Today’s Topics

- The Java Virtual Machine
- JVM memory areas
- Use of the stack to store operands and local variables
- JVM instructions
- How JVM instructions are executed by the microarchitecture
- Description of sample JVM instructions
- Compiling Java source code to JVM instructions
- What the microarchitecture must do to execute an IJVM instruction
- The MIC-1 Data Path: components which can do the needed work
- How the data path executes an IJVM IADD instruction

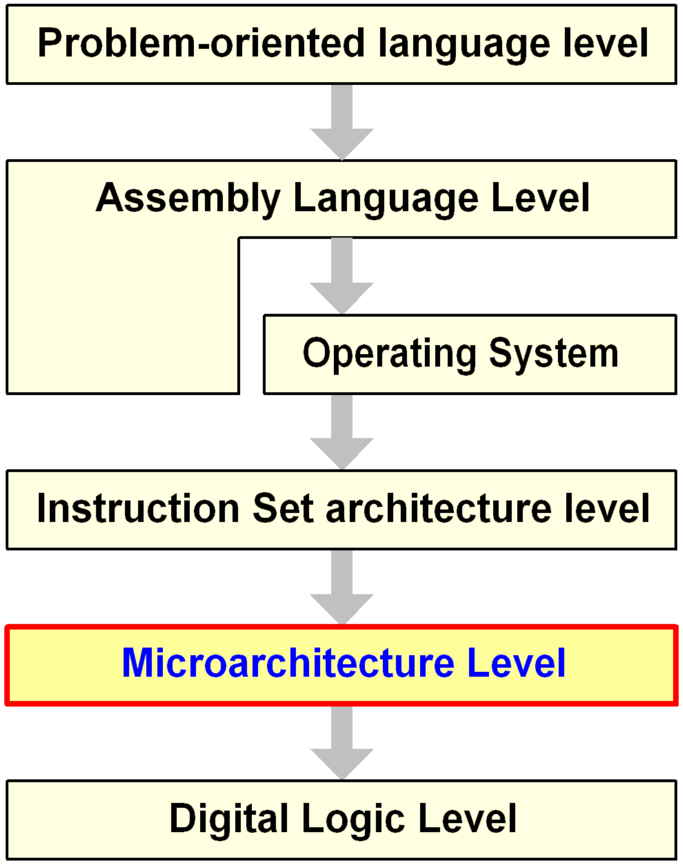
The Microarchitecture Level

We've learned how:

- **Transistors can be used to build Boolean logic circuits**
- **Boolean logic circuits can be combined to build functional units such as Adders and Arithmetic Logic Units (ALUs).**

This is the Digital Logic Level of a computer system. All computer systems use similar Boolean circuits at the Digital Logic Level.

We're now going to study the Microarchitecture Level of a computer system. The purpose of the Microarchitecture is to organize the functional units of the Digital Logic Level so that we can execute machine instructions stored in memory. The set of machine instructions that our system can execute is called the Instruction Set Architecture of the machine.



CPU that execute different sets of ISA instructions (and even different generations of CPUs that execute the same ISA instructions) can have very different microarchitectures.

In order to develop a sample microarchitecture, we need to define the ISA level that our microarchitecture will execute. The author of the textbook has chosen the Java Virtual Machine (JVM) as the sample ISA level.

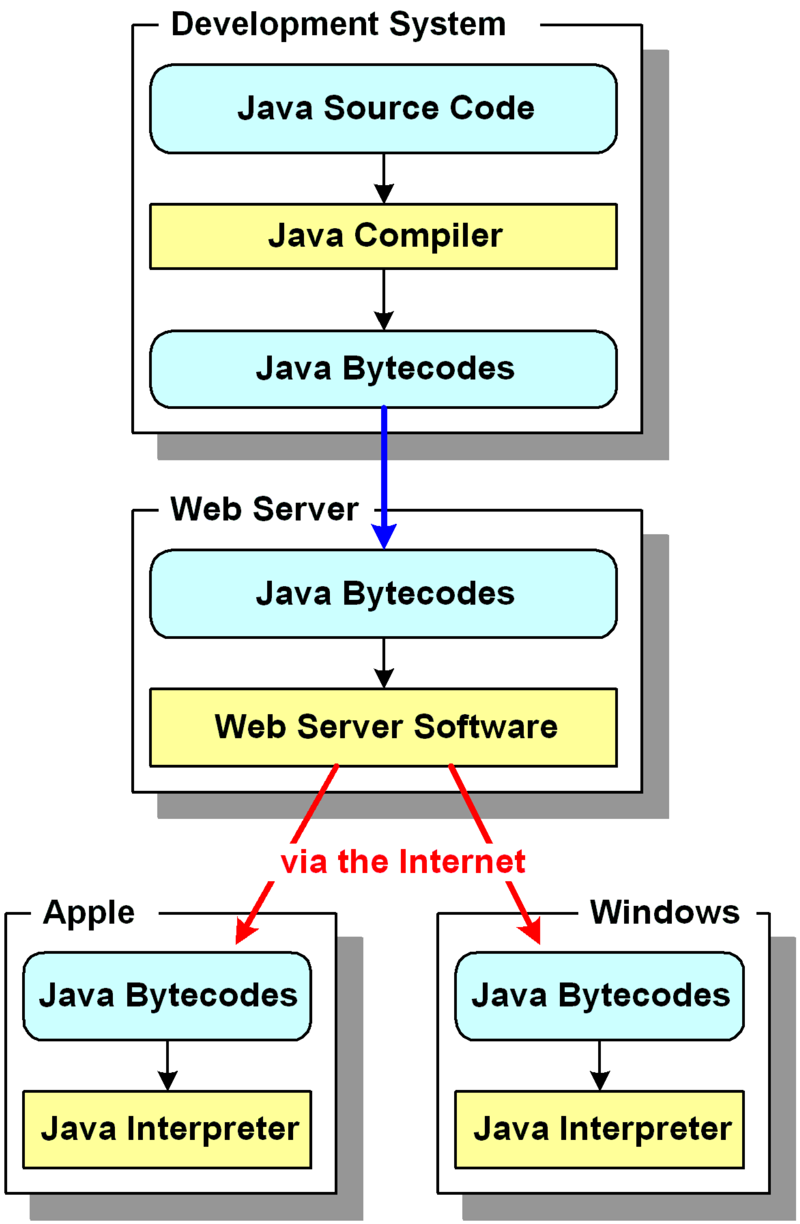
The Java Virtual Machine

Java was designed by Sun to solve the problem of how to download and run program code stored on a web site in the browser of different types of computers (WIntel, Macs, Suns, etc). Their solution was to use an interpeted language that works like this:

- **Programs are written in Java source code and compiled on a development system into "Java Bytecodes".**
- **The Java Bytecodes are placed onto a web server where they can be downloaded by web browsers running on a variety of computers.**
- **The web browser uses a "Java Interpreter" to read the bytecodes and interpret them as native instructions on the local machine.**

A different Java Interpreter is used on each client system to translate the bytecodes into the native client instructions – this means that a single set of Java Bytecodes can be used for all client machines.

The Java Interpeter is called the Java Virtual Machine, and Java Bytecodes are the Instruction Set Architecture level of this virtual machine.



The Sample Microarchitecture

Java was designed as an interpreted language and it was meant to be executed by software. But there's no fundamental reason that a hardware CPU can't be built to execute Java bytecodes.

The author of the textbook uses the JVM specification as the basis for a sample microarchitecture. The microarchitecture will be able to read Java bytecodes from memory and execute them just as a Java Interpreter would.

The full Java Virtual Machine has a very rich set of features and it would be challenging to implement as a microarchitecture. To keep things, simple the author's microarchitecture implements only a subset of the complete JVM:

- **Although Java supports many data types, the author's microarchitecture only implements Integer data.**
- **Not all JVM instructions are implemented.**
- **The object-oriented instructions have been simplified**

The author calls this subset the "Integer JVM" or "IJVM".

IJVM will allow us to learn the principles of how Java code is executed without having to burden ourselves with the all of the features of the Java Virtual Machine.

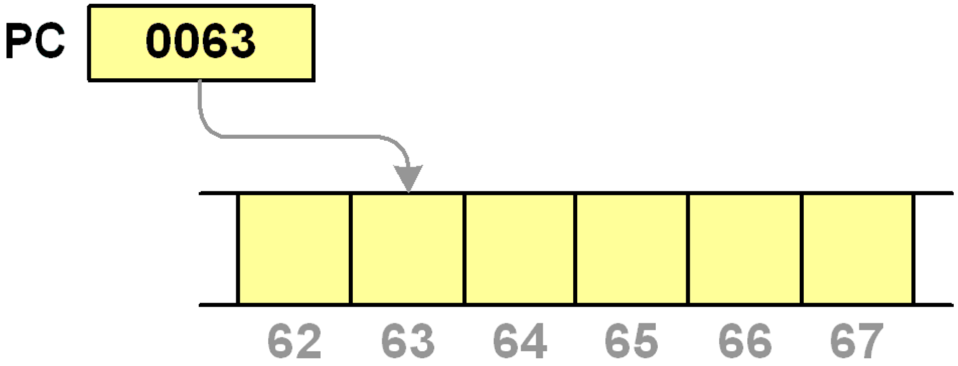
You can find the complete Java Virtual Machine specification on the Internet at:

http://java.sun.com/docs/books/vmspec/

IJVM Memory – The Method Area

Most ISAs have a simple unified byte-addressable memory model where the same memory is used to store both program instructions and program data. IJVM is quite different – it has two separate memory areas that have different purposes and addressing modes

The first area is called the Method Area:



The Method Area holds the IJVM instructions (bytecodes) that are to be executed. The Java Virtual Machine operates by fetching these instructions and doing the work that they specify.

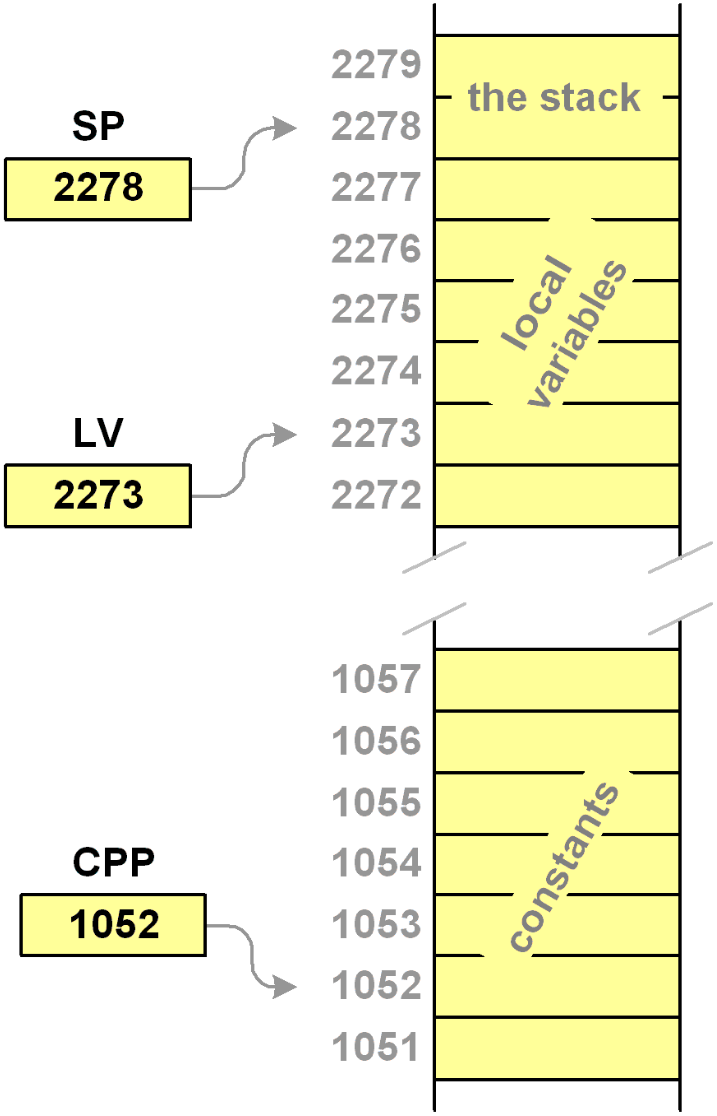
The Method Area has the following characteristics:

- **It is byte-addressable – that is, each address holds 8 bits (1 byte).**
- **A special register called the PC register contains the address of the next instruction that the Java Virtual Machine will execute. As IJVM executes each instruction it adjusts the PC register so that points to the following instruction.**

IJVM Memory – Data Areas

The other memory area is the data area of memory. This area has the following characteristics:

- **Each cell in the data area of memory contains 32-bits. So when you read from or write to a data address you are reading or writing 32 bits (4 bytes) of information. Because of this, we say that the data area of memory is word-addressable.**
- **There are three registers which contain the address of different areas of memory that are used for different purposes:**
  - **The SP register contains the address of the stack, an area used for temporary storage of data**
  - **The LV register contains the address of the local variable frame, the area where variables for the current program routine are stored**
  - **The CPP register contains the address of the constant pool, an area of memory which holds read-only data.**



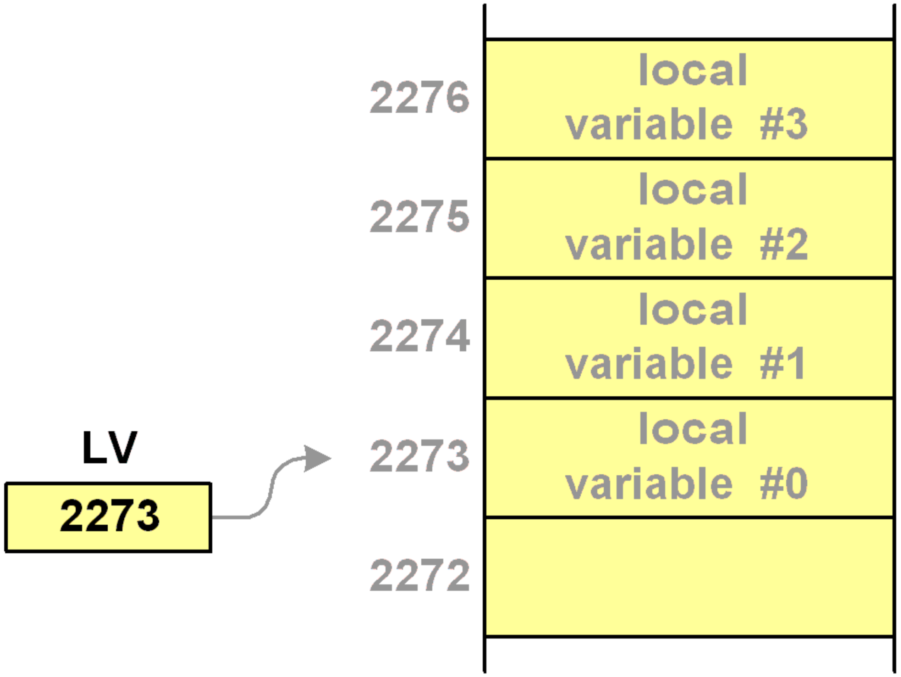
The method and data areas of memory are each separate and have their own addressing. Address "123" in the method area accesses an 8-bit byte that's in a totally different memory area than the 32-bit word stored at address "123" in the data area of memory.

IJVM Memory – Using Pointers

The LV and CPP registers contain the addresses of the memory areas used to store local variables and constants. When a register contains a memory address it's called a "pointer". These pointer are used to find the local variables or constants in a program.

For example, imagine that a subroutine has four local variables. In the Java source code these would be given names such as "i", "j", "k", and "l", but the Java virtual machine knows them only as "variable 0", "variable 1" and so on.

The Java compiler finds room for these variables in memory, stores the four values there, and then it puts the address of "variable 0" (the first variable) into the LV register.



The LV register is used to find the variables whenever the program wants to do something with them. For example, when the program wants to use the value of "variable 2", the Java Virtual Machine adds "2" to the number stored in the LV register:

2273 + 2 = <u>2275</u>

It then reads the variable from the resulting address.

The CPP register works exactly the same way for accessing constants in the constant pool.

The IJVM Memory Model

4.2.2

IJVM program instructions that access the data area of memory use different pointers depending on what kind of instruction is being executed. For example, a different instructions are used to load constants vs. local variables:

LDC 5 **"LDC" is the "Load Constant" instruction. "LDC 5" means to use constant number 5 in the Constant Pool**

ILOAD 5 **"LOAD" is the regular "Integer LOAD" instruction. "ILOAD 5" means to use variable number 5 in the local variable area.**

Notice that even though both instructions use "5" as an operand, they are using totally different data.

The Constant Pool contains read-only data, but it doesn't use "ROM" memory technology. The reason this area is read-only is because none of the IJVM instructions which access the constant pool allow data to be written into it.

Exercise 1 - Pointers

The diagram at right shows the contents of some of the words in memory as well as the LV and CPP registers.

1. What is the value of Constant Number 9?

A – 3370 E – 9

B – 3379 F – 53

C – 3381 G – 63

D – 3390 H – None of these

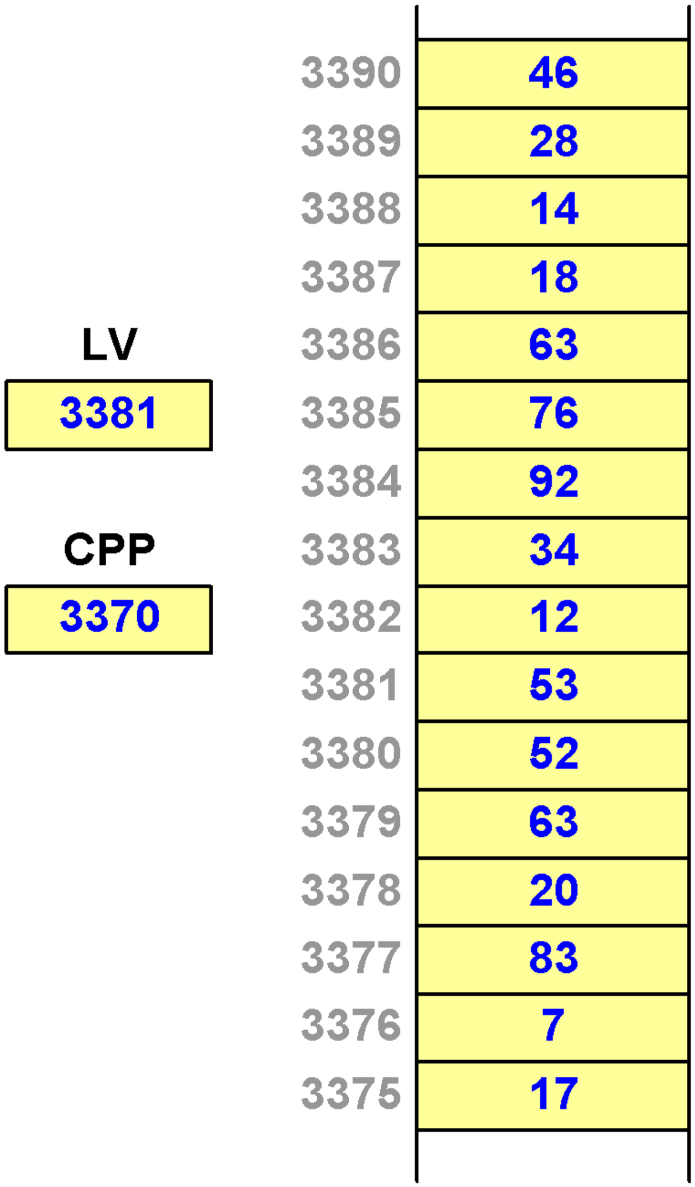
2. What is the value of Local Variable Number 5?

A – 3370 E – 5

B – 3375 F – 76

C – 3381 G – 63

D – 3386 H – None of these



LV

3381

CPP

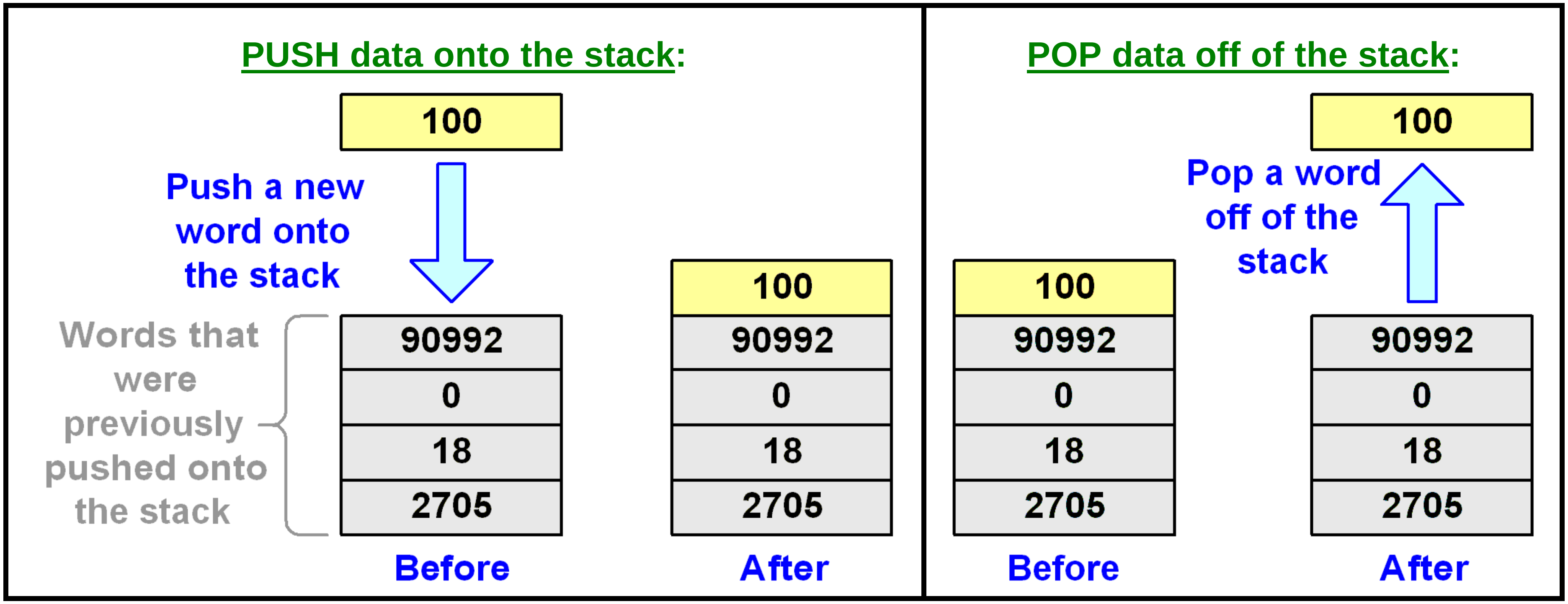
3370

3390	46
3389	28
3388	14
3387	18
3386	63
3385	76
3384	92
3383	34
3382	12
3381	53
3380	52
3379	63
3378	20
3377	83
3376	7
3375	17

Stacks

The Java Virtual Machine is stack-oriented, so a good understanding of stacks is essential in order to understand how it works.

A stack is a LIFO (Last-In, First-Out) list. You can do two things with a stack:


PUSH data onto the stack:
100
Push a new word onto the stack
Words that were previously pushed onto the stack
90992
0
18
2705
Before
100
90992
0
18
2705
After
POP data off of the stack:
100
90992
0
18
2705
Before
100
Pop a word off of the stack
90992
0
18
2705
After

The stack can only be accessed with these two operations. You can't change data already pushed onto the stack, nor can you read stack data (except by popping the top word off).

Stack data is word-oriented, not byte-oriented, and all data on the stack is the same size. In the Java Virtual Machine, stack words are 32 bits long.

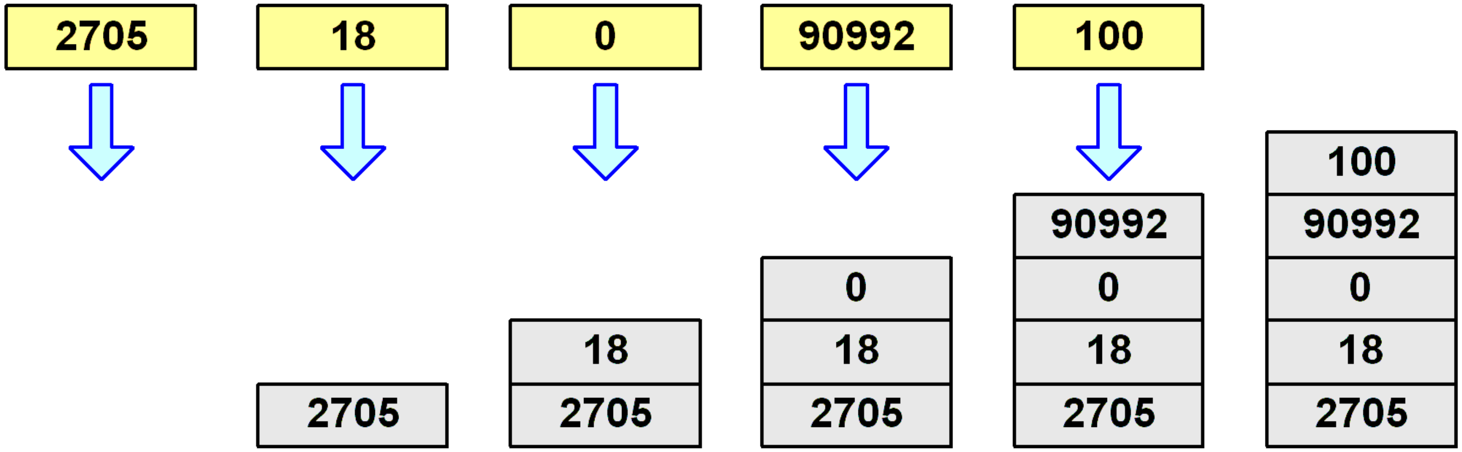
Stacks

The stack is a LIFO list because information is POPed off the stack in the reverse order that it was pushed onto the stack:

Pushing:

2705
18
0
90992
100

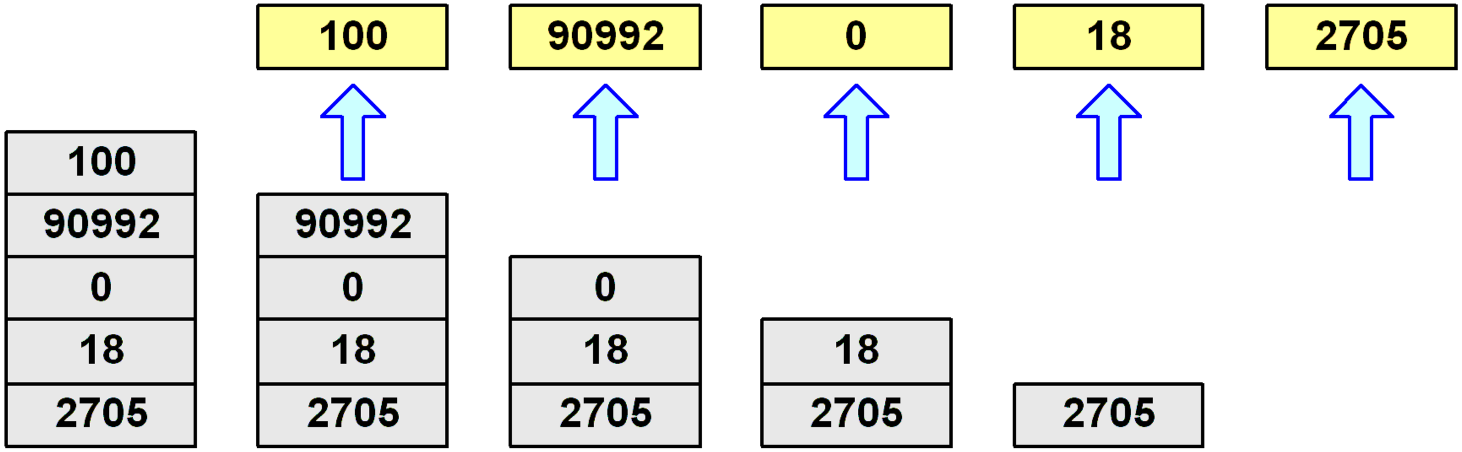
…onto the stack



Popping:

100
90992
0
18
2705

…off of the stack



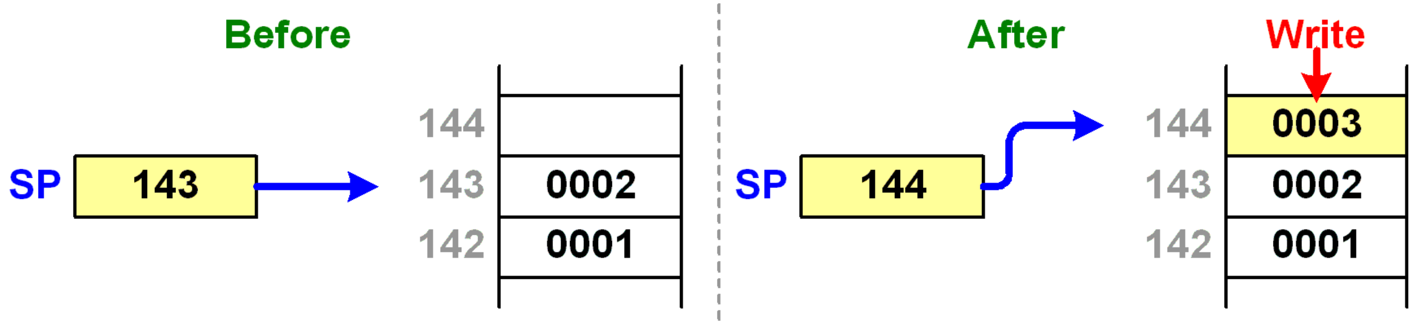
How the Stack Works

The stack is really just a series of words stored in the data area of memory. A special register called the "Stack Pointer" (SP) contains the memory address of the word that was most recently pushed onto the stack. The SP register is to the stack what the PC (Program Counter) register is to the program instructions – it tracks the stack's "current location".

The two basic stack operations simply move data to and from the memory word specified by the SP register, and adjust the address stored in the SP register:

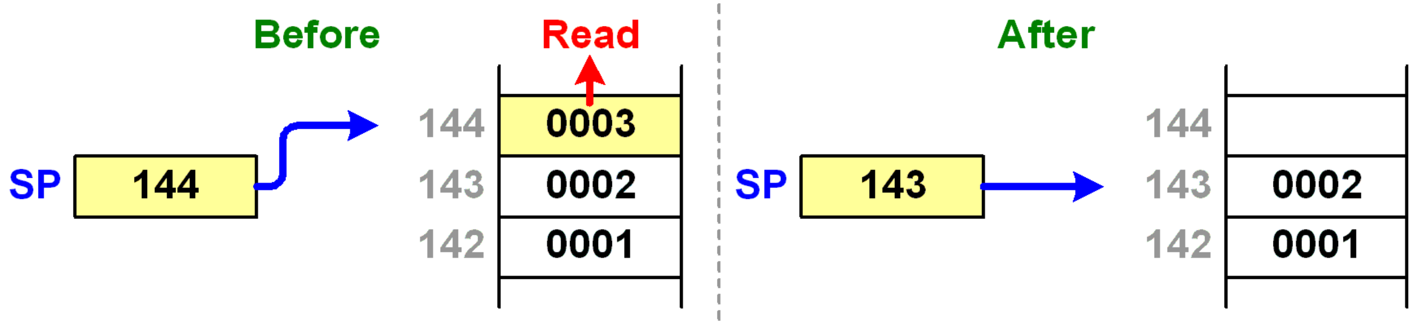
PUSH

- **Add 1 to SP**
- **Move new word to SP's memory address**



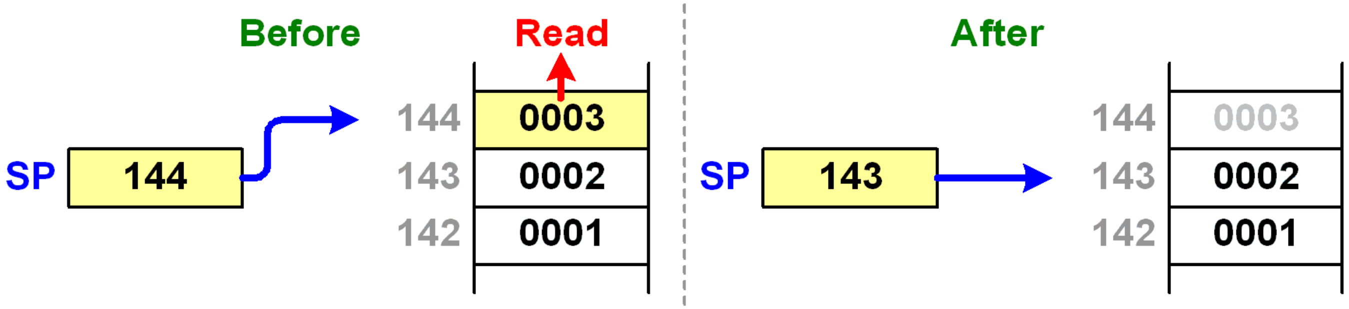
POP

- **Move word from SP's memory address**
- **Subtract 1 from SP**



A Note about the “Unused” part of the Stack

When you “Pop” a word off the stack, the popped data is actually still stored in the memory word it used to occupy:



Memory words are 0 and 1 bits, and they can’t simply be “blank”. So popping a value leaves the original value in the memory word.

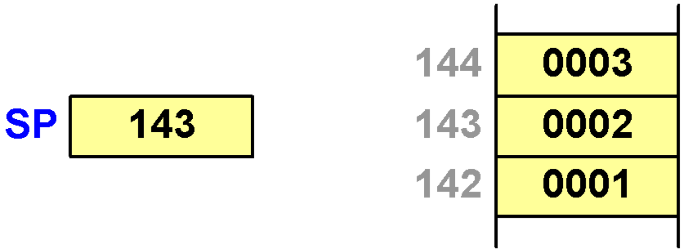
But – the program should never attempt to access words on the stack at addresses above the SP register.

There are two reasons for this:

- **It’s bad programming form to access variables that haven’t been pushed yet or have already been popped. This type of practice can lead to unexpected errors.**
- **The system uses the stack to store temporary data when it handles interrupts. So the memory words at addresses above the SP register can change without warning.**

Exercise 2 – The Stack

The following diagram shows the current state of the stack:



SP 143

144	0003
143	0002
142	0001

Two words are pushed onto the stack, then three words are popped off of the stack.

1. **What will the stack pointer contain after these operations are done?**

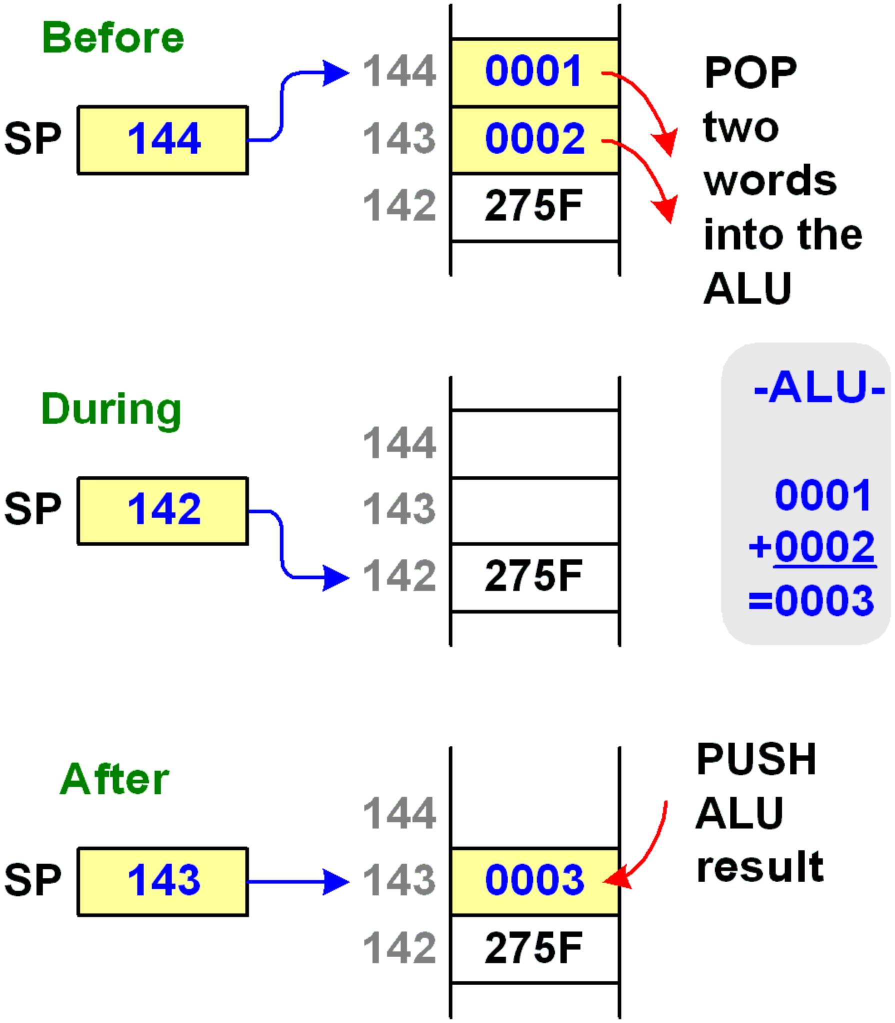
   A – 142
   B – 143
   C – 144
   D – 145
   E – 146
   F – 1
   G – 2
   H – 3

2. **What is the value of the word at the top of the stack after these operations are done?**

   A – 142
   B – 143
   C – 144
   D – 145
   E – 146
   F – 1
   G – 2
   H – 3

Using the Stack for Operands

In IJVM, the operands for most instructions are located on the stack. For example, consider the IJVM "IADD" machine instruction:



Before

SP 144

144 0001

143 0002

142 275F

POP two words into the ALU

During

SP 142

144

143

142 275F

-ALU-

0001
+0002
=0003

After

SP 143

144

143 0003

142 275F

PUSH ALU result

- **the IJVM "IADD" instruction doesn't specify <u>any</u> operands explicitly**
- **it expects the two values that are to be added to be the top two words on the stack**
- **it POPs the top two words from the stack, adds them together, and PUSHes the result back onto the stack again:**

The stack is used in a very similar way for most other IJVM instructions use one or more data values.

Using the stack to hold operands helps to make IJVM programs compact because it means that IJVM program instructions don't need to specify <u>what</u> is being added.

But a program that wants to add two numbers together must first PUSH both numbers onto the stack where the ADD instruction can find them – then it must POP the result from the stack and move it to wherever it's needed.

Using the Stack for Local Variables

The second use for the stack in IJVM (and in most other architectures), is to store local variables. Local variables are data values declared within a program module that aren't used anywhere else. Consider this piece of Java source code:

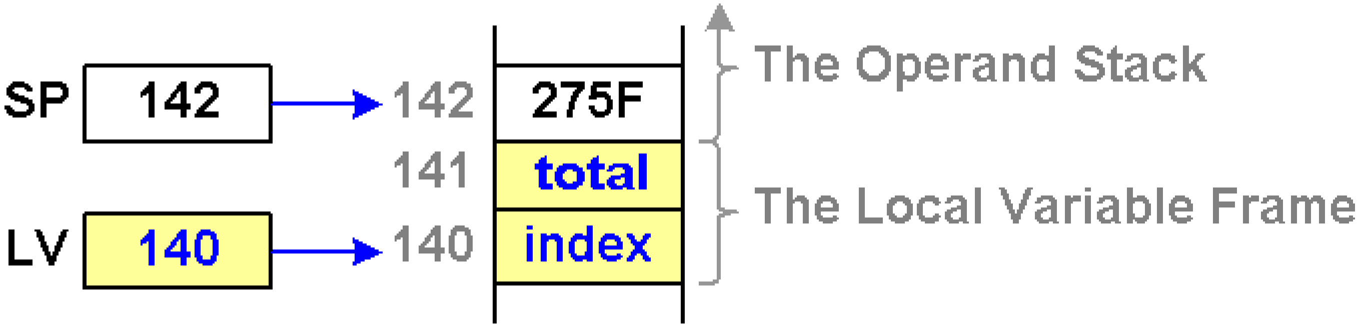
```
public int SumArray(int[] anArray )
{
   int index;
   int total;

   for (index= 0; index < anArray.length; i++)
         total = total + anArray[index];

   return total;
}
```

"`index`" and "`total`" are local variables. These variables are created when the "`SumArray`" subroutine is called, active while it executes, and destroyed when it exits.

Because local variables are only needed while a particular routine is active, and because nested calls and returns can create and destroy several sets of local variables, the stack is an ideal place to hold these variables.

Using the Stack for Local Variables

The Java Virtual Machine uses another special register called "LV" (the Local Variable pointer). This register points to the area of the stack that contains the local variables for the currently-executing routine. When the "`SumArray`" routine shown on the previous page is executing, the stack may look like this:



The "Local Variable Frame" is the area of memory that holds the variables for the currently-executing subroutine. In this example there are only two words that are part of the local variable frame, because only two local variables were declared in the "`SumArray`" routine. If more local variables had been declared then the local variable frame would contain more words.

The "LV" register contains the address of the first variable in the local variable frame. This allows the IJVM bytecodes that are executing to find the local variables:

index – can be found at the memory address contained in the LV register

total – can be found at the memory address contained in the LV register plus one.

Using the Stack for Local Variables

4.2.1

Let’s see what happens when the following Java program is executed:

```
public void main( String [] args )  {
   int m_a;
   int m_b;
   Sub1();                          }

private void Sub1()  {
   int s1_a;
   int s1_b;
   int s1_c;
   Sub2();           }

private void Sub2()  {
   int s2_a;
   int s2_b;
   int s2_c;
   s2_a = s2_a + s2_b;  }
```

Each program module contains two or three local variables. The “`main`” program calls the “`Sub1`” subroutine, which in turn calls the “`Sub2`” subroutine.

Using the Stack for Local Variables

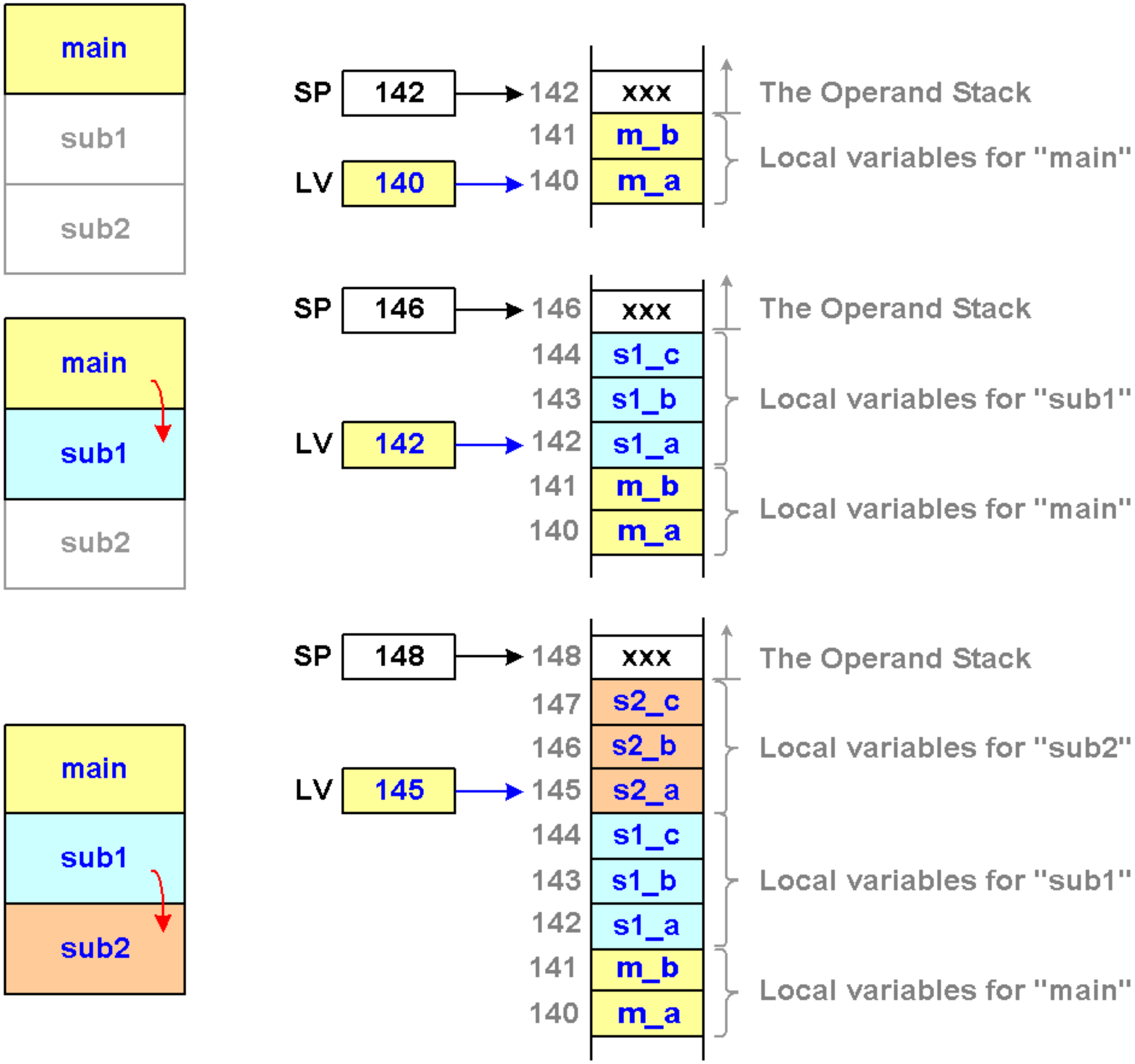
When the "main" routine is executing, only its local variables exist.

When "sub1" is called, space is allocated on the stack to hold it's local variables and LV is pointed to them.

When "sub2" is called, space is allocated on the stack to hold it's local variables and LV is pointed to them.

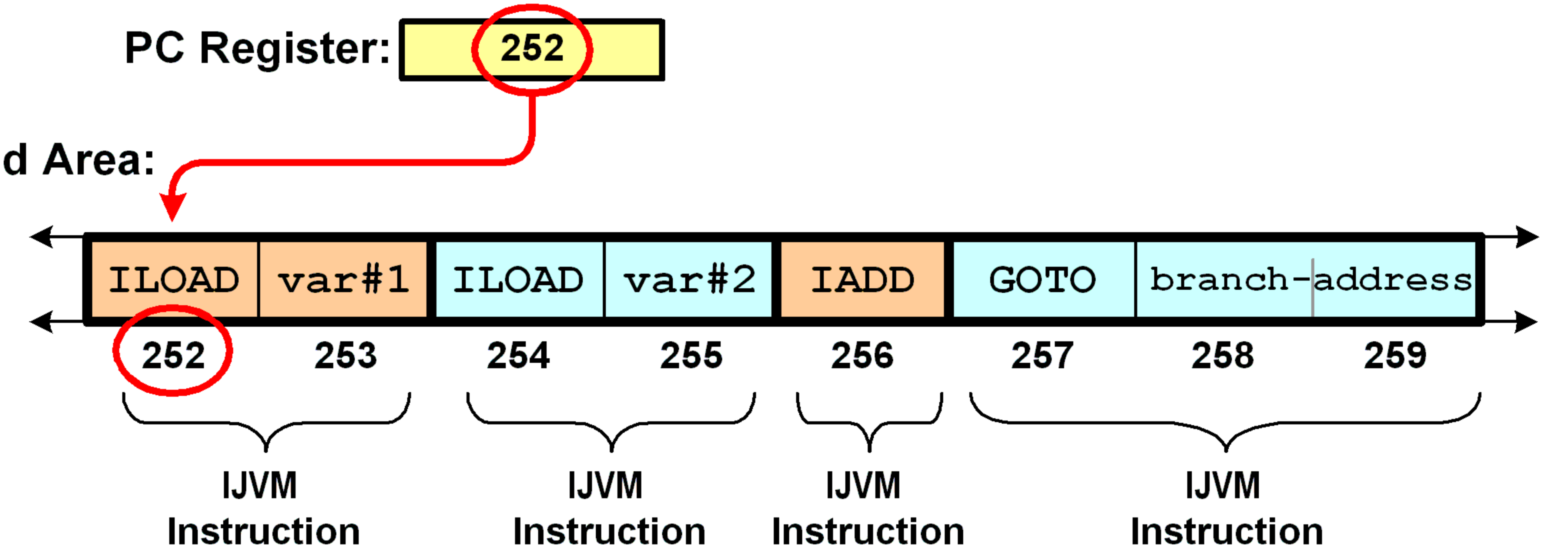
When sub2 exits it's local variables are removed from the stack and the LV register is adjusted. The same happens to sub1's local variables when it exits.

The area that holds the local variables called the "Local Variable Frame".



IJVM Instructions

The Java Virtual Machine executes IJVM instructions. These instructions take the form of "bytecodes" stored in the method area. Instructions are stored in sequential memory locations, and the PC register contains the address of the next instruction which is to be executed.



The first byte of each instruction is an "opcode" that identifies what the instruction is supposed to do. This is shown in the diagram above using mnemonic names like "ILOAD" and "IADD" – but in reality the opcode is a different unique pattern of bits for each instruction.

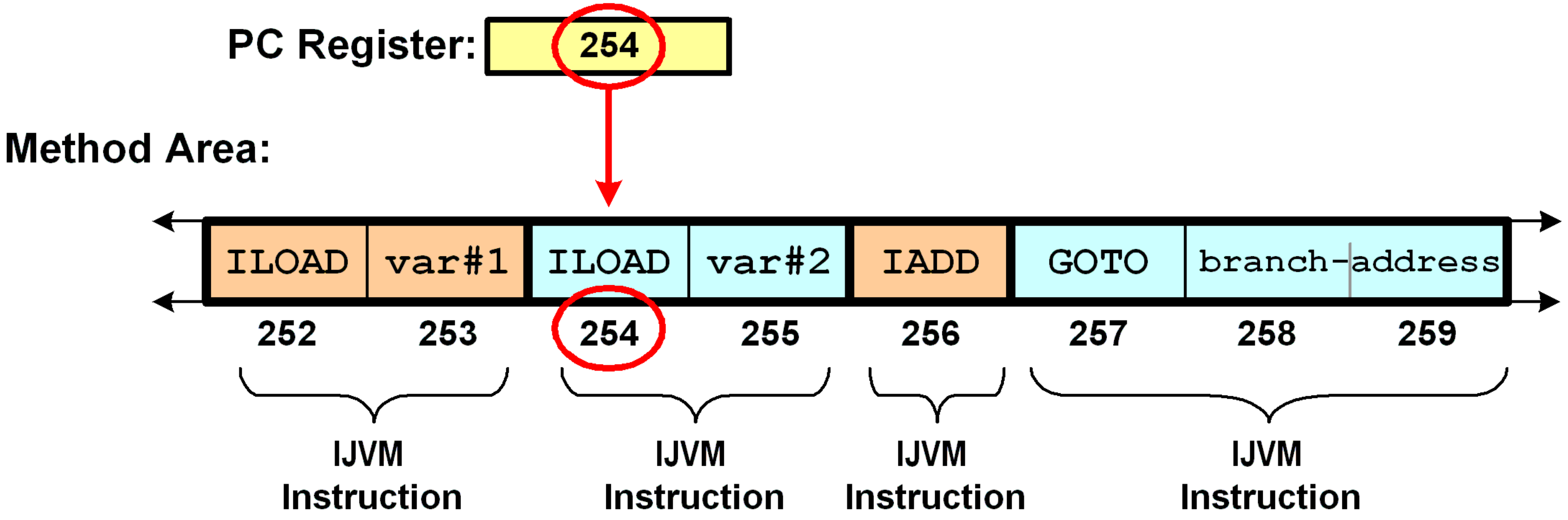
Some instructions have operands that identify what data they will use. For example, the first "ILOAD" opcode above is followed by a byte that contains a variable number. Instructions can have 0, 1 or 2 operands.

Operands can be 8 bits (1 byte) or 16 bits (2 bytes) long. 2-byte operands are stored in Big Endian format – in other words, the high-order portion of the operand comes first.

Executing IJVM Instructions

The CPU executes instructions by repeatedly following these steps:

- **The CPU fetches the information from the method area using the address that's in the PC Register**
- **The CPU examines the instruction and performs the work that the instruction specifies**
- **The CPU adjusts the PC register to point to the following instruction**



These actions cause the CPU to execute program instructions one after another. These steps will have to be done by our microarchitecture in order to execute the IJVM program.

Java Source Code and IJVM Instructions

The IJVM instructions that become part of a program are created by the Java compiler. It chooses sequences of instructions that will perform the work specified by Java source code.

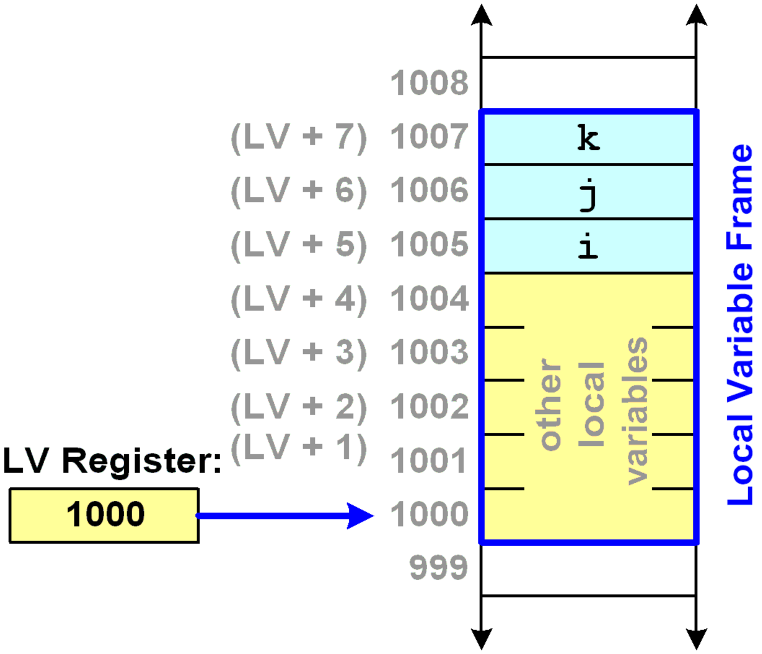
For example, consider the following Java source code:

```
int  i, j, k;
i = j + k;
```

The Java compiler does two things with this source code:

- **It decides where in memory the variables i, j, and k will be stored**
- **It decides which IJVM instructions are needed to add j and k together and store the result in i**

Let's assume that the compiler has decided to store the variables in words 5, 6 and 7 of the local variable frame:

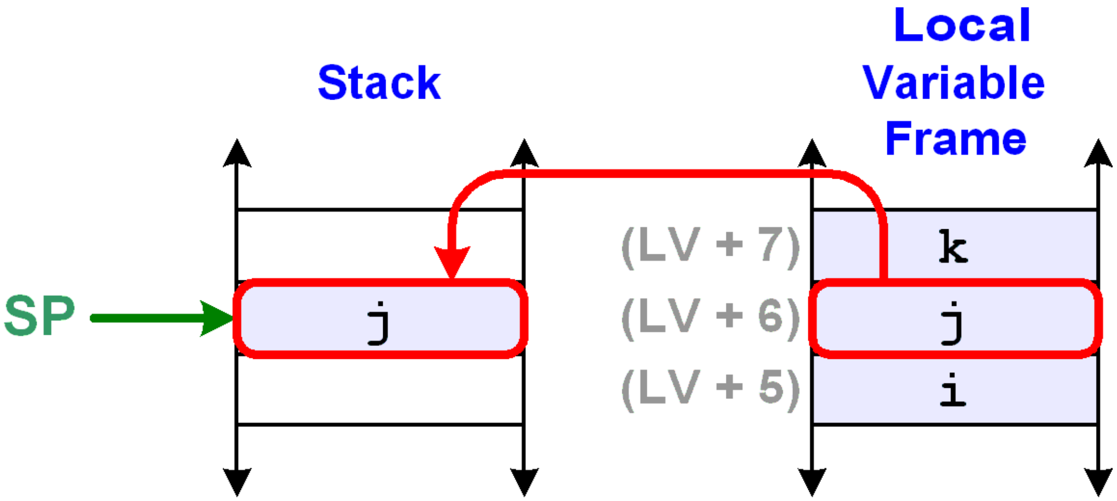


Java Source Code and IJVM Instructions

The Java compiler might use the following IJVM instructions to perform "i = j + k":

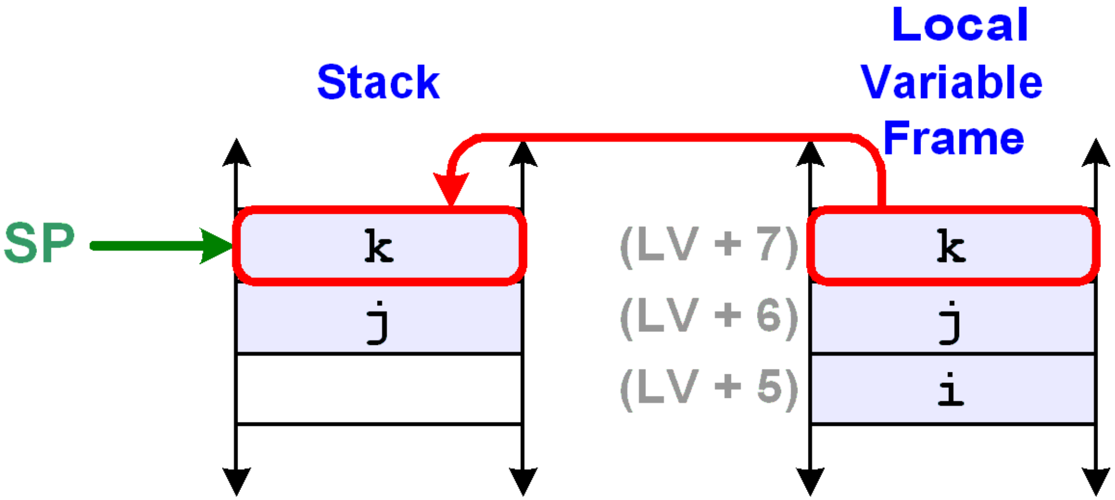
ILOAD j

This instruction pushes the value of "j" onto the top of the stack



ILOAD k

This instruction pushes the value of "k" onto the top of the stack



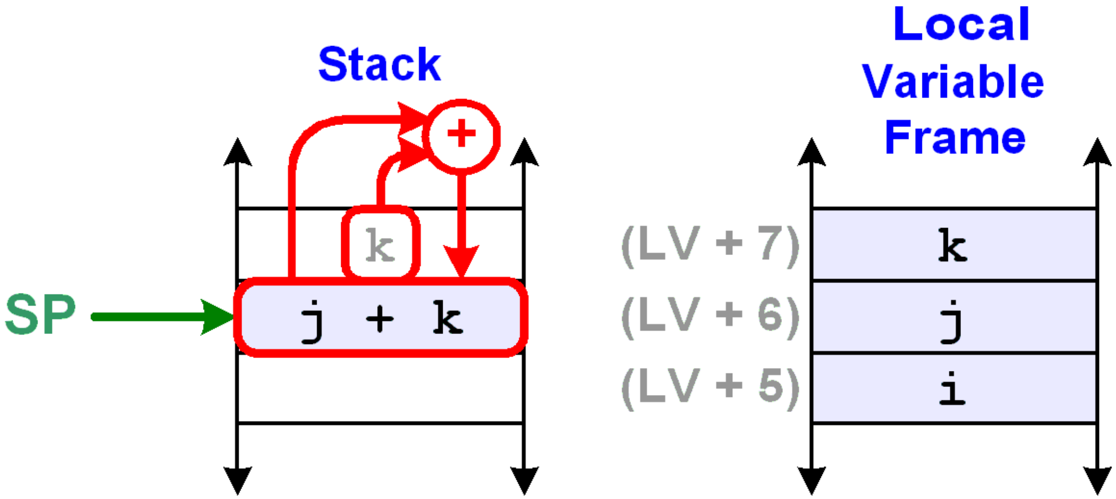
(continued on the next page…)

Java Source Code and IJVM Instructions

Now that the stack has the j and k variables on the stack, they can be added together and the result stored into the i variable:

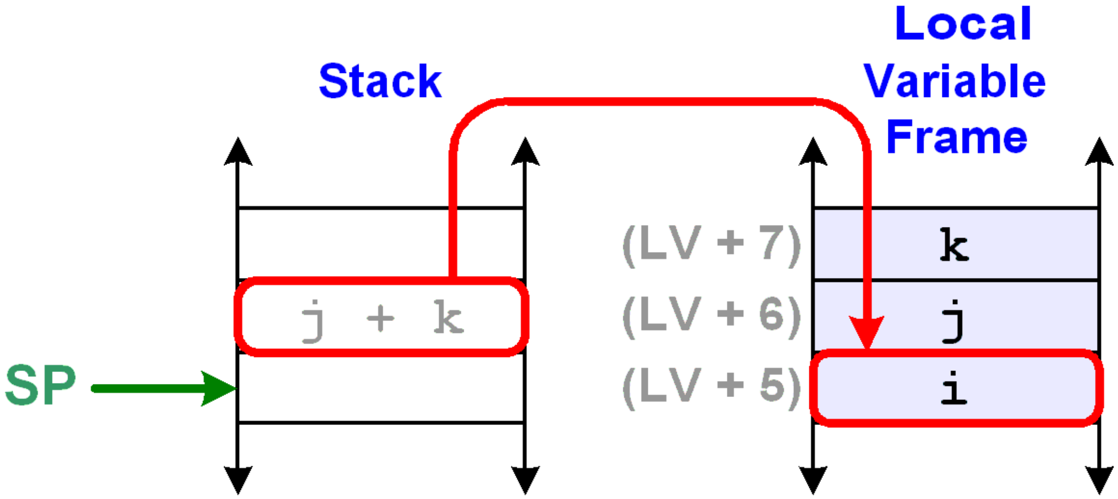
IADD

Pops the top two values off the stack, adds them together, and pushes the result back onto the stack again.



ISTORE i

Pops the top word off the stack and stores in in variable i.



Java Source Code and IJVM Instructions

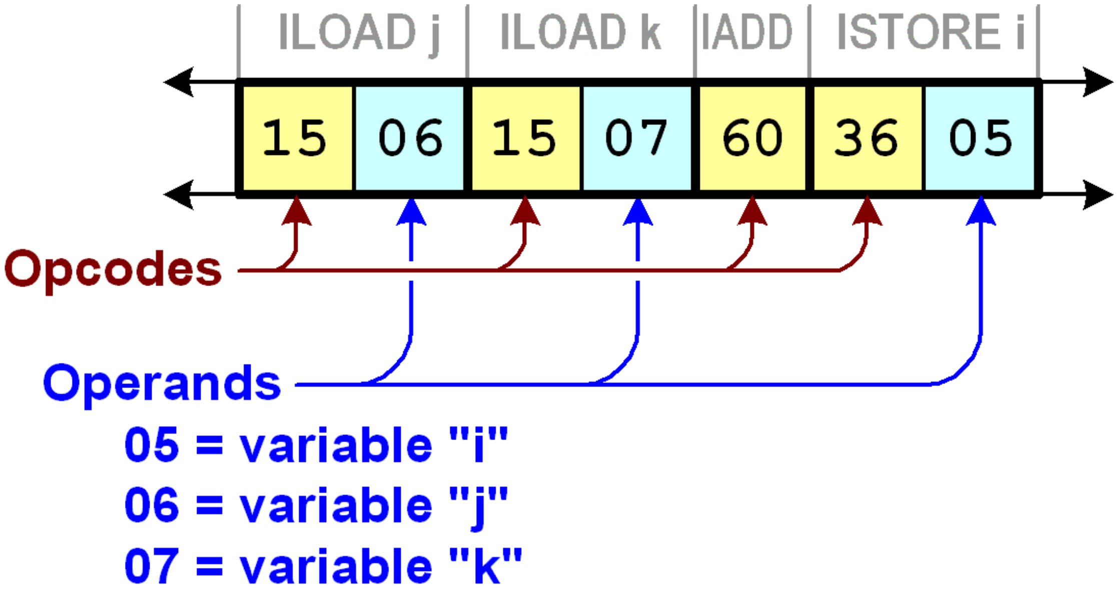
So this Java source statement:

```
i = j + k;
```

Would be turned into these IJVM instructions:

```
ILOAD j      (pushes variable j onto the stack)
ILOAD k      (pushes variable k onto the stack)
IADD         (adds top two words on stack)
ISTORE i     (pops result off of the stack and stores in variable i)
```

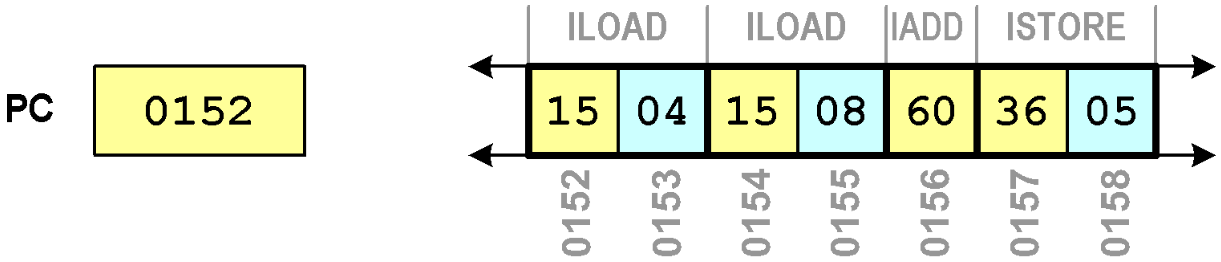
This is a symbolic representation of these instructions. What's really stored the method area is a series of bytes with binary values as shown below (all numbers shown in hexadecimal):



05 = variable "i"
06 = variable "j"
07 = variable "k"

Exercise 3 – Execution

The Java Virtual Machine is about to execute four instructions stored in the method area:

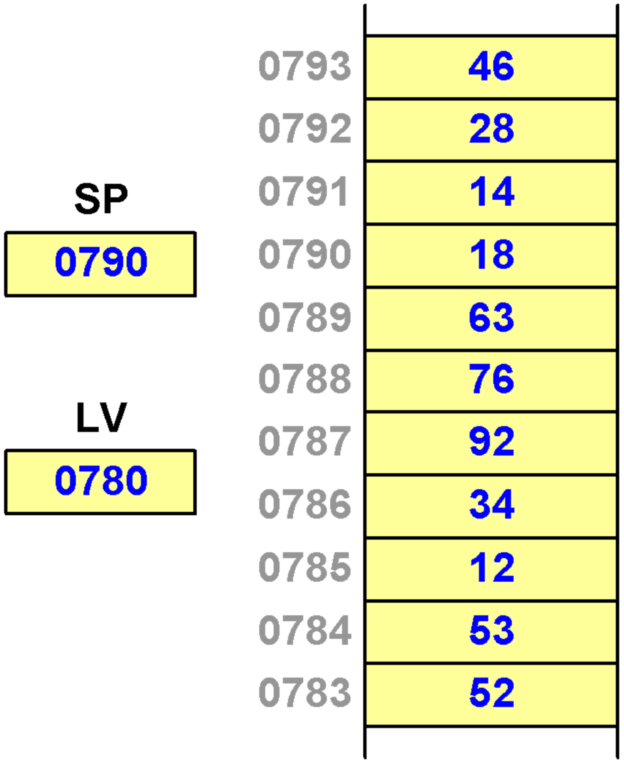


PC 0152

ILOAD		ILOAD		IADD	ISTORE	
15	04	15	08	60	36	05
0152	0153	0154	0155	0156	0157	0158

The initial contents of memory are shown at the right.
What will be the result of executing these four instructions?

- **A** **The SP register will be different**
- **B** **The LV register will be different**
- **C** **The PC register will be different**
- **D** **The memory word at address 0785 will be different**
- **E** **A, B, and C above are all true**
- **F** **A, C, and D above are all true**
- **G** **C and D above are both true**
- **H** **A and D above are both true**



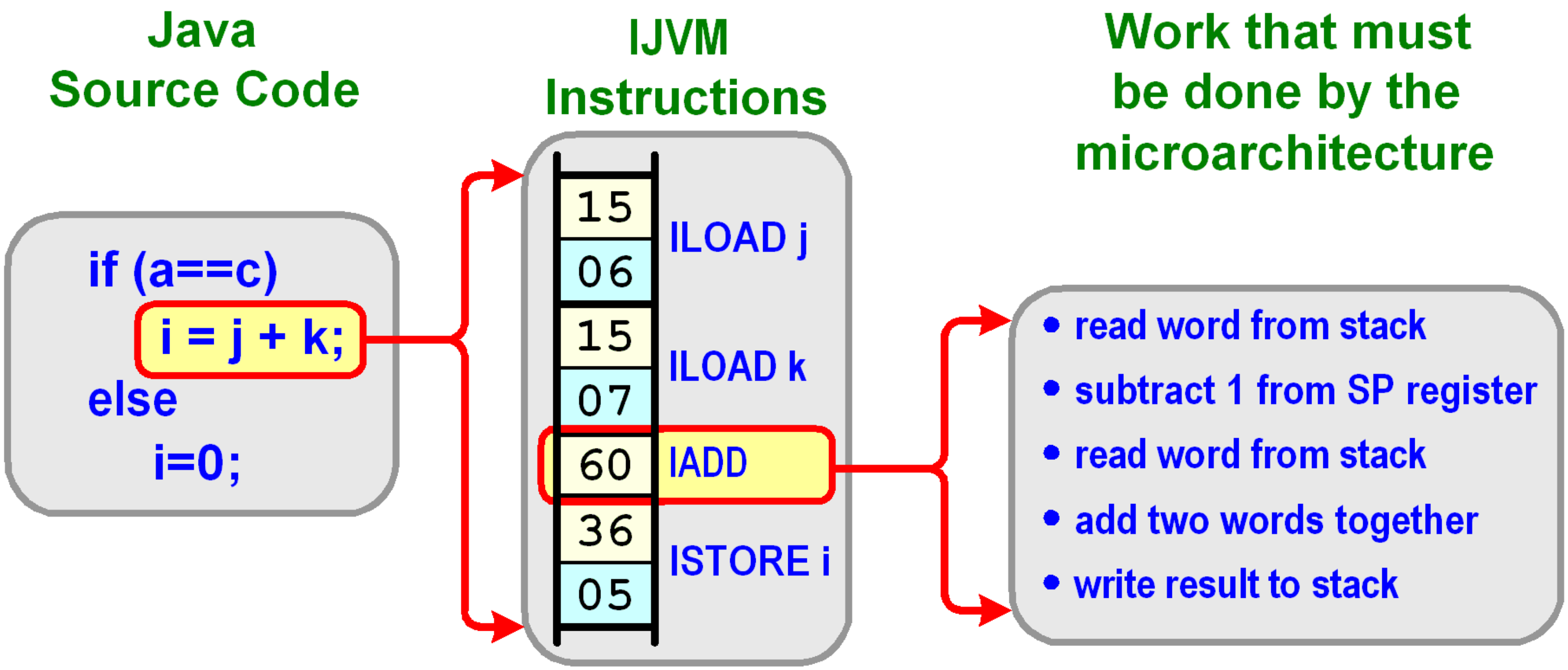
SP 0790

LV 0780

0793	46
0792	28
0791	14
0790	18
0789	63
0788	76
0787	92
0786	34
0785	12
0784	53
0783	52

From Source Code to Microarchitecture

So we’ve seen how Java source code is converted into a series of IJVM instructions that perform the equivalent work. What we need to do now is put together a microarchitecture which in turn does the work required by each of the IJVM instructions:



Java Source Code

```
if (a==c)
    i = j + k;
else
    i=0;
```

IJVM Instructions

```
15  ILOAD j
06
15  ILOAD k
07
60  IADD
36  ISTORE i
05
```

Work that must be done by the microarchitecture

- read word from stack
- subtract 1 from SP register
- read word from stack
- add two words together
- write result to stack

So, as indicated by the above diagram, when our microarchitecture reads an opcode from memory that contains the value hex “60”, it will have to perform the associated actions to do the work that’s supposed to be done for an IJVM “IADD” instruction.

The IJVM Instruction Set

The following table contains all of the instructions our microarchitecture will be able to execute. Note that the “mnemonic” column indicates the operands for each instruction as well as their size (in bits):

Opcode	Mnemonic	Description
0x10	BIPUSH *byte-8*	Push byte onto stack
0x59	DUP	Duplicate top word on the stack
0xA7	GOTO *offset-16*	Unconditional branch
0x60	IADD	Add top two words on the stack and replace with result
0x7E	IAND	Boolean AND two words on stack and replace with result
0x99	IFEQ *offset-16*	Pop word from stack and branch if it is zero
0x9B	IFLT *offset-16*	Pop word from stack and branch if it is < 0 (i.e. negative)
0x9F	IF_CMPEQ *offset-16*	Pop two words from stack and branch if they are equal
0x84	IINC *varnum-8* *const-8*	Add a constant to a local variable
0x15	ILOAD *varnum-8*	Push local variable onto the stack

(continued on the next page…)

The IJVM Instruction Set

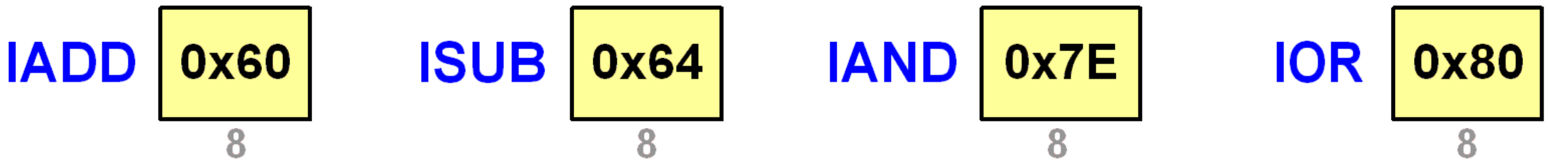
Opcode	Mnemonic	Description
0xB6	INVOKEVIRTUAL *disp-16*	Invoke a method
0x80	IOR	Boolean OR two words on stack and replace with result
0xAC	IRETURN	Return from method with integer result
0x36	ISTORE *varnum-8*	Pop word from stack and store in local variable
0x64	ISUB	Subtract top word on stack from previous, push result
0x13	LDC_W *index-16*	Push constant pool word onto the stack
0x00	NOP	Do nothing
0x57	POP	Remove top word from stack
0x5F	SWAP	Swap the top two words on the stack
0xC4	WIDE	Prefix: following instruction uses 16-bit operand

The actual Java Virtual Machine has many more opcodes – but these are enough to be able to write an actual Java program and to demonstrate how to design a microarchitecture.

These instructions form the specification for the microarchitecture we're going to build. Now we'll look at each instruction in detail so we know what our microarchitecture needs to do in order to execute it.

IJVM Arithmetic / Logical Instructions

Instruction formats:



IADD 0x60 8

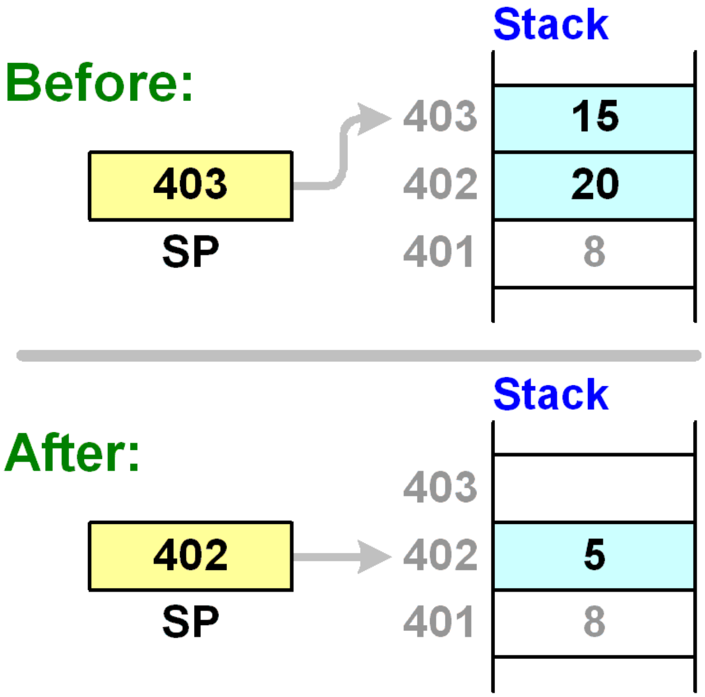
ISUB 0x64 8

IAND 0x7E 8

IOR 0x80 8

Operation:

- Two words are popped off the stack
- The words are added, subtracted, ANDed or ORed together
- The result is pushed onto the stack



Before:

Stack

403	15
402	20
401	8

403 SP

After:

Stack

403	
402	5
401	8

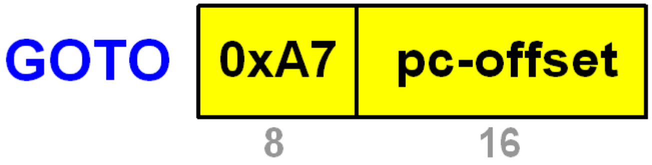
402 SP

The example at right is for the ISUB instruction. Note that the value most recently pushed onto the stack is the “subtrahend” – it is subtracted from the previous value (the “minuend”):

$$20 - 15 = 5$$

IJVM “Branch” Instruction

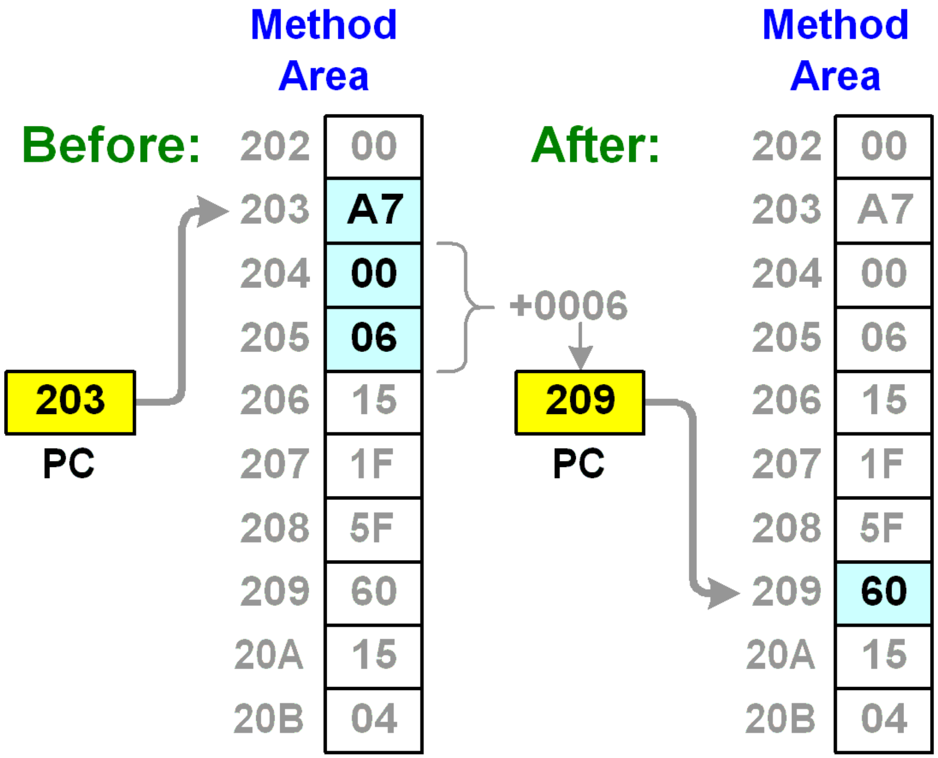
Instruction format:



Operation:

- The 16-bit operand is treated as a signed number.
- The number is added to the PC register
- The PC register now points to a new instruction – this new instruction will be the next one to be executed

Since the operand is a *signed* number, the number in the PC register may be increased (when the operand is positive) or decreased (when the operand is negative).   This means that the instruction can be used to jump forward to subsequent instructions or backward to previous instructions.



	Method Area
202	00
203	A7
204	00
205	06
206	15
207	1F
208	5F
209	60
20A	15
20B	04

	Method Area
202	00
203	A7
204	00
205	06
206	15
207	1F
208	5F
209	60
20A	15
20B	04

IJVM “Conditional Branch” Instructions

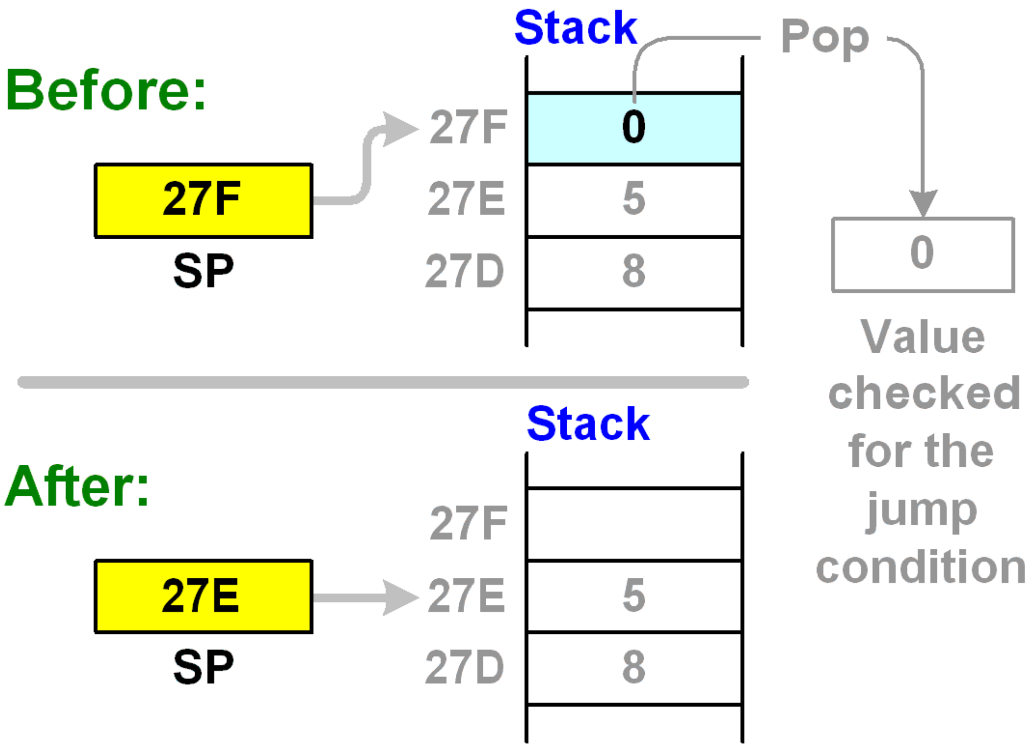
Instruction formats:

IFEQ	0x99	pc-offset
	8	16

IFLT	0x9B	pc-offset
	8	16

Operation:

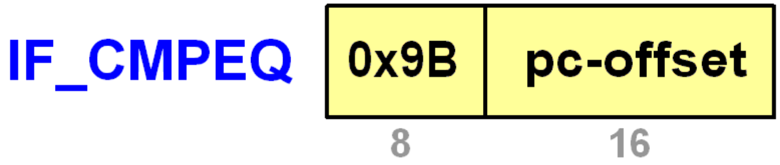
- The top word is popped off the stack and checked to see if it matches the branch condition
- For IFEQ, a branch is taken if the number is zero
- For IFLT, a branch is taken if the number is negative
- If a branch is taken, the 16-bit signed offset is added to the PC and the effect is the same as described for GOTO on the previous page.



- If no branch is taken, then the instruction following this one is executed next.

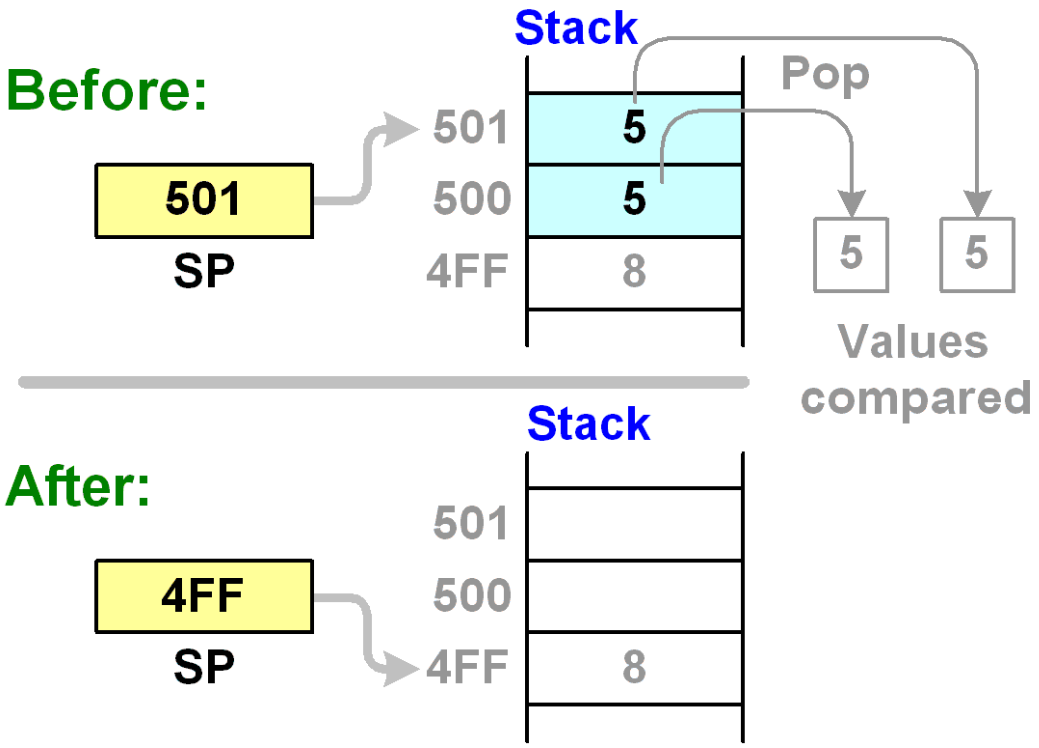
IJVM “Conditional Branch” Instruction

Instruction format:



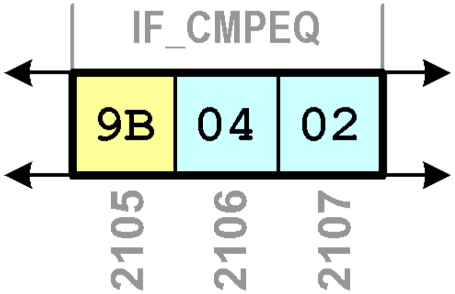
Operation:

- The top two words are popped off the stack and checked to see if they both contain the same value
- If the numbers match, a branch is taken by adding the 16-bit signed offset is added to the PC. The effect is the same as described for the GOTO instruction.
- If no branch is taken, then the instruction following this one is executed next.



Exercise 4 – Conditional Branch

The method area contains the following IF_CMPEQ instruction at address 2105:



If the top two words on the stack are equal when this instruction is executed, from which address will the CPU fetch the next instruction to be executed?

A – 0402

B – 0204

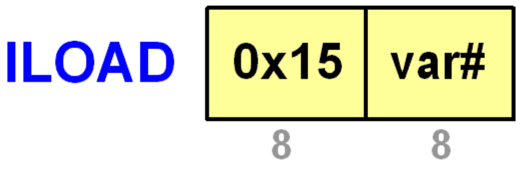
C – 2108

D – 2309

E – 2507

IJVM “Load Local Variable” Instruction

Instruction format:



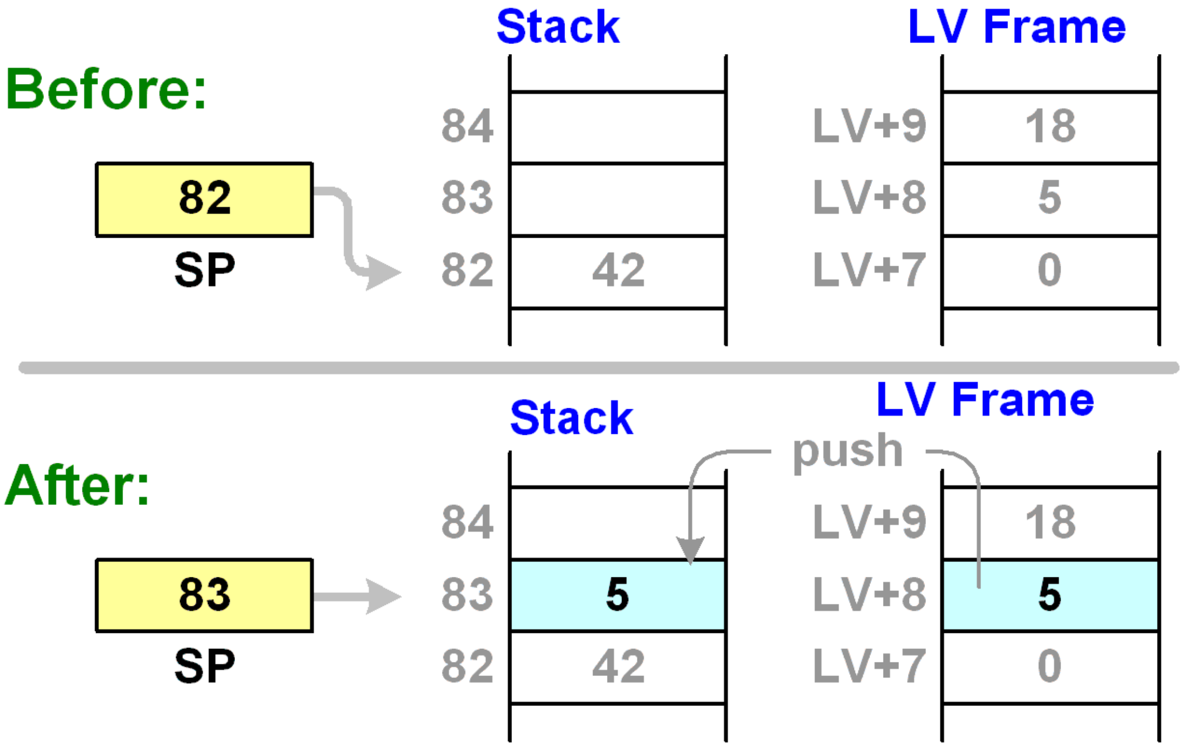
ILOAD

0x15	var#
8	8

Operation:

Note - the example shows an “ILOAD 8” instruction

- **The operand is an unsigned variable number in the local variable frame.**
- **The value of the variable is located by adding the operand to the LV pointer and reading the word from the resulting memory location**
- **The variable’s value is pushed onto the stack**



Before:

SP: 82

Stack	
84	
83	
82	42

LV Frame	
LV+9	18
LV+8	5
LV+7	0

After:

SP: 83

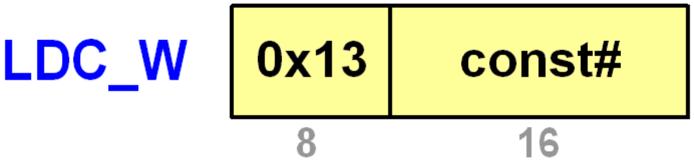
push

Stack	
84	
83	5
82	42

LV Frame	
LV+9	18
LV+8	5
LV+7	0

IJVM “Load Constant” Instruction

Instruction format:



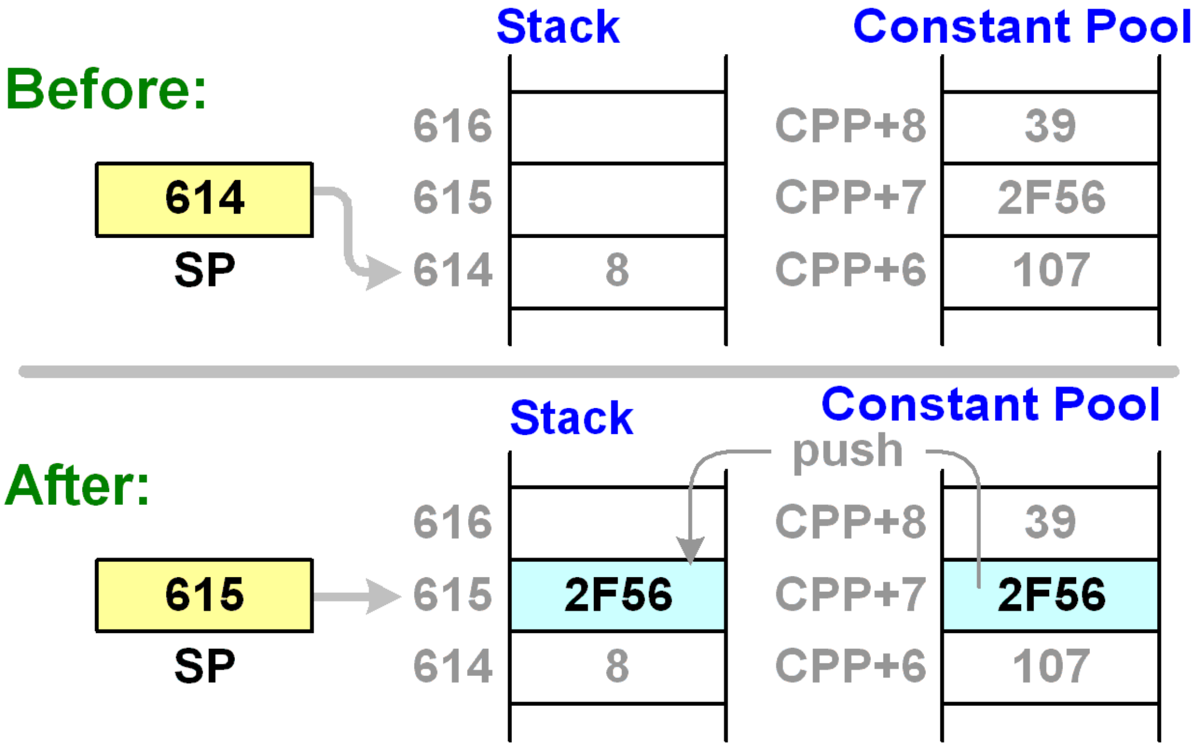
LDC_W

0x13	const#
8	16

Operation:

Note - the example shows an “LDC_W 7” instruction

- The operand is an unsigned variable number in the constant pool.
- The value of the constant is located by adding the operand to the CPP register and reading the word from the resulting memory location
- The constant value is pushed onto the stack



Before:

SP: 614

Stack		Constant Pool	
616		CPP+8	39
615		CPP+7	2F56
614	8	CPP+6	107

After:

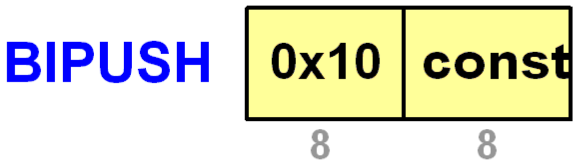
SP: 615

push

Stack		Constant Pool	
616		CPP+8	39
615	2F56	CPP+7	2F56
614	8	CPP+6	107

IJVM “Load Immediate” Instruction

Instruction format:

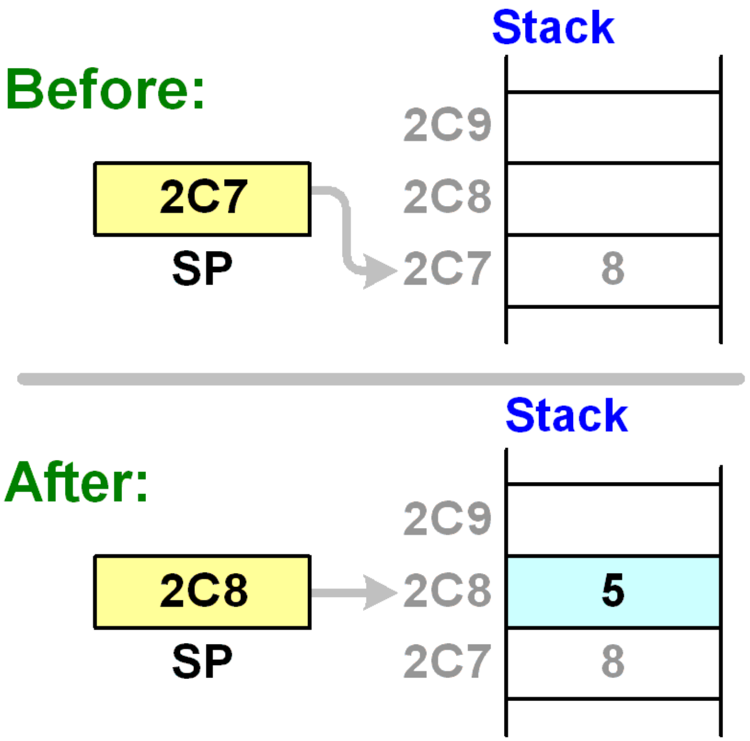


Operation:

Note - the example shows a “BIPUSH 5” instruction

- **The operand is the signed value to be pushed onto the stack. This value is stored as part of the instruction itself and there is no need to read it from memory. (Values that are stored as part of the instruction itself are known as “immediate” values).**
- **The value is pushed onto the stack**

Note that the value is a signed number, so it must be sign-extended from 8 bits to 32 bits when it is pushed onto the stack.



Exercise 5 – BIPUSH

The Java Virtual Machine is about to execute a “BIPUSH 8” instruction. If memory holds the values shown on the right, what is the value of the word that’s pushed onto the top of the stack?

A – 8

B – 20

C – 28

D – 3370

E – 3381

F – 3378

G – 3389

LV

3381

CPP

3370

3390	46
3389	28
3388	14
3387	18
3386	63
3385	76
3384	92
3383	34
3382	12
3381	53
3380	52
3379	63
3378	20
3377	83
3376	7
3375	17

IJVM “Store Local Variable” Instruction

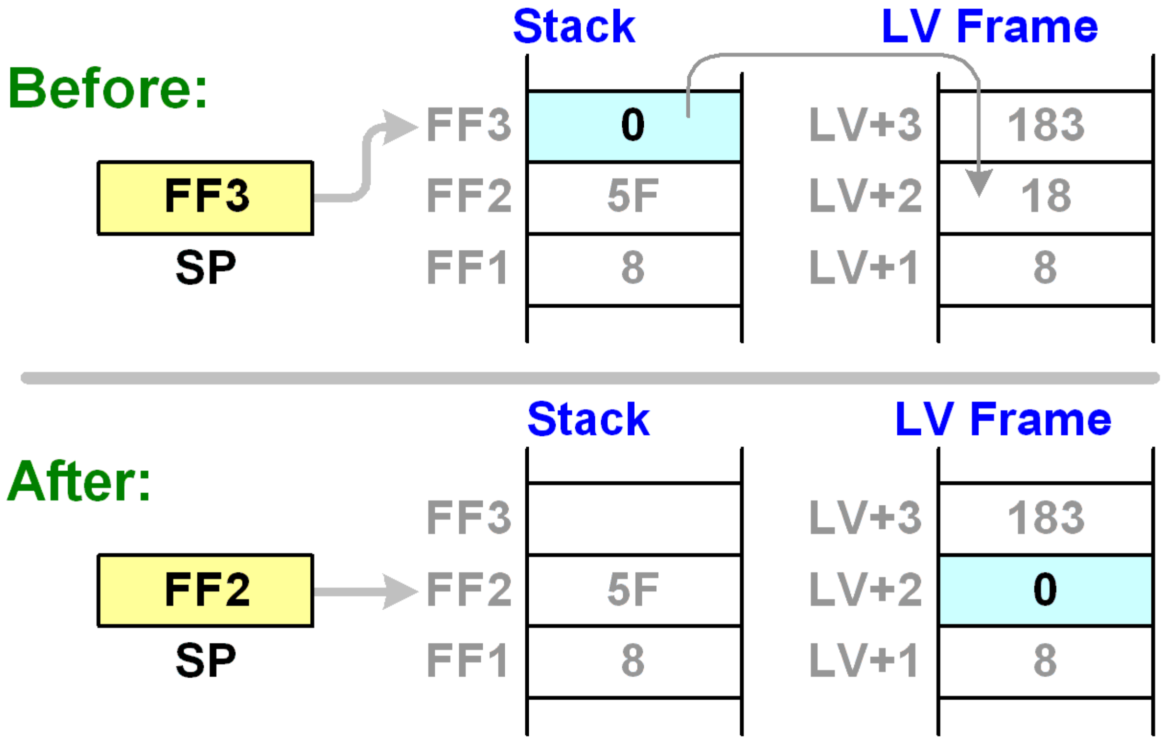
Instruction format:

ISTORE	0x36	var#
	8	8

Operation:

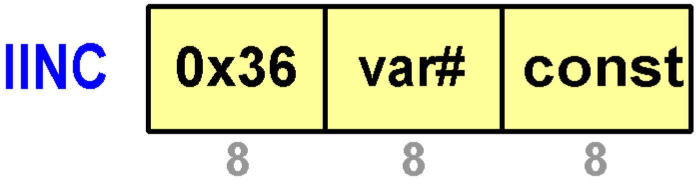
Note - the example shows an “ISTORE 2” instruction

- **The operand is an unsigned variable number in the local variable frame.**
- **The top word is popped off of the stack**
- **The variable is located by adding the operand to the LV pointer, and the popped value is written to the resulting memory location**



IJVM “Increment Local Variable” Instruction

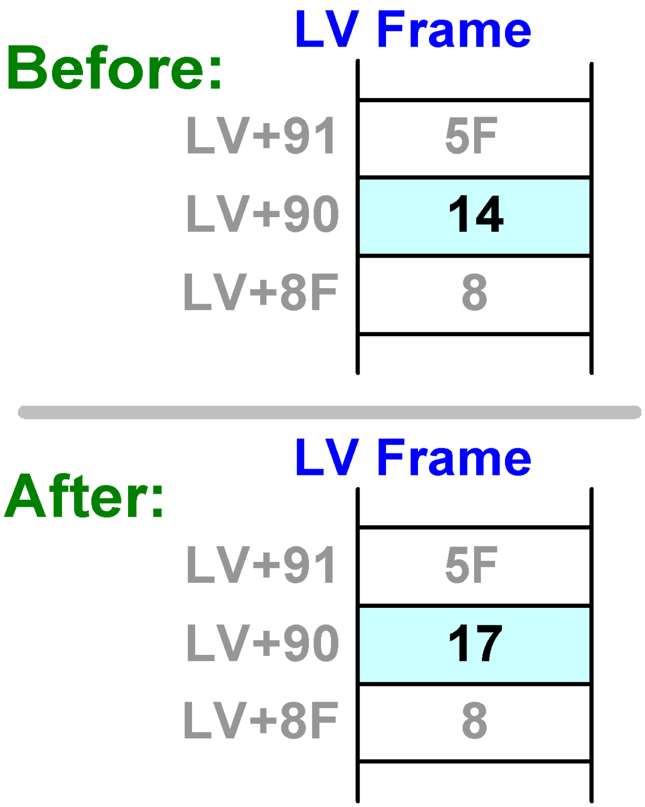
Instruction format:



IINC

0x36	var#	const
8	8	8

Operation:



Before:

LV Frame

LV+91	5F
LV+90	14
LV+8F	8

After:

LV Frame

LV+91	5F
LV+90	17
LV+8F	8

Note - the example shows an “IINC 90 3” instruction

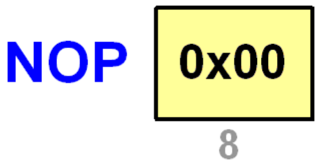
- **The first operand is an unsigned variable number in the local variable frame. This is added to the LV pointer to find the variable in memory.**
- **The second operand is a signed immediate value that is added to the local variable.**

This is an unusual IJVM instruction because it can perform arithmetic on a local variable without using the stack.

IJVM “No-Operation” Instruction

4.2.3

Instruction format:



NOP 0x00

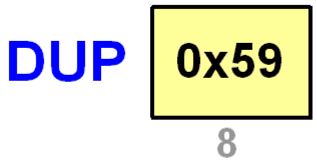
8

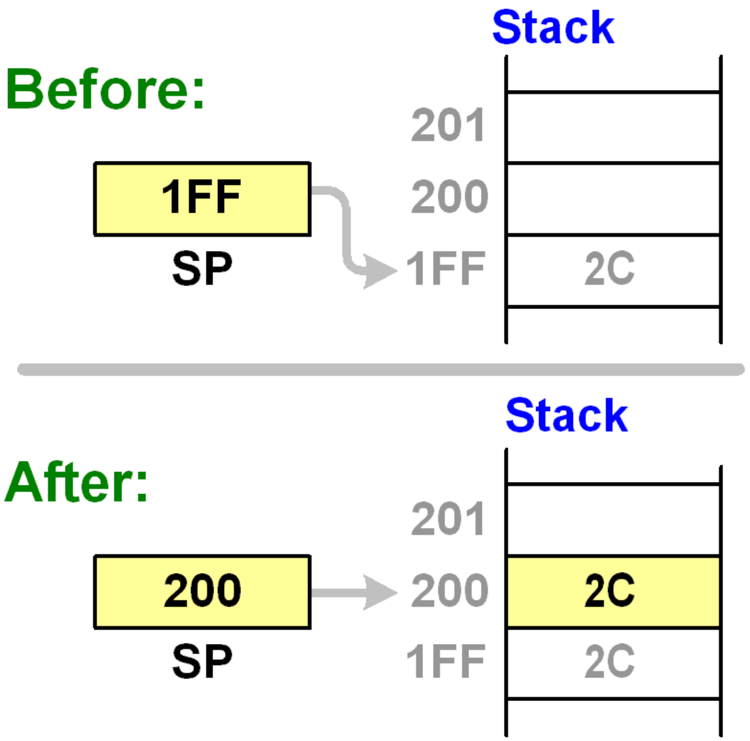
Operation:

This instruction doesn’t do anything.

IJVM “Duplicate Top of Stack” Instruction

Instruction format:



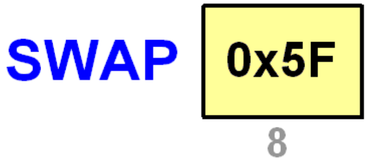


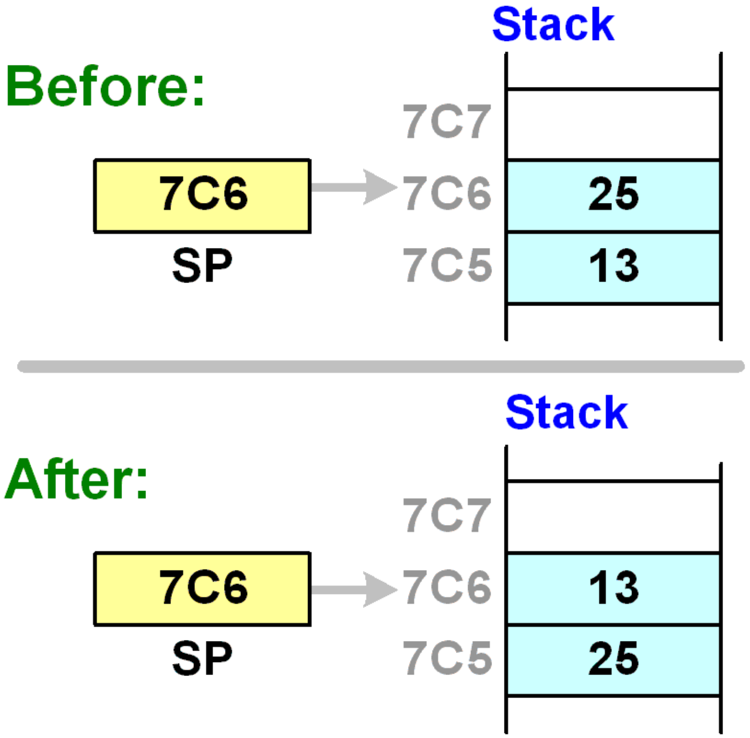
Operation:

- The value on the top of the stack is pushed onto the stack again, so that now two copies of the same value exist on the stack

IJVM “Swap Top of Stack” Instruction

Instruction format:



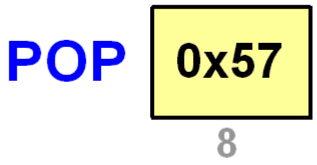


Operation:

- The top two values are popped off the stack and then pushed back on in the reverse order

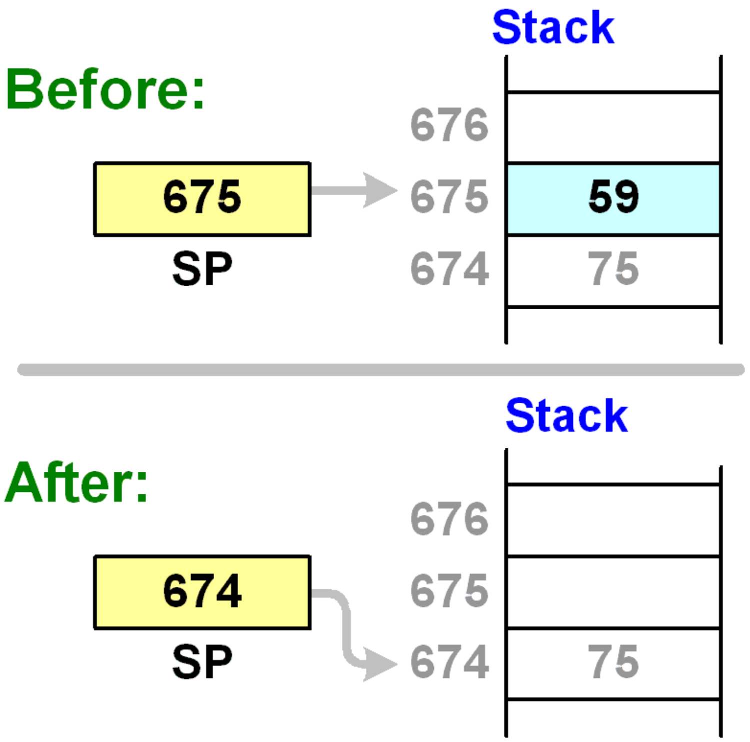
IJVM “Pop” Instruction

Instruction format:



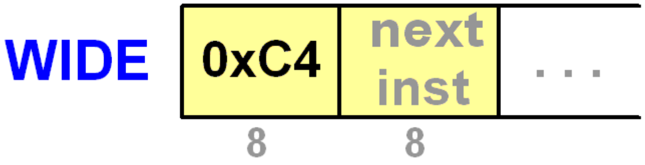
Operation:

- The top value is popped off the stack. The value is not used or stored anywhere – it simply disappears.



IJVM “WIDE” Prefix

Instruction format:

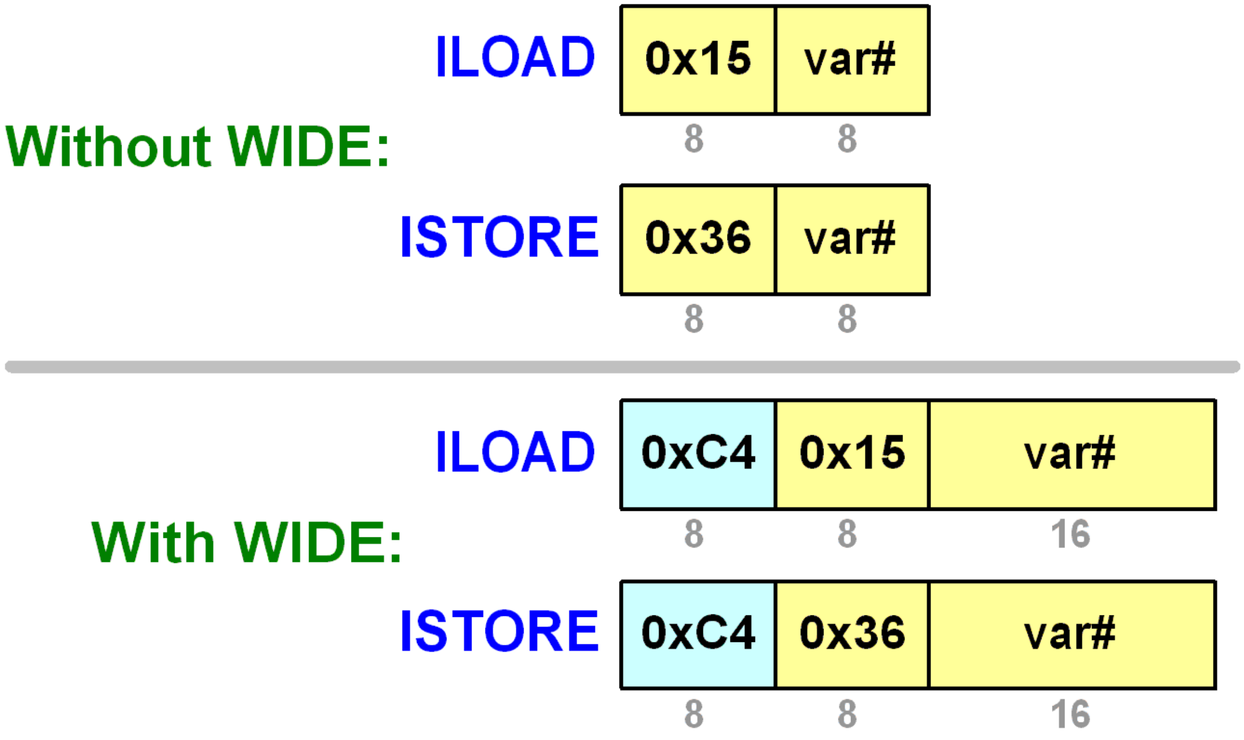


Operation:

“**WIDE**” is not an instruction per se, but rather a “prefix” byte which is placed in front of another instruction opcode to change it’s meaning.

When the “**WIDE**” prefix comes before an instruction, the size of that instruction’s operand changes from **8 bits** to **16 bits**.

This allows the instruction to access local variables beyond variable number 255.



Exercise 6 – The Wide Prefix

Which of the following statements is true?

A **The WIDE prefix allows a Java program to handle variables whose values are larger than 256**

B **The WIDE prefix allows a Java program to have more than 256 local variables**

C **The WIDE prefix allows a Java program to have more than 256 items on the stack**

D **The WIDE prefix allows a Java program to have more than 256 instructions**

Calling a Subroutine

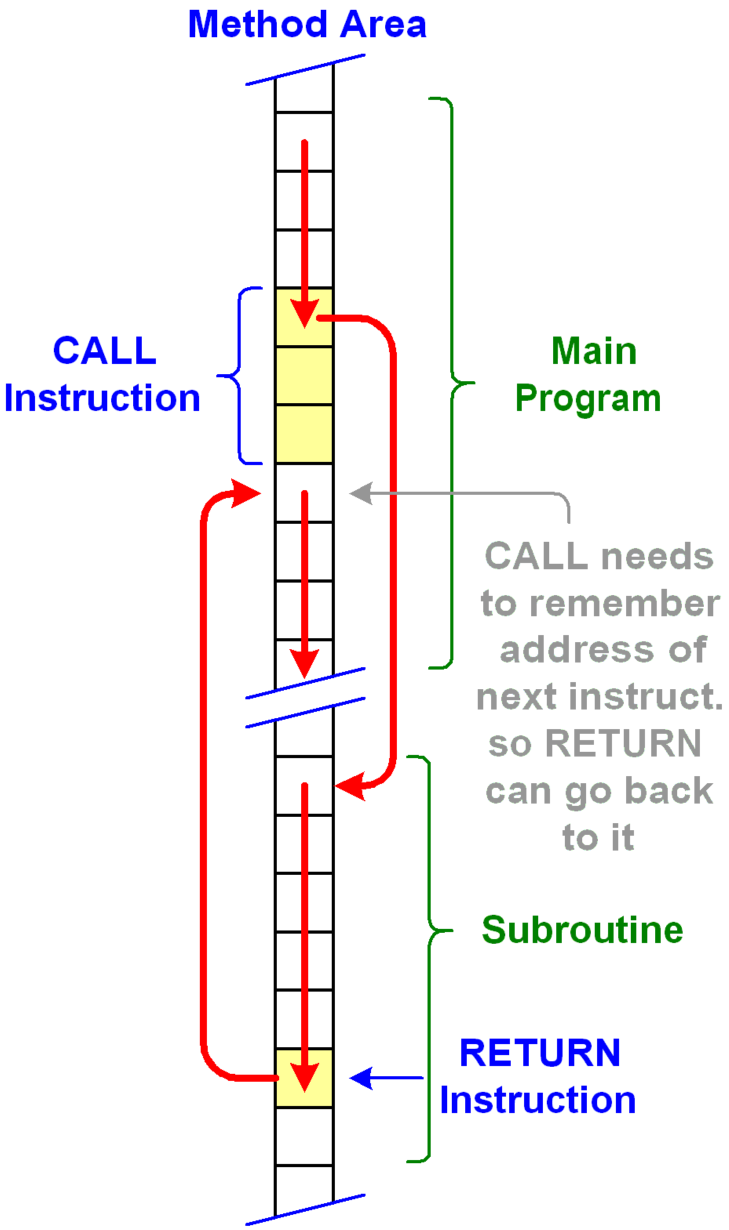
The last two IJVM instructions are used to call a subroutine and to return back to the calling program. In order to understand these two instructions, we need to understand how calls and returns work.

A call statement is much like a branch statement in that it transfers execution to a different set of instructions in the method area. But unlike a simple branch statement, a call statement must "remember" where it was so that the subroutine can return to the same point once it's finished.

To "remember" where the call came from, the call statement pushes the address of the next instruction onto the stack.

When the subroutine has finished executing, the RETURN instruction pops the return address off the stack and puts it into the PC register, and as a result the system carries on executing the instructions that followed the CALL statement.

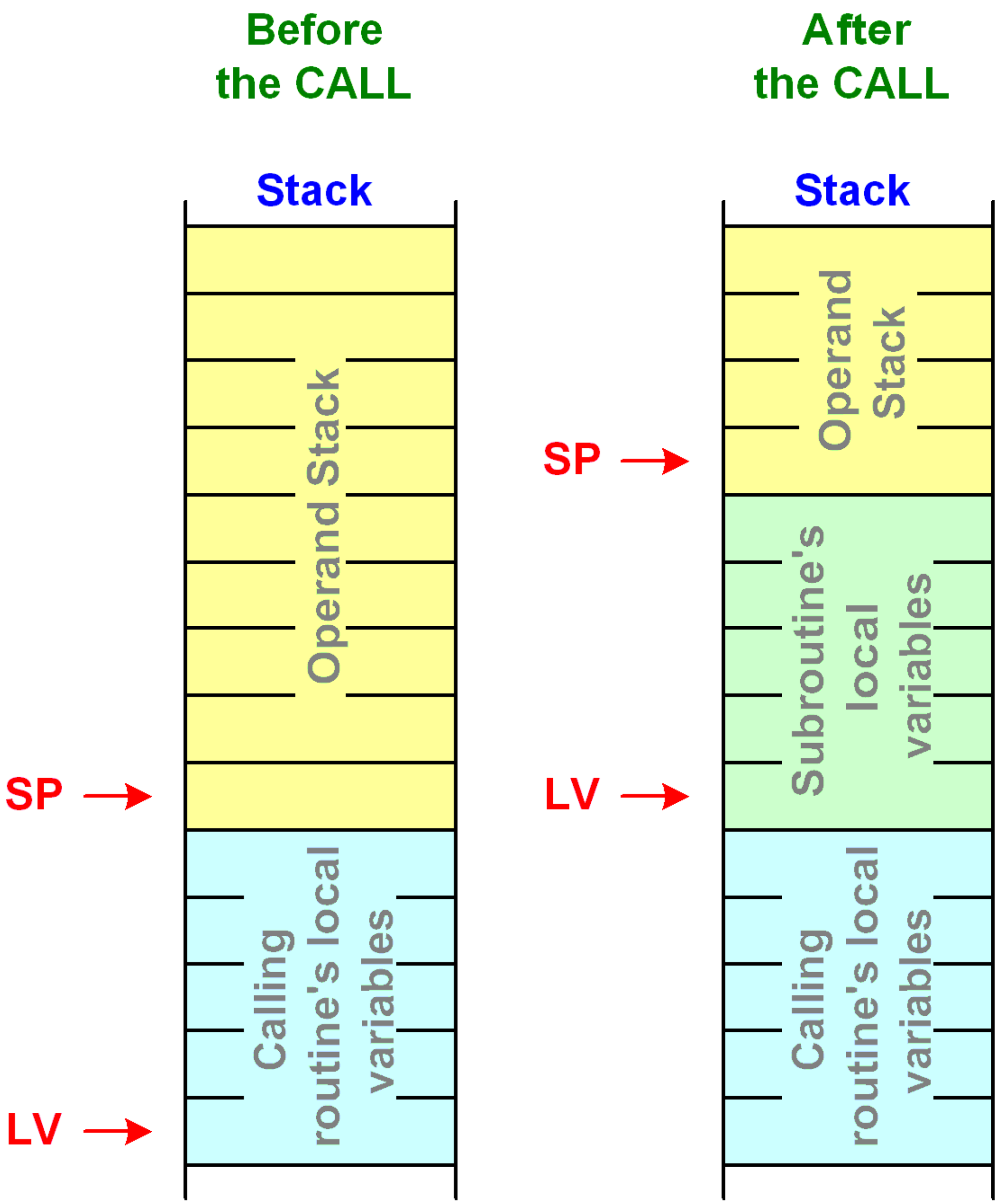
Using the stack makes nested subroutine calls very easy to handle, because the last return address pushed onto the stack is the first one popped off of it (because the stack is a LIFO list).


Method Area
CALL Instruction
Main Program
CALL needs to remember address of next instruct. so RETURN can go back to it
Subroutine
RETURN Instruction

Calling a Subroutine

In addition to saving the return address, the CALL instruction must also create a local variable frame to hold the variables used by the subroutine, and it must adjust the LV pointer to point to that frame.   In order to do this, the CALL instruction has to know how many local variables the subroutine requires.

The RETURN instruction will have to remove the local variable frame and restore the original value of the LV pointer. To make this possible, the CALL instruction must also save the original LV value on the stack alone with the return address.

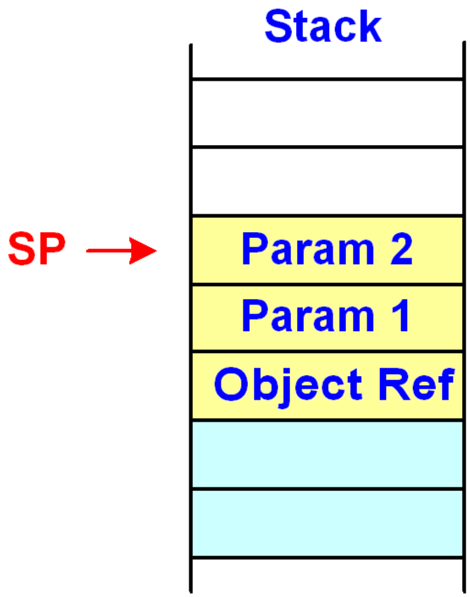


Calling a Subroutine

In IJVM, when you call a subroutine you must first do the following:

- **Push an object reference (the "this" pointer) onto the stack. The object reference is a pointer to the type of object that is being called. This is necessary because the same method code can be used for different polymorphic object types.**
- **Push any required parameters onto the stack. The subroutine will read and use these parameters.**

Once this is done you use the INVOKEVIRTUAL instruction to actually call the subroutine. So the stack will look like this when the INVOKEVIRTUAL instruction is executed:



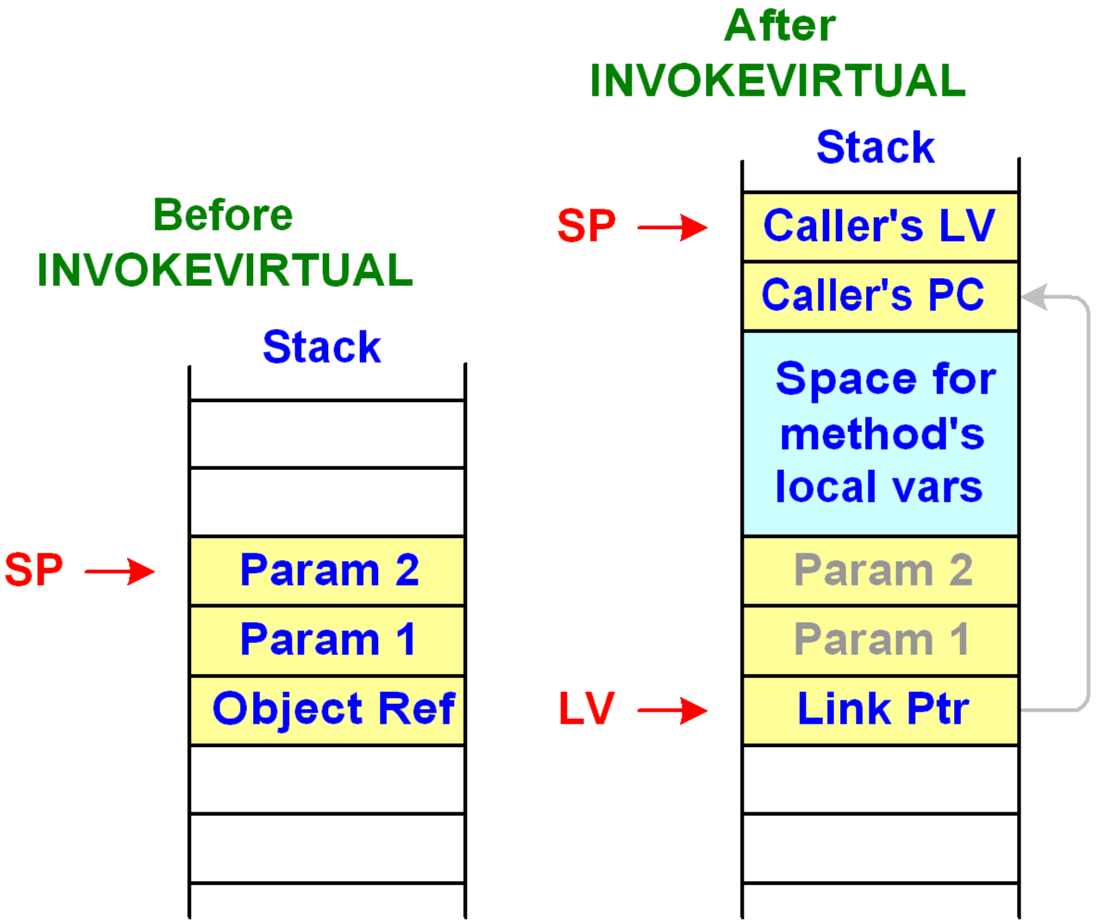
The IJVM “INVOKEVIRTUAL” Instruction

Instruction format:

INVOKEVIRTUAL	0xB6	method reference
	8	16

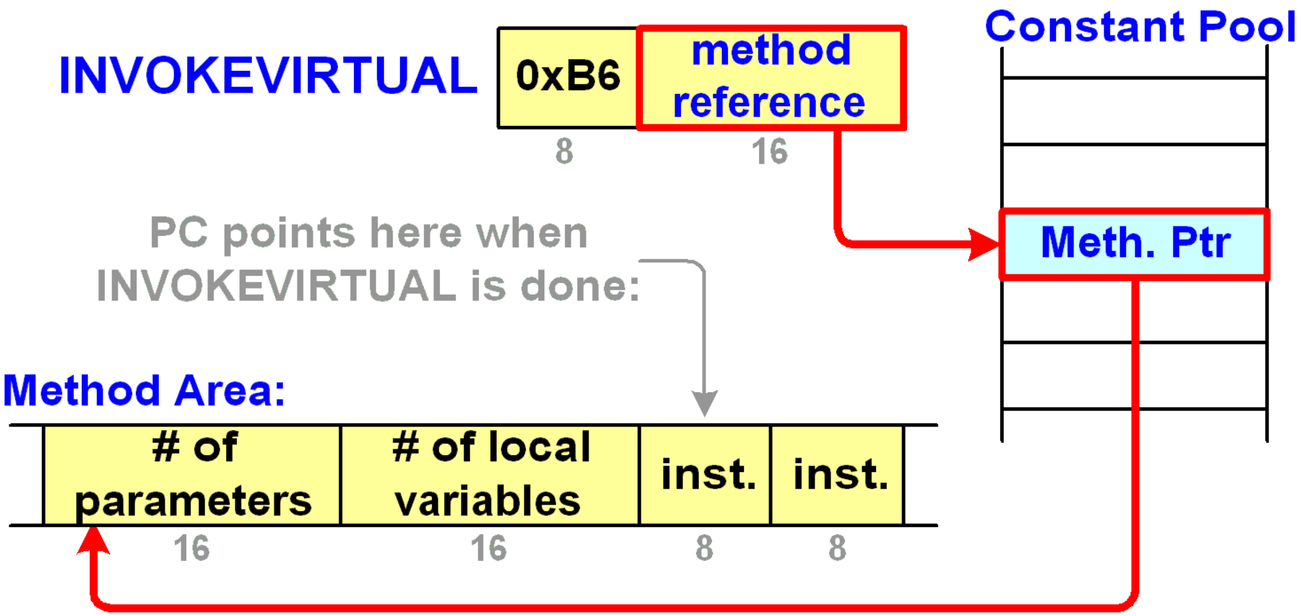
Operation:

- The method reference is used to find the method and determine the number of parameters it takes and the size required for it’s local variables (see next page)
- A new stack frame is built and the caller’s LV and PC register values are saved for use by the IRETURN instruction
- The object reference is replaced by a link pointer
- Control is passed to the first instruction in the method



Finding the Method

The “method reference” in the INVOKEVIRTUAL instruction is not a pointer to the method itself, but rather it’s a pointer to the method pointer stored in the Constant Pool.



The Constant Pool method pointer points to the start of the method in the method area. The method’s instructions are prefixed by two 16-bit words which contain:

- **The number of parameters that should be passed to this method**
- **The number of local variables that the method requires**

This information is used by INVOKEVIRTUAL when it builds the stack frame for the method.

Returning from a Subroutine

To return from a subroutine, the program must do the following:

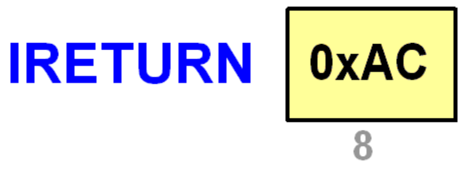
- **Push the return value onto the stack.**
- **Issue the IRETURN instruction**

The IRETURN instruction undoes the work done by the INVOKEVIRTUAL instruction and returns control to the program instruction immediately following it. This includes:

- **Restoring the original LV register to point to the local variable frame for the calling procedure**
- **Removing the method’s local variable frame from the stack**
- **Placing the return value on the stack so that it can be used by the calling procedure**
- **Putting the return address into the PC register so that control resumes in the main procedure at the instruction following INVOKEVIRTUAL.**

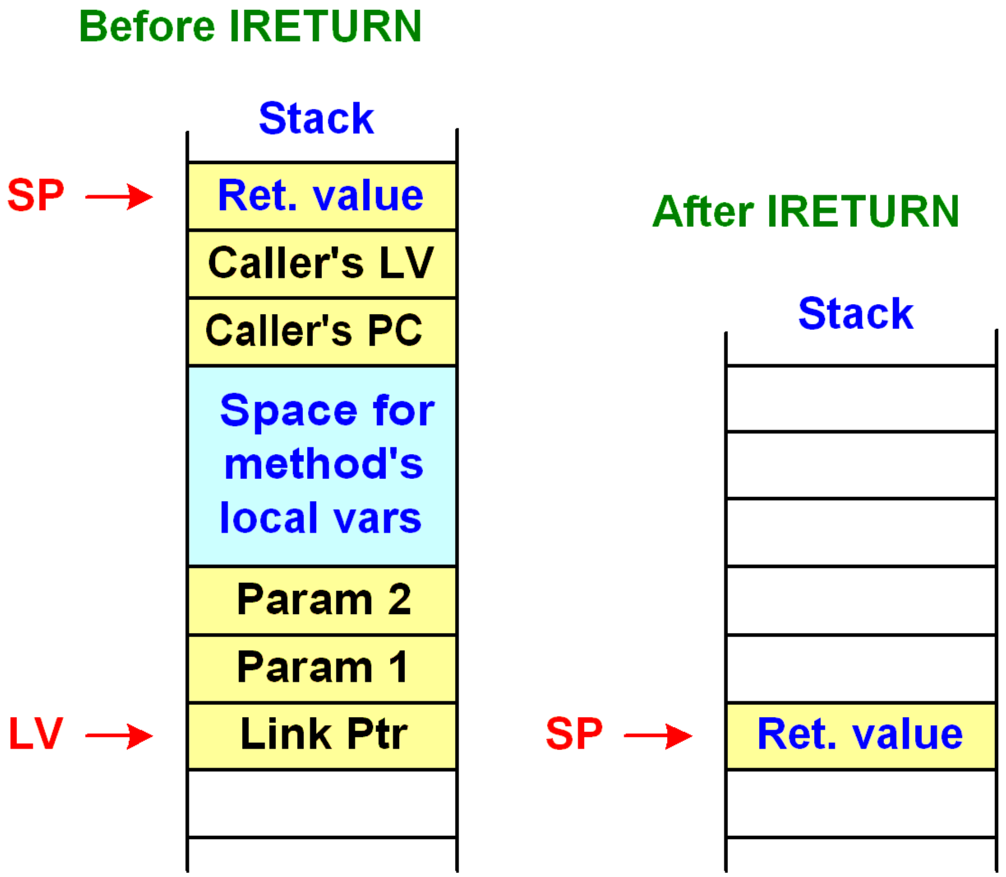
The IJVM “IRETURN” Instruction

Instruction format:



Operation:

- The called methods local variable frame is removed from the stack
- The SP and LV registers are restored to the original values they had before the INVOKEVIRTUAL instruction was issued
- The method’s return value is pushed onto the stack



General Observations about IJVM Instructions

Note the following points about the IJVM instructions we've just studied:

- **The first byte of each instruction is always the op-code – a byte containing a unique binary pattern that identifies what the instruction is supposed to do.**
- **The op-code may be followed by one or two operands**
- **Operands can be 8 or 16 bits in length**
- **Operands can contain immediate values (values to be used as the data), or they can contain pointers to the local variable frame or constant pool, or for branch instructions, a signed offset to be added to the PC**

Note the different operand sizes – for example, the ILOAD and ISTORE instructions use an 8-bit variable number, but this can be expanded to a 16-bit variable number with the use of the WIDE prefix. This allows the code to be more compact when referencing the first 256 local variables.

Also note that there are no I/O instructions. All I/O in Java is done by calling special methods that are part of the JVM environment. This helps to avoid security problems with code that attempts to do unauthorized I/O operations.

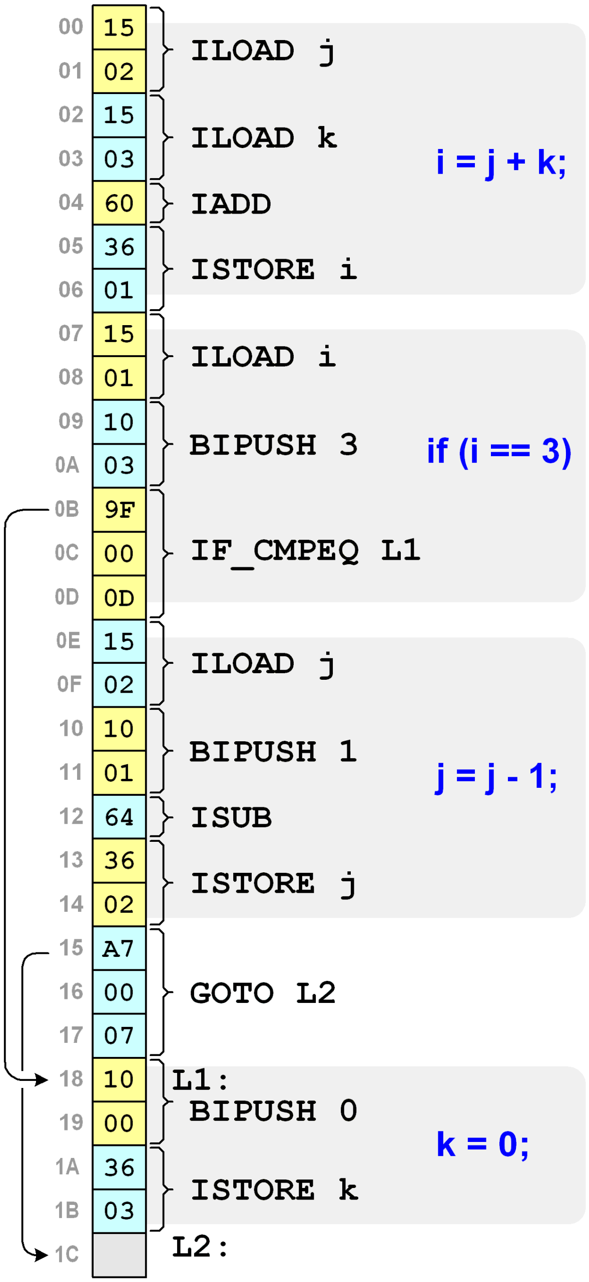
Compiling Java to IJVM

The diagram at left shows how the Java source code:

```
i = j + k;
if  (i == 3)
    k = 0;
else
    j = j – 1;
```

…would be converted into IJVM instructions.   Note the following:

- Branch instructions are used to ensure that the "k=0" instructions are executed when the "i == 3" condition is true, and the "j = j – 1" instructions are executed otherwise.
- The IJVM instructions aren't in the same order as the source code, but they achieve the same result.
- The operands specified in the branch instructions are the distance between the address of the branch instruction opcode and the address of the target instruction opcode. For example, the "GOTO L2" instruction has an operand of "0007" because there's 7 bytes between the GOTO L2 instruction and the instruction that it branches to.



00	15	ILOAD j	i = j + k;
01	02		
02	15	ILOAD k	
03	03		
04	60	IADD	
05	36	ISTORE i	
06	01		
07	15	ILOAD i	if (i == 3)
08	01		
09	10	BIPUSH 3	
0A	03		
0B	9F	IF_CMPEQ L1	
0C	00		
0D	0D		
0E	15	ILOAD j	j = j - 1;
0F	02		
10	10	BIPUSH 1	
11	01		
12	64	ISUB	
13	36	ISTORE j	
14	02		
15	A7	GOTO L2	
16	00		
17	07		
18	10	L1: BIPUSH 0	k = 0;
19	00		
1A	36	ISTORE k	
1B	03		
1C		L2:	

Exercise 7 – Compiling Java Source Code

Which of the IJVM bytecode sequences shown below corresponds to the following Java source statement?

k = (i + j) – 6;

…assuming that the variables are at the following locations in the local variable frame:

i	offset 5
j	offset 6
k	offset 7

A

```
ILOAD     5
ILOAD     6
IADD
BIPUSH    6
ISUB
ISTORE    7
```

B

```
ILOAD    5
BIPUSH   6
IADD
ILOAD    6
ISUB
ISTORE   7
```

C

```
ILOAD    5
ILOAD    6
IADD
ILOAD    6
ISUB
ISTORE   7
```

D

```
BIPUSH   5
BIPUSH   6
IADD
ILOAD    6
ISUB
ISTORE   7
```

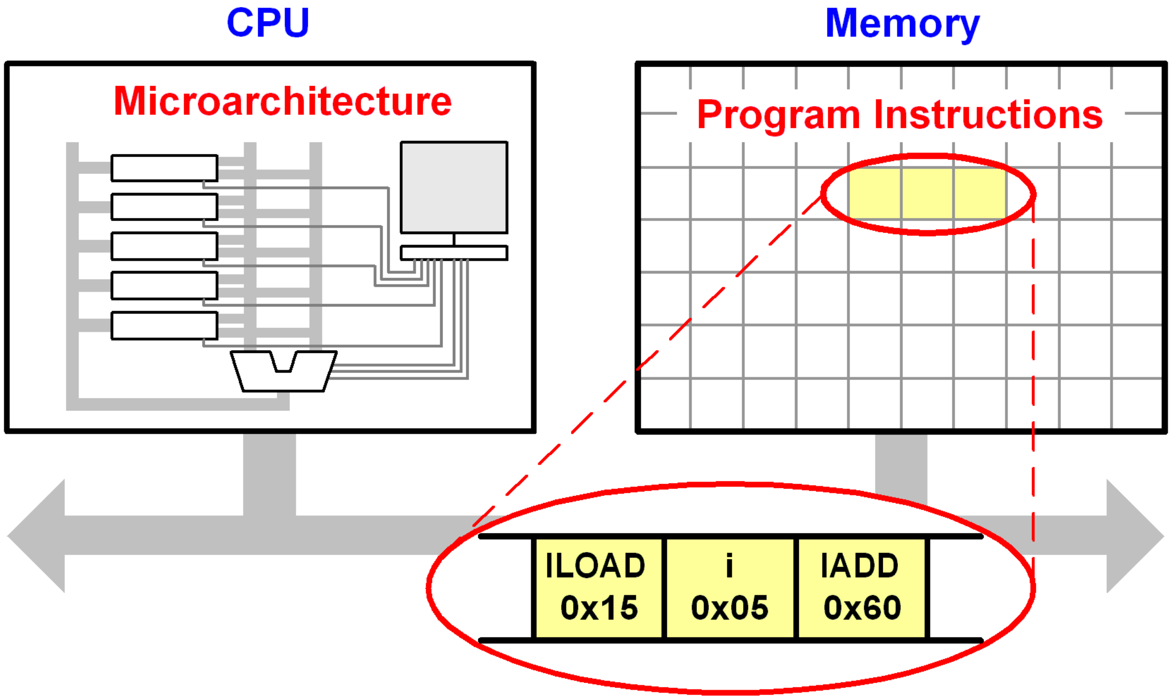
MIC-1 Microarchitecture

We'll now study a sample microarchitecture which can execute IJVM instructions. We'll call this first version of the microarchitecture "MIC-1" to distinguish it from additional versions we'll develop to show how designers improve the performance of computer systems.

But what is a microarchitecture?

The microarchitecture is the organization of the circuitry within the CPU chip. It consists of the registers, internal buses, and control logic that enables the CPU to do it's job. And remember, the CPU's job is to:

Fetch IJVM instructions from memory and execute them.

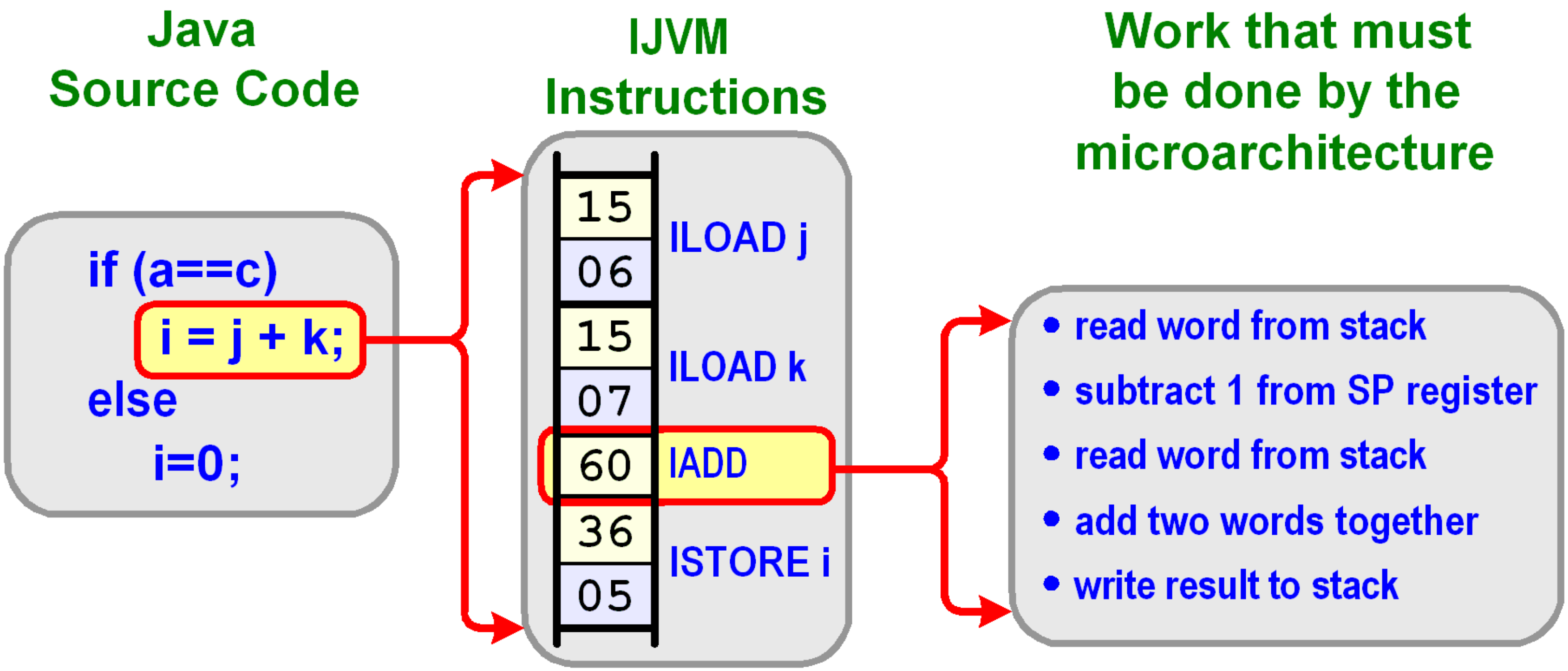


So the microarchitecture must be able to:

- **Fetch the next IJVM instruction from memory by using the address in the Program Counter (PC) register**
- **Decode ("recognize") the instruction and execute it (do the work of the instruction)**
- **Increment the PC so it points to the following instruction**
- **Repeat**

Microarchitecture Level vs. IJVM Instructions

As we study the microarchitecture level, remember that it's purpose is to do the work required by the instructions that are specified at the ISA level (which in this case is IJVM).



To do this, we have to understand the basic operations that each IJVM instruction does and come up with a circuit which can do those operations in sequence.

The IJVM “IADD” Instruction

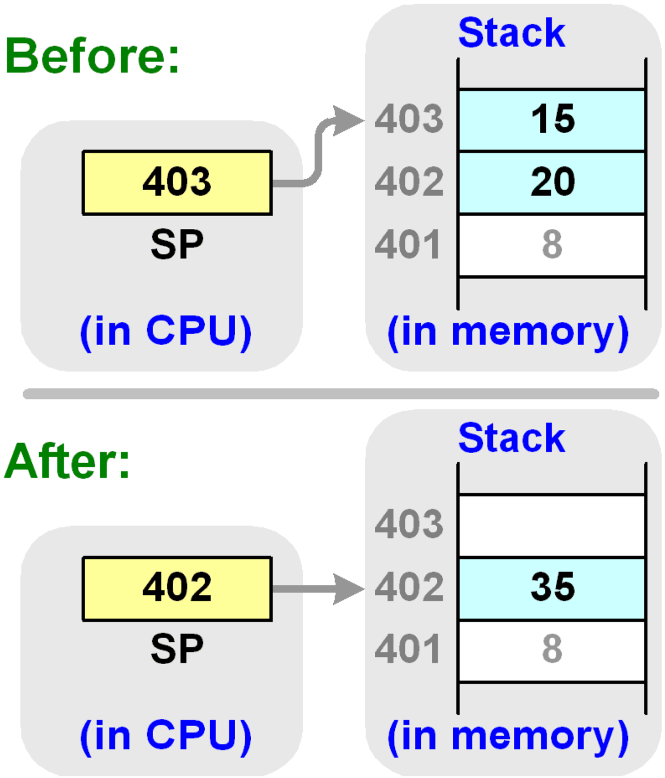
Consider the IJVM “IADD” instruction. That instruction is supposed to do the following:

Pop the top two words off the stack,

Add them together, and

Push the result back onto the stack

This is the “specification” for the instruction – it tells us what needs to be done, but it doesn’t say how to make it happen.



Before:

Stack

403 15
402 20
401 8

403
SP
(in CPU)

(in memory)

After:

Stack

403
402 35
401 8

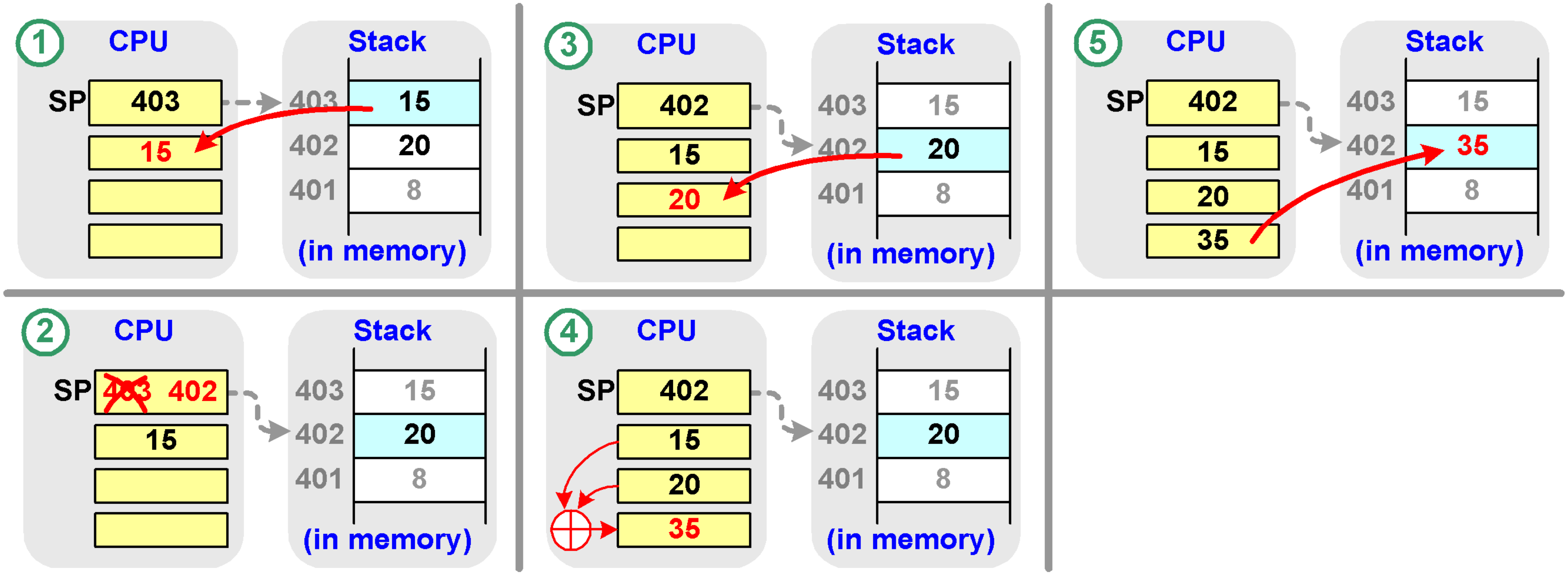
402
SP
(in CPU)

(in memory)

IJVM “IADD” Instruction

To make the IADD instruction actually work our microarchitecture will have to do these steps:

1. Read data from the memory address held in the Stack Pointer register
2. Subtract 1 from the stack pointer
3. Read data from the memory address held in the Stack Pointer register
4. Add the two data values together
5. Write the result back to the memory address held in the Stack Pointer register



This is how our microarchitecture will implement the IADD instruction.

IJVM "IADD" Instruction

In addition to making the IADD (and other) instructions work, our microarchitecture must also handle the task of fetching the IJVM instructions from the method area and decoding them (recognizing which instructions they are) so that we choose the right steps to perform for each one.

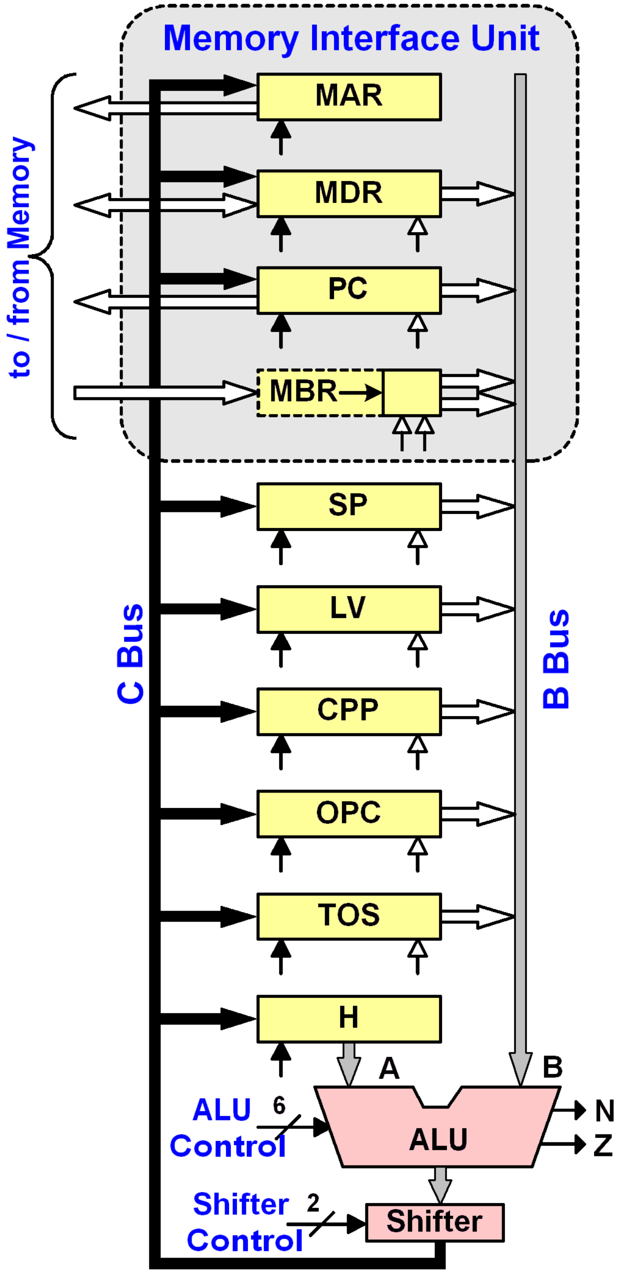
To do all these things, the microarchitecture will need to include:

- **A set of registers such as "SP", "PC", "LV", "CPP", etc.**
- **An arithmetic logic unit (ALU) to perform arithmetic and other operations**
- **A memory access unit to allow data to be moved between memory and the CPU**
- **A set of buses to move data between the registers, ALU and memory**
- **A control unit to govern the operation of everything**

MIC-1 Data Path

The diagram at the right shows the data path of the MIC-1 microarchitecture. It includes:

- **A set of 32-bit registers to hold the various information we need to use (described on the next page).**
- **A memory interface unit which includes several registers used to move data to and from memory.**
- **An ALU identical to the one we learned about earlier, except that this ALU has two additional outputs:**
  - **N – set to TRUE when the ALU result is negative (ie, when the high-order bit is "1")**
  - **Z – set to TRUE when the ALU result is zero**
- **A shifter which can shift the output of the ALU one bit to the right or eight bits to the left**
- **A 32-bit wide B Bus to carry data from one of the registers to the B input of the ALU (the A input always comes from the "H" register)**
- **A 32-bit wide C Bus to carry the result from the ALU/shifter back to one or more of the registers.**



MIC-1 Registers

The data path contains the following MIC-1 registers:

MAR, MDR, PC, MBR These registers are used by the memory control unit to transfer data between the CPU and memory. They'll be explained in detail shortly.

SP The **stack pointer**. It always contains the address of the data word that is the current top of stack.

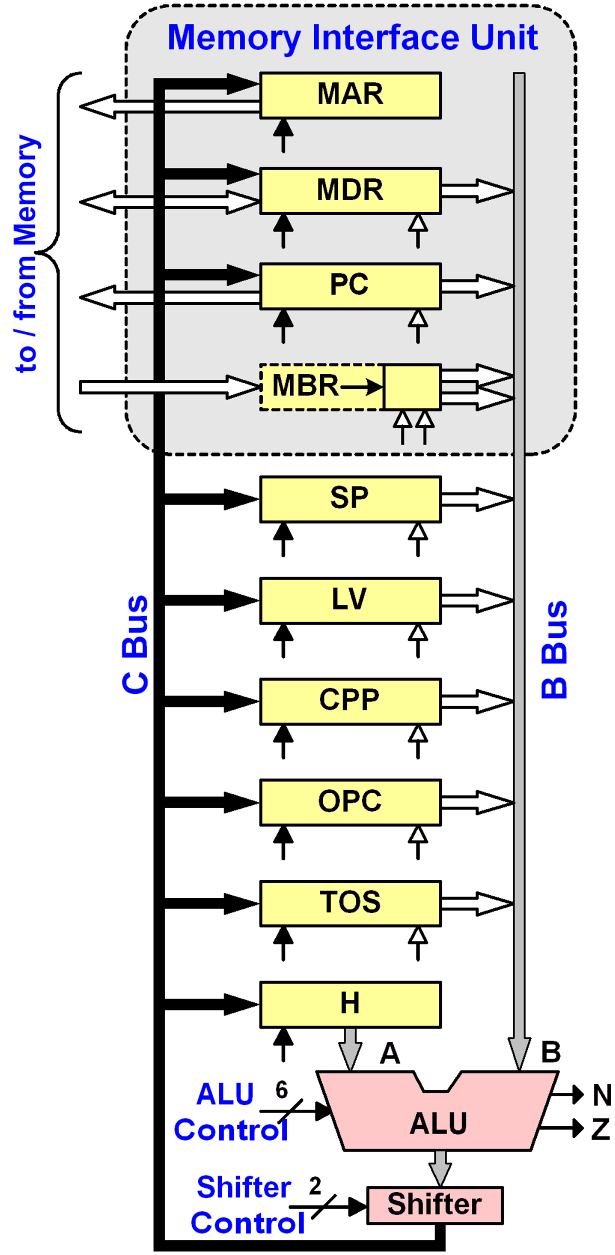
LV The **local variable pointer**. It always contains the address of the first local variable for the current subroutine.

CPP The **constant pool pointer**. It always contains the address of the first constant in the constant pool.

OPC The "**Old PC**" register. This is used for certain instructions to keep a copy of the PC value before it is changed by the instruction.

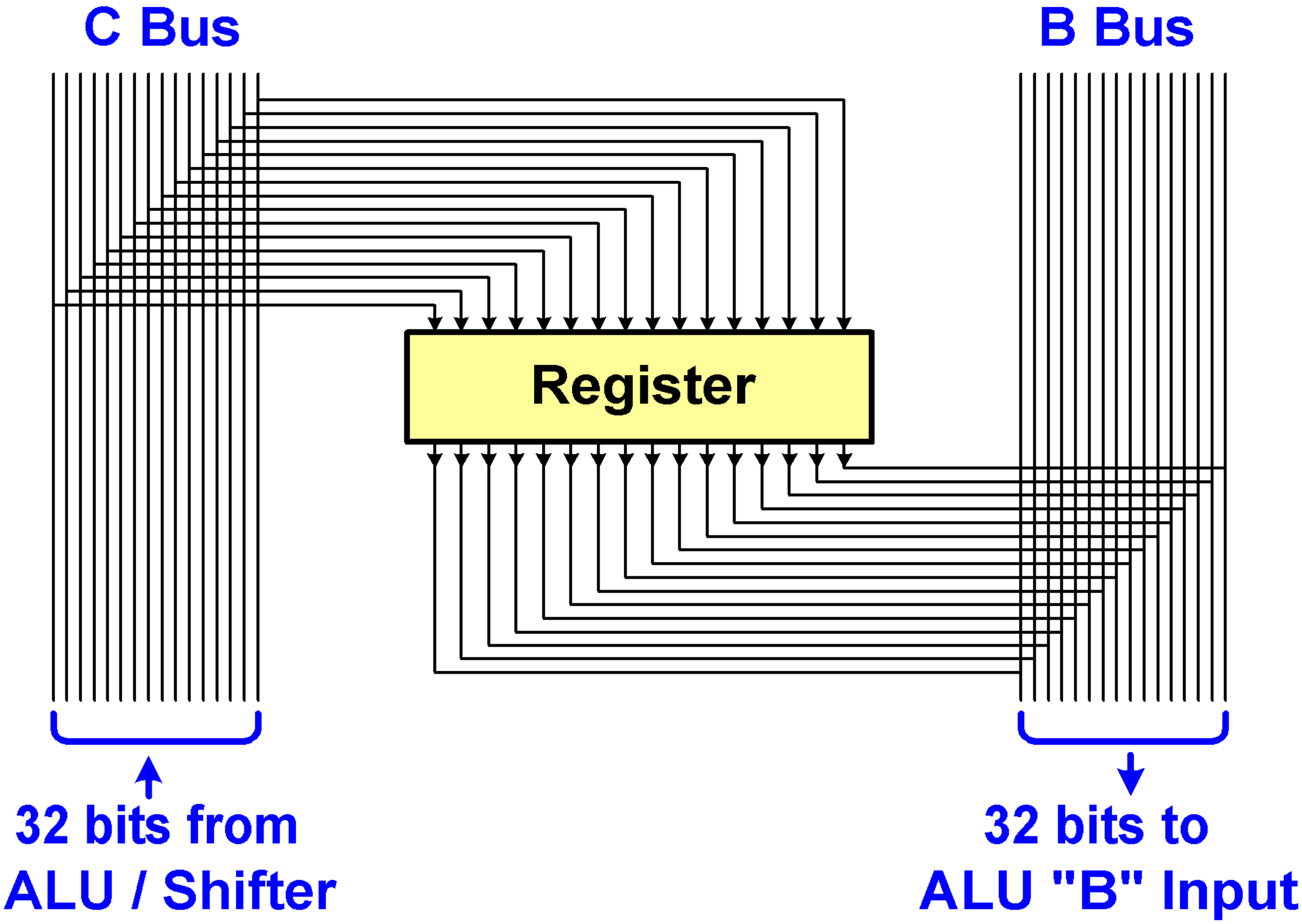
TOS The **Top Of Stack** register. It always contains a copy of the data word that is the current top of stack. Having a copy in this register saves us from having to read it from memory every time we need to use it.

H This register supplies the value for the A input of the ALU, and is sometimes also used to hold temporary values.



MIC-1 Registers

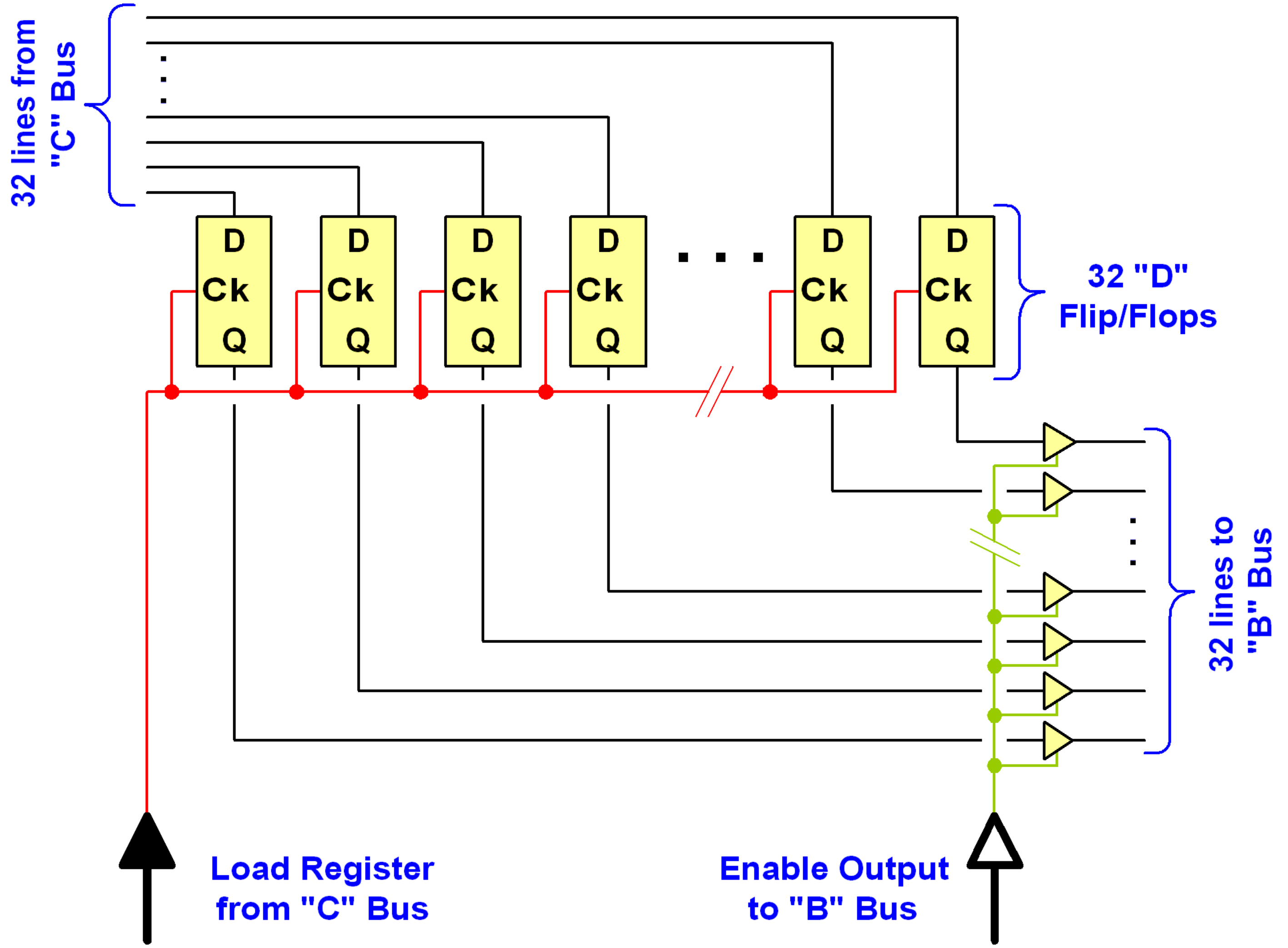
The registers are 32-bit storage circuits that have their inputs connected to the 32 wires on the C bus and their outputs connected to the 32 wires on the B bus:



MIC-1 Registers

Each of the registers is made from 32 flip-flops.

The outputs of the flip-flops go through a series of tri-state buffers. When a TRUE signal is delivered through the white arrow at the bottom right, the contents of the register is delivered to the “B” bus and travels to the “B” input of the ALU.



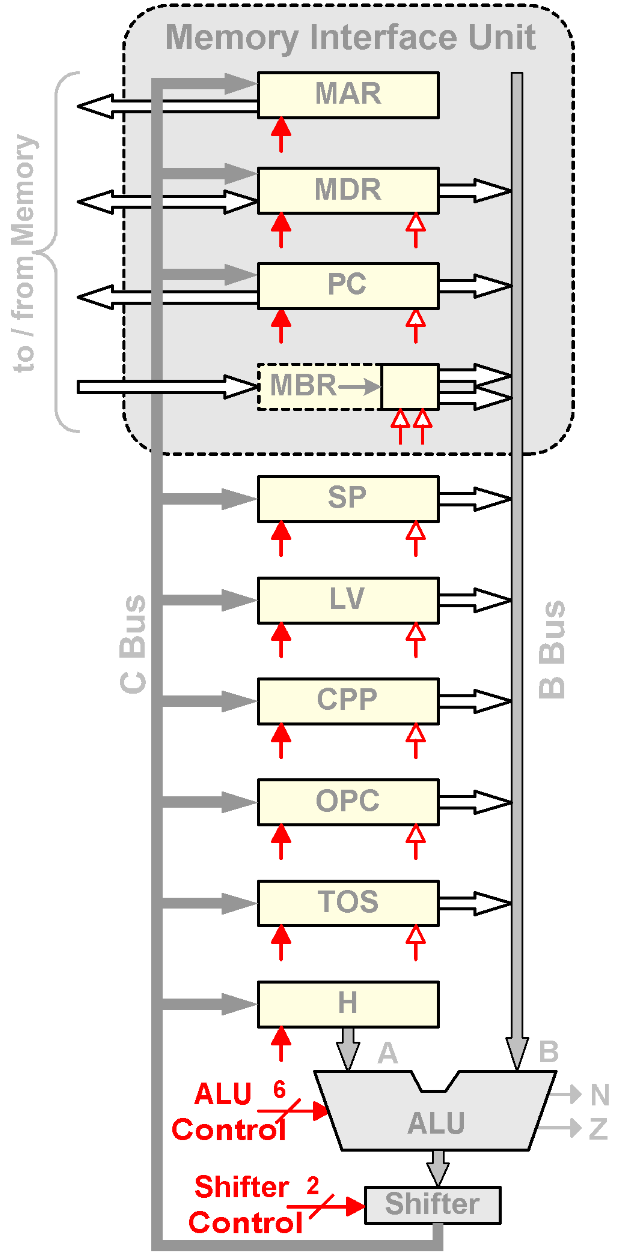
The inputs of the flip-flops are connected to the “C” bus. When a TRUE signal is delivered through the black arrow at the bottom left, the flip-flops load the current contents of the “C” bus (which is the result coming out of the ALU and shifter).

MIC-1 Data Path Control Signals

The operation of the data path is governed by a series of control signals which:

- Connect the output of one of the registers to the B bus so it can flow into the B input of the ALU. Note that only one register at a time can be connected to the B bus (if two registers are connected then they will "fight" over the value being put onto the B Bus).
- Selects the function that the ALU and Shifter are to perform. The ALU uses exactly the same control inputs as we described earlier in the course. As an example, the "B+1" function would produce an output of one more than the input value present on the B bus.
- Selects one or more registers to be loaded with the ALU result on the C bus. Note that you can load the C bus result into as many registers as you want.

There is additional logic (which we'll cover next week) that generates these control signals. For now, it's enough to understand that, by applying the right control signals, the data path can be made to move or manipulate the values in the registers by passing the information through the ALU.



MIC-1 ALU Control Signals

The ALU control signals are exactly the same signals as those used for the ALU we built in the first half of the course:

F0 / F1 – Control what function the ALU performs (A OR B, A AND B, A+B, NOT-B)

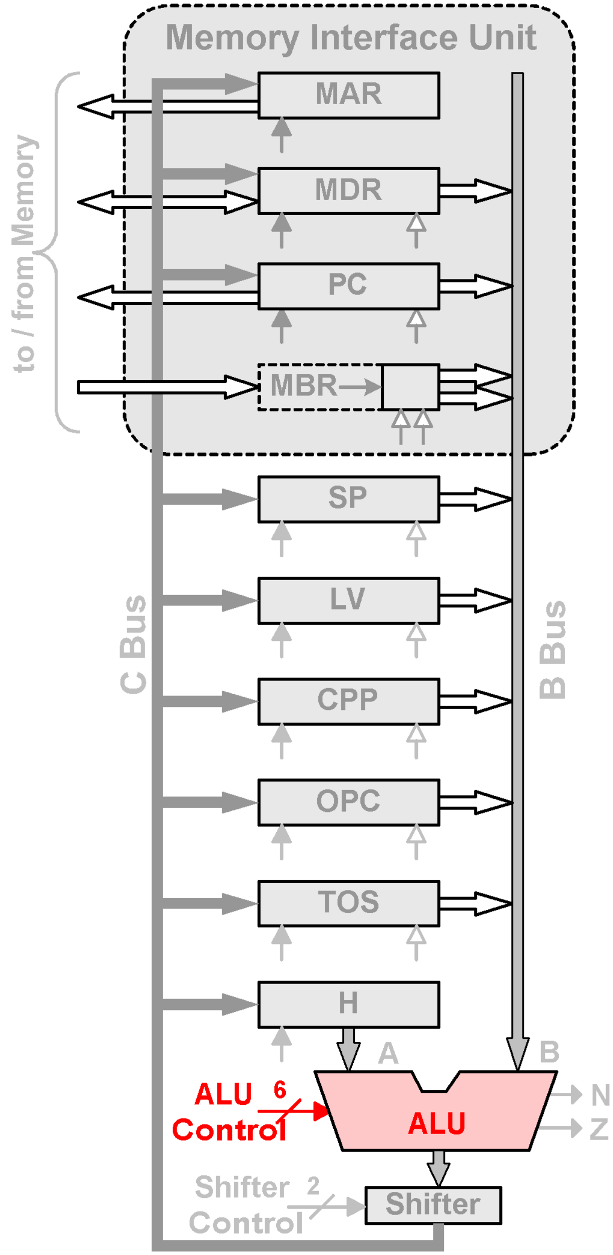
ENA / ENB – control whether the A or B inputs are allowed to enter the ALU

INVA – Controls whether the A input is inverted

INC – Controls whether "1" is added to the ALU result.

The table on page 4-2 shows all of the ALU functions that can be performed. Note that:

- The "A" input to the ALU is always the H register.
- The "B" input to the ALU is whatever register has been connected to the B Bus



IJVM-1 ALU Control Signals

These are the functions that the ALU can perform:

A
B
NOT-A
NOT-B

A+B
A+B+1
A+1
B+1

B – A
B – 1
–A
–1

A AND B
A OR B
0
1

Remember that:

- **The “A” input always comes from the “H” register.**
- **The “B” input comes from whatever register is connected to the B Bus (only one register can be connected to the B bus at any one time).**

This means that the ALU can perform calculations like:

SP – H   LV + 1   PC – 1

But not:

H – SP   H – 1   PC + SP

MIC-1 Shifter Control Signals

The output of the ALU goes through a shifter which also has two control signals:

SLL8 – Shift Logical Left by 8 bits. When this signal is asserted the output of the ALU is shifted to the left by 8 bits. For example, if the 32-bit ALU output was:

0000 0000 0000 0000 0001 0010 0011 0100 (hex 00001234)

then the result after shifting left by 8 bits would be:

0000 0000 0001 0010 0011 0100 0000 0000 (hex 00123400)

We'll see what this is used for when we examine the microcode for the IJVM-1 machine.

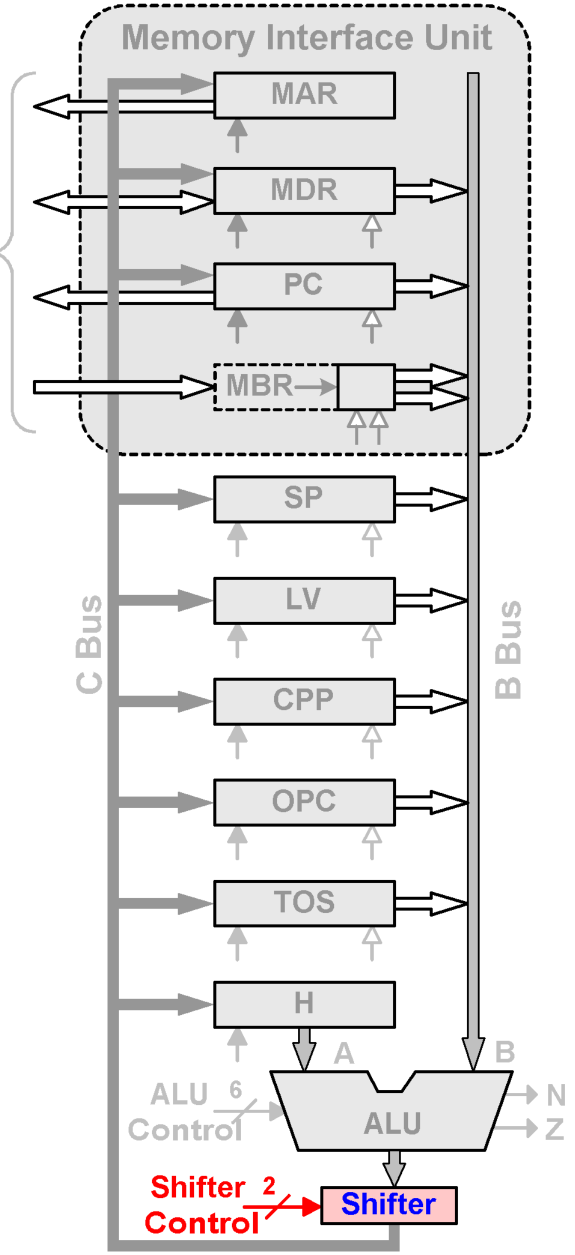
SRA1 – Shift Right Arithmetic by 1 bit. When this signal is asserted, the output of the ALU is shifted to the right by 1 bit. For example, if the 32-bit ALU output was:

0000 0000 0000 0000 0001 0010 0011 0100 (hex 00001234)

then the result after shifting right by 1 bit would be:

0000 0000 0000 0000 0000 1001 0001 1010 (hex 0000091A)

This capability isn't actually used in the IJVM-1.



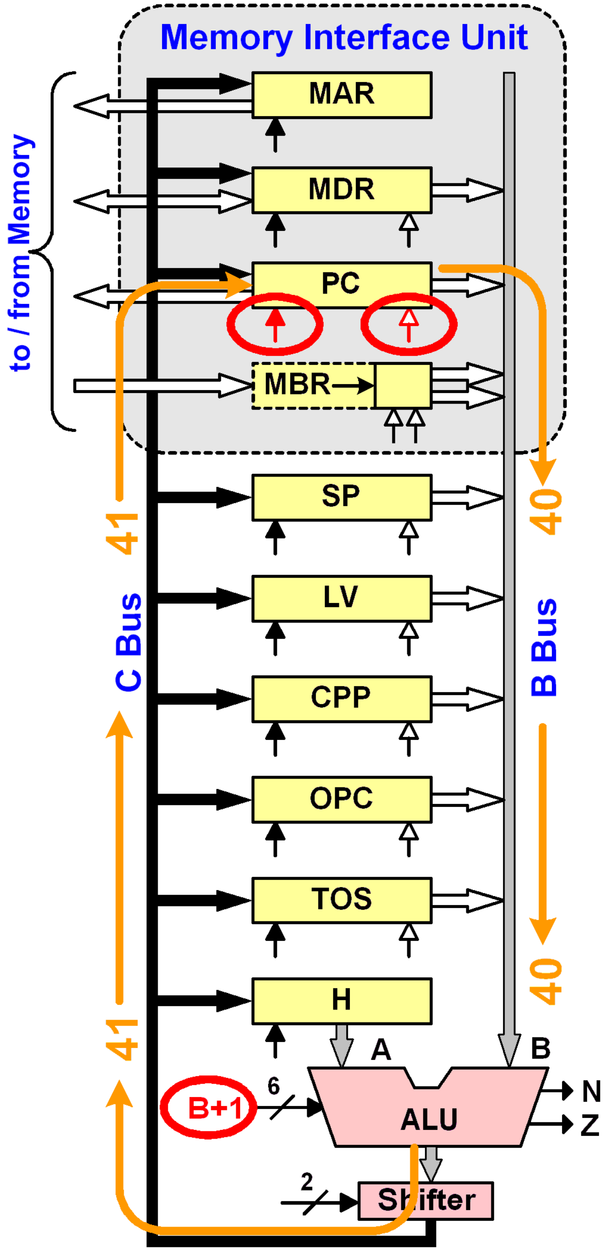
Sample MIC-1 Data Path Operation

Here's an example of how the control signals drive the data path. One of the things our microarchitecture needs to do is to increment the PC register to find the next instruction. Let's say that the PC register contains the number "40". The diagram at the right shows how the data path can be configured to add 1 to it to produce a result of "41":

- **The control signal which connects the output of the PC register to the B Bus is asserted.**
- **The ALU function code inputs are set to select the "B+1" function**
- **The control signal which loads the PC register from the C Bus is asserted.**

With this combination of control signals activated, here's what happens:

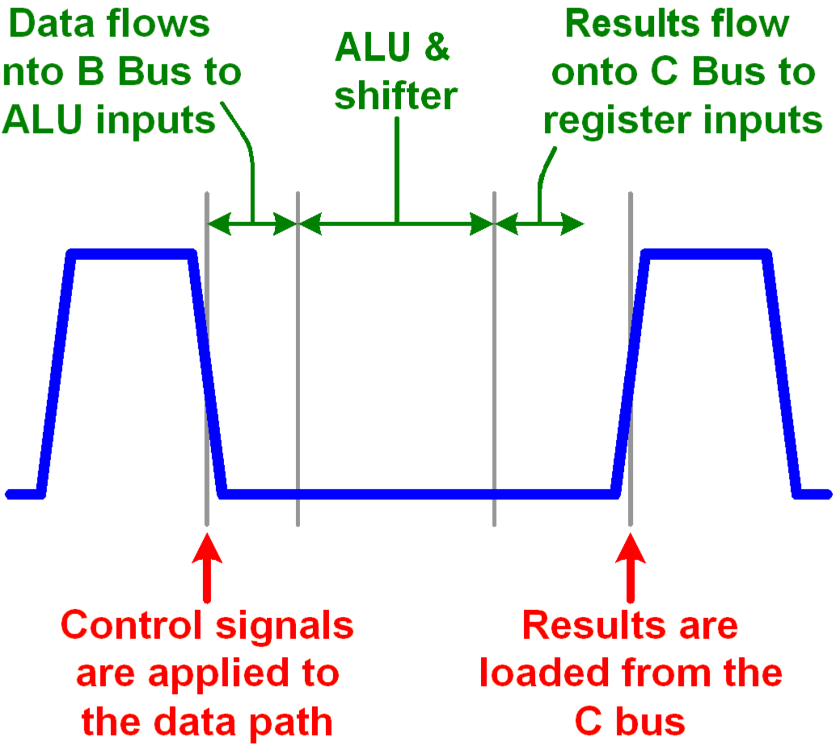
- **The PC contents ("40") go down the B bus into the ALU's "B" input.**
- **The ALU adds one to the B input and the result ("41") flows back up the C Bus**
- **The result is loaded from the C Bus into the PC register**



Data Path Timing

A master "clock" signal is used to synchronize the operation of the data path. An asymmetric clock (whose "high" time is shorter than it's "low" time) is used:

- The control logic (to be discussed later) selects which control signals to use during the "high" portion of the clock cycle.
- The control signals are applied to the registers and the ALU at the falling edge of the clock signal.
- Data flows from the selected register onto the B bus, through the ALU, and back up the C bus during the "low" portion of the clock cycle.
- The results on the C bus are loaded into the selected register(s) on the rising edge of the clock.


Data flows onto B Bus to ALU inputs
ALU & shifter
Results flow onto C Bus to register inputs
Control signals are applied to the data path
Results are loaded from the C bus

This explains why, in the previous example of adding 1 to the PC register, the register doesn't just keep incrementing over and over again. The clock ensures that data flows through the data path exactly once with each cycle.

The clock speed is chosen to be as fast as possible and still leave a safety margin to ensure that the results reach the C bus before the rising edge of the clock.

Exercise 8 –Data Path Control Signals

What do the control signals shown in the diagram cause the data path to do?

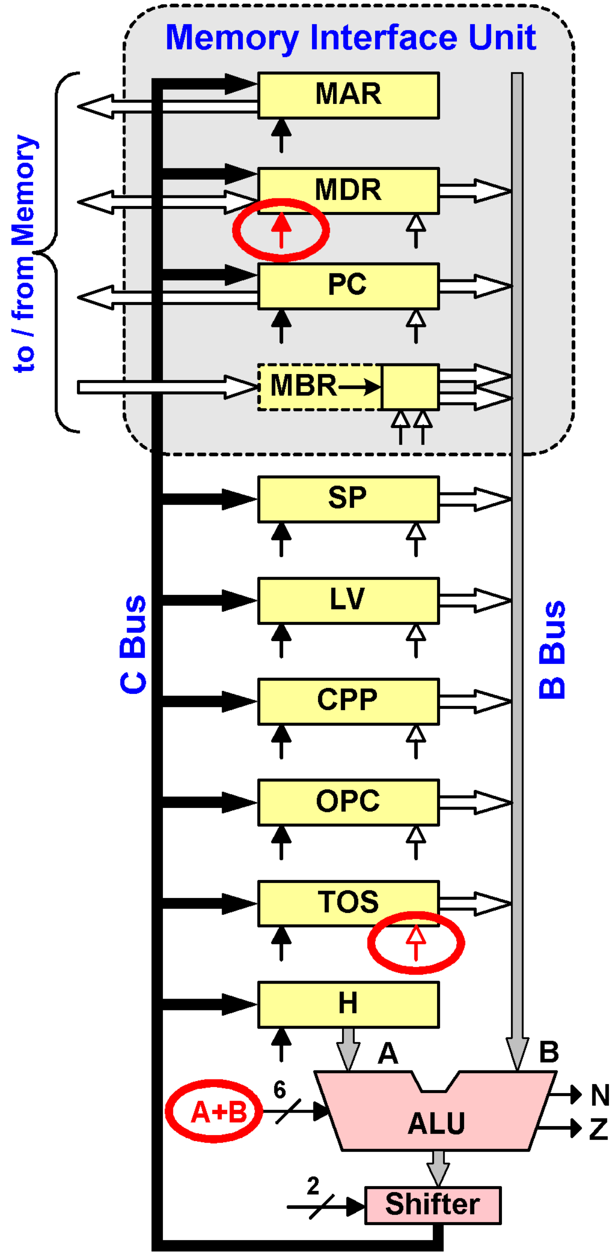
A Add the values in the “A” and “B” registers together and store the result in the TOS register

B Add the values in the “TOS” and “H” registers together and store the result in the MDR register

C Add the values in the “TOS” and “MDR” registers together and store the result in the “H” register

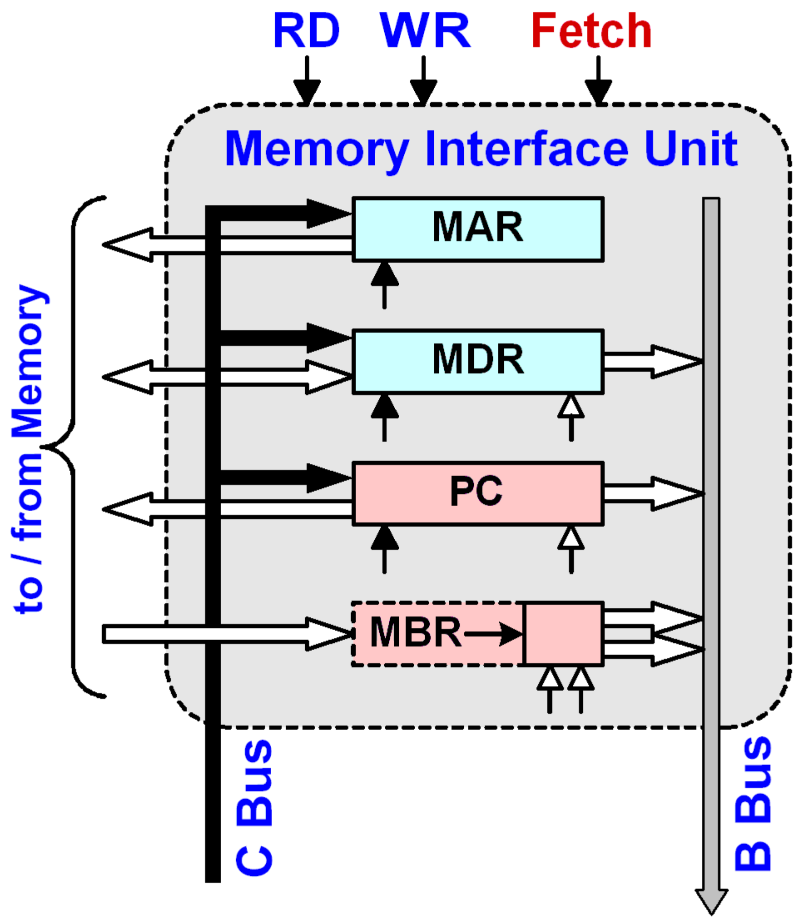
D Add one to the value in the “TOS” register and store the result in the “H” register

E Add one to the value in the “TOS” register and store the result in the “MDR” register



The Memory Control Unit

The memory control unit allows data to moved between the memory and the CPU. We're not going to into the circuit details, but we do need to understand how it works.



The memory control unit accepts three control signals:

RD

The RD signal causes the memory control unit to read a word from the data area of memory.

WR

The WR signal causes the memory control unit to write a word to the data area of memory.

Fetch

The Fetch signal causes the memory control unit to read a byte from the method area of memory.

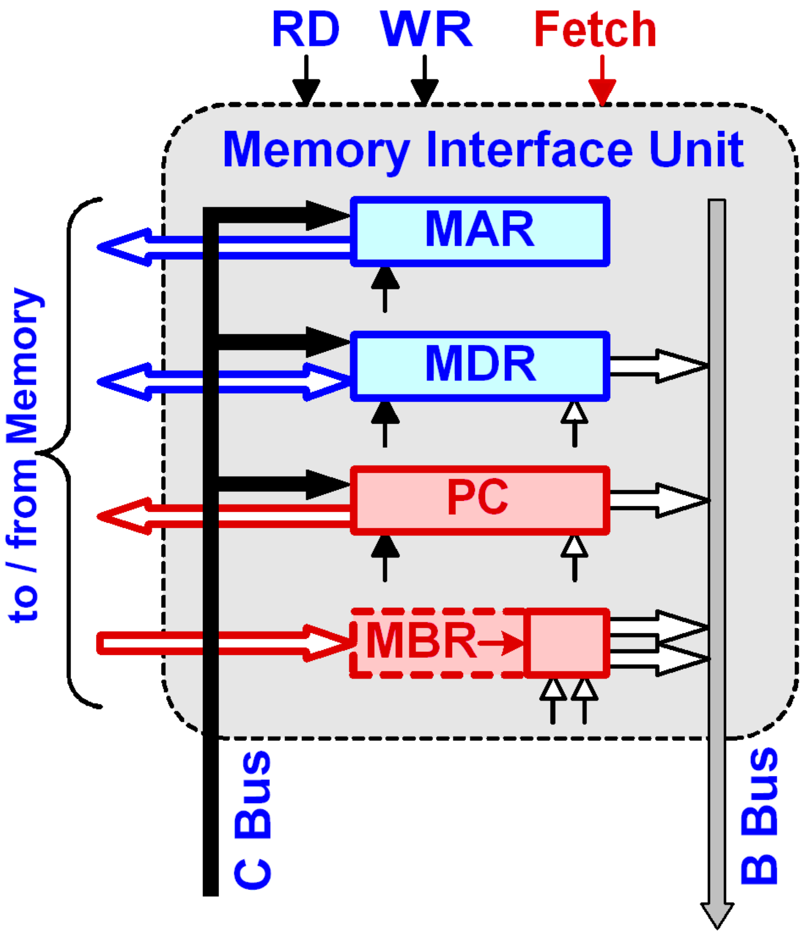
Notice that the system can read and write information in the data area of memory, but it can only read ("fetch") data from the method area. This is because the method area contains program instructions which are not allowed to be modified.

Memory Control Unit Registers

The MAR and MDR registers are used to read or write data to or from the data area of memory when the RD or WR signals are asserted:

MAR – Memory Address Register
This register contains the address of the word that is being read or written.

MDR – Memory Data Register
This register contains the data being read or written.



The PC and MBR registers are used to read data from the method area of memory when the Fetch signal is asserted:

PC – Program Counter Register
This register contains the address of the byte being read from the method area.

MBR – Memory Buffer Register
This register contains the data byte read from the method area.

The method area contains the program being executed, so the PC and MBR registers are used to read instructions and operands from memory and then execute them.

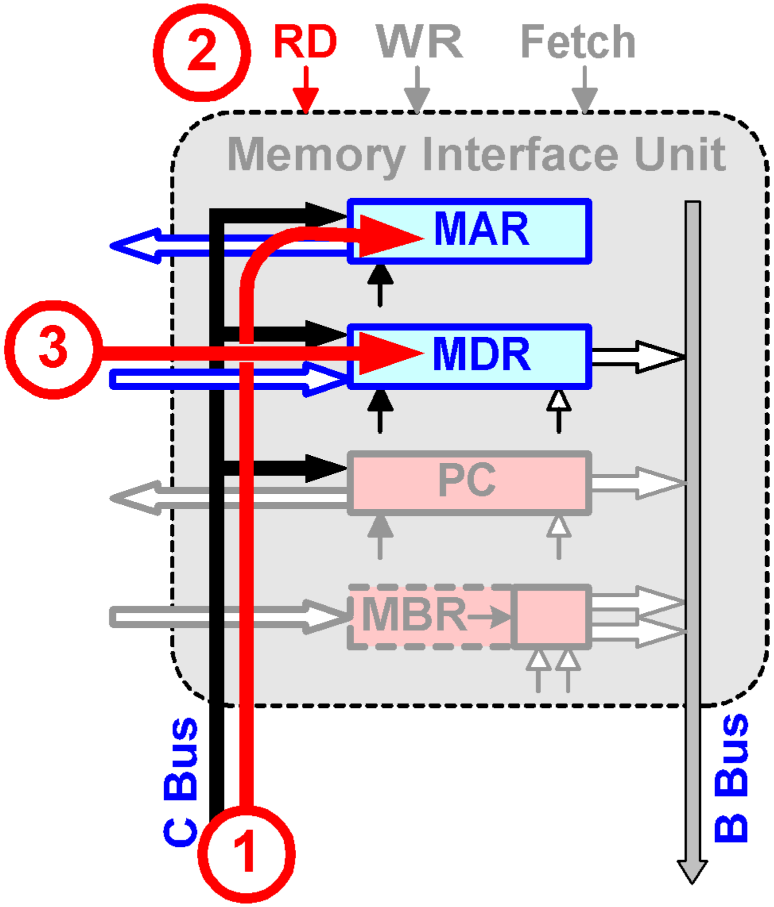
Reading from the Data Area of Memory

Here’s the sequence of events that must happen in order to read a word from the data area of memory:

1. **The address of the memory word to be read must be put into the MAR register.**
2. **The RD signal to the memory control unit must be asserted (this can be done in the same clock cycle that loads the MAR register address).**

When the RD signal is triggered, the memory interface unit sends the address to the system bus along with the control signals that request a memory read.

3. **The data returned from the requested memory address is placed into the MDR register.**



Writing to the Data Area of Memory

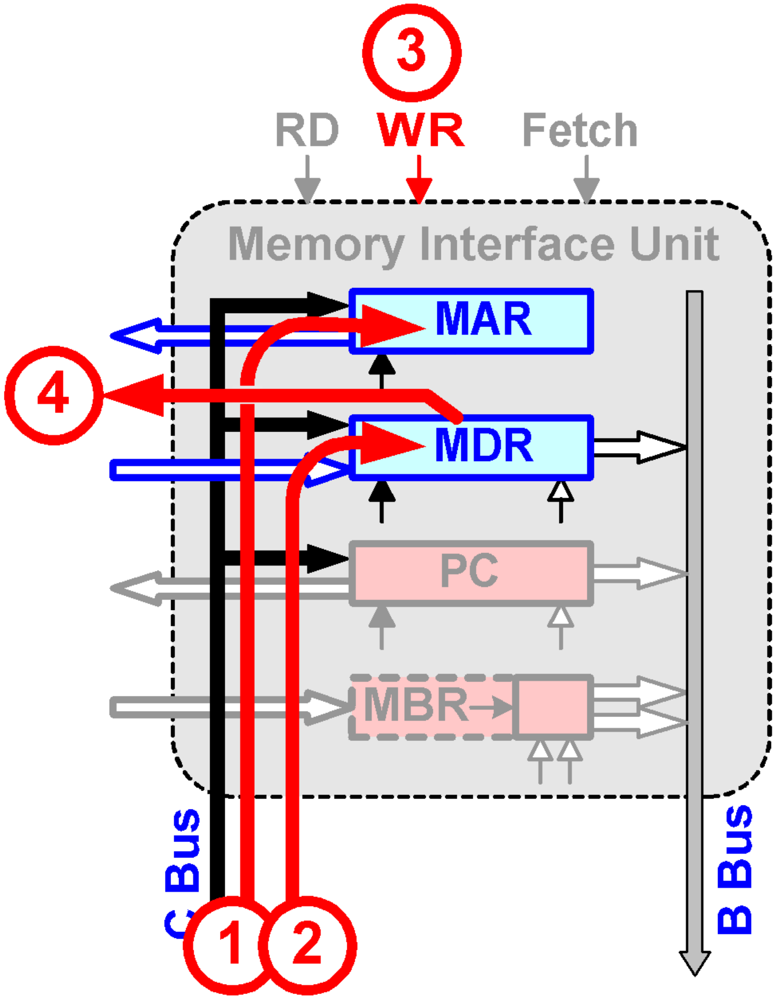
Here's the sequence of events that must happen in order to write a word of data to the data area of memory:

1. **The address of the memory word to be written must be put into the MAR register.**
2. **The data to be written to the word must be put into the MDR register.**
3. **The WR signal to the memory control unit must be asserted (this can be done in the same clock cycle that loads the data into the MDR register).**

When the WR signal is triggered the memory interface unit sends the address and data out over the system bus along with control signals that request a memory write operation, so that:

4. **The memory writes the requested data value to the requested address.**

(Note – it doesn't matter whether the address is put into the MAR first or the data is put into the MDR register first – as long as both registers have the correct values at the end of the clock cycle that the WR signal is asserted on).



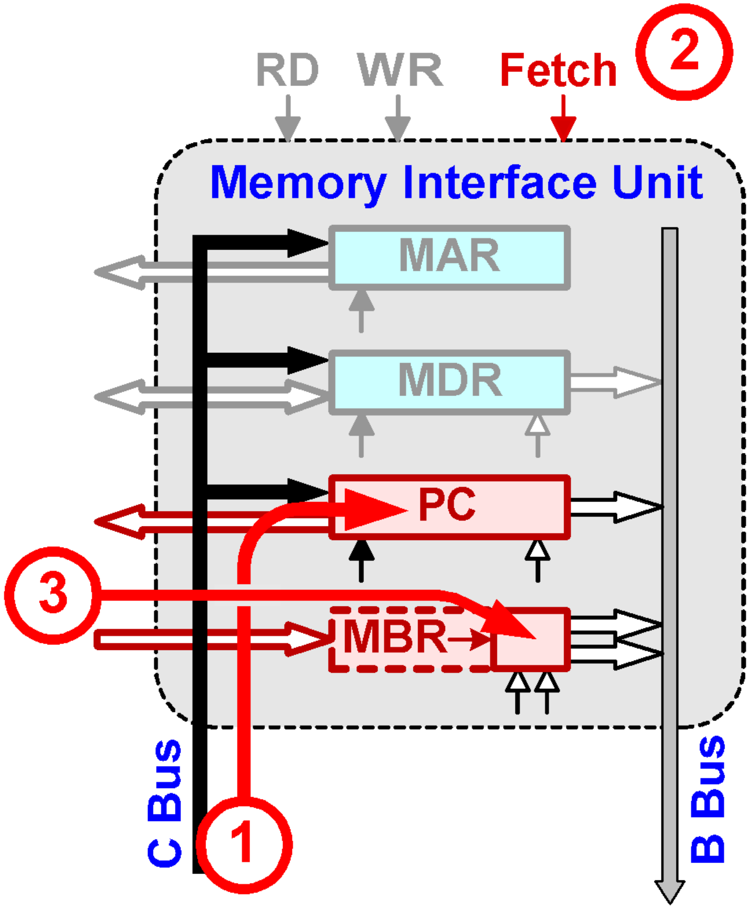
Fetching data from the Method Area of Memory

Here is the sequence of events that must happen in order to fetch a byte from the method area of memory:

1. **The address of the instruction byte to be read must be put into the PC register. (Since instruction bytes are usually read sequentially, one after another, this is usually done by adding "1" to the PC register)**
2. **The Fetch signal to the memory control unit must be asserted (this can be asserted on the same clock cycle that puts the new value into the PC register).**

When the Fetch signal is triggered, the memory interface unit will send the PC address to the system bus along with the control signals that request a memory read.

3. **The instruction byte returned from the requested memory address in the method area is placed into the MBR register.**



At least one byte must be fetched from the method area for every instruction that is executed. For instructions that have operands, the operands must be fetched from the method area as well.

Exercise 9 – Memory Access

1. What do the control signals shown in the diagram cause the data path to do?

   A Read the value at the top of the stack from memory

   B Write the value at the top of the stack to memory

   C Read the contents of the MAR register from memory

   D Write the contents of the MAR register to memory

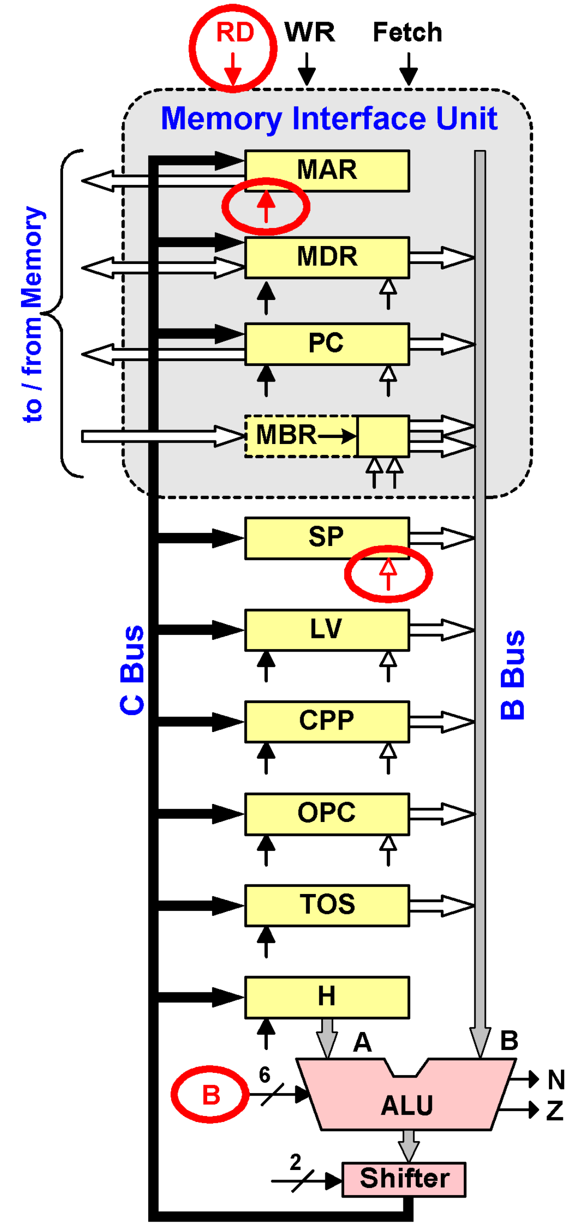
2. When the memory operation completes, what register holds the result?

   A The “MAR” register

   B The “MDR” register

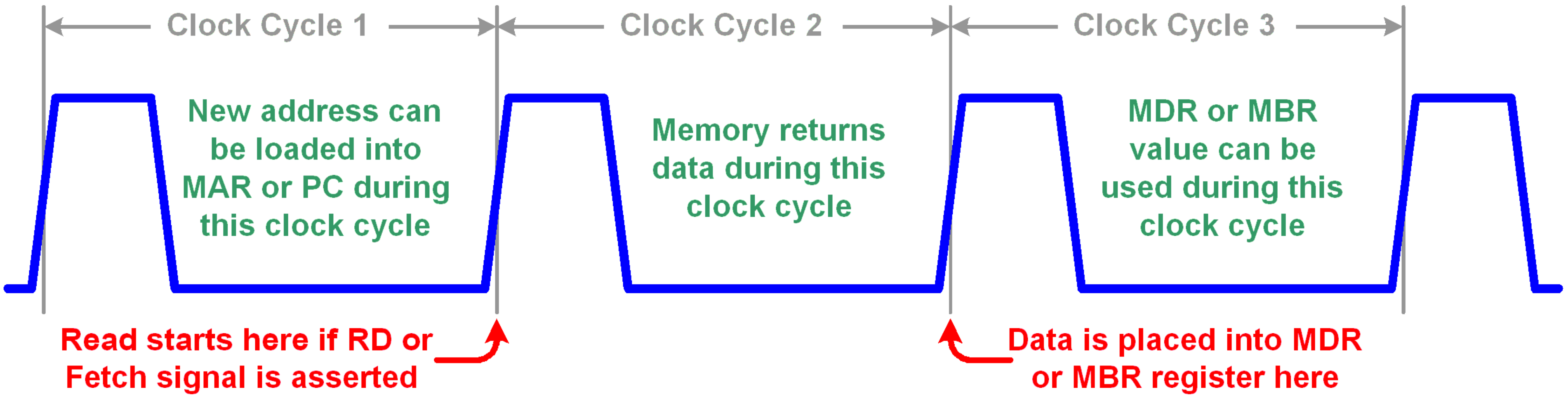
   C The “MBR” register

   D The “SP” register



Timing of Memory Reads

Because the memory is slower than the CPU, data that is read via "RD" or fetched via "Fetch" control signals is not available in the MDR or MBR registers immediately. The results are delayed by a clock cycle, so the values being read can't be used right away:


Clock Cycle 1
Clock Cycle 2
Clock Cycle 3
New address can be loaded into MAR or PC during this clock cycle
Memory returns data during this clock cycle
MDR or MBR value can be used during this clock cycle
Read starts here if RD or Fetch signal is asserted
Data is placed into MDR or MBR register here

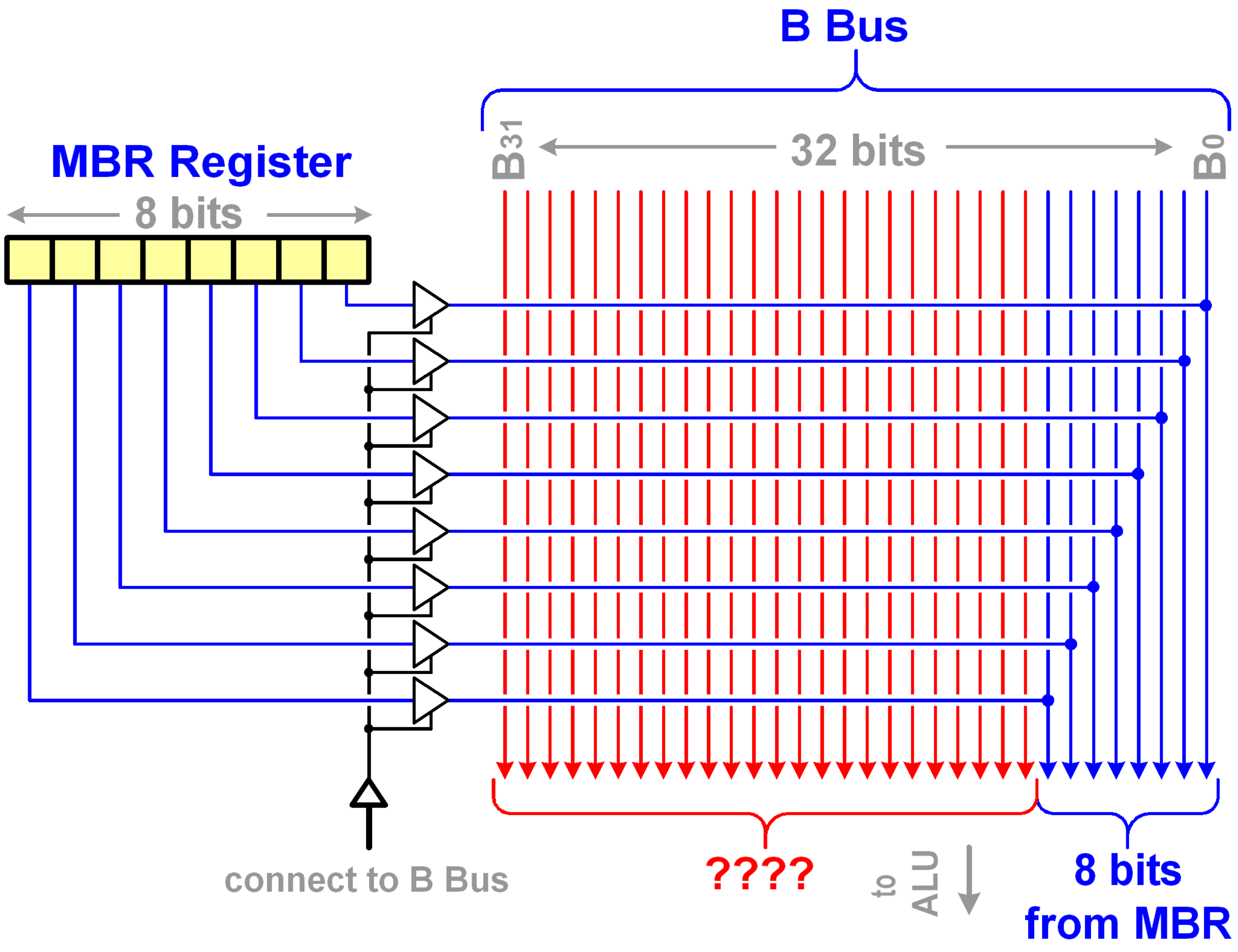
Because of the delay, data path operations that require the use of a value just read from memory have to wait for one clock cycle.

We're assuming that high-speed static memory is being used. If dynamic RAM were used then it would take even longer. If we used cache memory then we'd have to allow a variable length of time for data to be returned depending on whether it was found in the cache or not.

MBR and the B Bus

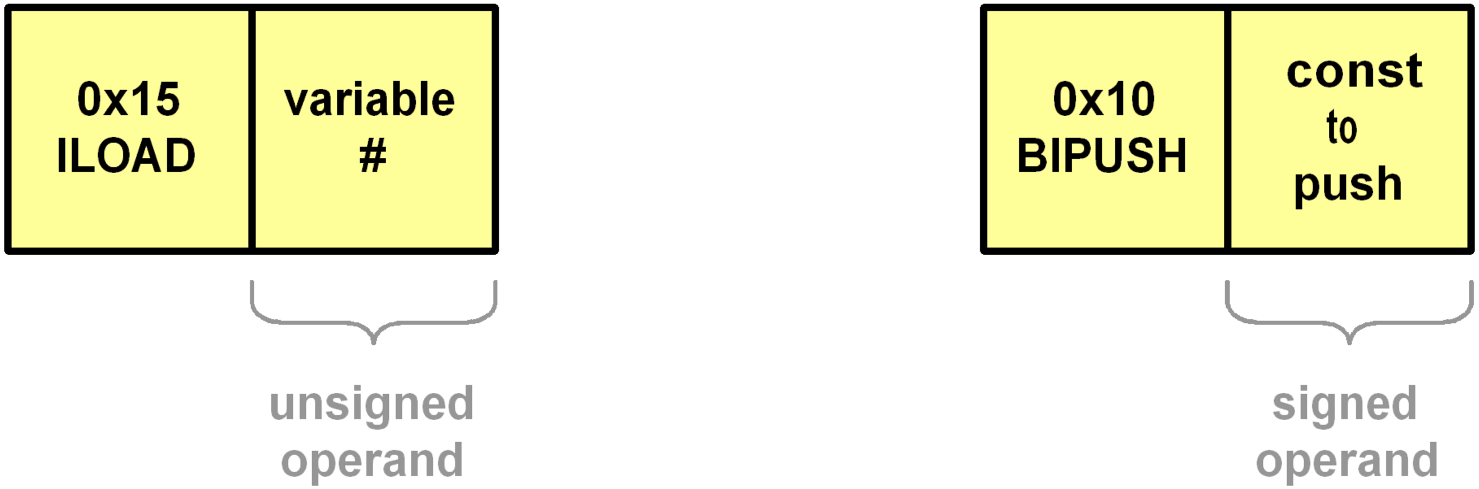
Unlike the other registers, the MBR register only holds 8 bits (because, unlike the data area of memory, the Method Area is byte-addressable and reads from it come in byte-sized chunks).

But if the MBR register only holds 8 bits, then what happens when it's output is connected to the 32-bit B Bus?



MBR and the B Bus

MBR is used to read bytes from the instruction stream, and some of those bytes contain signed numbers and other bytes contain unsigned numbers. For example consider the following instructions. Both have one byte for an opcode and a second byte to hold an operand:



0x15 ILOAD | variable #

unsigned operand

0x10 BIPUSH | const to push

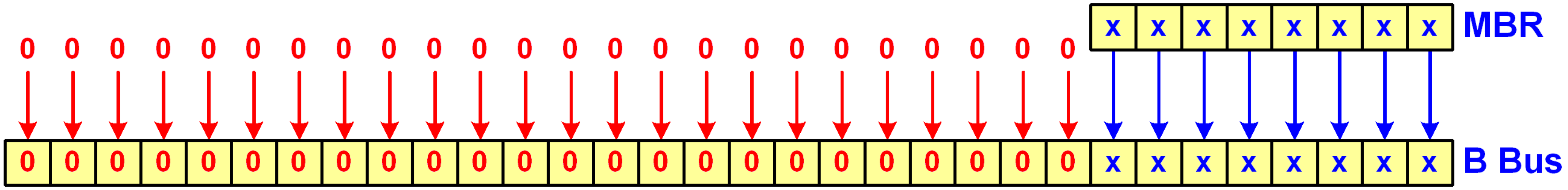
signed operand

The ILOAD instruction opcode is followed by an 8-bit unsigned variable number, while the BIPUSH opcode is followed by an 8-bit signed constant.

So when MBR is loaded onto the B bus and the 8-bit value is converted to a 32-bit value, there has to be a way to make it work for both signed and unsigned numbers. What needs to be done is different for the two types of numbers.

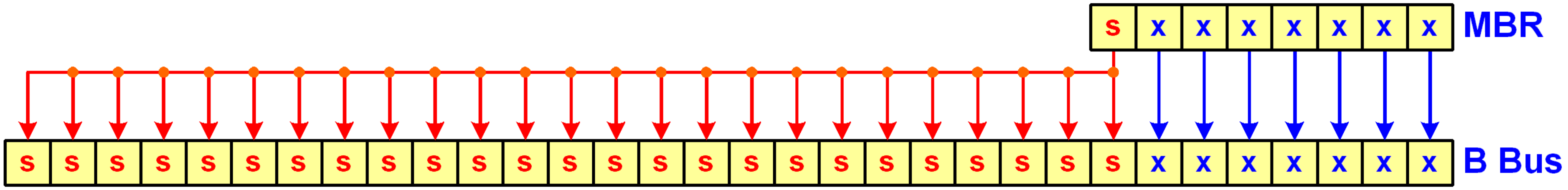
MBR and the B Bus

For unsigned numbers, the high-order 24 bits are filled with zeros:



This is equivalent to changing "123" to "00123" – it doesn't affect the value of a number.

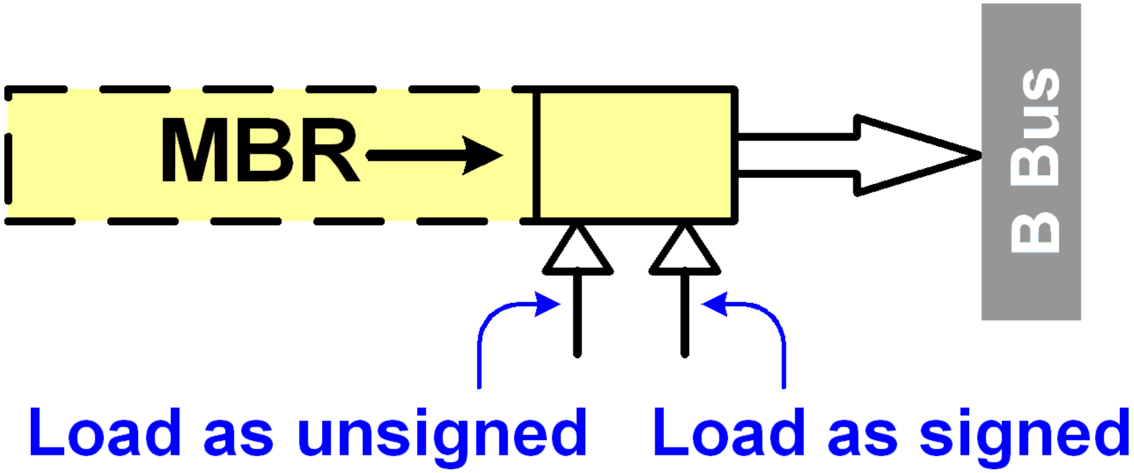
For signed numbers, the high-order 24 bits are replicated from the MBR register's sign bit (bit 7). Note that the sign bit may be "0" or "1". This operation is known as "sign extension".


s x x x x x x x
MBR
s s s s s s s s s s s s s s s s s s s s s s s s s x x x x x x x
B Bus

This preserves the sign and the value of a signed number.

MBR and the B Bus

This is why the MBR register has <u>two</u> control signals to connect it's output to the B Bus. One connects MBR as an unsigned number and the other connects it as a signed number:



One control signal or the other is activated depending on which type of value is being used (both control signals should not be activated at the same time).

The reason the MBR register is shown as a small solid box with a dashed-line extension is to indicate that it's actually an 8-bit register (solid box) whose value is converted to a full 32 bits (dashed-lines) as it's loaded onto the B bus.

Exercise 10 - MBR

The MBR register contains the hexadecimal value “FF”. The control signals shown at right are activated to copy the value into the MDR register.

1. If the MBR control signal is “Load as unsigned”, then which of these values does the MDR register receive?

   A – FF

   B – FFFF

   C – FFFFFF

   D – FFFFFFFF

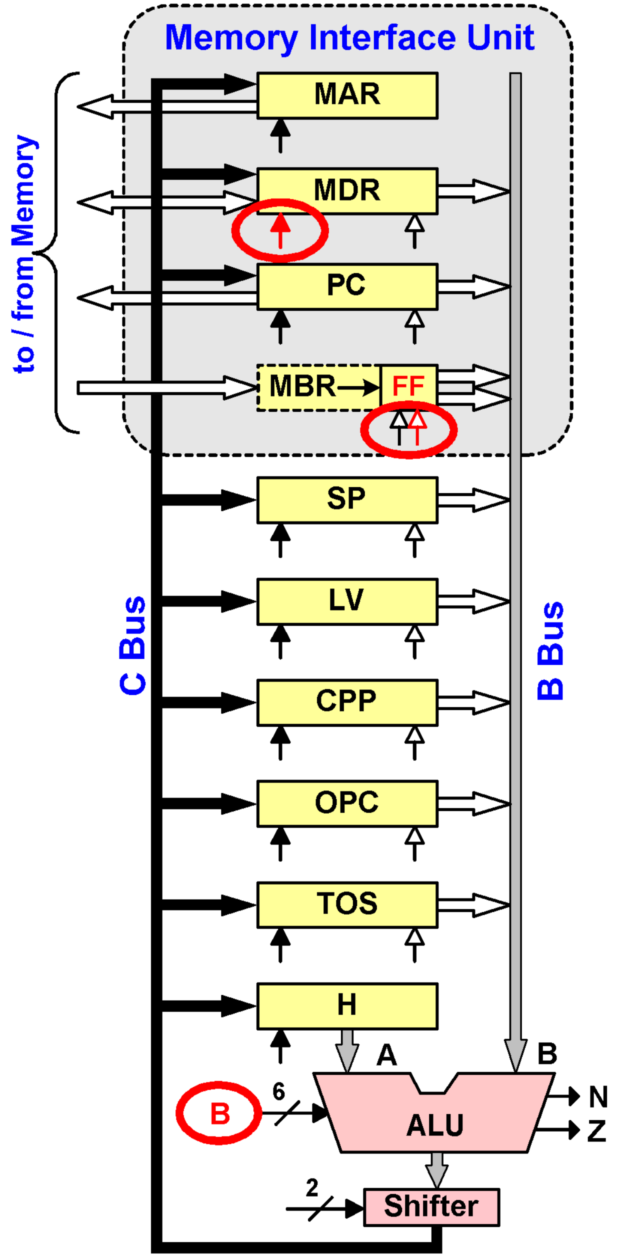
2. If the MBR control signal is “Load as signed”, then which of these values does the MDR register receive?

   A – FF

   B – FFFF

   C – FFFFFF

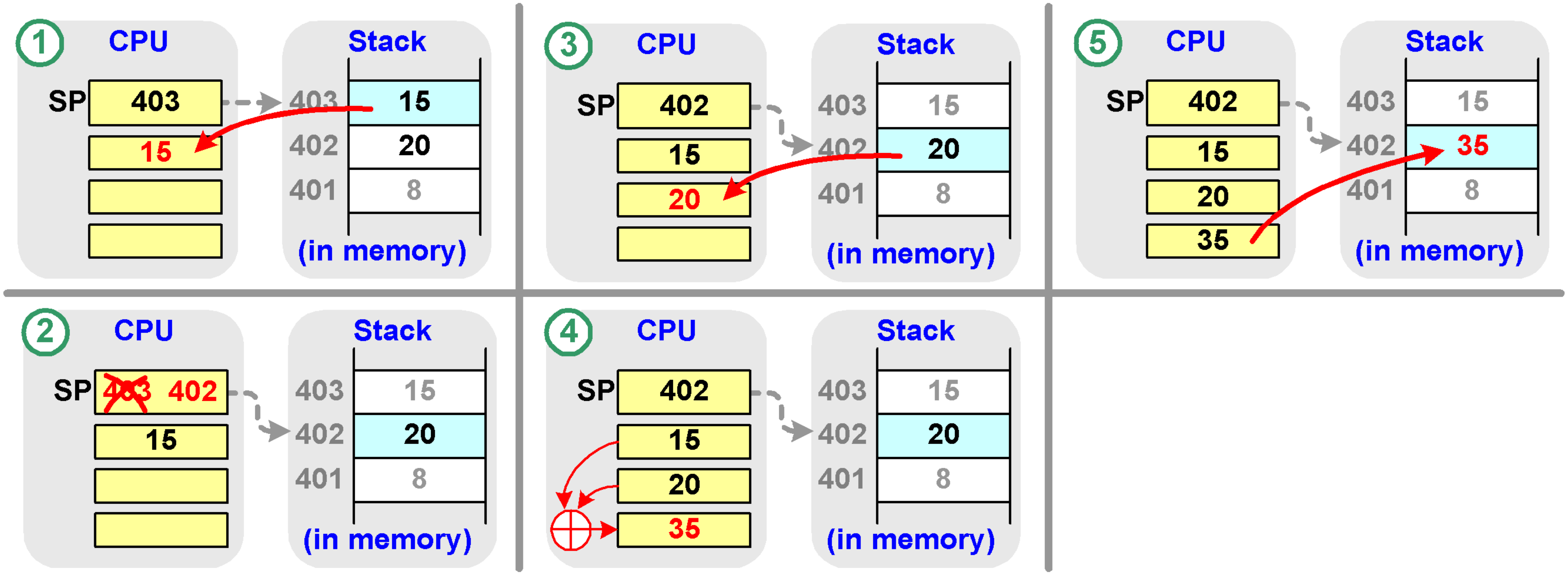
   D – FFFFFFFF



Executing the IJVM IADD Instruction

Now that we have a data path and know how it works, let’s put it to work by having it do the steps required for the IJVM “IADD” instruction. Recall that we need to do the following steps:

1. Read data from the memory address held in the Stack Pointer register
2. Subtract 1 from the stack pointer
3. Read data from the memory address held in the Stack Pointer register
4. Add the two data values together
5. Write the result back to the memory address held in the Stack Pointer register



The TOS Register

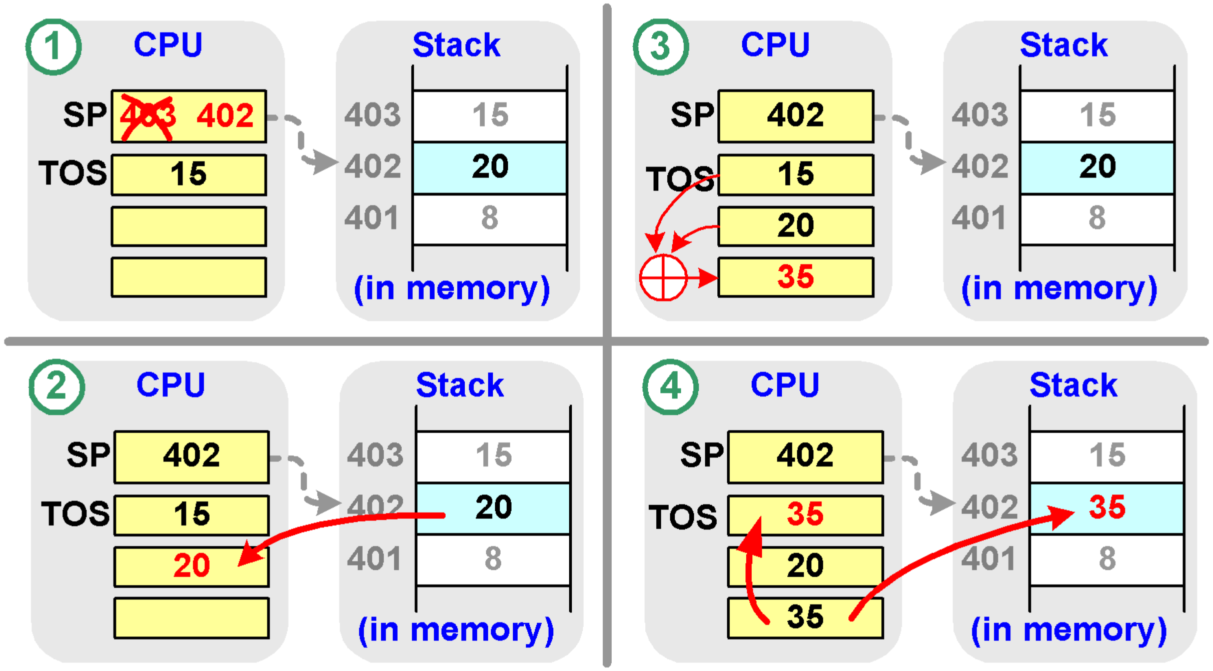
Because memory is slow, and because IJVM instructions often access data stored on the stack, our microarchitecture includes a "TOS" (Top Of Stack) register. We're going to keep a copy of the word that the stack pointer points to in this register so that we don't have to read the word from memory each time we use it. This means that our IADD instruction only has to:

1. Subtract 1 from the stack pointer
2. Read the memory word that SP points to
3. Add the value in the TOS register to the value read from memory
4. Write the result to the memory word that SP points and also put a copy in TOS as well

By using a TOS register we have avoided one memory read. Memory reads are slow, so this saves a lot of time.

The only thing we have to do is to remember to update the TOS register with the new top of stack value each time it changes.

We could do everything we need without a TOS register, but having one is worthwhile because of the performance improvement it gives us.



The IJVM IADD Instruction – Cycle 1

OK, now let’s actually do the work. Here’s the first two steps we have to do:

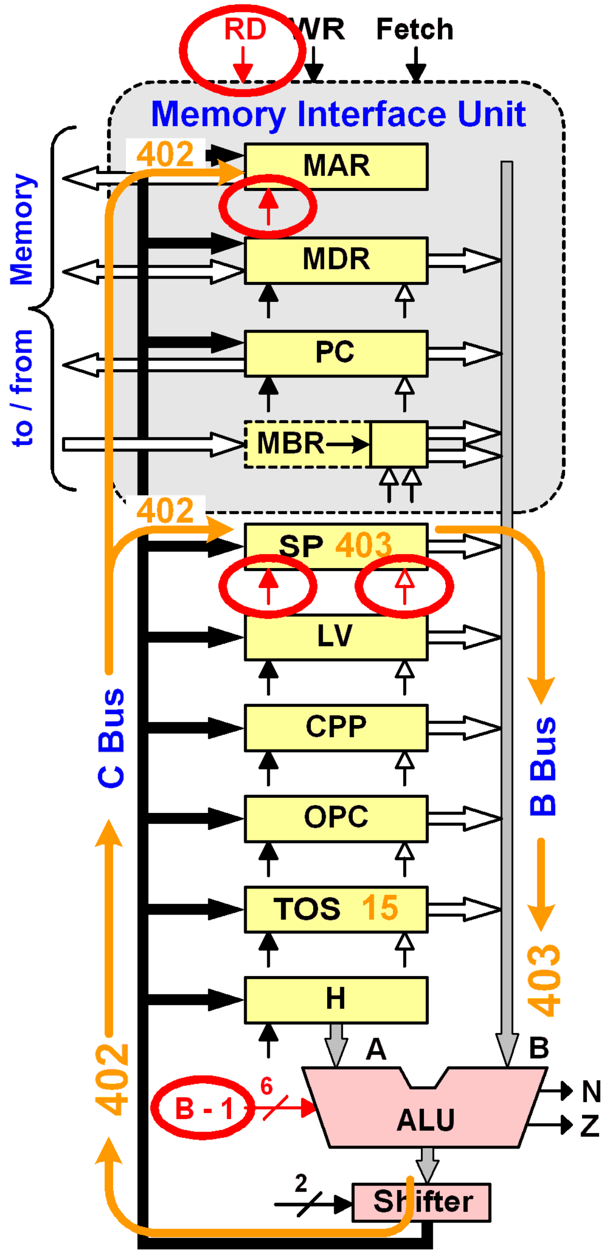
Subtract 1 from the stack pointer
Read the memory word that the stack pointer points to

To do this we assert the following control signals:

- Connect the output of the SP register to the B bus
- Select the ALU “B-1” function
- Load the SP register from the C Bus result
- Load the MAR register from the C bus result
- Activate the memory control unit “RD” signal

The result is:

- The old SP value (403) flows down the B bus into the ALU.
- The ALU subtracts one from it, and the result (402) flows up the C bus.
- The result on the C Bus is loaded into SP and MAR. The MAR register is now ready to read address 402.
- The RD signal causes the memory control unit to read the word at address 402. That word will be put into the MDR register at the <u>end</u> of the next cycle.



The IJVM IADD Instruction – Cycle 2

The next step we're going to have to do is to add the TOS value to the word that is read from memory. To prepare for this, we're going to move the TOS value into the H register. There are two reasons why we need to do this:

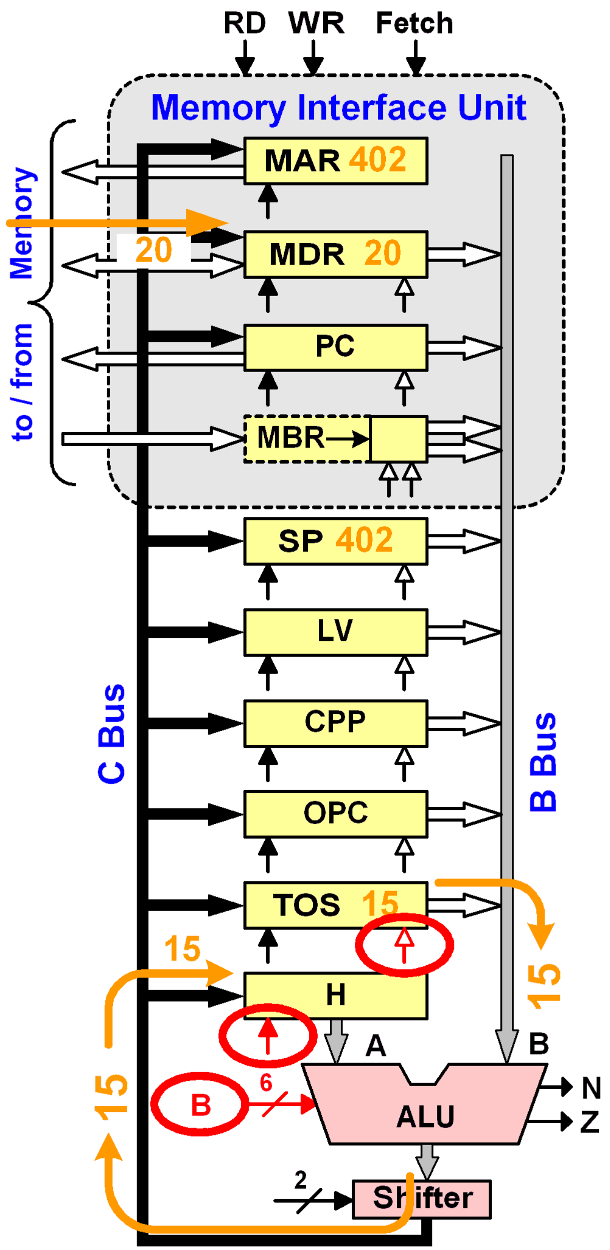
- To add two numbers together, you must put them into the ALU's A and B inputs. But the A input can only come from the H register – so one of the numbers has to be moved to H before it can be added.
- The word that is being read from memory doesn't arrive in the MDR register until the end of this clock cycle, so we can't add it until the next clock cycle.

To move TOS to H we activate the following control signals:

- Connect the output of the TOS register to the B Bus
- Select the ALU "B" function (so the B input goes through unchanged)
- Load the H register from the C Bus result

As a result, the H register ends up with a copy of the value in TOS.

Notice that the MDR register is also updated at the end of this clock cycle with the value of the memory word read from address 402 during cycle one.



The IJVM IADD Instruction – Cycle 3

The last two steps we have to do are:

> **Add the two data values together**
> **Write the result back to the memory word that the stack pointer points to**

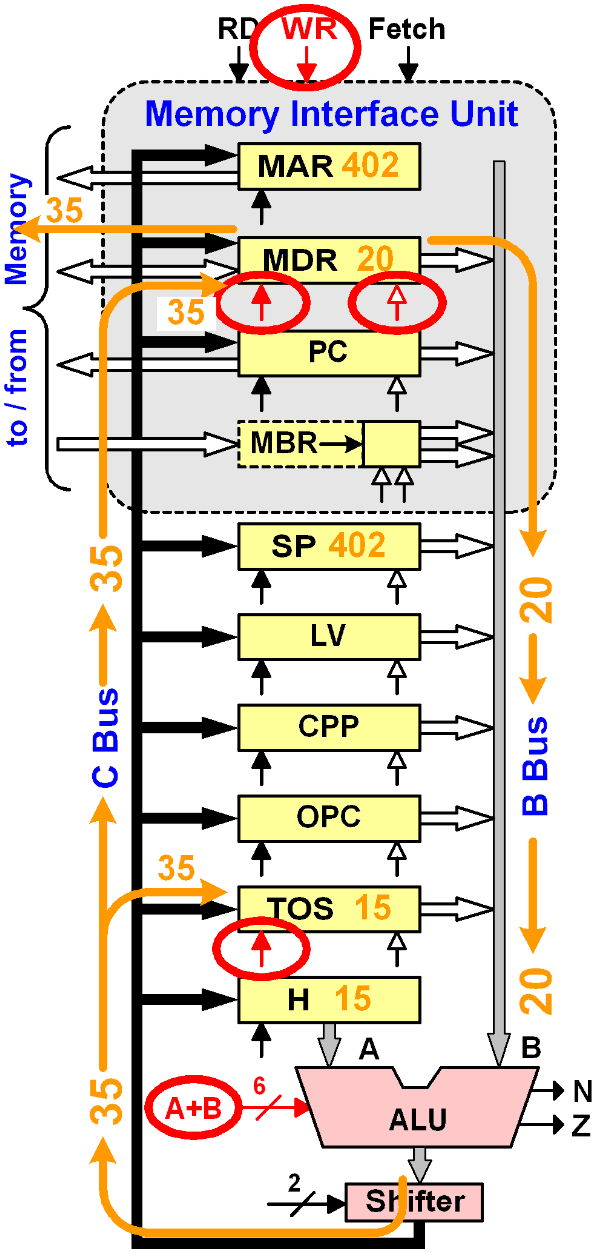
One data value is now in MDR and the other is in TOS, so we need to activate the following control signals:

- **Connect the output of the MDR register to the B bus**
- **Select the ALU "A+B" function (adds MDR and H)**
- **Load the MDR register from the C Bus result**
- **Load the TOS register from the C bus result**
- **Activate the memory control unit "WR" signal**

This adds together the two data values and puts them in the MDR and TOS registers:

- **We put the result in TOS because this is going to be the new "top of stack" word and we need to make sure TOS has the same value for subsequent use.**
- **We put the result in MDR because we need to write it to memory**

Finally, the memory control "WR" signal write the resulting MDR value to the memory address in the MAR register.



The Data Path Control Signals

We’ve figured out how to execute an IADD instruction! All we needed to do was to assert a particular series of control signals to the registers, the ALU, and to the memory control unit. By sending the right signals in the right order, we made the data path do the work that was needed.

This example shows us that making the data path do the work we want is simply a matter of sending it the right control signals at the right time.

Control signals drive the data path

Control signals are the key to making the MIC-1 do what we want. To make our microarchitecture do what we want, we need to come up with a way to:

- **Decode the instruction being executed so we know which one it is and what it’s supposed to do.**
- **Select which control signals to assert so that the right work is done.**
- **Be able to step through different sets of control signals, one for each clock cycle.**

Next week we’ll add some the “control logic” that will do this for us. That will give us a complete, functional CPU design that can execute Java programs.

Key Concepts

- **The Java source code, the Java Virtual Machine, and IJVM instructions**
- **Use of the stack to store operands and local variables**
- **How JVM instructions are executed by the microarchitecture**
- **What the sample JVM instructions do**
- **How Java source code is converted into JVM instructions**
- **The components of the MIC-1 data path**
- **How control signals cause the data path to perform an operation**
- **How the memory control unit is used to access memory**
- **Timing of memory reads**
- **Executing an IJVM instruction using several data path cycles**

What’s Next

- **Look at Review Questions for this week**
- **Sign on to WebCT and do Module 5 – “Data Types”**
- **Start working on Assignment 2 (for the material covered so far)**
- **Pre-read the material for Week 8**