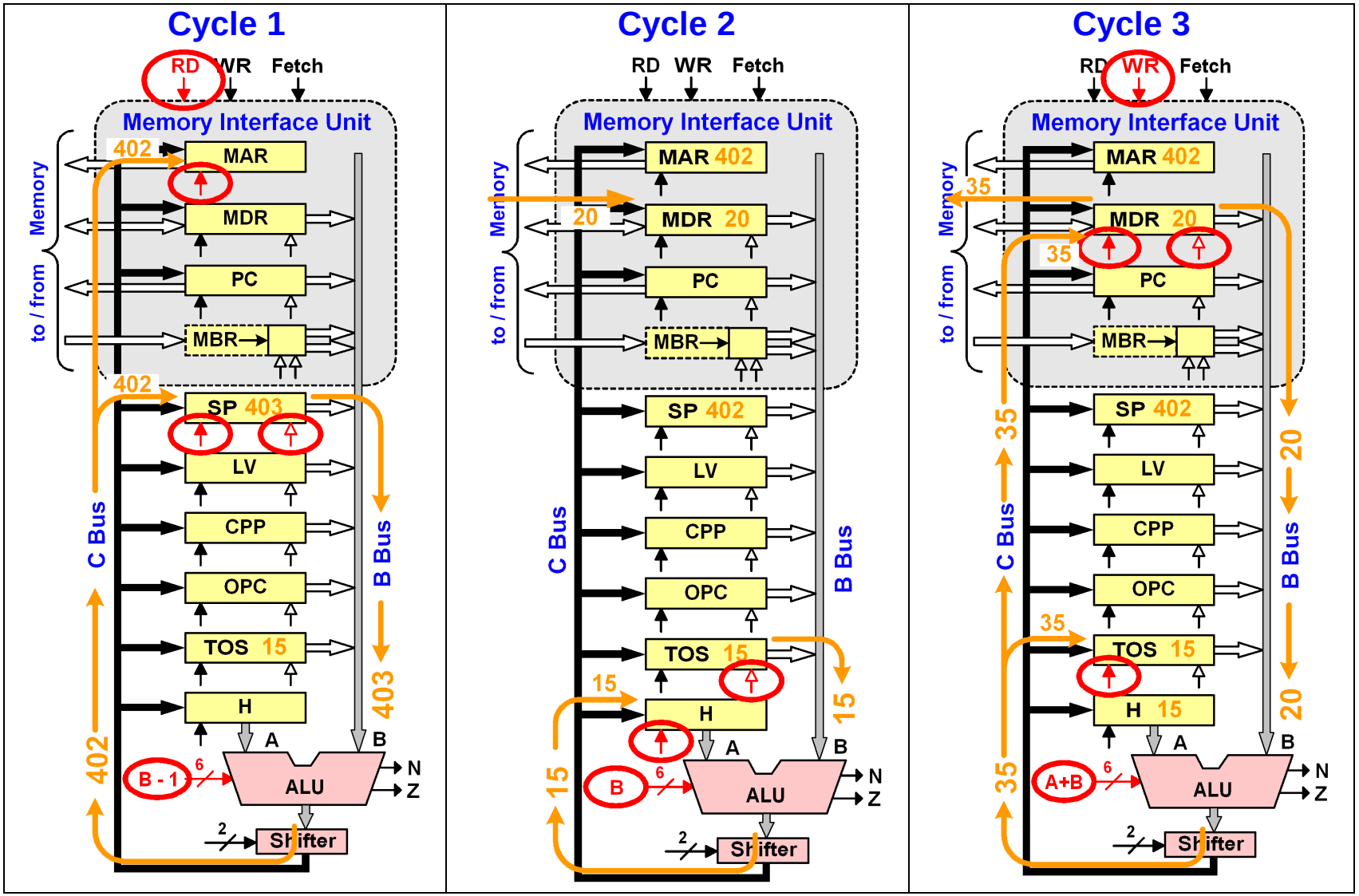
Today's Topics

- **Controlling the data path: the Microinstruction Register**
- **How microinstruction words are sequenced**
- **Decoding IJVM instructions**
- **Use of MAL to write microinstructions**
- **Writing MAL instructions to control the MIC-1 data path**
- **How the MIC-1 registers are used**
- **Decoding IJVM instructions**
- **How the IADD instruction works**
- **How the other IJVM instructions work**
- **How the WIDE prefix works**

Data Path Cycles for IADD

We saw last week how to control the data path of our microarchitecture so that it can do the work of an IADD instruction:



The Data Path Control Signals

As we saw, we can cause the data path to perform different kinds of work by asserting the correct control signals to the register inputs and outputs, to the ALU, and to the memory control unit.

Control signals drive the data path

Control signals are the key to making the MIC-1 do what we want. We need to come up with a way to:

- **Select which control signals we want to assert**
- **Have a way to step through different combinations of control signals, one for each clock cycle.**

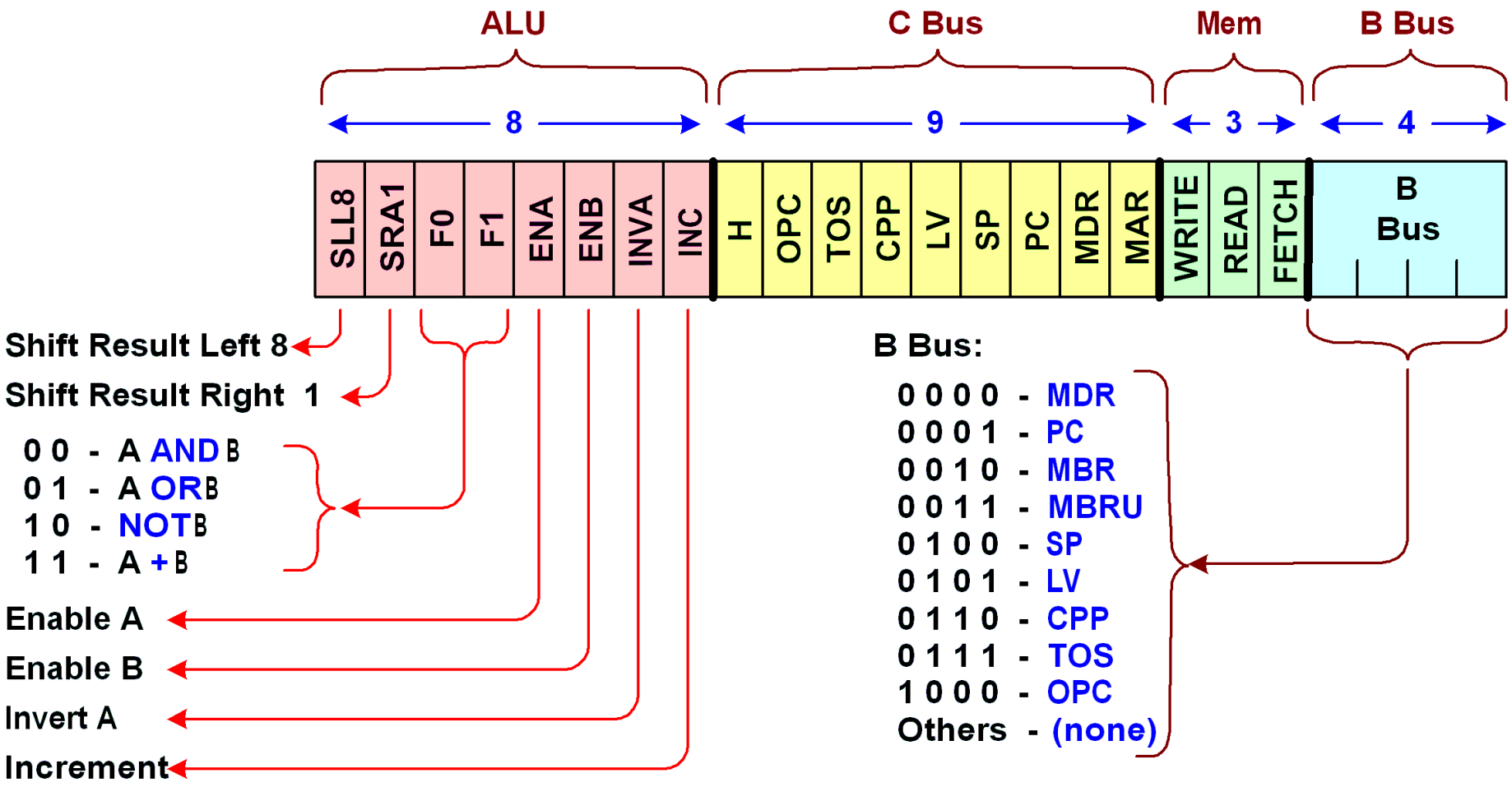
To do this, we're going to use a Microinstruction Register. This outputs of this register will be connected to all of the control signals in the data path. This means that the action taken by the data path will be controlled by what bits are turned on in the Microinstruction Register.

This is called the microinstruction register because each data path cycle is only a small piece of the work needed to execute a complete IJVM instruction such as IADD.

A diagram of the microinstruction register is shown on the next page.

The Microinstruction Register

Here is what the microinstruction register looks like:



The register contains a bit for each of the control signals that drive the data path, except for the control signals that control what register is connected to the B Bus. Since only one register at a time can connect to the B Bus, an encoded field is used to select the register. This saves us a little bit of space.

Driving the Data Path

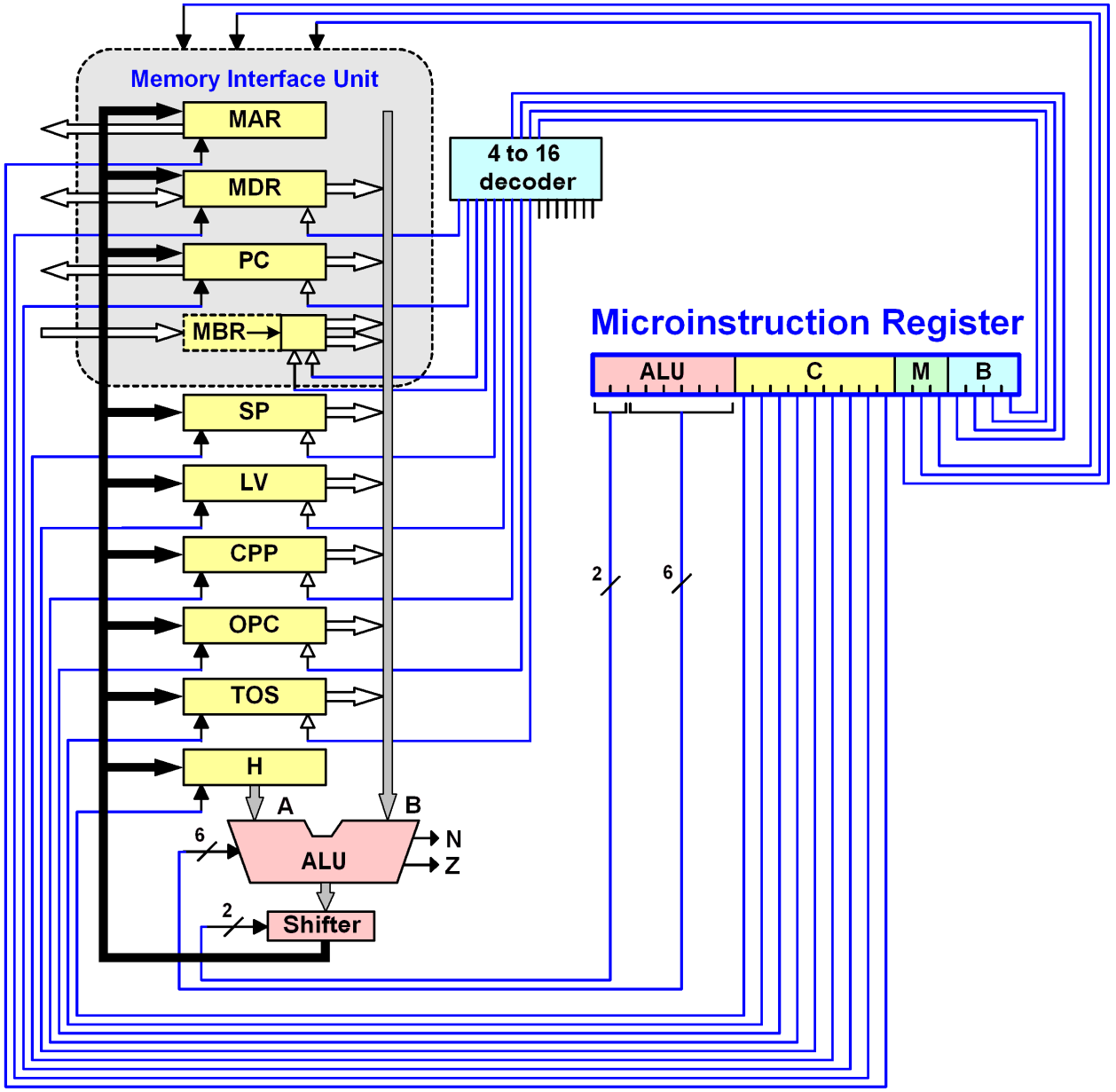
This is how the bits in the microinstruction register (MIR) are connected to the control signals that drive the data path.

Each bit that is turned ON in the MIR will assert the corresponding control signal on the data path. So the pattern of bits in the MIR controls what the data path will do.

Note how the "B Bus" bits are passed through a decoder to select one of the registers to be connected to the B bus.

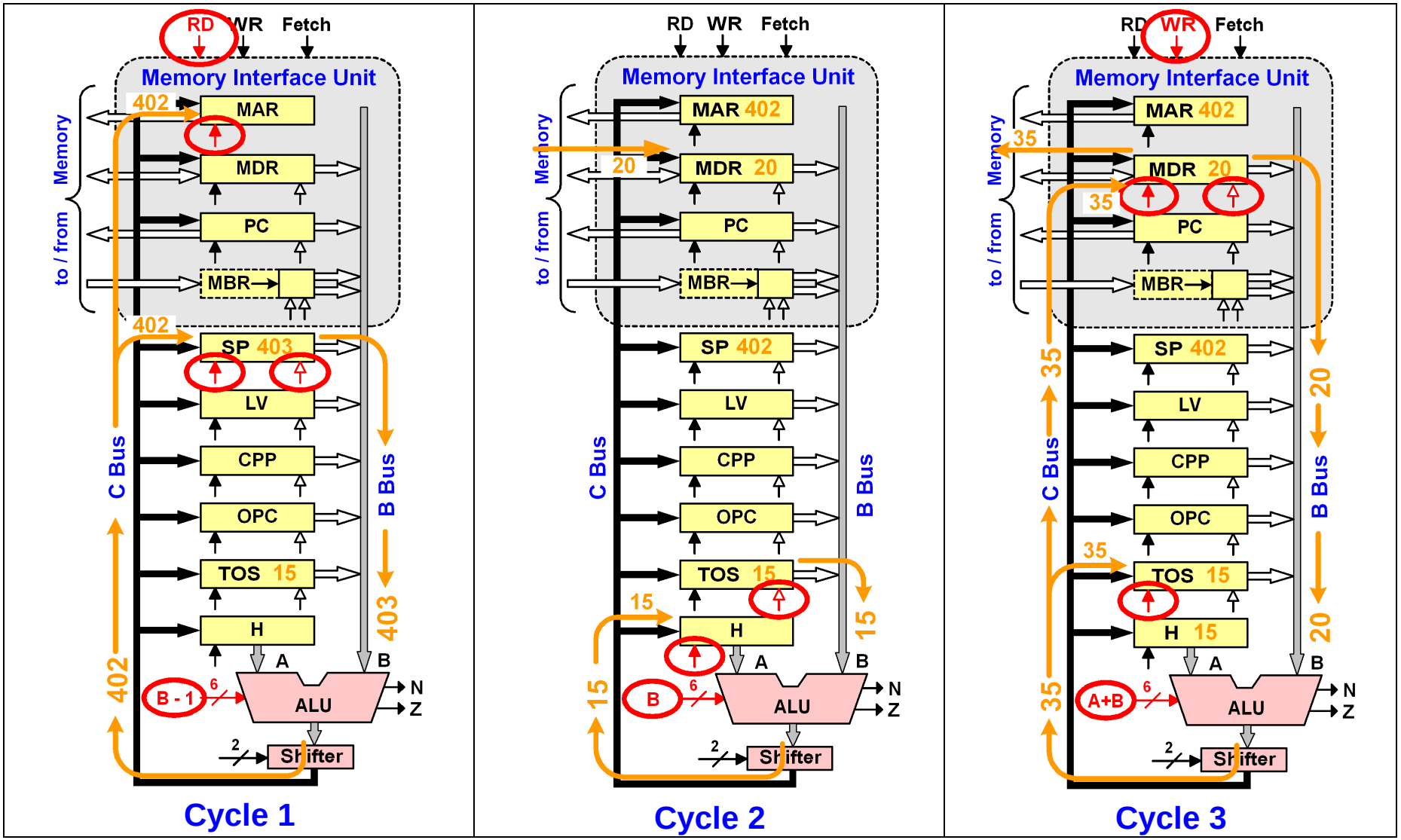
Notice the two control signals on the MBR register for signed vs. unsigned numbers. There are two MIR bits that control this:

"MBR" – signed
"MBRU" – unsigned



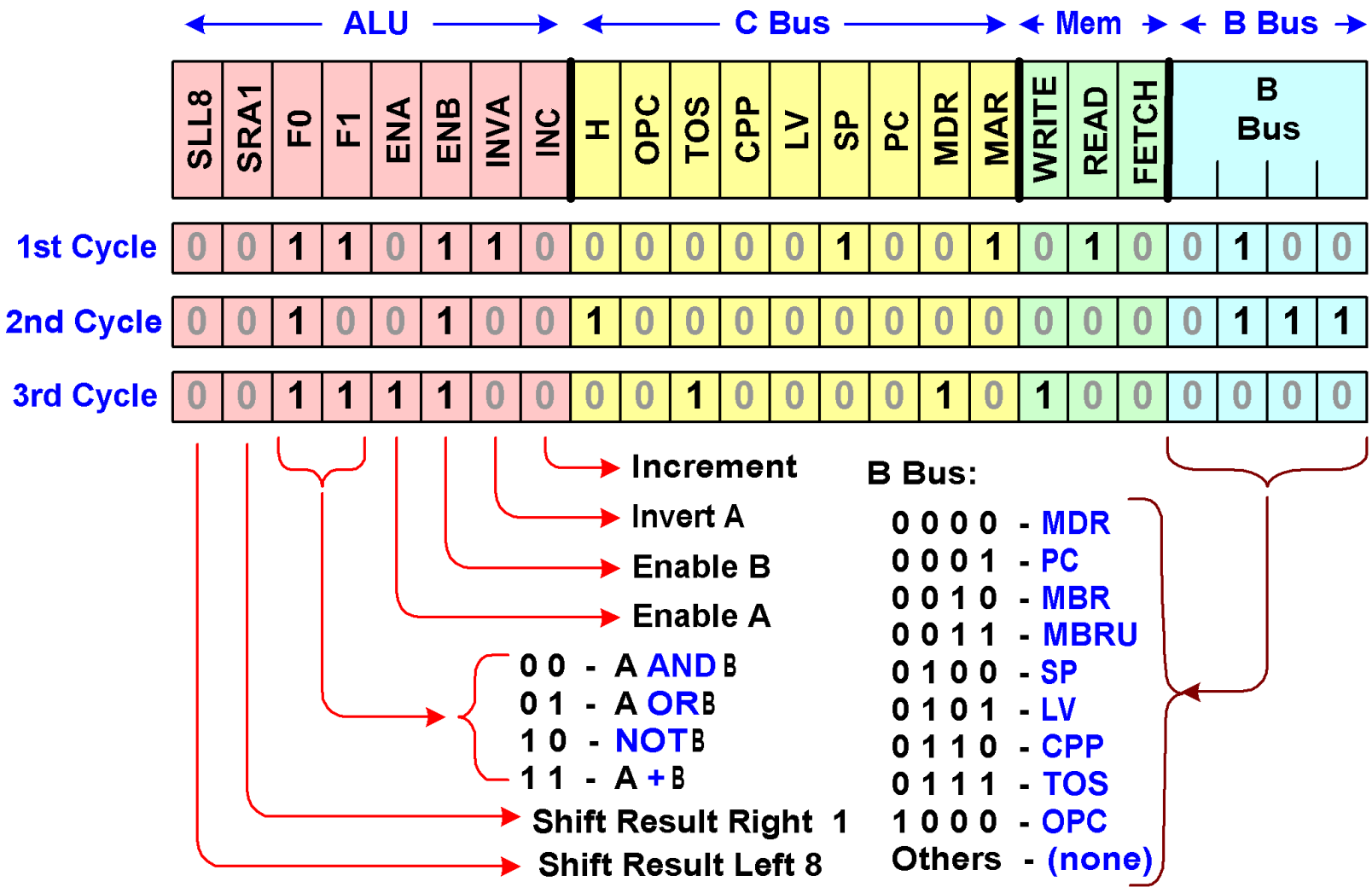
Microinstruction Words for IADD

Let's look again at the data path cycles we used to perform the IADD instruction and see how they translate into microinstruction words. Each cycle required a different combination of control signals to drive the data path:



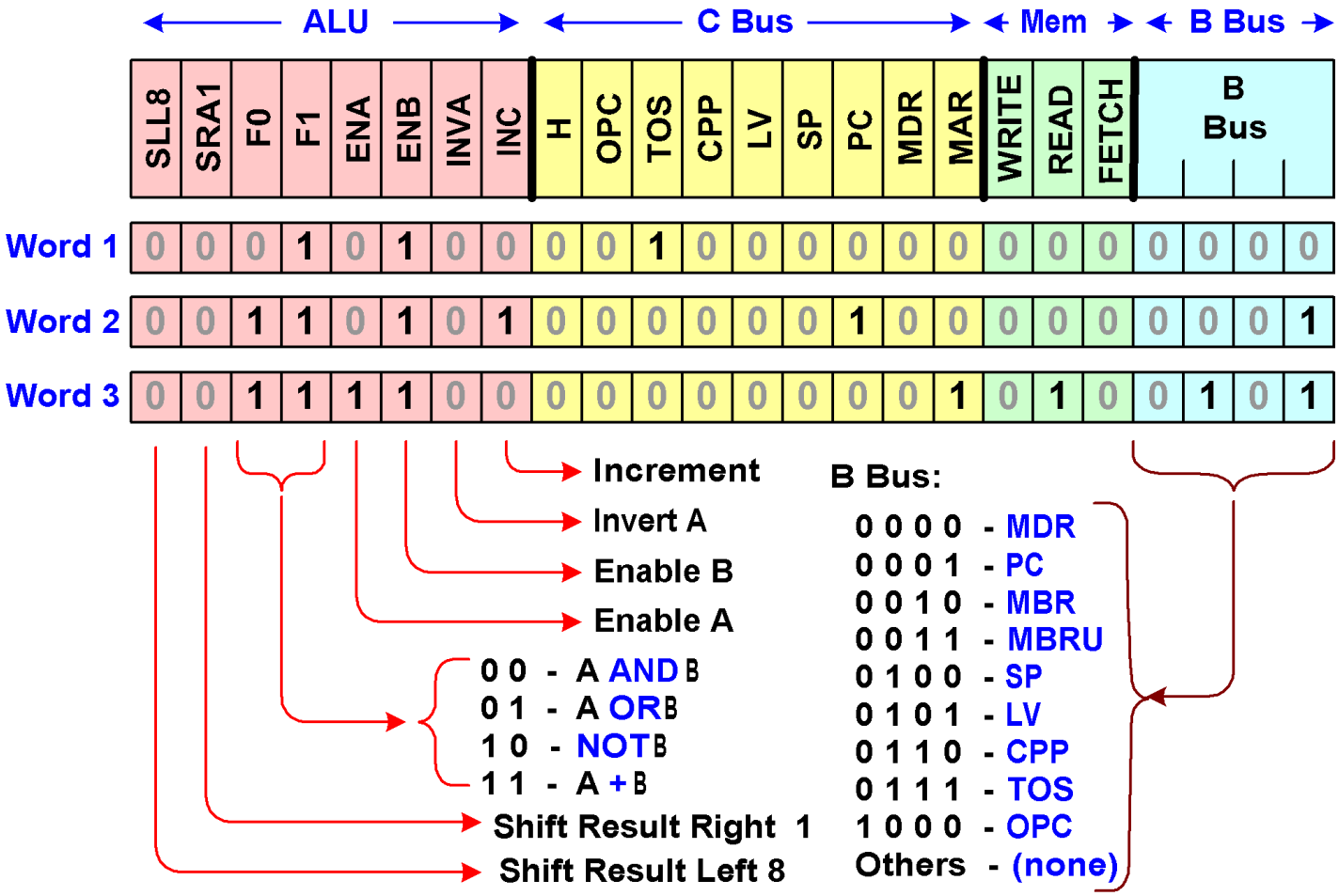
Microinstruction Words for IADD

The control signals to do the three data path cycles needed for IADD translate into three microinstruction words with the following bits:



Exercise 1 – Microinstruction Word

What would the following microinstruction words do?



A – Read from memory at the address equal to the sum of the LV and H registers
B – Add one to the PC register
C – Read the memory word at the top of the stack
D – Copy the MDR register to the TOS register

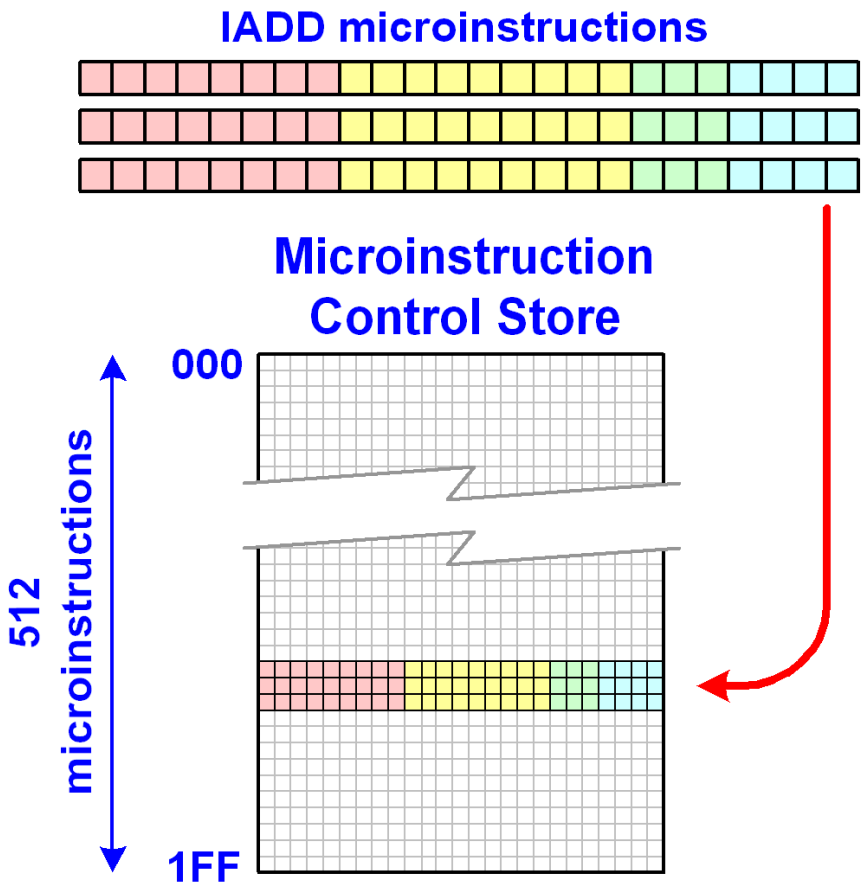
The Microinstruction Control Store

We'll use a Read-Only Memory to store the microinstructions we'll need for all of the IJVM instruction. This memory will be built right into the CPU chip along with the other components of the data path.

The read-only memory will hold 512 cells, and each cell will contain one complete microinstruction. Each cell has an address, and since there are 512 of them then the addresses are 9 bits long and range from 000 to 1FF hex.

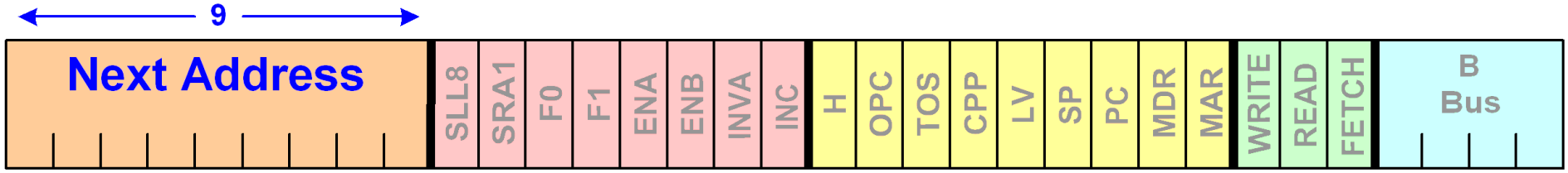
We'll drive the data path of our system by reading microinstructions out of the control store and putting them into the Microinstruction Register (MIR), one per clock cycle.

The last thing we need is a way to figure out which microinstruction to read out of the control store.



Microinstruction Sequencing

We're going to add a Next Address field to our microinstruction word to control which microinstructions are read from the control store. Since there are 512 words in the control store, the Next Address field will need to be 9 bits long:



These extra 9 bits will be part of every microinstruction word in the control store, and on each clock cycle it will control which microinstruction is the next one that's read out of the control store.

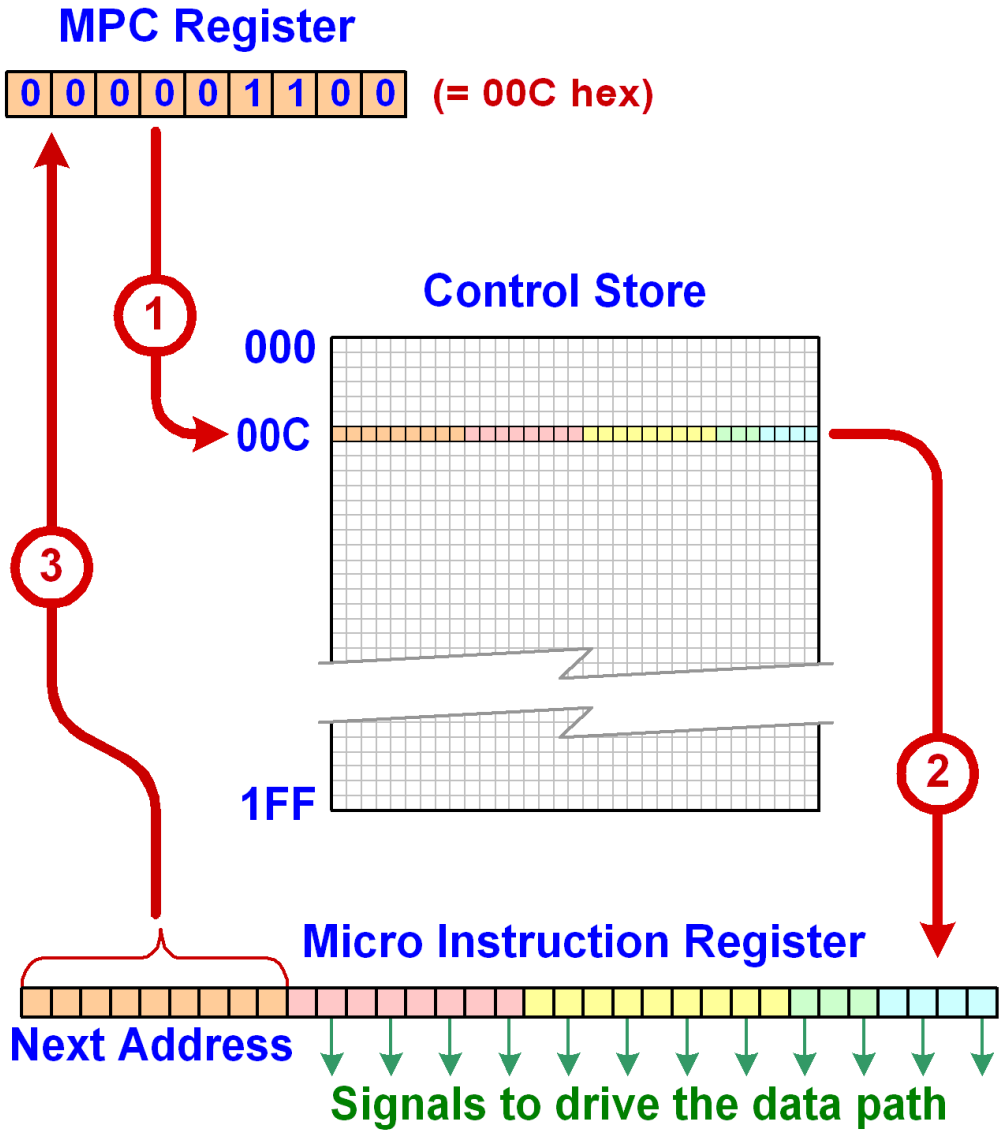
Microinstruction Sequencing

The next microinstruction will be selected using a new register, the MPC (Micro PC) Register. This register is 9 bits long so that it can point to any of the 512 words in the control store.

The sequence of events goes like this:

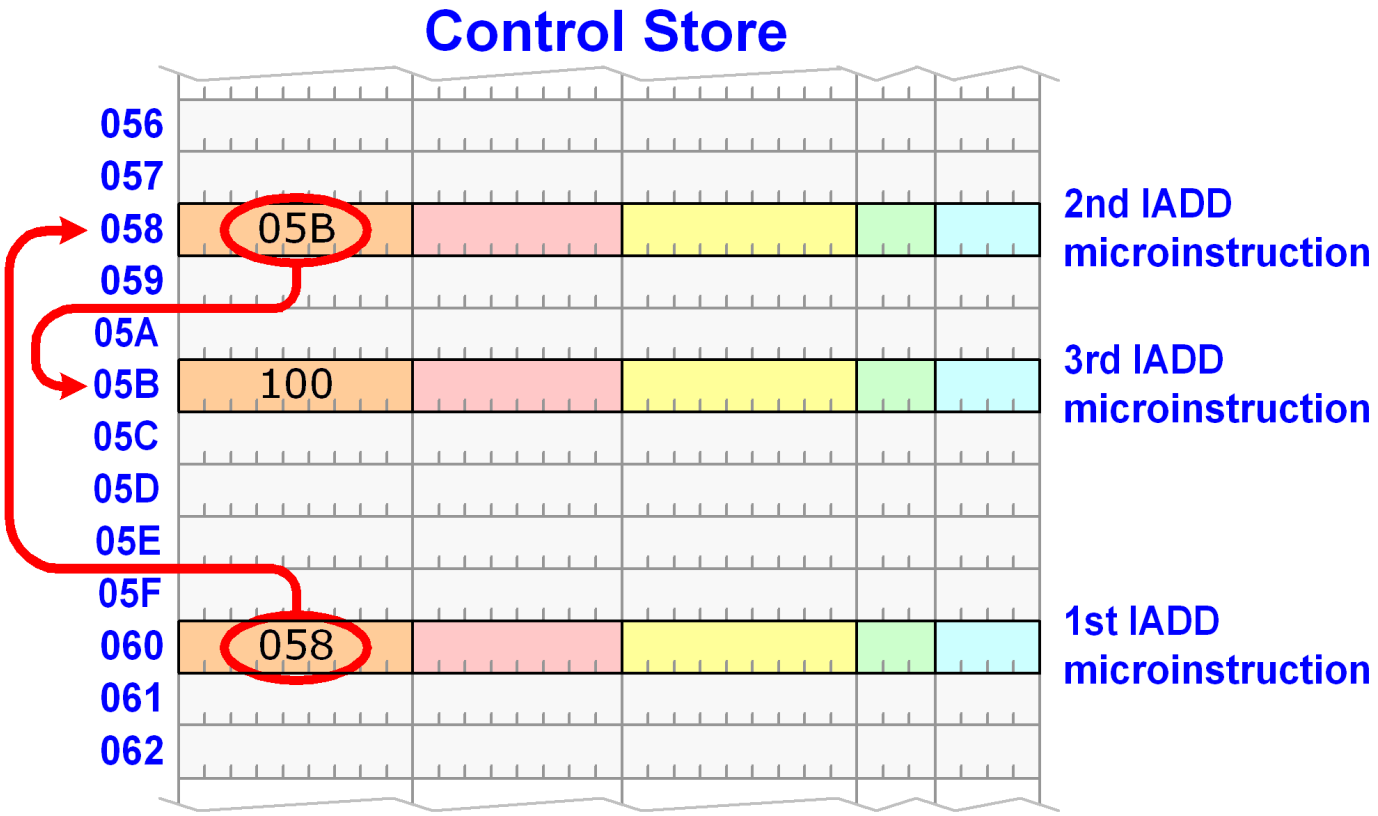
1. **The MPC register contains the address of the next microinstruction**
2. **The selected microinstruction is read from the control store and loaded into the MIR register. The ALU, B, C and MEM bits of the register are sent to the data path to control it's operation.**
3. **The "Next Address" field in the microinstruction is moved into the MPC register.**

These steps are repeated over and over.



Microinstruction Sequencing

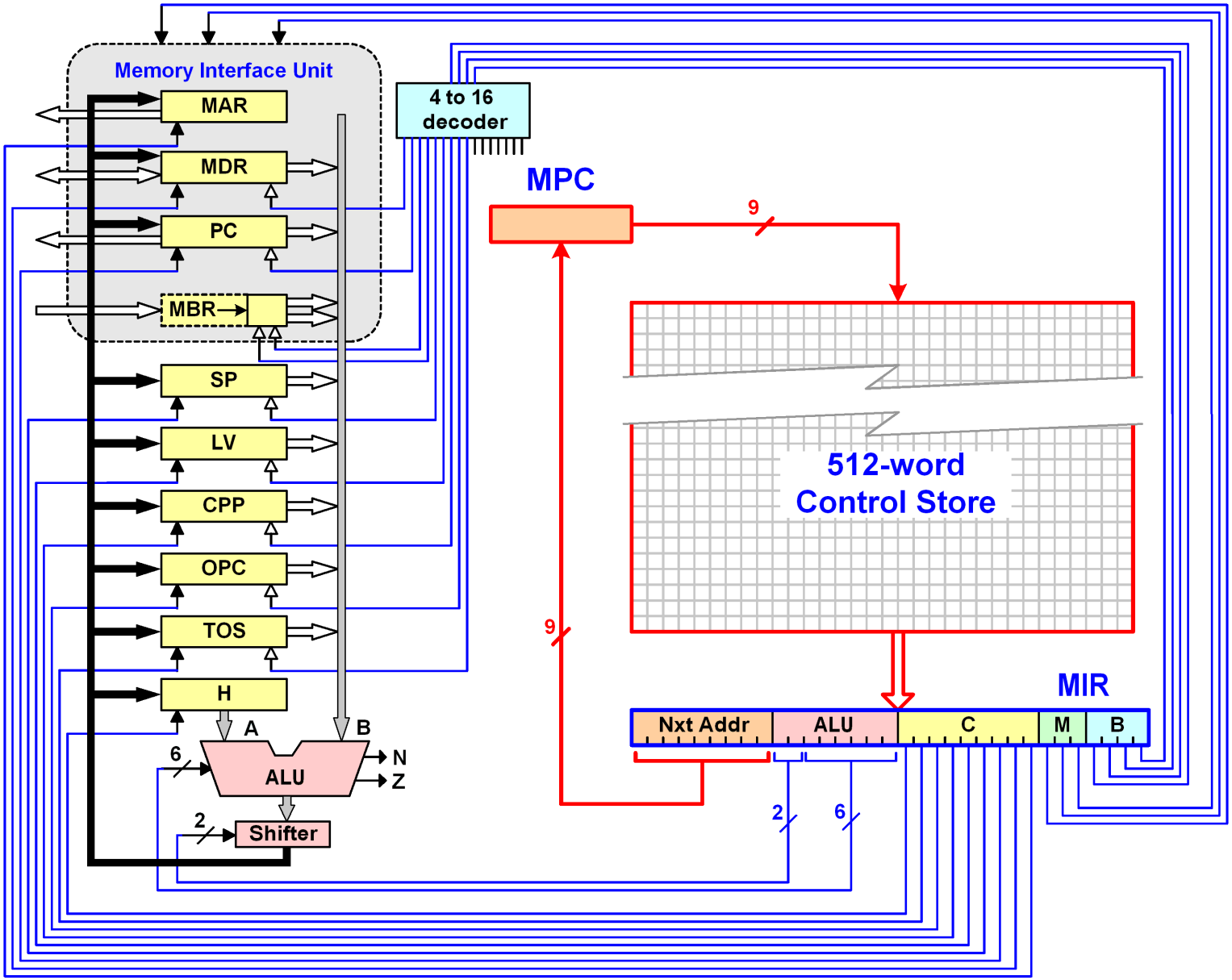
Since each microinstruction specifies the control store address of the next microinstruction, a series of microinstructions doesn't have to be put into consecutive addresses in the control store. For example, the three microinstructions which do the work of the IJVM "IADD" instruction could potentially be located as follows in the control store:



Each microinstruction "links" to the next one. So, for example, after executing the 1st IADD microinstruction, the "058" in it's next address field is loaded into the MPC register, so the next microinstruction is the 2nd IADD microinstruction read from control store address "058".

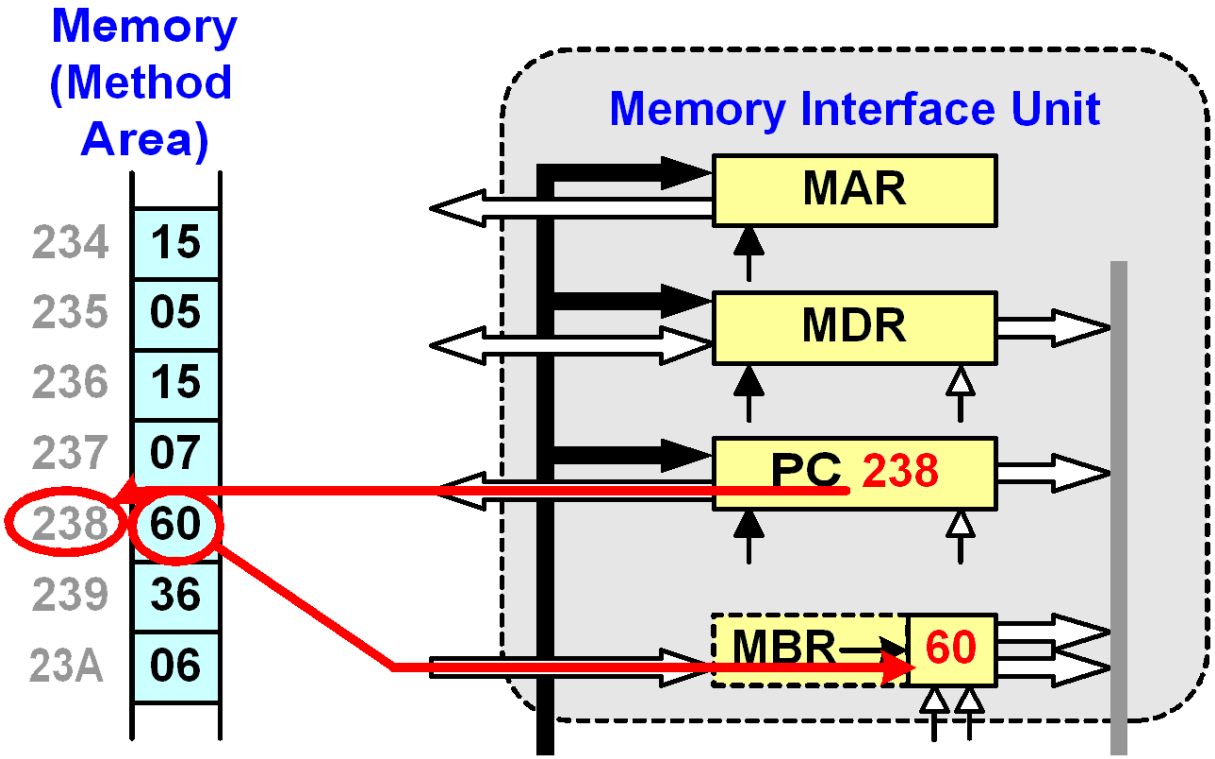
Data Path, MIR and Control Store

Here's what our MIC-1 microarchitecture looks like so far:



Instruction Decoding

What's missing from our microarchitecture so far is a means to decode the IJVM instructions fetched from the method area. In the following example, the PC is pointing to a byte in the method area containing hex "60", and that byte is fetched into the MBR register:

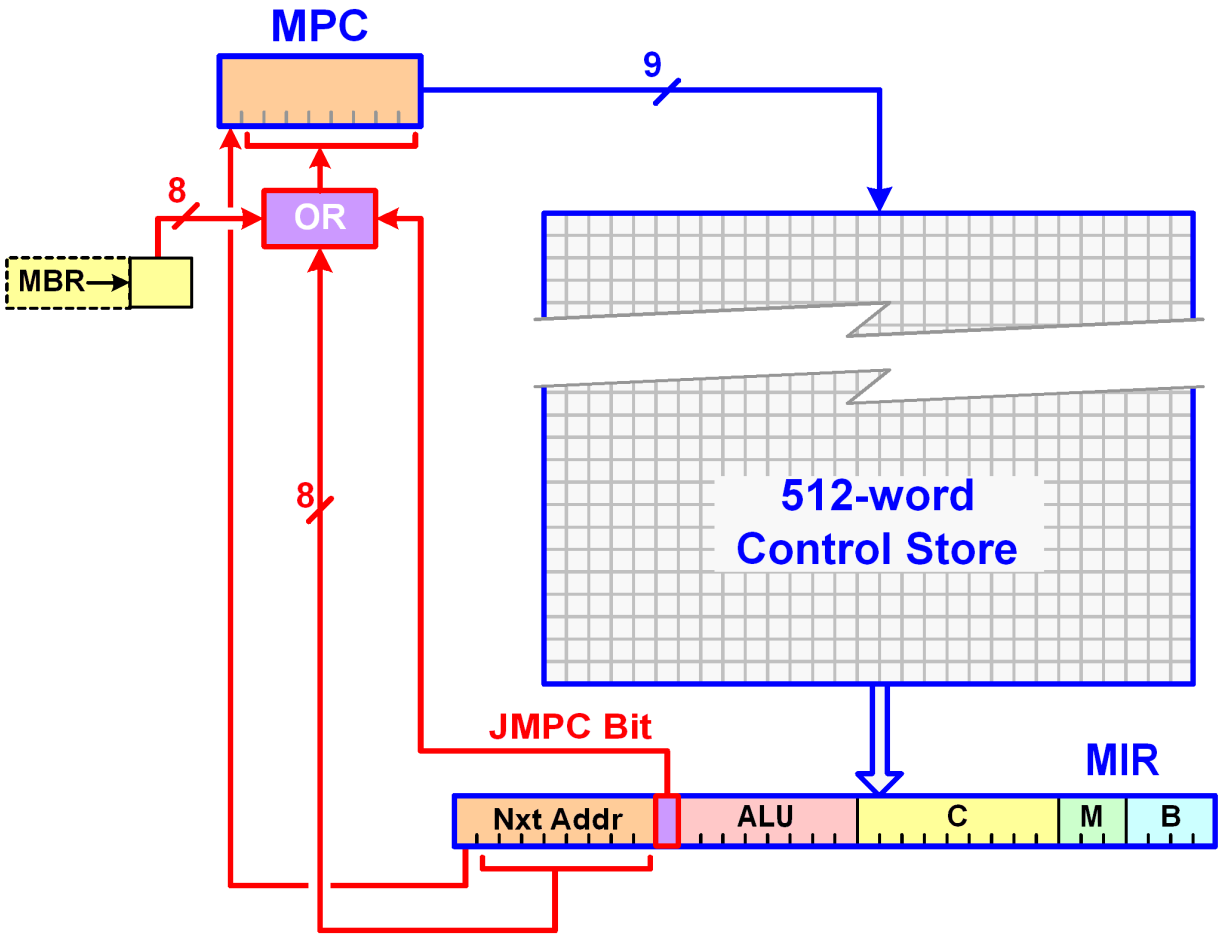


Hex "60" is the opcode for the IJVM "IADD" instruction. How do we know which set of microinstructions to use when we see this opcode?

Instruction Decoding

We're going to arrange for the contents of the MBR register to be sent to the MPC register so they can be used to determine the next microinstruction address. We'll do this by:

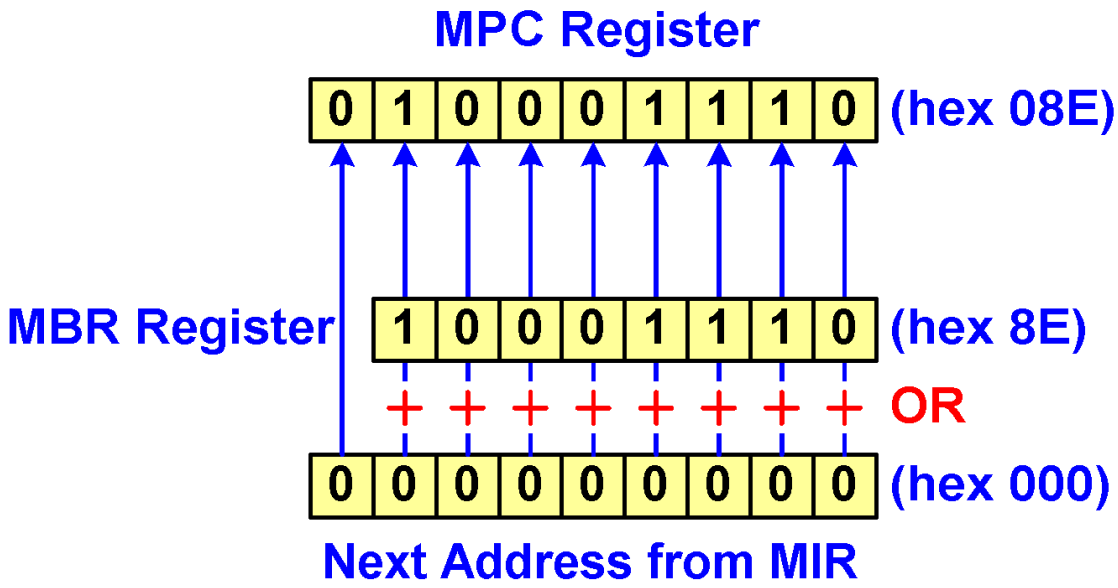
1. **Adding a new bit called "JMPC" (Jump to MPC) to the microinstruction word. When this bit is turned on it will signal that we want the MBR contents to be incorporated into the next microinstruction address.**
2. **Passing the low-order 8 bits of the microinstruction "next address" field through an optional "OR" circuit. This circuit will "OR" the contents of the 8-bit MBR register with the low-order 8 bits of the microinstruction "next address" field if the JMPC bit is TRUE.**



As shown in the diagram, the high-order bit of the "Next Address" is not affected by this operation.

JMPC Example

This example shows how the JMPC bit operates:

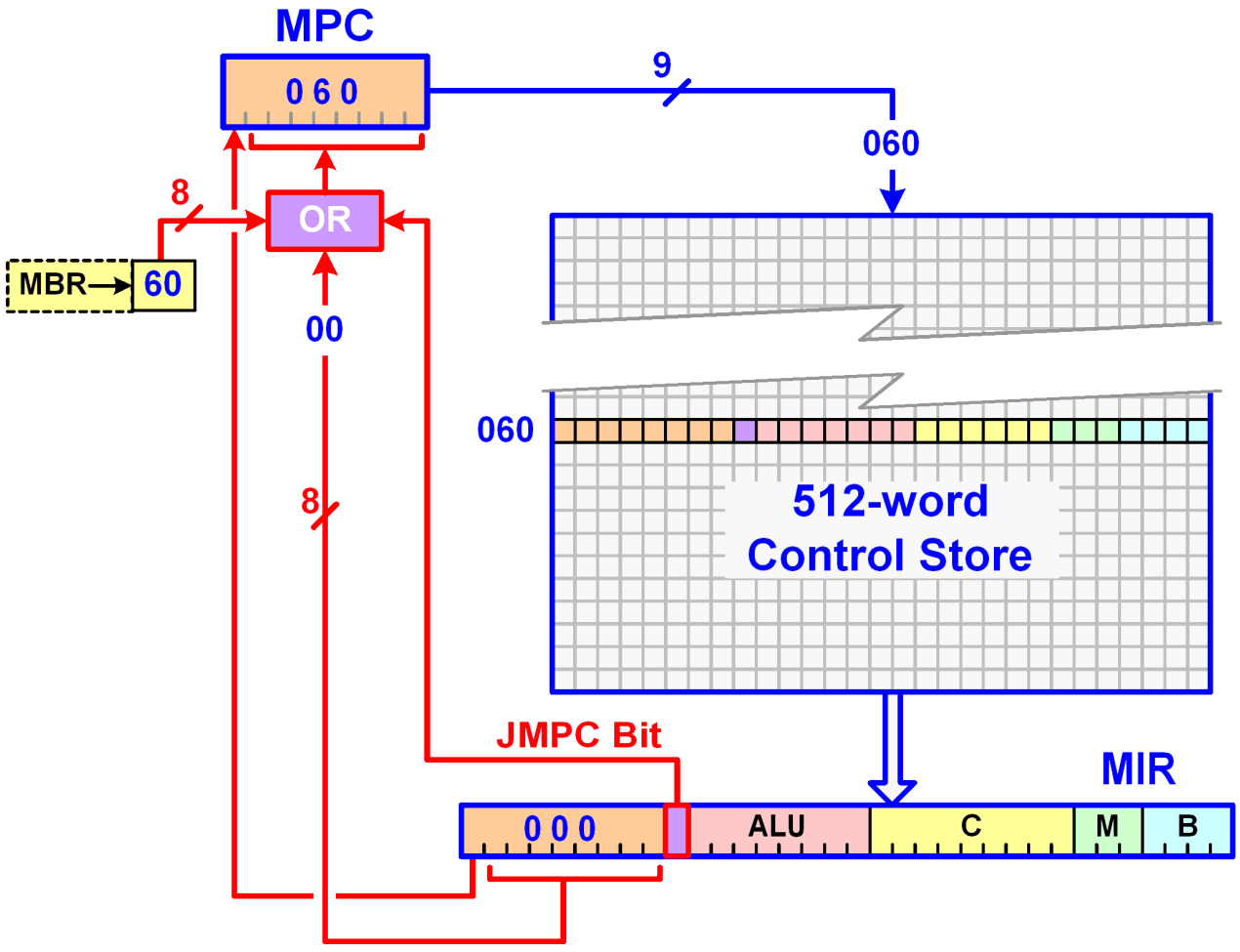


When the JMPC bit is turned on in the microinstruction word, the low-order 8 bits of the "next address" are ORed with the 8-bit MBR register. The result in the MPC register is used to select which microinstruction word to read from the control store next.

Instruction Decoding

Here's an example of how the JMPC bit can be used to decode the IJVM "IADD" instruction:

1. **The "next address" field in the MIR register is hex 000, and the JMPC bit is TRUE.**
2. **The MBR register contains hex 60 because the IADD opcode has been fetched from the method area.**
3. **The JMPC bit is TRUE, so the OR circuit ORs the MBR register with "Next address" field and the resulting hex 60 is put into the MPC register.**
4. **The next microinstruction will be read from control store address hex 060.**



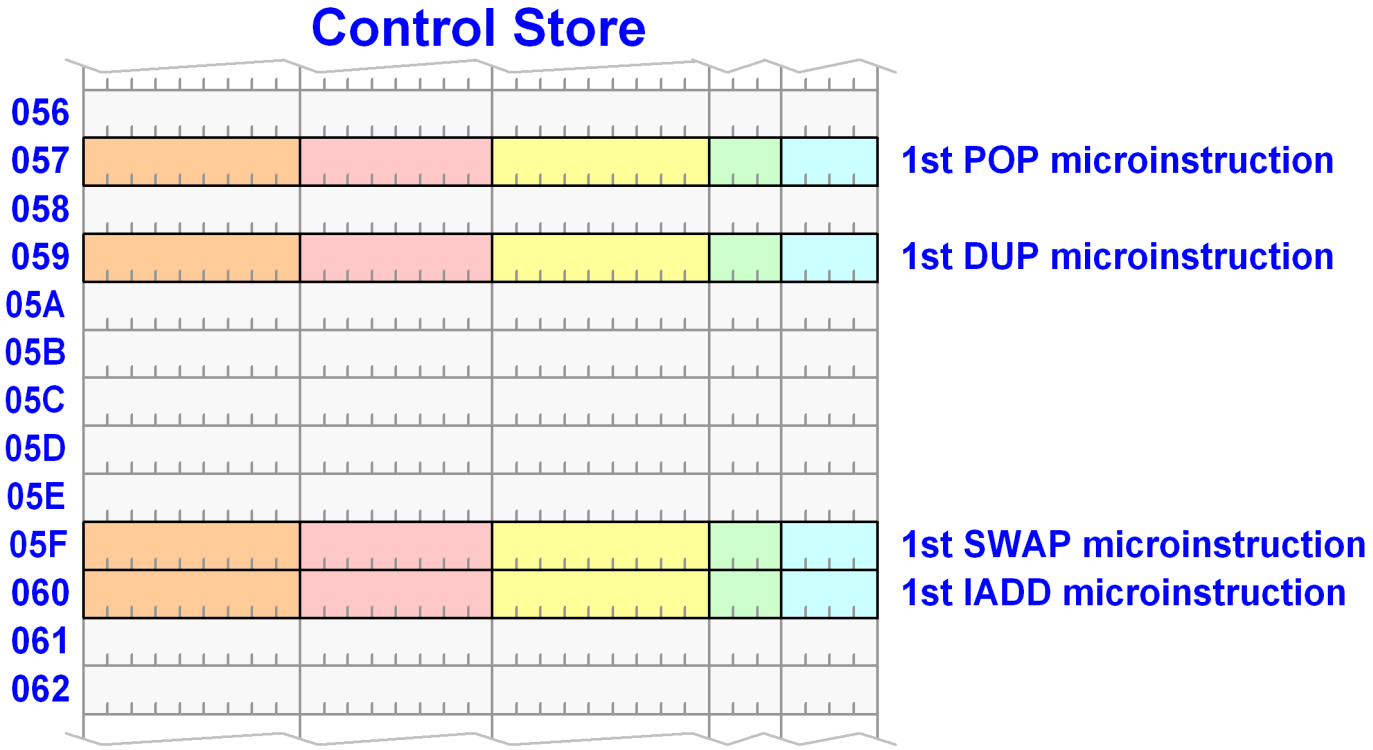
The microinstruction at control store address hex 060 is the first of the microinstructions that does the work for the IJVM IADD instruction.

Instruction Decoding

So the secret to decoding IJVM instructions is that the opcode is read into the MBR register, and then the JMPC bit is turned on so that the opcode is used as the next microinstruction address. Therefore, the first of the microinstructions that does the work of each IJVM instruction is located at a control store address that is the same as the opcode for the instruction. Note the opcodes of the following instructions:

DUP	59
IADD	60
POP	57
SWAP	5F

As you can see, the first microinstruction for each of the IJVM instructions is located at the control store address that is the same as the opcode.



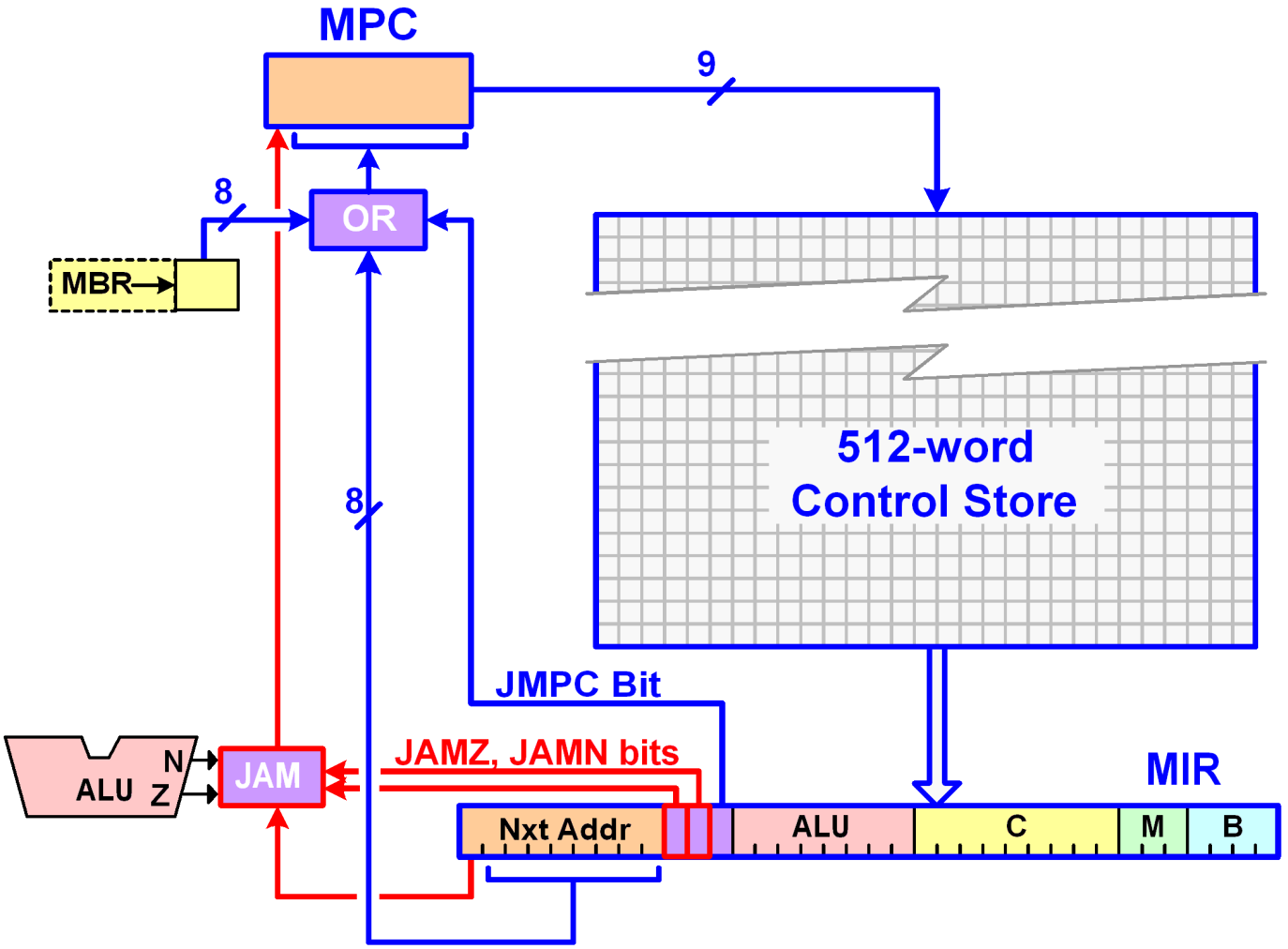
Notice that SWAP at hex 5F is immediately followed by IADD at hex 60. There's no room for a second SWAP microinstruction right after the first one. This is why each microinstruction has a "next address" to point to the next one in sequence – it allows the microinstructions to be placed at free addresses that don't correspond to an IJVM instruction opcode.

Conditional Branching

There's one more thing we need to make our microarchitecture fully functional – the ability to conditionally branch to one of two different microinstructions. We need this to do the work for IJVM conditional branch instructions such as IF_CMPEQ.

We add the ability to do a conditional branch as follows:

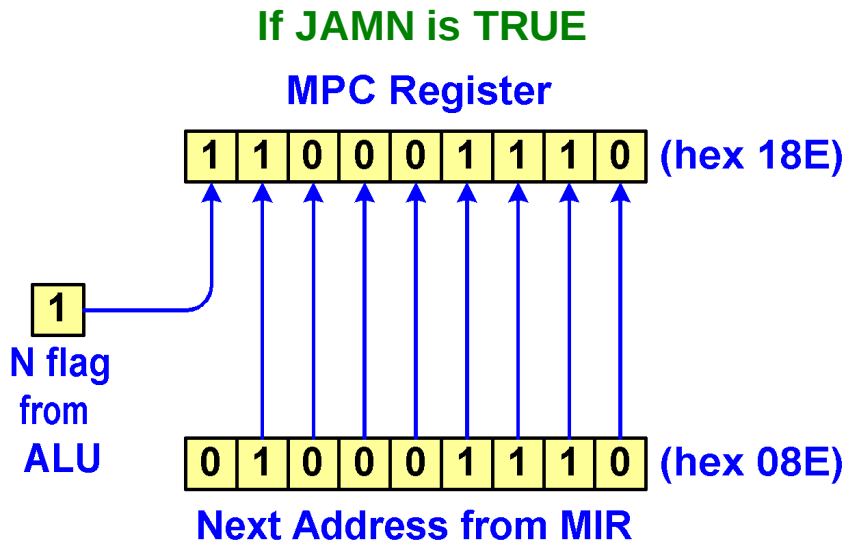
1. **We add two new bits called "JAMN" and "JAMZ" to the microinstruction word.**
2. **These two bits control a "JAM" circuit which replaces the high order bit of the next address value with:**
   - **the ALU's N flag (if JAMN is TRUE)**
   - **the ALU's Z flag (if JAMZ is TRUE).**

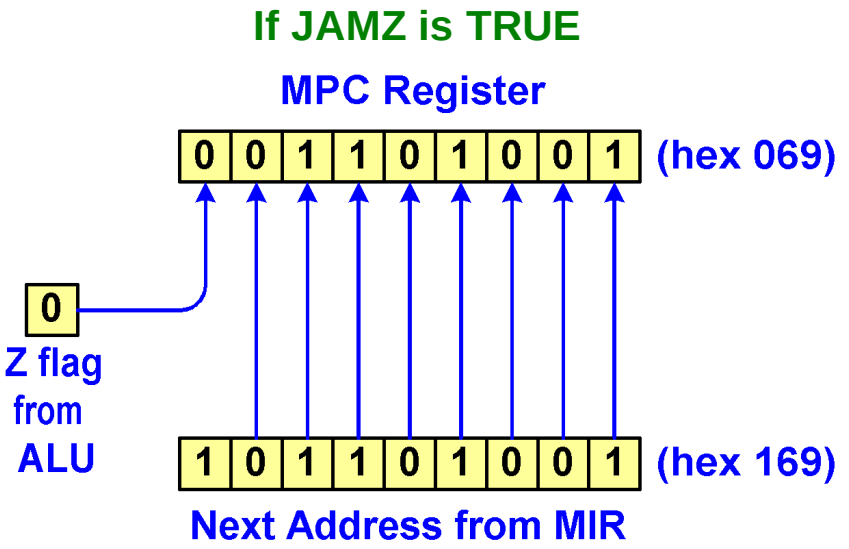


This lets us choose one microinstruction word or another depending on the result that comes out of the ALU.

JAM Bits Examples

Here are a couple of examples of how the JAM bits work:





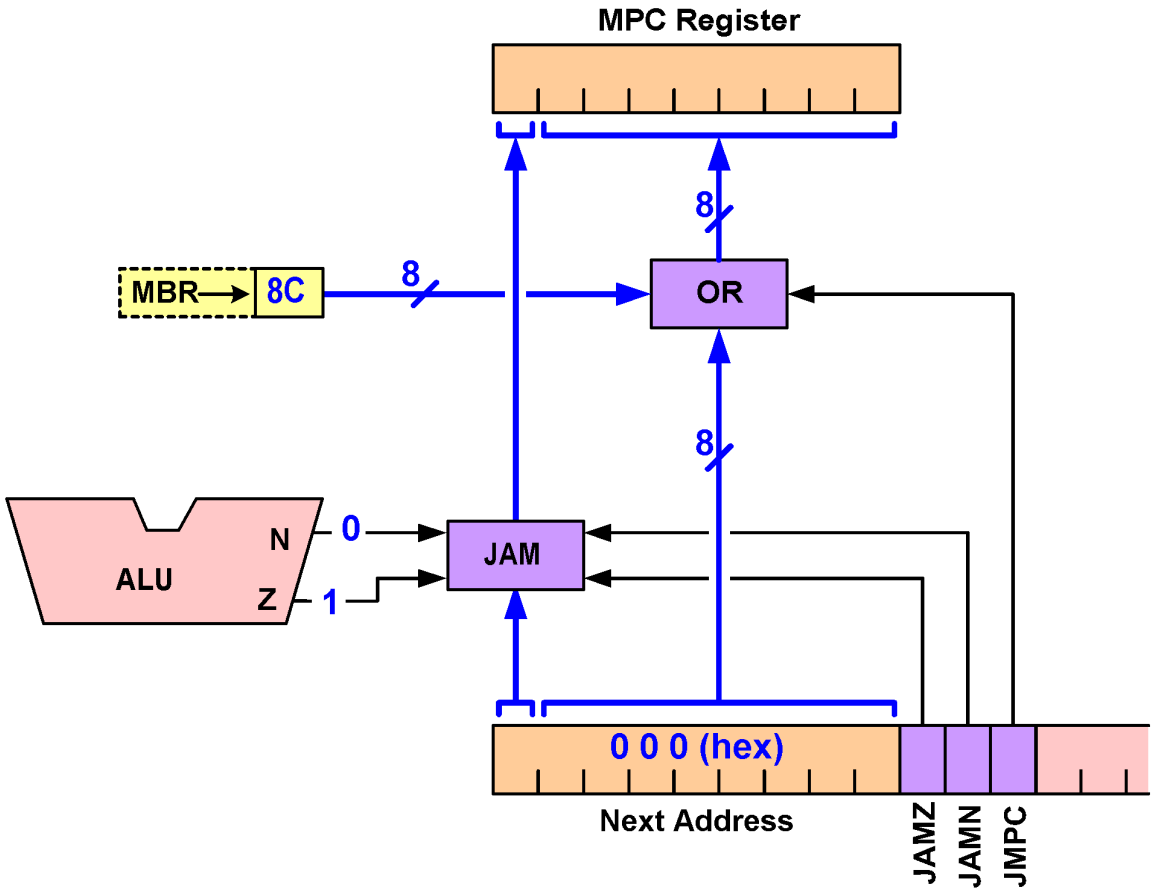
When the JAMZ or JAMN bit in the microinstruction word is TRUE, the high order bit of the next microinstruction address is replaced by the appropriate ALU flag (which will have a TRUE or FALSE value depending on the result that comes out of the ALU)

When the JAMZ and JAMN bits are both false, the next microinstruction address is used as is without change.

Exercise 2 – JAM / JMPC

The microinstruction "Next Address" and ALU "N" and "Z" outputs are shown in the diagram:

1. **What will be put into the MPC register if the JAMZ bit is turned on?**
2. **What will be put into the MPC register is the JAMN bit is turned on?**
3. **What will be put into the MPC register if the JMPC bit is turned on?**



A 000 hex

B 08C hex

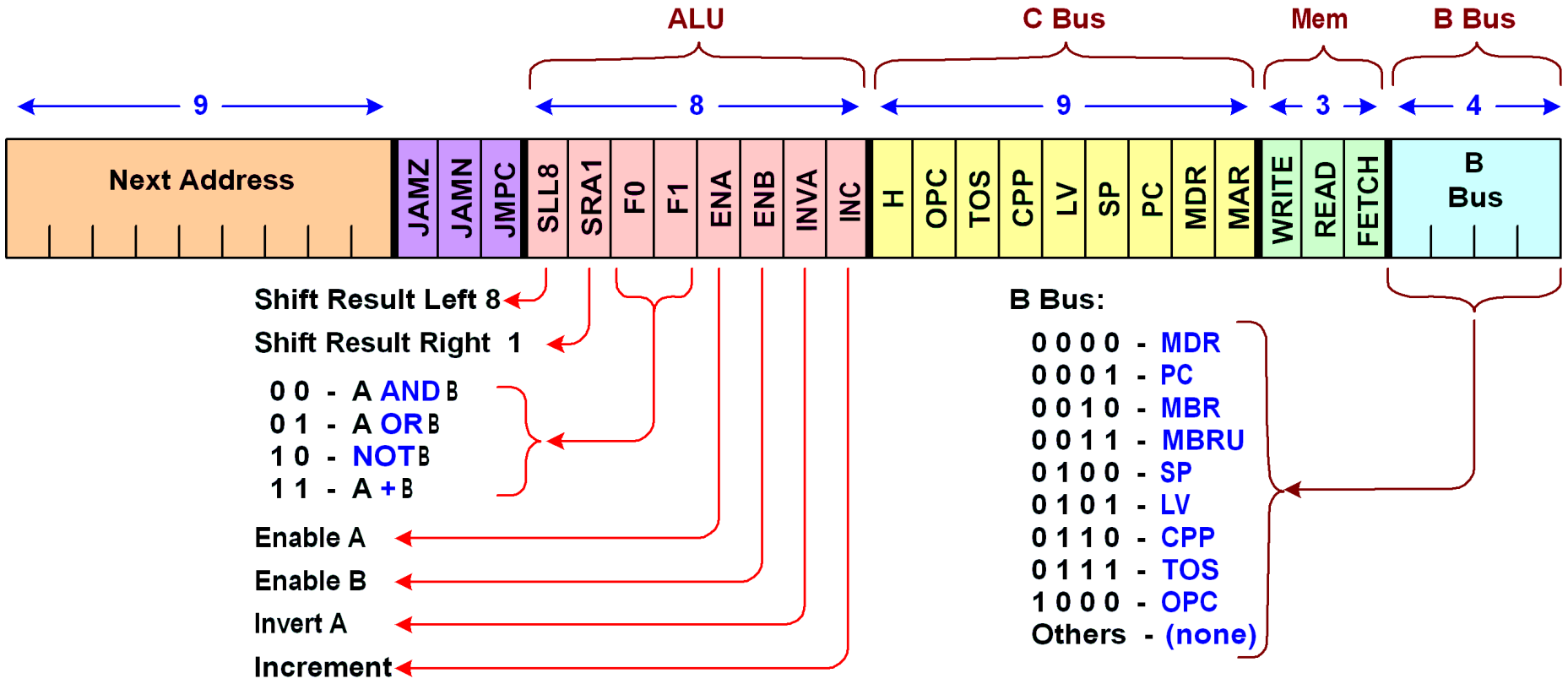
C 100 hex

D 18C hex

E none of the above

The Complete Microinstruction Word

This is the final layout of the microinstruction word for the MIC-1 microarchitecture:

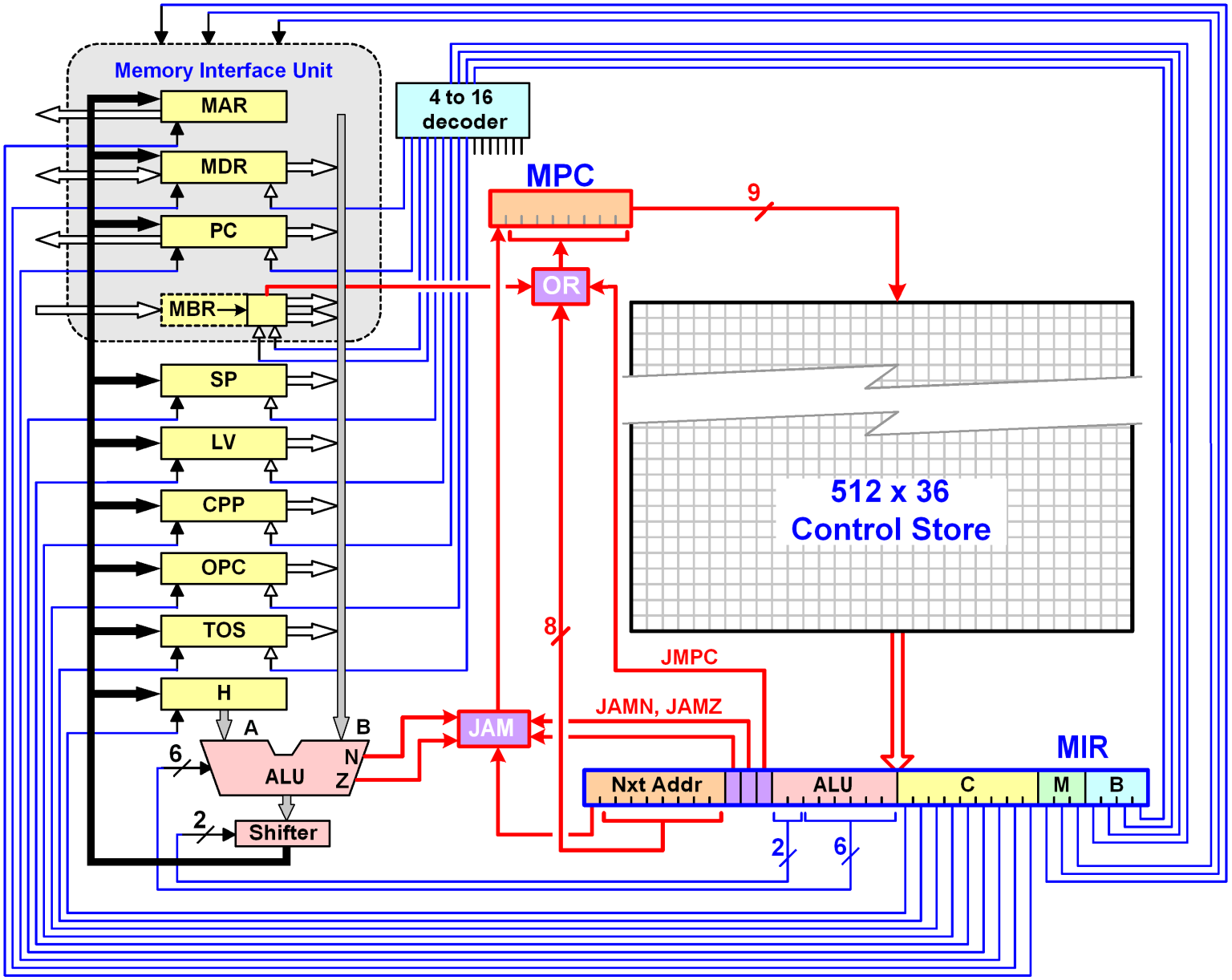


This word contains 36 bits, so our control store will have to be able to store 36 bits per cell.

(Note – the order of the some of the bits is different from that shown in the textbook, but that's not important)

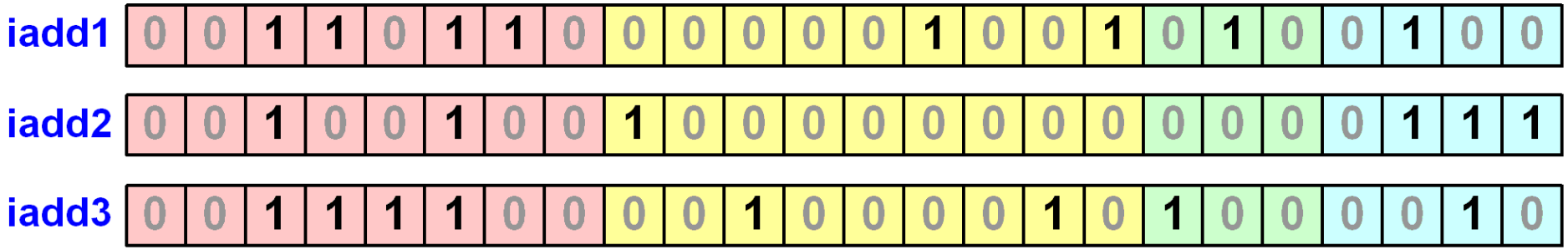
The Complete Microarchitecture

And here is the complete microarchitecture for the MIC-1:



Using MAL to Write Microinstructions

Now that microarchitecture is complete, all that remains is to create microinstructions that will drive the data path to do the work that is required of each IJVM instruction. We've already figured out most of what needs to be done for the IADD instruction. Remember these microinstructions?



Although these microinstructions do the work needed for an IJVM "IADD" instruction, they're very hard to understand. Rather than creating microinstructions out of "1"s and "0"s, we'll use a symbolic notation that the textbook's author calls "MAL" (Micro Assembler Language).

Using MAL, we'd write the above microinstructions as:

```
iadd1    MAR = SP = SP – 1;  rd
iadd2    H = TOS
iadd3    MDR = TOS = MBR + H; wr
```

This notation makes it much easier to understand what the microinstructions do. But it's important to remember that the symbolic notation translates directly into the "1"s and "0"s of the microinstruction word, and that those "1"s and "0"s drive the data path control signals and control the work that our microarchitecture does.

How to Write Microinstructions in MAL

The theory in writing microinstructions is to create them as text and then feed them into a "MAL Assembler" which will convert them into bits and decide where they go in the control store. In fact, no such "MAL Assembler" exists, but we're going to pretend that it does.

The basic format of the microinstruction is a label followed by the symbolic language that describes what the microinstruction does:

```
iadd1    MAR = SP = SP – 1;  rd
iadd2    H = TOS
iadd3    MDR = TOS = MBR + H; wr
```

The convention for writing the labels is to use the name of the IJVM instruction that these microinstructions are for, followed by "1", "2", "3" for the sequence that they are executed in.

We don't have to specify the control store addresses for the microinstructions – the MAL Assembler would do this. When we write microinstructions sequentially as in the above "IADD" example, the MAL Assembler will place the microinstructions at available addresses in the control store and then set up the "next address" field in each microinstruction so that it points to the next. This way, the microinstructions are executed in the order they are written, even if they are not in consecutive addresses in the control store.

A real MAL Assembler would need some sort of table to tell it what opcode is used for each IJVM instruction. It would need this information because the first microinstruction in each sequence must be stored at a control store address that matches the IJVM instruction's opcode. But we're going to just assume the existence of that table and not worry about it for the microinstructions that we write.

How to Write Microinstructions in MAL

There are four parts to a microinstruction written in MAL:

```
main      PC = PC + 1;      fetch;    GOTO (MBR)
(1)          (2)             (3)          (4)
```

(1) Label

As discussed above

(2) Data Path Operation

The data path operation describes the control signals that drive the data path. It includes the register which will be connected to the B Bus, the ALU and shifter operation, and the register(s) which will accept the result from the C bus.

(3) Memory Operation

"fetch", "rd" (read), or "wr" (write) are written to activate the equivalent control signals for the memory control unit.

(4) Next Address

The next address field is normally left blank, but a "GOTO" or "IF" clause can be included which will cause the MAL assembler to set the "next address" field of the microinstruction word to the appropriate value and turn on the JAMN, JAMZ or JMPC bits in the microinstruction word if necessary.

MAL – Data Path Operation

The data path operation portion of a MAL statement looks similar to an assignment statement in a high level language:

PC = PC + 1

But this statement directly translates into microinstruction control bits which will cause the data path to perform the written operation. For example, the above statement creates a microinstruction word with the following bits turned on:

- **The "B Bus" field is set "PC Register" so that the PC register is connected to the B Bus**
- **The "ALU" field is set to the "B+1" function**
- **The "C Bus" field is set to load the C Bus result into the PC register.**

Unlike a high level language statement, the data path operation can specify multiple outputs:

MDR = TOS = H + TOS

Everything to the left of the last equal sign is used to set which registers are loaded from the "C Bus" result.

MAL – Data Path Operation

You must be careful not to write MAL statements that the data path is not capable of doing. For example, the following statement is not valid:

MAR = LV + MDR

…because there's no way for the MIC-1 data path to put both the LV and MDR registers onto the B bus and add them together at the same time.

At the right is list of the operations that the data path is capable of.

"SOURCE" represents any single register that can be connected to the B bus.

"DEST" represents one or more registers that can be connected to the C bus.

(Note – this table omits the "NOT" operations shown in the textbook – they are actually not required for the microinstructions we're going to discuss).

DEST = H
DEST = SOURCE
DEST = H + SOURCE
DEST = H + SOURCE + 1
DEST = H + 1
DEST = SOURCE + 1
DEST = SOURCE – H
DEST = SOURCE – 1
DEST = -H
DEST = H AND SOURCE
DEST = H OR SOURCE
DEST = 0
DEST = 1
DEST = -1

MAL - Shifter Operation

The data path operation can also include syntax which controls how the shifter will operate. There are two possibilities:

SLL8 – The "Shift Left Logical 8" function is indicated by including "<< 8" at the end of the data path operation. For example:

MDR = H OR TOS << 8

This performs the "H OR TOS" operation first and then shifts the result 8 bits to the left.

SRA1 – The "Shift Right Arithmetic 1" function is indicated by including ">>1" at the end of the data path operation. For example:

TOS = MDR + 1 >> 1

This performs the "MDR+1" operation first and then shifts the result 1 bit to the right.

(Note – the SRA1 function is not actually used anywhere within the microcode that we'll be studying)

MAL – Memory Operation

The MAL statement can optionally include a memory operation by including one of the following:

fetch – **Activate the "fetch" signal which will read the Method Area byte that the PC points to into the MBR register. A microinstruction that changes the PC value can include this action – the byte fetched is the one that is at the address corresponding to the new PC value. The fetched byte will not be available for use until after the next microinstruction.**

rd – **Activate the "rd" signal which will read a data word that the MAR points to into the MDR register. A microinstruction that changes the MAR value can include this action – the word fetched is the one that is at the address corresponding to the new MAR value. The fetched word will not be available for use until after the next microinstruction.**

wr – **Activate the "wr" signal which will write the data word in the MDR register to the data area of memory at the address that the MAR register points. A microinstruction that changes the MAR or MDR value can include this action – the data and address used are the ones corresponding to the new values.**

If a memory operation is included, it should be separated from the data operation with a semicolon. For example:

PC = PC + 1; fetch

MAL – Next Address

The MAL assembler normally fills in the "Next Address" field of every microinstruction with the address of the following microinstruction – this results in sequential operation of the microinstructions. But you can override the "next address" by including one of the following expressions in the MAL statement:

goto *label*

Sets the "next address" field of the microinstruction to point to the microinstruction at the given label. The named microinstruction will be executed next.

goto (MBR OR *value*)
goto (MBR)

Turns on the JMPC bit in the microinstruction so that the next microinstruction address will be ORed with the MBR register. If no "OR value" clause is given then the next microinstruction address in this microinstruction word is set to zero and the next microinstruction to be executed will be determined by the value in MBR.

if (*flag*) goto *label1*; else goto *label2*

"flag" can be "N" or "Z". Turns on the JAMN or JAMZ bit in the microinstruction and ensures that the microinstructions at "label1" and "label2" are hex 100 apart in the control store so that, depending on the value of the N or Z flag, one or the other will be executed following this microinstruction.

If next address syntax is included, it should be separated from the other parts of the MAL statement with a semicolon. For example:

PC = PC + 1; fetch; goto (MBR)

MAL – "IF (flag)" syntax

When you use the "IF (flag)" syntax in a microinstruction word, the result in the N or Z flag depends on what goes through the ALU. Sometimes you want to check to see if a value is (for example) zero, but you don't really want to actually store the result anywhere. In such a circumstance, you can include "Z = source" as the data path operation portion of the microinstruction. For example:

Z = TOS; if (Z) goto T; else goto F

The Z flag is a single bit, and you can't really put the contents of TOS into it. But what this statement is really saying is "send the TOS value through the ALU and don't store it anywhere". Even though the result isn't stored, the ALU's Z flag is still set to indicate if the TOS value was equal to zero or not.

In this example, the TOS register is connected to the B Bus and the ALU function is set to "B" so that the value passes through the ALU unchanged. But none of the registers in the C Bus field of the microinstruction is turned on. Even though a result travels up the C Bus, none of the registers load it, and therefore it is not stored anywhere.

Exercise 3 – MAL to Microinstruction

Which of the microinstructions below correspond to the MAL statement:

MDR = TOS; wr

(the "next address" field has been omitted)

ALU F0/F1 Codes

00 - A AND B
01 - A OR B
10 - NOT B
11 - A + B

B Bus Register Codes

0000 - MDR
0001 - PC
0010 - MBR
0011 - MBRU
0100 - SP
0101 - LV
0110 - CPP
0111 - TOS
1000 - OPC

A

JAMN	JAMZ	JMPC	SLL8	SRA1	F0	F1	ENA	ENB	INVA	INC	H	OPC	TOS	CPP	LV	SP	PC	MDR	MAR	WRITE	READ	FETCH	B Bus
0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0 1 1 1

B

JAMN	JAMZ	JMPC	SLL8	SRA1	F0	F1	ENA	ENB	INVA	INC	H	OPC	TOS	CPP	LV	SP	PC	MDR	MAR	WRITE	READ	FETCH	B Bus
0	0	0	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0 0 0 0

C

JAMN	JAMZ	JMPC	SLL8	SRA1	F0	F1	ENA	ENB	INVA	INC	H	OPC	TOS	CPP	LV	SP	PC	MDR	MAR	WRITE	READ	FETCH	B Bus
0	0	0	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	1	0	0	0 1 1 1

Exercise 4 – MAL to Microinstruction

Which of the microinstructions below correspond to the MAL statement:

MAR = LV + H; rd; goto iload 3

(the "next address" field has been omitted)

ALU F0/F1 Codes

0 0 - A AND B
0 1 - A OR B
1 0 - NOT B
1 1 - A + B

B Bus Register Codes

0 0 0 0 - MDR	0 1 0 0 - SP
0 0 0 1 - PC	0 1 0 1 - LV
0 0 1 0 - MBR	0 1 1 0 - CPP
0 0 1 1 - MBRU	0 1 1 1 - TOS
	1 0 0 0 - OPC

A

JAMN	JAMZ	JMPC	SLL8	SRA1	F0	F1	ENA	ENB	INVA	INC	H	OPC	TOS	CPP	LV	SP	PC	MDR	MAR	WRITE	READ	FETCH	B Bus
0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0 1 0 1

B

JAMN	JAMZ	JMPC	SLL8	SRA1	F0	F1	ENA	ENB	INVA	INC	H	OPC	TOS	CPP	LV	SP	PC	MDR	MAR	WRITE	READ	FETCH	B Bus
0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0 1 0 1

C

JAMN	JAMZ	JMPC	SLL8	SRA1	F0	F1	ENA	ENB	INVA	INC	H	OPC	TOS	CPP	LV	SP	PC	MDR	MAR	WRITE	READ	FETCH	B Bus
0	0	0	0	0	1	1	1	1	0	1	0	0	1	0	0	0	0	0	1	0	1	0	0 1 0 1

Exercise 5 – MAL to Microinstruction

Which of the microinstructions below correspond to the MAL statement:

MAR = SP = SP – 1
if (Z) goto T; else goto F

(“next address” has been omitted)

ALU F0/F1 Codes

0 0 - A AND B
0 1 - A OR B
1 0 - NOT B
1 1 - A + B

B Bus Register Codes

0 0 0 0 - MDR	0 1 0 0 - SP
0 0 0 1 - PC	0 1 0 1 - LV
0 0 1 0 - MBR	0 1 1 0 - CPP
0 0 1 1 - MBRU	0 1 1 1 - TOS
	1 0 0 0 - OPC

A

JAMN	JAMZ	JMPC	SLL8	SRA1	F0	F1	ENA	ENB	INVA	INC	H	OPC	TOS	CPP	LV	SP	PC	MDR	MAR	WRITE	READ	FETCH	B Bus
0	1	0	0	0	1	1	0	1	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0 1 0 0

B

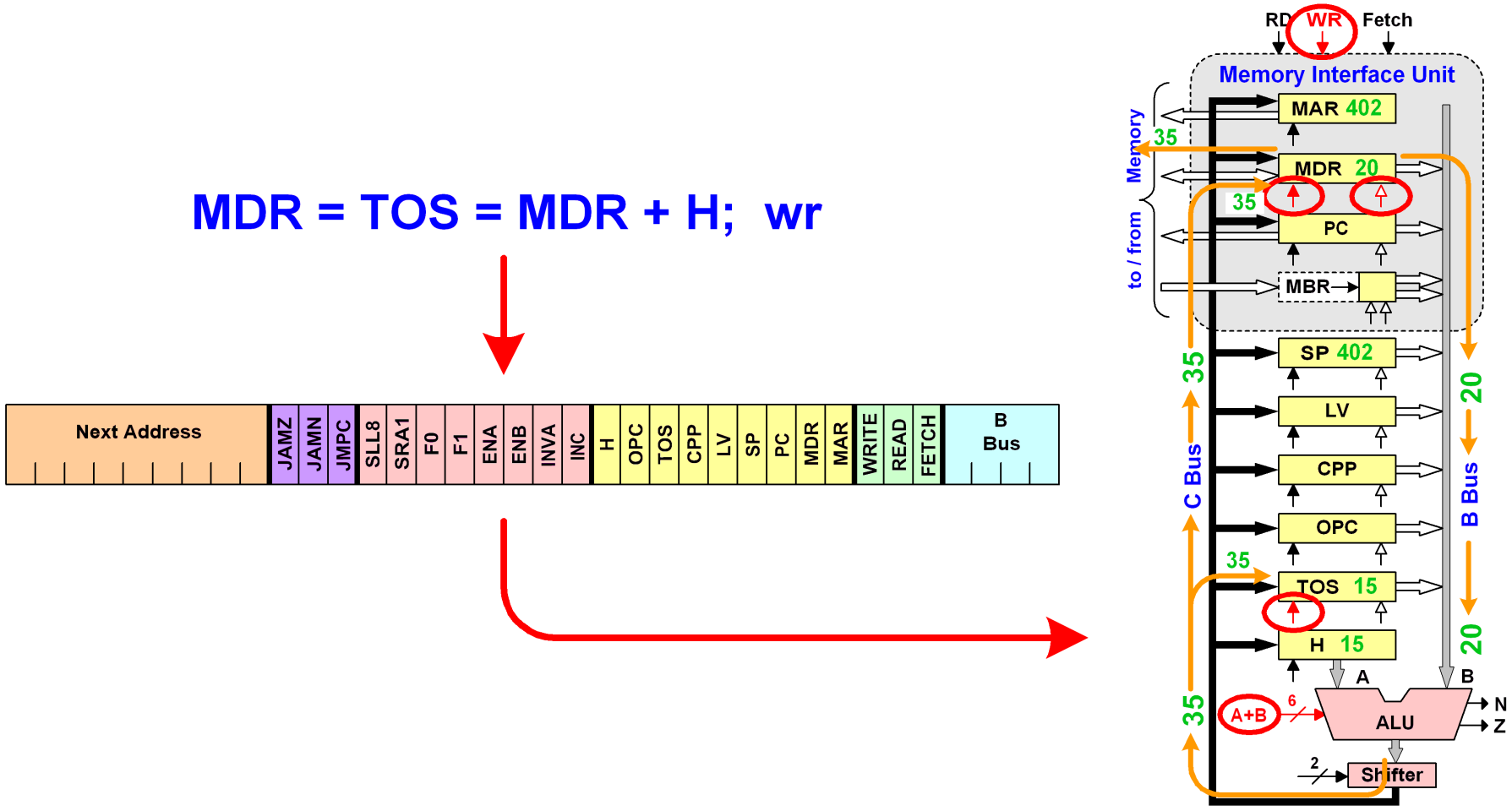
JAMN	JAMZ	JMPC	SLL8	SRA1	F0	F1	ENA	ENB	INVA	INC	H	OPC	TOS	CPP	LV	SP	PC	MDR	MAR	WRITE	READ	FETCH	B Bus
0	1	0	0	0	1	1	0	1	1	0	0	0	0	0	0	1	0	0	1	0	0	0	0 1 0 0

C

JAMN	JAMZ	JMPC	SLL8	SRA1	F0	F1	ENA	ENB	INVA	INC	H	OPC	TOS	CPP	LV	SP	PC	MDR	MAR	WRITE	READ	FETCH	B Bus
1	0	0	0	0	1	1	0	1	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0 1 0 0

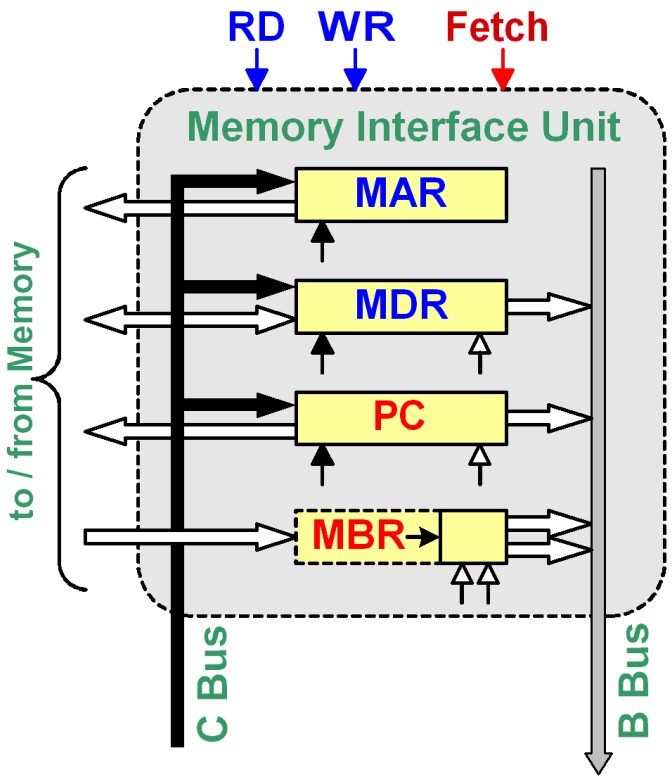
MIC-1 Microcode

Now that we've built the MIC-1 hardware, we're going to develop the microcode which will enable it to execute IJVM instructions. We'll write the microcode using MAL (Micro Assembler Language). As we do this, remember that MAL statements are simply a convenient way to express the bits in a microinstruction word, and that the microinstruction word in turn controls the data path of our microarchitecture:



Memory Control Unit Review

Also, keep in mind how the Memory Control Unit works:



To Read information from the Data area of memory:

1. **Put the address of the memory word to be read into the MAR register.**
2. **Issue the RD signal.**
3. **The requested word appears in the MDR register for use two clock cycles later**

To Write information to the Data area of memory:

1. **Put the address of the memory word to be written into the MAR register.**
2. **Put the data to be written into the MDR register.**
3. **Issue the WR signal**

To Read an instruction byte from the Method Area of memory:

1. **The PC register must contain the address of the instruction byte to be read.**
2. **Issue the Fetch signal.**
3. **The requested byte is put into the MBR register for use two clock cycles later.**

Register Usage

And finally, remember what the registers are used for:

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.

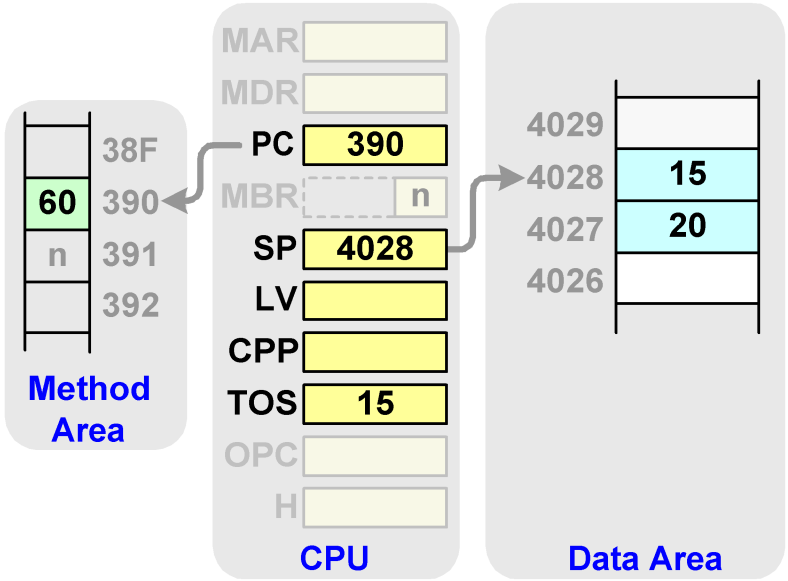
H This register supplies the value for the A input of the ALU, and is sometimes also used to hold temporary values.

And, very important:

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.

MIC-1 Microcode – Machine State

The MAR, MDR, MBR, OPC and H registers don't contain any permanent information that needs to be preserved from one IJVM instruction to another. These registers are only used for temporary storage of data while the microcode is doing the work that's required for a given IJVM instruction.



The other registers are important to the state of the machine and so it's important for the microcode to make sure they have the correct values after doing the work for each IJVM instruction. It's not essential for them to be correct all the time, just at the end of the processing for each IJVM instruction.

- **PC – points to the next opcode to be executed in the method area of memory**
- **SP – points to the word at the top of the stack in the data area of memory**
- **LV – points to the start of the local variables in the data area of memory**
- **CPP – points to the start of the constants in the data area of memory**
- **TOS – contains a copy of the word at the top of the stack**

When we say that a register "points" to memory, we mean that the number it contains is a memory address. For example the SP register above contains address 4028, which is the address of the word in memory where the top of stack is located.

Note that the IADD and several other instructions don't use local variables or constants, and so for those instruction we don't need to worry about what the LV or CPP registers contain.

The Main Loop

The main interpretation loop has the job of decoding the IJVM instructions and deciding which set of microinstructions to execute. It consists of a single microinstruction:

Main1 PC = PC + 1; fetch; goto (MBR)

This microinstruction performs the following functions:

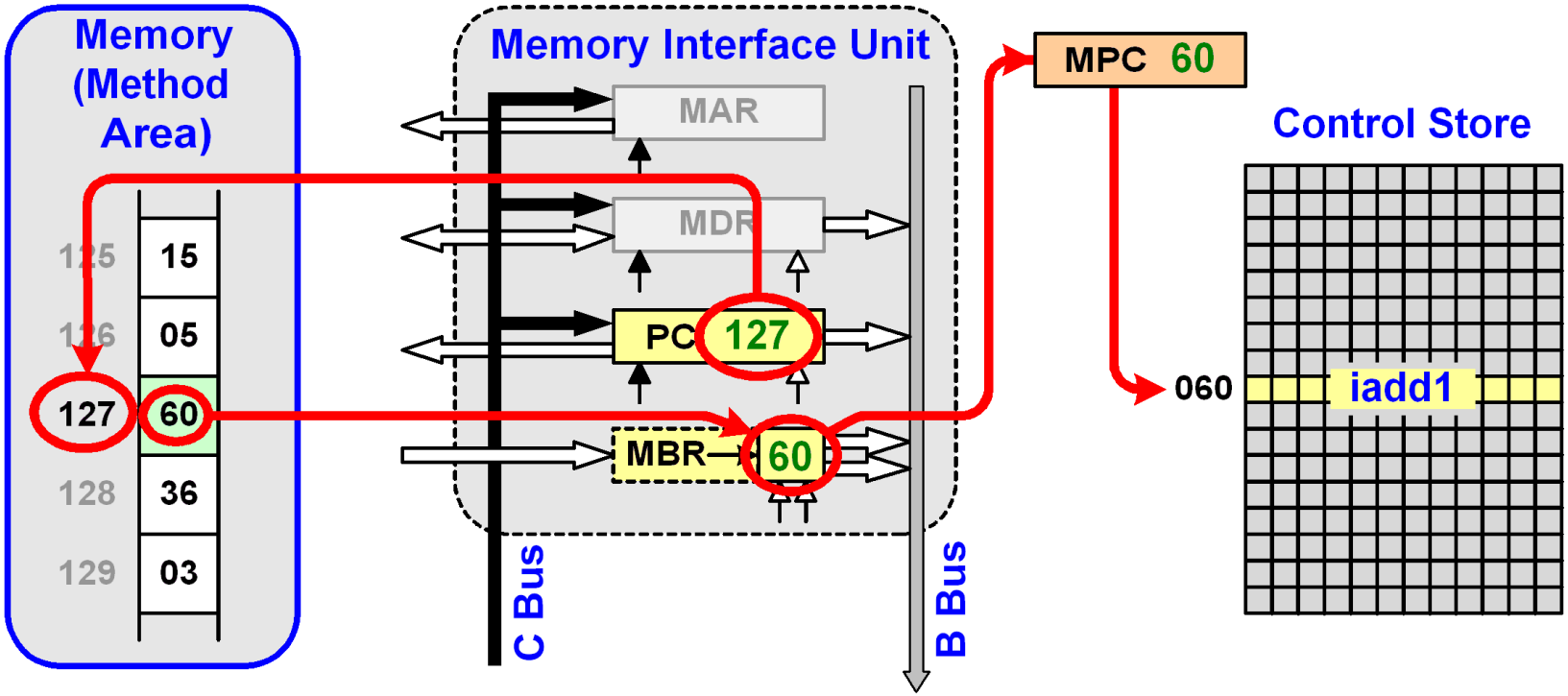
- **Increments the PC register so that it points to the next instruction byte in the method area**
- **Fetches the instruction byte into the MBR register. Note that the byte being fetched by this instruction won't actually reach the MBR register until the end of the next clock cycle.**
- **Selects the next microinstruction to execute based on the opcode byte that was previously fetched into the MBR register**

The Main Loop – Decoding the Instruction

The "goto (MBR)" portion of the main loop decodes the instruction (ie, this how the microarchitecture recognizes what instruction it is and how to execute it). Recall how "goto (MBR)" works:

- **Turns on the "JMPC" bit in the microinstruction word.**
- **The "JMPC" bit causes the "next microinstruction address" to be ORed with the MBR register.**
- **Because "goto (MBR)" doesn't specify it's own address, the MBR register is ORed with zero – in other words, it's used as is without change.**

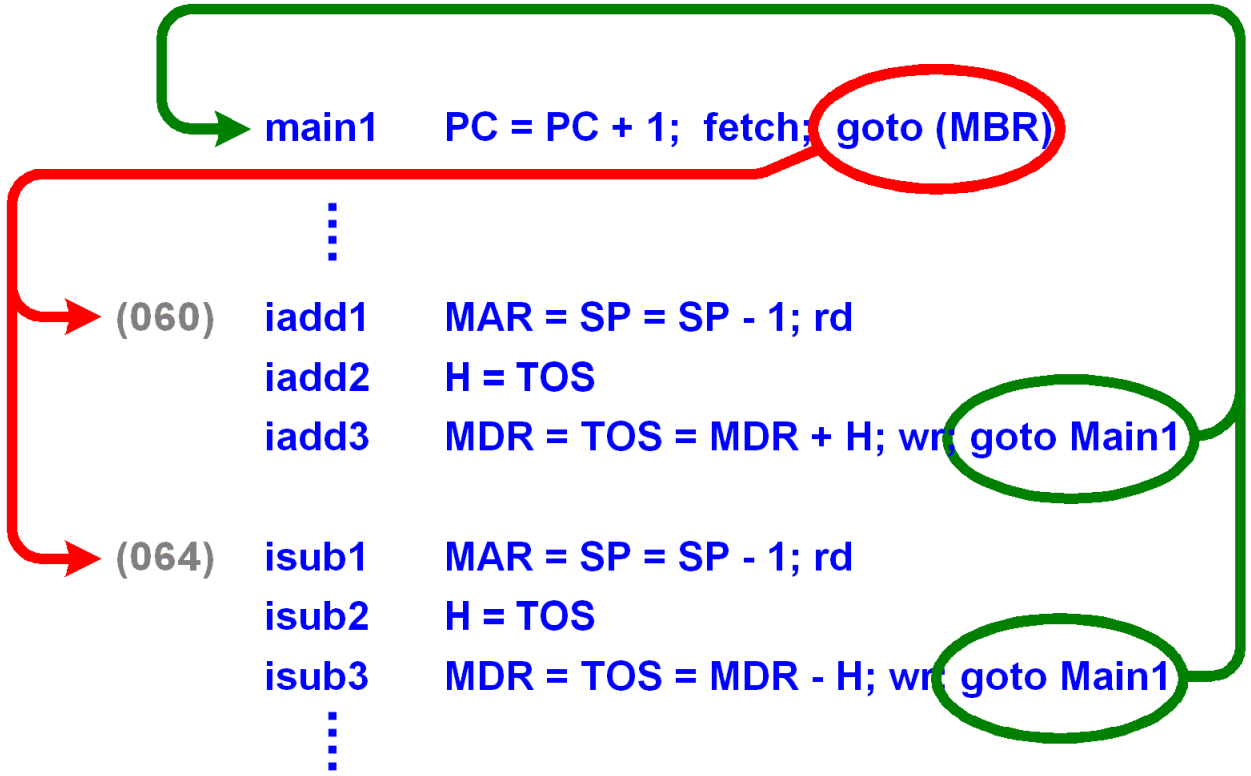
The effect is that the contents of the MBR register become the next microinstruction address:



The Main Loop

So the IJVM opcode controls which set of microinstructions is executed. The "main1" microinstruction jumps to the first of the sequence of microinstructions which do the work of each IJVM instruction.

In turn, the last microinstruction in each sequence jumps back to the "main1" microinstruction so that it can decode the next IJVM instruction.



The Main Loop

Note that the instruction being decoded is not the same one that is being fetched:

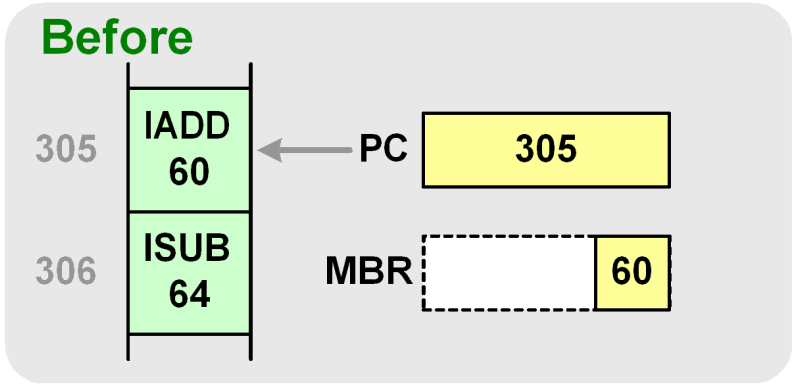
Main1 PC = PC + 1; fetch; goto (MBR)

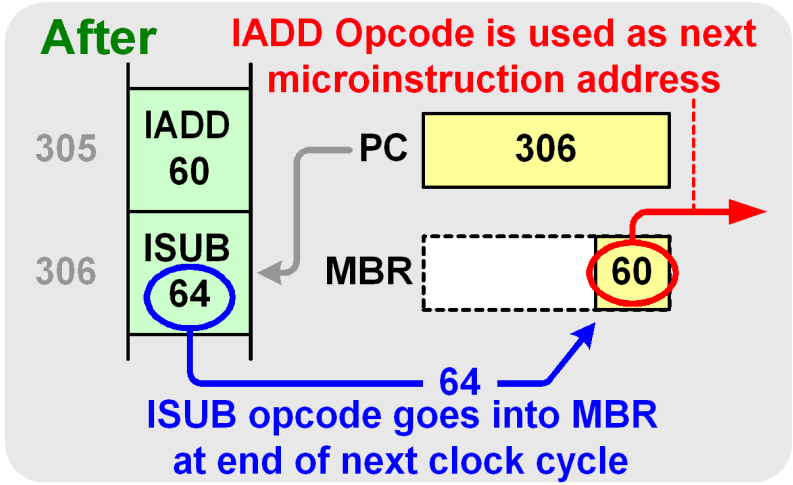
The "goto (MBR)" part of the microinstruction is using an opcode that was already fetched into the MBR register by a previous microinstruction. This microinstruction fetches a new opcode which will be decoded on the next execution of this microinstruction.

For example, let's assume that the PC register points to an IADD instruction and it's followed by an ISUB instruction.

The action of the main1 microinstruction is to:

- **Add 1 to the PC register so that it points to the ISUB opcode.**
- **Fetch the ISUB opcode. That opcode will not arrive in the MBR register until the end of the next clock cycle.**
- **Use the IADD opcode (hex 60) as the next microinstruction address. This means the next microinstruction to execute will be "iadd1"**





Exercise 6 – Main Loop

In the main loop microinstruction:

Main1 PC = PC + 1; fetch; goto (MBR)

1. **What is in the MBR register?**

   A – the address of the instruction being decoded
   B – the opcode of the instruction being decoded
   C – the operand of the instruction being decoded

2. **What is in the PC register?**

   A – the address of the instruction being decoded
   B – the opcode of the instruction being decoded
   C – the operand of the instruction being decoded

3. **What register is used by the memory control unit to perform the fetch request?**

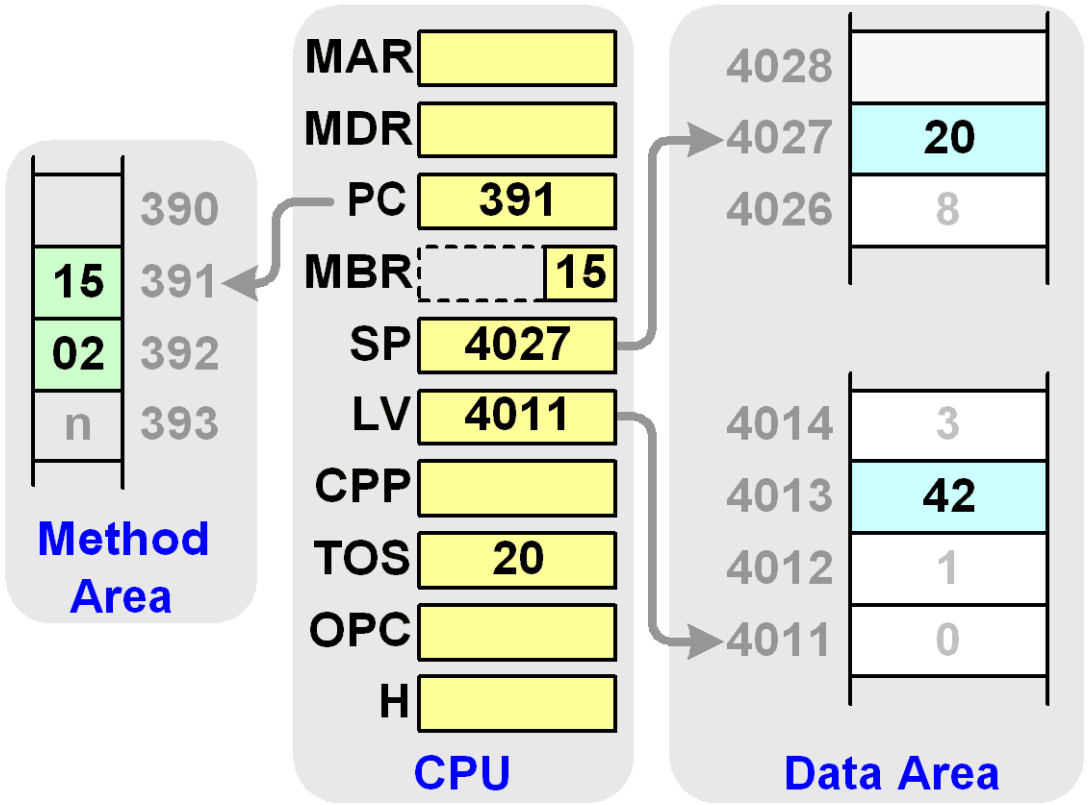
   A – the MDR register
   B – the MAR register
   C – the PC register

Showing How the Microcode Works

Now we are going to examine how the MIC-1 microcode works for each IJVM instruction. To do this, we're going to use simplified diagrams of the CPU and relevant memory areas as shown here.

Only the registers and their contents are shown in these diagrams, not the data path or control logic. But the execution of each microinstruction still causes data to flow through the buses, ALU and shifter and is still subject to the same restrictions inherent in the MIC-1 microarchitecture.

Registers whose contents are not known or whose values are not relevant to the IJVM instruction being discussed are left blank. In reality, all of the registers always contain *some* value, but if the register isn't used by a particular instruction then the diagram doesn't show the value.



Note that all of the numbers shown on these diagrams will be in hexadecimal.

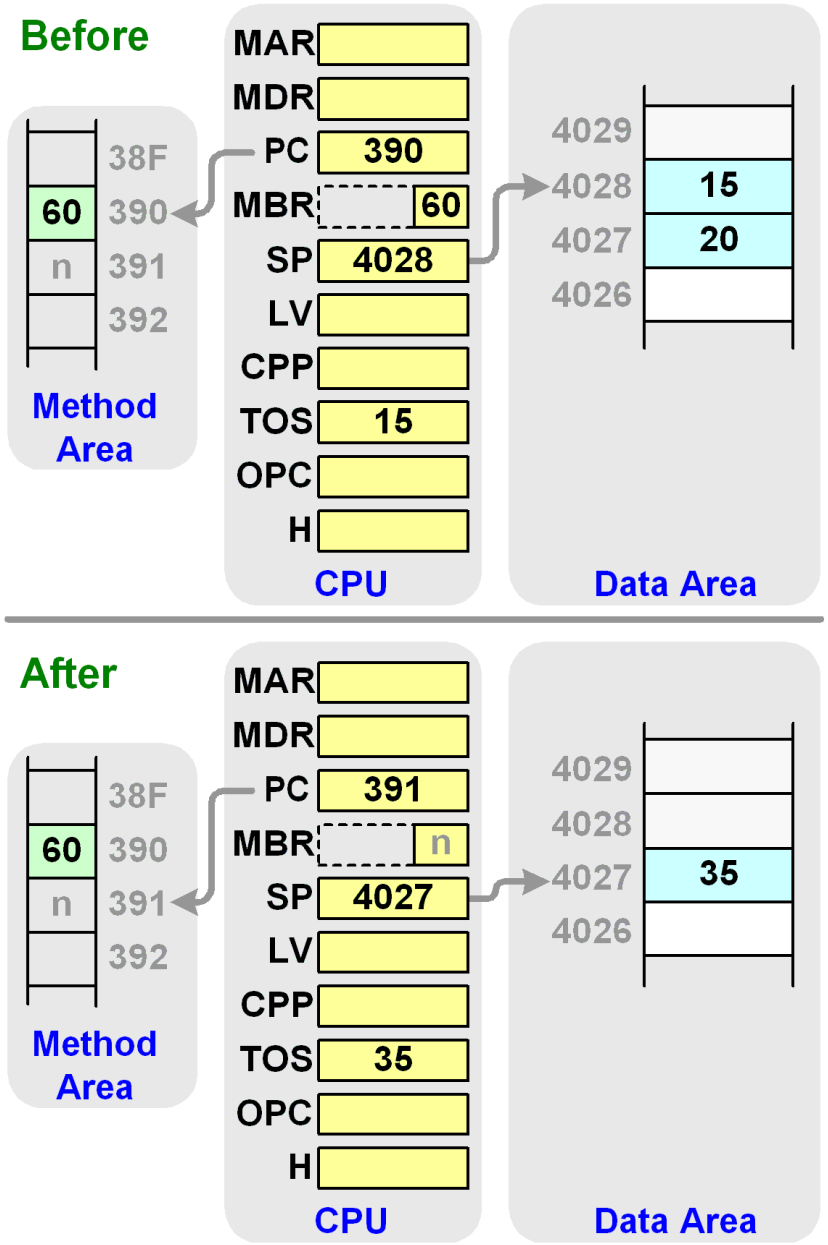
Microcode for the IADD Instruction

At the IJVM level, the IADD instruction does the following:

- **Pops two words off the stack**
- **Adds the two values together**
- **Pushes the result back onto the stack**

This means that the microinstructions for IADD must do the following work:

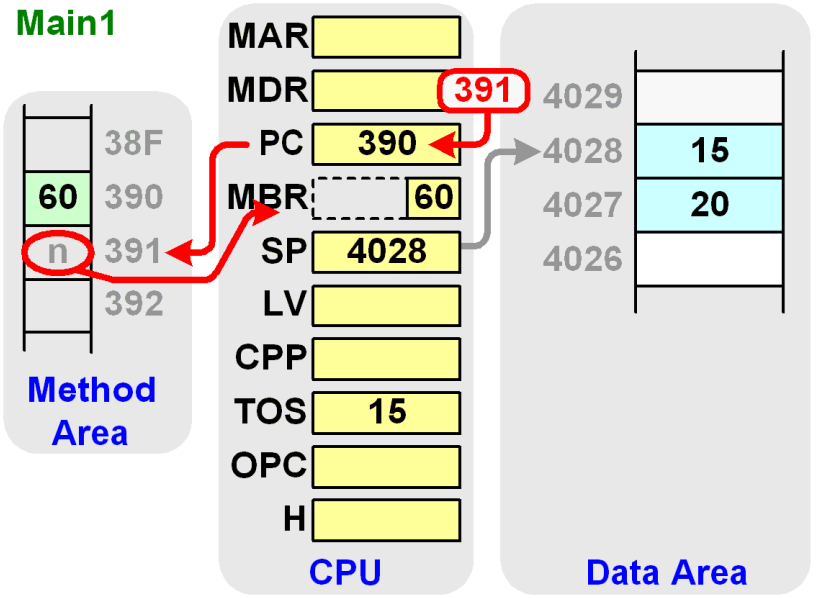
- **Subtract one from the stack pointer (changing from 4028 to 4027 in the example at right)**
- **Read the 2nd operand (20) from memory using the new SP register address (note that the microprogram doesn't have to read the 1st operand (15) from the top of the stack because a copy of it is already in the TOS register)**
- **Add the read 2nd operand to the TOS register**
- **Store the result (35) in the TOS register and also write it to memory at the SP register address.**
- **Point the PC to the opcode of the next IJVM instruction ("n" in the diagram) and fetch it into the MBR register**



Microcode for the IADD Instruction

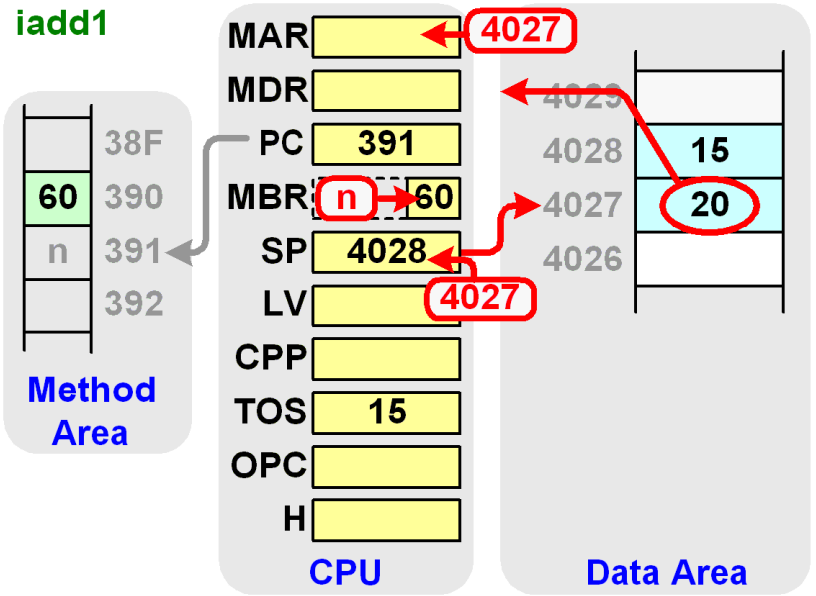
Main1 PC = PC + 1; fetch; goto (MBR)

- Increments the PC register (it changes from 390 to 391 in the example)
- Fetches the next instruction opcode ("n" in the diagram). The opcode is read but it doesn't arrive in the MBR register until the next clock cycle.
- The "goto (MBR)" transfers control to the "iadd1" microinstruction because the opcode is hex 60 and "iadd1" is stored at control store address hex 60.



iadd1 MAR = SP = SP – 1; rd

- Decrements the SP register (it changes from 4028 to 4027 in the example)
- Also stores this address in MAR so that it can read the 2nd IADD operand from that address.
- Initiates a read to the new SP address. The word is read but isn't placed into the MDR register until the next clock cycle.
- The next instruction opcode ("n") fetched in "Main1" is put into the MBR register.

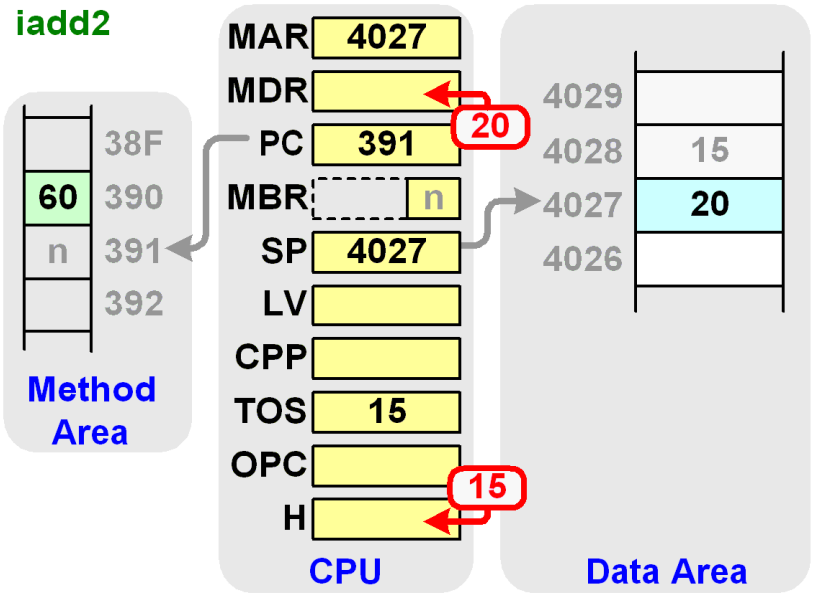


Microcode for the IADD Instruction

iadd2 H = TOS

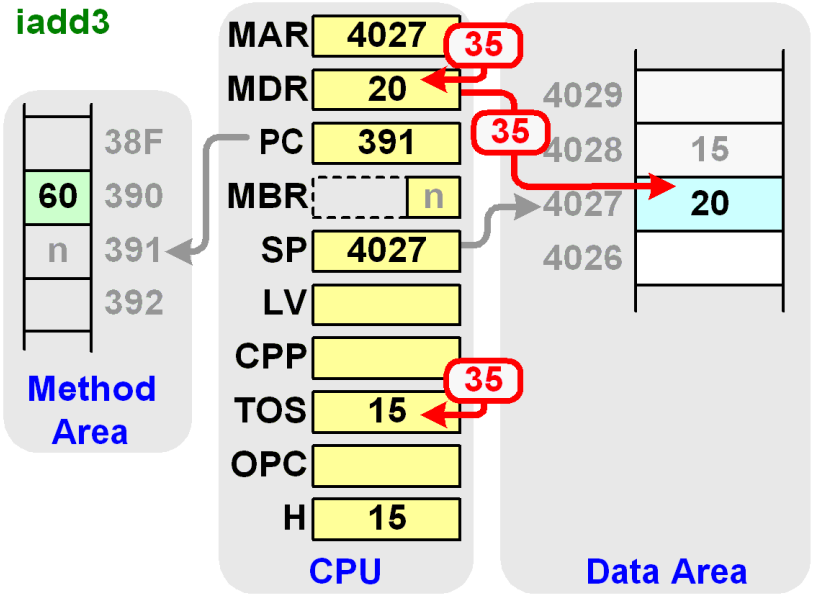
- Copies the old top-of-stack value (the 1st IADD operand) into the H register. This is one of the two data words that has to be added together.

  We move this into H because the only way to add two numbers is to have them go into the ALU's "A" and "B" inputs, and the "A" input comes from the H register.
- The IADD 2nd operand that was read from the stack in the last clock cycle is put into the MDR register at the end of this clock cycle.



iadd3 MDR = TOS =MDR + H; wr; goto Main1

- Adds the 1st and 2nd IADD operands (in MDR and H) together and stores the result (which will be the new top of stack) in TOS
- Also stores the result in MDR so it can be written to memory at the top-of-stack address
- Initiates a write. The data in MDR is written to the address in MAR. This writes the result to the stack.
- Goes to Main1 to decode and execute the next instruction ("n") in the MBR register.



Microcode for the IADD Instruction

Note carefully what the IADD instruction did:

- **It used the MAR and MDR registers to read and write move words to and from the data area of memory.**
  - **MAR and MDR must be used for every access to the data area of memory.**
- **It used the PC and MBR registers to fetch bytes from the instruction stream.**
  - **Each instruction byte must be accessed by putting it's address into the PC register and then fetching it into the MBR register.**
- **It placed a new value onto the top of the stack, and put a copy of the value into the TOS register.**
  - **Whenever the instruction results in a new top of stack value, a copy of the new value must be put into the TOS register.**
- **It left the PC register pointing at the opcode of the next instruction and the MBR register containing that opcode.**
  - **The sequence of microinstructions must always leave the PC and MBR registers set up so that the next instruction can be decoded and executed.**
- **The last thing it did was to branch back to the Main1 microinstruction so that the next IJVM instruction could be executed.**
  - **The sequence of microinstructions must always end by going back to "Main1"**

Microcode for the ISUB, IAND and IOR Instructions

The ISUB, IAND and IOR instructions are both very similar to IADD:

- **All have a 1-byte opcode (hex 64 for ISUB, 7E for IAND, 80 for IOR)**
- **All pop two values off the stack**
- **ISUB subtracts the old top-of-stack value from the value below it**
- **IOR does a Boolean OR operation on the two values, while IAND performs a Boolean AND**
- **The result is pushed back onto the stack**

The microcode for these instructions is identical to IADD and ISUB, except that the third microinstruction performs the appropriate operation:

```
iadd3    MDR = TOS = MDR + H;  wr;  goto Main1
isub3    MRD = TOS = MDR – H;  wr;  goto Main1
iand3    MDR = TOS = MDR AND H;  wr;  goto Main1
ior3     MDR = TOS = MDR OR H;  wr;  goto Main1
```

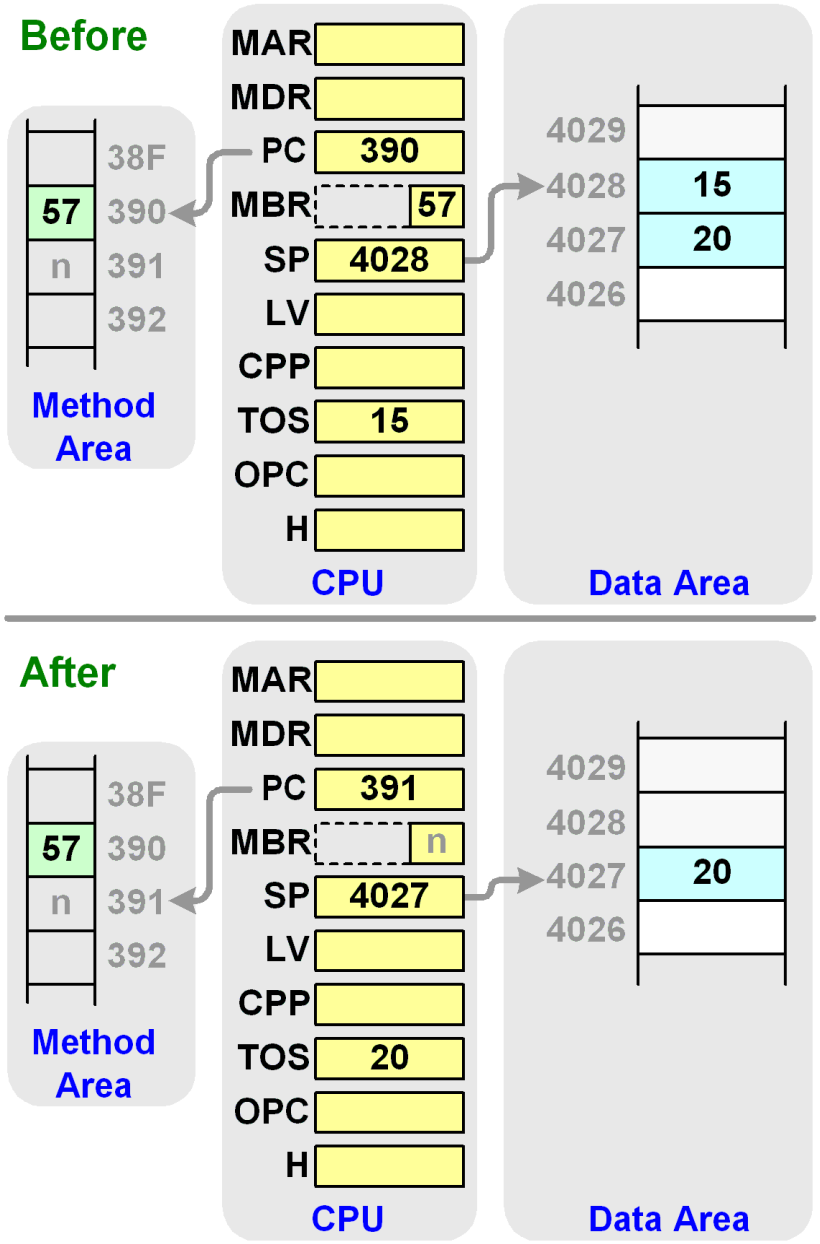
Microcode for the POP Instruction

At the IJVM level, the POP instruction does the following:

- **Subtracts one from the stack pointer. This has the effect of discarding the top word from the stack without doing anything with it.**

This means that the microinstructions for POP must do the following work:

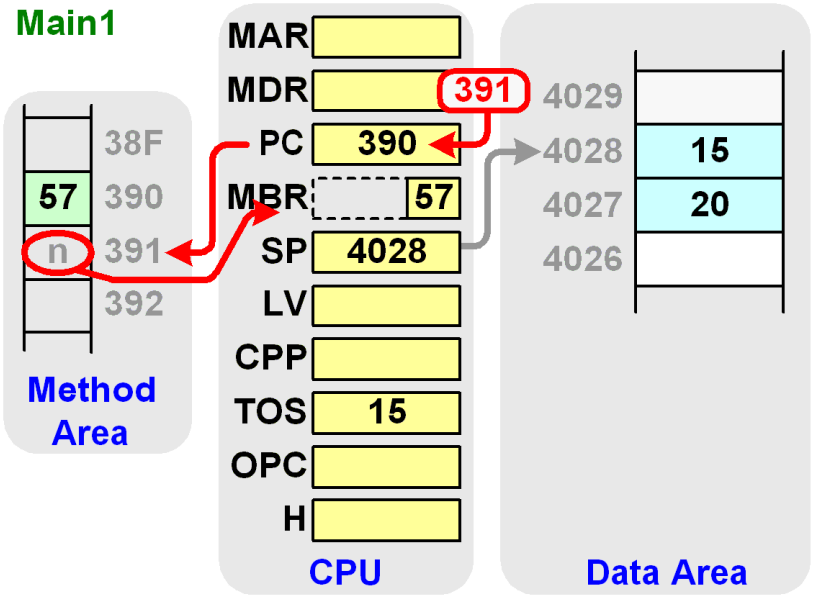
- **Subtract one from the stack pointer (changing from 4028 to 4027 in the example at right)**
- **Read the new top of stack value (20 in the example) into the TOS register.**
- **Point the PC to the opcode of the next IJVM instruction ("n" in the diagram) and fetch it into the MBR register.**



Microcode for the POP Instruction

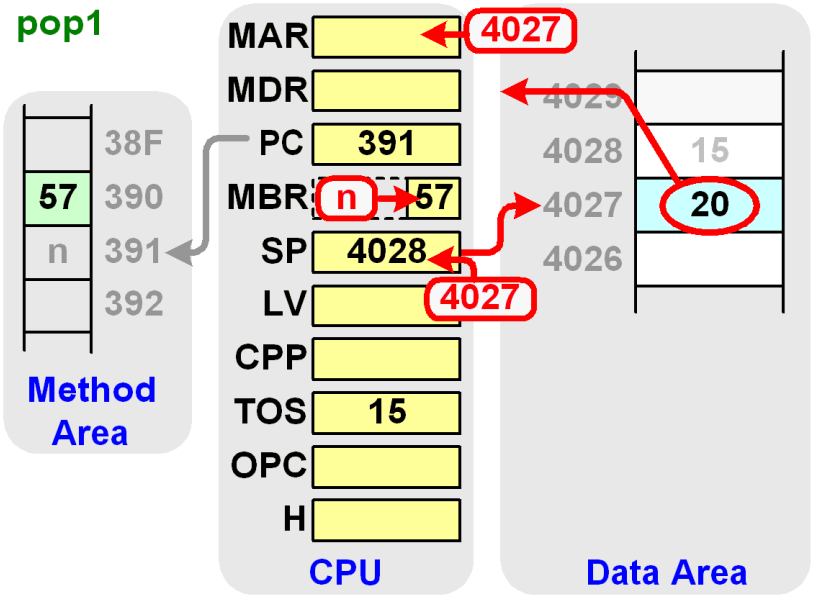
Main1 PC = PC + 1; fetch; goto (MBR)

- Increments the PC register (it changes from 390 to 391 in the example)
- Fetches the next instruction opcode ("n" in the diagram). The opcode is read but it doesn't arrive in the MBR register until the next clock cycle.
- The "goto (MBR)" transfers control to the "pop1" microinstruction because the opcode is hex 57 and "pop1" is stored at control store address hex 57.



pop1 MAR = SP = SP – 1; rd

- Decrements the SP register (it changes from 4028 to 4027 in the example)
- Also puts this value into MAR because we'll need to read the new top of stack value from this address
- Initiates a read to the new SP address. The word is read but isn't placed into the MDR register until the next clock cycle.
- The next instruction opcode ("n") fetched in "Main1" is put into the MBR register.



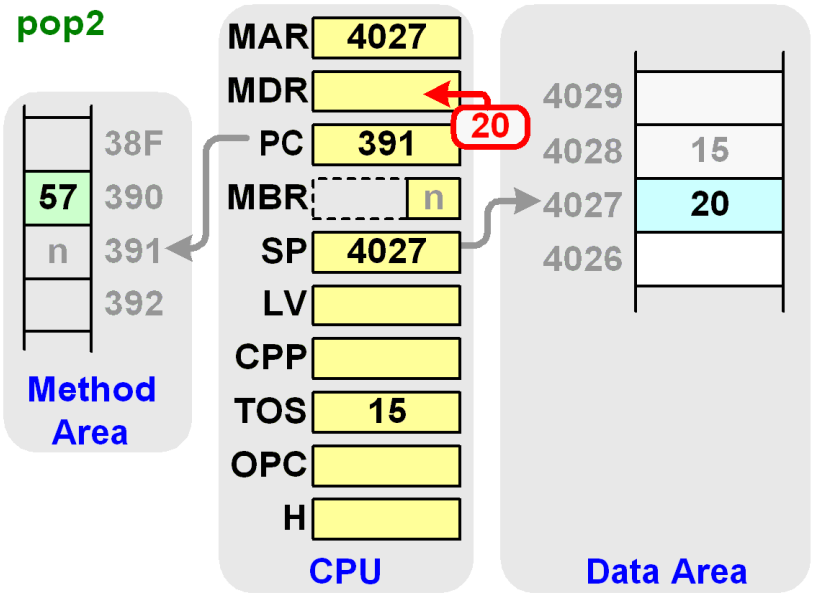
Microcode for the POP Instruction

pop2

- This microinstruction doesn't do anything. We need to wait until the end of this clock cycle before the new top-of-stack value read in "pop1" is available for use in the MDR register.

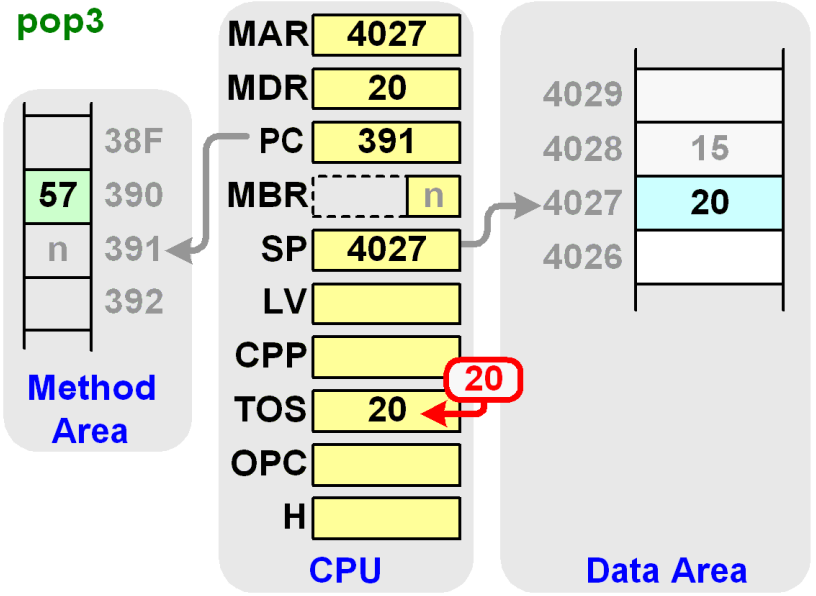
  The microinstruction word can specify any operation as long as it doesn't specify that any registers are loaded from the C Bus result

- The new top-of-stack value that was read from the stack in the last clock cycle is put into the MDR register at the end of this clock cycle.



pop3 TOS = MDR; goto Main1

- Copies the new top-of-stack value that was read from memory into the TOS register
- Goes to Main1 to decode and execute the next instruction ("n") in the MBR register.



Exercise 7 – POP Microcode

In the microcode for the POP microinstruction:

```
Main1    PC = PC + 1; fetch; goto (MBR)
pop1     MAR = SP = SP – 1;  rd
pop2
pop3     TOS = MDR;  goto Main1
```

Why do we need to set a new TOS value using TOS=MDR, even though we're not writing a new word onto the stack?

A – because the word read from the stack isn't available during the pop2 microinstruction cycle and so we can't perform the operation until the following cycle

B – because removing the top word from the stack reveals a different top-of-stack value which may be used in subsequent instructions

C – strictly speaking, it's unnecessary because the word popped from the top of the stack is discarded and not used

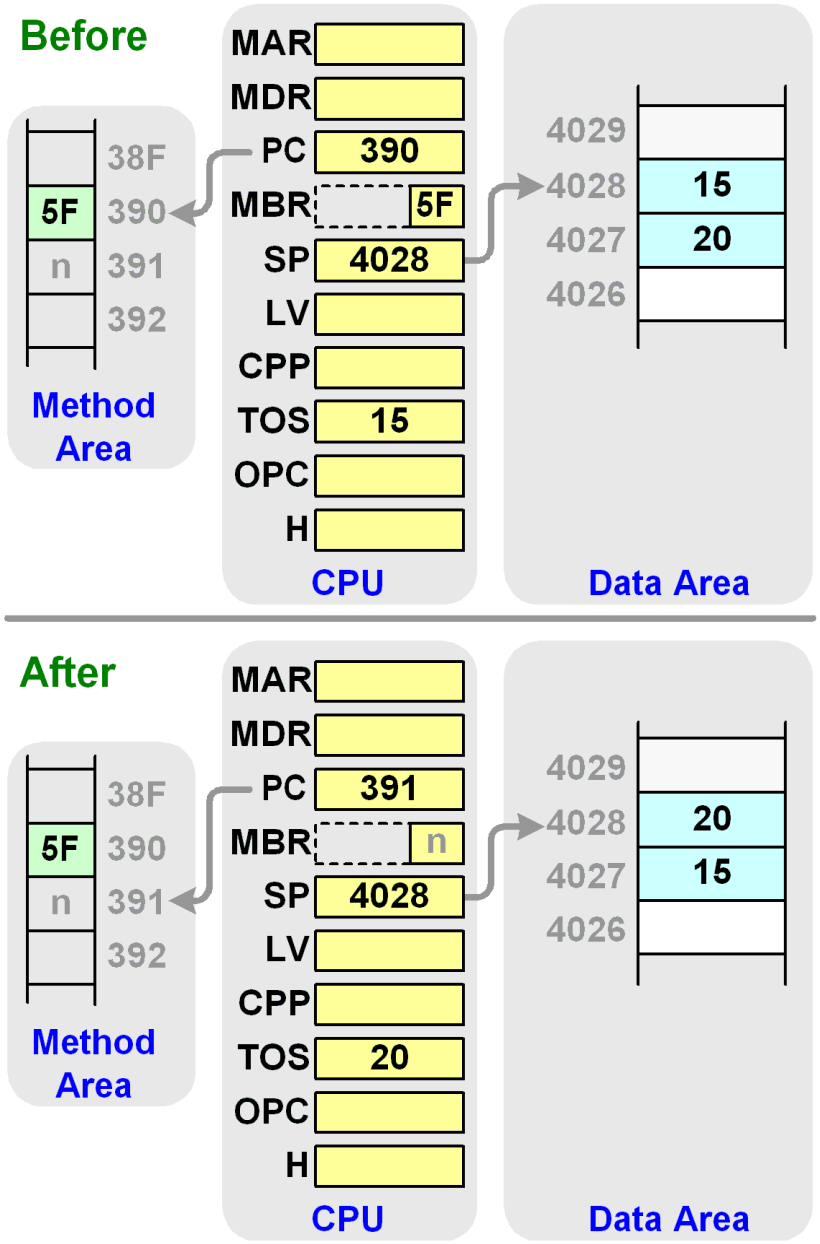
Microcode for the SWAP Instruction

At the IJVM level, the SWAP instruction does the following:

- **Reverses the position of the top two words on the stack. (This can be useful for instructions like ISUB in which the position of the values is important)**

This means that the microinstructions for SWAP must do the following work:

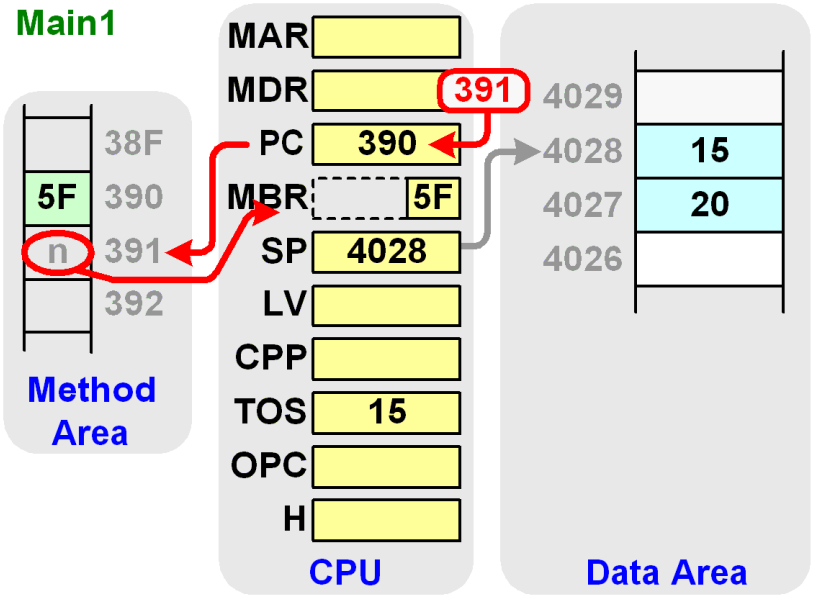
- **Read the word in memory at the address of the stack pointer less one.**
- **Write that word into memory at the address of the stack pointer – this is the new top-of-stack value. This value must also be saved somewhere so that it can be put into the TOS register when the instruction finishes.**
- **Write the TOS register value into memory at the address of the stack pointer less one.**
- **Copy the new top-of-stack value that was saved above into the TOS register.**
- **Point the PC to the opcode of the next IJVM instruction ("n" in the diagram) and fetch it into the MBR register.**



Microcode for the SWAP Instruction

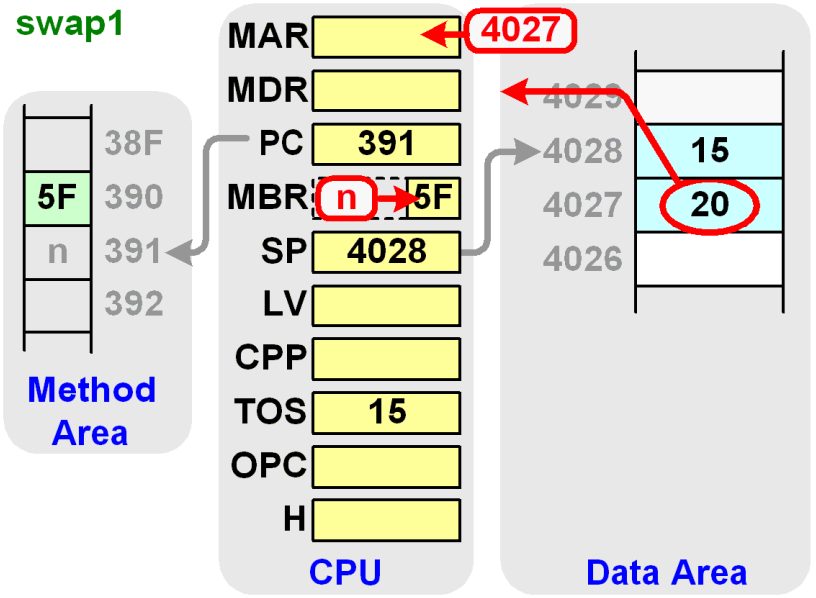
Main1 PC = PC + 1; fetch; goto (MBR)

- Increments the PC register (it changes from 390 to 391 in the example)
- Fetches the next instruction opcode ("n" in the diagram). The opcode is read but it doesn't arrive in the MBR register until the next clock cycle.
- The "goto (MBR)" transfers control to the "swap1" microinstruction because the opcode is hex 5F and "pop1" is stored at control store address hex 5F.



swap1 MAR = SP – 1; rd

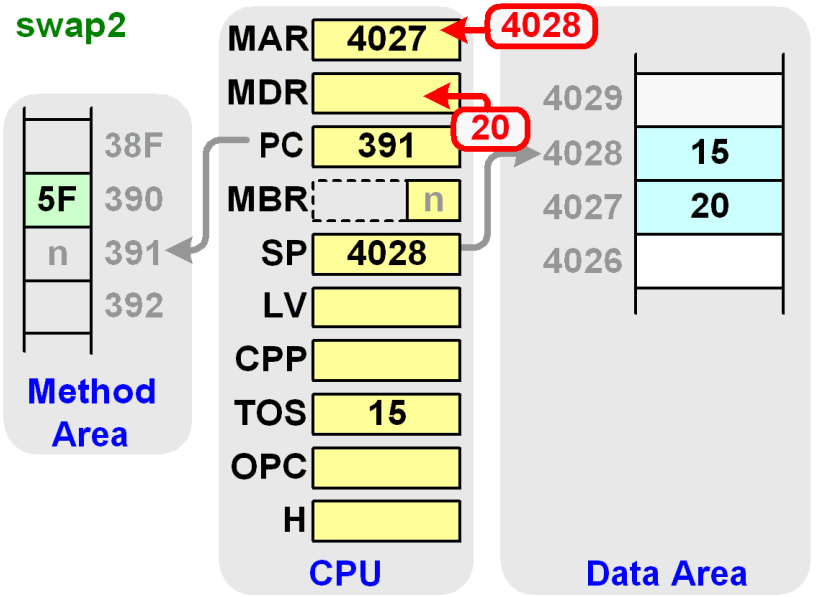
- Points the MAR register to the word just below the top-of-stack (points to word 20 at address 4027 in the example), Note that the SP register itself is not changed.
- Initiates a read to the word in memory just below the top-of-stack. The word is read but isn't placed into the MDR register until the next clock cycle.
- The next instruction opcode ("n") fetched in "Main1" is put into the MBR register.



Microcode for the SWAP Instruction

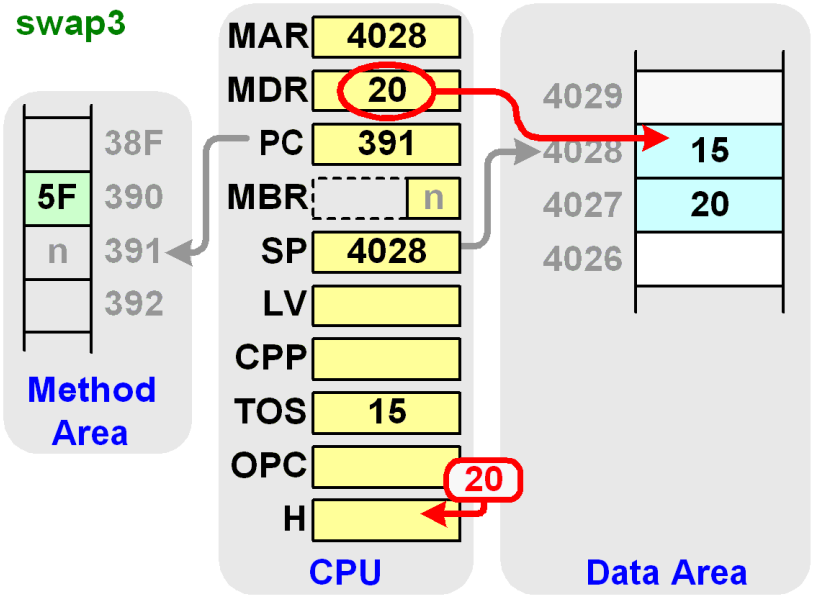
swap2 MAR = SP

- Sets the MAR register back to the SP address (4028). When the word below the top-of-stack (20) arrives in MDR we'll have to write it out to this address.
- The word below the top-of-stack (20) that was read from memory in the last clock cycle is put into the MDR register at the end of this clock cycle.



swap3 H = MDR; wr

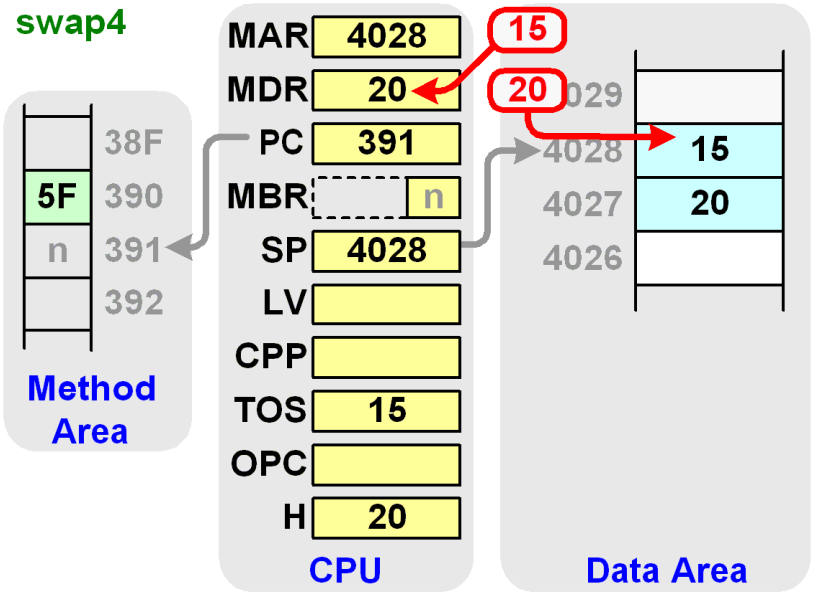
- Copies the new top-of-stack value (20) that was read from memory into the H register. This will eventually have to go into TOS, but we can't put it there until we've written the TOS value out to memory.
- A memory write is initiated. The new top-of-stack value (20) in MDR is written to the top-of-stack address (4028) in MAR. The word is actually written to memory in the next clock cycle.



Microcode for the SWAP Instruction

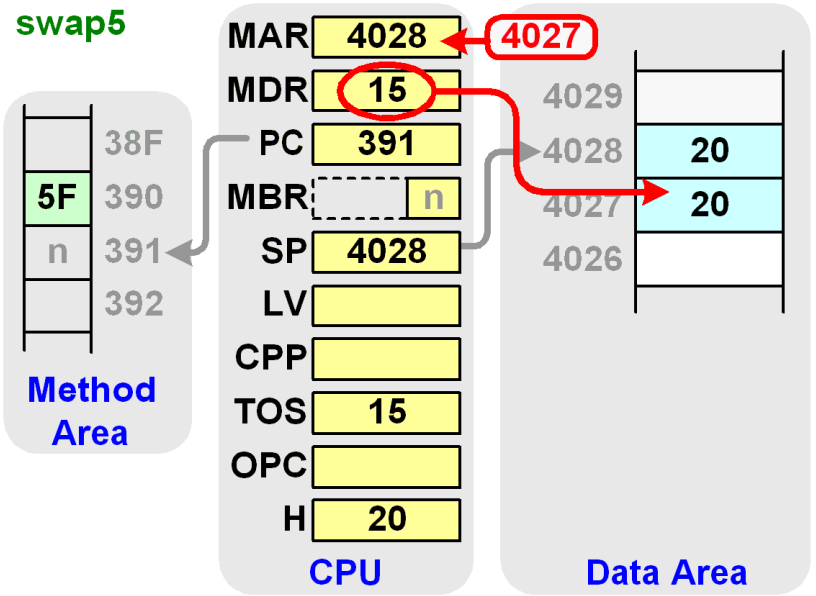
swap4 MDR = TOS

- The old TOS value (15) is put in MDR in preparation for writing it to the word just below the top-of-stack. We can't actually write the word until we also put the correct address into the MAR register.
- The new top-of-stack value that was written in the last clock cycle is actually stored in memory during this clock cycle.



swap5 MAR = SP – 1; wr

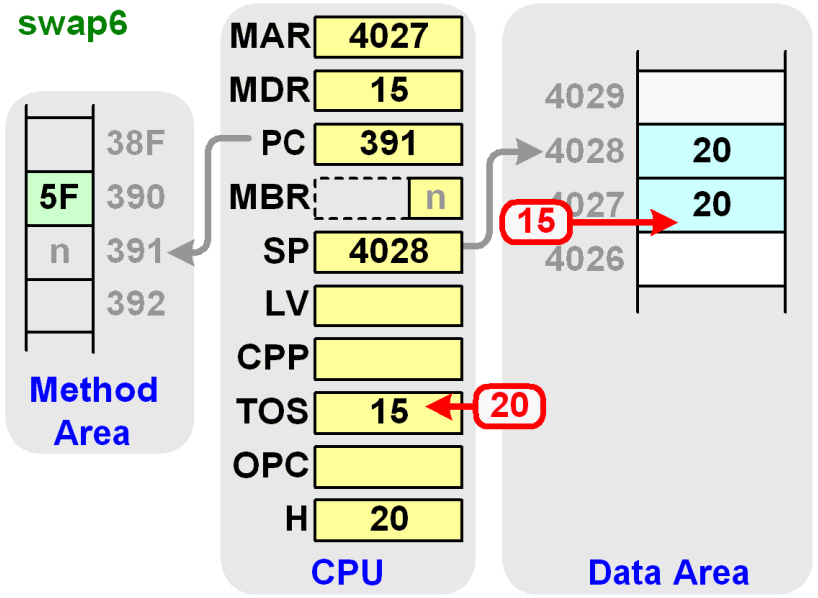
- Puts the address of the word just below the top-of-stack (address 4027) into the MAR register. Note that the SP register is not changed.
- Initiates a memory write operation. The word in the MDR register (15) is written to memory at the address specified by the MAR register (4027). The word is actually written to memory in the next clock cycle.



Microcode for the SWAP Instruction

swap6 TOS = H; goto Main1

- **Sets the TOS register to the new top-of-stack value that was saved in H by "swap3".**
- **Goes to Main1 to decode and execute the next instruction ("n") in the MBR register.**



Microcode for the BIPUSH Instruction

The BIPUSH instruction has a signed 8-bit operand which follows the opcode in the Method Area: (note that "0x10" means "hexadecimal 10")

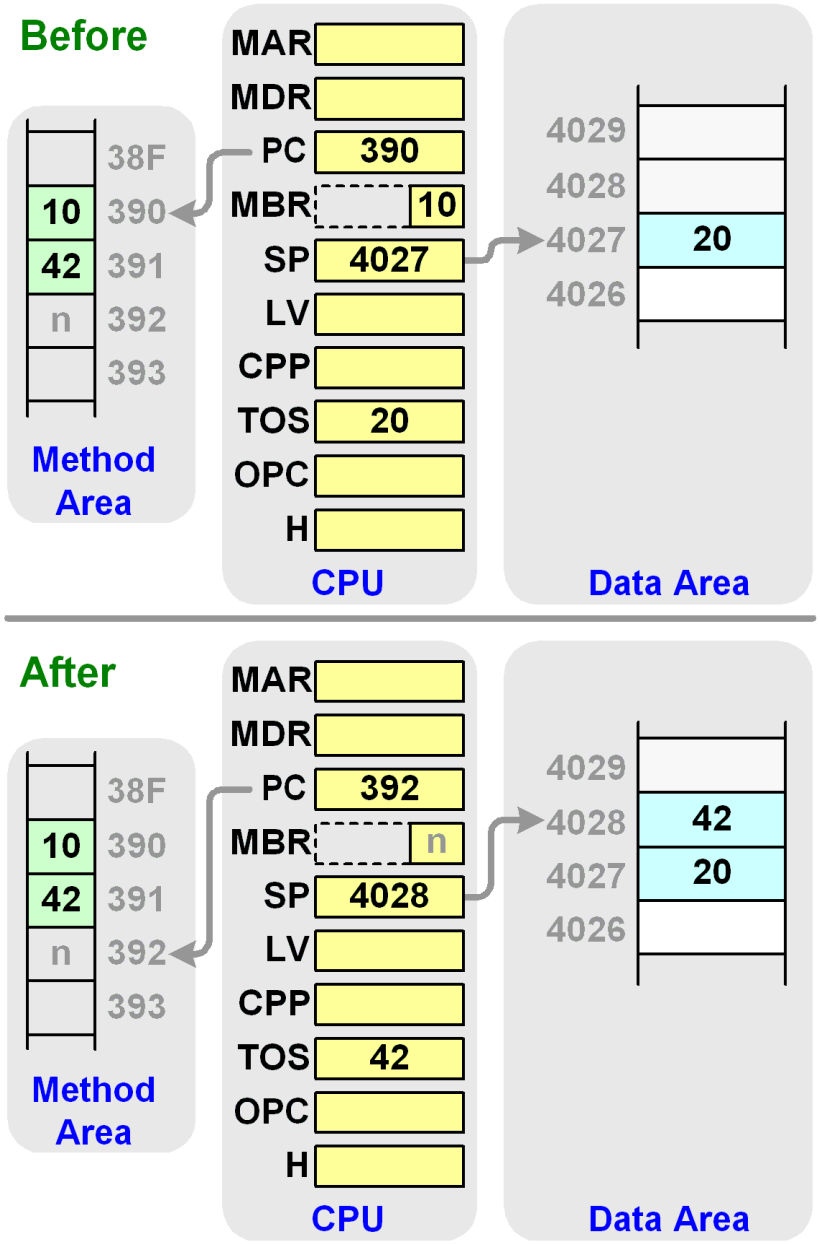
BIPUSH 0x10	value to be pushed

The BIPUSH instruction does the following:

- **Pushes the value specified in the operand onto the stack (this type of operand which contains the actual data to be used is called an "immediate" operand)**

This means that the microinstructions for BIPUSH must do the following work:

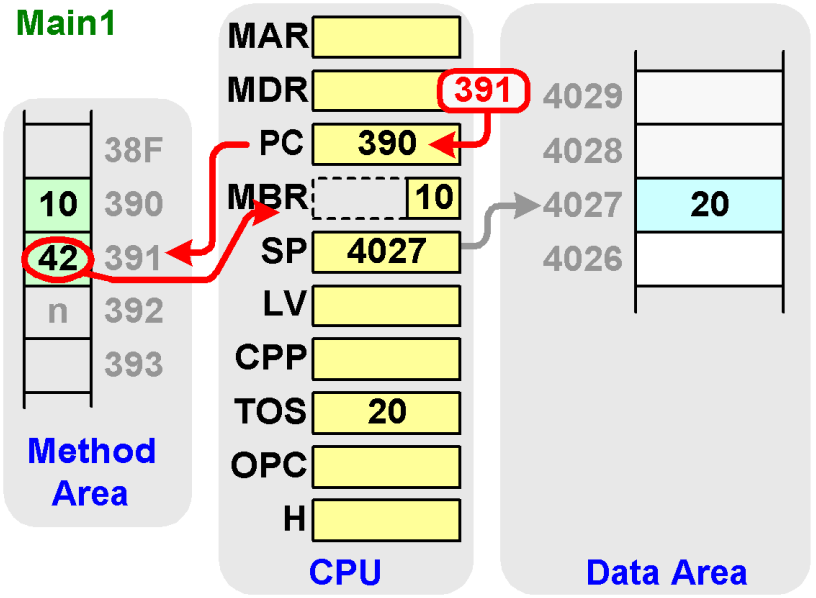
- **Add one to the stack pointer**
- **Add one to the PC and fetch the signed operand from the Method Area**
- **Write the signed operand to memory at the SP register address and also put a copy in TOS**
- **Add 1 to the PC again so that it points to the next IJVM instruction's opcode ("n" in the diagram) and fetch the opcode into MBR.**



Microcode for the BIPUSH Instruction

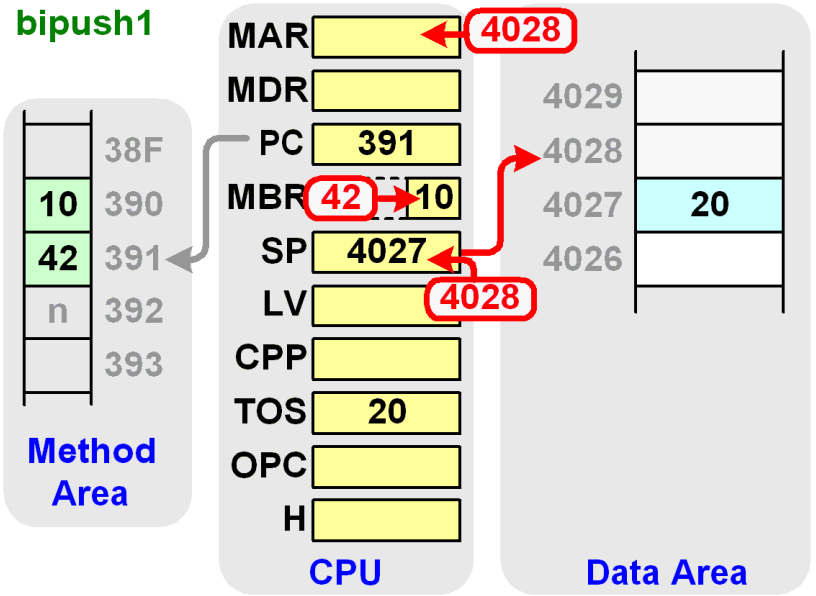
Main1 PC = PC + 1; fetch; goto (MBR)

- Increments the PC register (it changes from 390 to 391 in the example)
- Fetches the next byte – this is the operand (42) which will be pushed onto the stack. The operand is read but it doesn't arrive in the MBR register until the next clock cycle.
- The "goto (MBR)" transfers control to the "bipush1" microinstruction because the opcode is hex 10 and "bipush1" is stored at control store address hex 10.



bipush1 SP = MAR = SP + 1

- Increments the SP register (it changes from 4027 to 4028 in the example)
- Also puts this value into MAR because we're going to write the BIPUSH operand to that address.
- The BIPUSH operand (42) fetched in "Main1" is put into the MBR register.

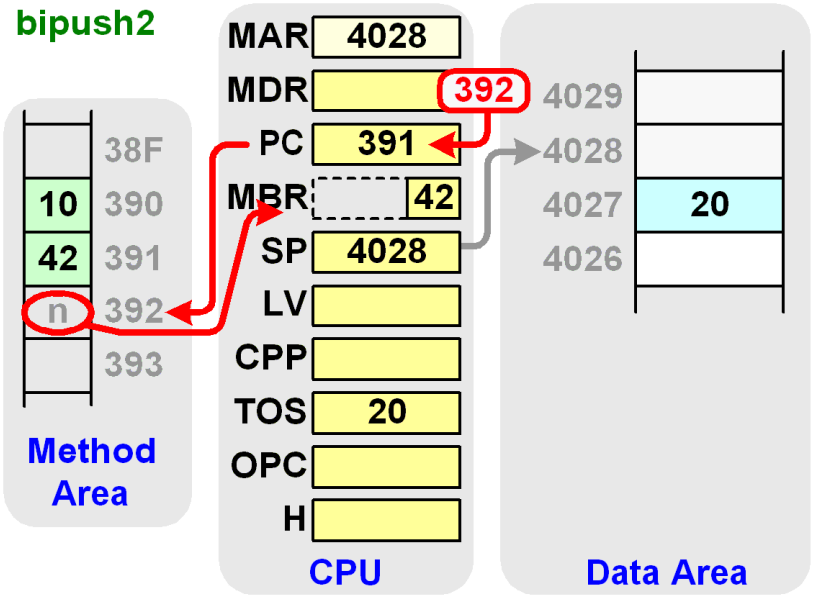


Microcode for the BIPUSH Instruction

bipush2 PC = PC + 1; fetch

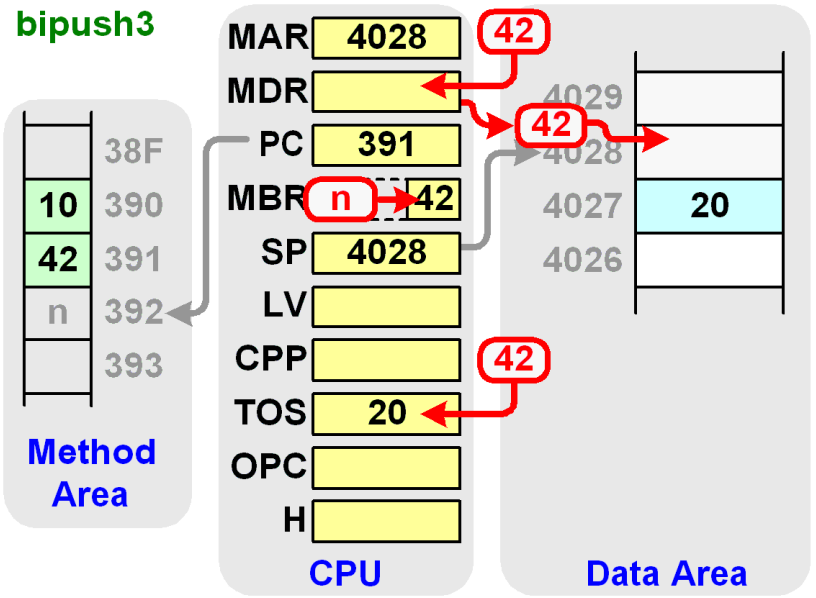
- Increments the PC so that it points to the opcode of the next IJVM instruction ("n" in the example).
- Fetches the next IJVM instruction's opcode into the MBR register. We have to do this so that the opcode is ready to be decoded when we go back to "Main1".

  The opcode is fetched by this microinstruction but it doesn't arrive in the MBR register until the next clock cycle.



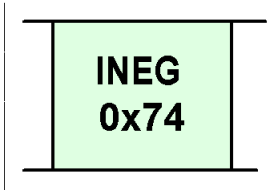
bipush3 MDR = TOS =MBR; wr; goto Main1

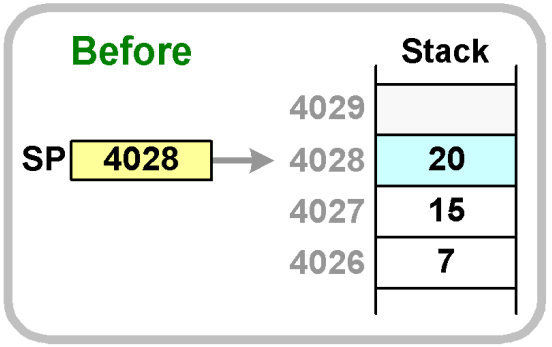
- Copies the operand into the TOS register and also into the MDR register so it can be written to memory. Note that "MBR" (not "MBRU") is used because the operand is a signed value.
- Initiates a write. The data in MDR is written to the address in MAR. This writes the operand to the stack.
- The next IJVM instruction opcode ("n") fetched in "bipush2" is put into the MBR register.
- Goes to Main1 to decode and execute the next instruction

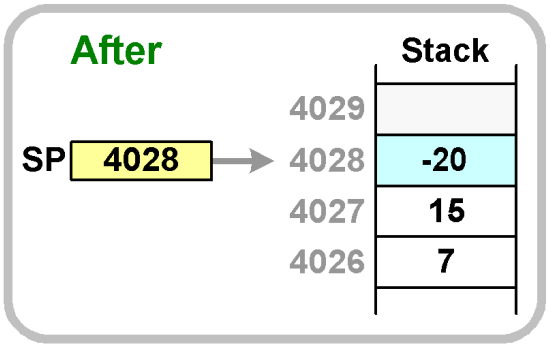


Exercise 8 – INEG Instruction

The full Java Virtual Machine includes an "INEG" instruction which reverses the sign of the word at the top of the stack.







To perform the work for this instruction, the microcode must:

- **Use the ALU to negate the value in the TOS register (i.e., perform the two's complement operation on it).**
- **Store the inverted value in the TOS register**
- **Write the inverted value to the word at the top of the stack.**

Which of the following sequences of microcode is correct?

A

```
H = TOS
MAR = SP
TOS = MDR = –H; wr;
                goto Main1
```

B

```
MAR = SP = SP – 1
TOS = –TOS; wr;
                goto Main1
```

C

```
MAR = SP
MDR = TOS = –TOS; wr
                goto Main1
```

Microcode for the ILOAD Instruction

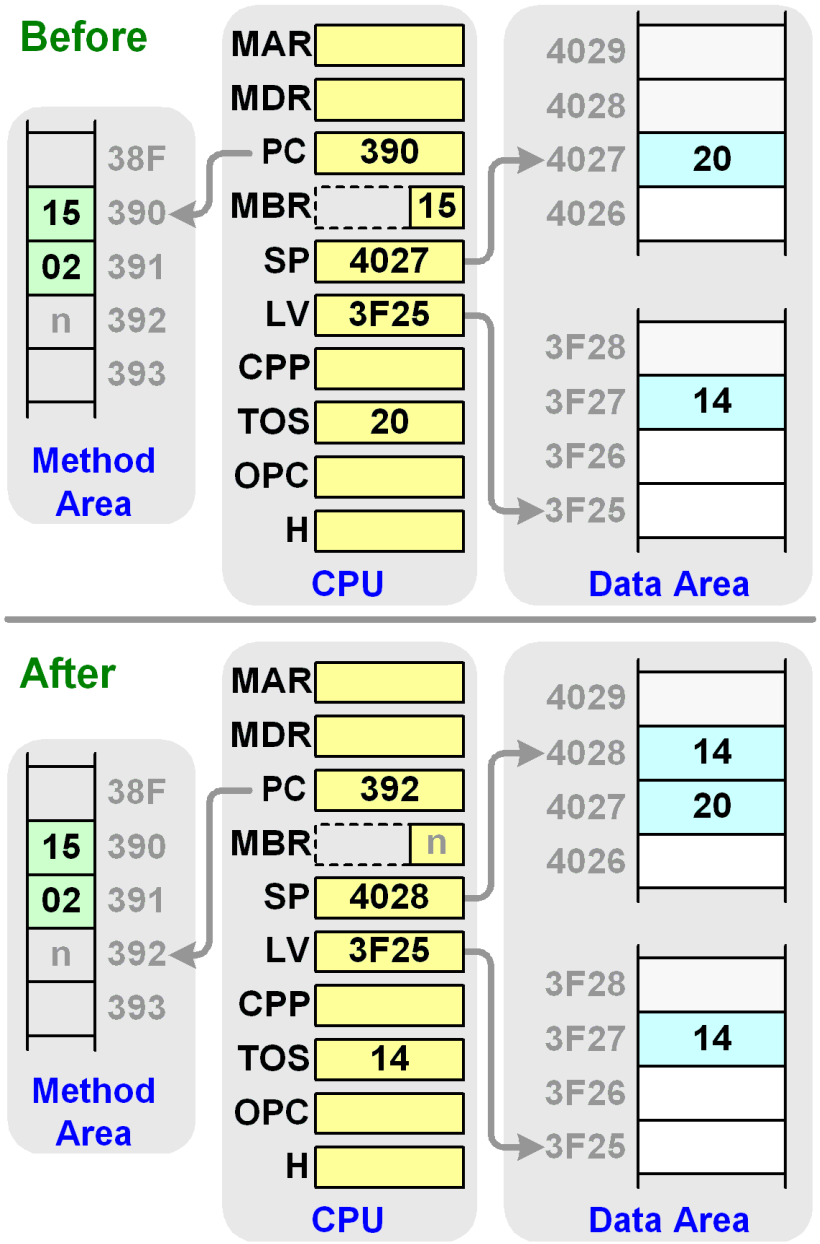
The ILOAD instruction includes an unsigned 8-bit operand after the opcode in the Method Area:

ILOAD 0x15	local variable #

ILOAD reads the given variable from the local variable frame and pushes it onto the stack.

This means that the microinstructions for ILOAD must do the following work:

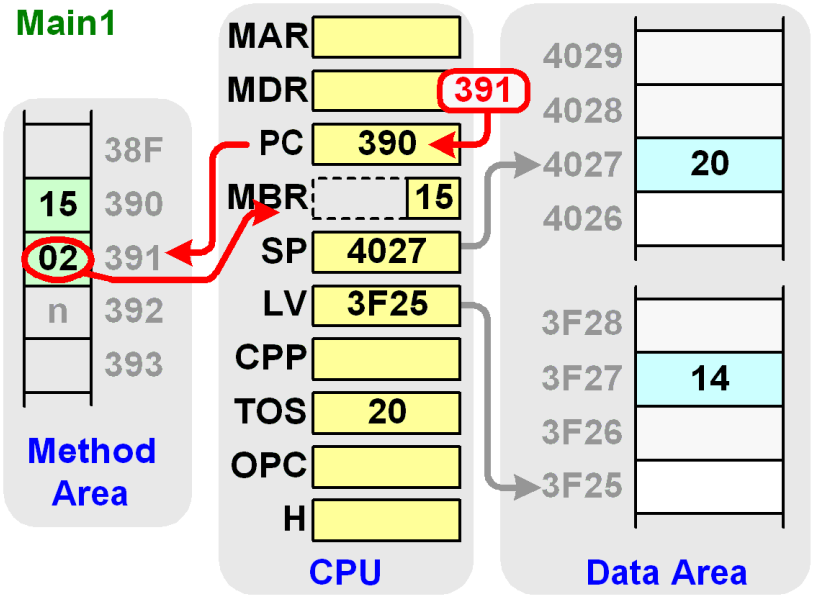
- **Add one to the stack pointer**
- **Add one to the PC and fetch the unsigned local variable number from the Method Area**
- **Calculate the address of the variable by adding the local variable number to the LV register**
- **Read the local variable from memory at the calculated address**
- **Write the variable to memory at the SP address and also put a copy in TOS**
- **Add 1 to the PC again so that it points to the next IJVM instruction's opcode ("n" in the diagram) and fetch the opcode into MBR.**



Microcode for the ILOAD Instruction

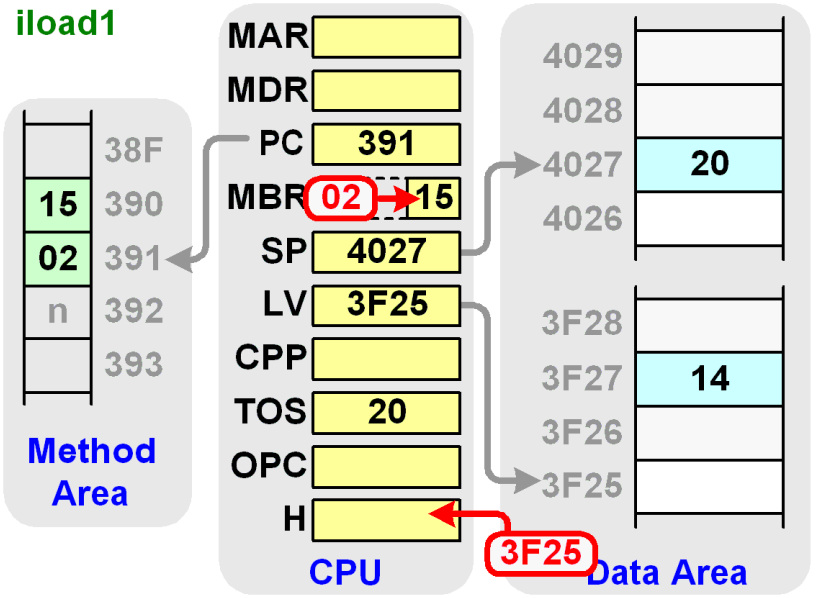
Main1 PC = PC + 1; fetch; goto (MBR)

- Increments the PC register (it changes from 390 to 391 in the example)
- Fetches the next byte – this is the local variable no. (2) which will be pushed onto the stack. The operand is read but it doesn't arrive in the MBR register until the next clock cycle.
- The "goto (MBR)" transfers control to the "iload1" microinstruction because the opcode is hex 15 and "iload1" is stored at control store address hex 15.



iload1 H = LV

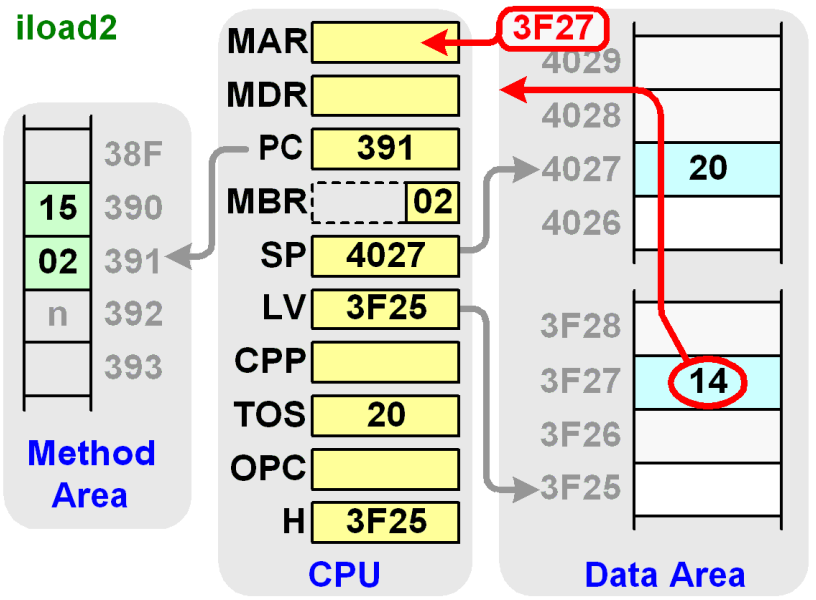
- Copies the LV pointer to the H register. This is to prepare for calculating the address of the local variable by adding it to the ILOAD operand. We have to copy LV to H because we need it to come into the "A" input of the ALU in order to add it.
- The ILOAD operand (local variable number 2) fetched in "Main1" is put into the MBR register.



Microcode for the ILOAD Instruction

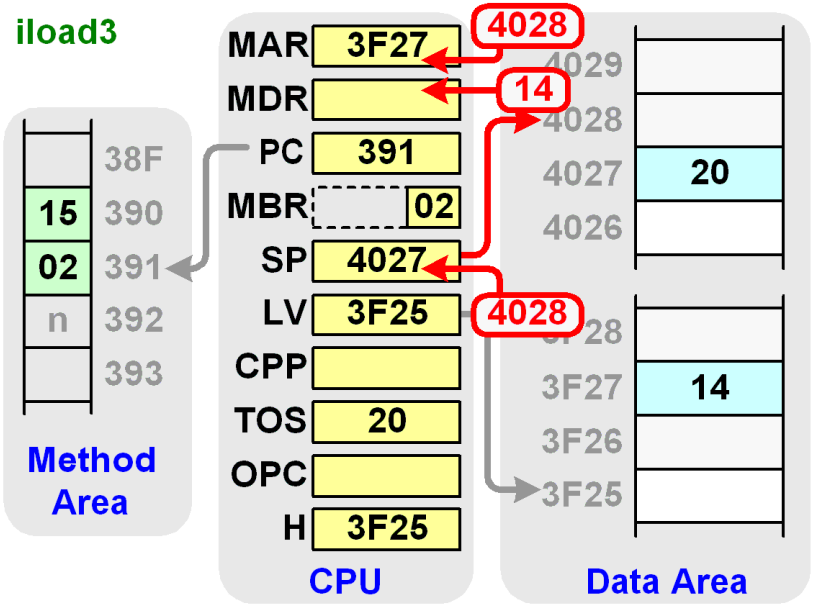
iload2 MAR = MBRU + H; rd

- Calculates the address of the local variable to be pushed onto the stack by adding the local variable number (2, in MBR) to the copy of the LV register in H (3F25).
- The calculated address (3F27) is put into MAR so that we can read the local variable from the data area of memory.
- A memory read is initiated. The local variable (at address 3F27 with a value of 14) will be loaded into the MDR register at the end of the next clock cycle.



iload3 MAR = SP = SP + 1

- Increments the SP register to point to the net top of stack address (4028).
- Also puts a copy of the same address into MAR because we'll have to write the local variable to that address in memory.
- The local variable (with a value of 14) that was read in "iload2" is put into the MDR register.

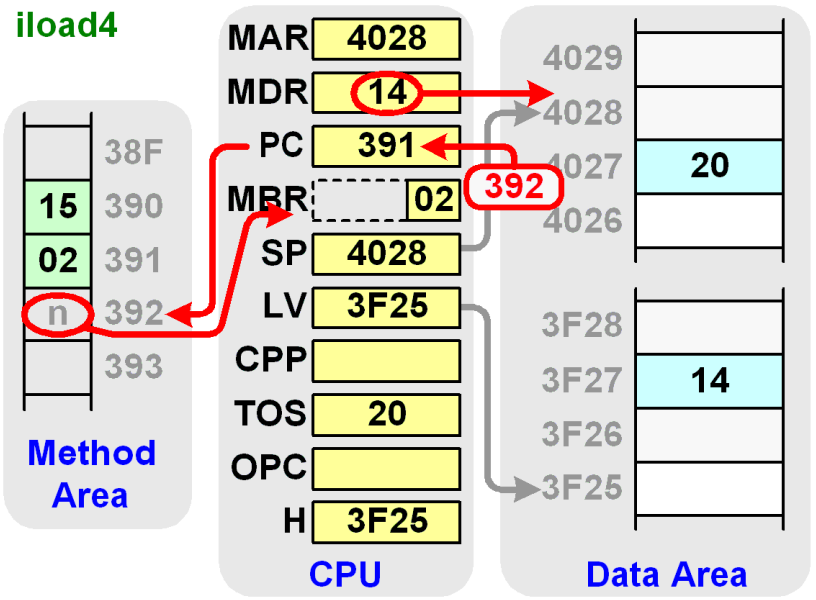


Microcode for the ILOAD Instruction

iload4 PC = PC + 1; fetch; wr

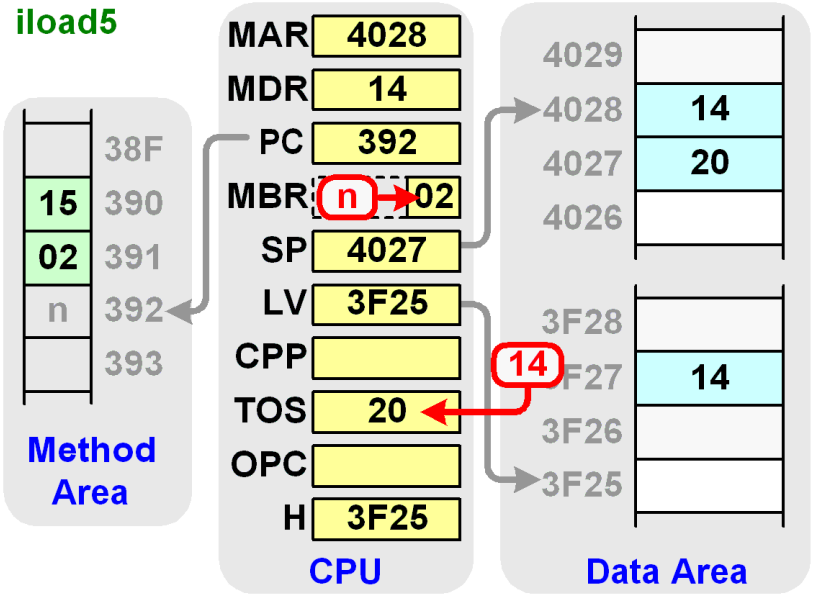
- Points the PC to the opcode of the next IJVM instruction ("n" at address 392 in the example).
- Fetches the opcode into the MBR register. We have to do this so that the opcode is ready to be decoded when we go back to "Main1" to execute the next IJVM instruction.

  The opcode doesn't arrive in the MBR register until the next clock cycle.
- Initiates a memory write to write the MDR contents (local variable value 14) to the address in MAR (top of stack address 4028)



iload5 TOS = MDR; goto Main1

- Copies the new top of stack value (14) to the TOS register.
- The next IJVM instruction opcode ("n") fetched in "iload4" is put into the MBR register.
- Goes to Main1 to decode and execute the next instruction



Microcode for the ISTORE Instruction

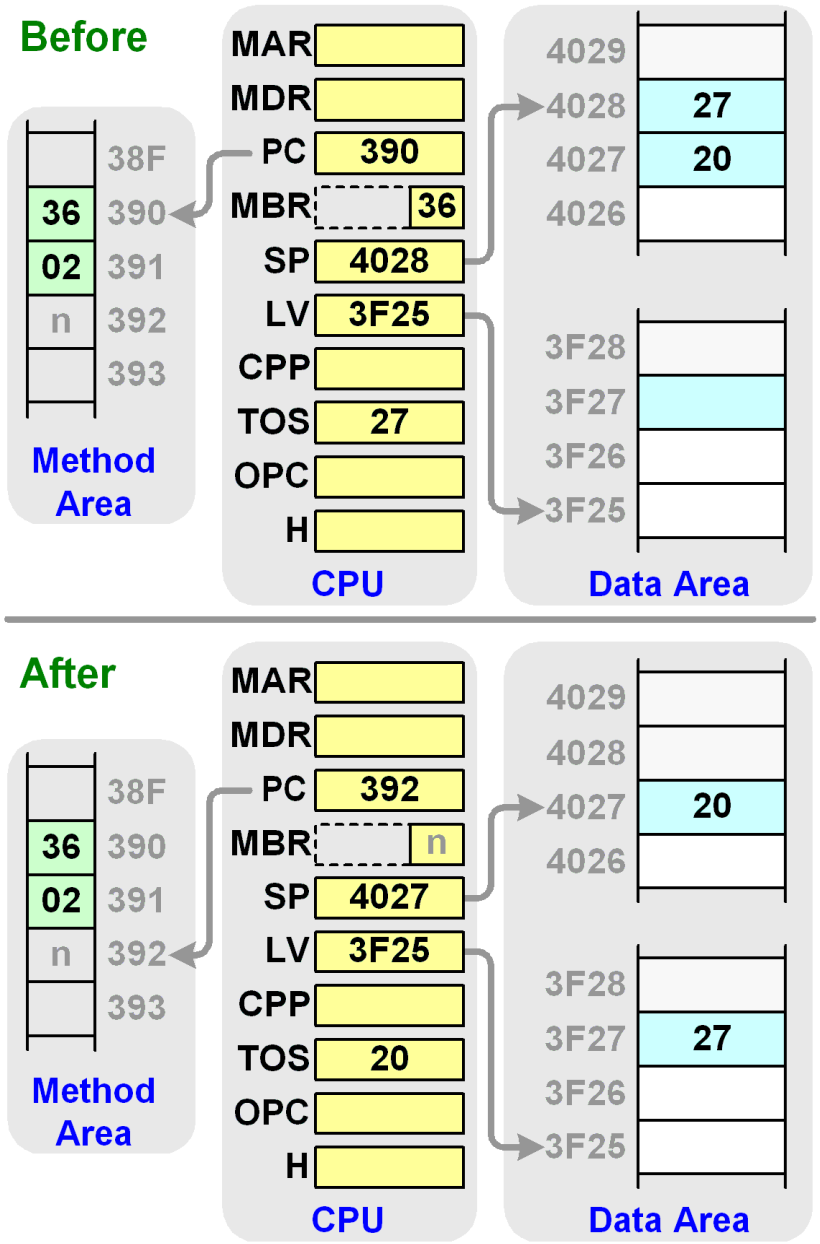
The ISTORE instruction includes an unsigned 8-bit operand after the opcode in the Method Area:

ISTORE 0x36	local variable #

ISTORE pops the top word off the stack and writes it to the given variable in the local variable frame.

This means that the microinstructions for ISTORE must do the following work:

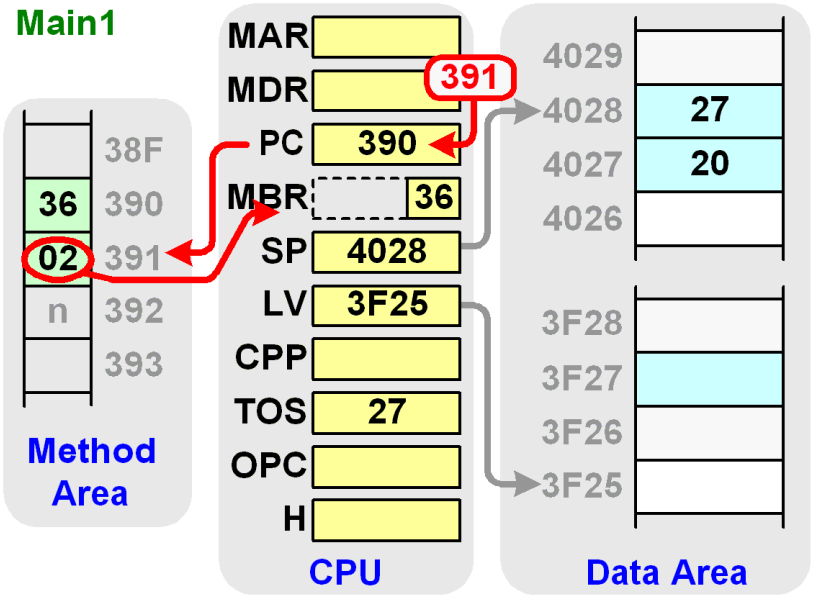
- **Add one to the PC and fetch the unsigned local variable number from the Method Area**
- **Calculate the address of the variable by adding the local variable number to the LV register**
- **Write the TOS register to the local variable in memory at the calculated address**
- **Subtract 1 from the SP register**
- **Read the new top-of-stack value from memory and put it into the TOS register**
- **Add 1 to the PC again so that it points to the next IJVM instruction's opcode ("n" in the diagram) and fetch the opcode into MBR.**



Microcode for the ISTORE Instruction

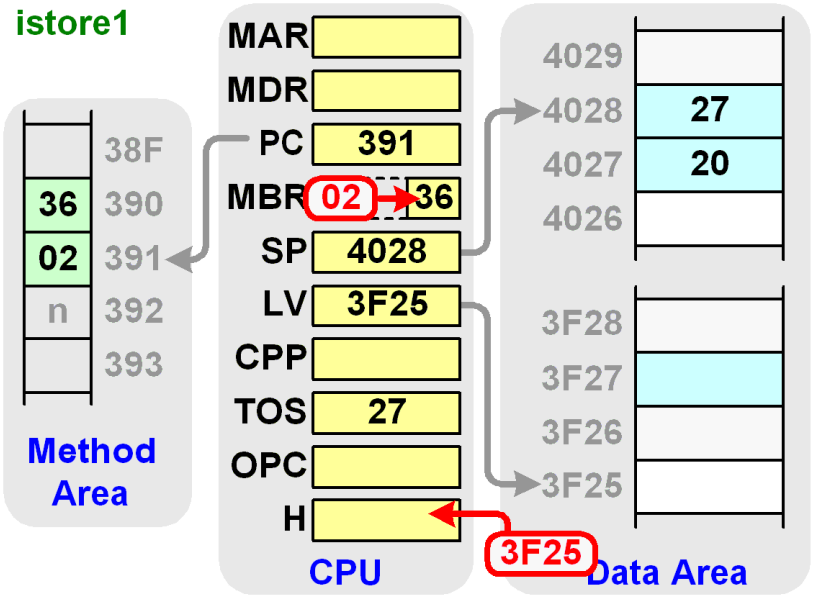
Main1 PC = PC + 1; fetch; goto (MBR)

- Increments the PC register (it changes from 390 to 391 in the example)
- Fetches the next byte – this is the local variable no. (2) which will be pushed onto the stack. The operand is read but it doesn't arrive in the MBR register until the next clock cycle.
- The "goto (MBR)" transfers control to the "istore1" microinstruction because the opcode is hex 36 and "istore1" is stored at control store address hex 36.



istore1 H = LV

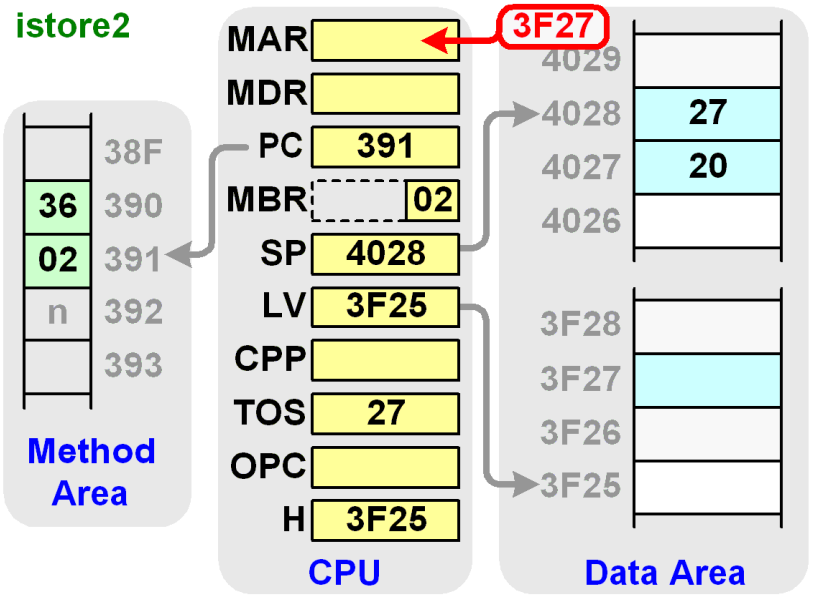
- Copies the LV pointer to the H register. This is to prepare for calculating the address of the local variable by adding it to the ISTORE operand. We have to copy LV to H because we need it to come into the "A" input of the ALU in order to add it.
- The ISTORE operand (local variable number 2) fetched in "Main1" is put into the MBR register.



Microcode for the ISTORE Instruction

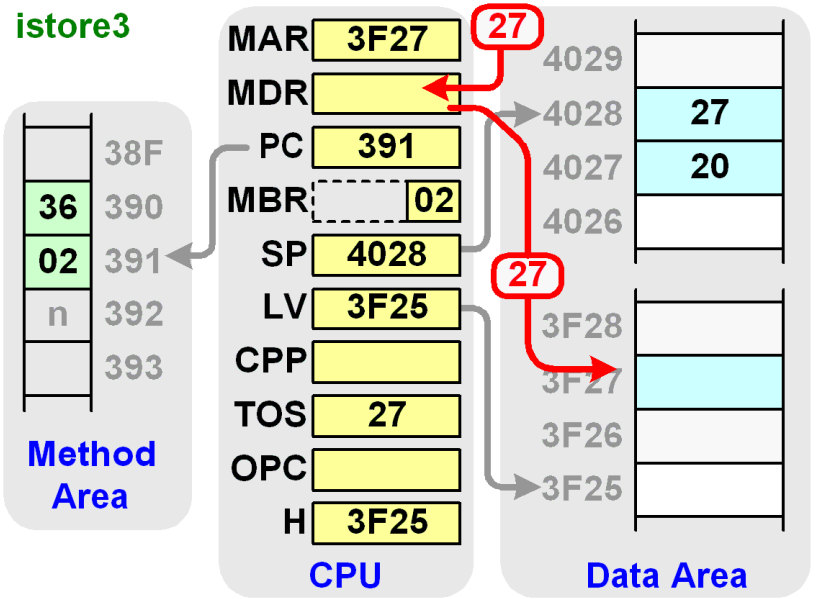
istore2 **MAR = MBRU + H**

- Calculates the address of the local variable to be pushed onto the stack by adding the local variable number (2, in MBR) to the copy of the LV register in H (3F25).
- The calculated address (3F27) is put into MAR so that we will be able to write the top of stack value to it.



istore3 **MDR = TOS; wr**

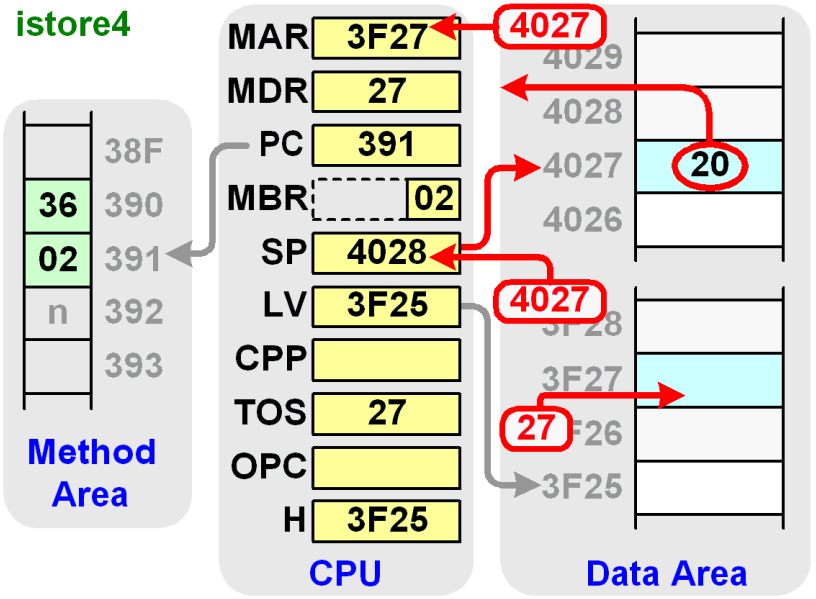
- The value being popped from the stack (27) is copied into the MDR register so that it can be written to the local variable.
- A memory write is initiated. The value in the MDR register (27) is written to the memory address specified by the MAR register (3F27). The value will actually be stored in memory at the end of the next clock cycle.



Microcode for the ISTORE Instruction

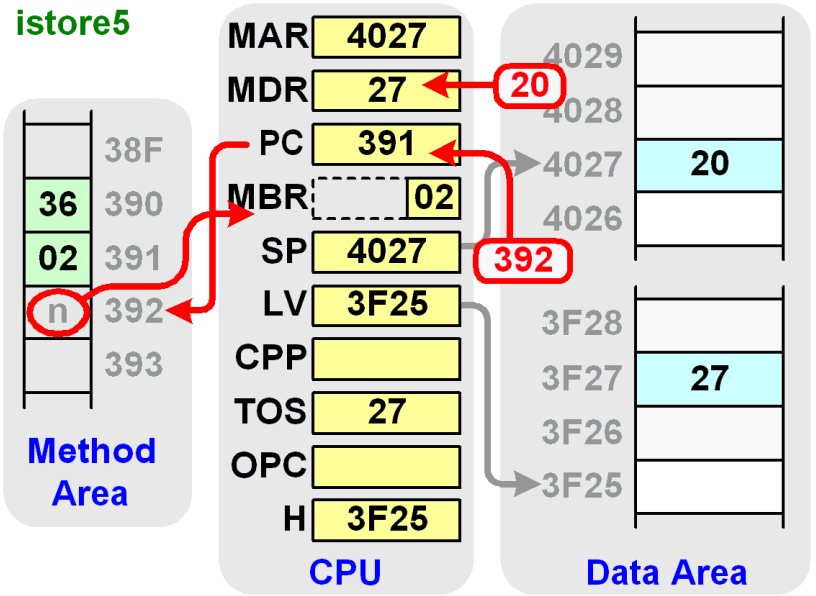
istore4 SP = MAR = SP – 1; rd

- Decrements the SP register to point to the new top of stack word (at address 4027).
- Also puts this address to MAR so it can be read
- Initiates a memory read from the address in MAR (4027) get the new top-of-stack value. This value will arrive in MDR at the end of the next clock cycle.
- The local variable (27) written by "istore3" is stored in memory.



istore5 PC = PC + 1; fetch

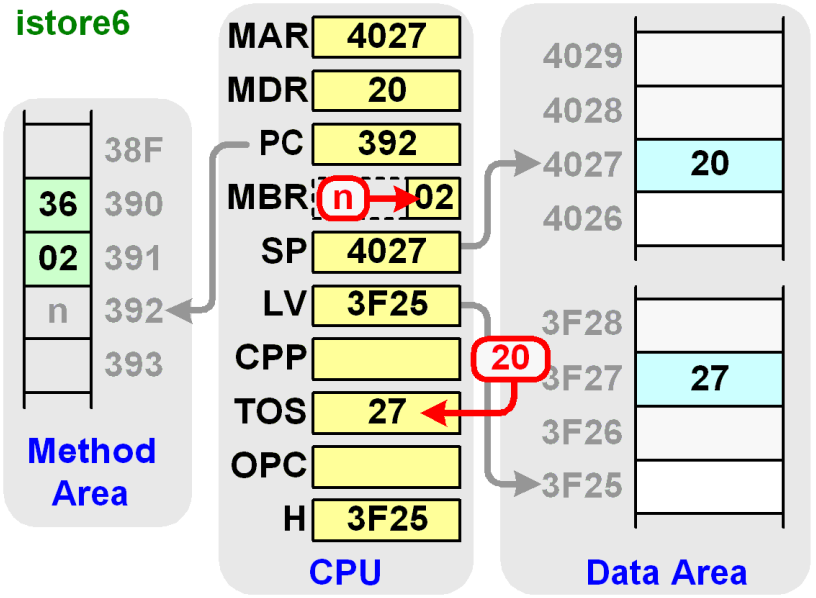
- Increments the PC to point to the opcode of the next IJVM instruction ("n" at address 392 in the diagram)
- Initiates a fetch to read the opcode into the MBR register. The opcode will arrive in the register at the end of the next clock cycle.
- The new top-of-stack value (20) that was read in "istore4" is placed into the MDR register.



Microcode for the ISTORE Instruction

istore6 TOS = MDR; goto Main1

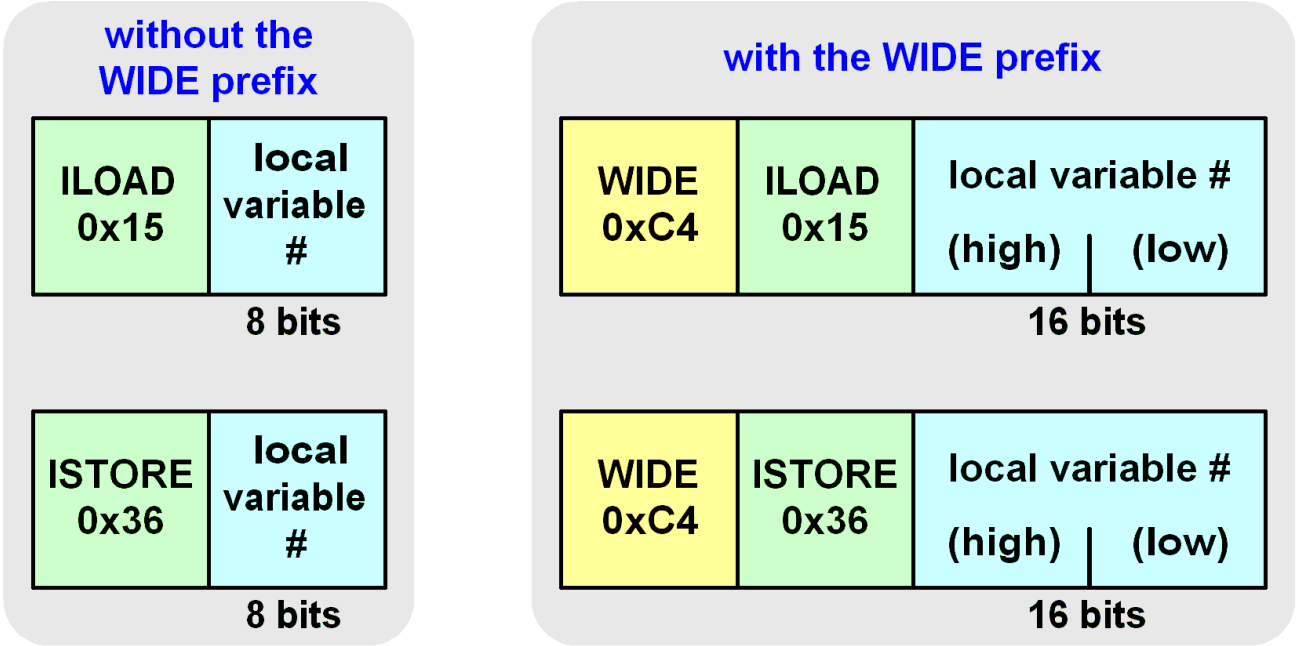
- **Saves the new top-of-stack value (20) in the TOS register.**
- **The next IJVM instruction opcode ("n") fetched in "istore5" is put into the MBR register.**
- **Goes to Main1 to decode and execute the next instruction**



The WIDE Prefix

A problem with the ILOAD and ISTORE instructions is that their 8-bit operand can only specify up to 256 local variables (variable numbers 0 through 255). If a Java procedure has more than 256 local variables then there's no way for ILOAD or ISTORE to access them.

The solution to this problem is the WIDE prefix. This is a special opcode byte which is placed in front of ILOAD, ISTORE, and other JVM instruction opcodes to change the size of their operands from 8 bits to 16 bits:



With the WIDE prefix, ILOAD and ISTORE can access up to 65,536 local variables.

Note that Java uses Big Endian notation for storing multibyte integers, so for the WIDE versions of ILOAD and ISTORE the high-order part of the variable number comes right after the opcode, followed by the low-order part.

Decoding the WIDE Prefix

Decoding instructions with the WIDE prefix is a challenge. The ILOAD opcode is the same (0x15) whether it has WIDE in front of it or not, but we need a different set of microinstructions for WIDE ILOAD that for a normal ILOAD (and the same applies to ISTORE).

When a WIDE prefix opcode is encountered in the instruction stream, the "Main1" microinstruction decodes it and jumps to control store address hex "0C4" (because the WIDE opcode is hex "C4"). That address holds the microinstruction which performs the decoding of the opcode following the WIDE prefix:

wide1 PC = PC + 1; fetch; goto (MBR OR 0x100)

When this microinstruction executes, the contents of the MBR register are the opcode byte (ie, the ILOAD or ISTORE opcode) following the WIDE prefix. This microinstruction looks very much like the "Main1" microinstruction, except for the "goto" clause:

goto (MBR OR 0x100)

The "Main1" microinstruction uses the JMPC bit with a zero "next address" value, so the next microinstruction address is the same as the opcode byte in the MBR register.

But the "wide1" microinstruction uses the JMPC bit with a "next address" value of hex "100". This means that the next 9-bit microinstruction address is hex 100 higher than the ISTORE or ILOAD opcode:

ILOAD Opcode (hex 15):	0 0001 0101
...ORed with hex 100:	1 0000 0000
new μinst address (hex 115):	1 0001 0101

ISTORE Opcode (hex 36):	0 0011 0110
...ORed with hex 100:	1 0000 0000
new μinst address (hex 136):	1 0011 0110

Decoding the WIDE Prefix

As a result of this, the "wide1" microinstruction decodes the ILOAD and ISTORE opcodes differently than the "Main1" microinstruction does.

Main1 – decodes the opcodes to microinstructions stored in the control store at an address equal to the opcode values (ie, the ILOAD opcode of 0x15 causes the microinstruction at control store address 0x015 to be executed)

wide1 – decodes the opcodes to microinstructions stored in the control store at an address that is hex 100 higher than the opcode values (ie, the ILOAD opcode of 0x15 causes the microinstruction at control store address 0x115 to be executed)

This means we have two different sequences of microinstructions which can be executed when an ILOAD opcode is decoded from the instruction stream.

iload1 – this microinstruction is located at control store address 0x015 and is executed when an ILOAD without a WIDE prefix is decoded.

wide_iload1 – this microinstruction is located at control store address 0x115 and is executed when a WIDE prefix is followed by an ILOAD opcode.

The same processing occurs for the ISTORE opcode, except that the control store addresses are 0x036 for a normal ISTORE and 0x136 for an ISTORE that follows a WIDE prefix.

ILOAD and ISTORE are the only two instructions our IJVM subset that can have a wide prefix, but there are many more instructions in the real Java Virtual Machine that can use it.

Decoding the WIDE Prefix

The diagram at right shows the layout of microinstructions in the control store. The sequence of microinstructions that's executed for the WIDE and non-WIDE versions of ILOAD and ISTORE is as follows:

For ILOAD

1. **Main1 at 0x100**
2. **iload1 at 0x015**

For ISTORE

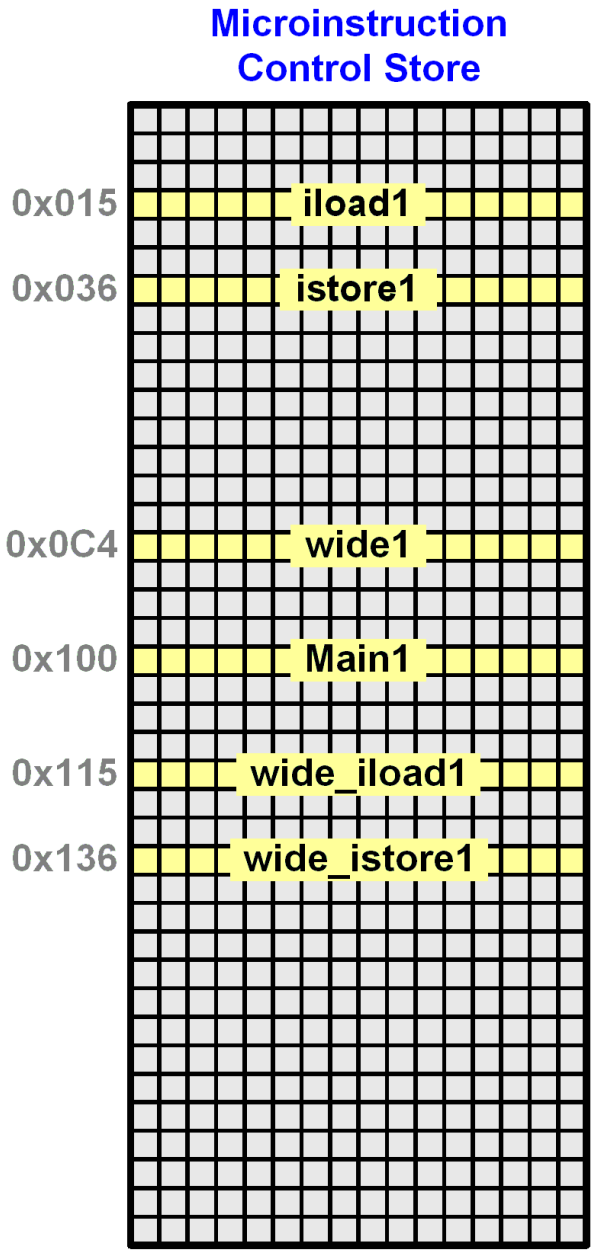
1. **Main1 at 0x100**
2. **istore1 at 0x036**

For WIDE ILOAD

1. **Main1 at 0x100**
2. **wide1 at 0x0C4**
3. **wide_iload1 at 0x115**

For WIDE ISTORE

1. **Main1 at 0x100**
2. **wide1 at 0x0C4**
3. **wide_istore1 at 0x136**



Exercise 9 – Wide Prefix

The INEG full Java Virtual Machine includes an instruction called LLOAD (load LONG variable) that's not part of the IJVM instructions we're implementing in our microarchitecture. Like ILOAD, it can be used with a WIDE prefix.

The LLOAD instruction has a hex opcode of 16.

If we were to implement this as an IJVM instruction, what control store address would we have to use for the first microinstruction that handles the WIDE version of LLOAD?

A – 000

B – 016

C – 0C4

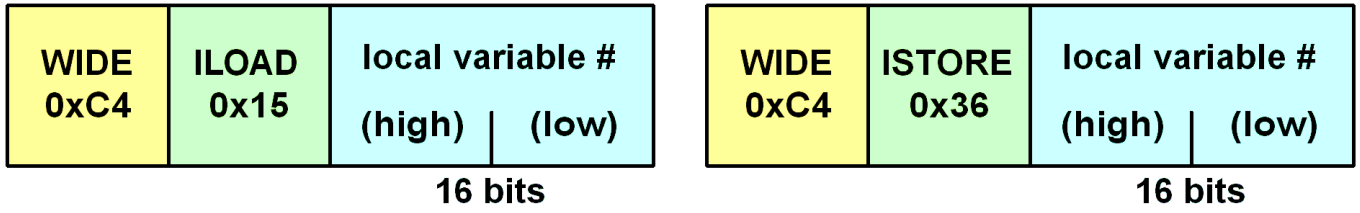
D – 100

E – 116

F – 1C4

Fetching the 16-bit Variable Number

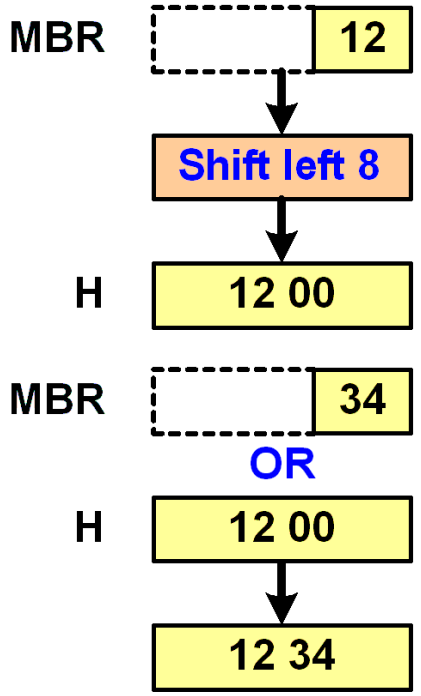
The opcodes for both the WIDE ILOAD and WIDE ISTORE IJVM instructions are followed by a 16-bit variable number:



Unfortunately, the memory control unit only allows us to fetch one byte (8 bits) at a time via the PC and MBR registers. So we have to read first the high-order byte of the variable number, then the low-order byte, then put the two of them back together again. This is done in two steps as follows (the examples show what happens for an operand of hex "1234"):

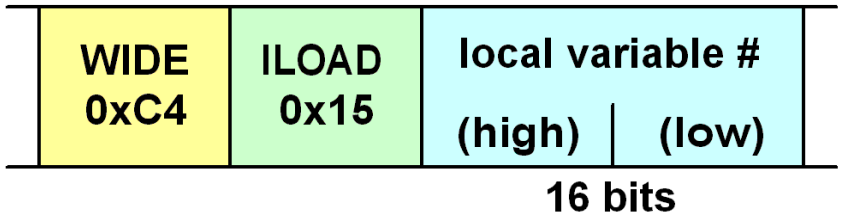
1. **After the high-order byte is fetched into MBR, it's copied to the H register and the shift register is activated to shift the value 8 bits to the left (via the SLL8 control signal).**

2. **After the low-order byte is fetched into MBR, it's ORed with the shifted value in H. The result is the 16-bit operand reassembled into one number.**



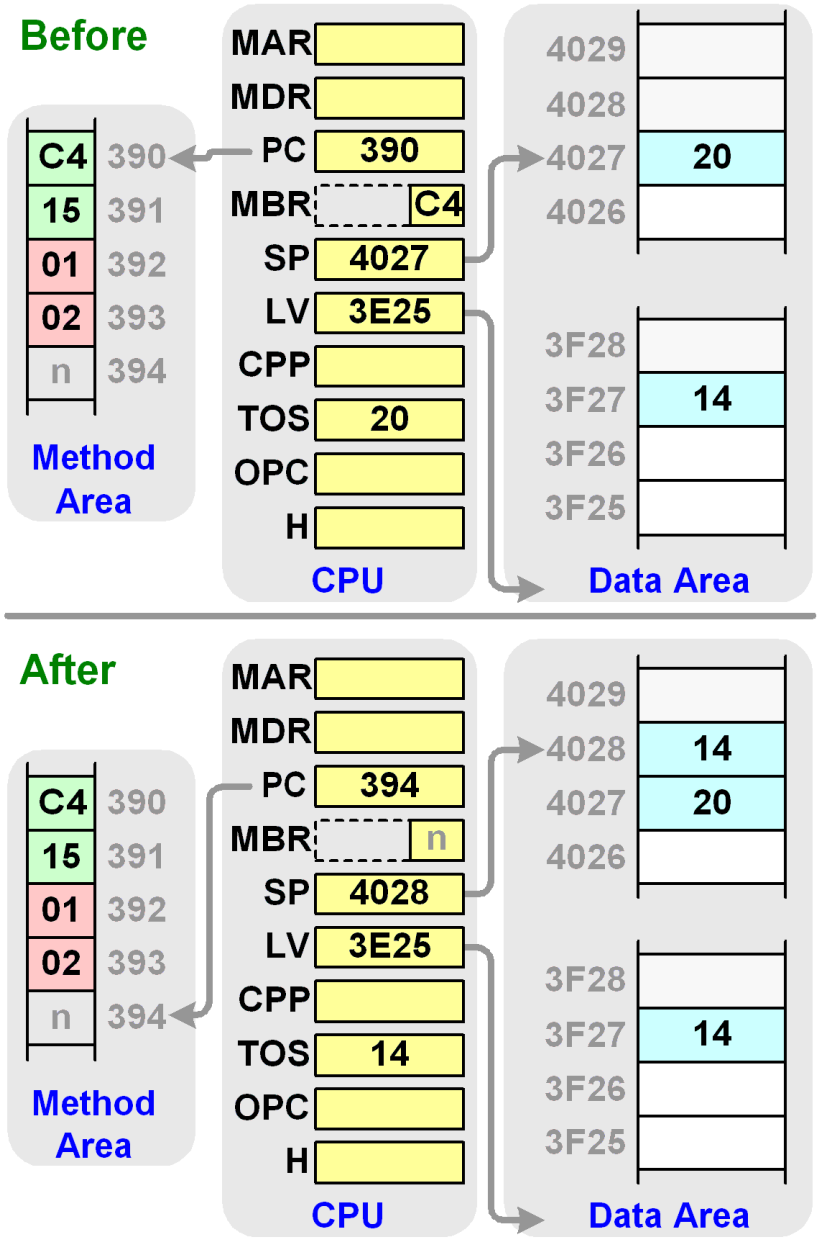
Microcode for the WIDE ILOAD Instruction

The format of the WIDE ILOAD instruction has been discussed above:



The microinstructions for ILOAD must do the following work:

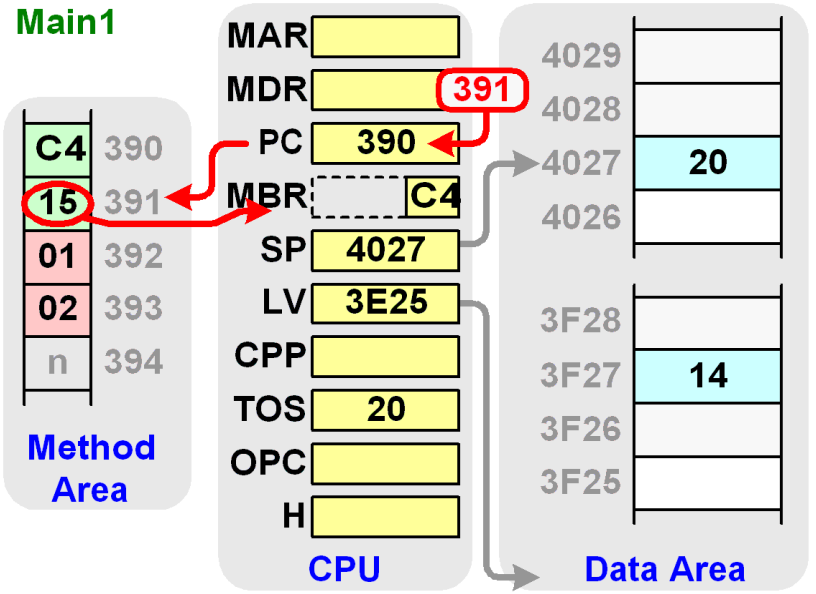
- **Decode the ILOAD opcode and go to the "wide_iload1" microinstruction**
- **Fetch and assemble the 2-byte variable number.**
- **Calculate the address of the variable by adding the variable number to the LV register**
- **Read the local variable from memory at the calculated address**
- **Write the variable to memory at the SP address and also put a copy in TOS**
- **Increment the PC so that it points to the next IJVM instruction's opcode ("n" in the diagram) and fetch the opcode into MBR.**



Microcode for the WIDE ILOAD Instruction

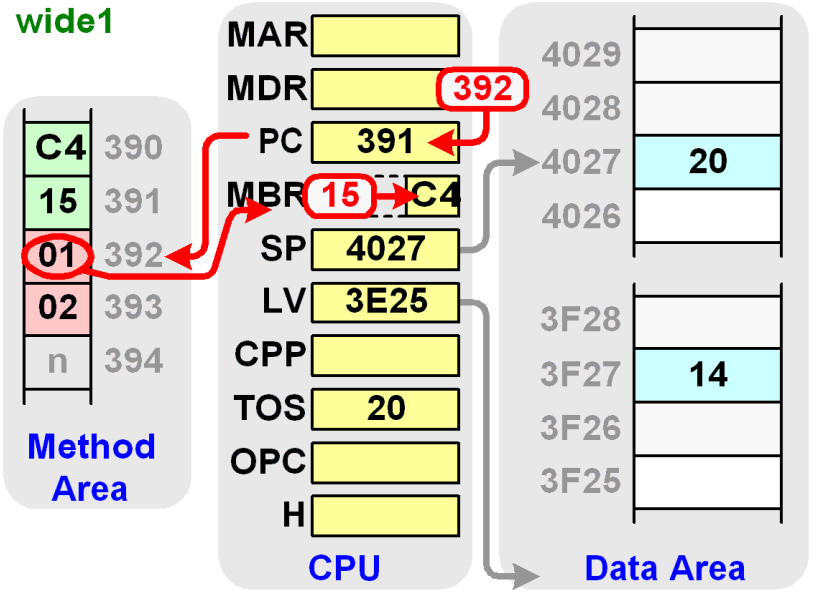
Main1 PC = PC + 1; fetch; goto (MBR)

- Increments the PC register (it changes from 390 to 391 in the example)
- Fetches the next byte – this is the ISTORE opcode (hex 15). The opcode is read but it doesn't arrive in the MBR register until the next clock cycle.
- The "goto (MBR)" transfers control to the "wide1" microinstruction because the opcode is hex C4 and "wide1" is stored at control store address hex C4.



wide1 PC = PC+1; fetch; goto (MBR OR 0x100)

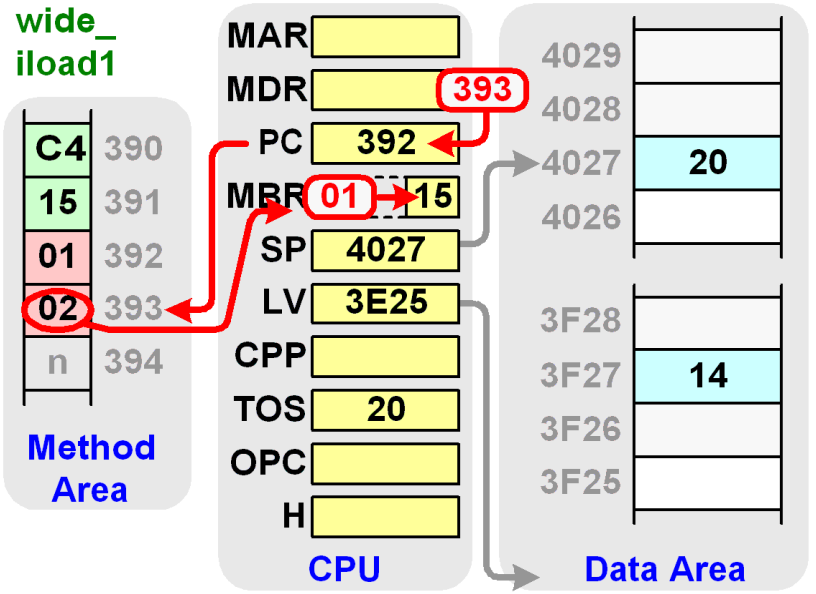
- Increments the PC register (it changes from 391 to 392 in the example)
- Fetches the next instruction byte – this is the high-order part of the 16-bit variable number. The number is read but it doesn't arrive in the MBR register until the next clock cycle.
- The "goto (MBR OR 0x100) transfers control to the "wide_iload1" microinstruction because the hex 15 opcode is ORed with hex 100 giving a result of hex 115, and "wide_iload1" is at control store address hex 115.



Microcode for the WIDE ILOAD Instruction

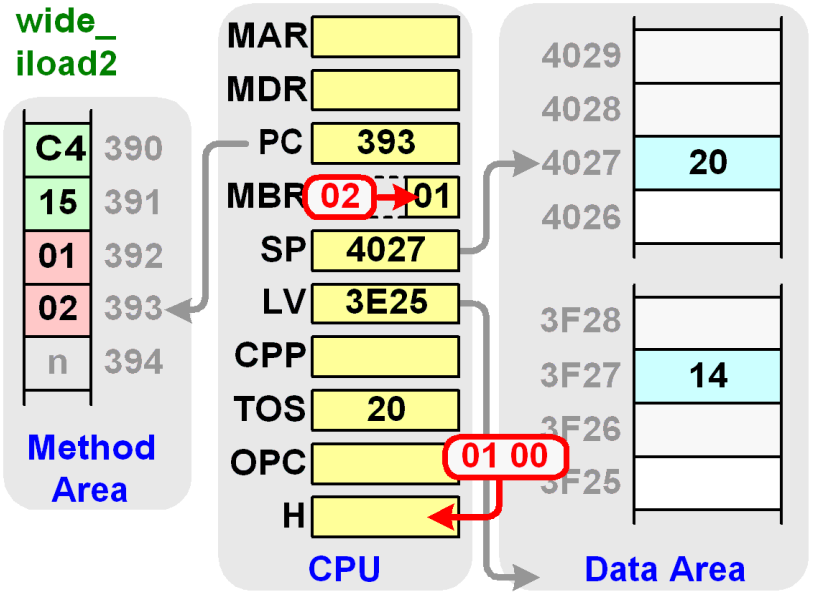
wide_iload1 PC = PC + 1; fetch

- Increments the PC register (it changes from 391 to 392 in the example)
- Fetches the next instruction byte – this is the low-order part of the 16-bit variable number. The number is read but it doesn't arrive in the MBR register until the next clock cycle.
- The high-order part of the variable number (01) that was fetched in "wide1" arrives in the MBR register in this clock cycle.



wide_iload2 H = MBRU << 8

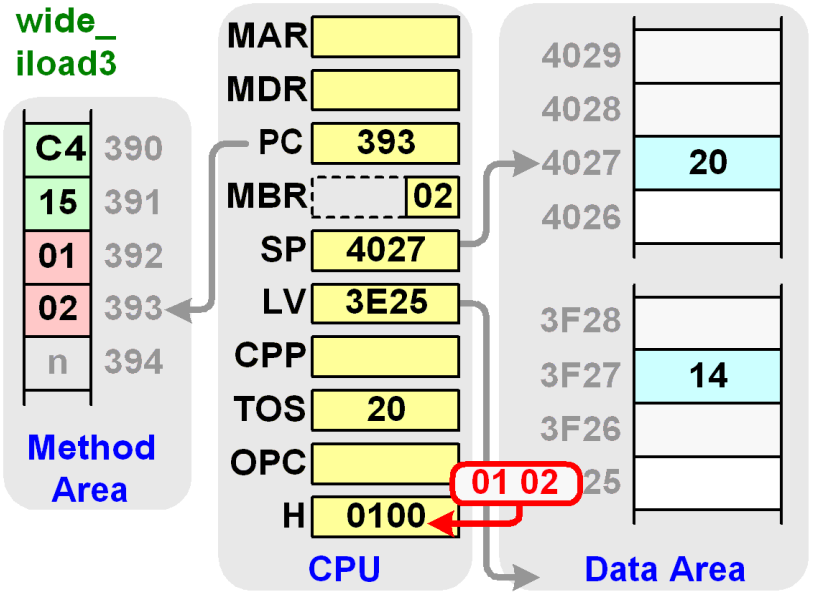
- Shifts the high order part of the local variable number (01 in the MBR register) to the left by 8 bits (resulting in hex 0100) and stores the result in H.
- The low-order part of the variable number (02) that was fetched in "wide_iload1" arrives in the MBR register in this clock cycle.



Microcode for the WIDE ILOAD Instruction

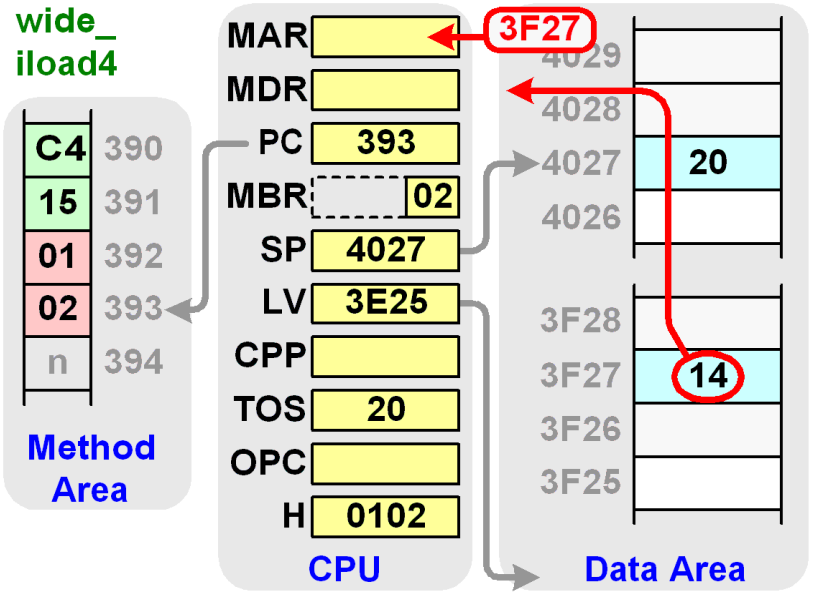
wide_iload3 H = MBRU OR H

- Combines the low-order part of the variable number (02 in MBR) with the high-order part (0100 in H) into a single value (0102) and stores the result in H



wide_iload4 MAR = LV + H; rd; goto iload3

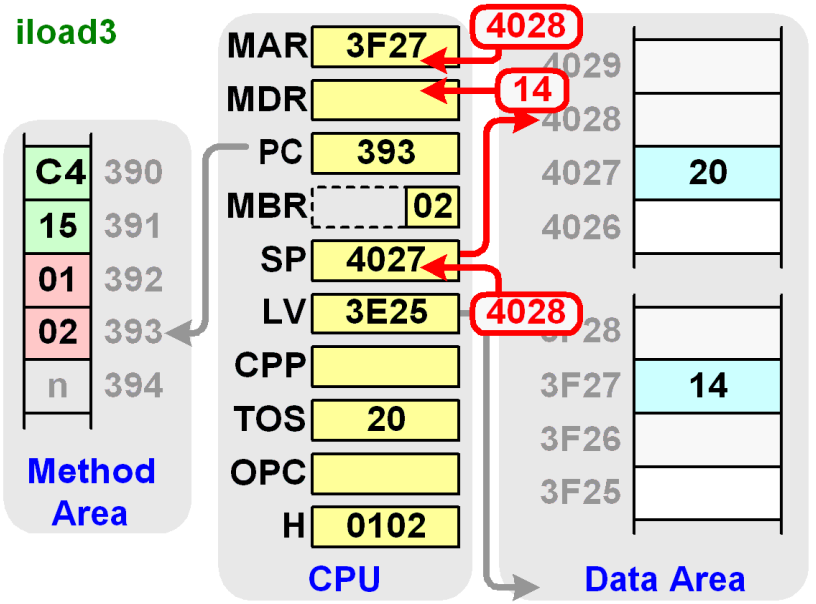
- Calculates the address of the local variable by adding the variable number (0102 in H) to the LV pointer.
- The address is put into MAR so that the local variable can be read.
- Initiates a read operation to the local variable whose address is in MAR. The read value will be put into MDR on the next clock cycle.
- The everything else is the same as the microinstructions for the ILOAD instruction, we jump to those to complete the work.



Microcode for the WIDE ILOAD Instruction

iload3 MAR = SP = SP + 1

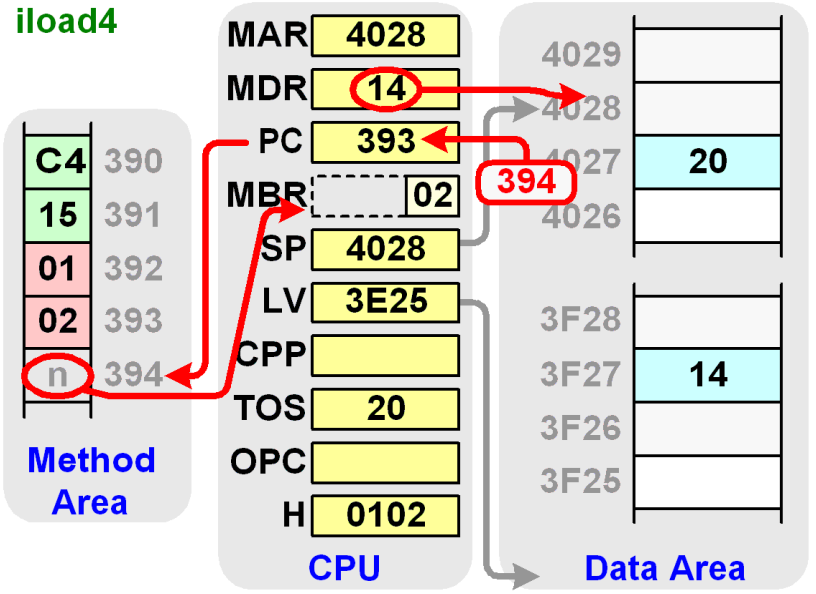
- Points the SP register to the new top-of-stack location (address 4028)
- Puts the same address into MAR so that we can write the local variable to the new top-of-stack location after it arrives in MDR
- The local variable read in "wide_iload4" is put into the MDR register during this clock cycle.



iload4 PC = PC + 1; fetch; wr

- Points the PC to the opcode of the next IJVM instruction ("n" at address 394 in the example).
- Fetches the opcode into the MBR register. We have to do this so that the opcode is ready to be decoded when we go back to "Main1" to execute the next IJVM instruction.

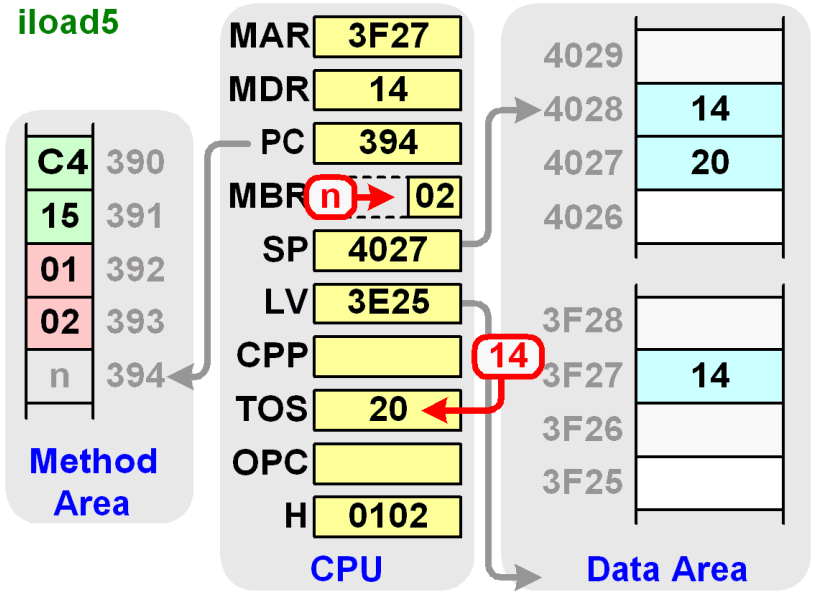
  The opcode doesn't arrive in the MBR register until the next clock cycle.
- Initiates a memory write to write the MDR contents (local variable value 14) to the address in MAR (top of stack address 4028)



Microcode for the WIDE ILOAD Instruction

iload5 TOS = MDR; goto Main1

- **Copies the new top of stack value (14) to the TOS register.**
- **The next IJVM instruction opcode (“n”) fetched in “iload4” is put into the MBR register.**
- **Goes to Main1 to decode and execute the next instruction**

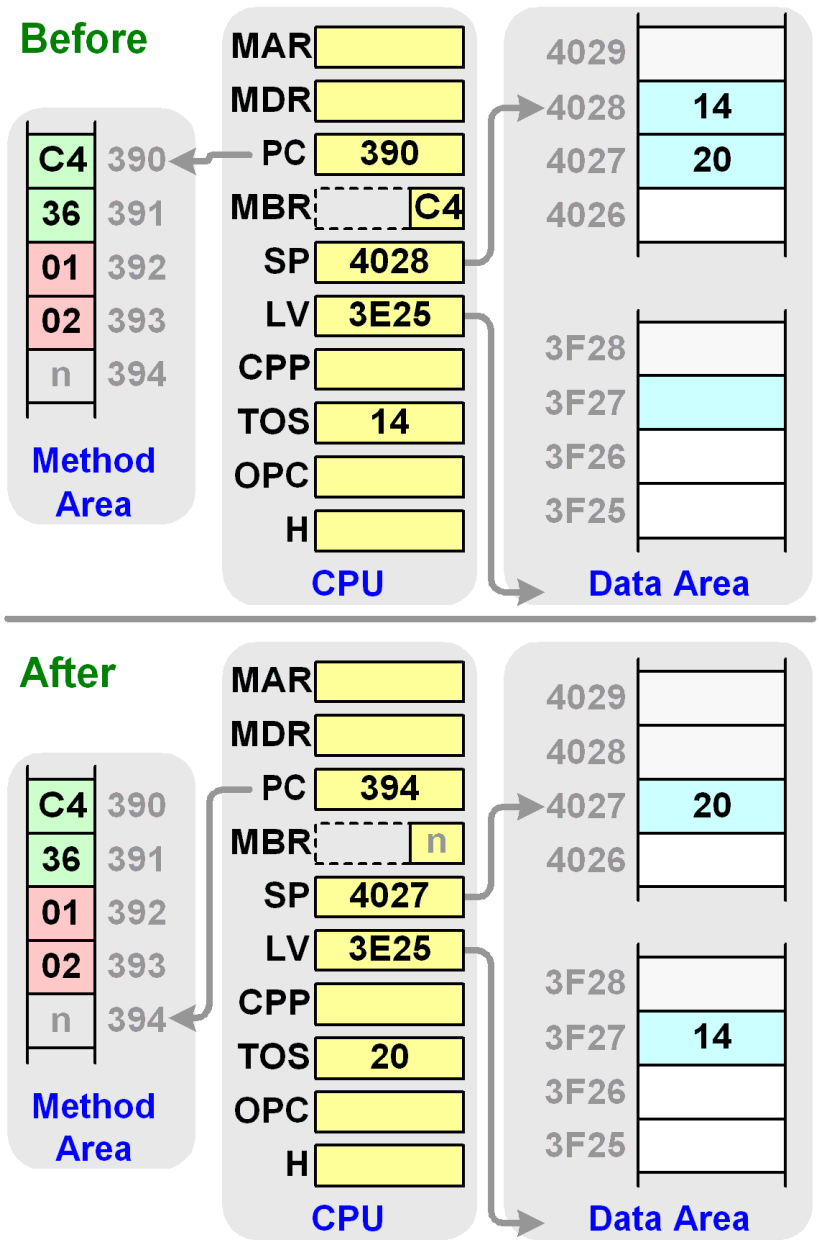


Microcode for the WIDE ISTORE Instruction

The format of the WIDE ISTORE instruction has been discussed above:

The microinstructions for ISTORE must do the following work:

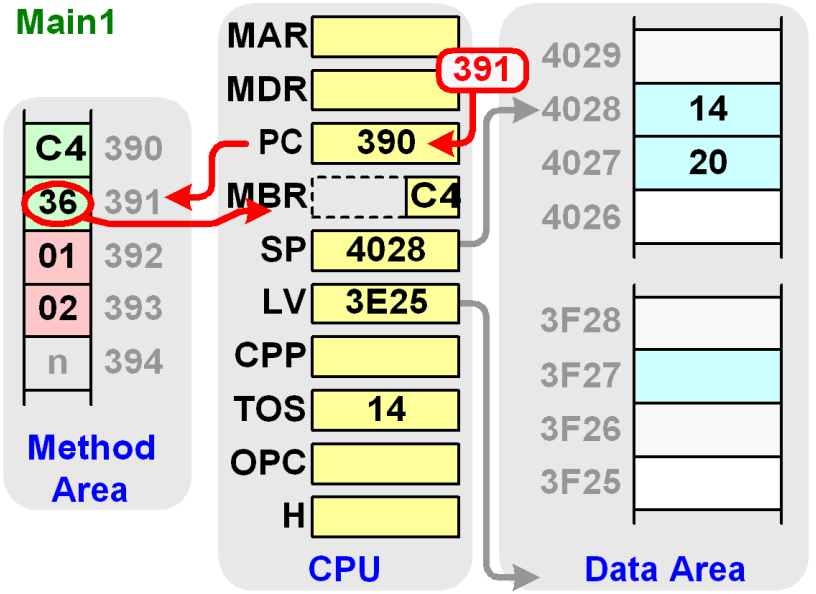
- **Decode the ISTORE opcode and go to the "wide_istore1" microinstruction**
- **Fetch and assemble the 2-byte variable number.**
- **Calculate the address of the variable by adding the variable number to the LV register**
- **Write the TOS register to the local variable in memory at the calculated address**
- **Adjust the SP register and read the new top-of-stack value into the TOS register.**
- **Increment the PC so that it points to the next IJVM instruction's opcode ("n" in the diagram) and fetch the opcode into MBR.**



Microcode for the WIDE ISTORE Instruction

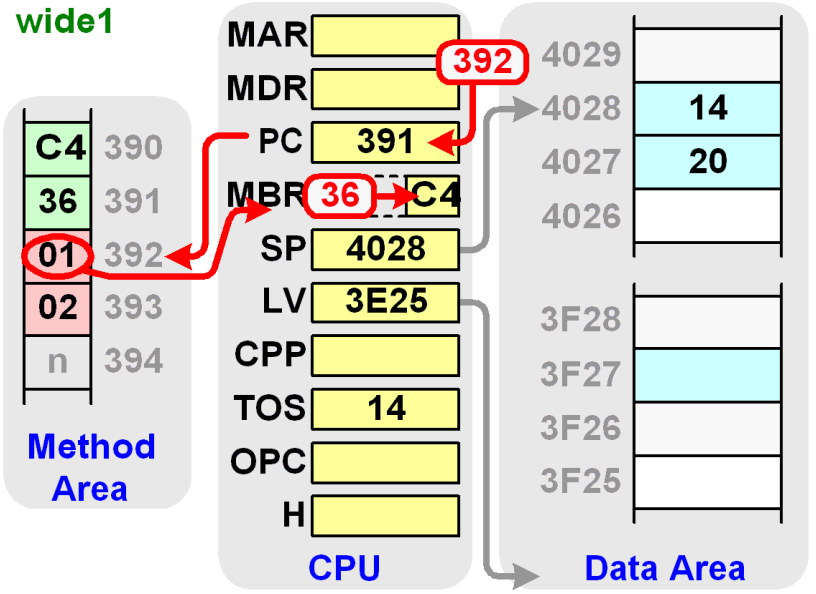
Main1 PC = PC + 1; fetch; goto (MBR)

- Increments the PC register (it changes from 390 to 391 in the example)
- Fetches the next byte – this is the ILOAD opcode (hex 36). The opcode is read but it doesn't arrive in the MBR register until the next clock cycle.
- The "goto (MBR)" transfers control to the "wide1" microinstruction because the opcode is hex C4 and "wide1" is stored at control store address hex C4.



wide1 PC = PC+1; fetch; goto (MBR OR 0x100)

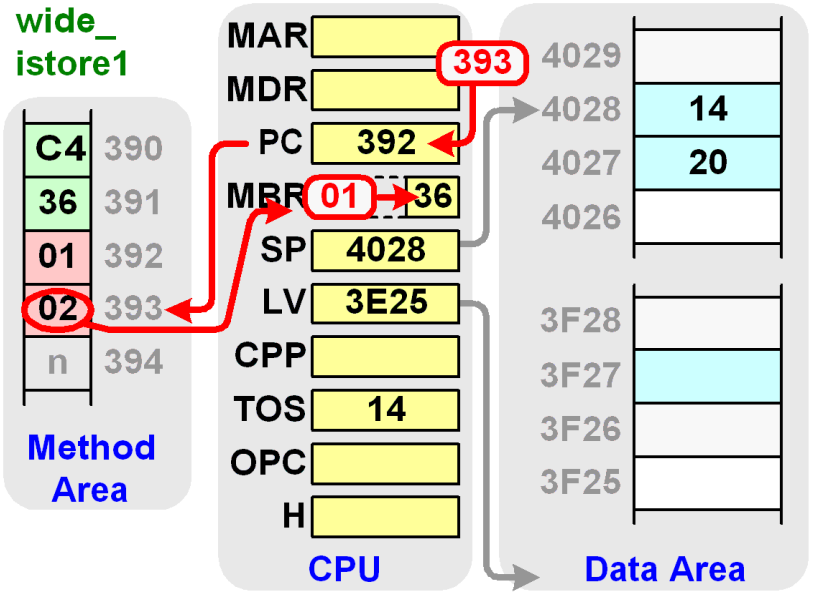
- Increments the PC register (it changes from 391 to 392 in the example)
- Fetches the next instruction byte – this is the high-order part of the 16-bit variable number. The number is read but it doesn't arrive in the MBR register until the next clock cycle.
- The "goto (MBR OR 0x100) transfers control to the "wide_istore1" microinstruction because the hex 36 opcode is ORed with hex 100 giving a result of hex 136, and "wide_istore1" is at control store address hex 136.



Microcode for the WIDE ISTORE Instruction

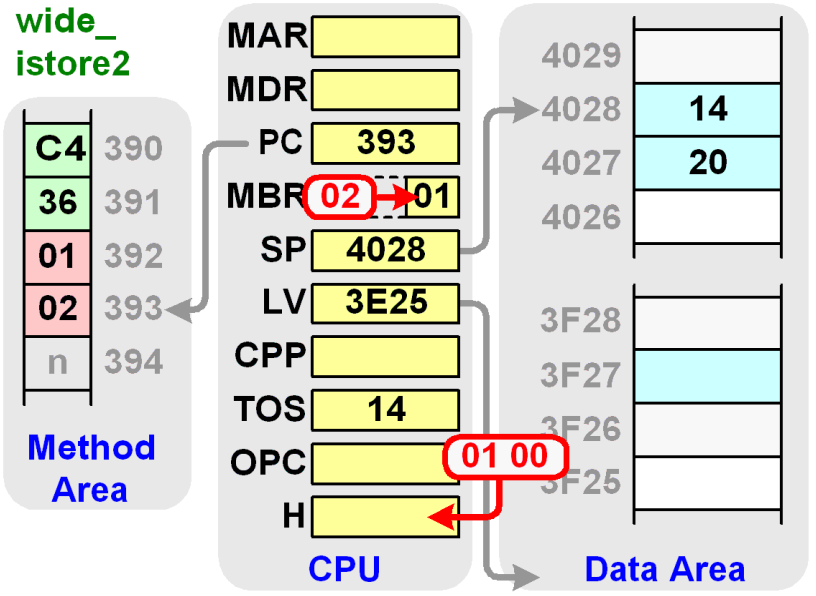
wide_istore1 PC = PC + 1; fetch

- Increments the PC register (it changes from 391 to 392 in the example)
- Fetches the next instruction byte – this is the low-order part of the 16-bit variable number. The number is read but it doesn't arrive in the MBR register until the next clock cycle.
- The high-order part of the variable number (01) that was fetched in "wide1" arrives in the MBR register in this clock cycle.



wide_istore2 H = MBRU << 8

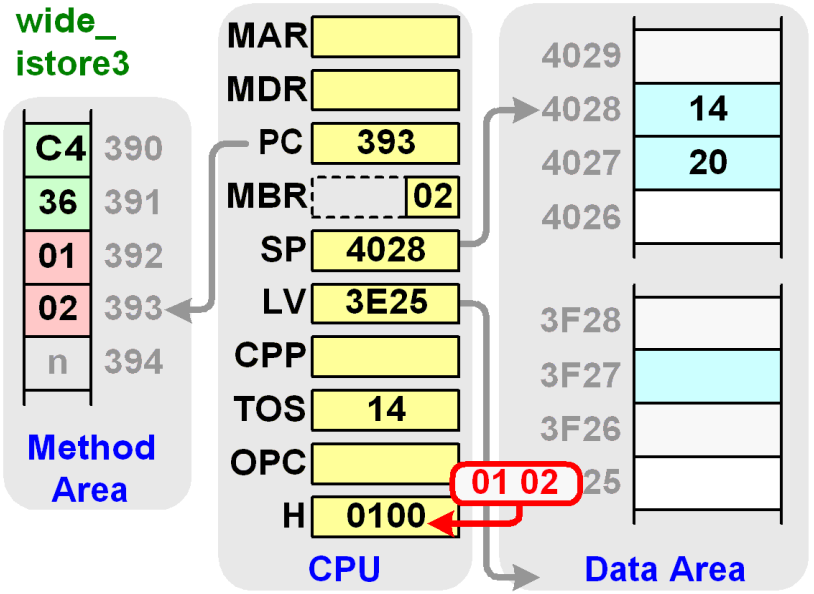
- Shifts the high order part of the local variable number (01 in the MBR register) to the left by 8 bits (resulting in hex 0100) and stores the result in H.
- The low-order part of the variable number (02) that was fetched in "wide_istore1" arrives in the MBR register in this clock cycle.



Microcode for the WIDE ISTORE Instruction

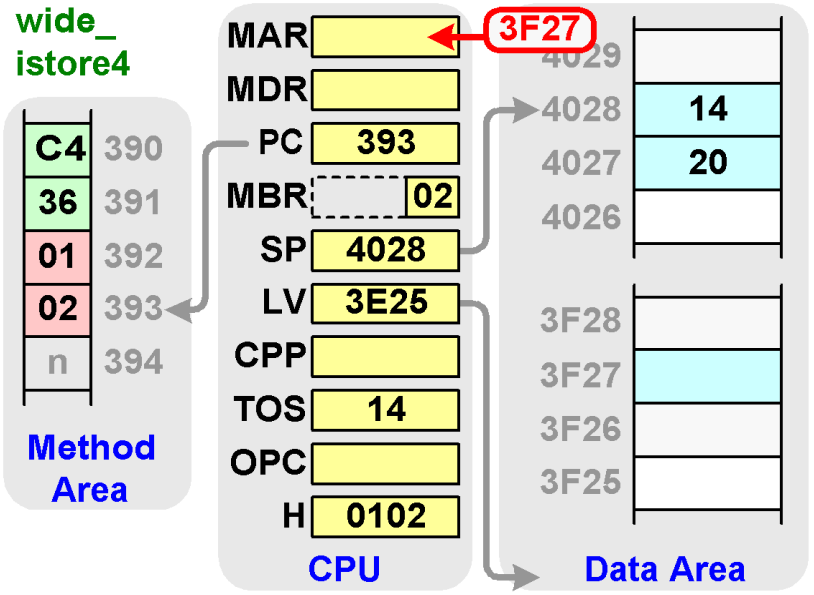
wide_istore3 H = MBRU OR H

- Combines the low-order part of the variable number (02 in MBR) with the high-order part (0100 in H) into a single value (0102) and stores the result in H



wide_istore4 MAR = LV + H; goto istore3

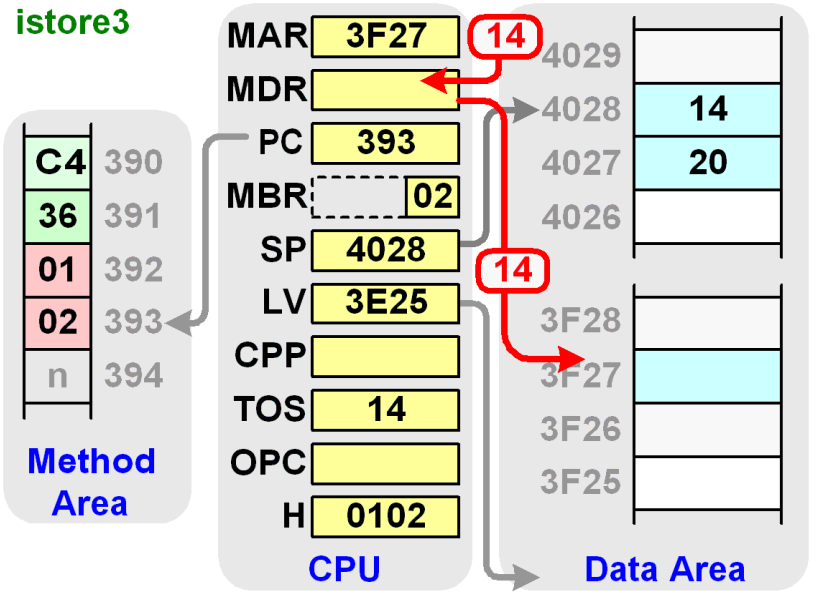
- Calculates the address of the local variable by adding the variable number (0102 in H) to the LV pointer.
- The address is put into MAR so that the local variable can be written.
- The rest of the work is the same as the microinstructions for the ISTORE instruction, we jump to those to complete the work.



Microcode for the WIDE ISTORE Instruction

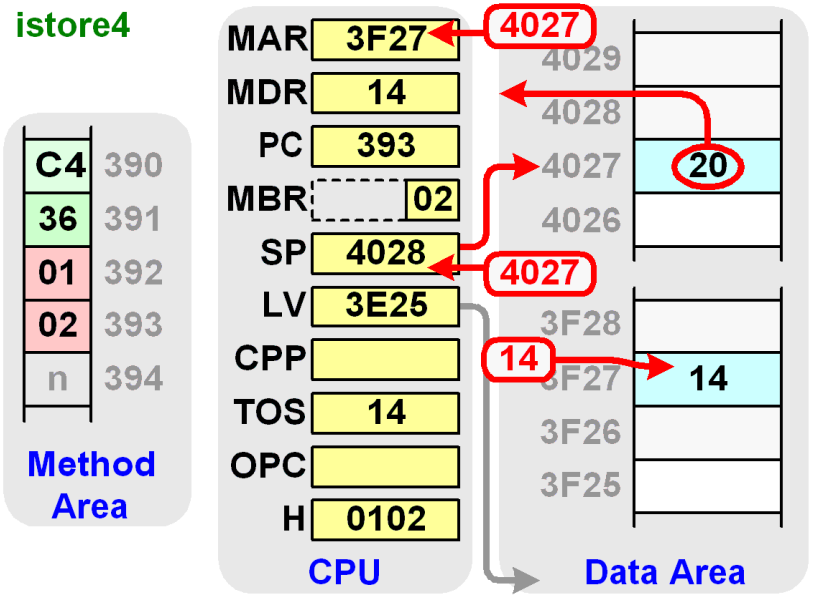
istore3 MDR = TOS; wr

- The value being popped from the stack (14) is copied into the MDR register so that it can be written to the local variable.
- A memory write is initiated. The value in the MDR register (14) is written to the memory address specified by the MAR register (3F27). The value will actually be stored in memory at the end of the next clock cycle.



istore4 SP = MAR = SP – 1; rd

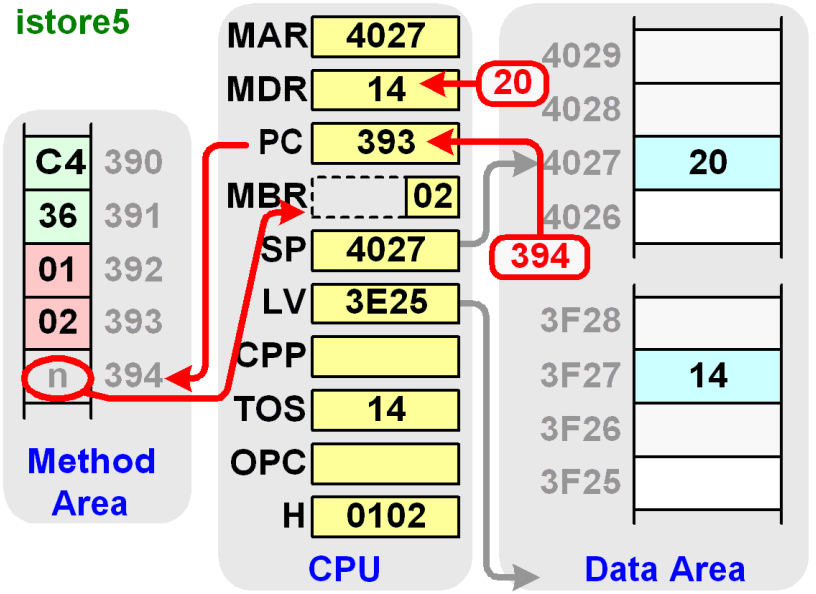
- Decrements the SP register to point to the new top of stack word (20 at address 4027).
- Also puts this address to MAR so it can be read
- Initiates a memory read from the address in MAR (4027) get the new top-of-stack value. This value will arrive in MDR at the end of the next clock cycle.
- The local variable (14) written by “istore3” is stored in memory.



Microcode for the WIDE ISTORE Instruction

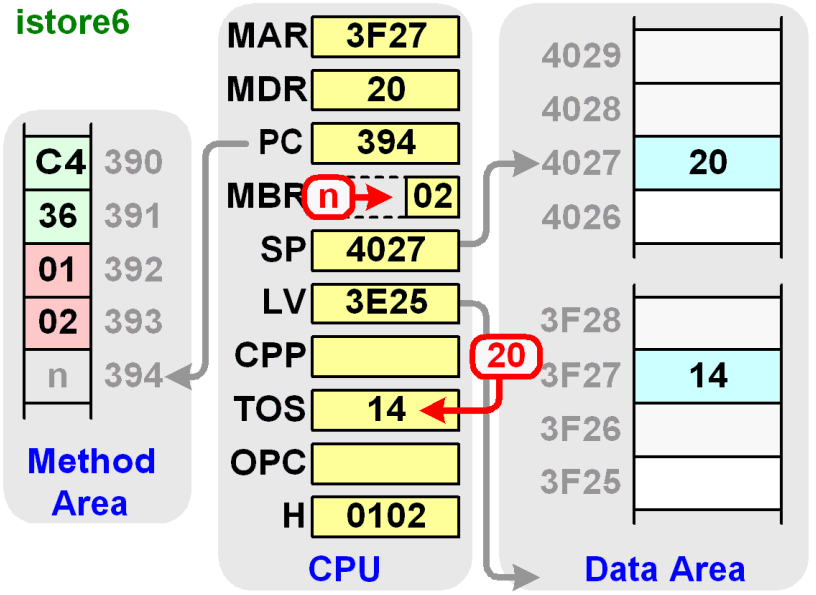
istore5 PC = PC + 1; fetch

- Increments the PC to point to the opcode of the next IJVM instruction ("n" at address 394 in the diagram)
- Initiates a fetch to read the opcode into the MBR register. The opcode will arrive in the register at the end of the next clock cycle.
- The new top-of-stack value (20) that was read in "istore4" is placed into the MDR register.



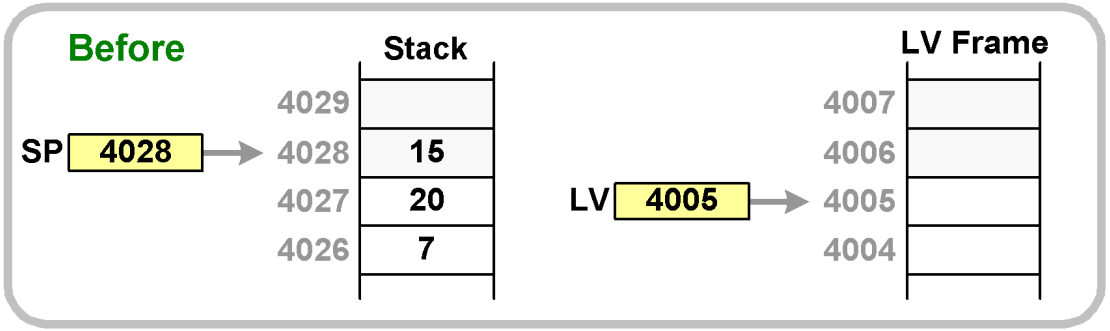
istore6 TOS = MDR; goto Main1

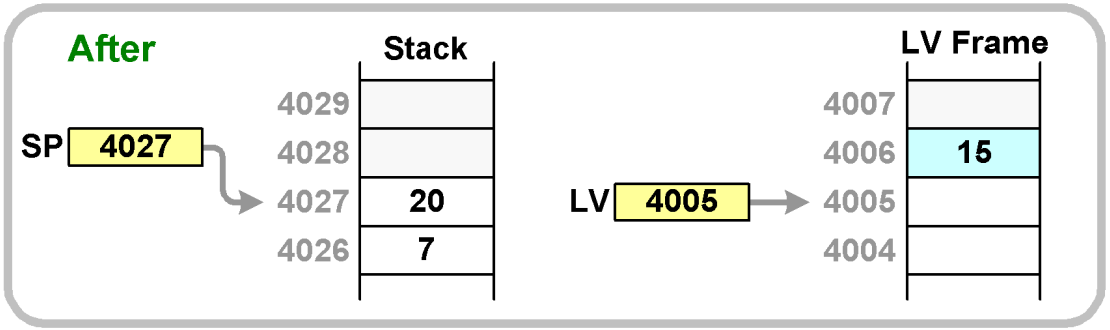
- Copies the new top of stack value (20) to the TOS register.
- The next IJVM instruction opcode ("n") fetched in "istore5" is put into the MBR register.
- Goes to Main1 to decode and execute the next instruction



Exercise 10 – The ISTORE_1 Instruction

The IJVM "ISTORE_1" instruction removes the top word from the stack and stores it into Local Variable 1.



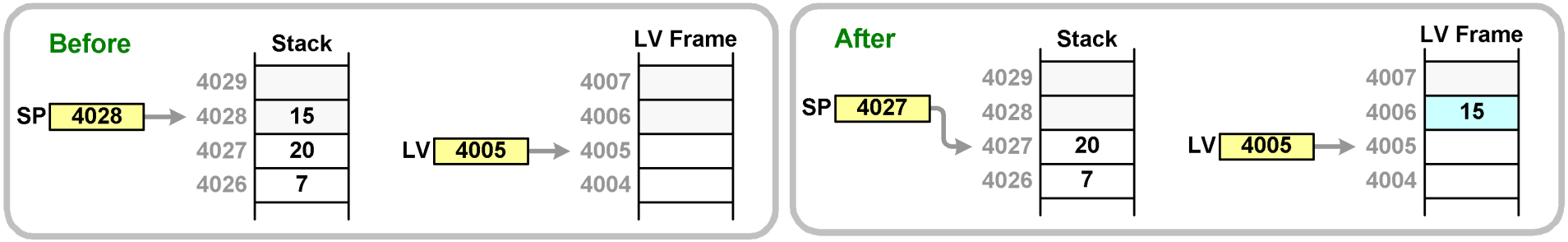


To perform the work for this instruction, the microcode must:

- **Calculate the address of the local variable (by adding "1" to the LV register)**
- **Write the TOS register to the local variable at the calculated address.**
- **Subtract 1 from the stack pointer**
- **Read the new top-of-stack value and save it in the TOS register.**
- **Point the PC register to the next instruction opcode and fetch it into the MBR register.**

(…continued next page)

Exercise 10 – The ISTORE_1 Instruction



Which of the following sequences of microcode is correct?

A

MAR = LV + 1; rd
MDR = TOS; wr
MAR = SP = SP – 1; rd
(no op)
TOS = MDR; goto Main1

B

MAR = LV + 1
MDR = TOS; wr
MAR = SP – 1; rd
TOS = MDR; goto Main1

C

MAR = LV + 1
MDR = TOS; wr
MAR = SP = SP – 1; rd
(no op)
TOS = MDR; goto Main1

Key Concepts

- How the microinstruction register controls the data path operation
- How microinstructions are stored in the control store
- How the next microinstruction is chosen
- Decoding of IJVM instructions
- Using MAL to symbolically represent microinstruction bits
- How the MIC-1 registers are used
- Keeping the correct value in the TOS register
- How the main interpretation loop works
- How IJVM instructions are decoded
- How the WIDE prefix is handled
- Where microinstructions for WIDE IJVM instructions are stored
- Fetching 16-bit numbers from the instruction stream

What's Next

- **Look at Review Questions for this week**
- **Sign on to WebCT and do Module 6 – "Infix Notation"**
- **Study for Quiz 4, which includes:**
  - **Week 7 and 8 lectures**
  - **WebCT modules 5 and 6**
- **Continue working on Assignment 2 (for the material covered so far)**
- **Pre-read the material for Week 9**