Microarchitecture Design & Operation

Suggested Reading: Chapter 4 from Tanenbaum, Structured Computer Organization, 5/6ed, Pearson

IJVM

THE MICROPROGRAM

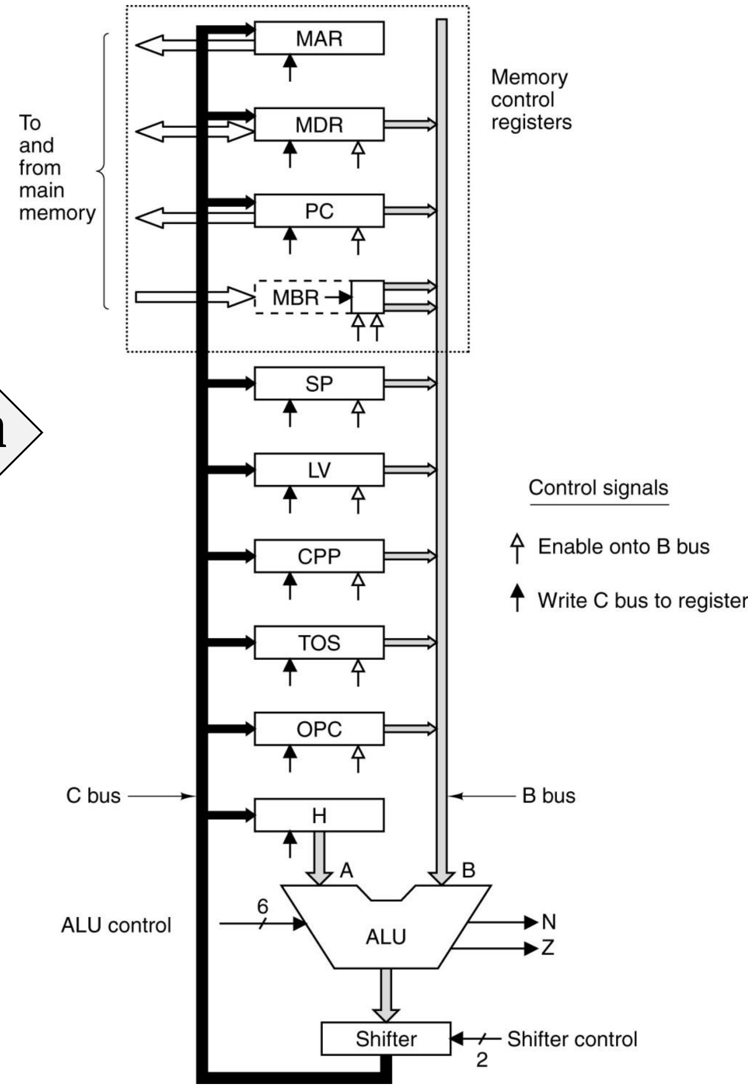
Objective

At the end of this module you will be able to

1. Understand, develop, and extend the Mic-1 microprogram

Hex	Mnemonic
0x10	BIPUSH *byte*
0x59	DUP
0xA7	GOTO *offset*
0x60	IADD
0x7E	IAND
0x99	IFEQ *offset*
0x9B	IFLT *offset*
0x9F	IF_ICMPEQ *offset*
0x84	IINC *varnum const*
0x15	ILOAD *varnum*
0xB6	INVOKEVIRTUAL *disp*
0x80	IOR
0xAC	IRETURN
0x36	ISTORE *varnum*
0x64	ISUB
0x13	LDC_W *index*
0x00	NOP
0x57	POP
0x5F	SWAP
0xC4	WIDE

microprogram



Invariants

The following are always maintained

1. SP points at the top of the stack, before and after each ISA instruction
2. TOS always contains the value at the top of the stack, before and after each ISA instruction
3. Each MI always fetches the opcode for the following MI in the program

Implementation of IJVM Using the Mic-1 (1)

Label	Operations	Comments
Main1	PC = PC + 1; fetch; goto (MBR)	MBR holds opcode; get next byte; dispatch
nop1	goto Main1	Do nothing
iadd1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
iadd2	H = TOS	H = top of stack
iadd3	MDR = TOS = MDR + H; wr; goto Main1	Add top two words; write to top of stack
isub1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
isub2	H = TOS	H = top of stack
isub3	MDR = TOS = MDR – H; wr; goto Main1	Do subtraction; write to top of stack
iand1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
iand2	H = TOS	H = top of stack
iand3	MDR = TOS = MDR AND H; wr; goto Main1	Do AND; write to new top of stack
ior1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
ior2	H = TOS	H = top of stack
ior3	MDR = TOS = MDR OR H; wr; goto Main1	Do OR; write to new top of stack
dup1	MAR = SP = SP + 1	Increment SP and copy to MAR
dup2	MDR = TOS; wr; goto Main1	Write new stack word
pop1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
pop2		Wait for new TOS to be read from memory
pop3	TOS = MDR; goto Main1	Copy new word to TOS
swap1	MAR = SP – 1; rd	Set MAR to SP – 1; read 2nd word from stack
swap2	MAR = SP	Set MAR to top word
swap3	H = MDR; wr	Save TOS in H; write 2nd word to top of stack
swap4	MDR = TOS	Copy old TOS to MDR
swap5	MAR = SP – 1; wr	Set MAR to SP – 1; write as 2nd word on stack
swap6	TOS = H; goto Main1	Update TOS

The microprogram for the Mic-1

Implementation of IJVM Using the Mic-1 (2)

bipush1	SP = MAR = SP + 1	MBR = the byte to push onto stack
bipush2	PC = PC + 1; fetch	Increment PC, fetch next opcode
bipush3	MDR = TOS = MBR; wr; goto Main1	Sign-extend constant and push on stack
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
istore1	H = LV	MBR contains index; Copy LV to H
istore2	MAR = MBRU + H	MAR = address of local variable to store into
istore3	MDR = TOS; wr	Copy TOS to MDR; write word
istore4	SP = MAR = SP – 1; rd	Read in next-to-top word on stack
istore5	PC = PC + 1; fetch	Increment PC; fetch next opcode
istore6	TOS = MDR; goto Main1	Update TOS
wide1	PC = PC + 1; fetch;	Fetch operand byte or next opcode
wide2	goto (MBR OR 0x100)	Multiway branch with high bit set
wide_iload1	PC = PC + 1; fetch	MBR contains 1st index byte; fetch 2nd
wide_iload2	H = MBRU << 8	H = 1st index byte shifted left 8 bits
wide_iload3	H = MBRU OR H	H = 16-bit index of local variable
wide_iload4	MAR = LV + H; rd; goto iload3	MAR = address of local variable to push
wide_istore1	PC = PC + 1; fetch	MBR contains 1st index byte; fetch 2nd
wide_istore2	H = MBRU << 8	H = 1st index byte shifted left 8 bits
wide_istore3	H = MBRU OR H	H = 16-bit index of local variable
wide_istore4	MAR = LV + H; goto istore3	MAR = address of local variable to store into
ldc_w1	PC = PC + 1; fetch	MBR contains 1st index byte; fetch 2nd
ldc_w2	H = MBRU << 8	H = 1st index byte << 8
ldc_w3	H = MBRU OR H	H = 16-bit index into constant pool
ldc_w4	MAR = H + CPP; rd; goto iload3	MAR = address of constant in pool

The microprogram for the Mic-1

Implementation of IJVM Using the Mic-1 (3)

Label	Operations	Comments
iinc1	H = LV	MBR contains index; Copy LV to H
iinc2	MAR = MBRU + H; rd	Copy LV + index to MAR; Read variable
iinc3	PC = PC + 1; fetch	Fetch constant
iinc4	H = MDR	Copy variable to H
iinc5	PC = PC + 1; fetch	Fetch next opcode
iinc6	MDR = MBR + H; wr; goto Main1	Put sum in MDR; update variable
goto1	OPC = PC – 1	Save address of opcode.
goto2	PC = PC + 1; fetch	MBR = 1st byte of offset; fetch 2nd byte
goto3	H = MBR << 8	Shift and save signed first byte in H
goto4	H = MBRU OR H	H = 16-bit branch offset
goto5	PC = OPC + H; fetch	Add offset to OPC
goto6	goto Main1	Wait for fetch of next opcode
iflt1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
iflt2	OPC = TOS	Save TOS in OPC temporarily
iflt3	TOS = MDR	Put new top of stack in TOS
iflt4	N = OPC; if (N) goto T; else goto F	Branch on N bit
ifeq1	MAR = SP = SP – 1; rd	Read in next-to-top word of stack
ifeq2	OPC = TOS	Save TOS in OPC temporarily
ifeq3	TOS = MDR	Put new top of stack in TOS
ifeq4	Z = OPC; if (Z) goto T; else goto F	Branch on Z bit

The microprogram for the Mic-1

Implementation of IJVM Using the Mic-1 (4)

if_icmpeq1	MAR = SP = SP – 1; rd	Read in next-to-top word of stack
if_icmpeq2	MAR = SP = SP – 1	Set MAR to read in new top-of-stack
if_icmpeq3	H = MDR; rd	Copy second stack word to H
if_icmpeq4	OPC = TOS	Save TOS in OPC temporarily
if_icmpeq5	TOS = MDR	Put new top of stack in TOS
if_icmpeq6	Z = OPC – H; if (Z) goto T; else goto F	If top 2 words are equal, goto T, else goto F
T	OPC = PC – 1; goto goto2	Same as goto1; needed for target address
F	PC = PC + 1	Skip first offset byte
F2	PC = PC + 1; fetch	PC now points to next opcode
F3	goto Main1	Wait for fetch of opcode
invokevirtual1	PC = PC + 1; fetch	MBR = index byte 1; inc. PC, get 2nd byte
invokevirtual2	H = MBRU << 8	Shift and save first byte in H
invokevirtual3	H = MBRU OR H	H = offset of method pointer from CPP
invokevirtual4	MAR = CPP + H; rd	Get pointer to method from CPP area
invokevirtual5	OPC = PC + 1	Save Return PC in OPC temporarily
invokevirtual6	PC = MDR; fetch	PC points to new method; get param count
invokevirtual7	PC = PC + 1; fetch	Fetch 2nd byte of parameter count
invokevirtual8	H = MBRU << 8	Shift and save first byte in H
invokevirtual9	H = MBRU OR H	H = number of parameters
invokevirtual10	PC = PC + 1; fetch	Fetch first byte of # locals
invokevirtual11	TOS = SP – H	TOS = address of OBJREF – 1
invokevirtual12	TOS = MAR = TOS + 1	TOS = address of OBJREF (new LV)
invokevirtual13	PC = PC + 1; fetch	Fetch second byte of # locals
invokevirtual14	H = MBRU << 8	Shift and save first byte in H
invokevirtual15	H = MBRU OR H	H = # locals

The microprogram for the Mic-1

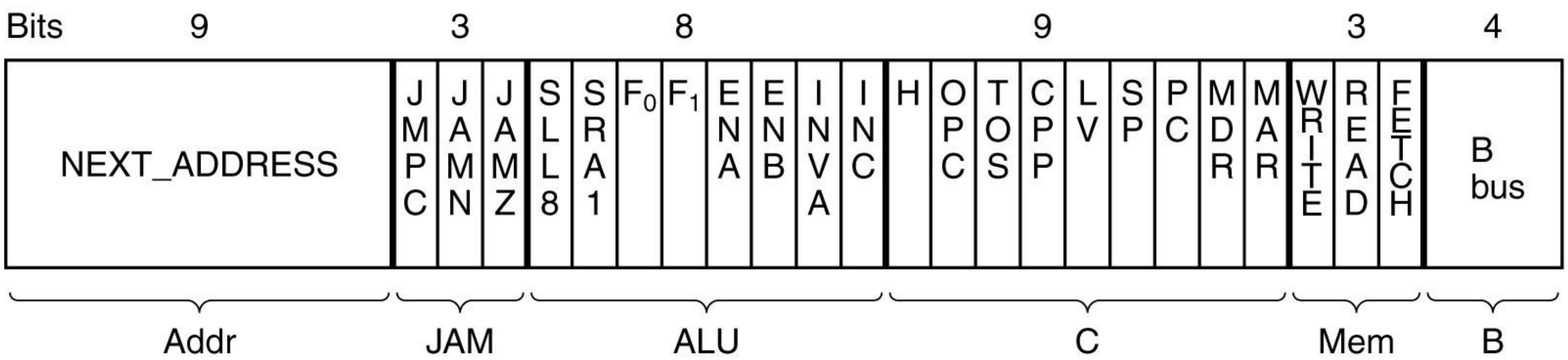
Implementation of IJVM Using the Mic-1 (5)

Label	Operations	Comments
invokevirtual16	MDR = SP + H + 1; wr	Overwrite OBJREF with link pointer
invokevirtual17	MAR = SP = MDR;	Set SP, MAR to location to hold old PC
invokevirtual18	MDR = OPC; wr	Save old PC above the local variables
invokevirtual19	MAR = SP = SP + 1	SP points to location to hold old LV
invokevirtual20	MDR = LV; wr	Save old LV above saved PC
invokevirtual21	PC = PC + 1; fetch	Fetch first opcode of new method.
invokevirtual22	LV = TOS; goto Main1	Set LV to point to LV Frame
ireturn1	MAR = SP = LV; rd	Reset SP, MAR to get link pointer
ireturn2		Wait for read
ireturn3	LV = MAR = MDR; rd	Set LV to link ptr; get old PC
ireturn4	MAR = LV + 1	Set MAR to read old LV
ireturn5	PC = MDR; rd; fetch	Restore PC; fetch next opcode
ireturn6	MAR = SP	Set MAR to write TOS
ireturn7	LV = MDR	Restore LV
ireturn8	MDR = TOS; wr; goto Main1	Save return value on original top of stack

The microprogram for the Mic-1

Microinstructions

- Best way to specify microinstructions is to give the 36 bit stream



- However, this is **tough**
- We will give a HL notation
  - Better for us to understand

Notation

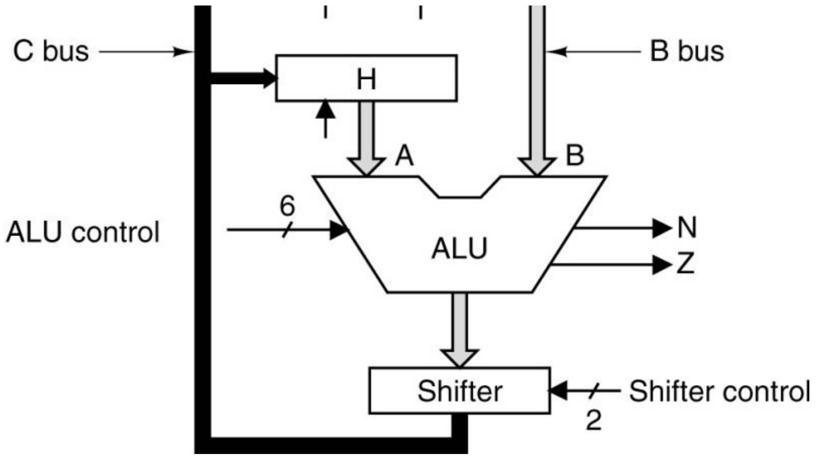
- Each line: activities in a single clock cycle
- SP = SP + 1; rd
    - Read SP onto B-Bus
    - ALU: add 1 operation
    - Store back in SP
    - Initiate a memory read instruction
        - MDR = Memory[MAR]

Notation

- SP = MDR = SP +1
    - B = SP
    - ALU + 1
    - Store result in SP and MDR
- MDR = SP; wr
    - Copy SP into MDR
    - Start a memory write
        - Memory[MAR] = MDR

Invalid Instructions

- MDR = SP + MDR
- Cannot be executed in one cycle
- Need 2 cycles:
  - H = MDR (alternatively H = SP)
  - MDR = SP + H



Invalid Instructions

- $H = H - MDR$
  - Why illegal?

F_0	F_1	ENA	ENB	INVA	INC	Function
0	1	1	0	0	0	A
0	1	0	1	0	0	B
0	1	1	0	1	0	$\overline{A}$
1	0	1	1	0	0	$\overline{B}$
1	1	1	1	0	0	A + B
1	1	1	1	0	1	A + B + 1
1	1	1	0	0	1	A + 1
1	1	0	1	0	1	B + 1
1	1	1	1	1	1	B – A
1	1	0	1	1	0	B – 1
1	1	1	0	1	1	–A
0	0	1	1	0	0	A AND B
0	1	1	1	0	0	A OR B
0	1	0	0	0	0	0
1	1	0	0	0	1	1
1	1	0	0	1	0	–1

ALU functions

Invalid Instructions

MAR = SP; rd

MDR = H

- Illegal:
  - rd => MDR = Memory[MAR]
  - Completes at the end of second microinstruction
  - Second microinstruction also assigns value to MDR
  - Result: unknown value in MDR
- Microassembler must reject such statements

Permitted Microinstructions

All permitted operations.

SOURCE in {MDR, PC, MBR, MBRU, SP, LV, CPP, TOS, OPC} (to B bus)
DEST in {MAR, MDR, PC, SP, LV, CPP, TOS, OPC, H} (from C bus)
Also **rd, wr, fetch**

Any of the above operations may be extended by adding "<< 8" to them to shift the result left by 1 byte. For example, a common operation is
H = MBR << 8

DEST = H
DEST = SOURCE
DEST = $\overline{\text{H}}$
DEST = $\overline{\text{SOURCE}}$
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

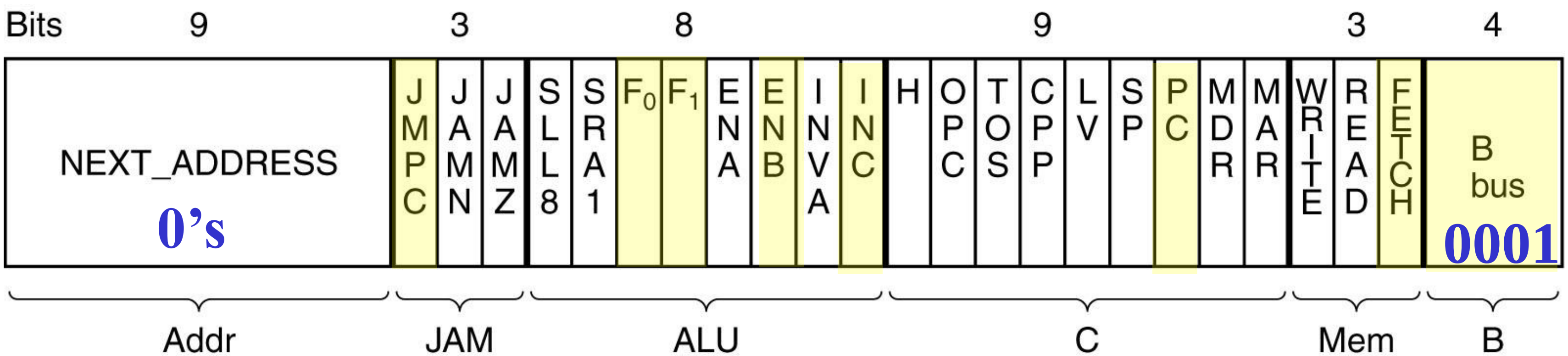
Interpreter

```
PC = starting_address;
while (run_bit) {
    instr = memory[PC];                      // fetch next instruction into instr
    PC = PC + 1;                             // increment program counter
    instr_type = get_instr_type(instr);      // determine instruction type
    data_loc = find_data(instr, instr_type); // locate data (–1 if none)
    if (data_loc >= 0)                       // if data_loc is –1, there is no operand
        data = memory[data_loc];             // fetch the data
    execute(instr_type, data);               // execute instruction
}

}

private static int get_instr_type(int addr) { ... }
private static int find_data(int instr, int type) { ... }
private static void execute(int type, int data) { ... }
}
```

An interpreter for a simple computer (written in Java).

Interpretation

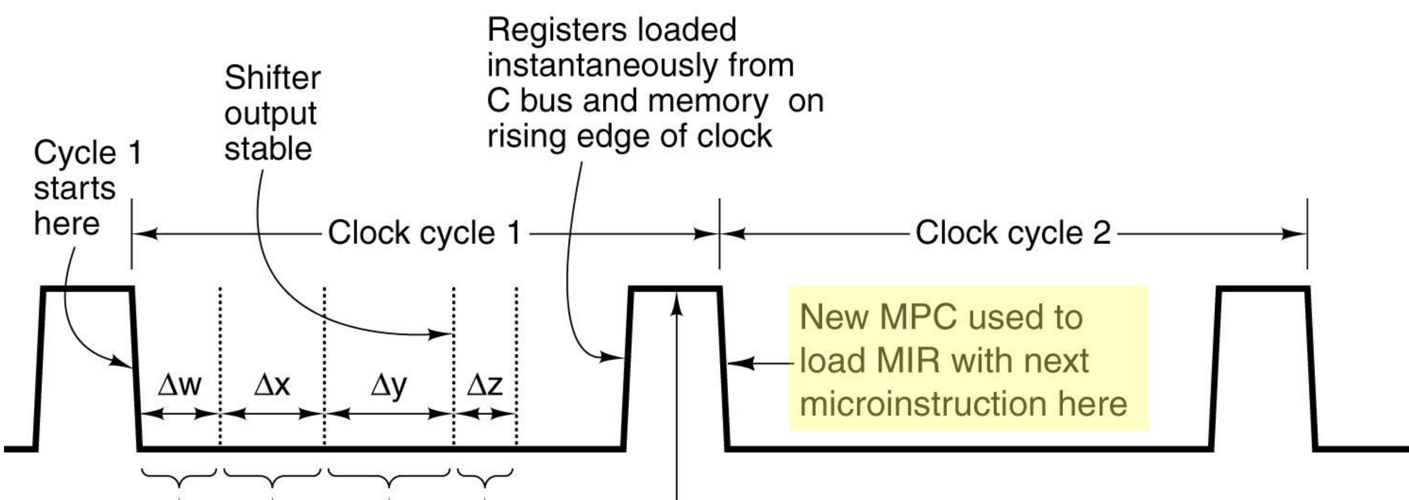
main1 PC = PC + 1; fetch; goto(MBR)



MPC = NEXT_ADDRESS OR MBR = MBR

B bus registers

: MDR	5 = LV
: PC	6 = CPP
: MBR	7 = TOS
: MBRU	8 = OPC
: SP	9 -15 none



Microstore

Address	
…	
0x10	
…	
0x57	Start of microcode to interpret POP
0x58	?
0x59	Start of microcode to interpret DUP

- POP needs three microinstructions
- Cannot be fit contiguously in microstore
- Every MI must point to its successor

NEXT_ADDRESS

- Last MI that interprets an ISA instruction has Main1 as its successor

Address	Microinstruction
	First microcode to interpret POP
…	
?	**Second** microcode to interpret POP
…	
?	**Third** microcode to interpret POP
…	
Main1	PC = PC+1; fetch; goto(MBR)

Interpretation

Main loop of the micro program

main1 PC = PC + 1; fetch; goto(MBR)

main1: label (address)

goto: unconditional branch

Assumes PC has been fetched previously and points to first byte in method area

Typically first byte in main method

PC is incremented and next byte is fetched

Next byte will be ready in MBR at the end of the next microinstruction

May be needed by the third microinstruction

Interpretation

main1 PC = PC + 1; fetch; goto(MBR)

MBR = POP = 0x57

MPC = 0x57

For the next MI

JMPC

- If JMPC == 1

  MBR OR (bitwize) NEXT_ADDRESS[0..7]

  Else

  Pass NEXT_ADDRESS as is from previous MI

NOP

No Operation

nop1 goto main1

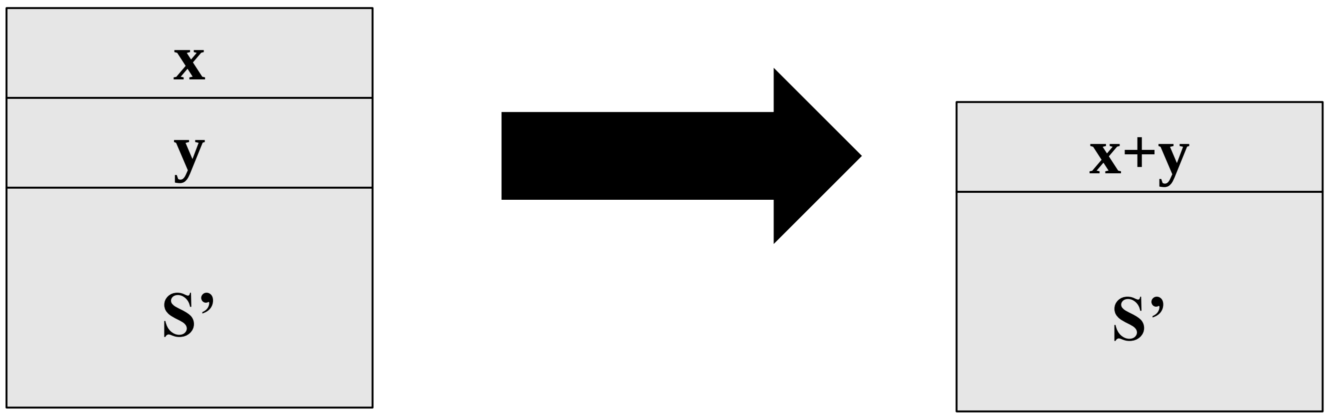
Do nothing

NEXT_ADDRESS	JMPC	JAMN	JAMZ	SLL8	SRA1	F_0	F_1	ENA	ENB	INVA	INC	H	OPC	TOS	CPP	LV	SP	PC	MDR	MAR	WRITE	READ	FETCH	B bus
main1																								

First microinstruction to be executed

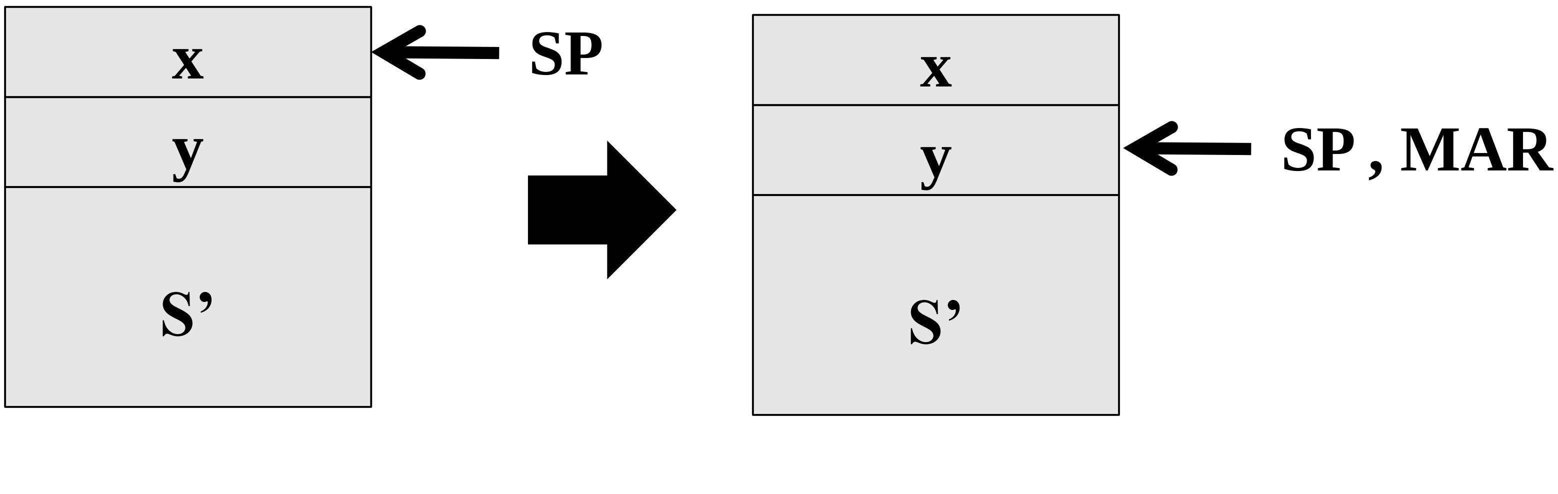
IADD

Pop 2 words from stack and add them; push result onto stack



IADD – cycle 1

Pop 2 words from stack and add them; push result onto stack

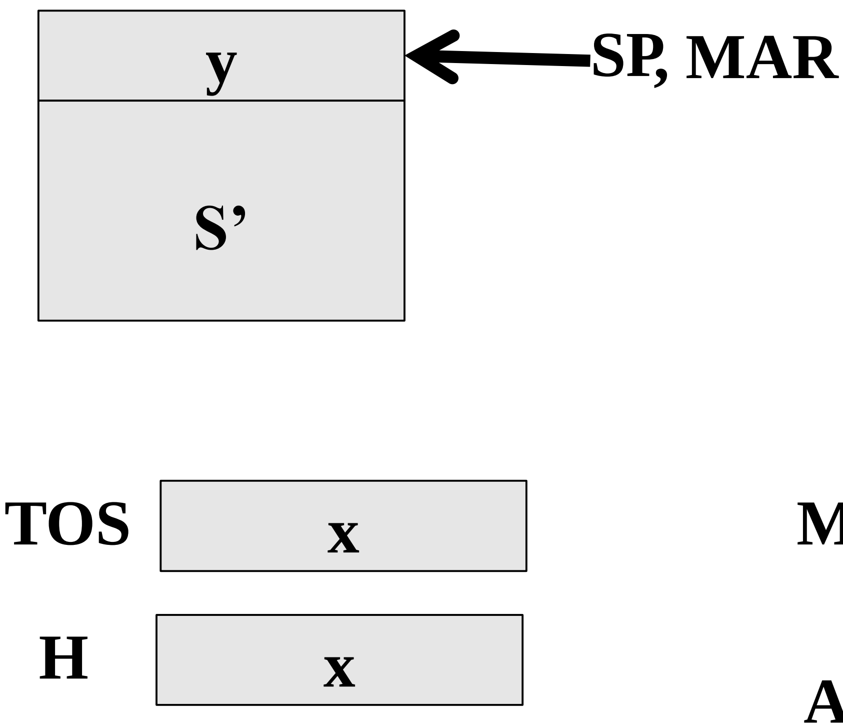


Need to read y to DP

Read: MDR = Memory[MAR] = y

IADD – cycle 2

Pop 2 words from stack and add them; push result onto stack



Wait for the read to finish

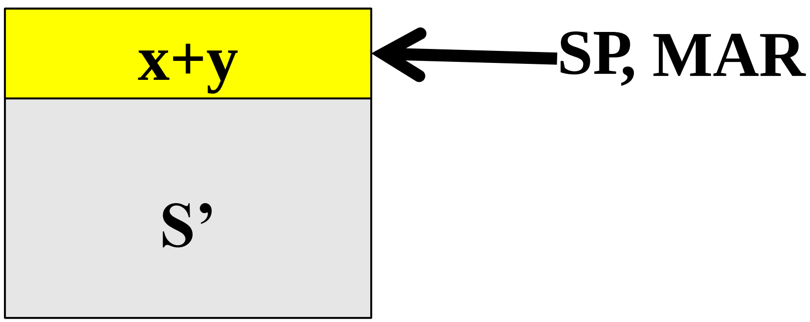
Mean while copy TOS to H to perform A+B (TOS=x+MDR=y)

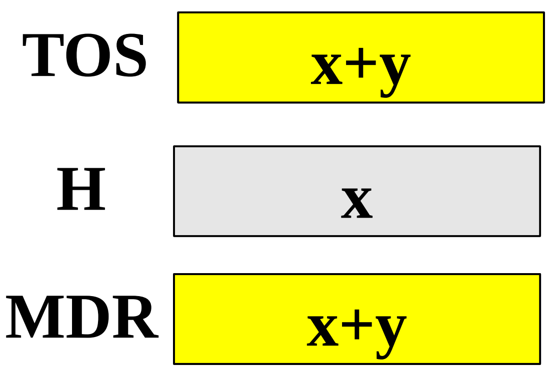
At the beginning of cycle 3, MDR = y

IADD – cycle 3

Pop 2 words from stack and add them; push result onto stack

End of cycle 4





Add MDR and H (x+y)

Need to write (x+y) back to memory:
MDR = (x+y) write

Also update TOS to (x+y), new top of stck

IADD

Pop 2 words from stack and add them; push result onto stack

iadd1 MAR = SP = SP – 1; rd; goto iadd2

iadd2 H = TOS; goto iadd3

iadd3 MDR = TOS= MDR+H; wr; goto main1

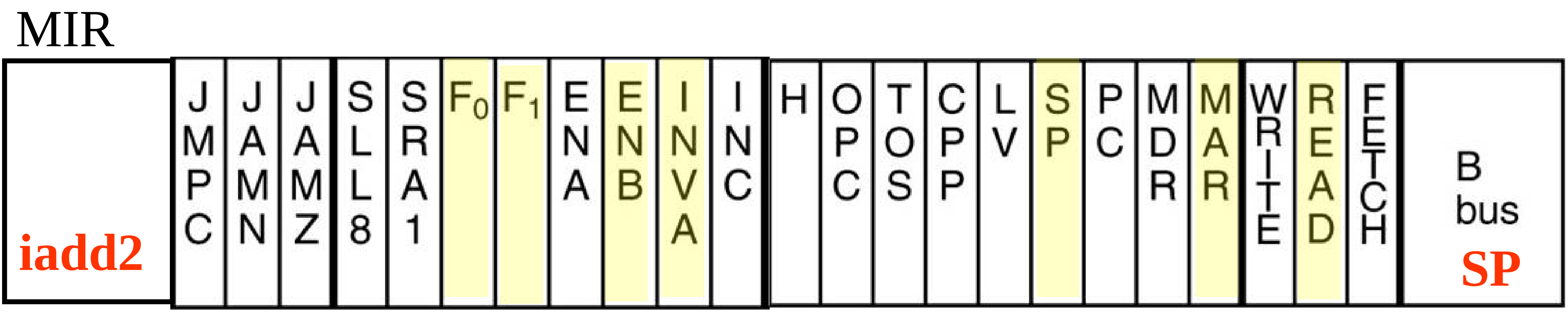
From now on implicit goto's will be omitted

Note: iadd1 = 0x60

IADD – iadd1

Pop 2 words from stack and add them; push result onto stack

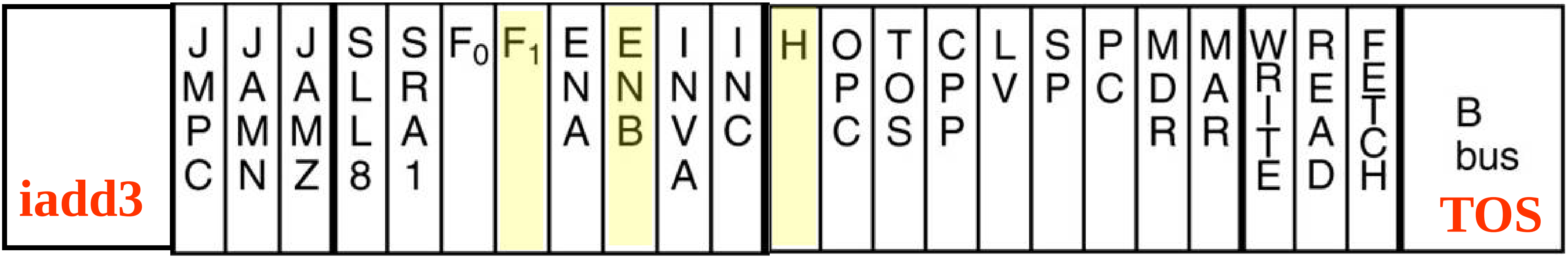
iadd1 MAR = SP = SP – 1; rd; goto iadd2



IADD – iadd2

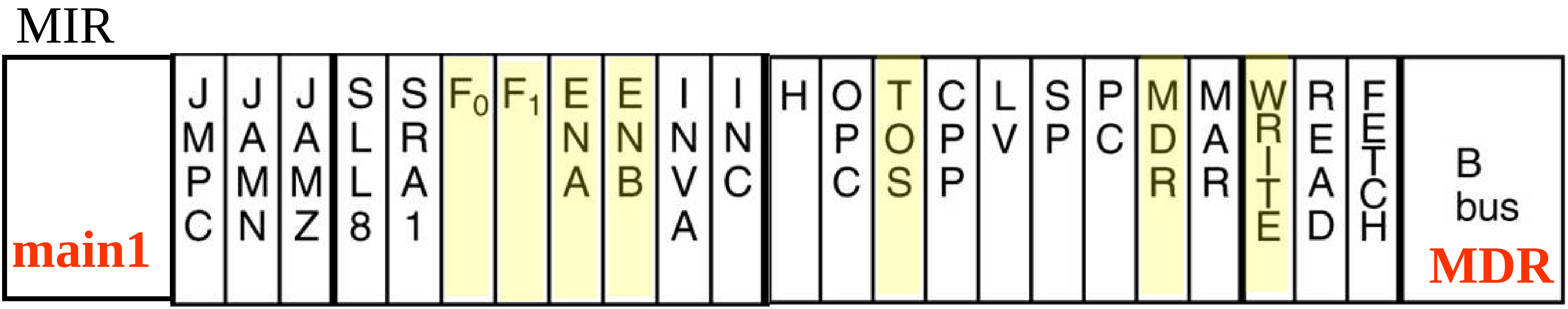
iadd2 H = TOS; goto iadd3

MIR



IADD – iadd3

iadd3 MDR = TOS= MDR+H; wr; goto main1



ISUB

Pop 2 words from stack and subtract them; push result onto stack

isub1 MAR = SP = SP – 1; rd;

isub2 H = TOS;

isub3 MDR = TOS= MDR – H ; wr; goto main1

Note: isub1 = 0x64

IAND

Pop 2 words from stack and and them; push result onto stack

iand1 MAR = SP = SP – 1; rd;

iand2 H = TOS;

iand3 MDR = TOS= MDR AND H ; wr; goto main1

Note: iand1 = 0x7E

IOR

Pop 2 words from stack and or them; push result onto stack

ior1 MAR = SP = SP – 1; rd;

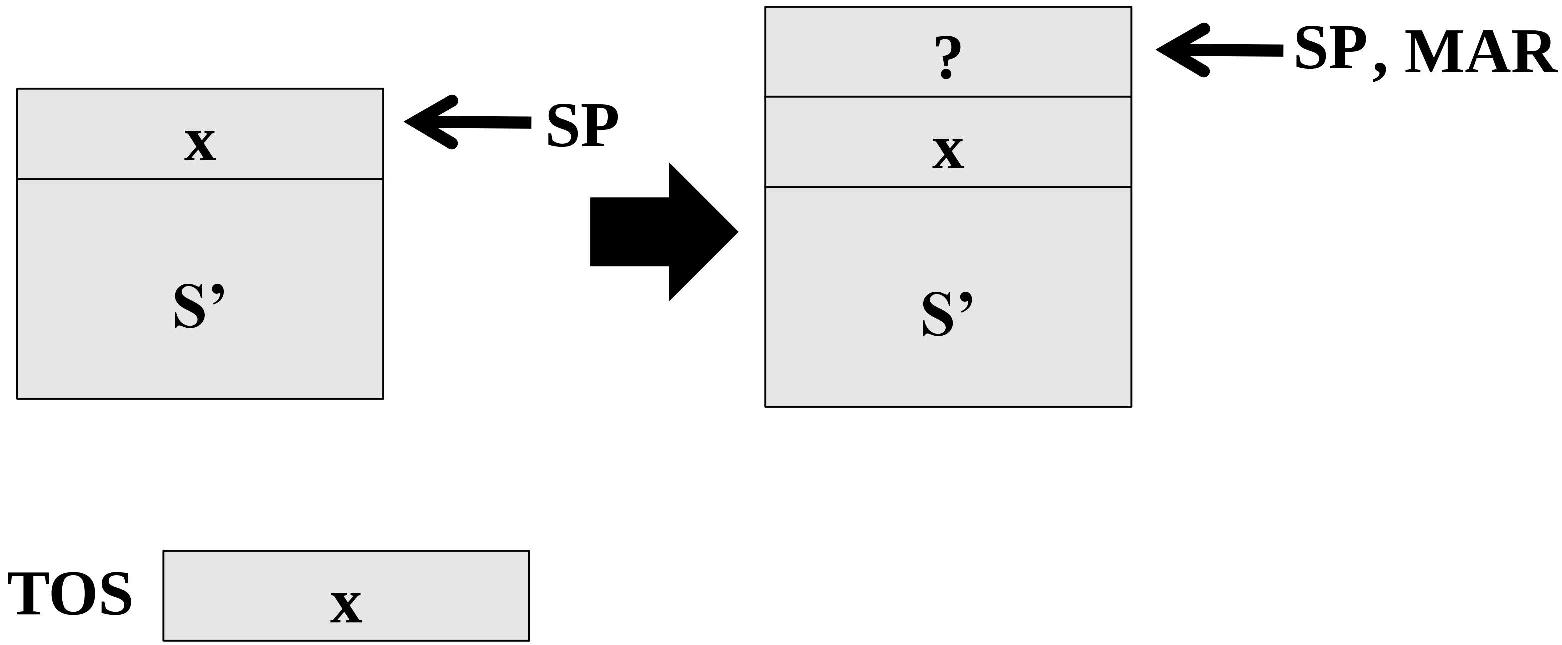
ior2 H = TOS;

ior3 MDR = TOS= MDR OR H ; wr; goto main1

Note: ior1 = 0x80

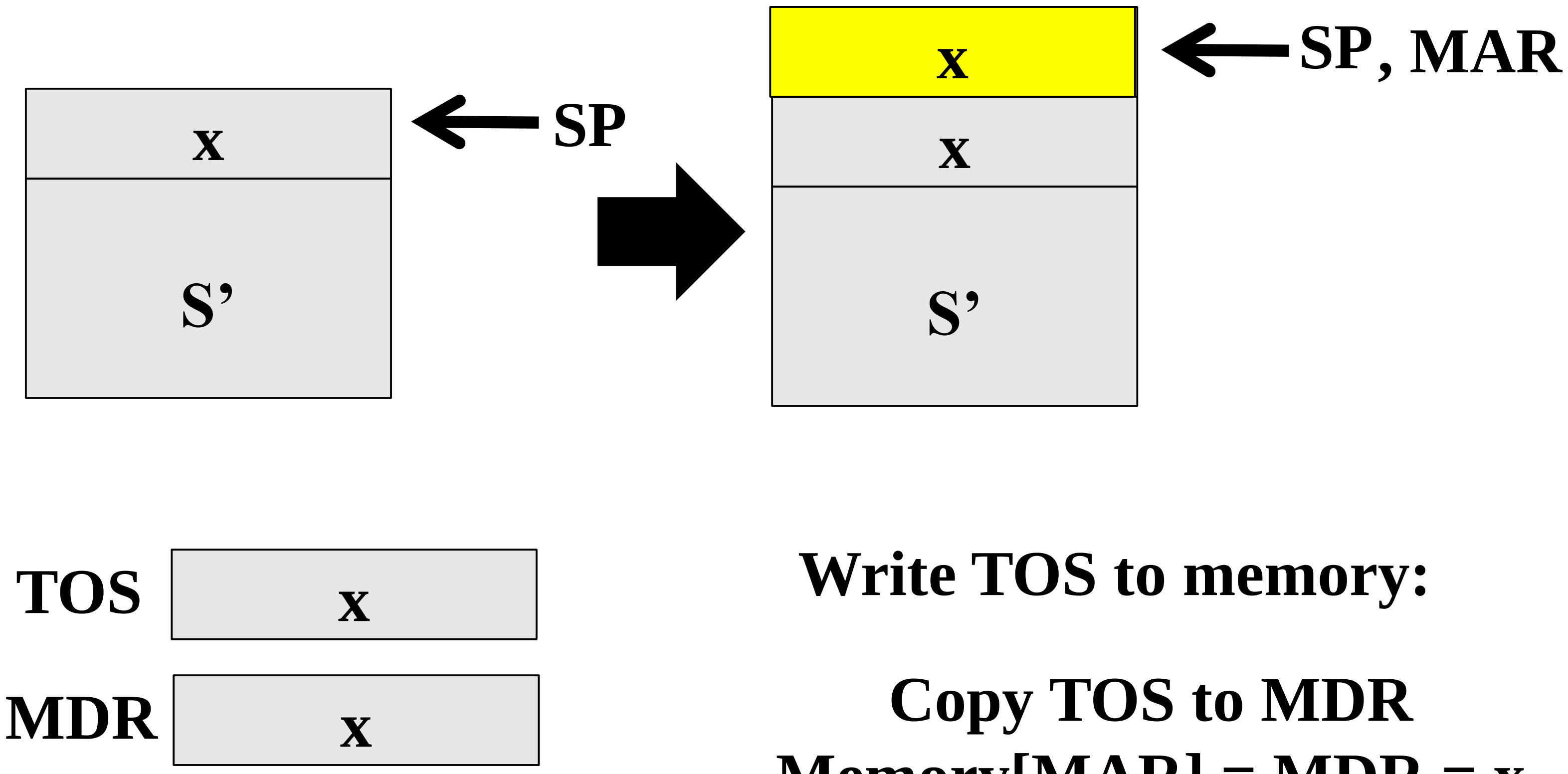
DUP – cycle 1

Duplicate top word on stack and push onto stack



DUP – cycle 2

Duplicate top word on stack and push onto stack



Write TOS to memory:

Copy TOS to MDR
Memory[MAR] = MDR = x

DUP

Duplicate top word on stack and push onto stack

```
dup1 MAR = SP = SP + 1;
dup2 MDR = TOS; wr; goto main1
```

Note: dup1 = 0x59

POP

Delete word from stack

pop1 MAR = SP = SP – 1; rd;

pop2

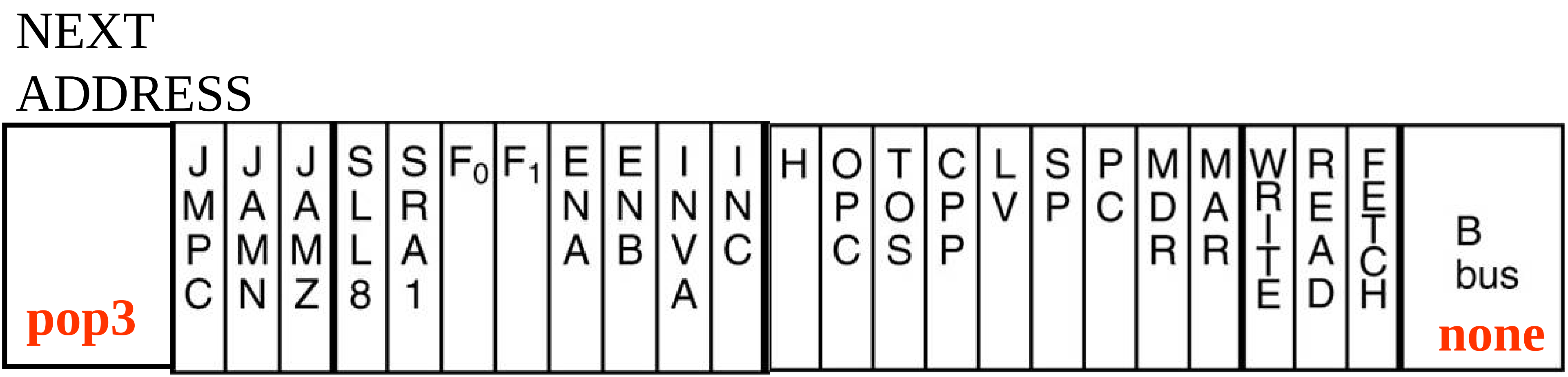
pop3 TOS= MDR; goto main1

Note: pop1 = 0x57

POP – pop2

pop2

Simply: goto pop3



SWAP

Swap top of stack word with next-to-top word

swap1 MAR = SP – 1; rd;

swap2 MAR = SP

swap3 H = MDR; wr

swap4 MDR = TOS

swap5 MAR = SP – 1; wr

swap6 TOS = H; goto main1

Note: swap1 = 0x5F

Exercise

- Write an implementation in MAL for a hypothetical IJVM instruction DNTT
  - Duplicate Next To Top
  - Duplicates the next-to-top value on the stack and push onto stack

Instructions with Operands

Fetching a head

…

1) IADD

2) DUP

3) ISUB

main1 PC = PC + 1; fetch; goto(MBR)

PC = 1, MBR = IADD

PC = 2, fetch DUP to MBR

Interpret IADD

End of add1: MBR = DUP

When IADD finishes, goto main1

PC = 3, fetch ISUB, goto dup1

End of dup1: MBR = ISUB

BIPUSH *byte*

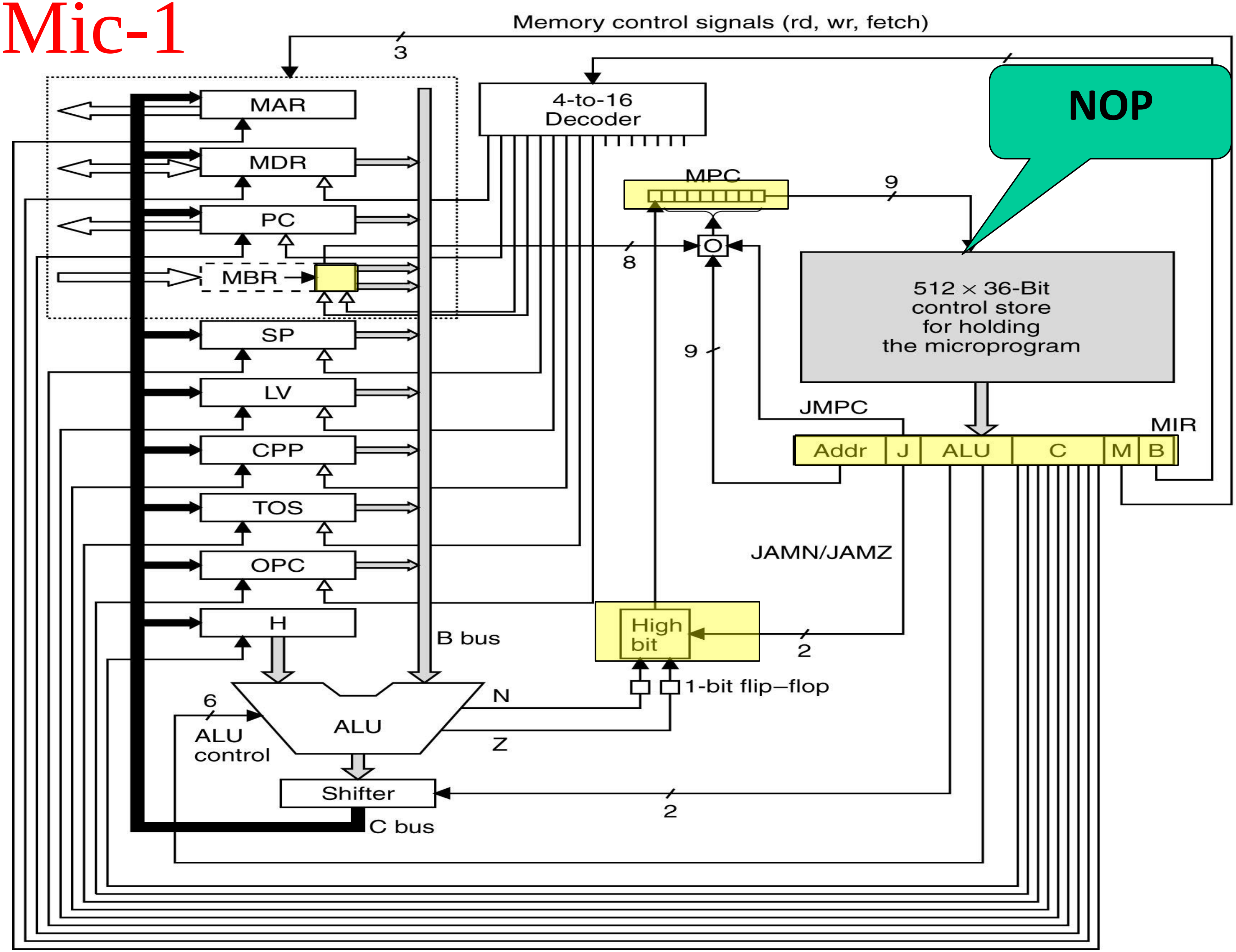
Push *byte* onto stack

BIPUSH 1

DUP

What does MBR contain?

Mic-1



Memory control signals (rd, wr, fetch)
3
NOP
MAR
MDR
PC
MBR
SP
LV
CPP
TOS
OPC
H
4-to-16
Decoder
MPC
9
8
O
512 × 36-Bit
control store
for holding
the microprogram
9
JMPC
MIR
Addr
J
ALU
C
M
B
JAMN/JAMZ
High
bit
2
1-bit flip–flop
B bus
N
Z
6
ALU
control
ALU
Shifter
2
C bus

nop1 goto main1

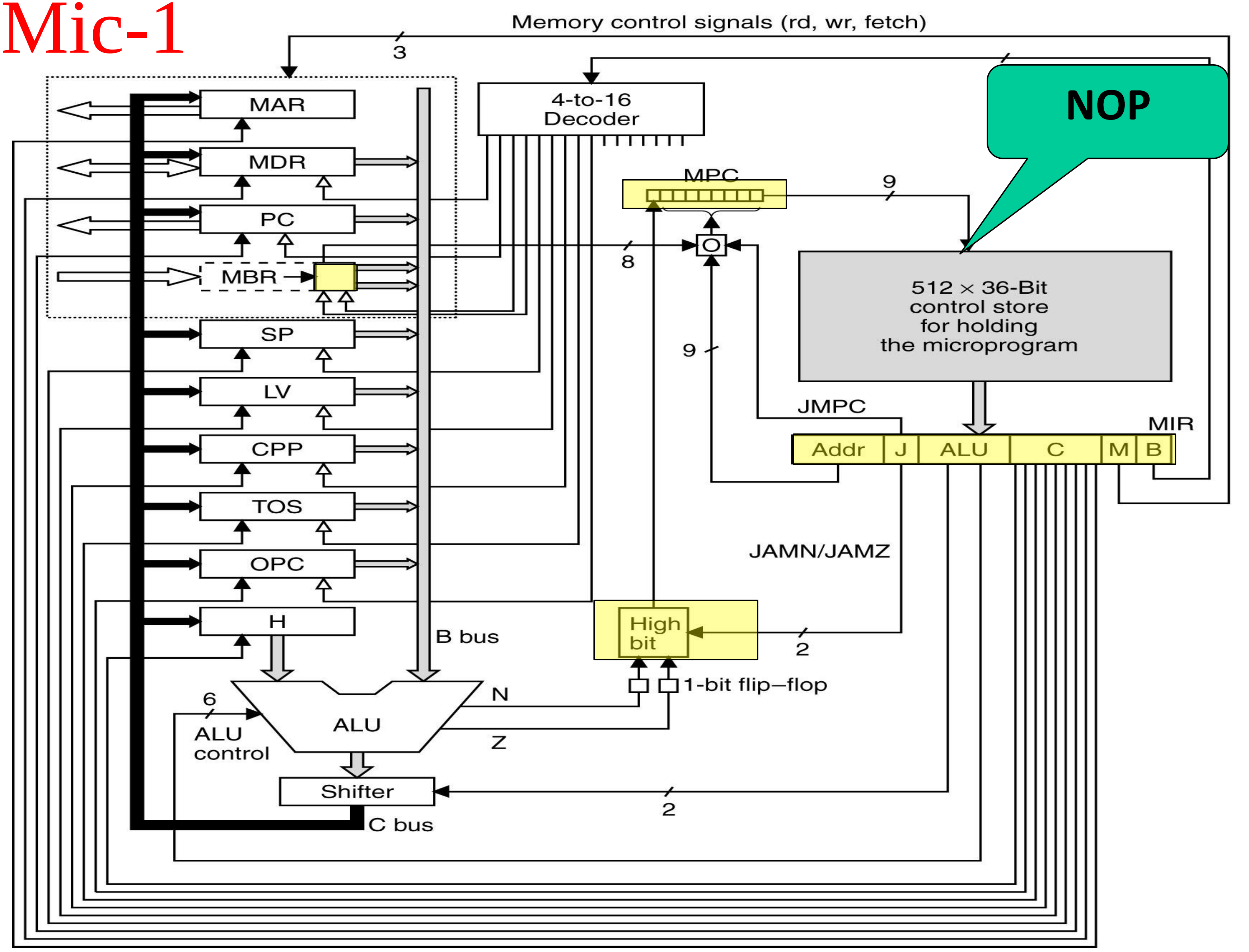
main1 PC = PC + 1; fetch; goto(MBR)

PC = 1, fetch the first opcode

MBR is still zero

MBR will contain the first opcode at the end of next cycle

Mic-1



Memory control signals (rd, wr, fetch)
3
MAR
MDR
PC
MBR
SP
LV
CPP
TOS
OPC
H
B bus
4-to-16
Decoder
MPC
9
8
O
9
NOP
512 × 36-Bit
control store
for holding
the microprogram
JMPC
MIR
Addr
J
ALU
C
M
B
JAMN/JAMZ
High
bit
2
1-bit flip–flop
N
Z
6
ALU
control
ALU
Shifter
2
C bus

BIPUSH

1

DUP

nop1 goto main1

main1 PC = PC + 1; fetch; goto(MBR)

PC = 2, fetch the second byte (which is operand 1)

MBR = BIPUSH, at the end of nop1

MBR will contain the operand 1 at the end of next cycle

BIPUSH *byte*

First attempt

BIPUSH

1

DUP

bipush1 SP = MAR = SP + 1

(now MBR = 1)

bipush2 MDR = TOS = MBR; wr; goto main1

main1 PC = PC + 1; fetch; goto(MBR)

Need another cycle before MBR contains next byte (DUP) in the ISA program

Goto (MBR) will go to 1, which is not a lead MI!

BIPUSH *byte*

Push *byte* onto stack

bipush1 SP = MAR = SP + 1

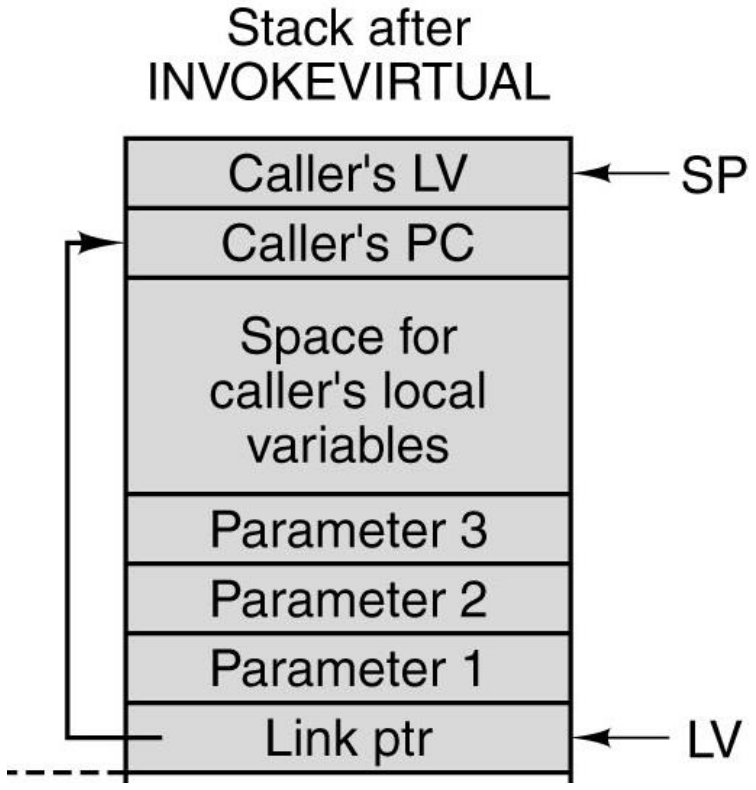
bipush2 PC = PC + 1; fetch

bipush3 MDR = TOS = MBR; wr; goto main1

Note: bipush1 = 0x10

ILOAD *varnum*

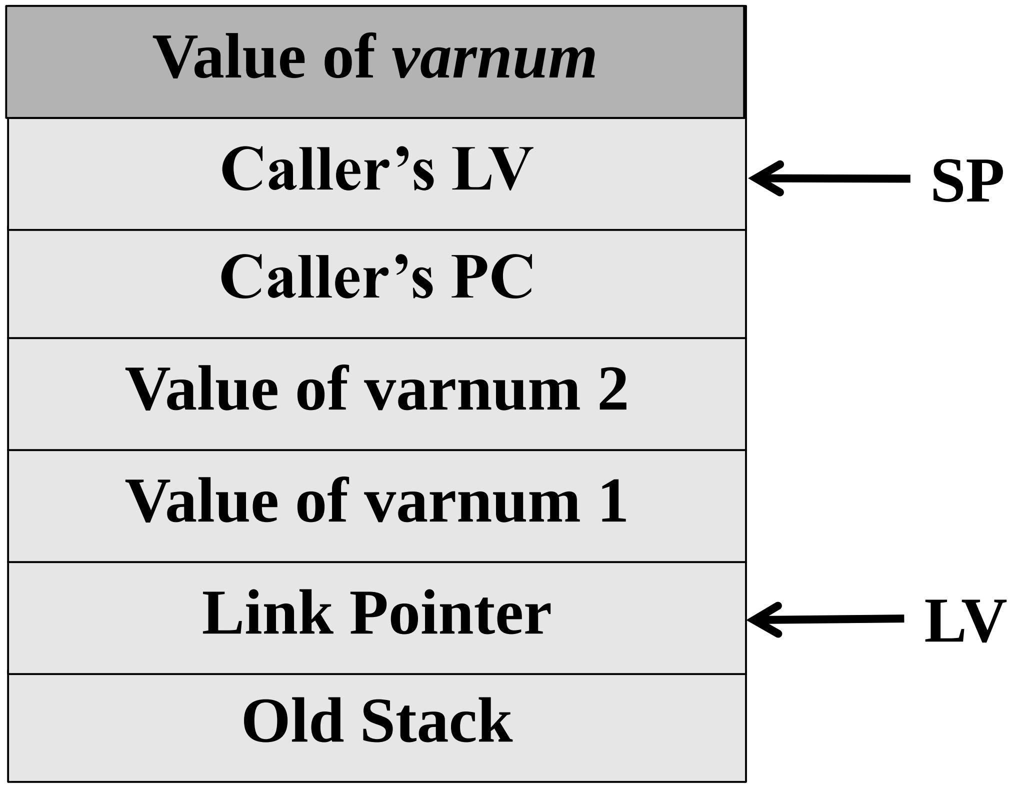
Push local variable onto stack, *varnum* is one byte (except in the WIDE format)



1. Local variables and parameters are treated similarly
2. Variables are referenced by LV + *varnum*
3. *varnum* 0 is OBJREF, which we do not use

ILOAD *varnum*

Push local variable onto stack, *varnum* is one byte (except in the WIDE format)



ILOAD *varnum*

Push local variable onto stack, *varnum* is one byte (except in the WIDE format)

iload1 H = LV

iload2 MAR = MBRU + H; rd

iload3 MAR = SP = SP + 1

iload4 PC = PC + 1; fetch; wr

iload5 TOS = MDR; goto main1

Note: iload1 = 0x15

ISTORE *varnum*

Pop word from stack and store it in a local variable

istore1 H = LV

istore2 MAR = MBRU + H

istore3 MDR = TOS; wr

istore4 SP = MAR = SP – 1; rd

istore5 PC = PC + 1; fetch

istore6 TOS = MDR; goto main1

Note: istore1 = 0x36

IINC

Add *const* to local variable, both operands are 1 byte each

IINC (0x84)	INDEX	CONST

The IINC instruction has two different operand fields.

IINC *varnum const*

Add *const* to local variable, both operands are 1 byte each

iinc1 H = LV

iinc2 MAR = MBRU + H; rd

iinc3 PC = PC + 1; fetch

iinc4 H = MDR

iinc5 PC = PC + 1; fetch

iinc6 MDR = MBR + H; wr; goto main1

Note: iinc1 = 0x84

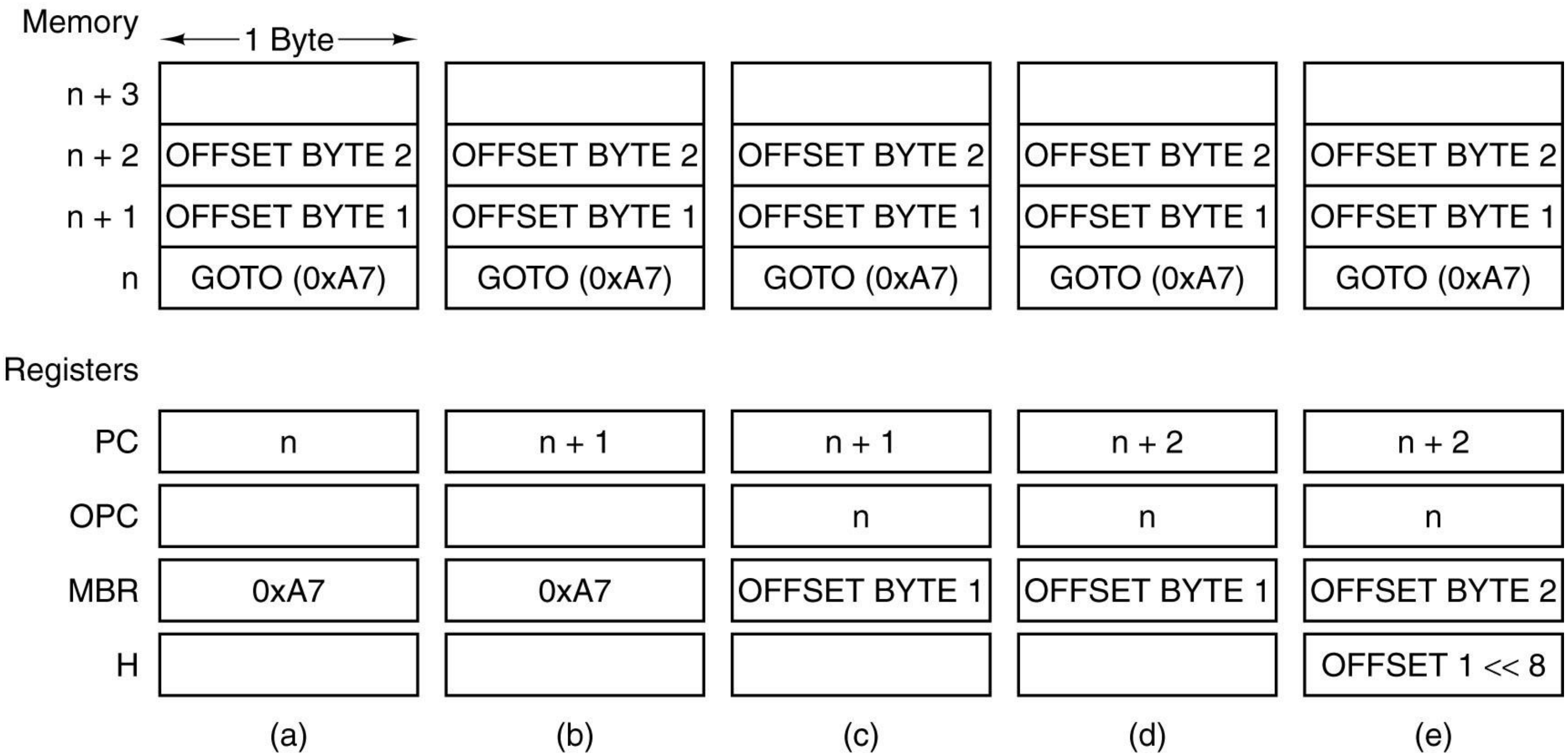
GOTO *offset*

Unconditional branch, *offset* is 16-bit signed

goto1 OPC = PC – 1	OPC = address of the GOTO
goto2 PC = PC + 1; fetch	MBR = 1st byte of offset; Fetch 2nd
goto3 H = MBR << 8	Shift signed MBR Left 1 byte
goto4 H = MBRU OR H	H = 16 bit signed offset
goto5 PC = OPC + H; fetch	Calculate absolute offset; fetch it for next MI
goto6 goto main1	

Note: goto1 = 0xA7

GOTO



The situation at the start of various microinstructions.

a) Main1. b) goto1. c) goto2. d) goto3. e) goto4.

Conditional Branching

- MPC[8] = (JAMZ AND Z) OR (JAMN AND N) OR NEXT_ADRRESS[8]

- Either:

  NEXT_ADDRESS does not change, or

  NEXT_ADDRESS[8] is set to 1

IFLT *offset*

Pop word from stack and branch to *offset* if < 0

iflt1 MAR = SP = SP – 1; rd

iflt2 OPC = TOS

iflt3 TOS = MDR

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

Note: iflt1 = 0x9B

iflt1

Read in next-to-top word from stack

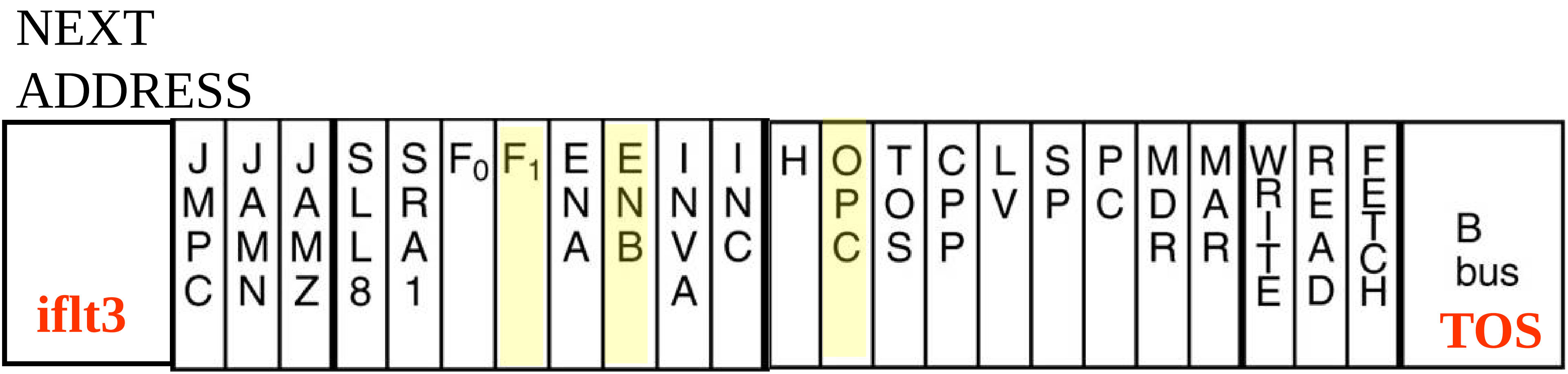
iflt1 MAR = SP = SP – 1; rd

NEXT ADDRESS	JMPC	JAMN	JAMZ	SLL8	SRA1	F_0	F_1	ENA	ENB	INVA	INC	H	OPC	TOS	CPP	LV	SP	PC	MDR	MAR	WRITE	READ	FETCH	B bus
iflt2																								**SP**

iflt2

Save top of stack in OPC, before it is lost

iflt2 OPC = TOS

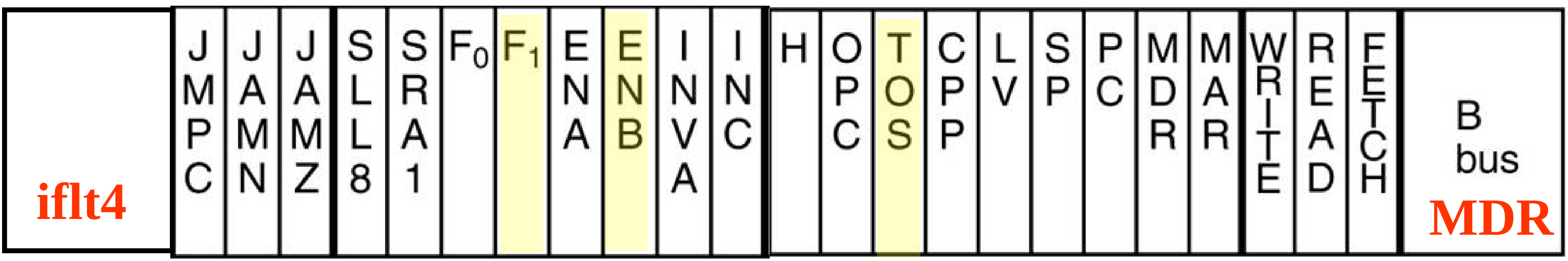


iflt3

MDR now has next-to-top word on stack; this is the new top of the stack

iflt3 TOS = MDR

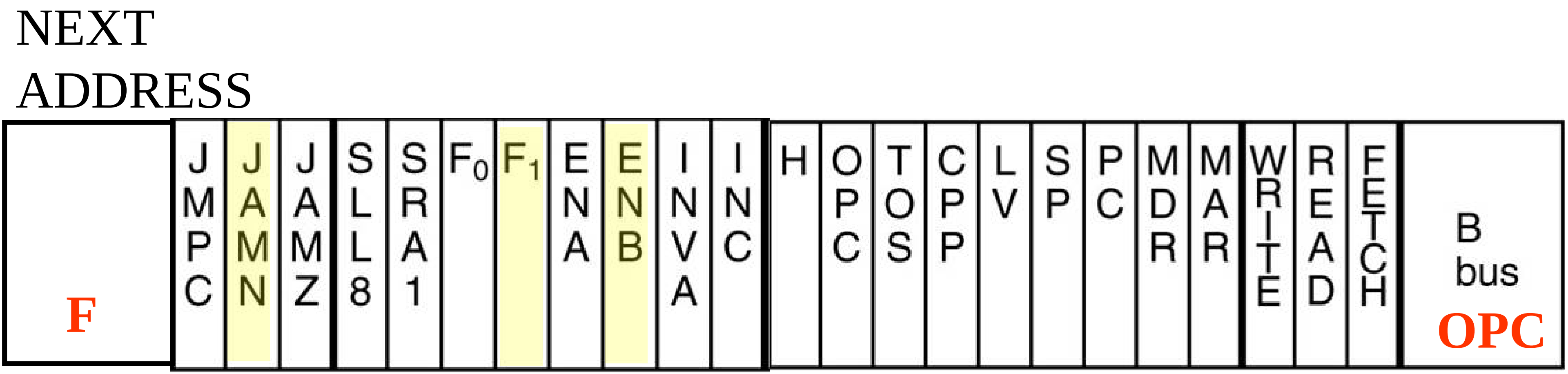
NEXT ADDRESS



iflt4

Run OPC through ALU to see if it is 0 (OPC has old top of stack); branch on N

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



Labels T and F

if (N) goto T; else goto F

- MPC[8] = (JAMZ AND Z) OR (JAMN AND N) OR NEXT_ADRRESS[8]

- If N = 1, MPC[8] = 1
- Else MPC[8] = 0
- MPC[0..7] does not change

Labels T & F

if (N) goto T; else goto F

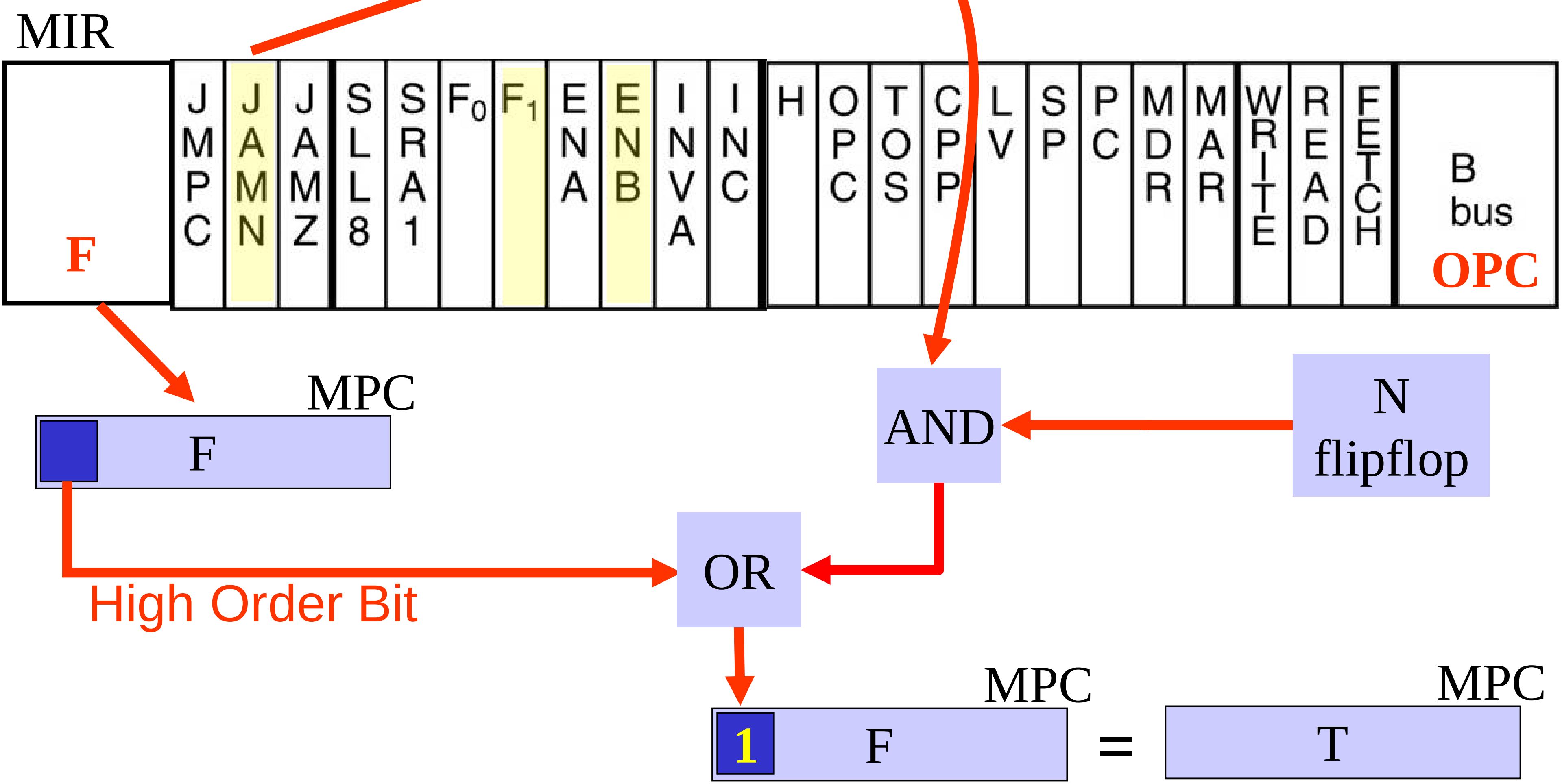
T & F must differ only in the high-order bit

For T it is 1 for F it is 0

Remaining bits are the same

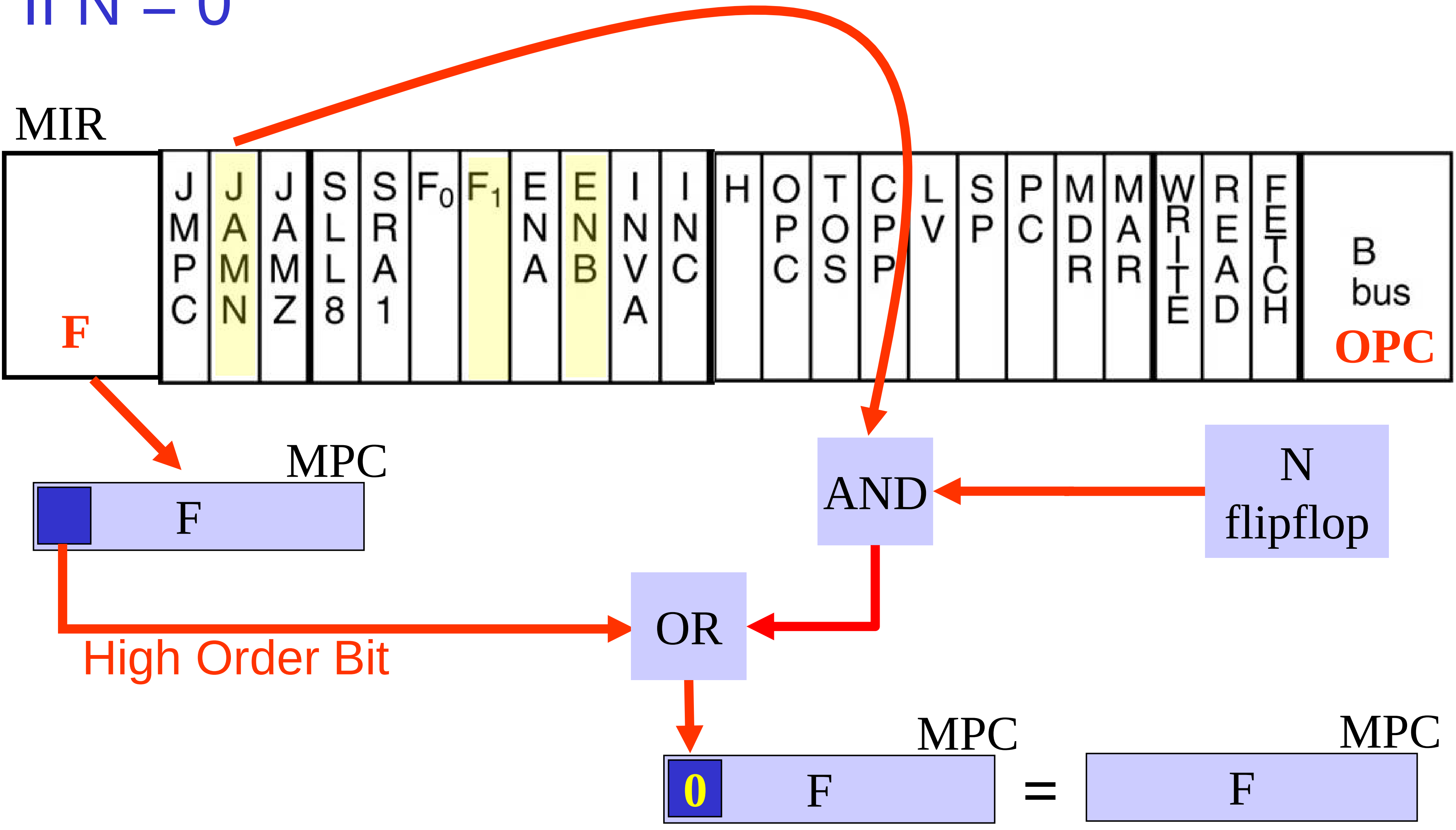
Branching to T

If N = 1



Branching to F

If N = 0



F
No need to Branch

F PC = PC + 1; goto F2

F2 PC = PC + 1; fetch; goto F3

F3 goto main1

Skip second offset byte; goto F2

Fetch new opcode

T
Branch to *offset*

T OPC = PC – 1; goto goto2

Same as goto (OPC = PC – 1 is goto1)

We could have said:

T goto goto1

But this wastes a cycle

GOTO

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

IFEQ *offset*

Pop word from stack and branch to offset if it is = 0

ieq1 MAR = SP = SP – 1; rd

ieq2 OPC = TOS

ieq3 TOS = MDR

ieq4 Z = OPC; if (Z) goto T; else goto F

(same as IFLT, but Z is used instead of N)

Note: ifeq1 = 0x99

IF_ICMPEQ *offset*

Pop 2 words from stack and branch to *offset* if equal

if_icmpeq1 MAR = SP = SP – 1; rd

if_icmpeq2 MAR = SP = SP – 1

if_icmpeq3 H = MDR; rd

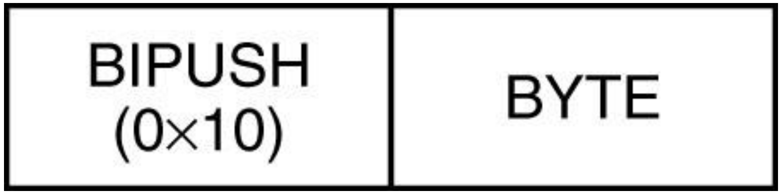
if_icmpeq4 OPC = TOS

if_icmpeq5 TOS = MDR

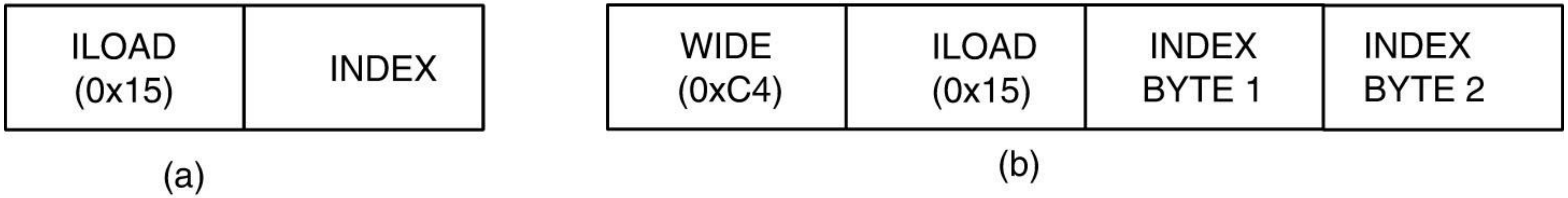
if_icmpeq6 Z = OPC – H; if (Z) goto T; else goto F

Note: if_icmpeq1 = 0x9F

WIDE



The BIPUSH instruction format.

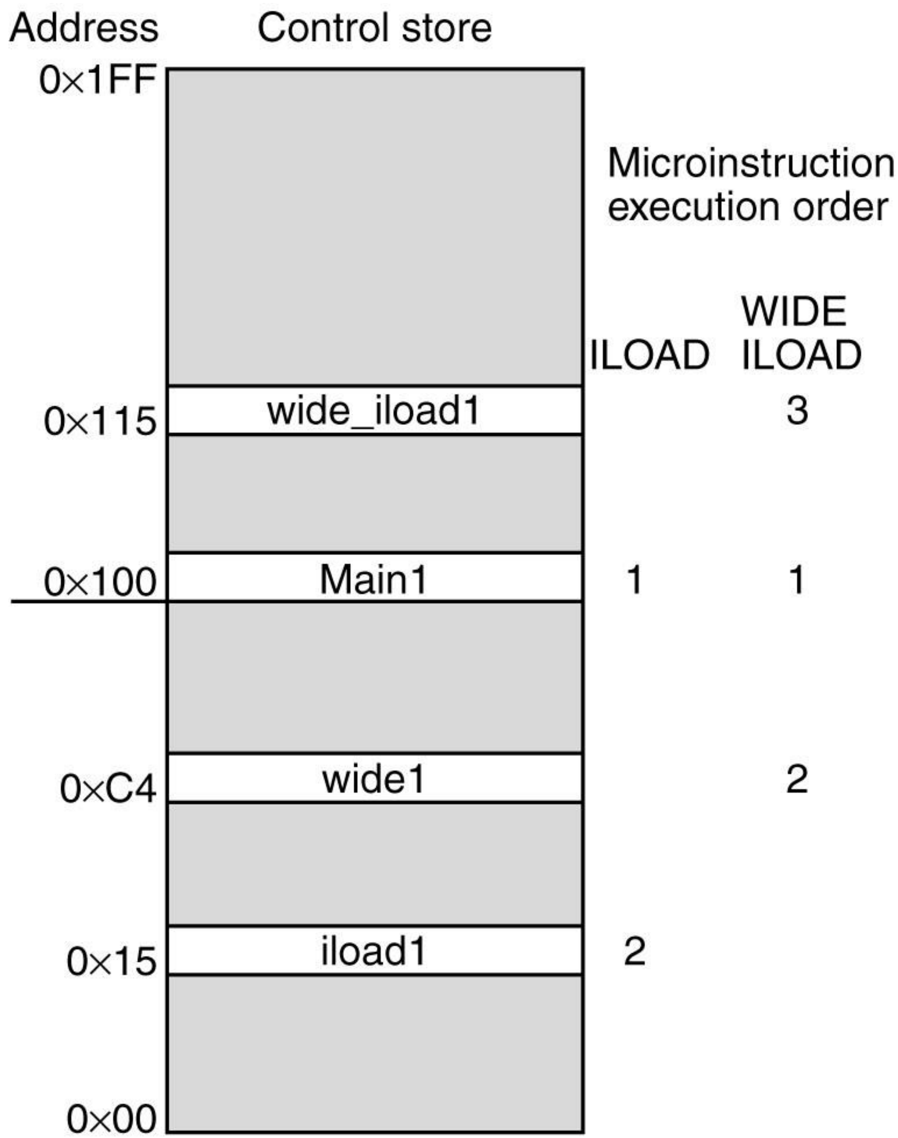


a) ILOAD with a 1-byte index.

b) WIDE ILOAD with a 2-byte index.

WIDE

The initial microinstruction sequence for ILOAD and WIDE ILOAD. The addresses are examples.



WIDE

wide1	PC = PC + 1; fetch;	Fetch operand byte or next opcode
wide2	goto (MBR OR 0x100)	Multiway branch with high bit set
wide_iload1	PC = PC + 1; fetch	MBR contains 1st index byte; fetch 2nd
wide_iload2	H = MBRU << 8	H = 1st index byte shifted left 8 bits
wide_iload3	H = MBRU OR H	H = 16-bit index of local variable
wide_iload4	MAR = LV + H; rd; goto iload3	MAR = address of local variable to push
wide_istore1	PC = PC + 1; fetch	MBR contains 1st index byte; fetch 2nd
wide_istore2	H = MBRU << 8	H = 1st index byte shifted left 8 bits
wide_istore3	H = MBRU OR H	H = 16-bit index of local variable
wide_istore4	MAR = LV + H; goto istore3	MAR = address of local variable to store into

wide1	PC = PC + 1; fetch;
wide2	goto (MBR OR 0x100)

WIDE LOAD

Fetch first byte of operand

wide1	PC = PC + 1; fetch;
wide2	goto (MBR OR 0x100)

Goto wide_iload1

wide_iload1	PC = PC + 1; fetch
wide_iload2	H = MBRU << 8
wide_iload3	H = MBRU OR H
wide_iload4	MAR = LV + H; rd; goto iload3

Fetch next opr byte

iload1	H = LV
iload2	MAR = MBRU + H; rd
iload3	MAR = SP = SP + 1
iload4	PC = PC + 1; fetch; wr
iload5	TOS = MDR; goto Main1

WIDE STORE

wide1	PC = PC + 1; fetch;
wide2	goto (MBR OR 0x100)

wide_istore1	PC = PC + 1; fetch
wide_istore2	H = MBRU << 8
wide_istore3	H = MBRU OR H
wide_istore4	MAR = LV + H; goto istore3

istore1	H = LV
istore2	MAR = MBRU + H
istore3	MDR = TOS; wr
istore4	SP = MAR = SP – 1; rd
istore5	PC = PC + 1; fetch
istore6	TOS = MDR; goto Main1

LDC_W

ldc_w1	PC = PC + 1; fetch
ldc_w2	H = MBRU << 8
ldc_w3	H = MBRU OR H
ldc_w4	MAR = H + CPP; rd; goto iload3

Calling INVOKEVIRTUAL

LDC_W OBJREF

INVOKEVIRTUAL *noParamMethod*

.constant

OBJREF 0x40

.end-constant

Calling INVOKEVIRTUAL

LDC_W OBJREF

ILOAD *param*

INVOKEVIRTUAL *oneParamMethod*

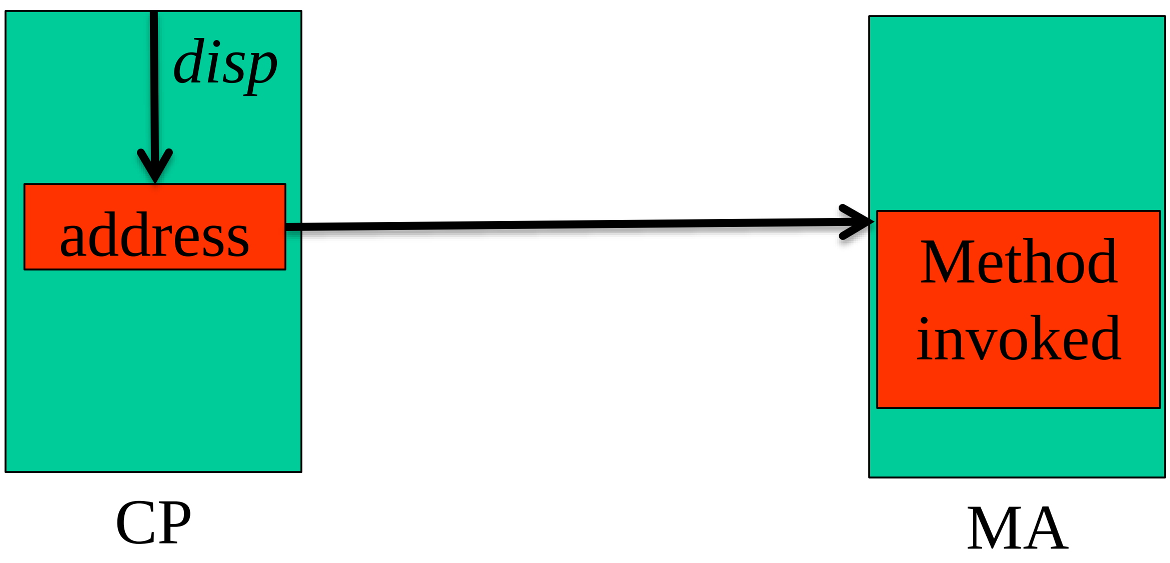
.constant

OBJREF 0x40

.end-constant

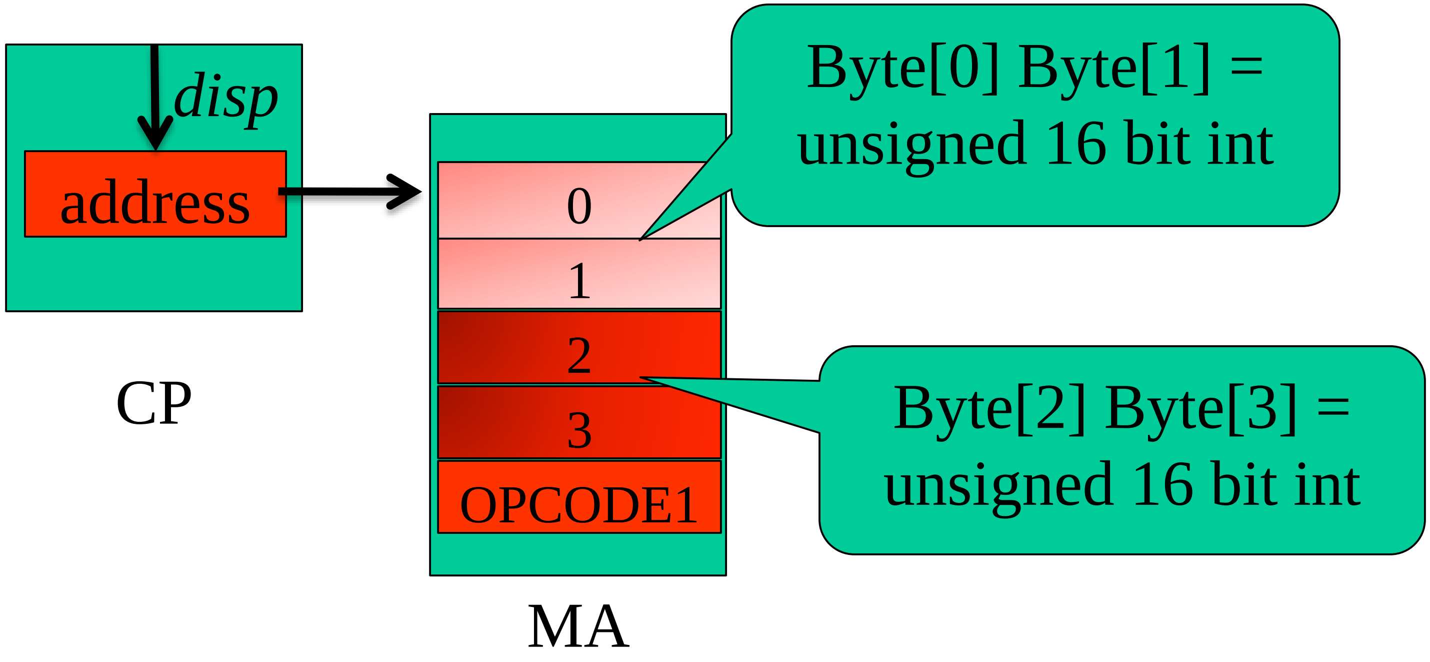
IVOKEVIRTUAL *disp*

- Invokes another method
- *disp* (16 bit) = position in constant pool that contains the address in method area where method starts

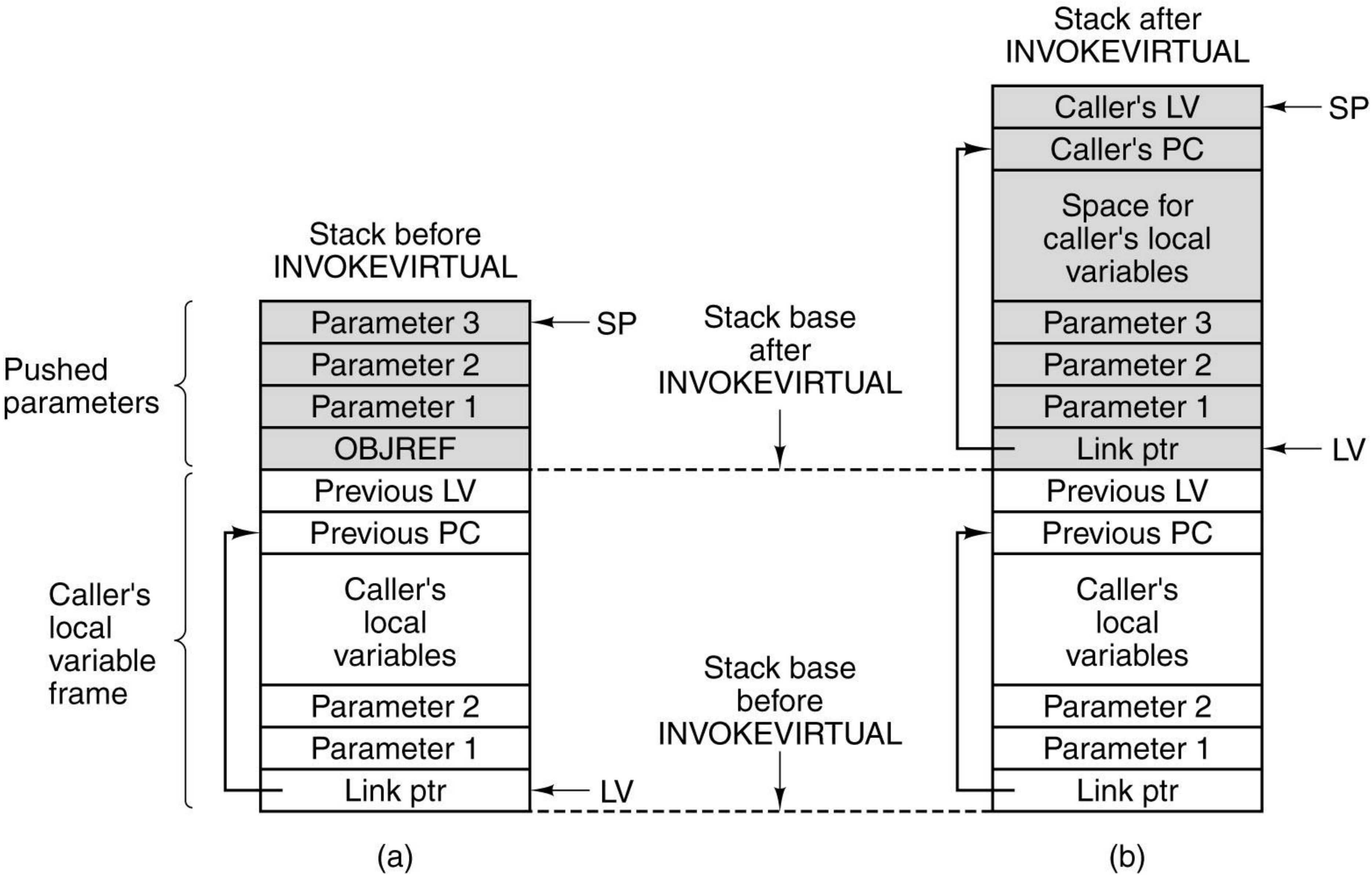


IVOKEVIRTUAL *disp*

- First 4 bytes of a method
  - First 2: number of parameters, including OBJREF (param 0)
  - Second 2: size of local variable area



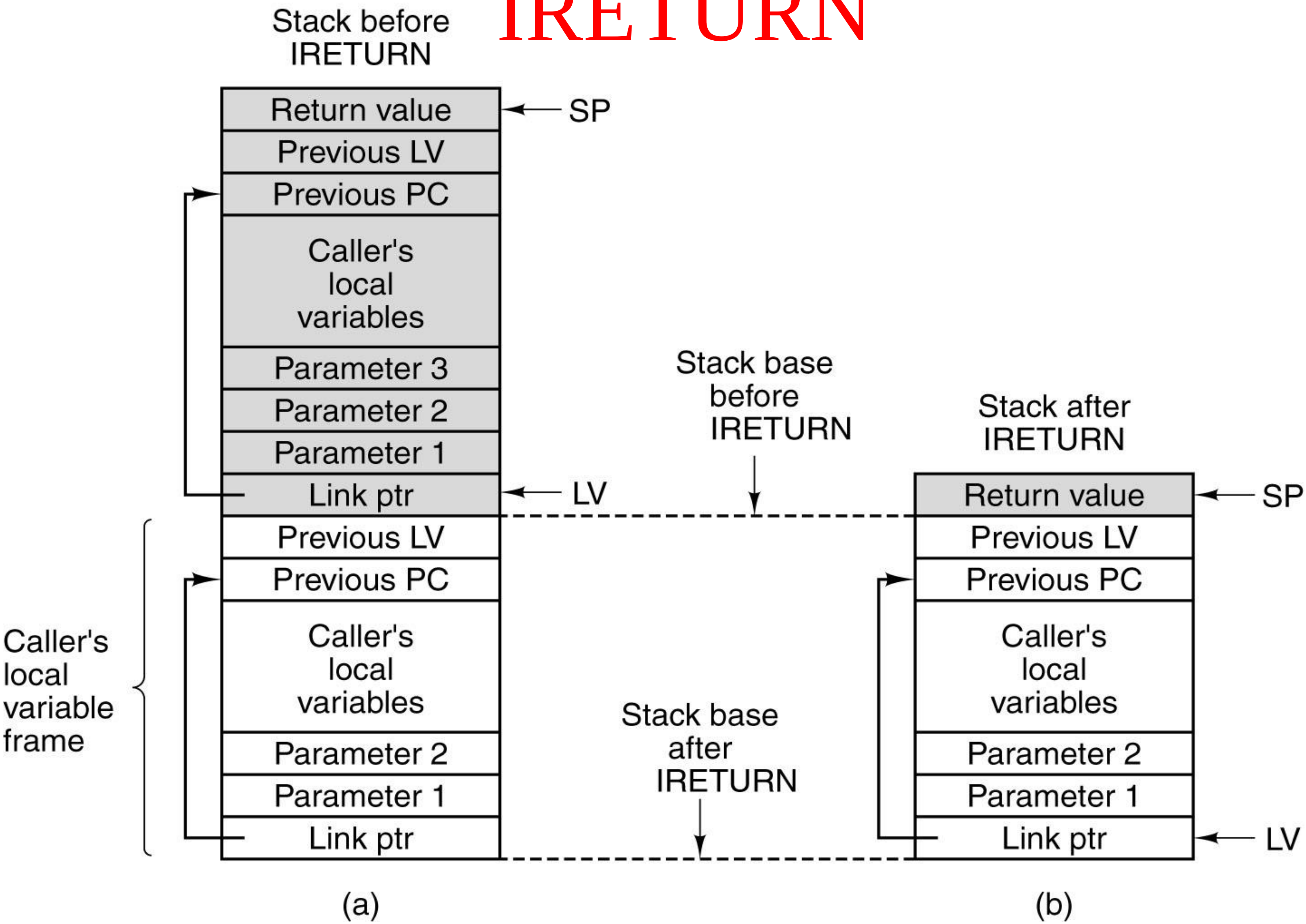
INVOKEVIRTUAL



INVOKEVIRTUAL *disp*

invokevirtual1	PC = PC + 1; fetch	MBR = index byte 1; inc. PC, get 2nd byte
invokevirtual2	H = MBRU << 8	Shift and save first byte in H
invokevirtual3	H = MBRU OR H	H = offset of method pointer from CPP
invokevirtual4	MAR = CPP + H; rd	Get pointer to method from CPP area
invokevirtual5	OPC = PC + 1	Save Return PC in OPC temporarily
invokevirtual6	PC = MDR; fetch	PC points to new method; get param count
invokevirtual7	PC = PC + 1; fetch	Fetch 2nd byte of parameter count
invokevirtual8	H = MBRU << 8	Shift and save first byte in H
invokevirtual9	H = MBRU OR H	H = number of parameters
invokevirtual10	PC = PC + 1; fetch	Fetch first byte of # locals
invokevirtual11	TOS = SP – H	TOS = address of OBJREF – 1
invokevirtual12	TOS = MAR = TOS + 1	TOS = address of OBJREF (new LV)
invokevirtual13	PC = PC + 1; fetch	Fetch second byte of # locals
invokevirtual14	H = MBRU << 8	Shift and save first byte in H
invokevirtual15	H = MBRU OR H	H = # locals
invokevirtual16	MDR = SP + H + 1; wr	Overwrite OBJREF with link pointer
invokevirtual17	MAR = SP = MDR;	Set SP, MAR to location to hold old PC
invokevirtual18	MDR = OPC; wr	Save old PC above the local variables
invokevirtual19	MAR = SP = SP + 1	SP points to location to hold old LV
invokevirtual20	MDR = LV; wr	Save old LV above saved PC
invokevirtual21	PC = PC + 1; fetch	Fetch first opcode of new method.
invokevirtual22	LV = TOS; goto Main1	Set LV to point to LV Frame

IRETURN



Stack before IRETURN
Return value
Previous LV
Previous PC
Caller's local variables
Parameter 3
Parameter 2
Parameter 1
Link ptr
Previous LV
Previous PC
Caller's local variables
Parameter 2
Parameter 1
Link ptr
SP
LV
Caller's local variable frame
Stack base before IRETURN
Stack base after IRETURN
Stack after IRETURN
Return value
Previous LV
Previous PC
Caller's local variables
Parameter 2
Parameter 1
Link ptr
SP
LV
(a)
(b)

IRETURN

ireturn1	MAR = SP = LV; rd	Reset SP, MAR to get link pointer
ireturn2		Wait for read
ireturn3	LV = MAR = MDR; rd	Set LV to link ptr; get old PC
ireturn4	MAR = LV + 1	Set MAR to read old LV
ireturn5	PC = MDR; rd; fetch	Restore PC; fetch next opcode
ireturn6	MAR = SP	Set MAR to write TOS
ireturn7	LV = MDR	Restore LV
ireturn8	MDR = TOS; wr; goto Main1	Save return value on original top of stack

MMV Example

MODIFYING THE MICROPROGRAM

MMV File Types

- JAS: source files containing IJVM ISA
  - For example *ex.jas* on BB
- IJVM: MMV assembles JAS files to IJVM files
- MAL: the microprogram is stored in a MAL file (mic1ijvm.mal)
- MIC1: MMV assembles MAL files to MIC1 files
- Once JAS or MAL files are assembled, the corresponding IJVM or MIC1 files need only be loaded

Adding MDUP

- Add the instruction MDUP *byte* which works like DUP except that it is repeated *byte* times
- For instance, MDUP 5 duplicates the top of the stack 5 times

1. Write your MAL code

```
mdup1   PC = PC + 1; fetch; // fetch next opcode
mdup2   OPC = MBR; // save operand
mdup3   N = OPC; if (N) goto myT1; else goto myF1 // if neg quit
mdup4   Z = OPC; if (Z) goto myT2; else goto myF2 // if zero quit
mdup5   SP = MAR = SP + 1 // one more word on the stack
mdup6   MDR = TOS; wr // write TOS to stack
mdup7   OPC = OPC -1; goto mdup4 // dec OPC and repeat

myT1    goto Main1 //quit
myT2    goto Main1 //quit
myF1    goto mdup4 //continue
myF2    goto mdup5 //continue
```

1. Write your MAL code

- Do not skip line numbers
  - Like mdup1, mdup3, without mdup2
- No need for implicit goto

```
mdup1   PC = PC + 1; fetch; goto mdup2   No need
mdup2   OPC = MBR;
```

- Keep in mind that “True” and “False” labels must be 100000000 apart
  - Microassembler takes care of this
  - Use new labels

2. Optimize your MAL code

```
mdup1   PC = PC + 1; fetch; // fetch next opcode
mdup2   OPC = MBR; // save operand
mdup3   N = OPC; if (N) goto myT1; else goto myF1 // if neg quit
mdup4   Z = OPC; if (Z) goto myT2; else goto myF2 // if zero quit
mdup5   SP = MAR = SP + 1 // one more word on the stack
mdup6      MDR = TOS; wr // write TOS to stack
mdup7   OPC = OPC -1; goto mdup4 // dec OPC and repeat

myT1    goto Main1 replace with PC = PC + 1; fetch; goto (MBR)
myT2    goto Main1 replace with PC = PC + 1; fetch; goto (MBR)
myF1    goto mdup4 replace with Z = OPC; if (Z) goto myT2; else
     goto myF2
myF2    goto mdup5 replace with SP = MAR = SP + 1 ; goto mdup6
```

3. Add your code to *mic1ijvm.mal*

- Code can be added after any instruction that ends with an explicit goto
- MDUP now needs a numeric value for the opcode
- Add it at the beginning of the file as

.label mdup1 0x01

- Use any hex value that is not in use

3. Add your code to *mic1ijvm.mal*

```
File  Edit  Search  View  Options  Help
C:\Mic1MMV\examples\MAL\mic1ijvm.mal
// labeled statements are "anchored" at the specified control store address
.label  nop1            0x00
.label  mdup1           0x01
.label  bipush1         0x10
.label  ldc_w1          0x13
.label  iload1          0x15
.label  wide_iload1     0x115
.label  mdup1           0x17
.label  istore1         0x36
.label  wide_istore1    0x136
.label  pop1            0x57
.label  dup1            0x59
.label  swap1           0x5F
.label  iadd1           0x60
.label  isub1           0x64
.label  iand1           0x7E
.label  iinc1           0x84
.label  ifeq1           0x99
.label  iflt1           0x9B
.label  if_icmpeq1      0x9F
.label  goto1           0xA7
.label  ireturn1        0xAC
F1=Help                                   Line:40    Col:46
```

4. Modify *ijvm.conf*

- Add:

0x01 MDUP byte // duplicates the top of the stack byte times

Anywhere in the file

5. Assemble the Microcode

- Run MMV from a directory that contains ijvm.conf
- From MMV, load/Assemble MAL file
  - Choose mic1ijvm.mal
- [if there are errors MMV will tell you]
- Load (effectively loading mic1ijvm.mic1)

5. Assemble the Microcode (NOTES)

You may get a J**ava.lang.NullPointerException** or **Java.lang.Exception** when assembling some MAL code in MMV. Two things can help you recover from these, which I have learned the hard way:

1. Make sure there is an empty line at the end of you MAL file
2. If your code contains multi-way branching (if-else), add your code gradually to the MAL file as follows:

(a) Add all microinstrcutions (MI) that precede the next MI that has if-else. (So this chunk you have just added does not have any if-else MIs). Compile this part alone and correct any errors before proceeding.

(b) add the following if-else MI, compile, and correct errors.

GOTO (a) until all MIs have been added to the MAL file

6. Write a JAS tester

- Such as:

```
.main
start:
    BIPUSH 0x1
    MDUP 2
    BIPUSH 0x2
    MDUP 5
.end-main
```

7. Load/Assemble JAS file

- From MMV, load/assemble JAS file
  - Choose the tester file
- Run the program