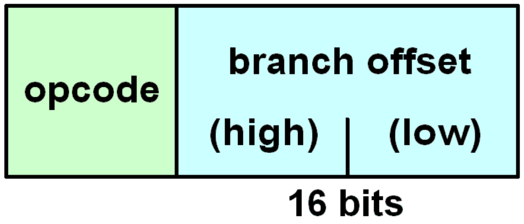
Today’s Topics

- How branch instructions work
- Improving the performance of the MIC-1 microarchitecture
- Reducing the number of microinstructions needed
- Adding dedicated hardware to fetch instructions
- Adding a third bus
- Adding a pipeline to the microarchitecture
- Writing pipelined microcode

Branch Instructions

4.3.2

The Branch instructions used in the Java Virtual Machine all have the same general format:



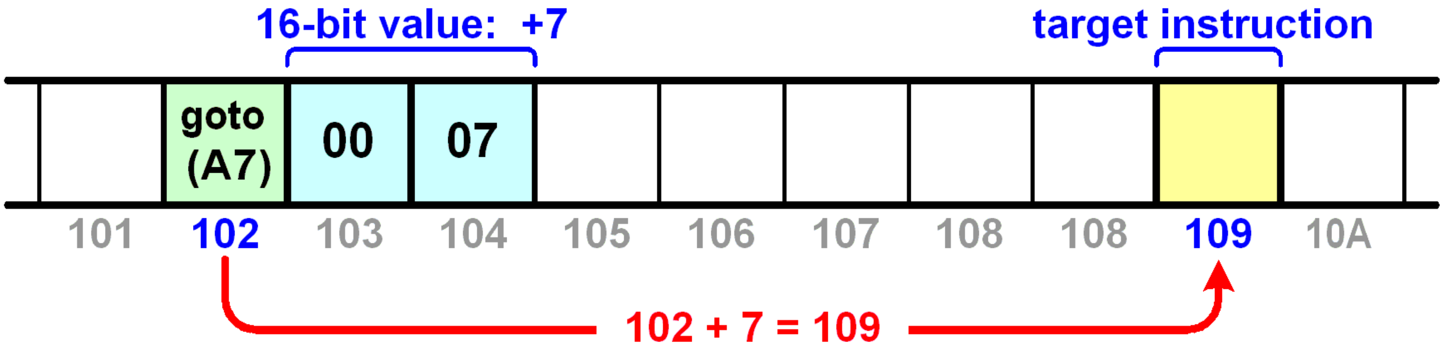
opcode

branch offset

(high) | (low)

16 bits

The purpose of a branch instruction is to transfer control to a different IJVM instruction other than the one which follows the branch. To do this, the 16-bit signed offset which follows the branch opcode is added to PC register. Since the PC register changes, a different IJVM instruction is executed next. Here are a couple of examples of how this works:



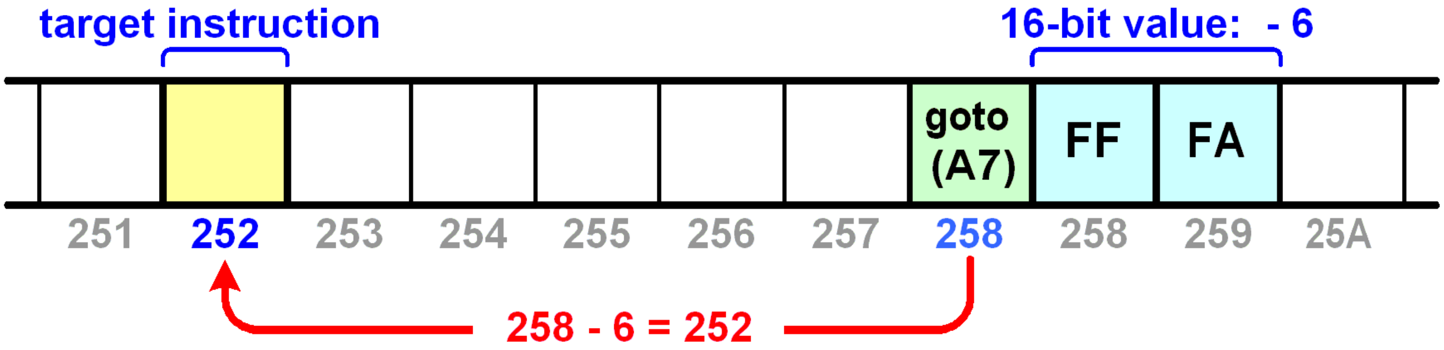
16-bit value: +7

target instruction

goto (A7) | 00 | 07

101 102 103 104 105 106 107 108 108 109 10A

102 + 7 = 109



target instruction

16-bit value: - 6

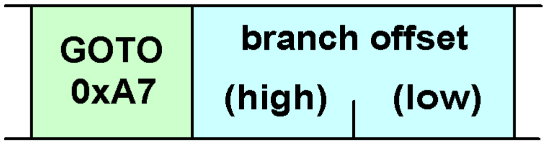
goto (A7) | FF | FA

251 252 253 254 255 256 257 258 258 259 25A

258 - 6 = 252

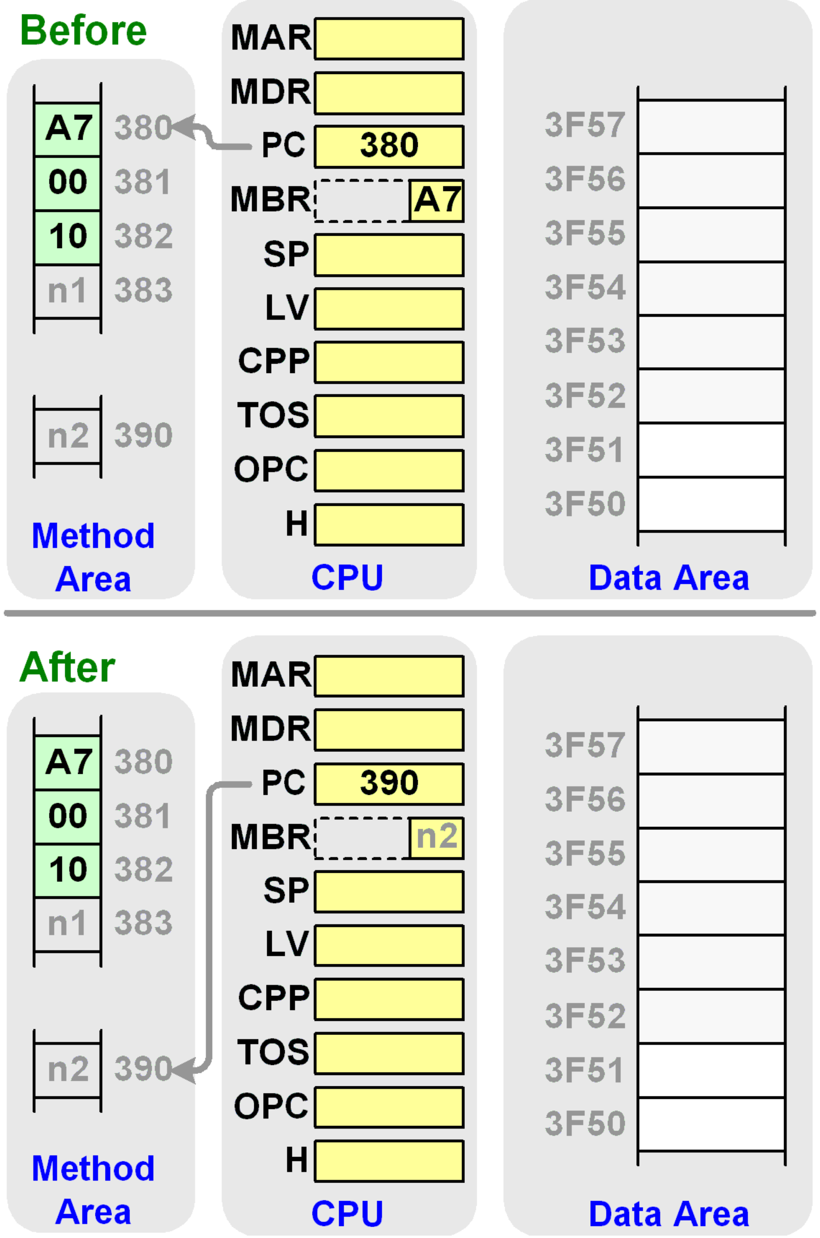
Microcode for the GOTO Instruction

The format of the GOTO instruction is as discussed above:



This microinstructions for GOTO must do the following work:

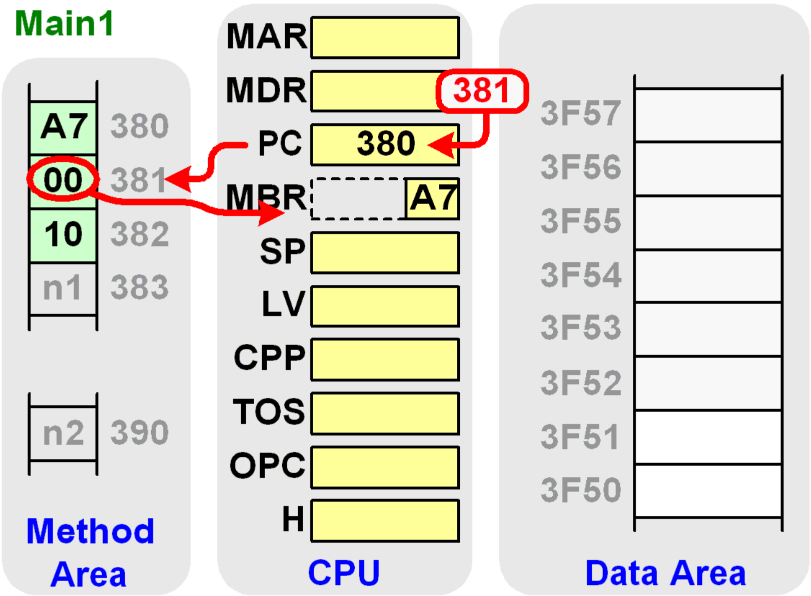
- Save the address of the GOTO opcode, since that is what we need to add the offset to
- Fetch the and assemble the 16-bit branch offset from the two operand bytes in the instruction
- Add the offset to the saved address of the GOTO opcode
- Fetch the opcode of the instruction at the new PC address (“n2” in the diagram)



Microcode for the GOTO Instruction

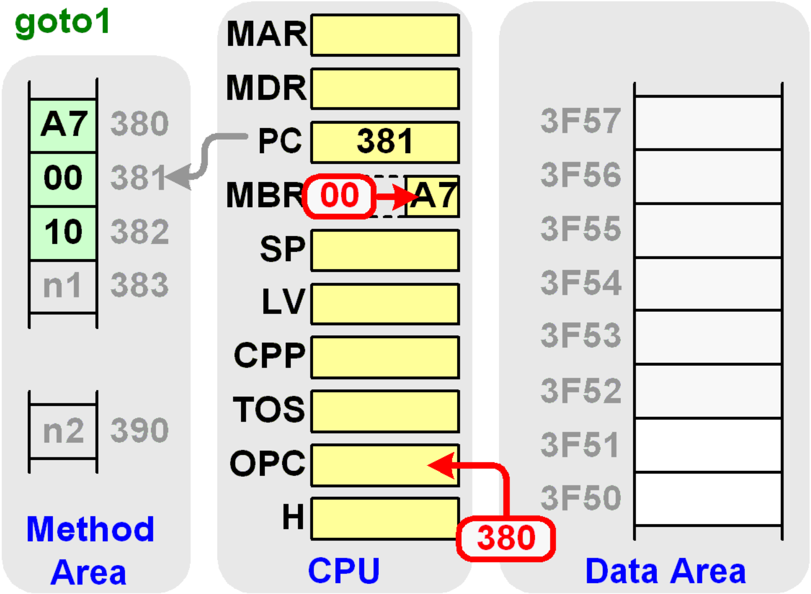
Main1 PC = PC + 1; fetch; goto (MBR)

- Increments the PC register (it changes from 380 to 381 in the example)
- Fetches the next byte – this is the low order part of the branch offset. The byte is read but it doesn't arrive in the MBR register until the next clock cycle.
- The "goto (MBR)" transfers control to the "goto1" microinstruction because the opcode is hex A7 and "goto1" is stored at control store address hex A7.



goto1 OPC = PC – 1

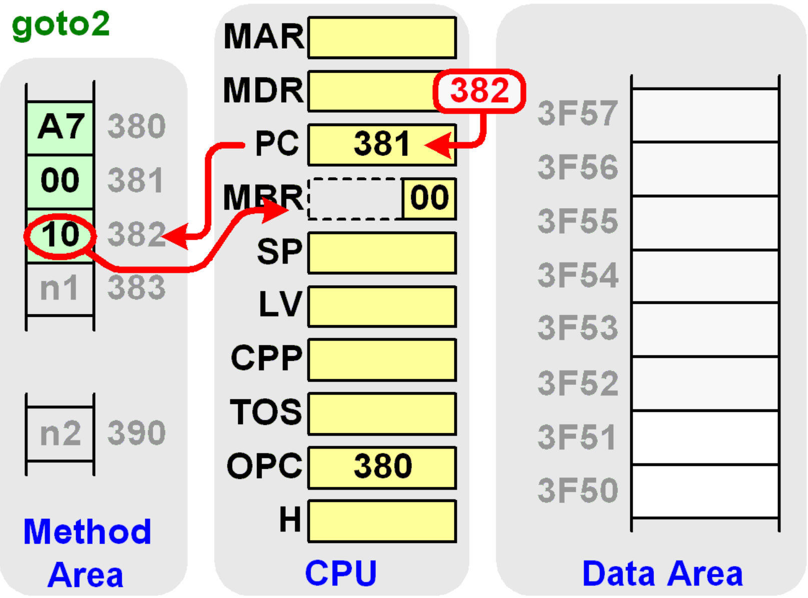
- Calculates the address of the GOTO opcode (at address 380) and saves it in the OPC register. We need to keep a copy of this address because the PC register will change as we fetch additional bytes from the instruction stream and we need the opcode address in order to add the branch offset to it.
- The low order byte of the branch offset (00) fetched in "Main1" is put into the MBR register.



Microcode for the GOTO Instruction

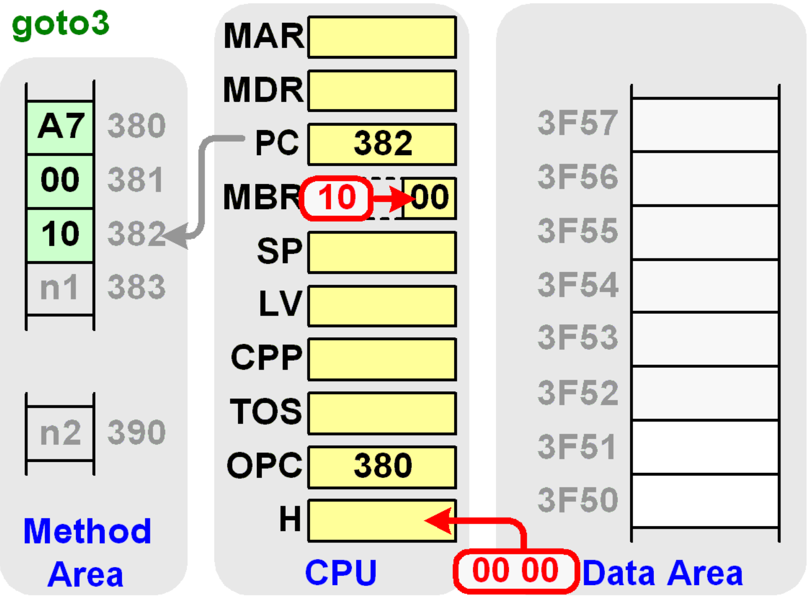
goto2 PC = PC + 1; fetch

- Increments the PC register (it changes from 381 to 382 in this example)
- Fetches the instruction byte that the PC register now points to. This is the high order byte of the branch offset (10). The byte is fetched but it doesn't arrive in the MBR register until the end of the next clock cycle.



goto3 H = MBR << 8

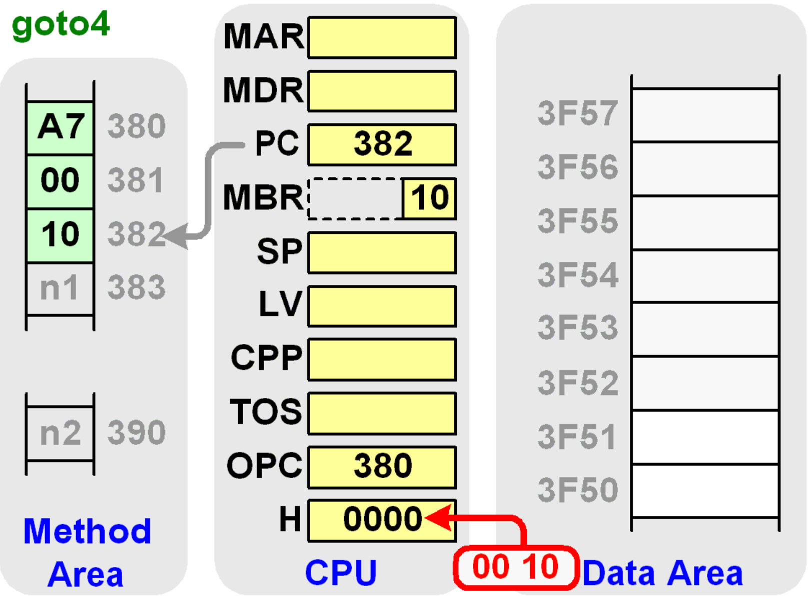
- Shifts the high order part of the branch offset (00 in the MBR register) to the left by 8 bits (resulting in hex 0000) and stores the result in H.
- The low-order part of the branch offset (10) that was fetched in "goto2" arrives in the MBR register during this clock cycle.



Microcode for the GOTO Instruction

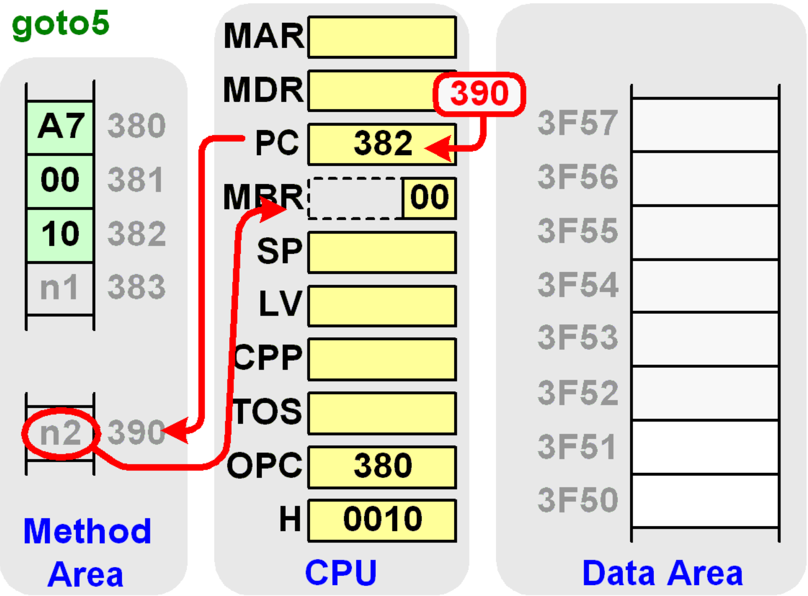
goto4 H = MBRU OR H

- Combines the low-order part of the branch offset (10 in MBR) with the high-order part (0000 in H) into a single value (0010) and stores the result in H. This is the signed value that will be added to the address of the GOTO opcode.



goto5 PC = OPC + H; fetch

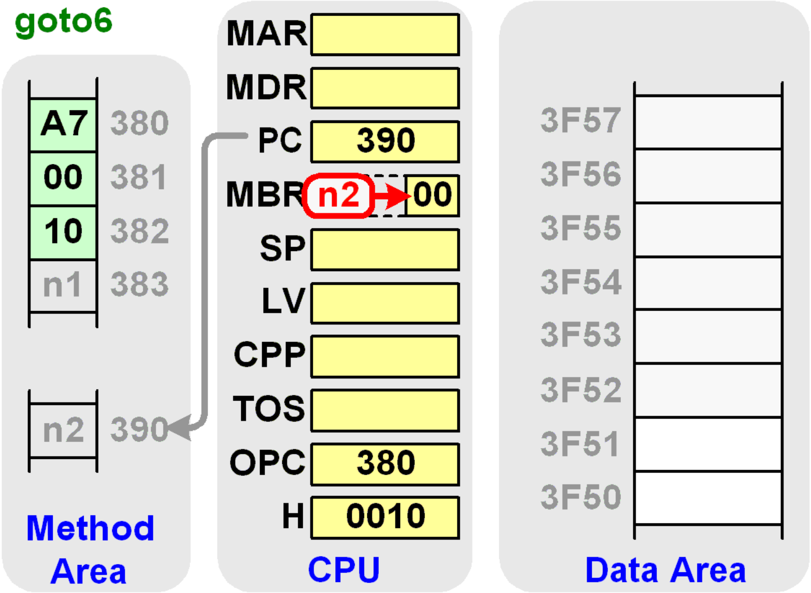
- Calculates the address of the next instruction by adding the GOTO opcode address (380 in OPC) to the branch offset (0010 in H).
- Initiates a fetch to read the opcode of the next instruction ("n2") into the MBR register. The opcode will arrive in the register at the end of the next clock cycle.



Microcode for the GOTO Instruction

goto6 goto Main1

- The opcode of the next instruction ("n2") fetched in "goto5" is put into the MBR register.
- Goes to Main1 to decode and execute the next instruction



Exercise 1 – The GOTO Instruction

In the microcode for the GOTO instruction:

```
Main1    PC = PC + 1; fetch; goto (MBR)
goto1    OPC = PC – 1
goto2    PC = PC + 1;  fetch
goto3    H = MBR << 8
goto4    H = MBRU OR H
goto5    PC = OPC + H;  fetch
goto6    goto Main1
```

The address of the next IJVM instruction to be executed is:

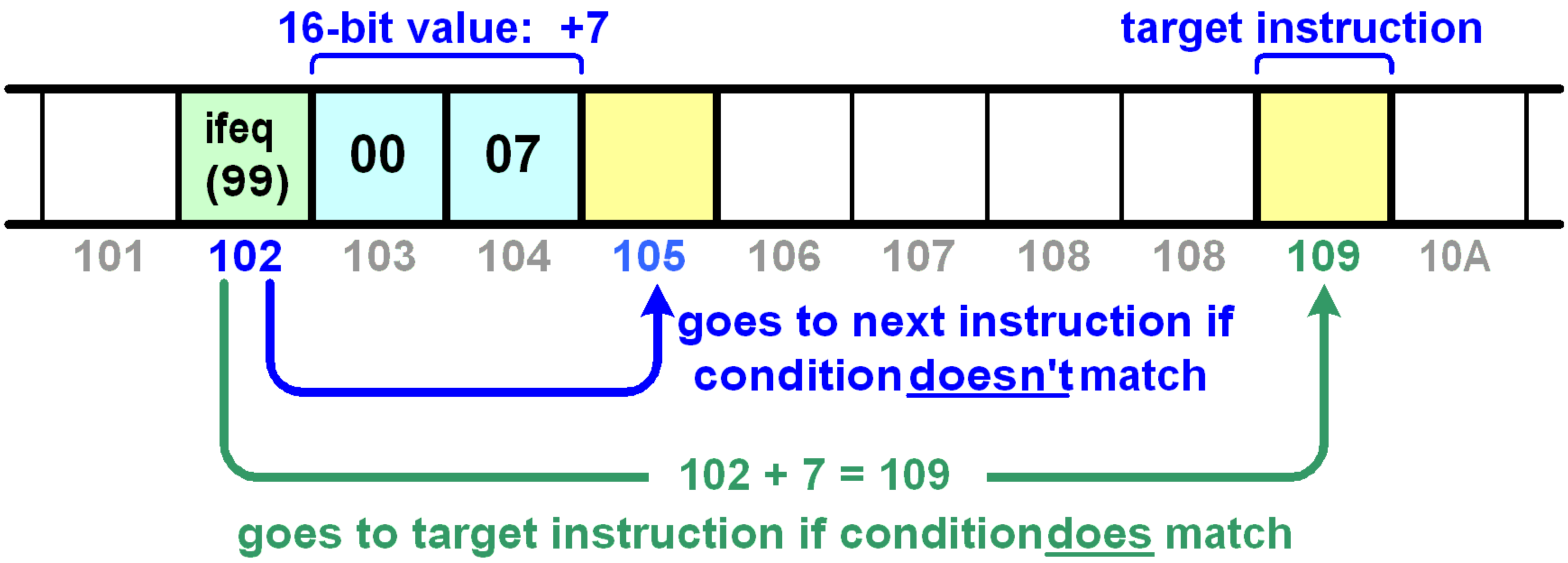
A – in the PC register

B – in the OPC register

C – Main1

Conditional Branch Instructions

The next instructions are conditional branch instructions. In general they work just like the GOTO instruction, but they have the option of branching or not branching depending on whether a condition is met or not, as shown by this “IFEQ” example:



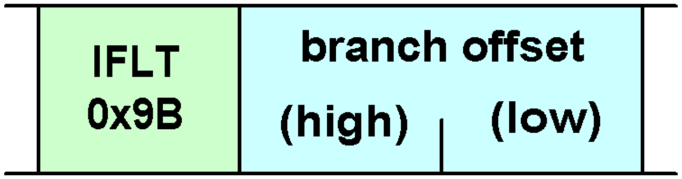
- If the condition is met, the instruction behaves just like a GOTO instruction by adding the given branch offset to the opcode address and putting the result in the PC register. This causes the target instruction to be executed next.

- If the condition is not met, then the instruction does not branch. When the conditional branch instruction finished, the instruction immediately following it is executed.

To show how these two choices work, we’ll show two instructions (IFEQ and IFLT) which (because of the sample data we’ve chosen) do and do not branch.

Microcode for the IFLT Instruction

The format of the IFLT instruction is as discussed above:



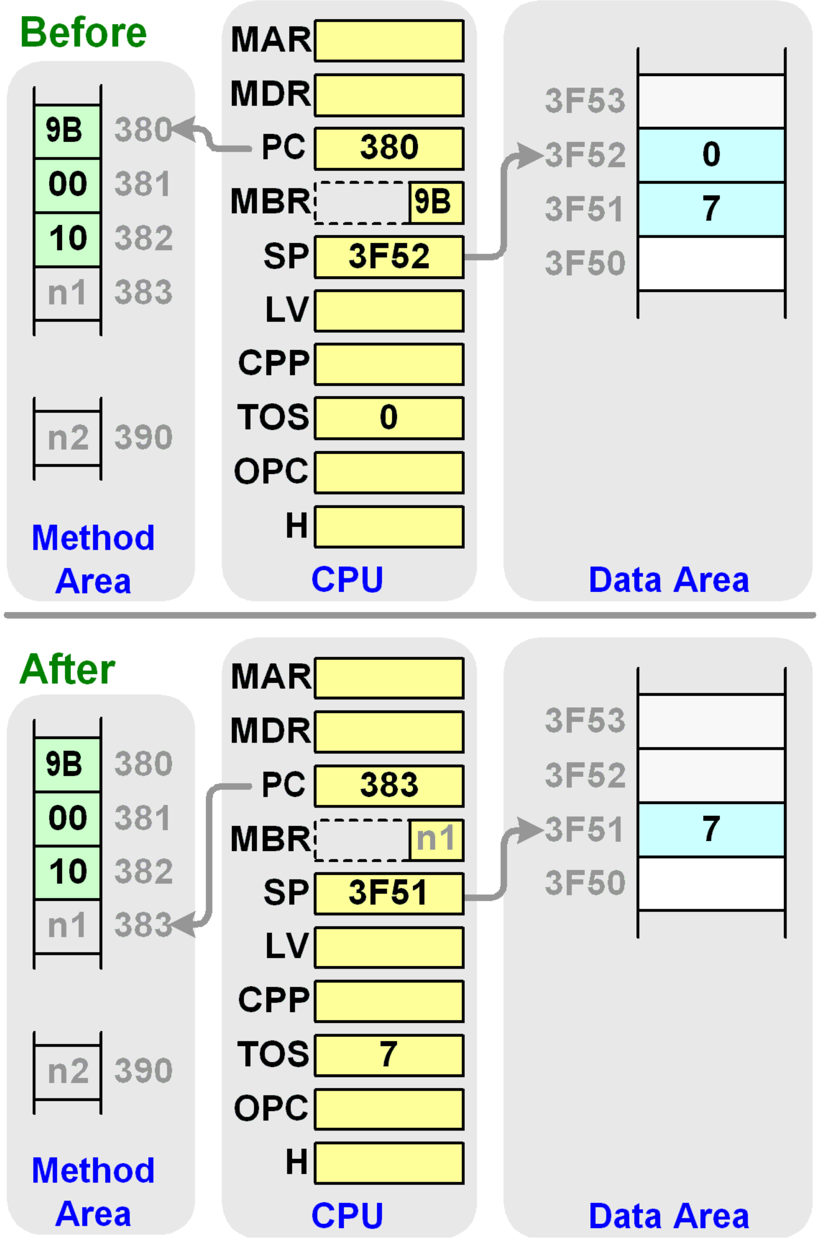
IFLT
0x9B

branch offset
(high) (low)

The instruction removes the top word from the stack and branches to the target address if the word is less than zero. In this example, the word is zero and so the instruction does not branch.

This microinstructions for IFLT must do the following work:

- Save the address of the IFLT opcode, since that is what we need to add the offset to.
- Pop a word off the stack, read the new top of stack value and put it into the TOS register.
- Check the word to see if it is less than zero.
- Since the word in this example is not less than zero, then the opcode of the next instruction must be fetched ("n1" in the diagram)



Before

Method Area
9B 380
00 381
10 382
n1 383
n2 390

CPU
MAR
MDR
PC 380
MBR 9B
SP 3F52
LV
CPP
TOS 0
OPC
H

Data Area
3F53
3F52 0
3F51 7
3F50

After

Method Area
9B 380
00 381
10 382
n1 383
n2 390

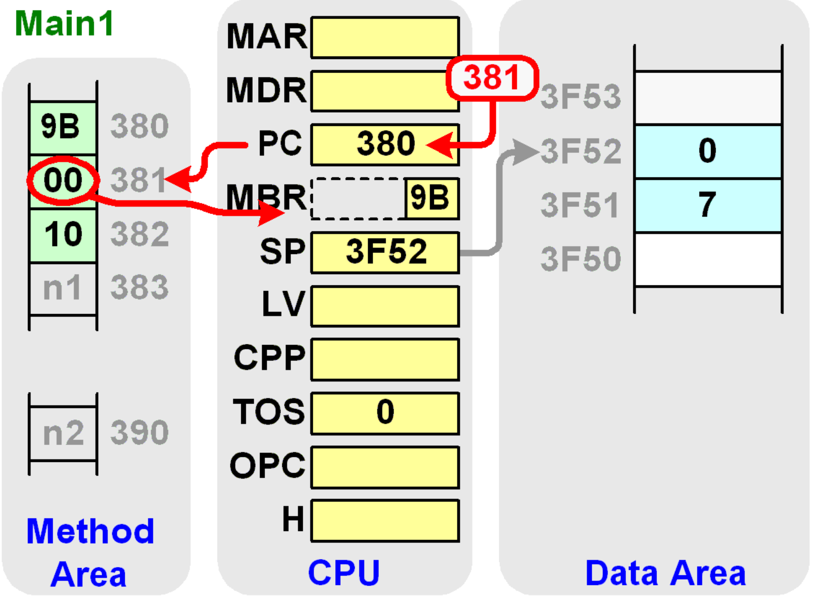
CPU
MAR
MDR
PC 383
MBR n1
SP 3F51
LV
CPP
TOS 7
OPC
H

Data Area
3F53
3F52
3F51 7
3F50

Microcode for the IFLT Instruction

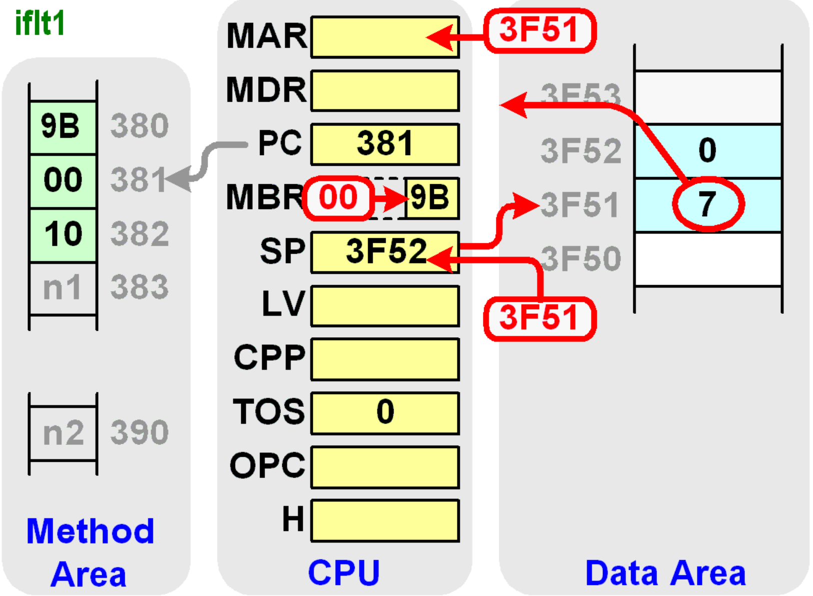
Main1 PC = PC + 1; fetch; goto (MBR)

- Increments the PC register (it changes from 380 to 381 in the example)
- Fetches the next byte – this is the low order part of the branch offset. The byte is read but it doesn't arrive in the MBR register until the next clock cycle.
- The "goto (MBR)" transfers control to the "iflt1" microinstruction because the opcode is hex 9B and "iflt1" is stored at control store address hex 9B.



iflt1 MAR = SP = SP – 1; rd

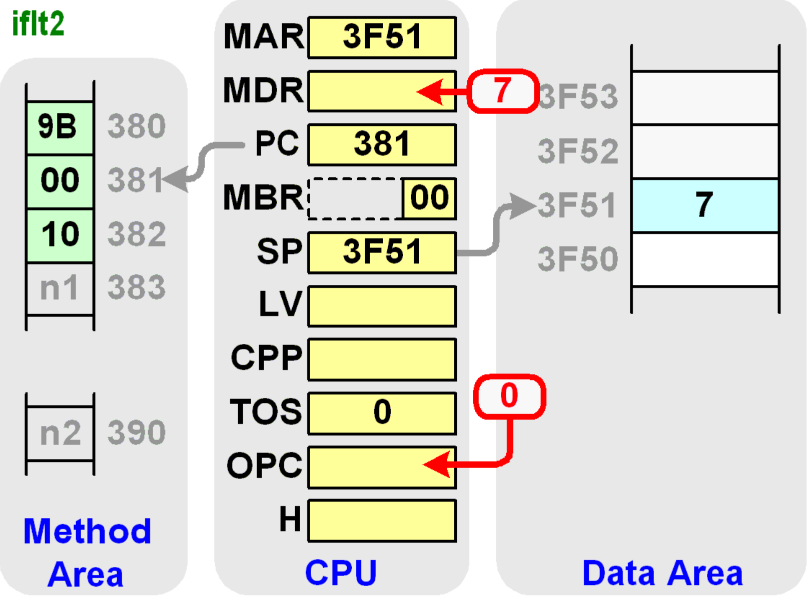
- Subtracts one from the stack pointer to get the address of the new top of stack
- Puts the address into MAR so that it can be read
- Initiates a read to read the new top of stack word from memory. The word arrives in the MDR register during the next clock cycle.
- The low order byte of the branch offset (00) fetched in "Main1" is put into the MBR register.



Microcode for the IFLT Instruction

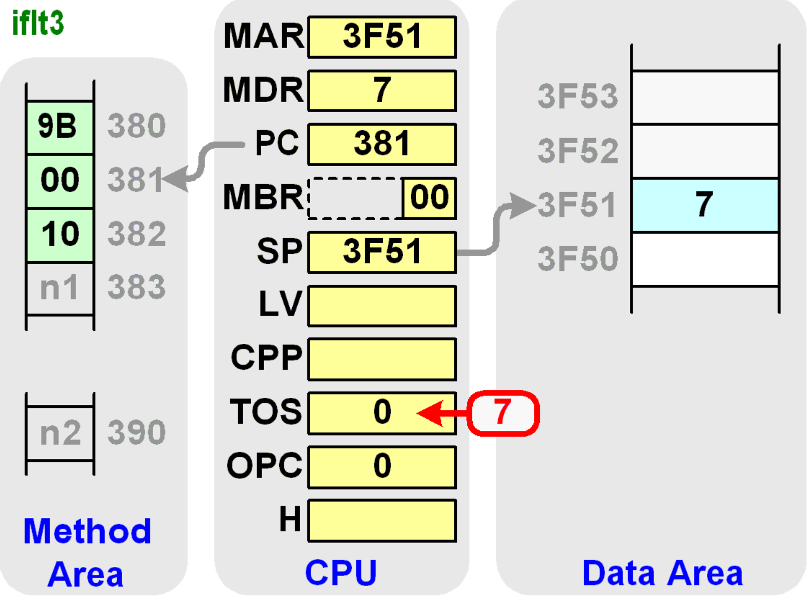
iflt2 OPC = TOS

- Saves a copy of TOS in the OPC register. We do this because we need to use the old TOS value to see if the instruction should jump or not, but we also have to put the new top of stack value in TOS.
- The new top of stack value read in "iflt1" arrives in the MDR register during this clock cycle.



iflt3 TOS = MDR

- Puts the new top of stack value into the TOS register.

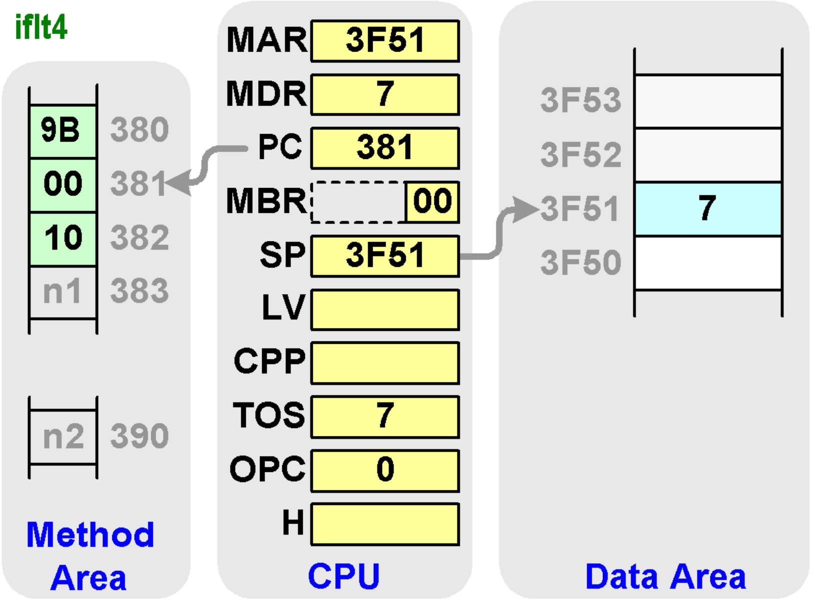


Microcode for the IFLT Instruction

iflt4 N = OPC; if (N) goto T; else goto F

- **Tests the value in the OPC register to see if it is negative (but doesn't store the result)**
- **If negative, the next microinstruction will be T and the branch will be executed.**
- **In this example, OPC is not negative so the next microinstruction will be F**
- **Microinstructions T and F are hex 100 apart in the control store so that one or the other can be selected by the JAMN bit (the high bit of the next microinstruction address)**

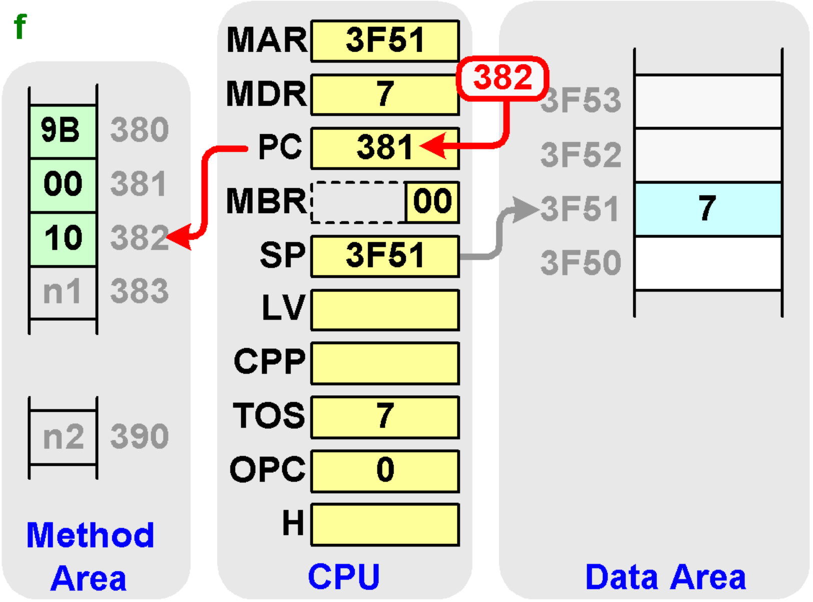
iflt4



f PC = PC + 1

- **Points the PC register to the second byte of the branch offset. Since we've arrived at this microinstruction we are not going to branch, and so the actual value of the branch offset can be ignored (note that we haven't fetched it).**

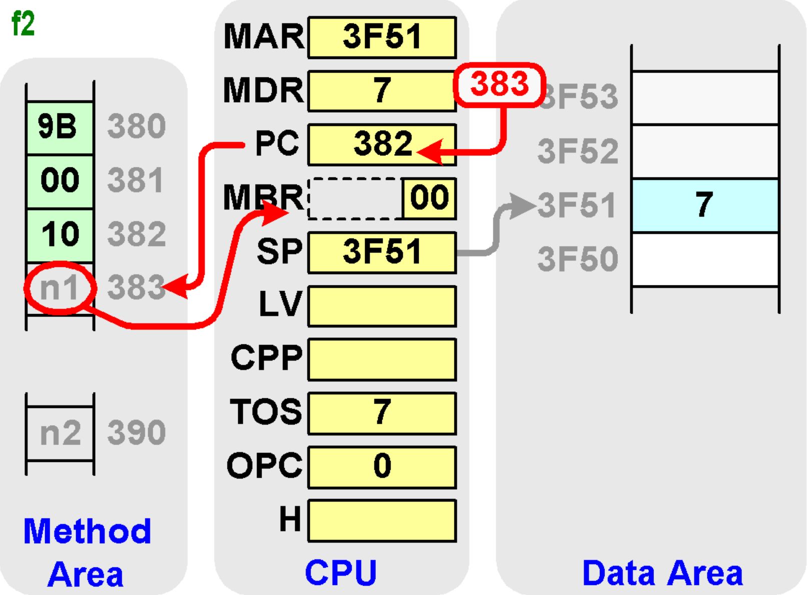
f



Microcode for the IFLT Instruction

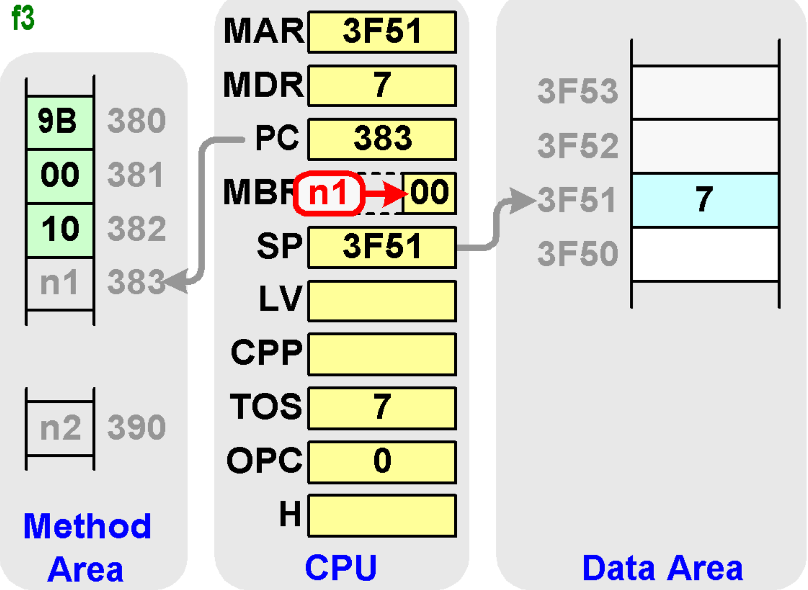
f2 PC = PC + 1; fetch

- Points the PC register to the opcode byte following the IFLT instruction ("n1" at address 383 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.



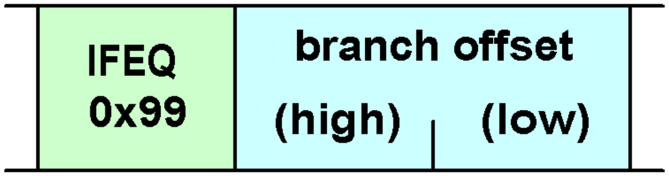
f3 goto Main1

- The opcode of the next instruction ("n2") fetched in "f2" is put into the MBR register.
- Goes to Main1 to decode and execute the next instruction



Microcode for the IFEQ Instruction

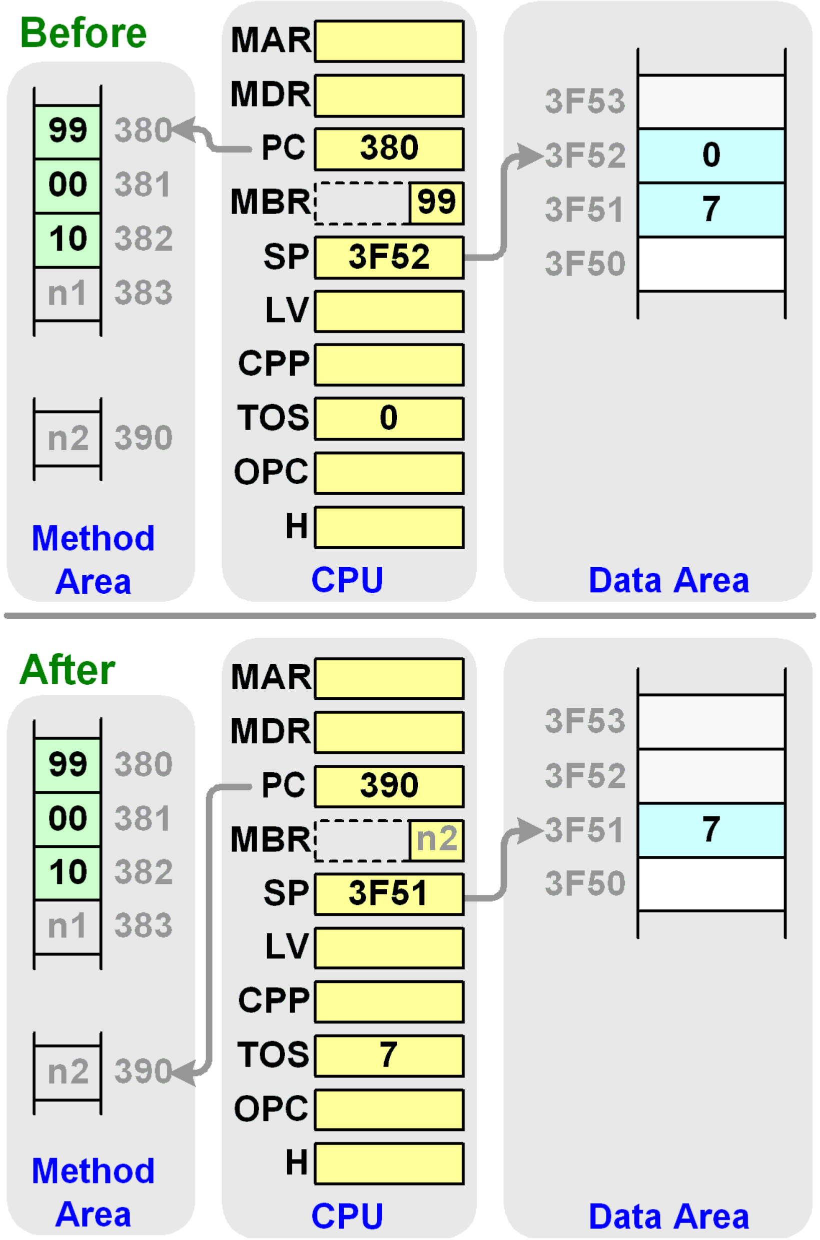
The format of the IFEQ instruction is as discussed above:



The instruction removes the top word from the stack and branches to the target address if the word is <u>equal zero</u>. In this example, the word is zero and so the instruction <u>does</u> branch.

This microinstructions for IFEQ must do the following work:

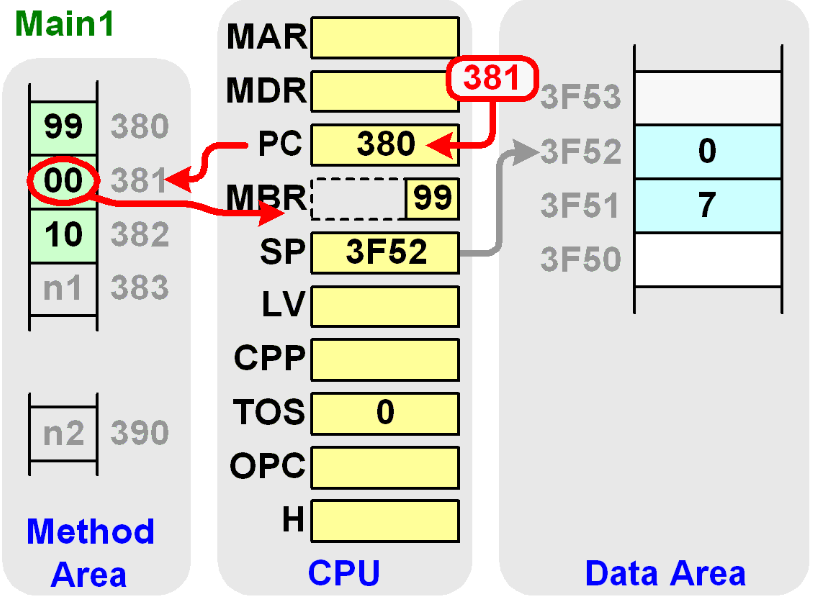
- Save the address of the IFEQ opcode, since that is what we need to add the offset to.
- Pop a word off the stack, read the new top of stack value and put it into the TOS register.
- Check the word to see if it is less than zero.
- Since the word in this example is less than zero, then the branch offset must be fetched and assembled, added to the IFEQ opcode address, stored in the PC register, and the opcode of the target instruction must be fetched ("n1" in the diagram)



Microcode for the IFEQ Instruction

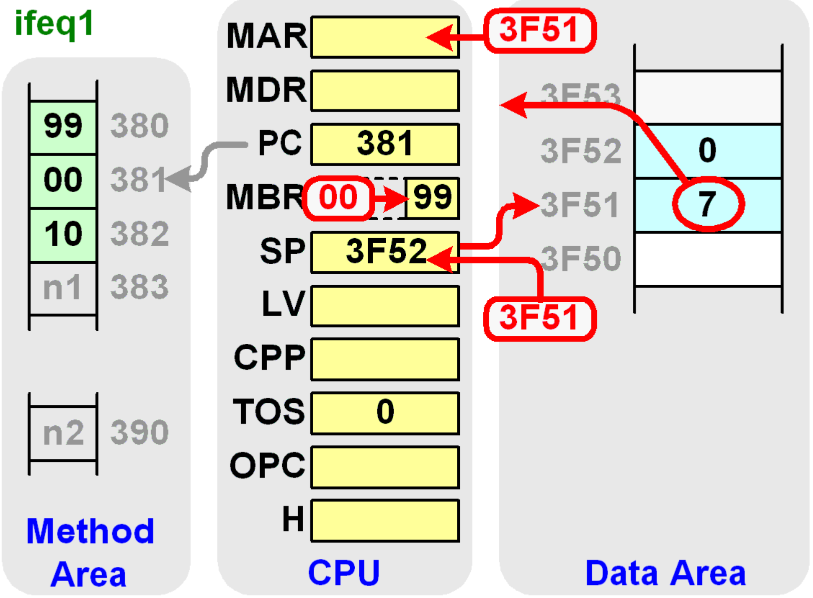
Main1 PC = PC + 1; fetch; goto (MBR)

- Increments the PC register (it changes from 380 to 381 in the example)
- Fetches the next byte – this is the low order part of the branch offset. The byte is read but it doesn't arrive in the MBR register until the next clock cycle.
- The "goto (MBR)" transfers control to the "ifeq1" microinstruction because the opcode is hex 99 and "ifeq1" is stored at control store address hex 99.



ifeq1 MAR = SP = SP – 1; rd

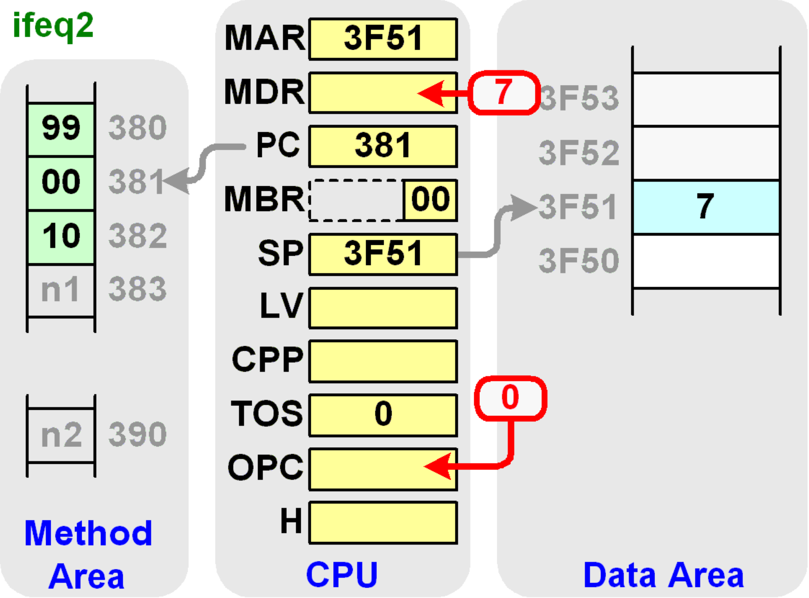
- Subtracts one from the stack pointer to get the address of the new top of stack
- Puts the address into MAR so that it can be read
- Initiates a read to read the new top of stack word from memory. The word arrives in the MDR register during the next clock cycle.
- The low order byte of the branch offset (00) fetched in "Main1" is put into the MBR register.



Microcode for the IFEQ Instruction

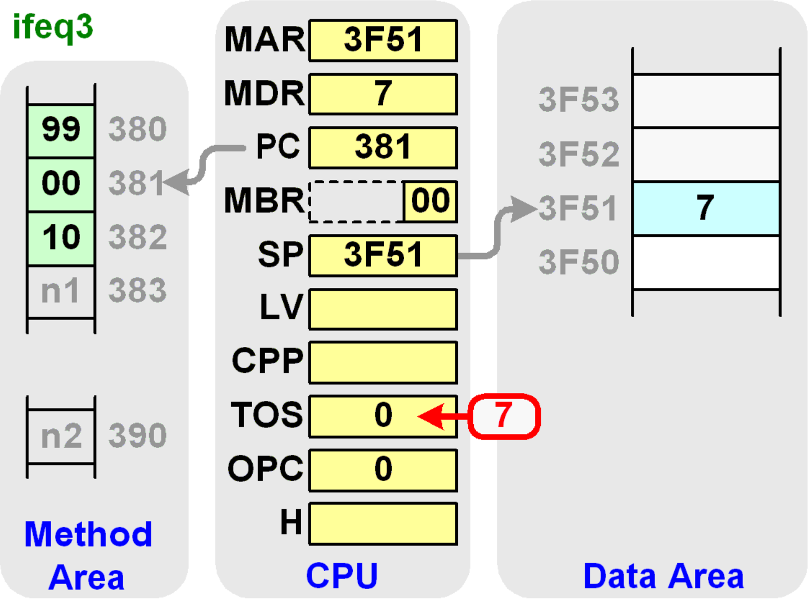
ifeq2 OPC = TOS

- Saves a copy of TOS in the OPC register. We do this because we need to use the old TOS value to see if the instruction should jump or not, but we also have to put the new top of stack value in TOS.
- The new top of stack value read in "ifeq1" arrives in the MDR register during this clock cycle.



ifeq3 TOS = MDR

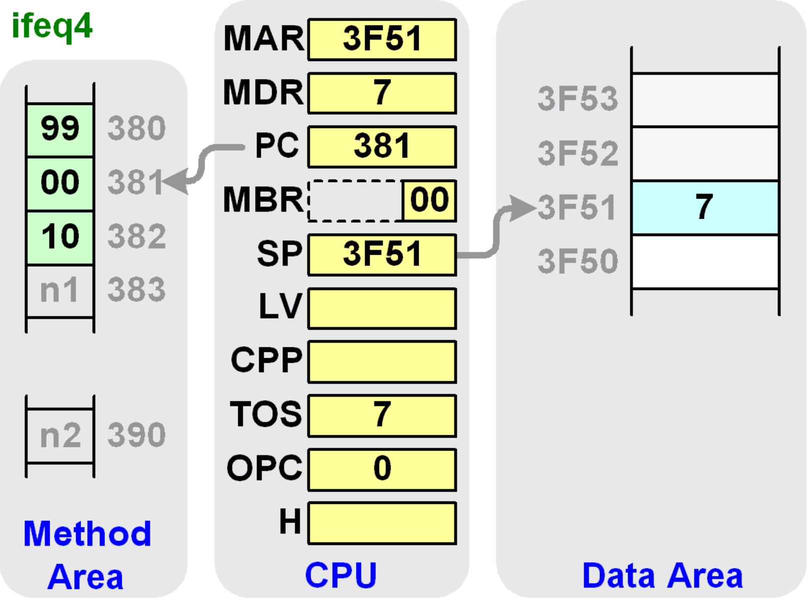
- Puts the new top of stack value into the TOS register.



Microcode for the IFEQ Instruction

ifeq4 Z = OPC; if (Z) goto T; else goto F

- **Tests the value in the OPC register to see if it is zero (but doesn't store the result)**
- **If nonzero, the next microinstruction will be F and the branch will not be executed.**
- **In this example, OPC is zero so the next microinstruction will be T**
- **Microinstructions T and F are hex 100 apart in the control store so that one or the other can be selected by the JAMN bit (the high bit of the next microinstruction address)**

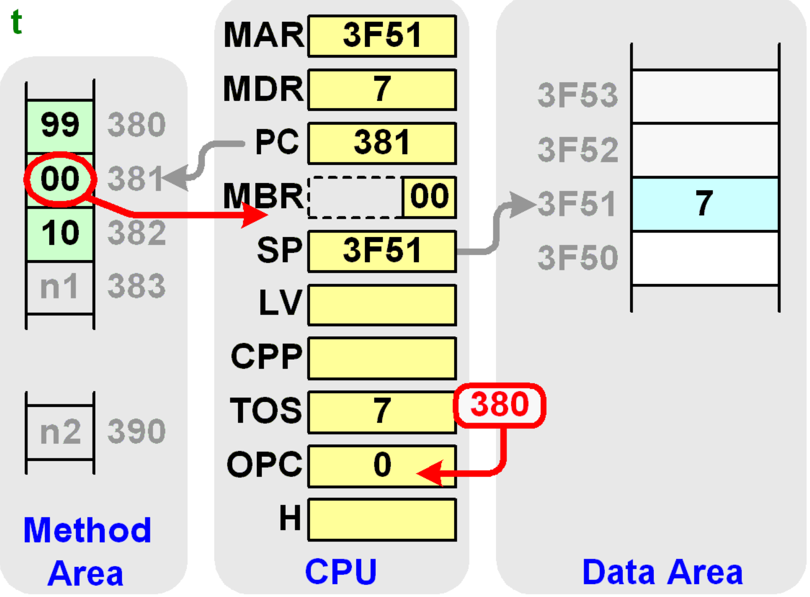


---

t OPC = PC – 1; fetch; goto goto2

- **Calculates the address of the IFEQ opcode and stores it in the OPC register. We'll need to add the branch offset to this value to calculate where the target instruction is.**

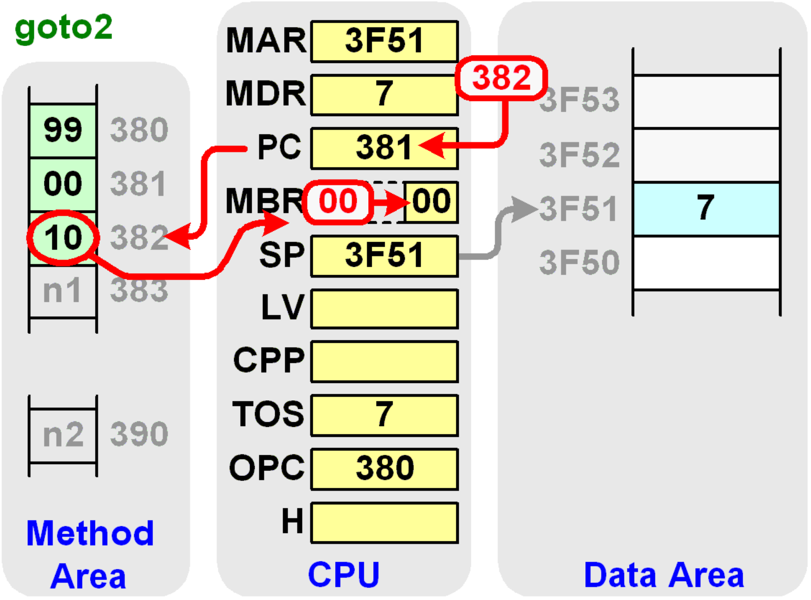
  The remainder of the work that we need to do in order to branch is identical to what the "GOTO" instruction does, so we branch to "goto2" in order to do that work.



Microcode for the IFEQ Instruction

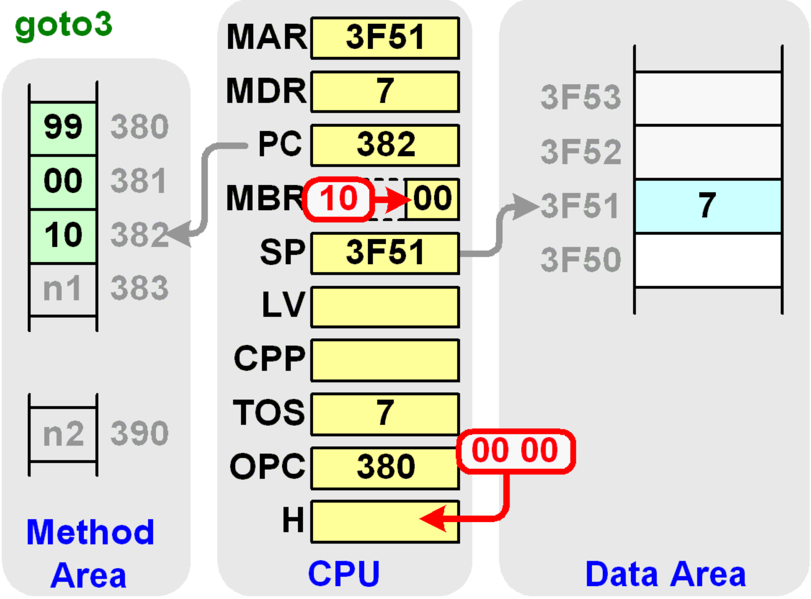
goto2 PC = PC + 1; fetch

- Increments the PC register (it changes from 381 to 382 in this example)
- Fetches the instruction byte that the PC register now points to. This is the high order byte of the branch offset (10). The byte is fetched but it doesn't arrive in the MBR register until the end of the next clock cycle.



goto3 H = MBR << 8

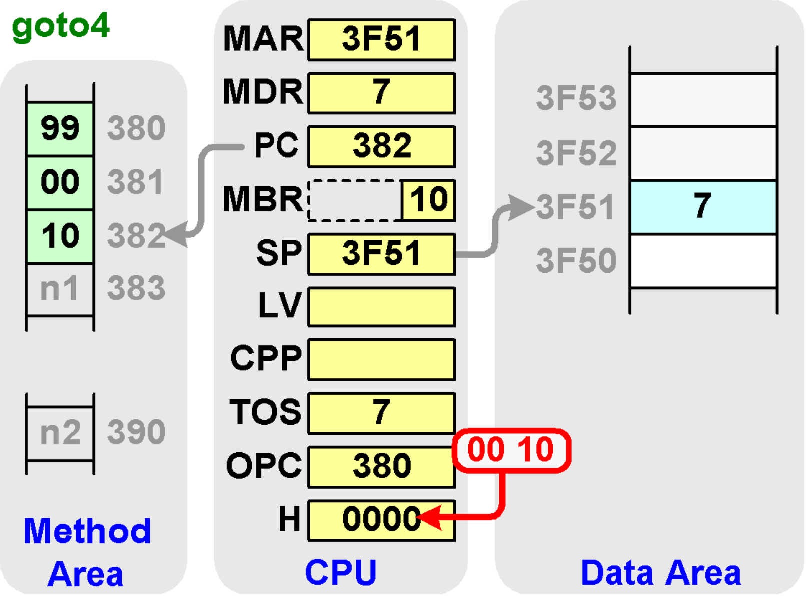
- Shifts the high order part of the branch offset (00 in the MBR register) to the left by 8 bits (resulting in hex 0000) and stores the result in H.
- The low-order part of the branch offset (10) that was fetched in "goto2" arrives in the MBR register during this clock cycle.



Microcode for the IFEQ Instruction

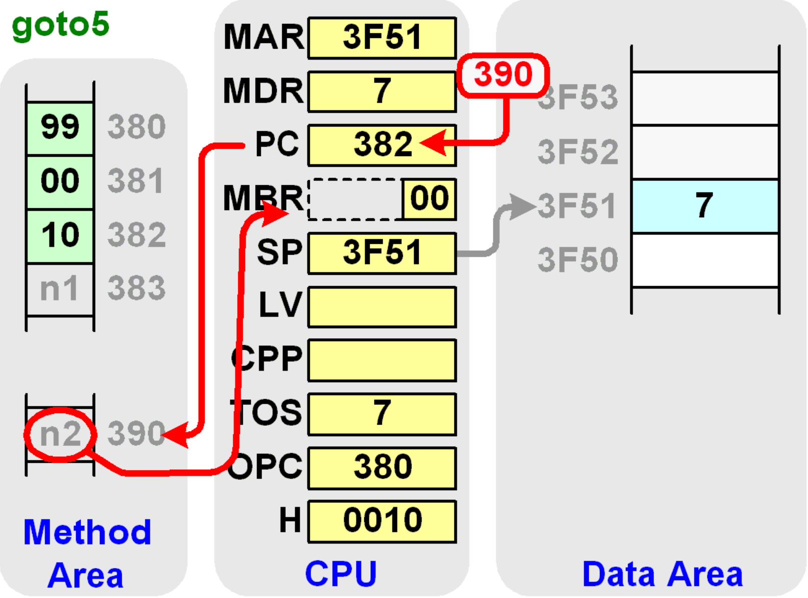
goto4 H = MBRU OR H

- Combines the low-order part of the branch offset (10 in MBR) with the high-order part (0000 in H) into a single value (0010) and stores the result in H. This is the signed value that will be added to the address of the GOTO opcode.



goto5 PC = OPC + H; fetch

- Calculates the address of the next instruction by adding the GOTO opcode address (380 in OPC) to the branch offset (0010 in H).
- Initiates a fetch to read the opcode of the next instruction (“n2”) into the MBR register. The opcode will arrive in the register at the end of the next clock cycle.

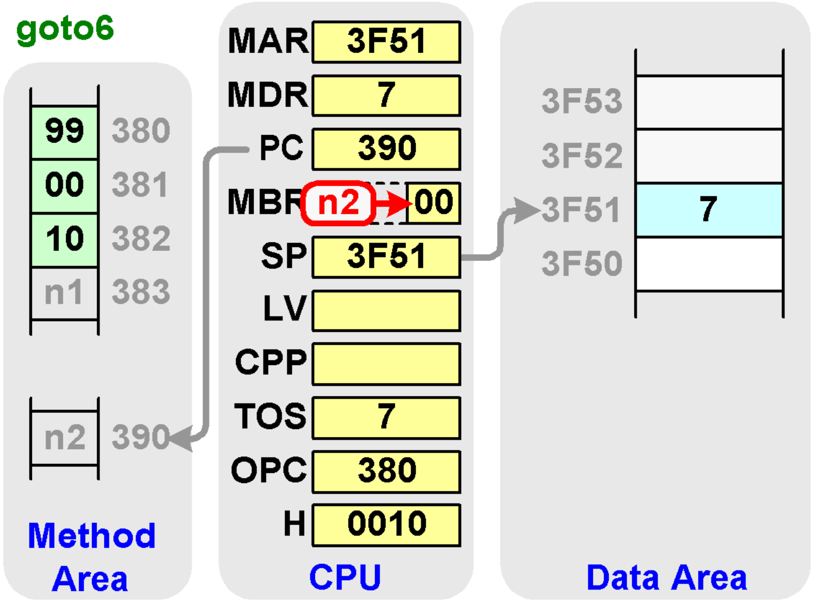


Microcode for the IFEQ Instruction

4.3.2

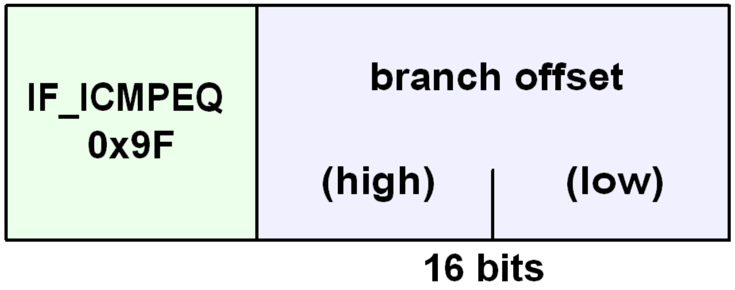
goto6 goto Main1

- **The opcode of the next instruction ("n2") fetched in "goto5" is put into the MBR register.**
- **Goes to Main1 to decode and execute the next instruction.**



The IF_ICMPEQ Instruction

The format of the IF_ICMPEQ instruction is similar to the other conditional branch instructions:



IF_ICMPEQ
0x9F

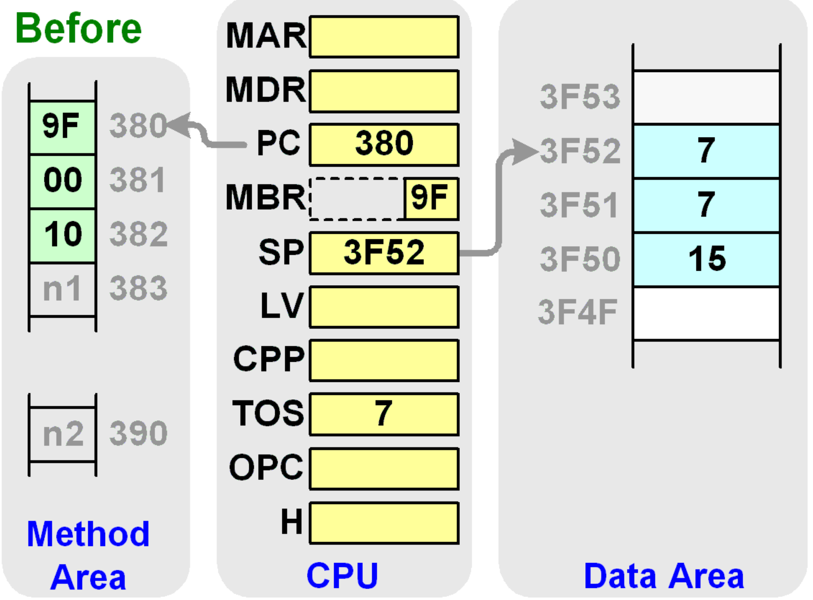
branch offset

(high) (low)

16 bits

IF_ICMPEQ pops two words off the stack and branches if they are equal.

The microcode for this instruction is very similar to the other conditional branch instructions and so we won't bother to examine it in detail.

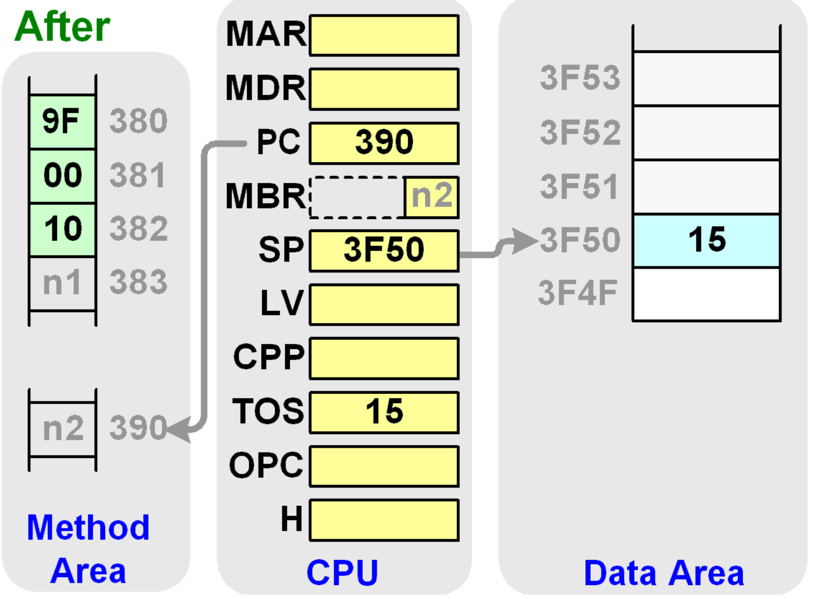


Before

Method Area: 9F 380, 00 381, 10 382, n1 383, n2 390

CPU: MAR, MDR, PC 380, MBR 9F, SP 3F52, LV, CPP, TOS 7, OPC, H

Data Area: 3F53, 3F52 7, 3F51 7, 3F50 15, 3F4F



After

Method Area: 9F 380, 00 381, 10 382, n1 383, n2 390

CPU: MAR, MDR, PC 390, MBR n2, SP 3F50, LV, CPP, TOS 15, OPC, H

Data Area: 3F53, 3F52, 3F51, 3F50 15, 3F4F

MIC-1 Summary

4.3.2

We’ve shown that with relatively simple components we can build a system which is able to execute an ISA-level computer program. We’ve only implemented a couple of dozen instructions, but those instructions have all the basic capabilities we need to run a computer program.

This basic system would be similar in design and capabilities to the one-chip CPUs in widespread use about the time that the IBM PC was introduced in the early 1980’s.

Modern processors are much more sophisticated because we can now put so many more transistors on a single silicon chip.   The extra transistors in modern processors are used to:

- Increase the capabilities of the system (for example, add hardware multiply / divide, floating point support, enhanced memory capabilities and exception handling, etc.)
- Improve the performance of the system

Next we’ll look at some improved MIC microarchitectures which provide better performance than the MIC-1, and also discuss advanced performance enhancement techniques.

Exercise 2 – MIC-1 Summary

Which of the following statements is true?

- **A** When an IJVM instruction finishes executing, the microcode must leave the PC register pointing to start of the instruction right after it in the method area
- **B** Every IJVM instruction is implemented by a set of microinstructions which cause the data path to perform the required operations
- **C** The TOS register must always contain the address of the memory word at the top of the stack
- **D** A value stored in memory must first be read into one of the CPU registers before a microinstruction can use it as part of a data cycle operation.
- **E** The JMPC bit in the microinstruction word is used to perform a conditional jump to one of two different microinstructions.
- **F** All of the above
- **G** B and D above
- **H** A, B, and D above

Improving Performance

The MIC-1 microarchitecture used a very simple hardware and microcode design to execute IJVM instructions. It's the type of design that would have been typical of a microprocessor built around 1980, such as the Intel 8088 used in the original IBM PC.

As IC technology has advanced, more and more transistors are able to be packed onto a single-chip CPU. This allows designers to build more complex CPUs. The additional complexity can be used to either:

Improve performance **or** **Add functionality**

Now we're going to turn our attention to various ways to improve performance. Before we do, though, here some typical items of functionality that have been added to microprocessors since the 1970's:

- **Larger word sizes (from 16 -> 32 -> 64 bit words)**
- **Hardware multiply / divide units**
- **Floating point capabilities**
- **Protected execution modes that permit the OS to shield the system from programming errors or malicious code**
- **Sophisticated memory mapping techniques used for virtual memory**

Improving Performance

Over the past few decades, the greatest performance gains have come from increased speed of the basic transistors themselves. As transistors and the distances between them shrink, so do the propagation delays. This allows clock speeds to be raised and the basic cycle time of the machine to be reduced.

But what we're going to focus on is how changes to the microarchitecture (i.e., how the transistors are organized) can improve performance. For a CPU built from transistors of a certain speed, the overall performance of the system can be improved in three basic ways:

1. Speeding up the data path so that the clock cycle can be shorter
   The more work that the data path has to do per clock cycle, the longer the clock cycle needs to be in order to make sure the results can always be ready on the C bus before the end of the clock cycle. If we can simplify the data path then we can shorten the clock cycle and speed up the machine.
2. Reduce the execution path length
   We've seen that the amount of time that it takes to execute an ISA level instruction is directly proportional to the number of microinstructions (i.e., clock cycles) that need to be executed. If we can find a way to reduce the number of microinstructions then we can improve the performance of the system.
3. Use parallelism to do different tasks at the same time.
   By breaking tasks up and doing several in parallel, the overall time required goes down.

Most of these techniques involve adding more transistors, and that equates to added cost (but the cost per transistor comes down with each new generation of technology).

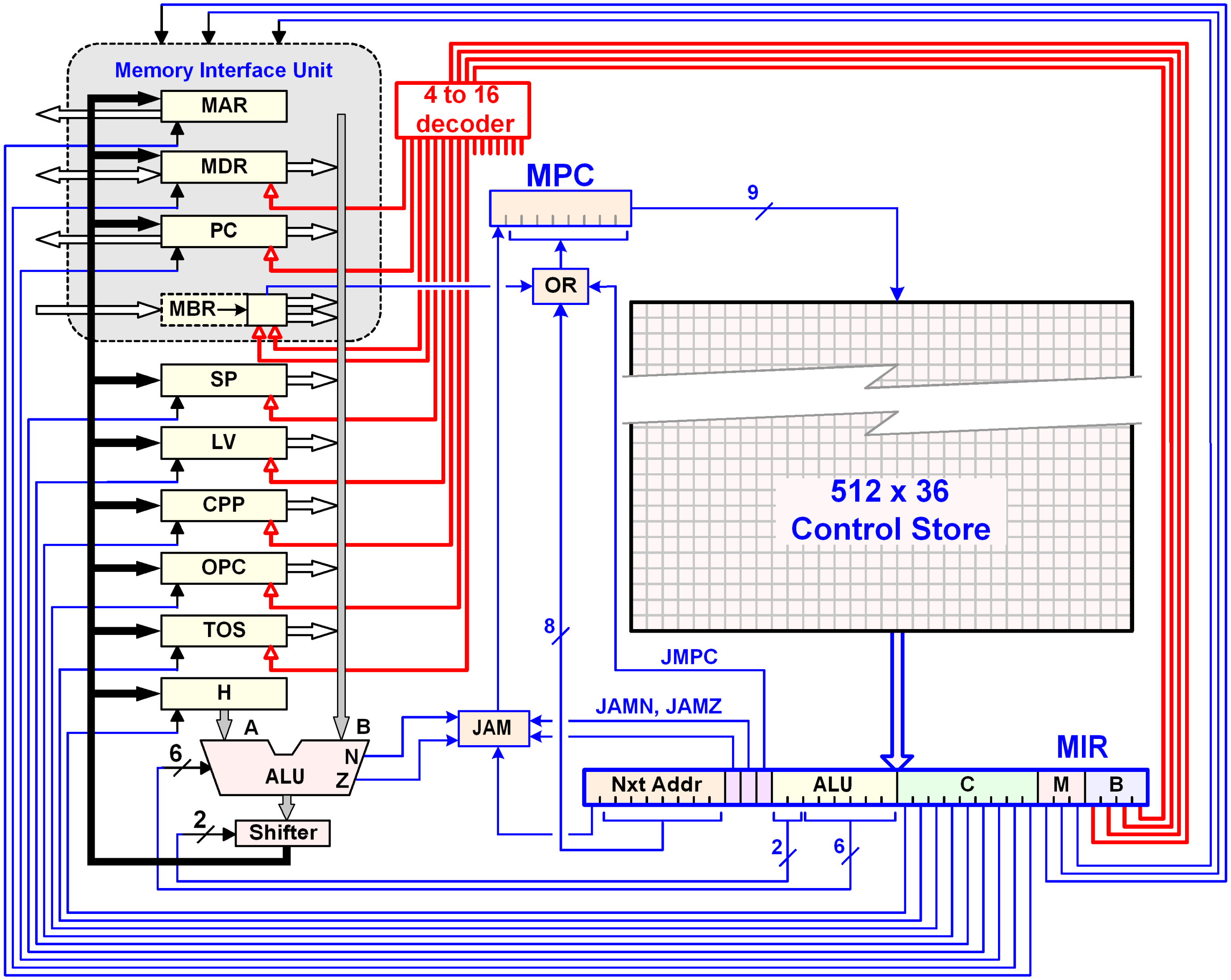
Speeding up the Data Path

As an example of how to speed up the data path, consider the decoder which selects which register to connect to the “B Bus”.

It takes a bit of time for this decoder to activate the selected register outputs.

We could eliminate this decoder by adding more bits to the microinstruction word so that there is one bit per register control line, just as for the “C” bus.

This would eliminate the time needed to decode the “B Bus” bits and would allow us to shorten the clock cycle.



But it would take extra bits in the micro instruction word and therefore in the control store, too.

Speeding up the Data Path

There are other things we could do to speed up the data path. For example, we could replace our simple ALU with a more complex one that produces results faster. We already saw how the performance of a simple adder could be improved by replacing it with a “Carry Select Adder” – there are many other techniques for speeding up an ALU as well.

But the faster the adder, the more transistors are required. For example, the Carry Select Adder requires almost twice as many AND, OR and XOR gates as a normal adder does. So there is a trade-off between speed and cost (because more transistors means a bigger chip, a bigger chip means less chips per wafer, and less chips per wafer means that the semiconductor manufacturer needs to charge more per chip to recoup it’s fabrication costs).

Reducing Execution Path Length

Merging the Interpreter Loop with the Microcode

There are a few different ways in which we can reduce the execution path length. The simplest (and cheapest) way is to simply find a way to use fewer microinstructions. We can do this for some instructions by merging the "Main1" interpreter microinstruction into each sequence of microinstructions.

Here's an example of how we could do this for the microinstructions for the IJVM "POP" instruction.

Original IJVM-1 Microinstructions

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

Improved IJVM-1 Microinstructions

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

The original sequence includes a "no-operation" clock cycle ("pop2") which is needed because we have to wait for data to be read from memory.

The improved sequence uses this "dead" clock cycle to do some of the work required by the "Main1" interpretation microinstruction. As a result, we don't have to go back to "Main1" at the end of this sequence, we can simply jump directly to the set of microinstructions to execute the next IJVM instruction.

This performance improvement is free because we don't need to change any hardware at all to take advantage of it. But there aren't a lot of microinstruction sequences that have a "dead" clock cycle that we can take advantage of.

Reducing Execution Path Length

A Three-Bus Architecture

Another way to eliminate microinstructions is to add a third bus to our system as shown in this diagram. This allows us to add any two registers together in a single clock cycle, whereas before we needed an extra cycle to load one of the data values into the "H" register.

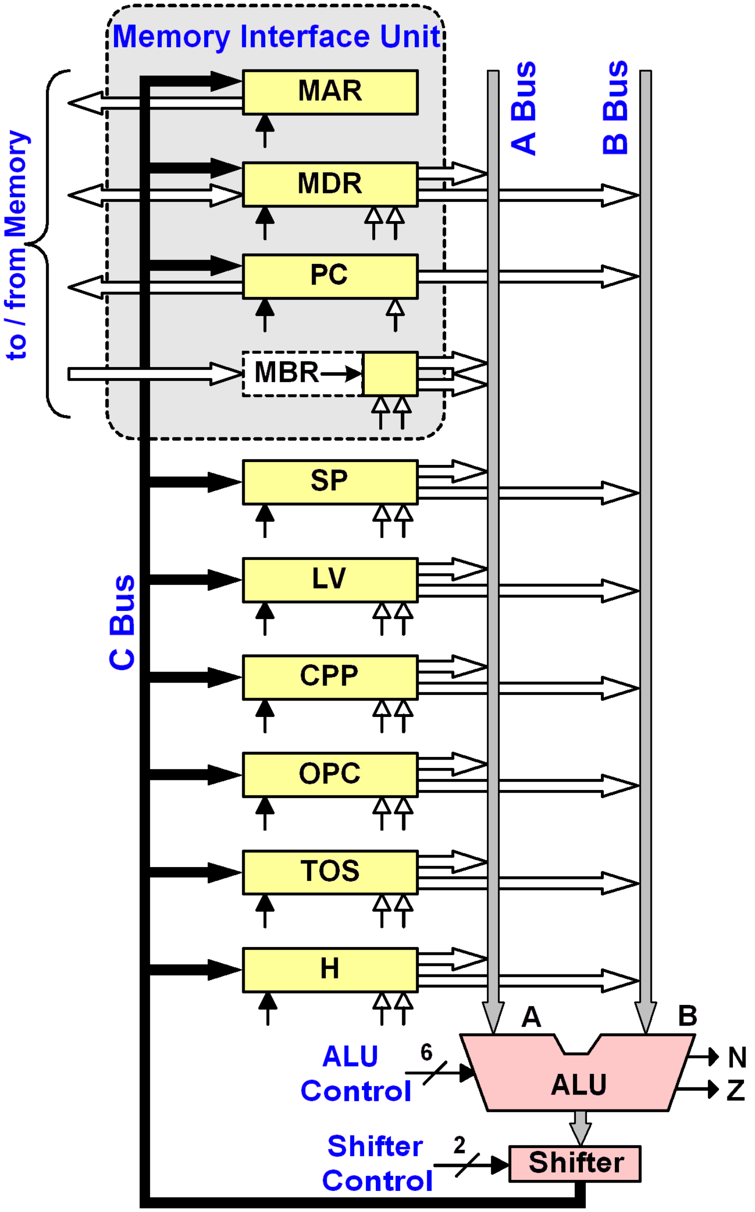
In order to add the second bus, we need to add a second output control signal to most of the registers. So, for example, the SP register now has two output control signals – one to connect it to the A bus and one to connect it to the B bus.

These extra control signals mean that this system would have more bits in it's microinstruction word, and this would in turn increase the size of the control store.

But the third bus lets us use microinstructions like these:

MAR = LV + MBRU

MDR = TOS = TOS + TOS



Exercise 3 – Three-Bus Architecture

A third bus requires a larger control store to hold the microinstructions because:

A – each microinstruction word will need to have more bits to supply the extra control signals that drive the connections to the third bus

B – more microinstruction words will be needed to supply the extra control signals that drive the connections to the third bus

C – two copies of each microinstruction word will be needed – one for the A bus and one for the B bus

D – all of the above

E – A and B above

F – A and C above

G – B and C above

H – none of the above

Reducing Execution Path Length

A Three-Bus Architecture

Let’s see how the third bus has helped by seeing how the microinstructions for ILOAD can take advantage of it:

Original

```
iload1    H = LV                           MBR contains index – copy LV to H
iload2    MAR = MBRU + H; rd               MAR = address of local variable to push
iload3    MAR = SP = SP + 1                SP points to new top of stack; prepare write
iload4    PC = PC + 1; fetch; wr           Inc PC, get next opcode; write top of stack
iload5    TOS = MDR;  goto Main1           Update TOS
Main1     PC = PC + 1;  fetch;  goto (MBR) MBR holds opcode; get next byte; dispatch
```

3-Bus Version

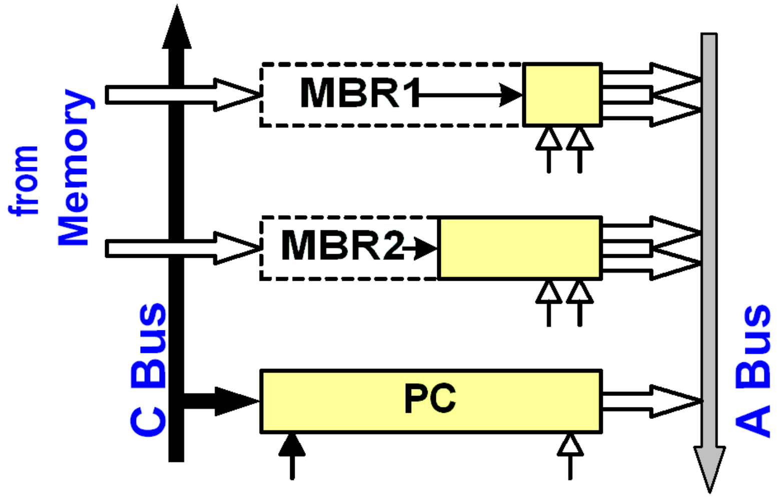
```
iload1    MAR = MBRU + LV; rd              MAR = address of local variable to push
iload2    MAR = SP = SP + 1                SP points to new top of stack; prepare write
iload3    PC = PC + 1; fetch; wr           Inc PC, get next opcode; write top of stack
iload4    TOS = MDR                        Update TOS
iload5    PC = PC + 1;  fetch;  goto (MBR) MBR already holds opcode; fetch next byte
```

The third bus has allowed us to eliminate one of the microinstructions from ILOAD.

There are lots of microinstruction sequences that add two values together that can benefit from this improvement.

Reducing Execution Path Length

An Instruction Fetch Unit



One of the things that takes time in every sequence of microinstructions is incrementing the PC register and fetching bytes from the instruction stream. IJVM instructions that contain operands have to do this multiple times, and IJVM instructions which have 16-bit opcodes need especially long sequences of microinstructions to assemble the individual bytes from the instruction stream.

We can include dedicated hardware called an “Instruction Fetch Unit” to automate these operations and eliminate the microinstructions needed to do the work. The instruction fetch unit will logically look like the diagram at the right. It includes the following registers:

MBR1 – The IFU will automatically put the memory byte that the PC register points to into this register.

MBR2 – The IFU automatically puts the next two memory bytes that the PC register points to into this register (one of these will be the same as the byte in MBR1). This makes it easy for the microcode to read 16-bit operands from the instruction stream.

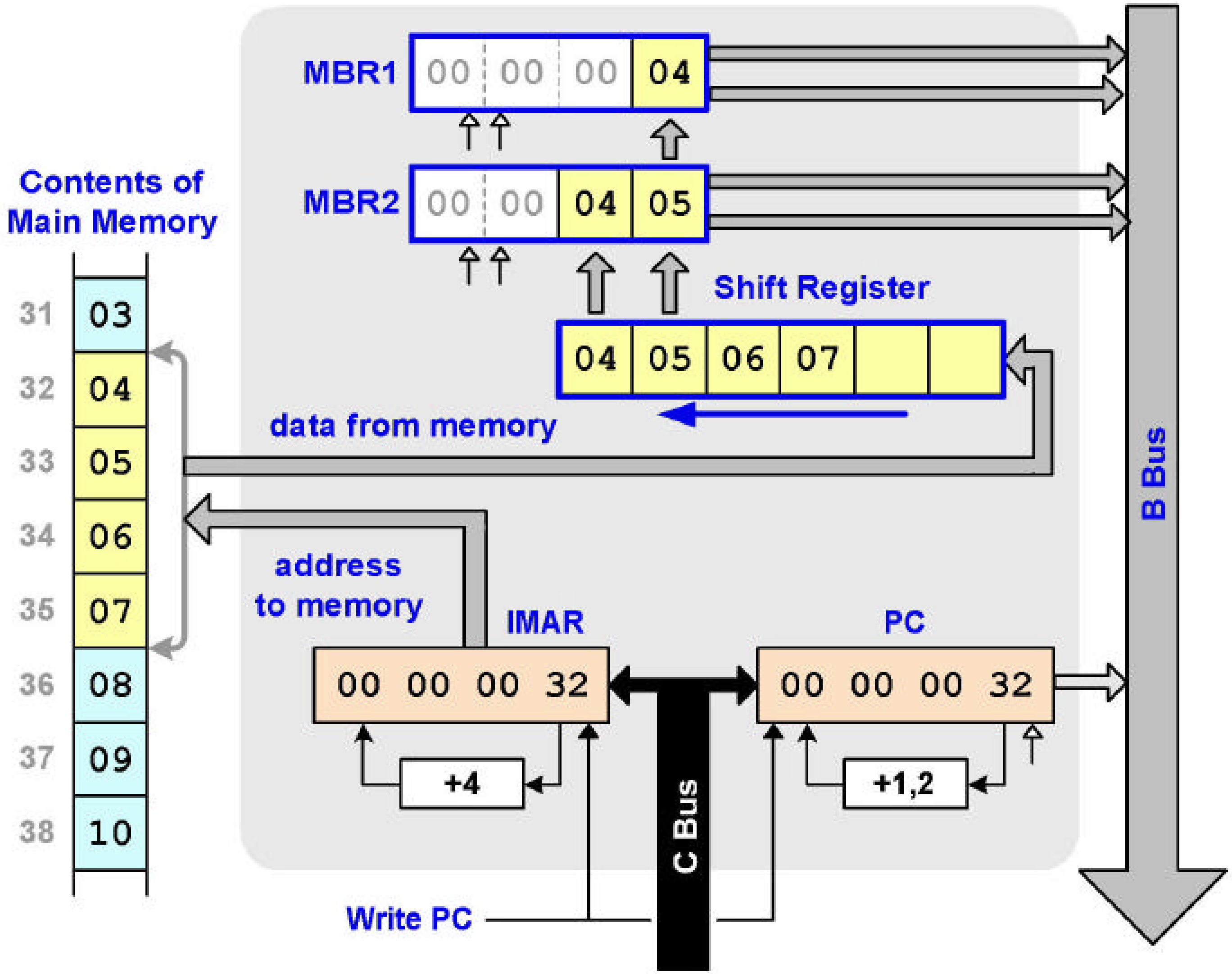
PC – Every time the microcode uses the value in the MBR1 or MBR2 registers, the PC register will automatically be incremented by 1 or 2. The microcode will no longer have to use a clock cycle for incrementing the PC.

Note that MBR1 and MBR2 can both be connected to the A bus as unsigned values by referring to “MBR1U” or “MBR2U”.

Reducing Execution Path Length

An Instruction Fetch Unit

We're not going to show all of the circuitry needed to build an IFU, but here is a block diagram of the major pieces:



Reducing Execution Path Length

An Instruction Fetch Unit

Here is how the instruction fetch unit works:

- **Information is read from memory 4 bytes at a time and is put into a Shift Register.**
- **The first byte in the shift register is copied to the MBR1 register, and the first two bytes are copied to the MBR2 register.**
- **When data is read from MBR1 or MBR2, the remaining information in the Shift Register is shifted "left" by 1 or 2 bytes, loading the next byte(s) into these registers. At the same time, the PC is automatically incremented by 1 or 2 as appropriate. This is done by a special dedicated "incrementer" circuit.**
- **When the Shift Register has 4 or more empty bytes then the IMAR is incremented by 4 (using a second dedicated incrementer circuit) and a new set of data is loaded into the Shift Register.**

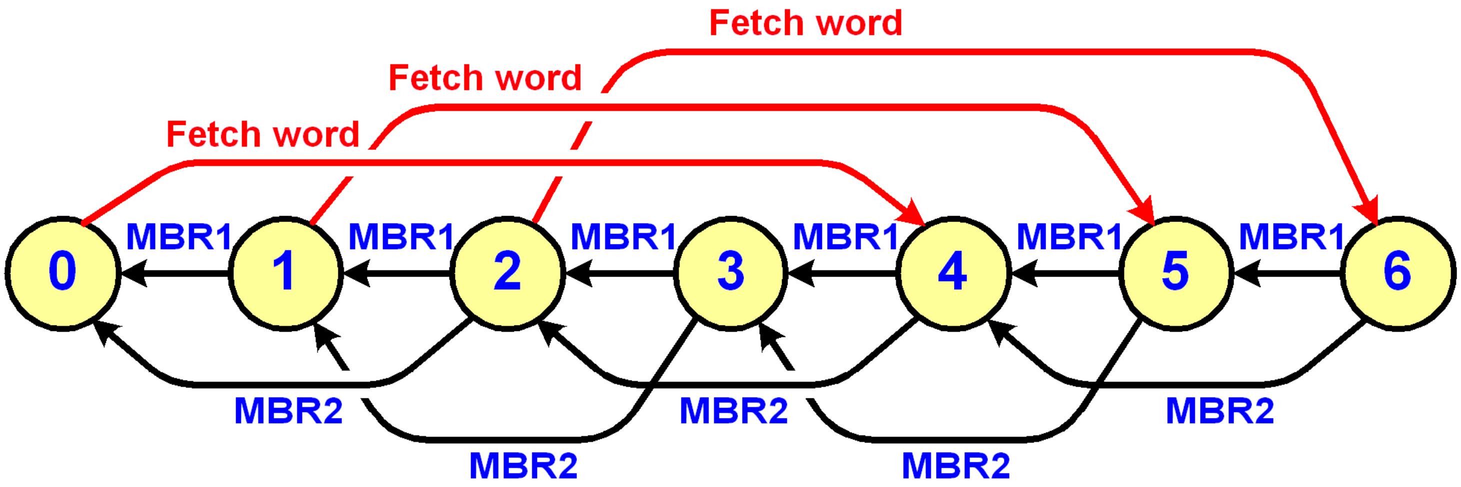
The shift register guarantees that the next bytes in the instruction stream will always be available to the microcode without having to wait for memory reads.

But when the program loads a new value into the PC register (which happens for GOTO and other branch instructions), the IFU must clear it's shift register and reload it from the new PC address. This means that the MBR1 and MBR2 registers will not have the correct values until the memory can supply them. Therefore, when a microcode changes the value in the PC register it must wait for one extra clock cycle before it can use the MBR1 or MBR2 values.

Reducing Execution Path Length

An Instruction Fetch Unit

The operation of the instruction fetch unit can be modeled by a Finite State Machine (FSM):



A Finite State Machine has several discrete states (represented by circles in the diagram) and transitions between states (represented by arrows).

In the IFU, the states are the number of bytes in the shift register. Transitions occur when a microinstruction reads data from the MBR1 or MBR2 register. In addition, the IFU itself triggers additional transitions by fetching 4 more bytes from memory whenever there are two or less bytes (i.e., four or more free bytes) in the shift register.

So, for example, if the shift register contains 3 bytes and a microinstruction reads data from the MBR2 register, the FSM transitions to state 1, and then the IFU fetches a word and transitions to state 5.

Reducing Execution Path Length

An Instruction Fetch Unit

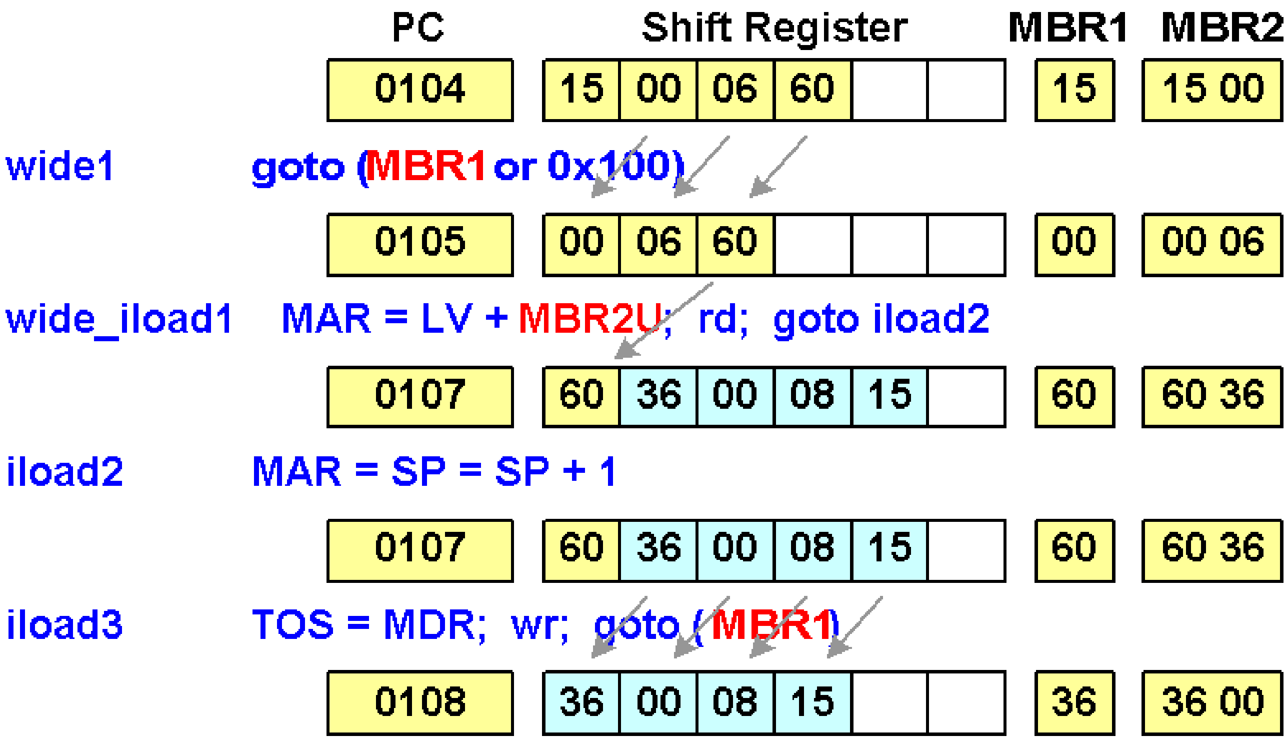
Here's an example of how the instruction fetch unit works.   Assume that the Method Area contains the following data and the system is executing the "WIDE ILOAD" at byte 104:

C4	15	00	06	60	36	00	08	15
103	104	105	106	107	108	109	10A	10B

Here are the contents of the IFU registers after each microinstruction for "WIDE ILOAD" is executed. Note the following:

- 1 or 2 bytes are removed from the shift register each time a microinstruction uses MBR1 or MBR2

- The MBR1 and MBR2 registers are updated with the next available bytes as soon as data in them is used



	PC	Shift Register	MBR1	MBR2
	0104	15 00 06 60	15	15 00
wide1	goto (MBR1 or 0x100)			
	0105	00 06 60	00	00 06
wide_iload1	MAR = LV + MBR2U; rd; goto iload2			
	0107	60 36 00 08 15	60	60 36
iload2	MAR = SP = SP + 1			
	0107	60 36 00 08 15	60	60 36
iload3	TOS = MDR; wr; goto (MBR1)			
	0108	36 00 08 15	36	36 00

- New data (shown in blue) is read into the shift register as soon as at least 4 bytes are free

Exercise 4 – Instruction Fetch Unit

The Method Area and the Instruction Fetch Unit (IFU) contains the following data:

Memory

324	325	326	327	328	329	32A	32B	32C
C4	36	07	F2	15	24	15	1C	60

IFU

PC	Shift Register						MBR1	MBR2
0325	36	07	F2				36	36 07

Which of the following shows what the instruction fetch unit will hold after executing this microinstruction?

goto (MBR1 or 0x100)

	PC	Shift Register						MBR1	MBR2
A	0325	36	07	F2				36	36 07
B	0326	07	F2					07	7F2
C	0326	07	F2	15	24	15	1C	07	07 F2
D	0327	F2	15	24	15	1C		F2	F2 15

Exercise 5 – Instruction Fetch Unit

The Method Area and the Instruction Fetch Unit (IFU) contains the following data:

Memory

C4	36	07	F2	15	24	15	1C	60
324	325	326	327	328	329	32A	32B	32C

IFU

PC	Shift Register						MBR1	MBR2
0325	36	07	F2				36	36 07

Which of the following shows what the instruction fetch unit will hold after executing this microinstruction?

MAR = SP = SP – 1; rd

	PC	Shift Register						MBR1	MBR2
A	0325	36	07	F2				36	36 07
B	0326	07	F2					07	7F2
C	0326	07	F2	15	24	15	1C	07	07 F2
D	0327	F2	15	24	15	1C		F2	F2 15

Exercise 6 – Instruction Fetch Unit

The Method Area and the Instruction Fetch Unit (IFU) contains the following data:

Memory

C4	36	07	F2	15	24	15	1C	60
324	325	326	327	328	329	32A	32B	32C

IFU

PC	Shift Register						MBR1	MBR2
0325	36	07	F2				36	36 07

Which of the following shows what the instruction fetch unit will hold after executing this microinstruction?

TOS = MBR1U; goto (MBR1)

	PC	Shift Register						MBR1	MBR2
A	0325	36	07	F2				36	36 07
B	0326	07	F2					07	7F2
C	0326	07	F2	15	24	15	1C	07	07 F2
D	0327	F2	15	24	15	1C		F2	F2 15

Reducing Execution Path Length

There are other techniques that can be used to reduce the execution path length, most of which involve adding more hardware (and therefore increasing cost). Some examples:

- **Extra logic could be added to the IFU to decode the instructions and automatically fetch the operands.**
- **Additional data paths could be added to allow direct transfers of data between commonly-used registers such as TOS and MDR. This would allow these transfers to take place in parallel with other ALU activity.**

The single most dramatic way to reduce execution path length is to eliminate the microcode altogether. As an example of this imagine a system which is designed so that the ISA level instructions actually specify the control bits needed to drive the data path. This type of system could simply read in ISA instruction from memory, load it into the MIR register, and let the control signals do their work.

This technique is behind the idea of the RISC architecture. By making the ISA level instructions very simple, less work is needed to interpret them.

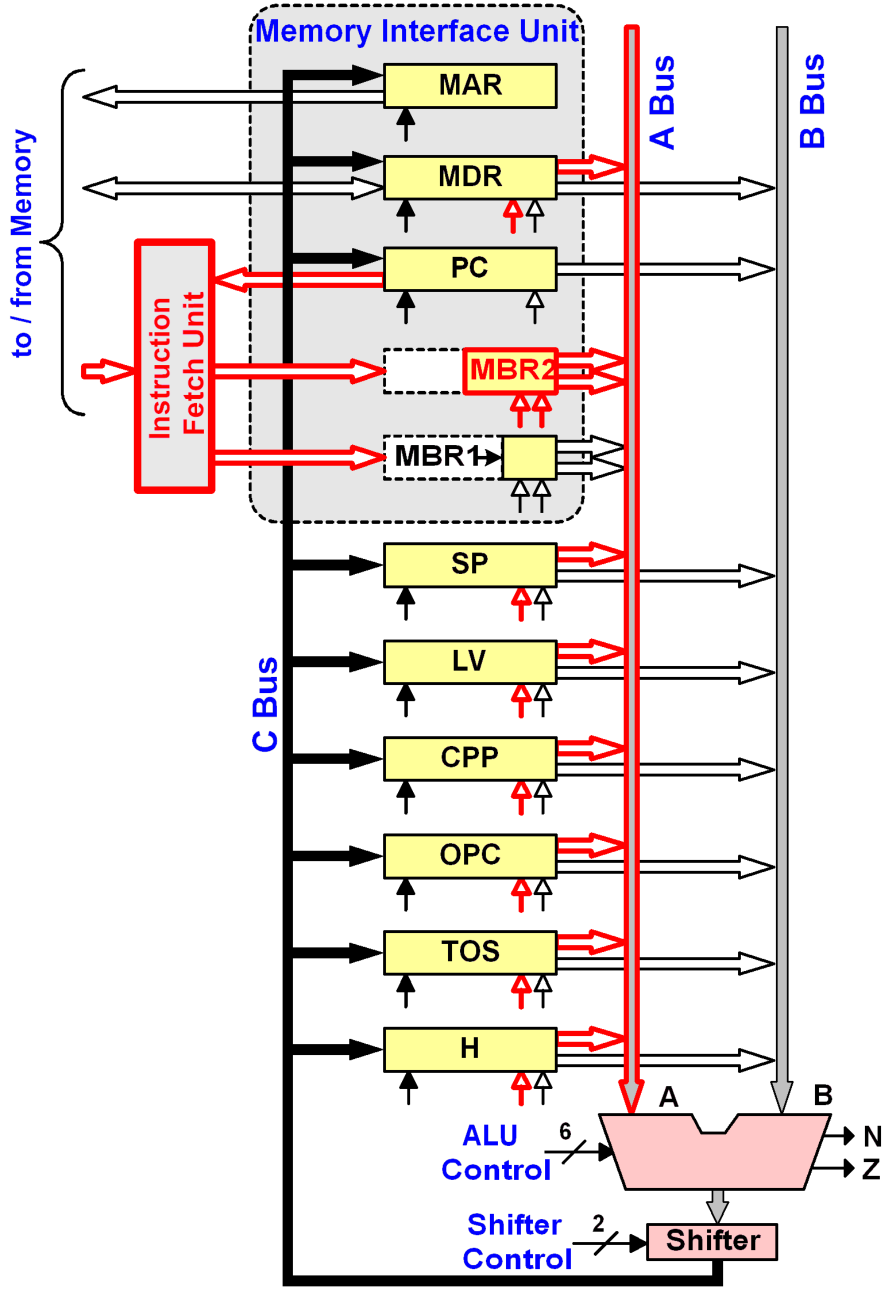
In reality, all modern processors use at least some microcode because even RISC processors need to handle complex tasks (which we haven’t discussed yet) such as interrupt and exception processing.

The MIC-2 Microarchitecture

The textbook's MIC-2 microarchitecture incorporates a 3-bus architecture and an Instruction Fetch unit to improve upon the performance of the MIC-1. The data path of the MIC-2 is shown here.

The control logic to drive the data path is not shown here, but it is essentially identical to the logic for the MIC-1 microarchitecture, except that the microinstruction word needs to be longer so that it can hold the additional control signals needed to connect the various registers to the A Bus. This means that the control store will also have to be enlarged to hold the longer microinstruction words.

This type of design was common in the mid 1980's in processor generations such as the Intel 80286.



MIC-1 vs. MIC-2 Microinstructions - IADD

These are MIC-1 and MIC-2 microinstructions for the IJVM IADD instruction:

	MIC-1 Microinstructions		MIC-2 Microinstructions
iadd1	MAR = SP = SP – 1; rd	iadd1	MAR = SP = SP – 1; rd
iadd2	H = TOS	iadd2	H = TOS
iadd3	MDR = TOS = MDR + H; wr; goto Main1	iadd3	MDR = TOS = MDR+H; wr; goto (MBR1)
Main1	PC = PC + 1; fetch; goto (MBR)		

With IADD, we save having to execute the Main1 microinstruction just as we did for NOP.

Note that in the MIC-2 version of “iadd2”, we don’t really need to put the TOS value in the H register because the 3-bus architecture allows us to add TOS and MDR together directly. So we could use the following sequence instead:

iadd1	MAR = SP = SP – 1; rd
iadd2	
iadd3	MDR = TOS = MDR + TOS; wr; goto (MBR1)

However, we don’t save any time because we still need to wait for the 2nd operand to be read from the stack.

ISUB, IAND, and IOR are the same except for the ALU operation used in the “iadd3” microinstruction.

MIC-1 vs. MIC-2 Microinstructions - SWAP

4.4.3

These are MIC-1 and MIC-2 microinstructions for the IJVM SWAP instruction:

	MIC-1 Microinstructions		MIC-2 Microinstructions
swap1	MAR = SP – 1; rd	swap1	MAR = SP – 1; rd
swap2	MAR = SP	swap2	MAR = SP
swap3	H = MDR; wr	swap3	H = MDR; wr
swap4	MDR = TOS	swap4	MDR = TOS
swap5	MAR = SP – 1; wr	swap5	MAR = SP – 1; wr
swap6	TOS = H; goto Main1	swap6	TOS = H; goto (MBR1)
Main1	PC = PC + 1; fetch; goto (MBR)		

With SWAP, we save having to execute the Main1 microinstruction just as we did for NOP. But there aren’t really any other savings, because the SWAP microinstruction has only a single opcode byte and it’s largely constrained by having to do three memory accesses.

MIC-1 vs. MIC-2 Microinstructions - BIPUSH

These are MIC-1 and MIC-2 microinstructions for the IJVM BIPUSH instruction:

MIC-1 Microinstructions

bipush1	**SP = MAR = SP + 1**
bipush2	**PC = PC + 1; fetch**
bipush3	**MDR = TOS = MBR; wr; goto Main1**
Main1	**PC = PC + 1; fetch; goto (MBR)**

MIC-2 Microinstructions

bipush1	**SP = MAR = SP + 1**
bipush2	**MDR = TOS = MBR1; wr; goto (MBR1)**

With BIPUSH, we save two clock cycles:

- **We don’t have to execute the MIC-1 “bipush2” microinstruction because the IFU does this work automatically when the MBR1 register is used in the MIC-2 “bipush2” microinstruction.**

- **We don’t have to execute the MIC-1 “Main1” microinstruction because the IFU does this work automatically when the MBR1 register is referenced in the MIC-2 “bipush2” microinstruction.**

MIC-1 vs. MIC-2 Microinstructions - ILOAD

These are MIC-1 and MIC-2 microinstructions for the IJVM ILOAD instruction:

	MIC-1 Microinstructions		MIC-2 Microinstructions
iload1	H = LV	iload1	MAR = LV + MBR1U
iload2	MAR = MBRU + H; rd	iload2	MAR = SP = SP + 1
iload3	MAR = SP = SP + 1	iload3	TOS = MDR; wr; goto (MBR1)
iload4	PC = PC + 1; fetch; wr		
iload5	TOS = MDR; goto Main1		
Main1	PC = PC + 1; fetch; goto (MBR)		

We get a big savings with the ILOAD instruction because it needs to process an operand and add it to the LV register:

- **We eliminate the MIC-1 “iload1” microinstruction because we can add LV and MBR1U together in the same clock cycle in the MIC-2 “iload1” microinstruction.**
- **We eliminate the MIC-1 “iload4” microinstruction because this work is done automatically by the IFU when MBR1U is references in the MIC-2 “iload1” microinstruction.**
- **We eliminate the MIC-1 “Main1” microinstruction in the same way we have for the other microinstruction sequences.**

MIC-1 vs. MIC-2 Microinstructions - ISTORE

These are MIC-1 and MIC-2 microinstructions for the IJVM ISTORE instruction:

MIC-1 Microinstructions		MIC-2 Microinstructions	
istore1	H = LV	istore1	MAR = LV + MBR1U
istore2	MAR = MBRU + H	istore2	MDR = TOS; wr
istore3	MDR = TOS; wr	istore3	MAR = SP = SP + 1; rd
istore4	SP = MAR = SP – 1; rd	istore4	
istore5	PC = PC + 1; fetch	istore5	TOS = MDR; goto (MBR1)
istore6	TOS = MDR; goto Main1		
Main1	PC = PC + 1; fetch; goto (MBR)		

We manage to gain two clock cycles with the ISTORE instruction even though we need a "dead" clock cycle in the middle while we wait for the new TOS value to be read from memory:

- **We eliminate the MIC-1 "istore1" microinstruction because we can add LV and MBR1U together in the same clock cycle in the MIC-2 "istore1" microinstruction.**
- **We eliminate the MIC-1 "istore5" microinstruction because this work is done automatically by the IFU when MBR1U is references in the MIC-2 "istore1" microinstruction.**
- **We eliminate the MIC-1 "Main1" microinstruction in the same way we have for the other microinstruction sequences.**

MIC-1 vs. MIC-2 Microinstructions - WIDE ILOAD

4.4.3

These are MIC-1 and MIC-2 microinstructions for the IJVM WIDE ILOAD instruction:

MIC-1 Microinstructions

wide1	**PC = PC + 1; fetch;** **goto (MBR OR 0x100)**
w_iload1	**PC = PC + 1; fetch**
w_iload2	**H = MBRU << 8**
w_iload3	**H = MBRU OR H**
w_iload4	**MAR = LV + H; rd; goto iload3**
iload3	**MAR = SP = SP + 1**
iload4	**PC = PC + 1; fetch; wr**
iload5	**TOS = MDR; goto Main1**
Main1	**PC = PC + 1; fetch; goto (MBR)**

MIC-2 Microinstructions

wide1	**goto (MBR1 OR 0x100)**
w_iload1	**MAR = LV + MBR2U; rd;** **goto iload2**
iload2	**MAR = SP = SP + 1**
iload3	**TOS = MDR; wr; goto (MBR1)**

We more than double the performance of WIDE ILOAD because it needs to process a 16-bit operand:

- **The MIC-1 "w_iload1" and "iload4" microinstructions are eliminated because the IFU does this work automatically when the MBR2 register is used in MIC-2 "w_iload1"**
- **The MIC-1 "w_iload2" and "w_iload3" microinstructions are eliminated because the IFU does this work automatically when the MBR2 register is referenced in MIC-2 "w_iload1".**

MIC-1 vs. MIC-2 Microinstructions - WIDE ISTORE

These are MIC-1 and MIC-2 microinstructions for the IJVM WIDE ISTORE instruction:

MIC-1 Microinstructions	
wide1	PC = PC + 1; fetch; goto (MBR OR 0x100)
w_istore1	PC = PC + 1; fetch
w_istore2	H = MBRU << 8
w_istore3	H = MBRU OR H
w_istore4	MAR = LV + H; goto istore3
istore3	MDR = TOS; wr
istore4	SP = MAR = SP – 1; rd
istore5	PC = PC + 1; fetch
istore6	TOS = MDR; goto Main1
Main1	PC = PC + 1; fetch; goto (MBR)

MIC-2 Microinstructions	
wide1	goto (MBR1 OR 0x100)
w_istore1	MAR = LV + MBR2U; goto istore2
istore2	MDR = TOS; wr
istore3	MAR = SP = SP + 1; rd
istore4	
istore5	TOS = MDR; goto (MBR1)

We almost double the performance of WIDE ILOAD because it needs to process a 16-bit operand:

- **The MIC-1 “w_istore1” and “istore5” microinstructions are eliminated because the IFU does this work automatically when the MBR2 register is used in MIC-2 “w_istore1”**
- **The MIC-1 “w_istore2” and “w_istore3” microinstructions are eliminated because the IFU does this work automatically when MBR2 is referenced in MIC-2 “w_istore1”.**

MIC-1 vs. MIC-2 Microinstructions - GOTO

These are MIC-1 and MIC-2 microinstructions for the IJVM GOTO instruction:

	MIC-1 Microinstructions		MIC-2 Microinstructions
goto1	OPC = PC - 1	goto1	H = PC - 1
goto2	PC = PC + 1; fetch	goto2	PC = H + MBR2
goto3	H = MBR << 8	goto3	
goto4	H = MBRU OR H	goto4	goto (MBR1)
goto5	PC = OPC + H; fetch		
goto6	goto Main1		
Main1	PC = PC + 1; fetch; goto (MBR)		

We more than double the performance of IINC because it needs to process two operands:

- **The MIC-1 "goto2" microinstructions has been eliminated because the IFU does this work automatically when the MBR2 register is used in MIC-2 "goto2"**
- **The MIC-1 "goto3" and "goto5" microinstructions are eliminated because the IFU does this work automatically when the MBR2 register is referenced in MIC-2 "goto2".**
- **Because H isn't being used to assemble the 16-bit operand, we can use it to save the old PC address and so we don't need to use the OPC register any more.**

Note that we need to have a "dead" clock cycle after putting a new value in the PC register. This gives the IFU time to fetch the next opcode byte from the Method area before it's decoded.

MIC-1 vs. MIC-2 Microinstructions - IFEQ

These are MIC-1 and MIC-2 microinstructions for the IJVM IFEQ instruction:

	MIC-1 Microinstructions
ifeq1	MAR = SP = SP – 1; rd
ifeq2	OPC = TOS
ifeq3	TOS = MDR
ifeq4	Z = OPC; if (Z) goto T; else goto F
F1	PC = PC + 1
F2	PC = PC + 1; fetch
F3	goto Main1
T	H = PC – 1; goto goto2
goto2	PC = PC + 1; fetch
goto3	H = MBR << 8
goto4	H = MBRU OR H
goto5	PC = OPC + H; fetch
goto6	goto Main1
Main1	PC = PC + 1; fetch; goto (MBR)

	MIC-2 Microinstructions
ifeq1	MAR = SP = SP – 1; rd
ifeq2	OPC = TOS
ifeq3	TOS = MDR
ifeq4	Z = OPC; if (Z) goto T; else goto F
F1	H = MBR2
F2	goto (MBR1)
T	H = PC – 1; goto goto2
goto2	PC = H + MBR2
goto3	
goto4	goto (MBR1)

The main microinstructions for the IJVM "IFEQ" instruction don't change much because it only interprets an opcode and decides whether to jump or not. But the MIC-1 "F1" microinstruction can been eliminated because the IFU does this work automatically when the MBR2 register is used in MIC-2 "F1" microinstruction.

The IJVM "IFLT" instruction is the same except for checking the "N" flag in "ifeq4"

MIC-1 vs. MIC-2 Microinstructions - IF_ICMPEQ

These are MIC-1 and MIC-2 microinstructions for the IJVM IFEQ instruction:

MIC-1 Microinstructions

```
if_icmpeq1  MAR = SP = SP – 1; rd
if_icmpeq2  MAR = SP = SP – 1
if_icmpeq3  H = MDR; rd
if_icmpeq4  OPC = TOS
if_icmpeq5  TOS = MDR
if_icmpeq6  Z = OPC – H; if (Z) goto T;
            else goto F
F1          PC = PC + 1
F2          PC = PC + 1; fetch
F3          goto Main1
T           H = PC – 1;  goto goto2
goto2       PC = PC + 1; fetch
goto3       H = MBR << 8
goto4       H = MBRU OR H
goto5       PC = OPC + H; fetch
goto6       goto Main1
Main1       PC = PC + 1;  fetch;  goto (MBR)
```

MIC-2 Microinstructions

```
if_icmpeq1  MAR = SP = SP – 1; rd
if_icmpeq2  MAR = SP = SP – 1
if_icmpeq3  H = MDR; rd
if_icmpeq4  OPC = TOS
if_icmpeq5  TOS = MDR
if_icmpeq6  Z = H – OPC; if (Z) goto T;
            else goto F
F1          H = MBR2
F2          goto (MBR1)
T           H = PC – 1;  goto goto2
goto2       PC = H + MBR2
goto3
goto4       goto (MBR1)
```

The main microinstructions for the IJVM "IFEQ" instruction don't change much because it only interprets an opcode and decides whether to jump or not. But the MIC-1 "F1" microinstruction can been eliminated because the IFU does this work automatically when the MBR2 register is used in MIC-2 "F1" microinstruction.

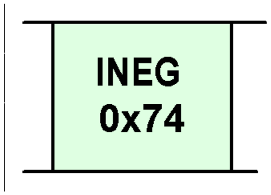
MIC-1 vs. MIC-2 Microinstructions - Summary

Here's a summary of the differences between the MIC-1 and MIC-2 implementations:

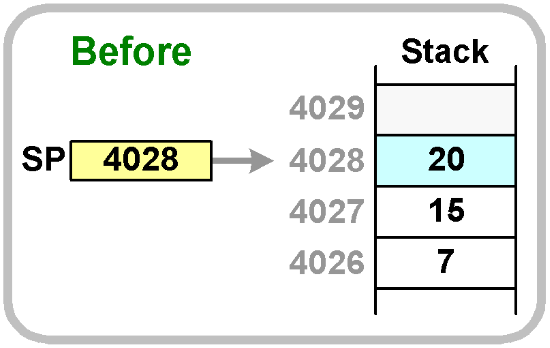
Instruction	MIC-1 Clock Cycles	MIC-2 Clock Cycles	MIC-2 Time
NOP	2	1	50%
IADD	4	3	75%
ISUB	4	3	75%
IAND	4	3	75%
IOR	4	3	75%
DUP	3	2	67%
POP	4	3	75%
SWAP	7	6	85%
BIPUSH	4	2	50%
ILOAD	6	3	50%
ISTORE	7	5	71%
WIDE_ILOAD	9	4	44%
WIDE_ISTORE	10	6	60%
LDC_W	8	3	38%
IINC	7	3	43%
GOTO	7	4	57%
IFLT	8	6	75%
IFEQ	8	6	75%
IF_ICMPEQ	10	8	80%
INVOKEVIRTUAL	22	11	50%
IRETURN	9	8	89%

Exercise 7 – MIC-2 Microcode

Write the microcode needed for the MIC-2 microarchitecture to implement the JVM "INEG" instruction:



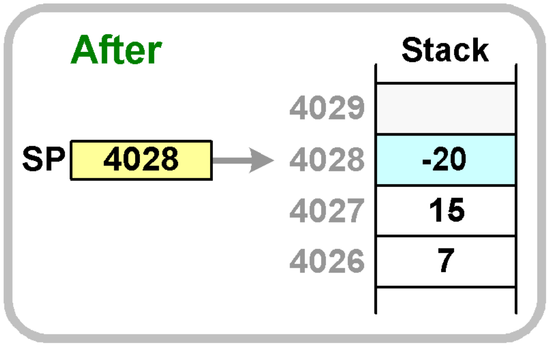
This instruction reverses the sign of the top item on the stack.



Before

SP 4028

	Stack
4029	
4028	20
4027	15
4026	7



After

SP 4028

	Stack
4029	
4028	-20
4027	15
4026	7

Which of the following versions of microcode is correct?

A

```
H = TOS
MAR = SP
TOS = MDR = –H; wr;
                     goto (MBR1)
```

B

```
MAR = SP = SP – 1
TOS = –TOS; wr;
                     goto (MBR1)
```

C

```
MAR = SP
MDR = TOS = –TOS; wr
                     goto (MBR1)
```

A Pipelined Microarchitecture

4.4.4

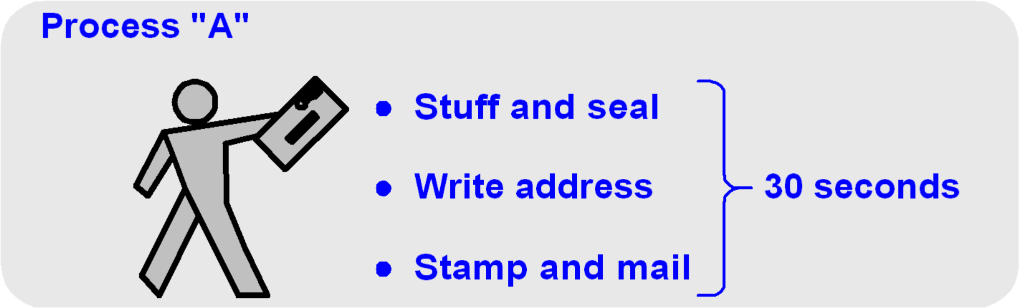
To improve performance further, we'll introduce parallelism into our basic data path by creating a pipeline. A pipeline allows us to break up the execution work into smaller pieces so we can do different pieces in parallel. For example, imagine two processes for mailing letters:

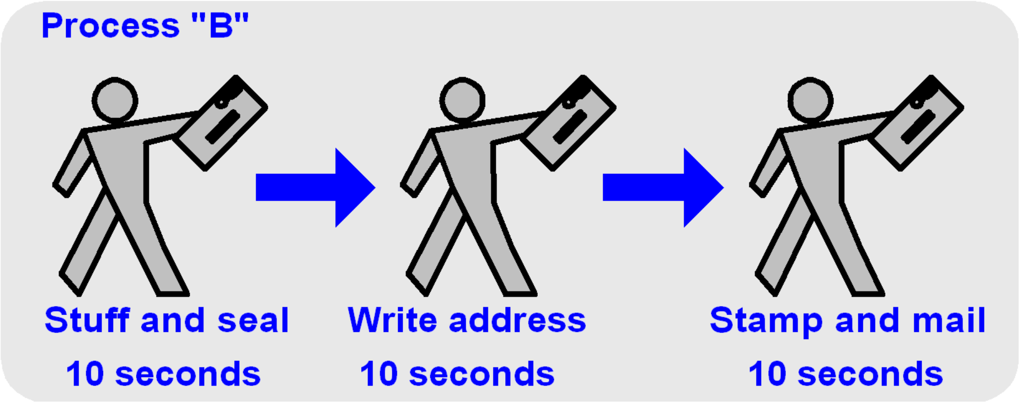
In "Process A", one person does all the work of mailing a letter sequentially and it takes 30 seconds.

In "Process B", the work is split up so that three people can each do one third of the work in 10 seconds each.

At first, it doesn't seem like we've gained anything – it still takes 30 seconds to get a letter mailed.

But what happens when there are a lot of letters to be mailed? "Process A" can only mail one letter every 30 seconds, so the maximum rate is two letters per minute.

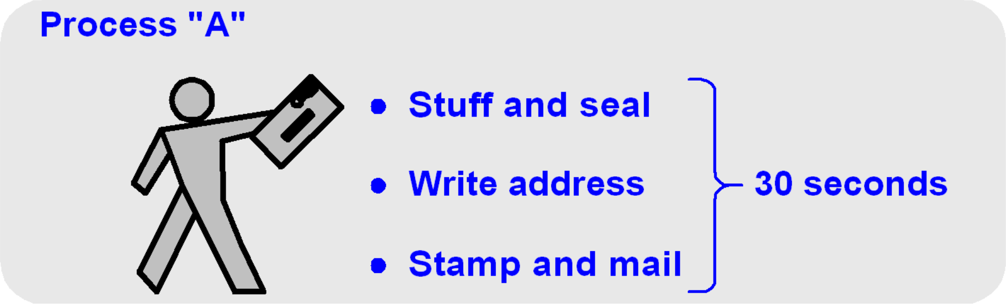


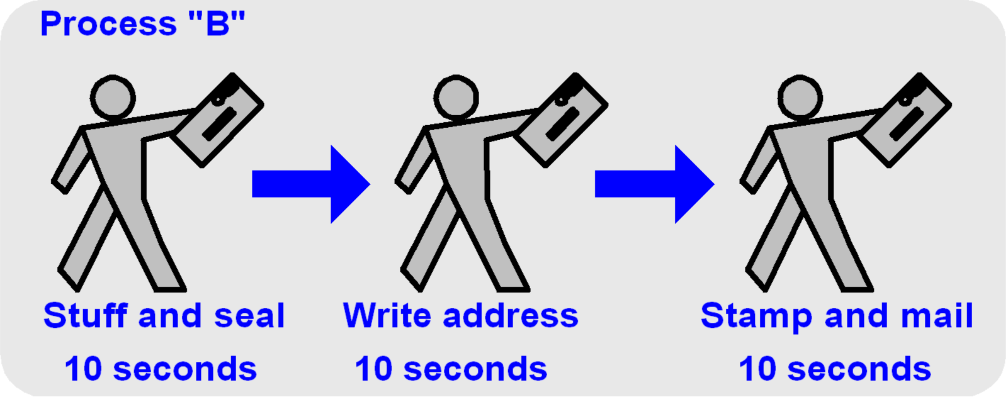


Now look at "Process B" – it can start working on a new letter every 10 seconds. So it's capable of mailing six letters per minute.

A Pipelined Microarchitecture

You might think: why not just have three people, each of whom does “Process A”. You’d also be able to mail 6 letters a minute by organizing things that way.





That works fine for humans mailing letters, but the problem with using that approach for building a parallel computer system is that each of the three workers has to be able to do all three stages. In other words, you need three complete replicas of the entire process.

With “Process B”, you can build the same functions you use in a simple, non-parallel processor, but simply divide them up into sub-sections so they can be used in parallel. This can often be done with very little extra circuitry or complexity – and it gives performance that’s just as good as completely functional parallel components.

MIC-3 – A Pipelined Microarchitecture

The textbook's "MIC-3" microarchitecture breaks the data path up into separate stages:

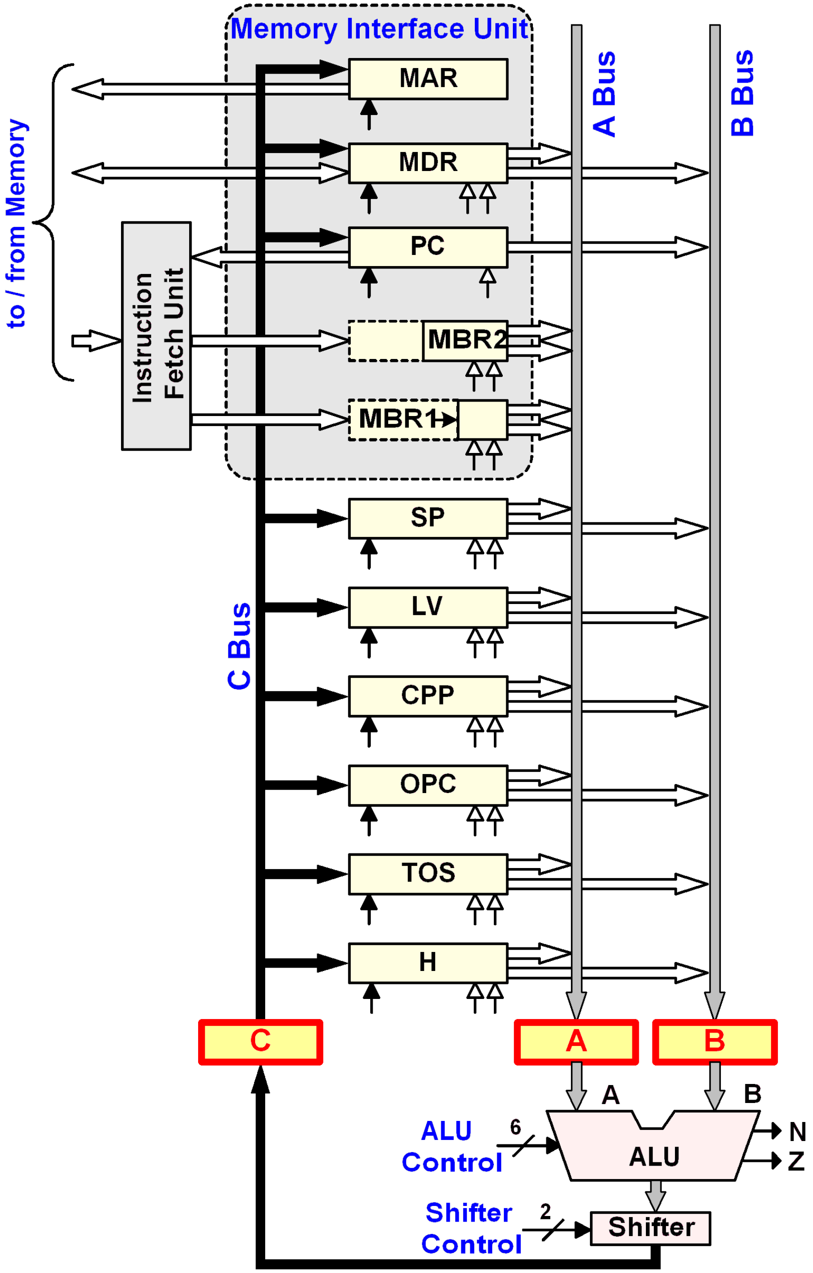
1. **Load ALU Inputs**
   This stage moves data from one or two registers to the inputs of the ALU

2. **Execute ALU Function**
   This stage moves the input data through the ALU and performs the requested functions on it.

3. **Store Results**
   This stage takes the output produced by the ALU and stores it in one or more registers.

It's remarkably easy to create the pipeline – we simply add three new registers called A, B and C into the data path as shown at the right.

Notice that these registers don't have any control signals like the other ones do. They simply load whatever is sent to them at the end of each clock cycle, and then forward it on to the next stage at the beginning of the next clock cycle.

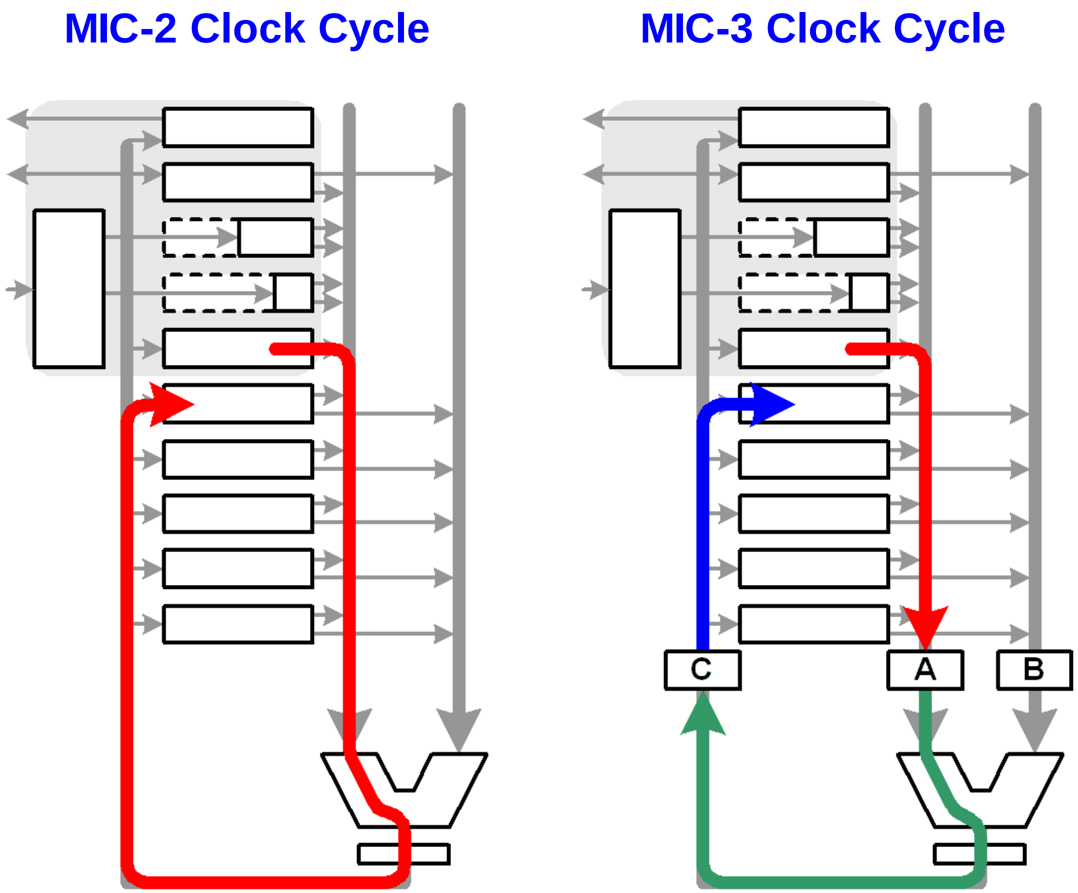


MIC-2 vs. MIC-3 Clock Cycles

The A, B and C registers break up the data path into separate pieces so that on each clock cycle information flows only part of the way from the source register to the target register.

In the MIC-1 and MIC-2, data flows all the way from the source register, through the ALU, and to the destination register in each clock cycle as represented by the red line immediately to the right.

In the MIC-3, data only flows from source register to ALU input (A or B register), from ALU input to output (C register), or from ALU output to target register. This is represented by the red, green and blue lines at the far right.


MIC-2 Clock Cycle

MIC-3 Clock Cycle

In the MIC-3, it takes <u>3 complete clock cycles</u> to move data from a source register, send it through the ALU, and store it in a destination register.

MIC-3 – A Pipelined Microarchitecture

At first it seems like it would be slower to break up the data path so that it takes three clock cycles to move information through it. But there are two reasons why this technique gives better performance:

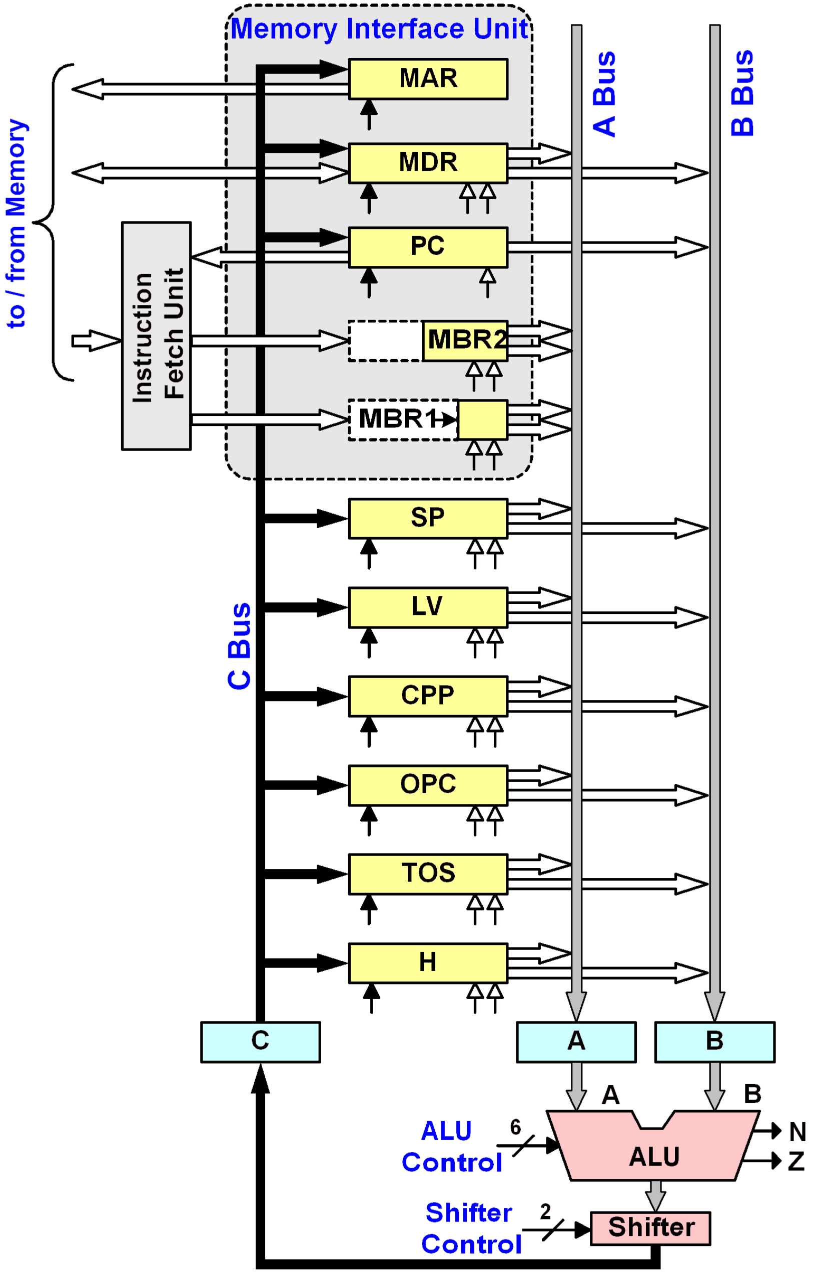
We can use a faster clock

Because there is less work done in each leg of the pipelined data path, we can use a shorter cycle time. The fastest clock speed will depend on the slowest part of the pipeline (probably the ALU). For the MIC-3, we're going to assume that we can increase the clock speed by a factor of 2.5 (in other words, if the MIC-2 had a 100MHz clock speed then the MIC-3 could run with a 250MHz clock).

Different parts of the data path can work in parallel

The ALU inputs can be loaded with one set of registers while a different set (loaded on the previous clock cycle) can be added together and the result of yet a different set (loaded two clock cycles ago) can be being stored.

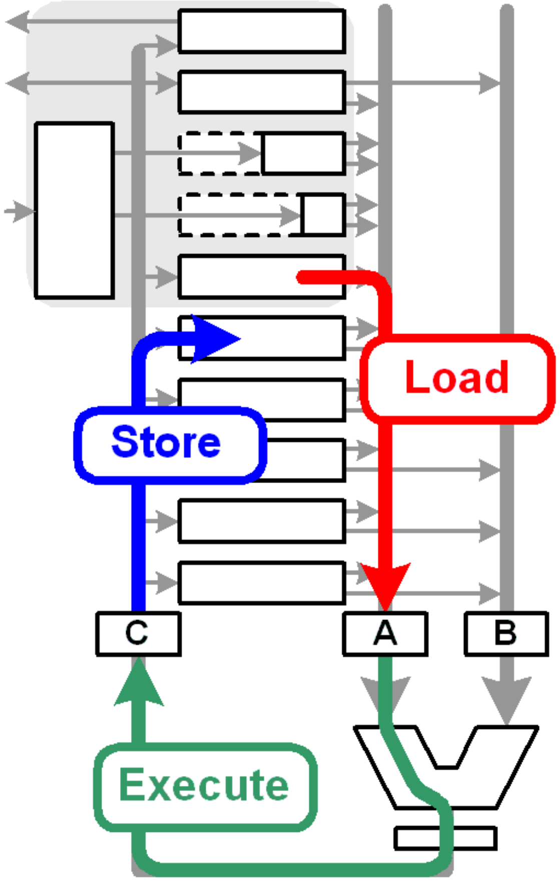
These are the basic reasons why pipelined microarchitectures can perform better.



MIC-3 – MAL Notation

The microinstruction word for the MIC-3 isn't all that different from the MIC-2 or MIC-1, but because there can be different things going on in different parts of the data path it's easier to create microprograms if we use a different MAL notation that treats the three pipeline stages as separate entities.

Again, the three pipeline stages are:



1. **Load ALU Inputs**
   In this stage we connect registers to the A and/or B bus so that they are loaded into the ALU inputs.

2. **Execute ALU Function**
   In this stage we specify what ALU and shifter operation is to be performed on the A and B inputs (note that the A and B inputs processed on any given clock cycle are the ones that were loaded during previous clock cycle). The result is stored in the C register.

3. **Store Results**
   In this stage we indicate which registers will be loaded with the C result from the ALU. Note that the result that is stored on any given clock cycle is the one that came out of the ALU at the end of the previous clock cycle.

MIC-3 – MAL Notation

To show the progress of information through the data path, we write MIC-3 microcode in a grid that clearly shows what's happening in each pipeline stage during each clock cycle. Here is a grid coded with the microinstructions to do the MIC-3 equivalent of:

MAR = SP = MDR + LV

Cycle	Load ALU Inputs	Execute ALU Function	Store ALU Results
1	A = MDR; B = LV		
2		C = A + B	
3			MAR = SP = C

This grid shows that:

- **During clock cycle one, the A and B registers are loaded with the contents of the MDR and LV registers, respectively**
- **During clock cycle two, the A and B inputs are added together (this is the MDR and LV contents put into A and B during clock cycle 1)**
- **During clock cycle three, the result (the sum of MDR and LV) is stored in MAR and SP**

(We can't see the value in the pipelined microarchitecture while executing just this one microinstruction – it will become more apparent when we have a series to execute.)

MIC-3 – MAL Notation

Use these guidelines while writing the MAL code for the "Load ALU Inputs" pipeline stage:

- **Use "A = *source-register*" (i.e., "A = TOS") to control which register will be connected to the "A" bus (and therefore loaded into the A input of the ALU on the next clock cycle)**
- **Use "B = *source-register*" (i.e., "B = MDR") to control which register will be connected to the "B" bus (and therefore loaded into the B input of the ALU on the next clock cycle)**
- **You may use one, both, or neither of "A = *source-register*" and "B = *source-register*" in any given clock cycle.**
- **The MBR1 and MBR2 registers are fed by the Instruction Fetch Unit as in the MIC-2. They always contain the next byte and word, respectively, in the instruction stream.**
- **Remember you can only connect one register to the A or B bus at any time and you can't perform any operations on it. Statements such as "A = TOS = MDR" or "B = LV + MBR1U" are not valid.**
- **Be sure to use valid registers. For example, "B = MBR1U" is not valid for the MIC-3 because the MBR1 register is not connected to the B bus.**

Use these guidelines while writing the MAL code for the "Perform ALU Operation" stage:

- **Use "C = *ALU-operation*" (i.e., "C = A + B") to set the ALU operation which will be used.**
- **Remember to use only valid ALU operations. For example, "C = A – B" is not valid because the ALU cannot perform this function (it can only perform "C = B – A").**
- **Remember that the A and B ALU inputs come from what was specified under "Load ALU Inputs" during the previous clock cycle.**

MIC-3 – MAL Notation

Use these guidelines while writing the MAL code for the "Store ALU Results" pipeline stage:

- **Use "*target-register* = C" (i.e., "TOS = C") to control which register will be loaded from the C register (which contains the ALU results from the previous clock cycle)**
- **More than one target register can be specified at a time (for example, "TOS = MDR = C")**
- **If you want to specify a memory operation ("rd" or "wr"), you can do so by appending it to the MAL statement in this stage. For example, "MAR = C; rd"**
- **The rule for memory reads is the same as for the MIC-1 and MIC-2 – the requested data will not arrive in the MDR register until the end of the next clock cycle. It's useful to put the notation "(MDR = mem)" in the clock cycle following the "rd" to remind you that the MDR register contents are changed at this time. This notation doesn't actually affect the microinstruction word, but serves like a comment to note when the requested data actually arrives from memory. You can do the same thing to note memory writes using "(mem = MDR)", but the microcode is not usually dependent on the timing of writes so this isn't as important.**
- **Include "goto (MBR1)" in the last microinstruction of the sequence to perform the decoding for the next IJVM instruction.**

MIC-3 – Microcode for “SWAP”

Here’s sample microcode showing how the SWAP microinstruction is implemented in the MIC-3 microarchitecture:

Original MIC-2 Microcode for SWAP

swap1	MAR = SP – 1; rd
swap2	MAR = SP
swap3	H = MDR; wr
swap4	MDR = TOS
swap5	MAR = SP – 1; wr
swap6	TOS = H; goto (MBR1)

Cycle	Load ALU Inputs	Perform ALU Operation	Store ALU Outputs
1	B = SP ①		
2	B = SP ②	C = B - 1 ①	
3		C = B ②	MAR = C; rd ①
4	(MDR = mem) ①		MAR = C ②
5	B = MDR ③		
6	B = TOS ④	C = B ③	
7	B = SP ⑤	C = B ④	H = C; wr ③
8	B = H ⑥	C = B - 1 ⑤	MDR = C ④
9		C = B ⑥	MAR = C; wr ⑤
10			TOS = C; goto (MBR1) ⑥

This version of SWAP is the same as the version in figure 4-33 (page 286) of the textbook, although it’s written in a slightly different format.

MIC-3 – Microcode for “SWAP”

Notice how that there is a gap in the microinstructions – there are a couple of clock cycles in each pipeline stage where nothing is happening. The “(MDR = mem)” in cycle 4 must complete before “B = MDR” in cycle 5 can be executed.

This situation is called a “Read After Write” (RAW) dependency. We’ll learn more about various types of dependencies in the next lesson.

The dependency causes a “pipeline stall” which results in a “bubble” of empty clock cycles propagating through the pipeline.

Cycle	Load ALU Inputs	Perform ALU Operation	Store ALU Outputs
1	B = SP 1		
2	B = SP 2	C = B - 1 1	
3		C = B 2	MAR = C; rd 1
4	(MDR = mem) 1		MAR = C 2
5	B = MDR 3		
6	B = TOS 4	C = B 3	
7	B = SP 5	C = B 4	H = C; wr 3
8	B = H 6	C = B - 1 5	MDR = C 4
9		C = B 6	MAR = C; wr 5
10			TOS = C; goto (MBR1) 6

We’ve gone from six clock cycles up to ten, but the MIC-3 clock is 2.5 times faster. So the net time is (10/6) x (1 / 2.5) = 1.6667 x 0.4 = 0.6667.

SWAP now takes 2 / 3 the time it used to.

MIC-3 – Optimized Microcode for “SWAP”

4.4.4

The textbook microcode for the MIC-3 SWAP implementation is based on MIC-2 microcode and is not the optimal solution for the pipelined microarchitecture. Here’s a version that takes better advantage of the pipeline and requires only 8 clock cycles:

1. Point to 2nd word on stack and read it.
2. Write old TOS to 2nd word on stack
3. Save 2nd word from stack in TOS (this is the new TOS value)
4. Write new TOS to the top of the stack.

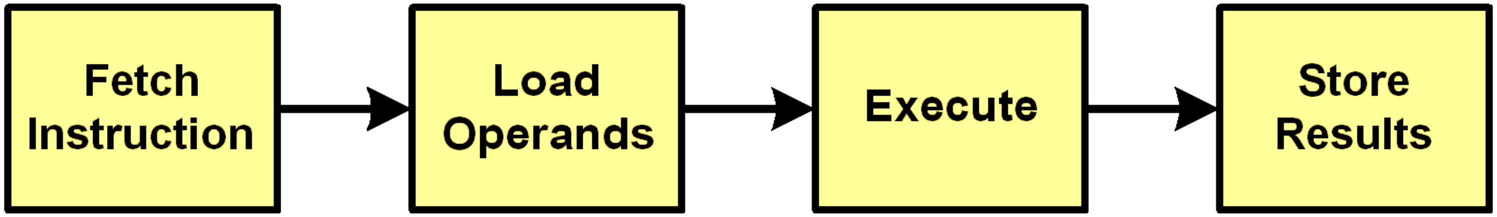
Cycle	Load ALU Inputs	Perform ALU Operation	Store ALU Outputs
1	B = SP 1		
2		C = B - 1 1	
3	B = TOS 2		MAR = C; rd 1
4	(MDR = mem) 1	C = B 2	
5	B = MDR 3		MDR = C; wr 2
6	B = SP 4	C = B 3	
7		C = B 4	TOS = MDR = C 3
8			MAR = C; wr goto (MBR1) 4

This implementation finesses information around different parts of the pipeline at the same time. For example, in clock cycle 5 the MDR register is being used as a source of data at the start of the clock cycle and is accepting a completely different value at the end of the clock cycle.

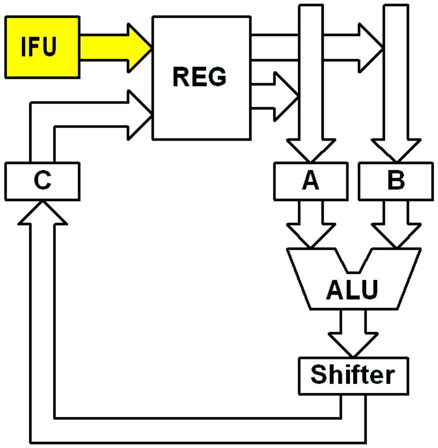
The net time is now (8/6) x (1 / 2.5) = 1.333 x 0.4 = 0.5333. SWAP now takes only about half as long as it used to.

MIC-3 – A Pipelined Microarchitecture

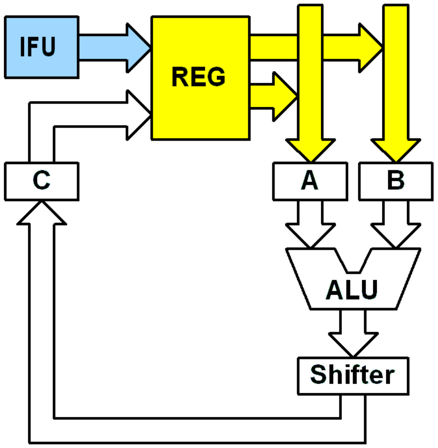
When we include the instruction fetch unit, we can see that the MIC-3 has a four-stage pipeline:



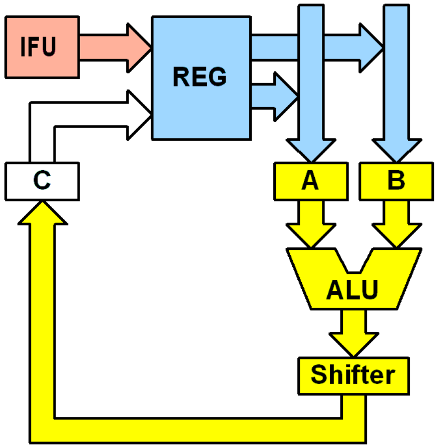
This pipeline diagram looks very similar to some of the pipelines we studied in the early lessons of the course, and now we can see exactly how they are built and operate:



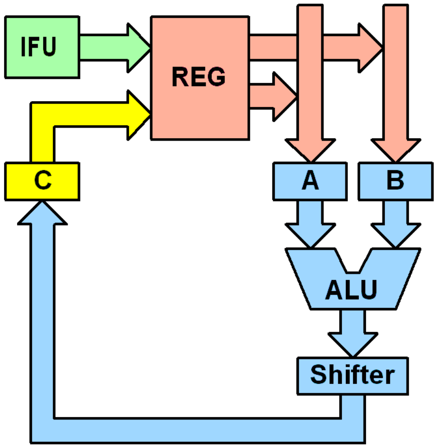
Cycle 1
1st Instruction enters pipeline



Cycle 2
2nd Instruction enters pipeline



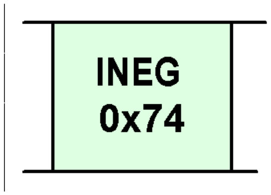
Cycle 3
3rd Instruction enters pipeline



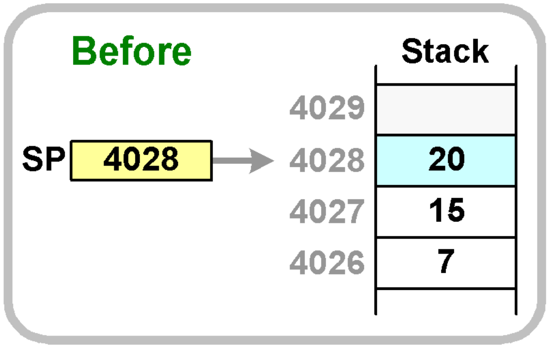
Cycle 4
4th Instruction enters pipeline

Exercise 8 – MIC-3 Microcode

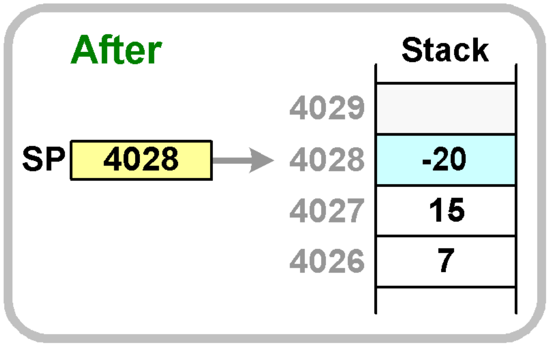
Write the microcode needed for the MIC-3 microarchitecture to implement the JVM “INEG” instruction:


INEG
0x74

This instruction reverses the sign of the top item on the stack.


Before

	Stack
4029	
SP 4028 → 4028	20
4027	15
4026	7


After

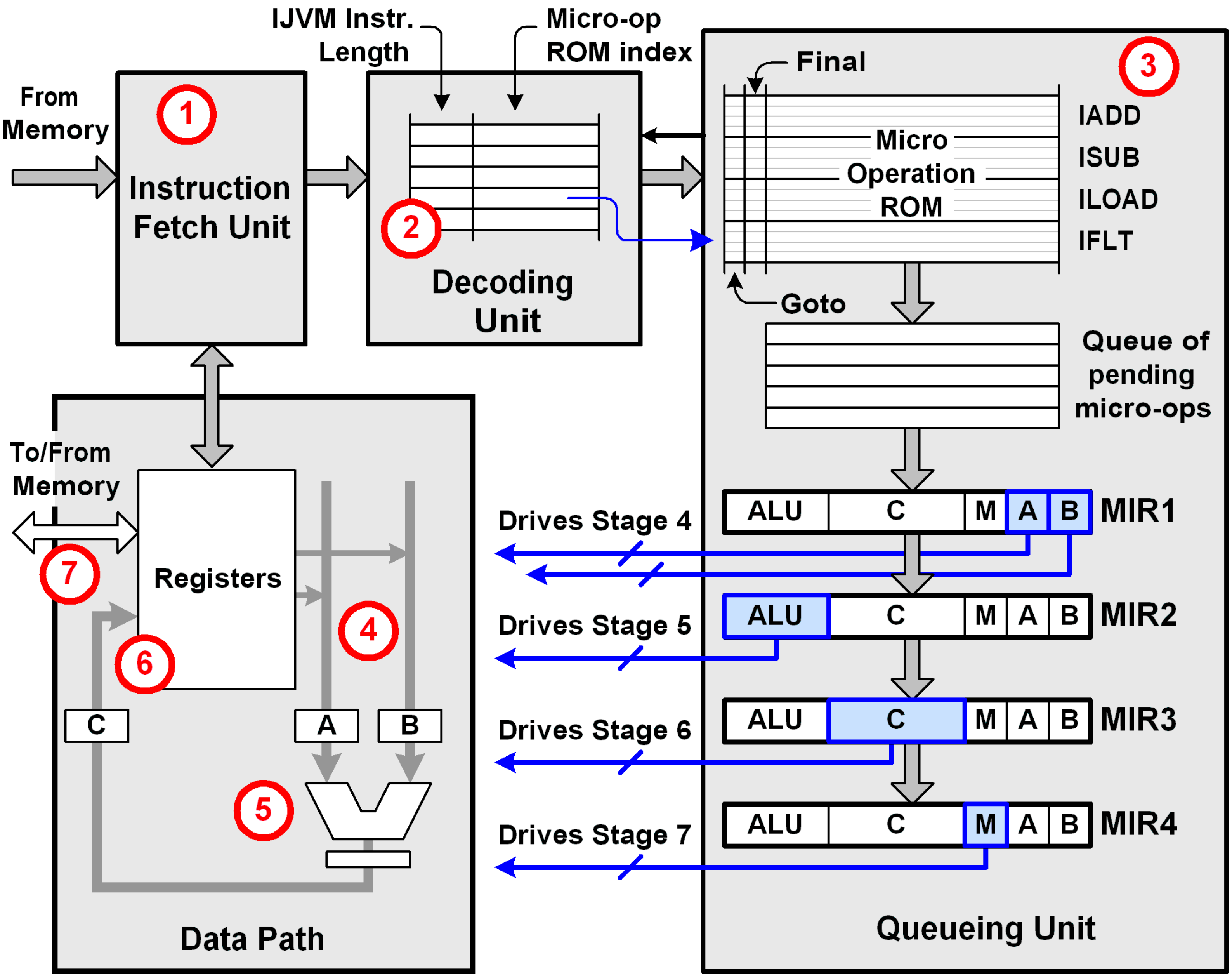
	Stack
4029	
SP 4028 → 4028	-20
4027	15
4026	7

Is the following microcode correct (A) or not (B)?

	Load Inputs	ALU Operation	Store Outputs
1	B = TOS		
2	A = SP	C = –B	
3		C = A	MDR = C
4			MAR = C; wr; goto (MBR1)

The MIC-4 - A Seven-Stage Pipeline

This is the textbook's last sample micro-architecture, with a seven-stage pipeline that has many of the same functional components as the Pentium-II.



The MIC-4 - A Seven-Stage Pipeline

We're going to discuss the MIC-4 in general terms only to give a sense of how it's more sophisticated control logic works.

Stage 1 - Instruction Fetch

- **The IFU in the MIC-4 works in a similar fashion to the MIC-2 and MIC-3, but it feeds bytes from the instruction stream to the Decoding Unit instead of the MBR1 / MBR2 registers.**

Stage 2 - Instruction Decode

- **The decode unit contains a read-only table with 256 entries, one for each IJVM opcode.**
- **Each entry contains the length of the corresponding IJVM instruction and an index into the micro-operation ROM (i.e., the ROM address of the first microinstruction to handle the IJVM instruction)**
- **The decode unit accepts an opcode from the IFU, looks up the appropriate table entry, and passes the micro-operation ROM address to the queuing stage.**
- **If the table entry indicates that the IJVM instruction is more than 1 byte long, the decode unit skips over the following non-opcode bytes until it reaches the next opcode.**
- **The WIDE prefix is handled by converting the following opcode (i.e., ILOAD or ISTORE) to a unique wide opcode value (i.e. "WIDE_ILOAD" or "WIDE_ISTORE"), which is then handled as if it were it's own special IJVM instruction.**

The MIC-4 - A Seven-Stage Pipeline

Stage 3 – Queuing Unit

- **Unlike the MIC-1 through MIC-3, the microinstructions for a given IJVM instruction are stored in consecutive ROM locations. So there is no "Next Address" field in the microinstruction.**
- **Execution of an IJVM instruction starts with the microinstruction at the ROM address passed to the queuing unit from the instruction decoder, and then proceeds through sequential microinstructions in the ROM.**
- **A special "Final" bit is turned on for the last microinstruction of a sequence so the system knows when to finish executing the current IJVM instruction and start with the next one.**
- **A second special "Goto" bit identifies conditional branch microinstructions. These have a different format which includes a "next address" field. "Goto" microinstructions feed information back to the Decode unit to stall the pipeline in case it causes a IJVM branch (this results in the need to fetch a new set if IJVM instructions into the pipeline)**
- **To execute an IJVM instruction, it's sequence of microinstructions is copied into a queue**
- **On each clock cycle, microinstructions from the queue are moved through a series of microinstruction registers called MIR1, MIR2, MIR3 and MIR4**
- **Each of the MIR registers drives a different pipeline stage:**

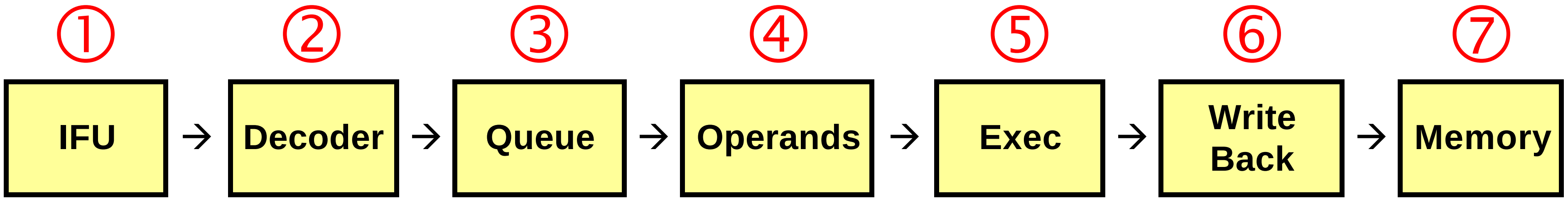
MIR1 – Drives the A and B buses	**MIR3 – Drives the C bus**
MIR2 – Controls the ALU operation	**MIR4 – Controls memory accesses**

- **As a given microinstruction moves through the different MIR registers, different parts of it are used to drive different parts of the data path in each clock cycle.**

The MIC-4 - A Seven-Stage Pipeline

4.4.5

The remaining four stages work the same as in the MIC-3 microarchitecture, except that we've noted memory accesses as a separate, final stage:



Not all of these stages process information at the same rate. For example, the IFU and decoder stages are often idle because they can fetch IJVM instructions faster than the following stages can execute the sequences of microinstructions for each one.

The MIC-4 design has many similarities to the Pentium-II and later Pentium microarchitectures. They take a stream of complex ISA instructions, break them down into a series of simpler micro operations, and then execute the micro operations in a data path.

Most RISC systems don't need to be this complex. For example, the ISA instructions used by the UltraSPARC II are fixed length, so there's no need for the decoder to determine where the instructions start.

Improving Performance

We’ve seen how the use of various techniques can used to improve the performance of our MIC system:

- Merging the main interpretation loop with the microcode (MIC-1)
- Using a 3-bus architecture (MIC-2)
- Building an Instruction Fetch Unit   (MIC-2)
- Pipelining the data path (MIC-3 and MIC-4)

All of these changes have been changes to the implementation of the MIC architecture. All of the improvements we made to the MIC system have been capable of executing the same IJVM instructions.   This is similar in concept to the evolution of the Intel processors from the 8088 through the 80286, 80386, 80486, and Pentium through Pentium 4 processors.

Performance improvements can also be made by changing the architecture of the system. Architectural changes are not compatible with older processors – they have differences at the ISA level that prevent them from executing the same programs that older processors did.

Because architectural changes require that new programs be written, they don’t happen very often.   They can only be justified if they can present a significant gain in performance, functionality, or economy.

RISC processors are an example of an architectural break with the past.   RISC has spawned a series of processor architectures which aim to use new design principles to create processors that are faster and/or cheaper than previous CISC designs.

Key Concepts

- How branch instructions are executed
- How conditional branches are executed
- Ways to improve performance:
  - ➢ Speed up the data path
  - ➢ Reduce the execution path length
  - ➢ Use parallelism
- Adding a third bus to eliminate the need to load the H register
- Operation of the instruction fetch unit
- Using a Finite State Machine to describe the operation of the IFU
- Adding A / B / C registers to create a pipeline
- Why the pipelined microarchitecture is faster even though it requires more clock cycles to execute a given IJVM instruction
- Writing microcode for the pipelined microarchitecture

What’s Next

- **Look at Review Questions for this week**
- **Sign on to WebCT and do Module 7 – “Shift and Boolean Instructions”**
- **Study for Quiz 5, which includes:**
  - **Week 9 lecture**
  - **WebCT module 7**
- **Continue working on Assignment 2 (for the material covered so far)**
- **Pre-read the material for Week 10**