The Microarchitecture Level (Chapitre 4)

Francesco Bronzino
ArchiSys

* Slides based on Tanenbaum, "Structured Computer Organization" 5e

Updates

- Course on May 12th, TD on May 11th
- Exam on June 2nd
- Tomorrow 1h TD + finishing TP 2

The microarchitecture level

The data path

- The job of the microarchitecture level is to implement the Instruction Set Architecture level above it
- Its design depends on the ISA being implemented, as well as the cost and performance goals of the computer.
  - RISC designs have simple instructions that can usually be executed in a single clock cycle.
  - The Core i7 instruction set may require many cycles to execute a single instruction

The data path

- No general principles; every ISA is a special case.
- Hence, we use a specific example instead.
- Subset of the Java Virtual Machine called IJVM.

The data path

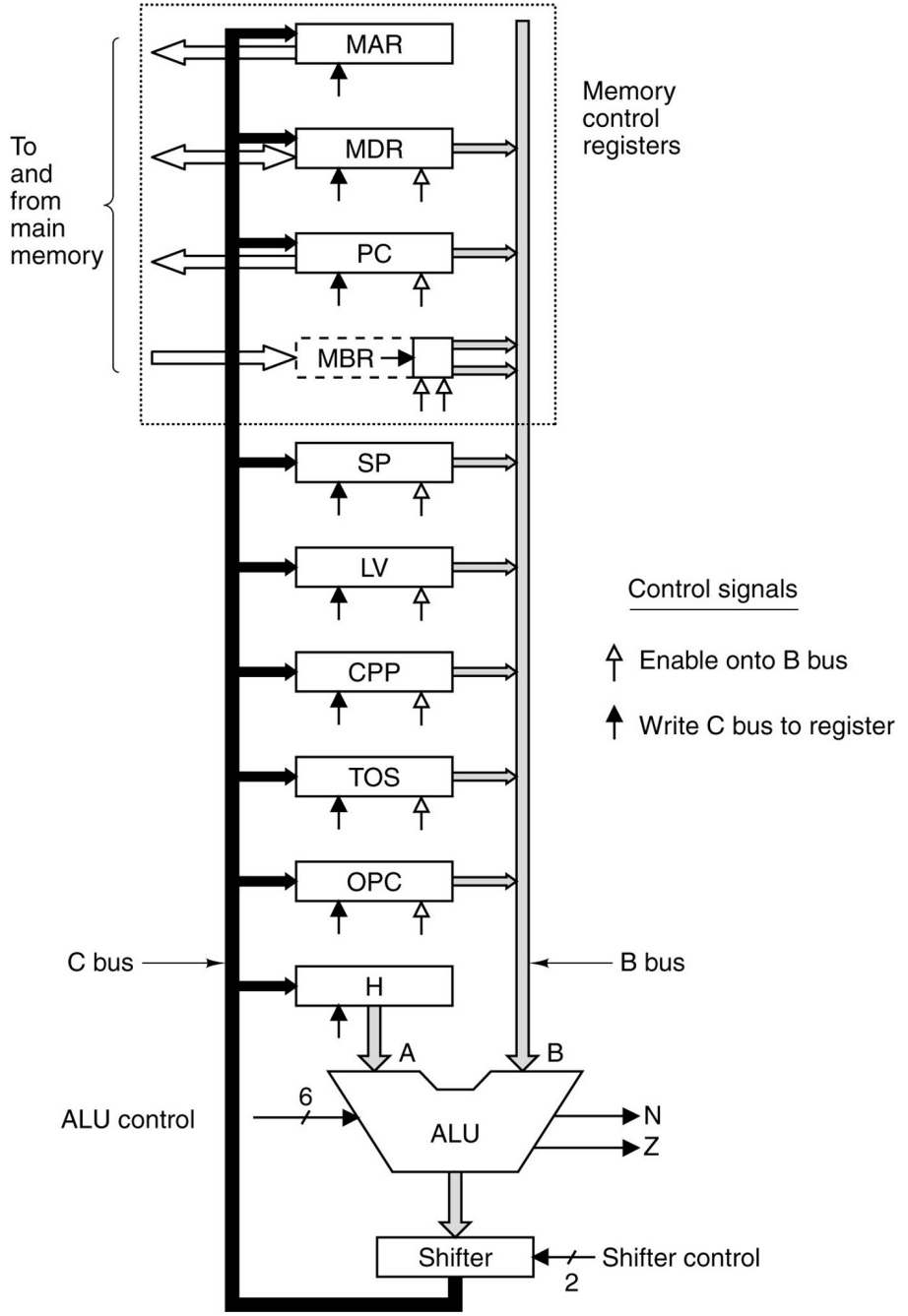
- The microarchitecture will contain a microprogram (in ROM),
- Fetch, decode, and execute instructions.
- We cannot use the Oracle JVM interpreter because we need a tiny microprogram that drives the individual gates in the actual hardware efficiently.
- The Oracle JVM interpreter was written in C for portability and cannot control the hardware at all.

The data path

Since the actual hardware used consists only of the basic components described in Chap. 3, in theory, after fully understanding this chapter, the reader should be able to go out and buy a large bag full of transistors and build this subset of the JVM machine. Students who successfully accomplish this task will be given extra credit (and a complete psychiatric examination).

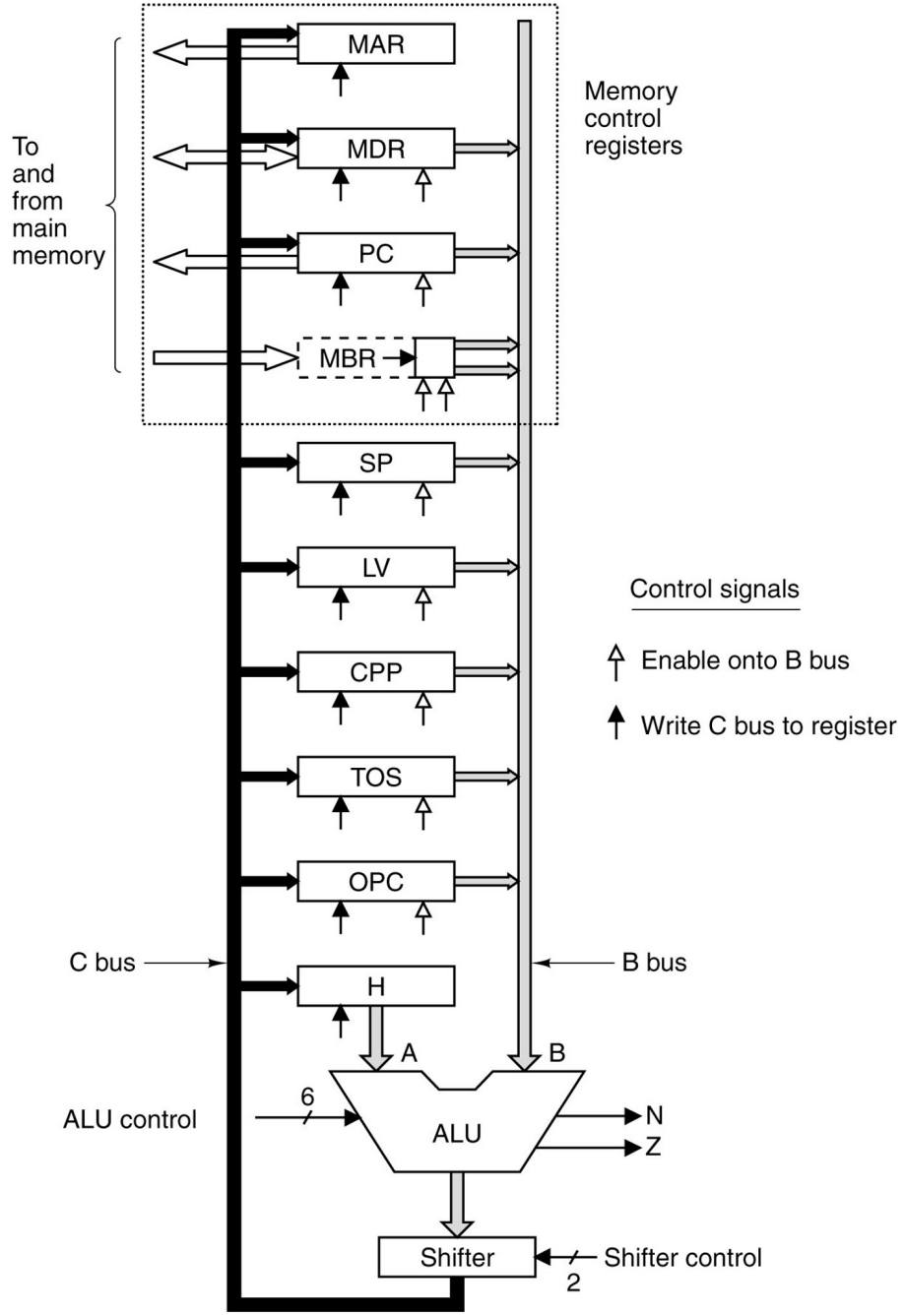
The data path

The data path of the example microarchitecture used in this chapter



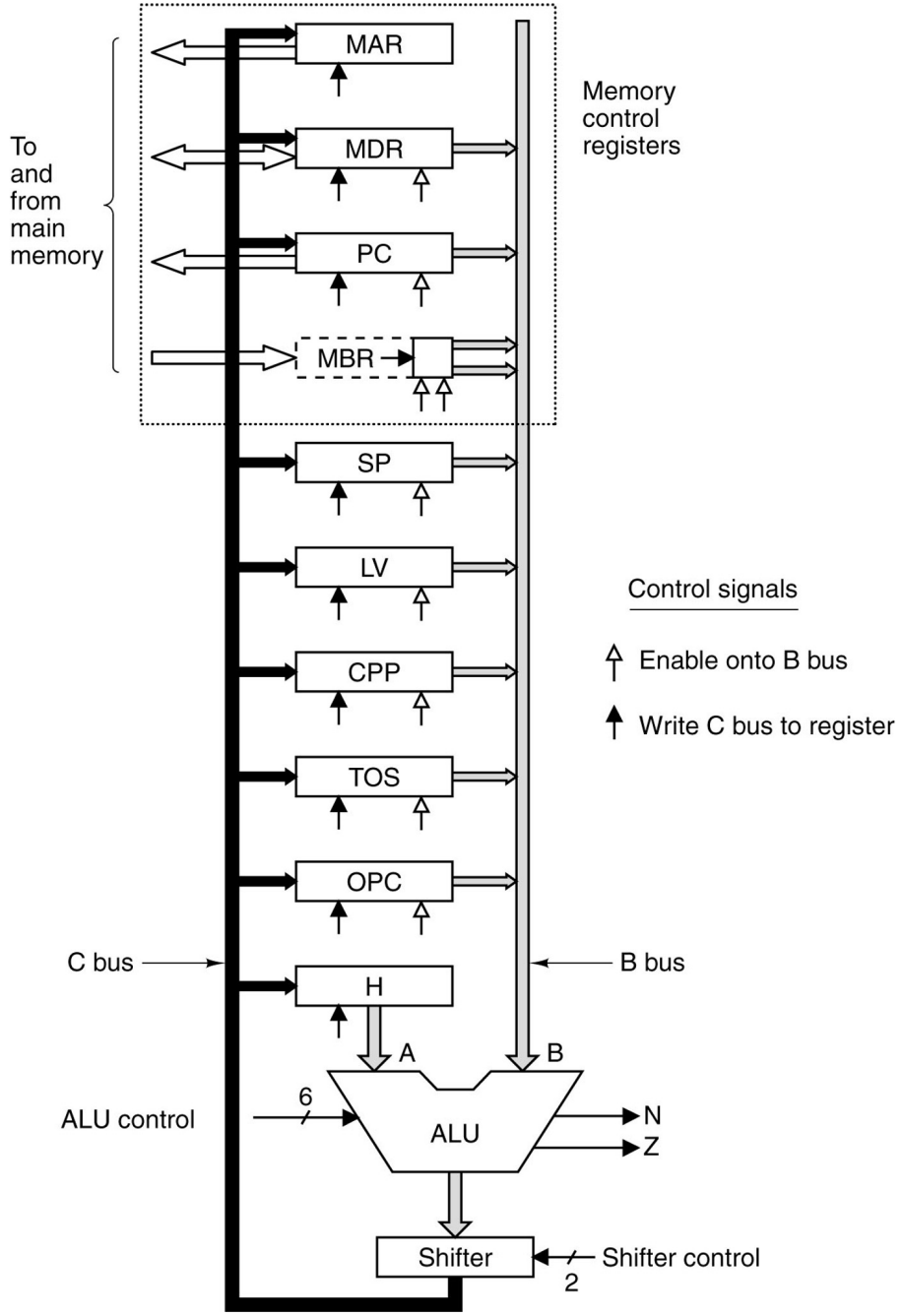
The data path

- PC and MBR are byte registers
- N and Z are bit registers
- MAR – Memory Address Register
- MDR – Memory Data Register
- PC – Program Counter
- MBR – Memory Byte Register



The data path

- SP – Stack pointer
- LV – Local Variables
- CPP – Constant Pool Pointer
- TOS – Top of Stack
- OPC – utility register
- H – Holding register

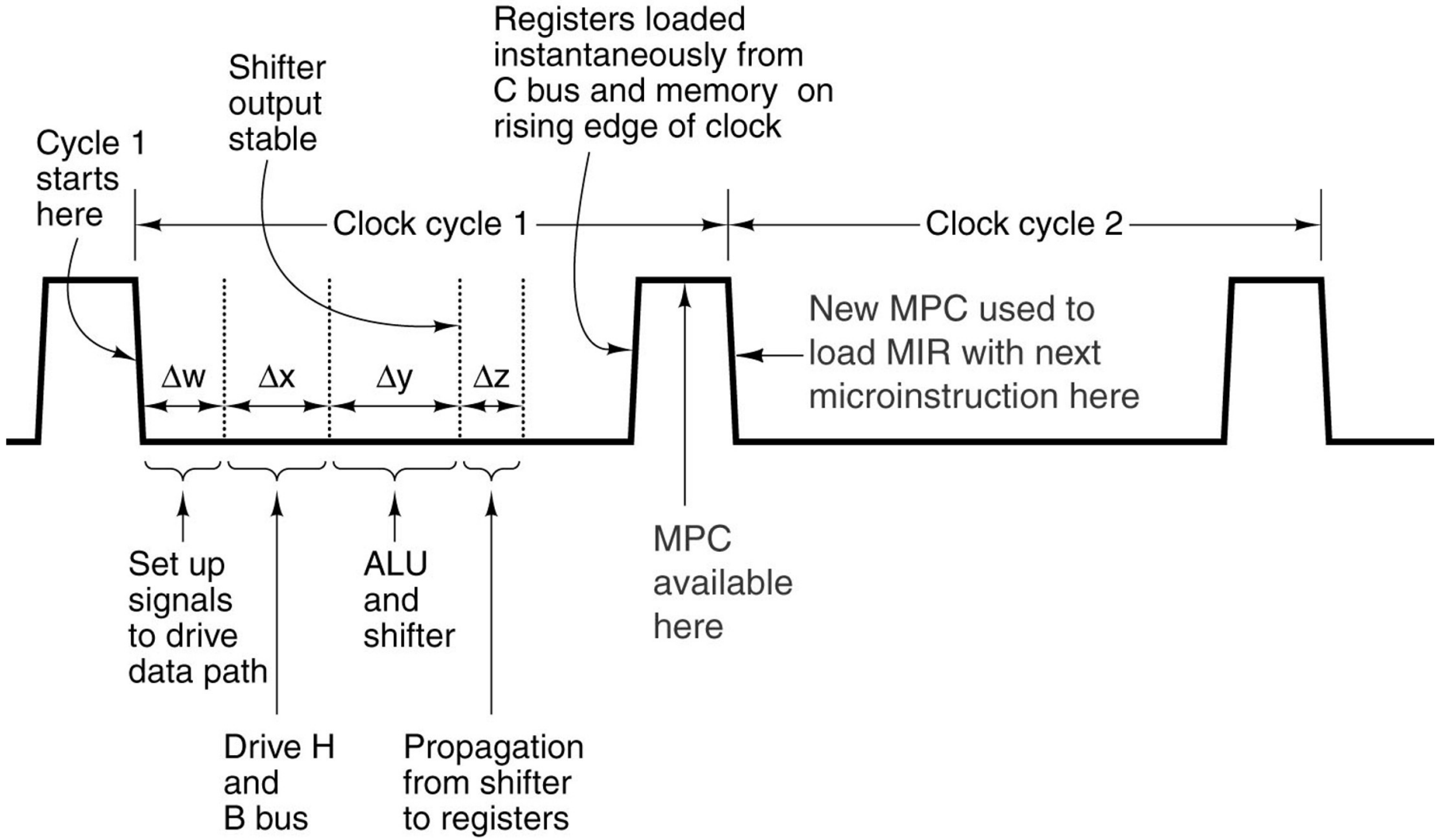


The data path

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

Useful combinations of ALU signals and the function performed

The data path timing

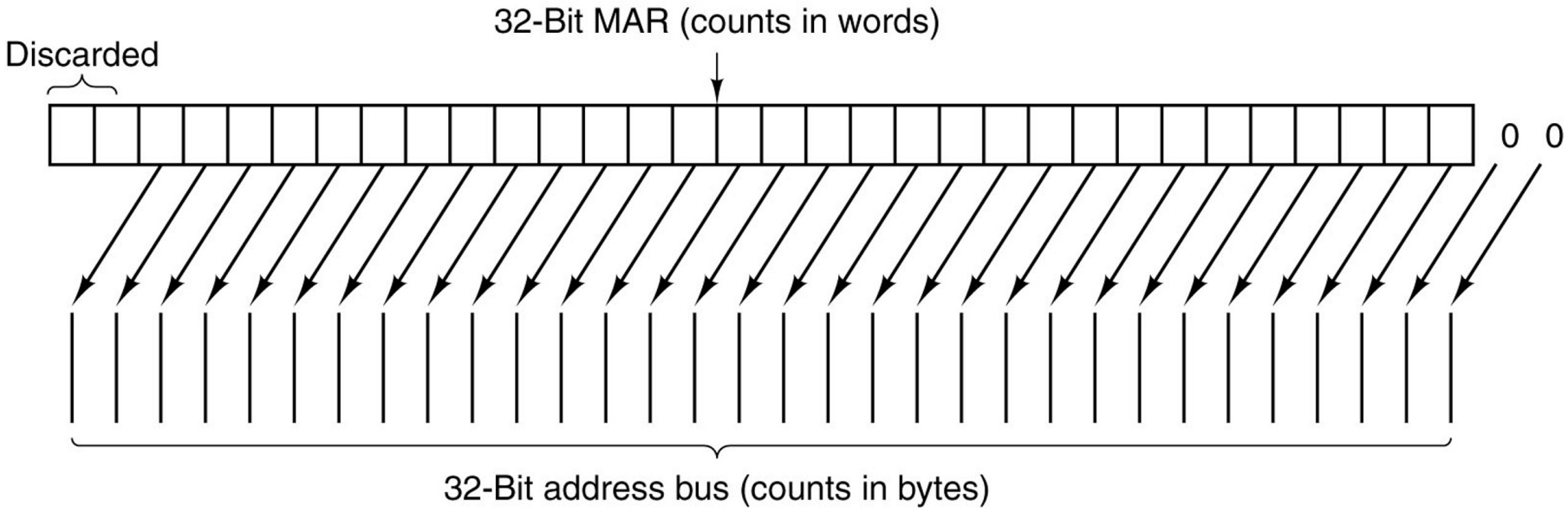


Timing diagram of one data path cycle

Clocks

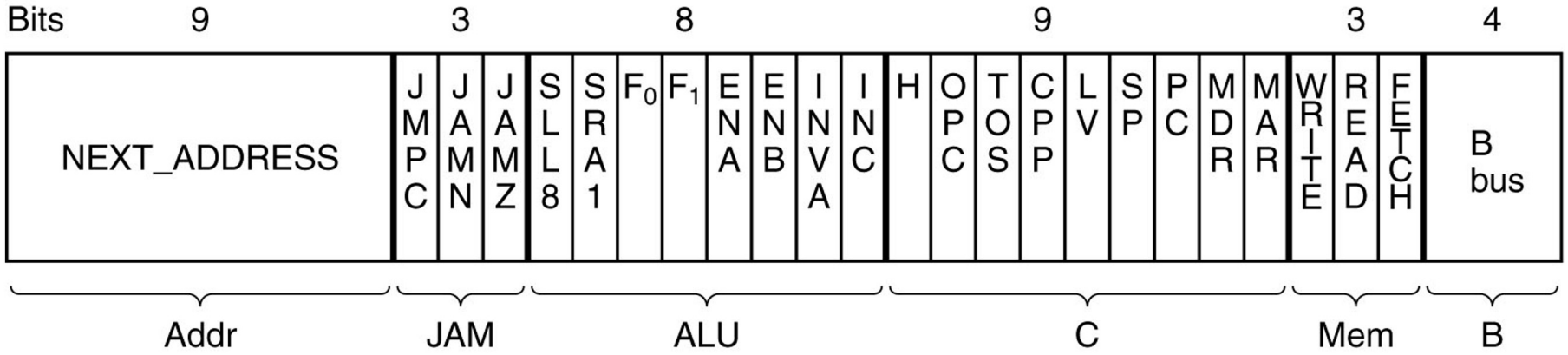
- Only actual clock is C1
- The use of delays effectively define C2, C3, etc.

Memory operation



Mapping of the bits in MAR to the address bus

Microinstructions



B bus registers

0 = MDR	5 = LV
1 = PC	6 = CPP
2 = MBR	7 = TOS
3 = MBRU	8 = OPC
4 = SP	9-15 none

The microinstruction format for the Mic-1

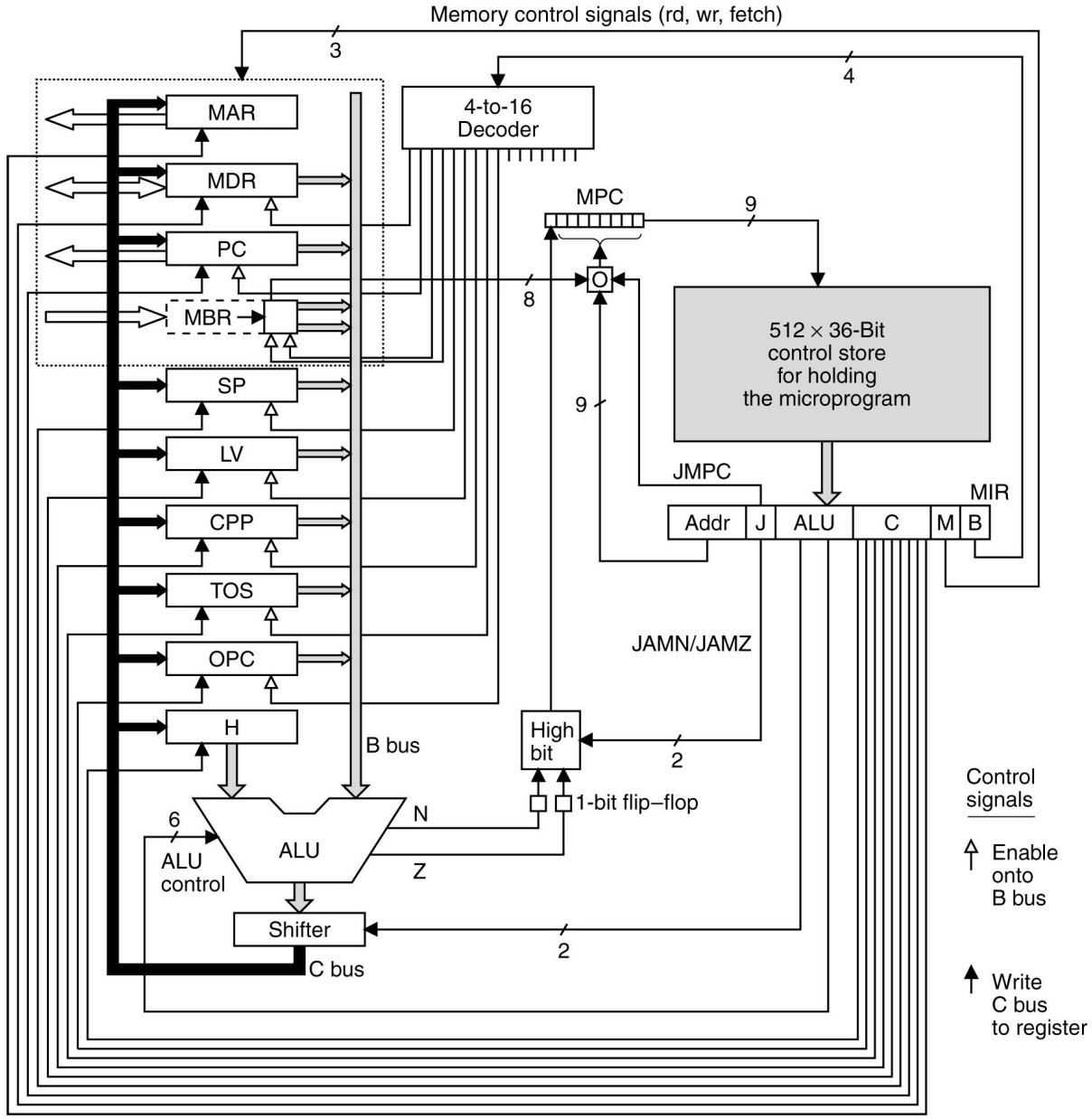
Microinstruction notes

- Next instruction refers to address in control store
- JMPC – set is unconditional jump (MBR or NEXT -> MPC)
- JAMN – set is jump if N is set (or 0x100)
- JAMZ – set if jump if Z is set (or 0x100)
- SLL8 – shift left 8
- SRA1 – shift right 1
- F0, F1, ENA, ENB, INVA, INC – ALU control lines
- Write – write memory (32 bit)
- Read – read memory (32 bit)
- Fetch – get byte of memory

Microinstruction Control: The Mic-1

The complete block diagram of our example microarchitecture, the Mic-1

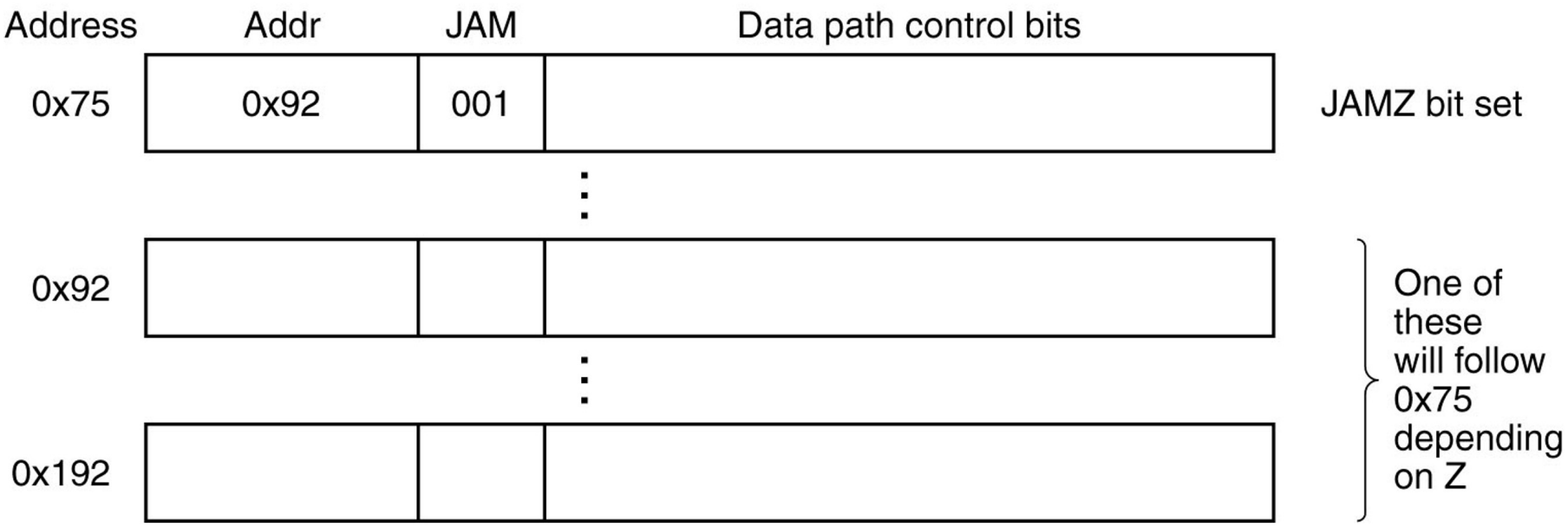
- MPC – Microcode program counter
- MIR – MicroInstruction Register



MIC-1 notes

- The microcode contains the instructions that translate the instruction set into operations that control the data path.
- One view is that the microcode is an interperter that runs the Instruction Set code.
- The other view is that the microcode is a set of subroutines that are called by the instructions at the instruction set Architecture level.

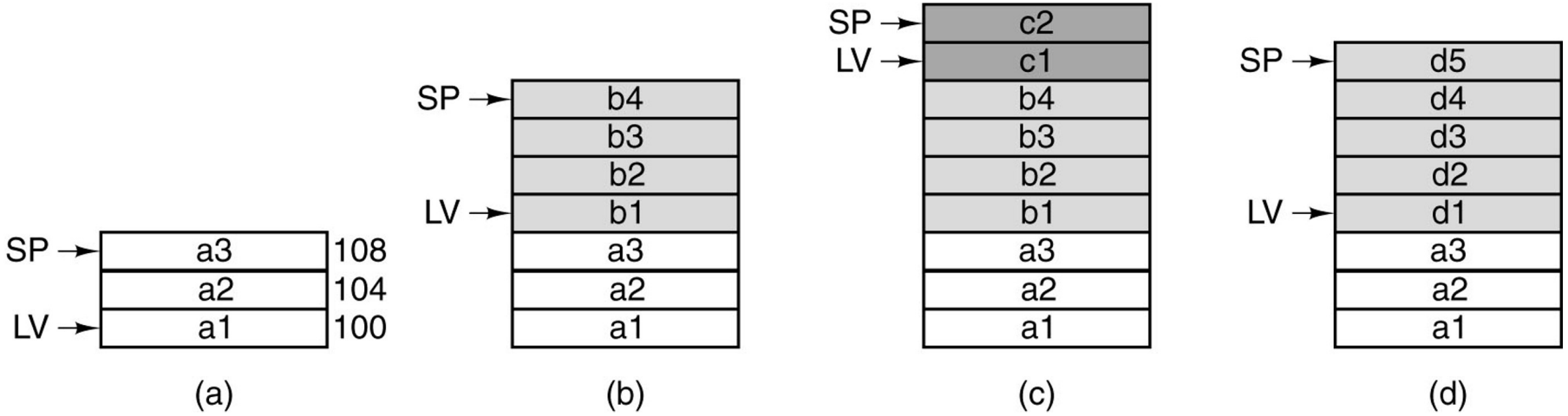
Microinstruction Control: The Mic-1



A microinstruction with JAMZ set to 1 has two potential successors

$$F = (JAMZ\ AND\ Z)\ OR\ (JAMN\ AND\ N)\ OR\ NEXT\ ADDRESS[8]$$

Stacks

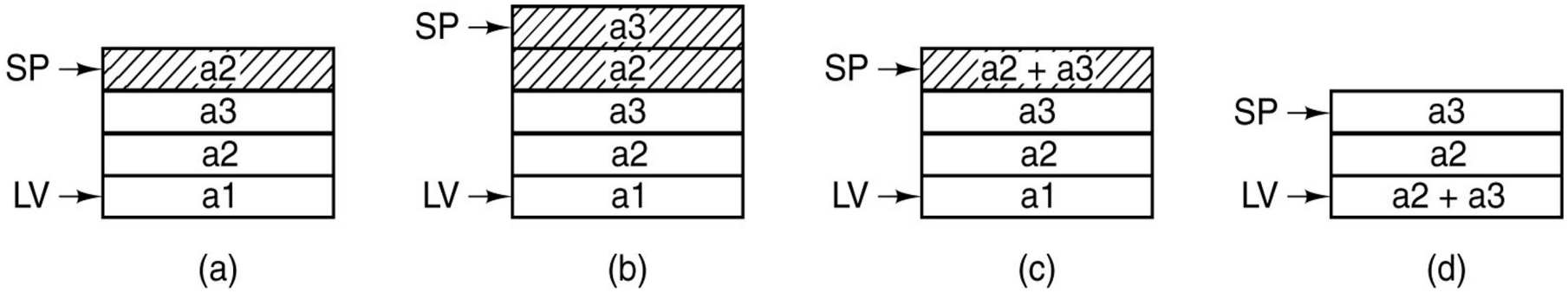


Use of a stack for storing local variables:

(a) While A is active. (b) After A calls B.

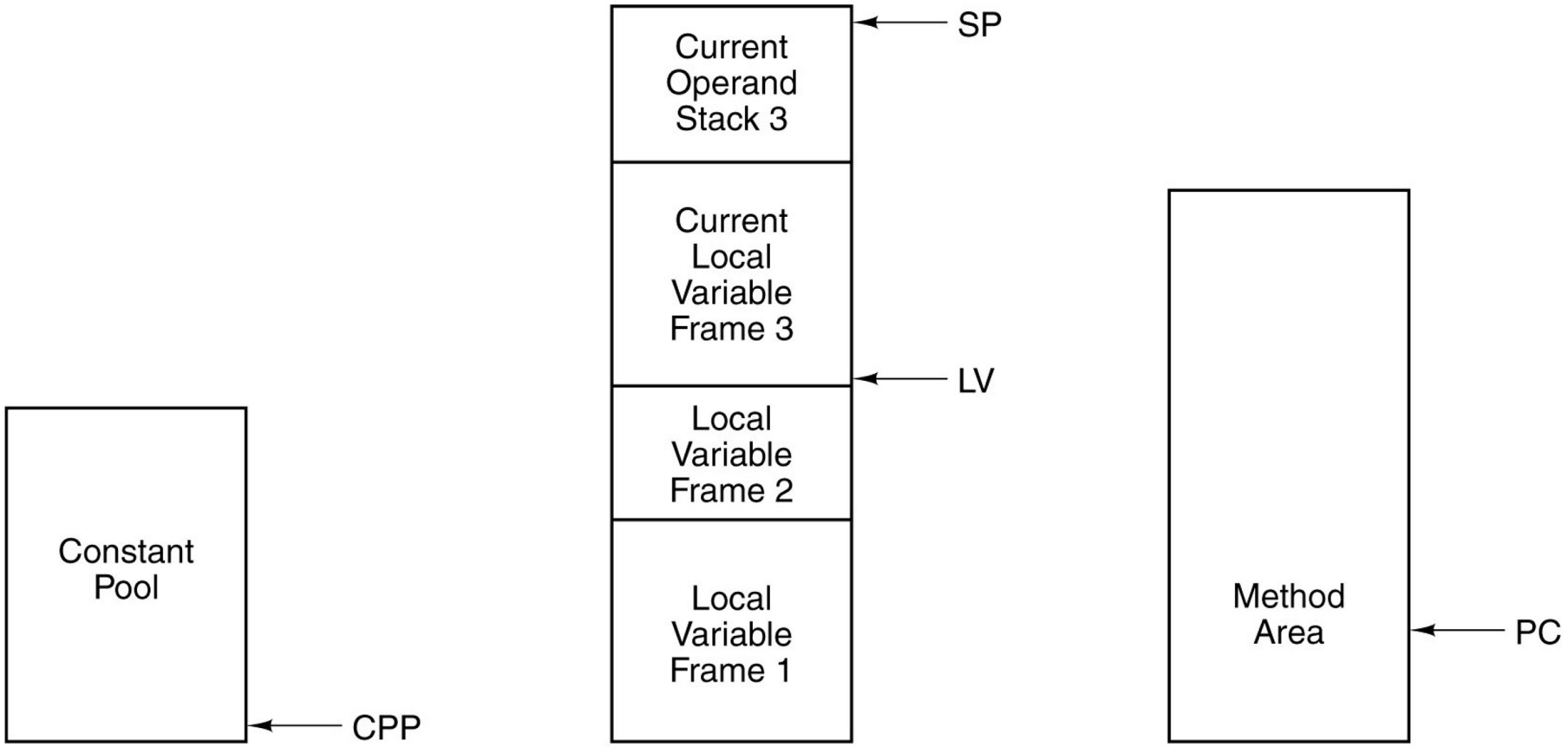
(c) After B calls C. (d) After C and B return and A calls D.

Stacks



Use of an operand stack for doing an arithmetic computation

The IJVM Memory Model



The various parts of the IJVM memory

Memory notes

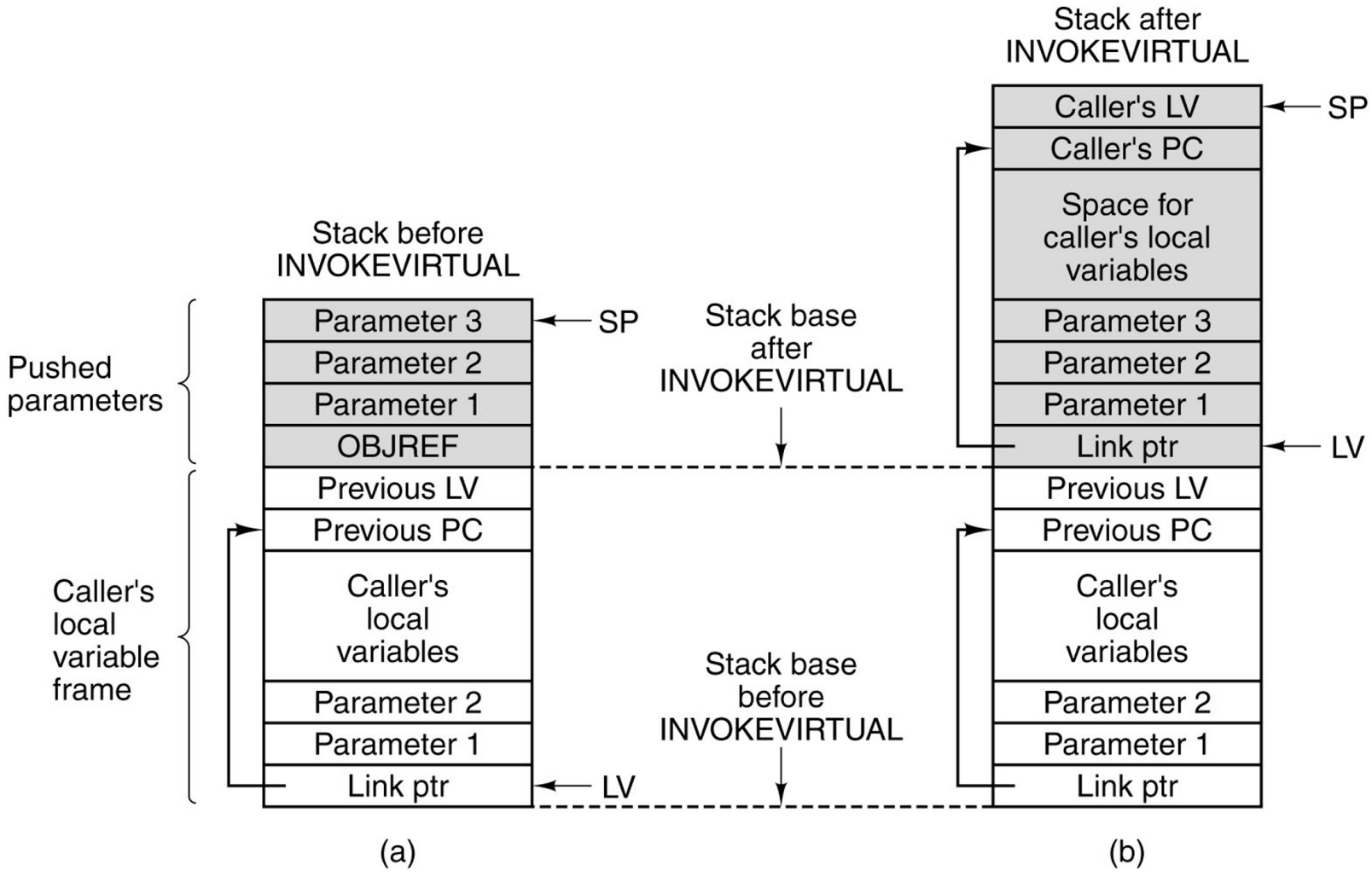
1. Constant Pool – read only
2. Local Variable Frame
3. Operand Stack (compiler guarantees that it will not exceed limits) (SP points to top of this stack)
4. Method Area Treated as a byte array (Holds program)

The IJVM Instruction Set

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

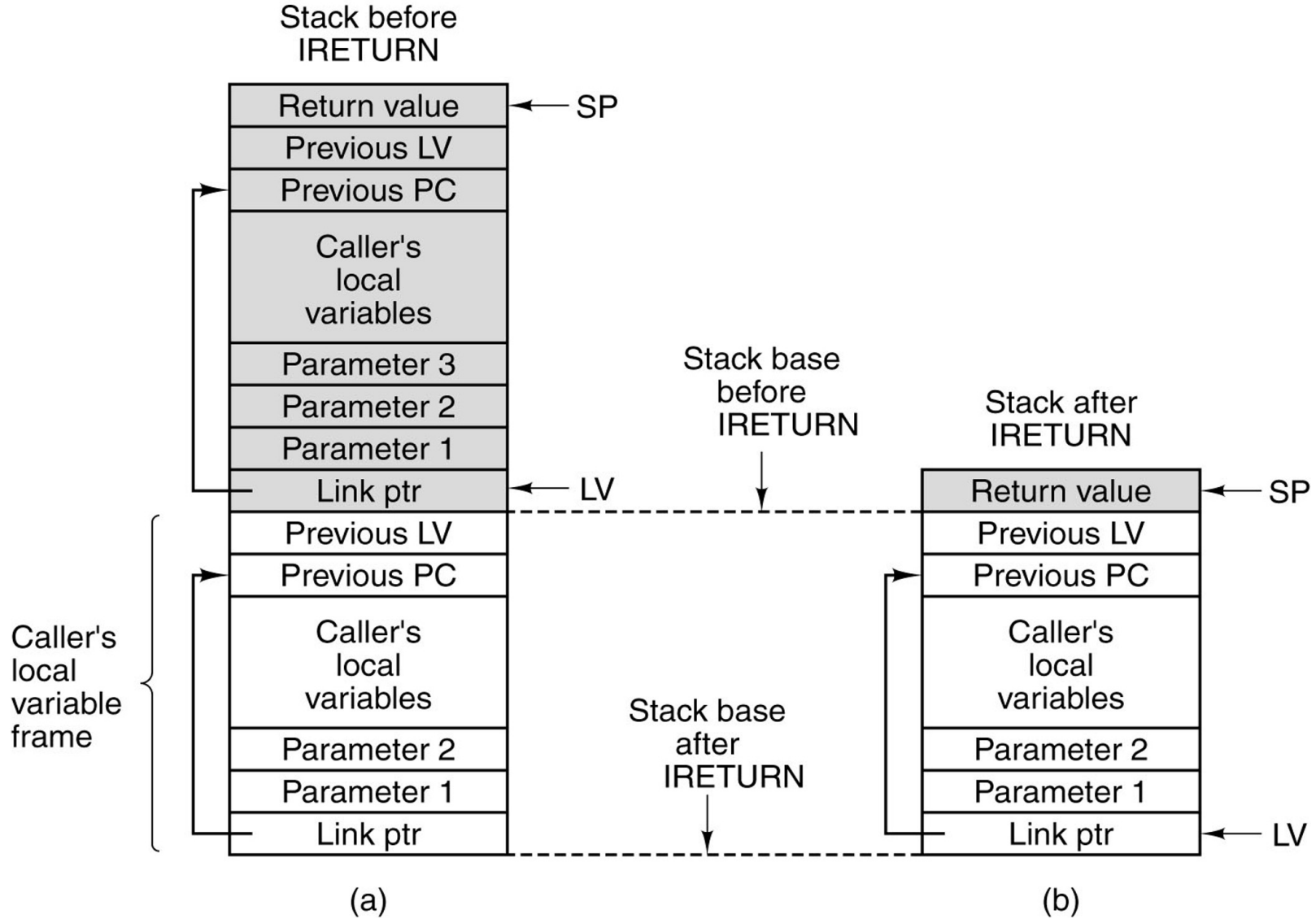
The IJVM instruction set. The operands byte, const, and varnum are 1 byte. The operands disp, index, and offset are 2 bytes.

The IJVM Instruction Set



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

The IJVM Instruction Set



(a) Memory before executing IRETURN. (b) After executing it.

Compiling Java to IJVM

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

(a)

```
 1        ILOAD j          // i = j + k
 2        ILOAD k
 3        IADD
 4        ISTORE i
 5        ILOAD i          // if (i == 3)
 6        BIPUSH 3
 7        IF_ICMPEQ L1
 8        ILOAD j          // j = j – 1
 9        BIPUSH 1
10        ISUB
11        ISTORE j
12        GOTO L2
13   L1:  BIPUSH 0         // k = 0
14        ISTORE k
15   L2:
```

(b)

```
0x15 0x02
0x15 0x03
0x60
0x36 0x01
0x15 0x01
0x10 0x03
0x9F 0x00 0x0D
0x15 0x02
0x10 0x01
0x64
0x36 0x02
0xA7 0x00 0x07
0x10 0x00
0x36 0x03
```

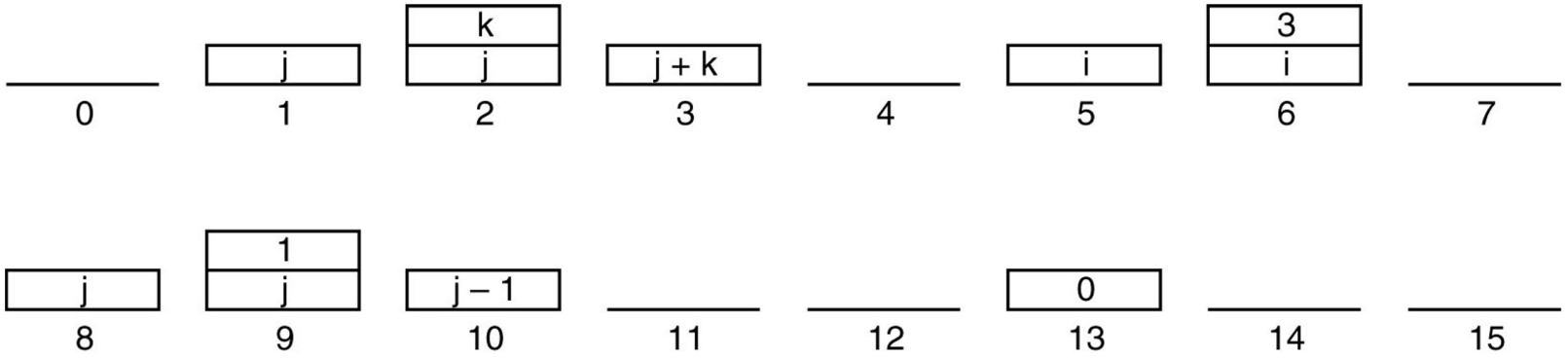
(c)

(a) A Java fragment.

(b) The corresponding Java assembly language.

(c) The IJVM program in hexadecimal.

The IJVM Instruction Set



The stack after each instruction

Microinstructions and notation

We also want to initiate a read operation, and we want the next instruction to be the one residing at location 122 in the control store. We might write

```
ReadRegister = SP, ALU = INC, WSP, Read, NEXT_ADDRESS = 122
```

where WSP means "write the SP register." This notation is complete but hard to understand. Instead we will combine the operations in a natural and intuitive way to capture the effect of what is happening:

```
SP = SP + 1; rd
```

Let us call our high-level Micro Assembly Language "MAL" (French for "sick," something you become if you have to write too much code in it). MAL is

Again Tanenbaum being funny

Microinstructions and notation

All permitted operations. 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 = $\overline{H}$
DEST = $\overline{SOURCE}$
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

Microinstructions and notation

$$MDR = SP + MDR$$

Why is this operation not possible?

Implementation of IJVM Using the Mic-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 (only 112 microinstructions total)

Implementation of IJVM Using the Mic-1

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 (only 112 microinstructions total)

Implementation of IJVM Using the Mic-1

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 (only 112 microinstructions total)

Implementation of IJVM Using the Mic-1

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

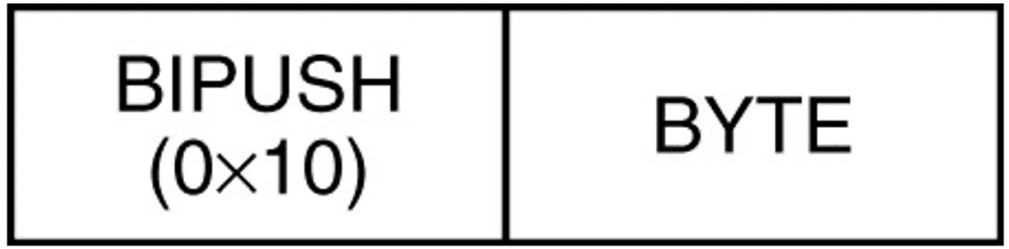
The microprogram for the Mic-1 (only 112 microinstructions total)

Implementation of IJVM Using the Mic-1

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

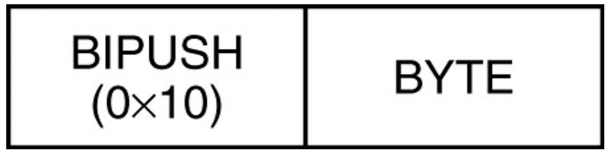
The microprogram for the Mic-1 (only 112 microinstructions total)

Implementation of IJVM Using the Mic-1



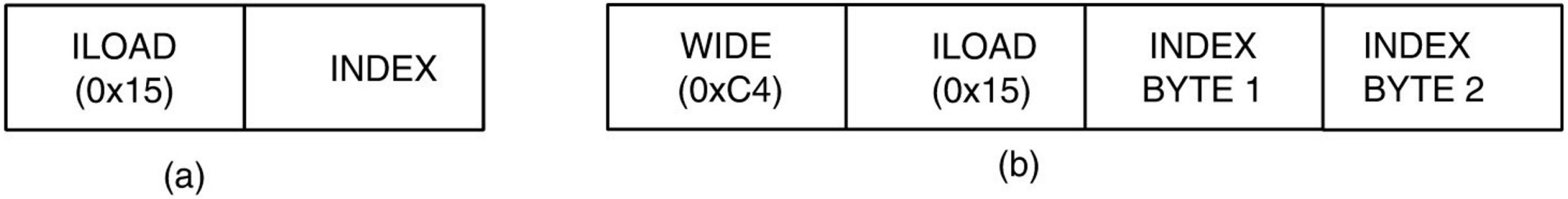
The BIPUSH instruction format

Implementation of IJVM Using the Mic-1



- The byte is to be interpreted as a signed integer (already fetched into MBR in Main1)
- Sign-extended to 32 bits and pushed onto the top of the stack.
- Sign-extend the byte in MBR to 32 bits, and copy it to MDR. Finally,
- SP is incremented and copied to MAR
- Written out to the top of stack and to TOS.
- Note: before returning to the main program, PC must be incremented and a fetch operation started

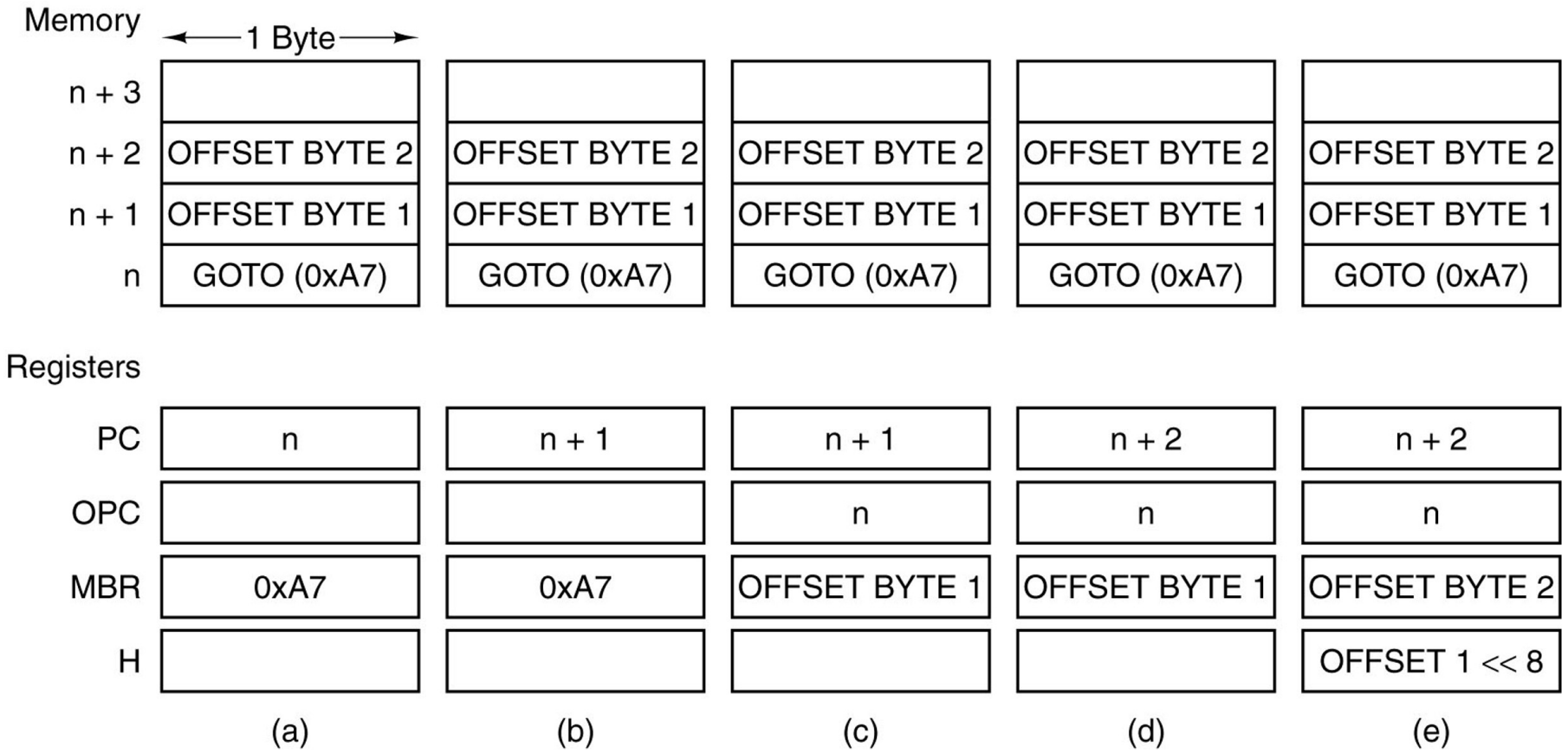
Implementation of IJVM Using the Mic-1



(a) ILOAD with a 1-byte index. (b) WIDE ILOAD with a 2-byte index.

Take a look at how to perform these two operations

Implementation of IJVM Using the Mic-1



The situation at the start of various microinstructions.

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

Speed vs cost

How to increase speed:

1. Reduce the number of clock cycles needed to execute an instruction.
2. Simplify the organization so that the clock cycle can be shorter.
3. Overlap the execution of instructions.
    - Path length
    - Potential of adding specialized hardware => Increased cost
    - Alternatives?

Merging the Interpreter Loop with the Microcode

Label	Operations	Comments
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
Main1	PC = PC + 1; fetch; goto (MBR)	MBR holds opcode; get next byte; dispatch

Original microprogram sequence for executing POP

Merging the Interpreter Loop with the Microcode

Label	Operations	Comments
pop1	MAR = SP = SP – 1; rd	Read in next-to-top word on stack
Main1.pop	PC = PC + 1; fetch	MBR holds opcode; fetch next byte
pop3	TOS = MDR; goto (MBR)	Copy new word to TOS; dispatch on opcode

Enhanced microprogram sequence for executing POP

A three bus architecture

Label	Operations	Comments
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

Mic-1 code for executing ILOAD

A three bus architecture

Label	Operations	Comments
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 index byte

Three-bus code for executing ILOAD

Other improvements

- Branch prediction
- Dynamic branch prediction
- Out-of-Order Execution and Register Renaming
- Speculative execution