation

Our machine has two different ways to communicate with memory: a 32-bit, word-addressable memory port and an 8-bit, byte-addressable memory port. The 32-bit port is controlled by two registers, *MAR* (*Memory Address Register*) and *MDR* (*Memory Data Register*).

• (Memory Operation

The 8-bit port is controlled by one register, PC, which reads 1 byte into the low-order 8 bits of MBR. This port can only read data from memory; it cannot write data to memory.

Each of these registers is driven by one or two control signals.

• (MicroInstructions

To control the data path, we need 29 signals. These can be divided into five functional groups, as described below.

- 9 Signals to control writing data from the C bus into registers.
- 9 Signals to control enabling registers onto the B bus for ALU input.
- 8 Signals to control the ALU and shifter functions.
- 2 Signals to indicate memory read/write via MAR/MDR.
- 1 Signal to indicate memory fetch via PC/MBR.

- **MicroInstructions**
- **Control: MIC-1**

A sequencer that is responsible for stepping through the sequence of operations for the execution of a single ISA instruction.

- **MicroInstructions**
- **Control: MIC-1**

The sequencer must produce two kinds of information each cycle:

1. The state of every control signal in the system.
2. The address of the microinstruction that is to be executed next.

• (MicroInstructions

• Control: MIC-1

The largest and most important item in the control portion of the machine is a memory called the *control store*.

• (Stacks

Virtually all programming languages support the concept of procedures (methods), which have local variables. These variables can be accessed from inside the procedure but cease to be accessible once the procedure has returned. The question thus arises: “Where should these variables be kept in memory?”

• (Stacks

Instead, a different strategy is used. An area of memory, called the stack, is reserved for variables, but individual variables do not get absolute addresses in it.

• (Stacks

Besides holding local variables, stacks have another use. They can hold operands during the computation of an arithmetic expression. When used this way, the stack is referred to as the *operand stack*. e.g. $a1 = a2 + a3$;

Stacks

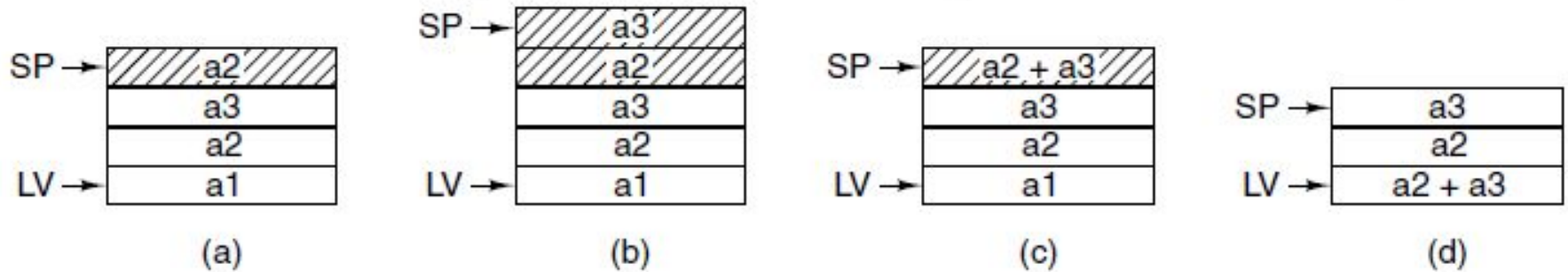


Figure 4-9. Use of an operand stack for doing an arithmetic computation.

Use of an operand stack for doing an arithmetic computation

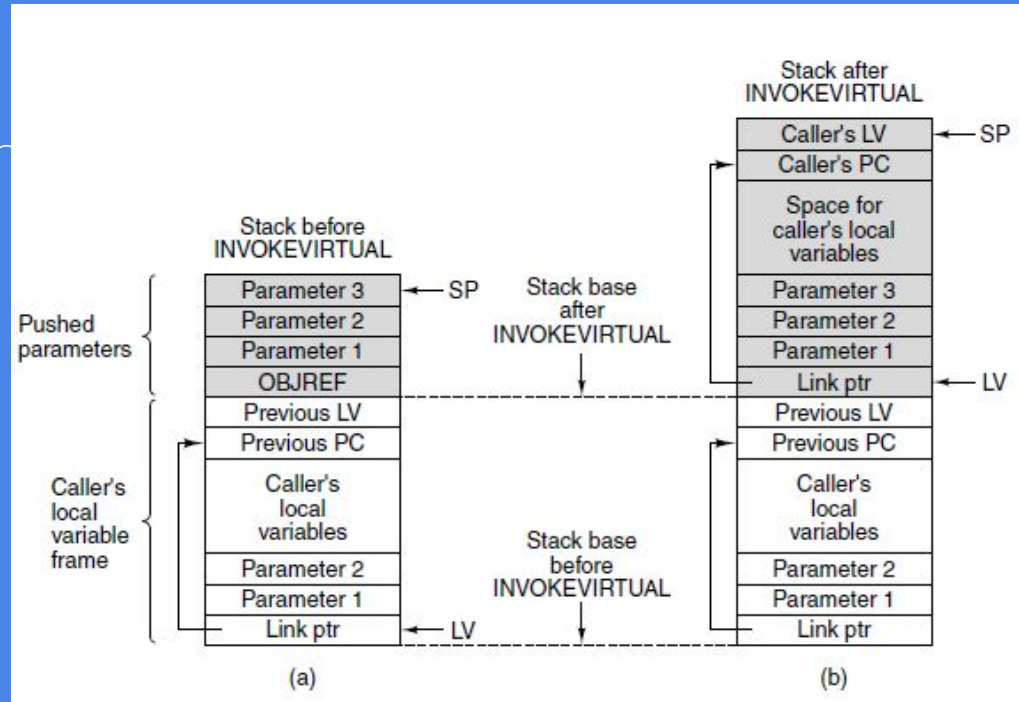
• (The JVM Instruction Set

The JVM instruction set.
The operands *byte*, *const*,
and *varrun* are 1 byte. The
operands *disp*, *index*, and
offset are 2 bytes.

Hex	Mnemonic	Meaning
0x10	BIPUSH <i>byte</i>	Push byte onto stack
0x59	DUP	Copy top word on stack and push onto stack
0xA7	GOTO <i>offset</i>	Unconditional branch
0x60	IADD	Pop two words from stack; push their sum
0x7E	IAND	Pop two words from stack; push Boolean AND
0x99	IFEQ <i>offset</i>	Pop word from stack and branch if it is zero
0x9B	IFLT <i>offset</i>	Pop word from stack and branch if it is less than zero
0x9F	IF_ICMPEQ <i>offset</i>	Pop two words from stack; branch if equal
0x84	IINC <i>varnum const</i>	Add a constant to a local variable
0x15	ILOAD <i>varnum</i>	Push local variable onto stack
0xB6	INVOKEVIRTUAL <i>disp</i>	Invoke a method
0x80	IOR	Pop two words from stack; push Boolean OR
0xAC	IRETURN	Return from method with integer value
0x36	ISTORE <i>varnum</i>	Pop word from stack and store in local variable
0x64	ISUB	Pop two words from stack; push their difference
0x13	LDC_W <i>index</i>	Push constant from constant pool onto stack
0x00	NOP	Do nothing
0x57	POP	Delete word on top of stack
0x5F	SWAP	Swap the two top words on the stack
0xC4	WIDE	Prefix instruction; next instruction has a 16-bit index

• (The IJVM Instruction Set

Instructions are provided to push a word from various sources onto the stack.



(a) Memory before executing `INVOKEVIRTUAL`. (b) After executing it.

- # Compiling Java to JVM
- Let us now see how Java and JVM relate to one another. (a) we show a simple fragment of Java code. When fed to a Java compiler, the compiler would probably produce the JVM assembly-language

• (Compiling Java to JVM

i = j + k;	1	ILOAD j	// i = j + k	0x15 0x02
if (i == 3)	2	ILOAD k		0x15 0x03
k = 0;	3	IADD		0x60
else	4	ISTORE i		0x36 0x01
j = j - 1;	5	ILOAD i	// if (i == 3)	0x15 0x01
	6	BIPUSH 3		0x10 0x03
	7	IF_ICMPEQ L1		0x9F 0x00 0x0D
	8	ILOAD j	// j = j - 1	0x15 0x02
	9	BIPUSH 1		0x10 0x01
	10	ISUB		0x64
	11	ISTORE j		0x36 0x02
	12	GOTO L2		0xA7 0x00 0x07
	13 L1:	BIPUSH 0	// k = 0	0x10 0x00
	14	ISTORE k		0x36 0x03
	15 L2:			
(a)	(b)	(c)		

a. A Java fragment. b. The corresponding Java assembly language. c. The JVM program in hexadecimal.

• (**Thank You!**
•

Your PC ran into a problem and needs to restart. This BSOD (blue screen of death) was faked for the sake of this presentation.