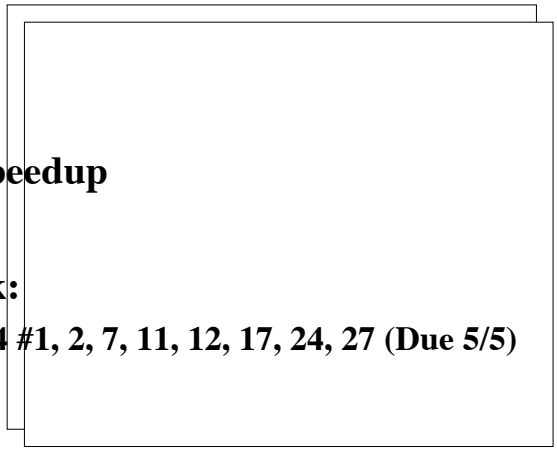
Chapter 4 - MicroArchitecture

- **Overview**
- **IJVM ISA**
- **Mic-1**
- **Mic 2-4**
- **Further Speedup**
- **Examples**
- **Homework:**
  - **Chapter 4 #1, 2, 7, 11, 12, 17, 24, 27 (Due 5/5)**

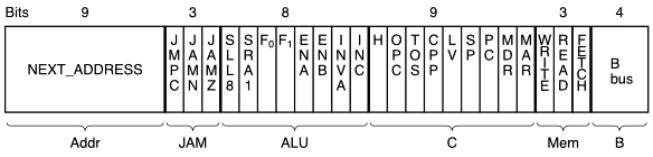


Chapter 3 - digital logic. We'll look at gates, basic digital logic, and boolean algebra. Then we'll see how these are used to build memory, cpu, and busses - the three core elements of a computer.

Homework: Here is the next

Basic flow of chapter: we will study in detail the implementation of a microarchitecture for the integer portion of the java virtual machine. Java virtual machine is an ISA that java compiler's produce code for. In most cases this machine doesn't actually exist, but rather it is one designed so that it can be efficiently supported by the actual ISA of whatever actual hardware is available.

The MicroArchitecture level



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

B bus registers

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

As computers got more complex, there was just too much stuff to try to go directly from digital logic to ISA, so an additional level was introduced: microarchitecture.

At digital logic level basic elements were boolean logic and a bit of low level detail for timing and storage.

At micro-arch level not really many principles, but some basic concepts appear (ie, new words:)

Datapath

Cache

Microprogram (sometimes)

Upper right shows the instruction format for micro-instructions. Narrow columns are single bit - often control lines that will go directly to cpu or datapath control inputs (notice F0, F1, ENA, ENB, INVA, INC)

Lower left shows instruction format for IJVM.

Our task is to design a microprogrammed cpu, AND microprogram, that can run the IJVM instruction set at the ISA level.

Java -> IJVM

```
i = j + k;              1    ILOAD j    // i = j + k      0x15 0x02
if (i == 3)             2    ILOAD k                   0x15 0x03
    k = 0;              3    IADD                      0x60
else                    4    ISTORE i                  0x36 0x01
   j = j – 1;           5    ILOAD i    // if (i < 3)    0x15 0x01
                        6    BIPUSH 3                  0x10 0x03
                        7    IF_ICMPEQ L1              0x9F 0x00 0x0D
                        8    ILOAD j    // j = j – 1     0x15 0x02
                        9    BIPUSH 1                  0x10 0x01
                       10    ISUB                      0x64
                       11    ISTORE j                  0x36 0x02
                       12    GOTO L2                   0xA7 0x00 0x07
                       13 L1:          BIPUSH 0        // k = 0 0x10 0x00
                       14    ISTORE k                  0x36 0x03
                       15 L2:
```

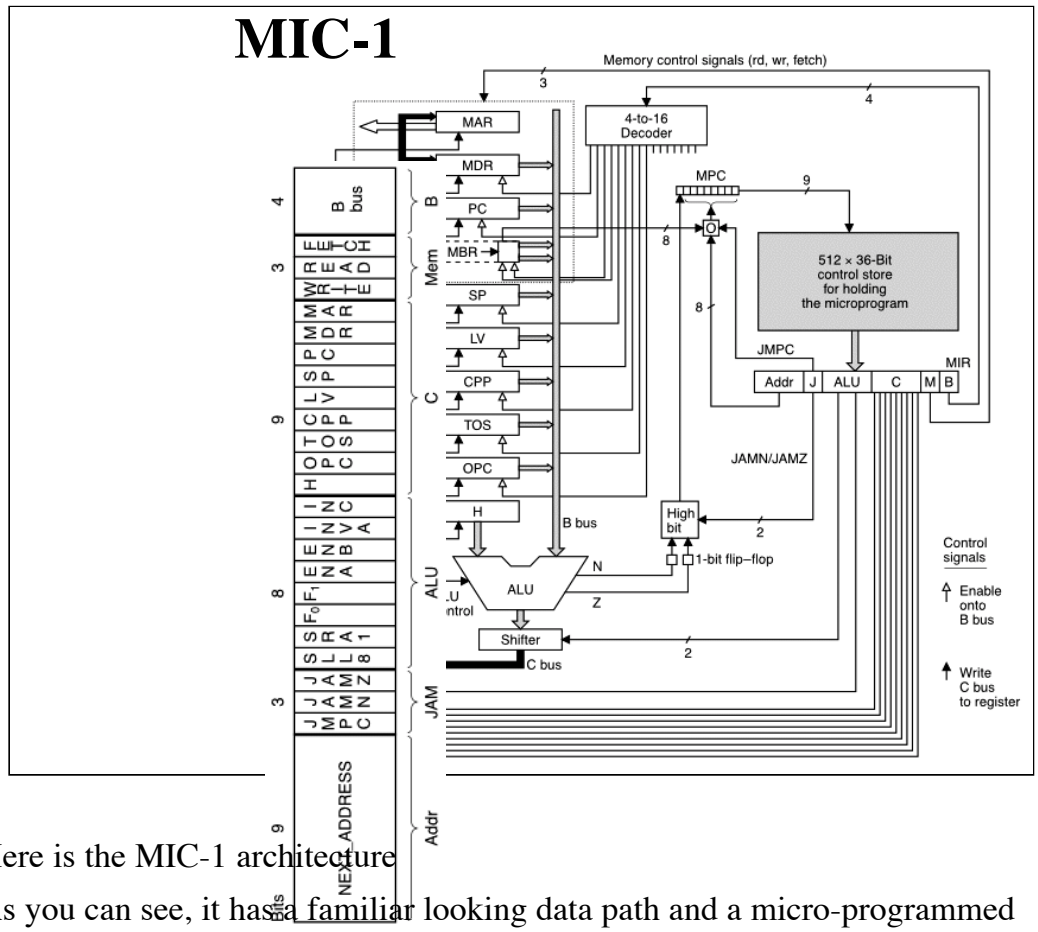
The java compiler translates java code into IJVM instructions

Above is a small sample piece of Java and the corresponding IJVM, in IJVM symbolic assembly language and in hex.

Look at first java line, and first four IJVM lines: I=j+k translates to load I, load j, add, store.

Now remember: IJVM is compiler output, the ISA level our microcode has to support.

This is just eye candy for now, we will actually understand the IJVM by the end of the chapter.



Here is the MIC-1 architecture

As you can see, it has a familiar looking data path and a micro-programmed control.

Let's look at instruction word again, in context.

Left part of word goes to micro program counter (every instruction is a branch!)

Next is Jump control, we'll be talking more about that.

Then shift control bits (note picture makes it look like they are to the right of alu control, actually to left

Then ALU control

Then lines for which register to load from C bus

Then lines for main memory control

Finally, b bus address. Why is B bus adress encoded in word, but C bus not?

MIC-1 datapath

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	$\bar{A}$
1	0	1	1	0	0	$\bar{B}$
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	1	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
0	1	0	0	0	1	1
0	1	0	0	1	0	–1

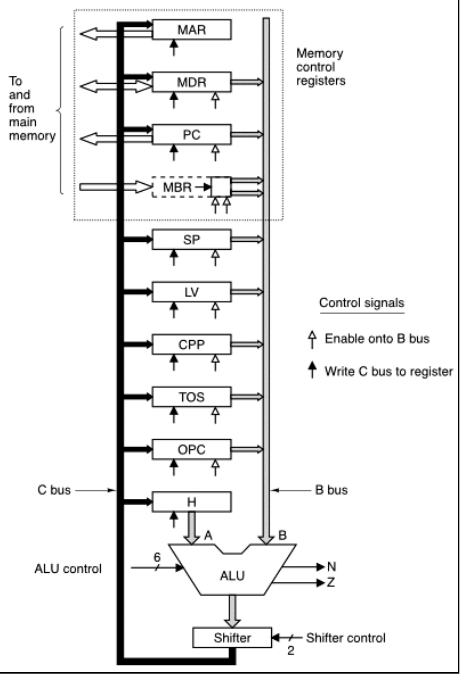


Figure 4-2. Useful combinations of ALU signals and the func

The MIC-1 ALU is constructed out of the alu bit slices we saw last chapter.

At left is an interpretation of the control line inputs to the ALU, in terms of what they cause the alu to do.

Note that B input comes directly from B bus (so, whatever register is instructed to write to B bus, whereas A input of ALU comes from special register H.

That means to add to registers together, we will first have to move contents of one to H.

What are control lines?

6 ALU control lines

9 “write from Cbus” lines, one for each reg that can store data from ALU

8-9 “write to B Bus” lines, one for each reg that can put data on B bus

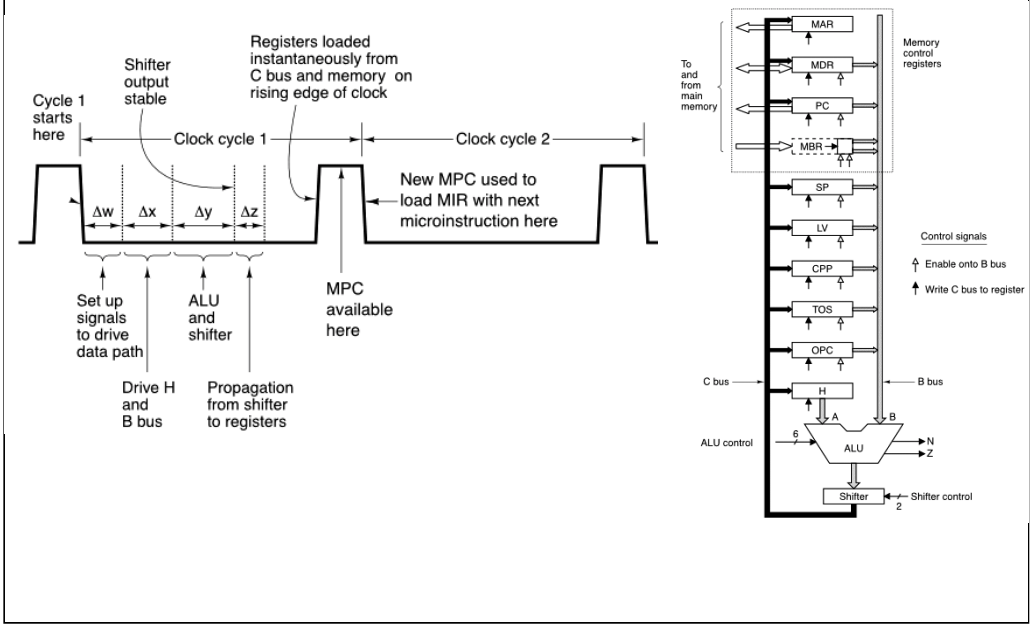
2 Shift control lines

3 memory (cpu external bus) lines.

These are what microprogram must control.

Do probs 3, 4!!!!

DataPath Timing



We can both read AND write a register in one cycle

This is possible, even though there are no storage elements in datapath, because of delays.

Example: add 1 to PC. Put PC on bus in x, store back into PC during z.

Memory: Note the MAR, MDR, PC, MBR registers on the datapath.

MAR contains WORD addresses.

PC contains BYTE addresses.

So, putting 2in MAR loads bytes 8-11 into MDR

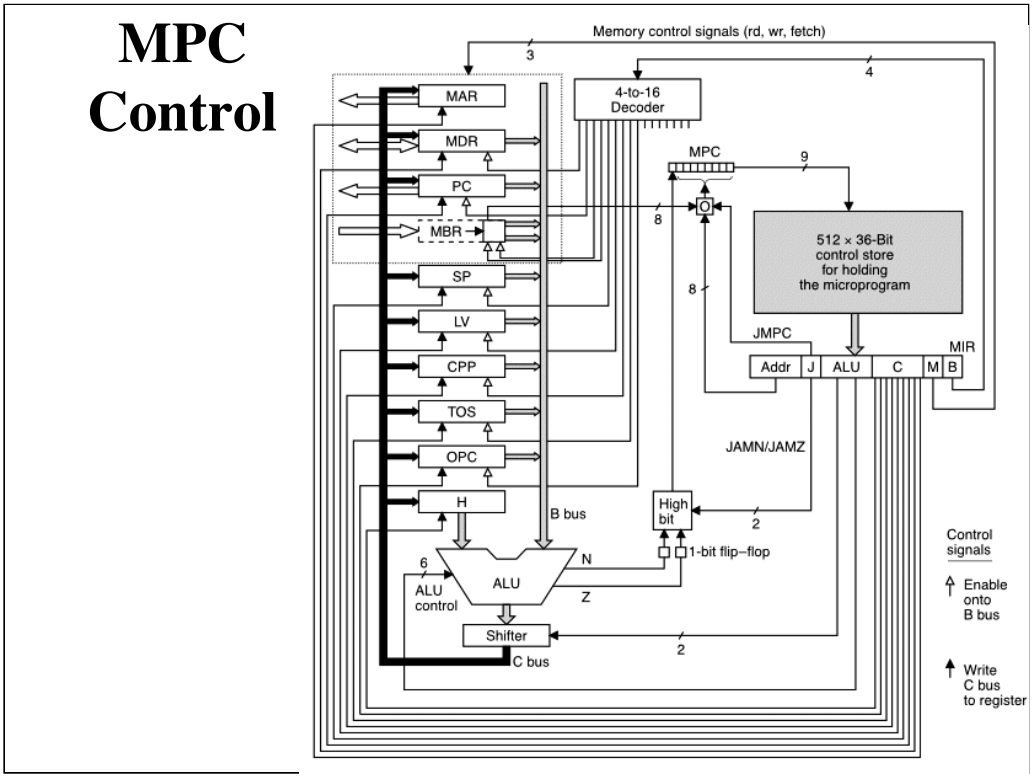
Putting 2 in PC loads byte 8 into MBR - How? Just tack two zeros on the end of MAR addresses before sending them to memory!

MAR/MDR reads/writes data

PC/MBR reads instructions

MDR/MBR data are available one cycle FOLLOWING address loads. In the meantime, old values can be assumed to persist.

MPC Control



ISA's assume sequential execution.

Microarchitectures rarely do.

MIC-1 takes next address from the current instruction, then:

1. Or's the high order bit with ALU Z or N output, if microinstruction JAM 0 or 1 is true

2. ALU N? ALU Z? Depending on the result of the function, the ALU outputs two bits, called N and Z. N is a one when the result is negative (i.e. the left-most bit is a one), and Z is a one when the result is zero

3. Or's the lower eight bits with MBR, if JMPC is true. This latter is for ISA instruction decode:
4. Remember MBR will hold IJVM instruction opcodes. So, this is a way to jump to a specific routine in micromemory depending on the ISA opcode to be executed. Instruction decode in 1 microcycle! Pretty tricky, huh?
5. Do Prob 5 at chap. end

MPC logic

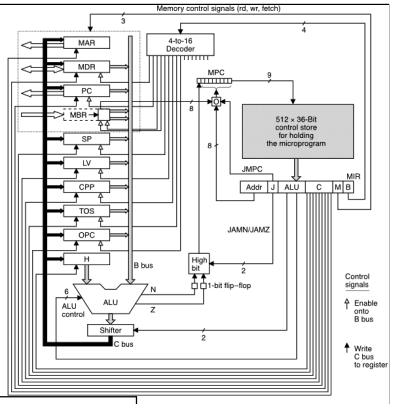
Instruction decode:

```
if (instr == x1)
        goto y1;
else if (instr == x2)
        goto y2;
else …
```

Goto y2 == MPC = y2

What if x1 == y1
x2 == y2
…
?
Then just:

MPC = instr;



- **if (!JMPC) {**
  - **MPC_{0-7} = $ADDR_{0-7}$**
  - **MPC_8 = (JAMZ & Z) + (JAMN & N)**
- **}**
- **else {**
  - **MPC = MBR**
  - **MPC_8 = (JAMZ & Z) + (JAMN & N)**
- **}**

MicroArch:
PC = MPC
IR = MIR

IJVM:
PC = OPC
IR = MBR

I want to spend a bit more time on the microprogram address logic, as this is crucial to understanding this chapter and the microcode we will study later.

What's the problem? Instruction decode. In general, in a microprogrammed machine we expect to see "software" for each part of instruction execution. Remember we talked earlier about fetch, decode, op-fetch, execute, result-store. Well, we are focusing, for the moment, on decode.

Remember also that most cpus' will have at least two dedicated registers: the program counter and the instruction register. Since we are dealing with two cpus (!), we shold expect to see two sets of these.

MPC and MIR are the micro-level program counter and instruction register.

MAR and MBR are the IJVM-level program counter and instruction register.

So here is the problem. Suppose MBR holds a 0x36 (WHAT IS this? 36 hex = 54 decimal = 00110110B) How do we "decode" this? Decode means figure out what to do (not actually do it, yet).

In the quiz we did this via an if statement: if (opcode == xx) {…} else if (opcode == yy) {…} else …

This is slow. Even if the opcode is at the front of the list, as we will see, doing a compare to a constant is going to take several microinstructions and [illegible]

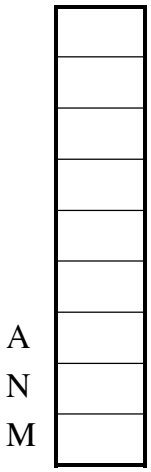
IJVM

- **Stack Architecture**
- **Memory Model**
- **The ISA**
  - **Basic instructions**
  - **Procedure invocation**

IJVM - a stack architecture

```
Public int funnyFunc(int m, int n) {
  int a;
  if (n == 0)
     return 1;
  else
     a = funnyFunc(m, n-1);
     return m*a + n;
}

mPowerN(3, 2);  ?
 3*funnyFunc(3,1)+2
 3*(3*funnyFunc(3,0)+1)+2
 3*(3*1+1)+2
 14
```



Stacks: What is a stack? Like a stack of dishes: you put things on the top, you take things off the top.

Two uses for stacks: (1) local variables during procedure calls. (2) arithmetic operations.

Local variables: methods can refer to local variables. Consider:

```
Public int funnyFunc(int m, int n) {
  int a;
  if (n == 0)
     return 1;
  else
     a = funnyFunc(m, n-1);
     return m*a + n;
 }
```

First idea: assign every local variable a fixed location in memory.

Try executing above: problem -

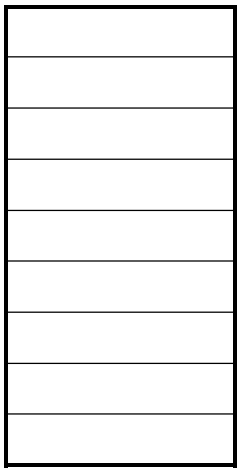
first by hand - right answer is (((3*1)+1)*3)+2 = 14

next with fixed addrs for m, n, a

note that n gets overridden and you get wrong answer!

IJVM - a stack architecture II

m*a + n;
3 *4 + 2



Load m
Load a
Multiply
Load n
Add

How do we do this?

Stack execution model of operand management

Load m

Load a

Multiply

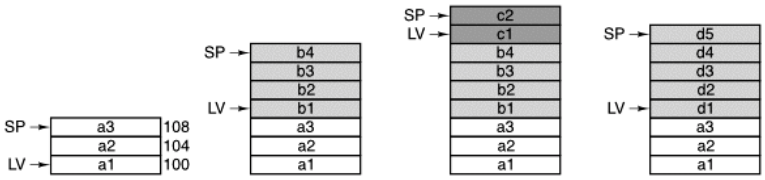
Load n

Add

Notice t his is different problem than we addressed earlier, but same basic solution.

Can we combine?

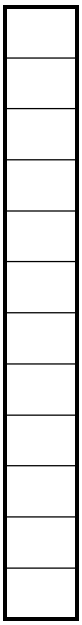
IJVM Stack Architecture III




```
Public int funnyFunc(int m, int n) {
  int a;
  if (n == 0)
     return 1;
  else
     a = funnyFunc(m, n-1);
     return m*a + n;
}
```

m*a + n;
3 *4 + 2

Load m
Load a
Multiply
Load n
Add



LV is base address for local variables in current method

SP is base addrress of next free entry in stack - 1

Step through funnyFunc one last time.

FunnyFunc (3,2) - LV at bottom, three for local vars,

then call to funnyFunc(3,1)

again three for local vars, then call to funnyFunc(3, 0)

again three for local vars, then return 1 (set a to 1)

back at (3,1) - push M on stack, push A on stack, *, push n, add

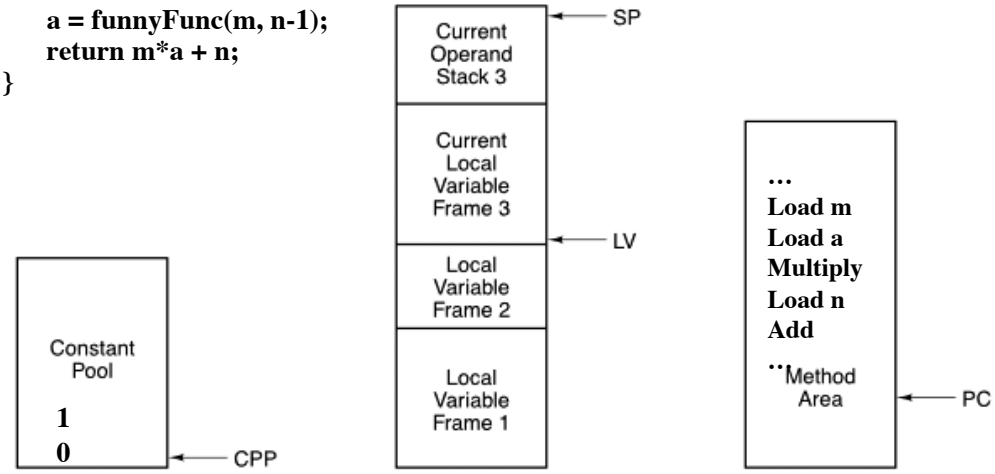
back at (3, 2) - set a to result

push M, pushA, *, push N, add.

return

IJVM Memory Model

```
Public int funnyFunc(int m, int n) {
  int a;
  if (n == 0)
     return 1;
  else
     a = funnyFunc(m, n-1);
     return m*a + n;
}
```



Ok, so we have seen that local variables are not referenced as absolute addresses to main memory, but rather as offsets from a current LV base address managed by the IJVM ISA.

There are two other base addresses:

(1) CPP is the base address for all the constants in the program. A separate base address protects them from being modified, one standard way to cause "unintended" consequences (ie, either a bug or a hack).

(2) PC is the base address for code, another chunk of stuff that (usually) shouldn't be modified, except, in Java's case, by dynamically loading classes at run-time.

Java -> IJVM Example

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

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

```
    ILOAD j    // i = j + k
    ILOAD k
    IADD
    ISTORE i
    ILOAD i    // if (i < 3)
    BIPUSH 3
    IF_ICMPEQ L1
    ILOAD j    // j = j – 1
    BIPUSH 1
    ISUB
    ISTORE j
    GOTO L2
L1: BIPUSH 0
    ISTORE k
L2:
```

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

We now know enough to actually map some java code to real IJVM instructions.

Top left is java

Below is IJVM assembler

Top right is actual hex for the first few insructions.

Note loads and stores as expected.

Note upper right: j is local var 2, k is local var 3, I is local var 1

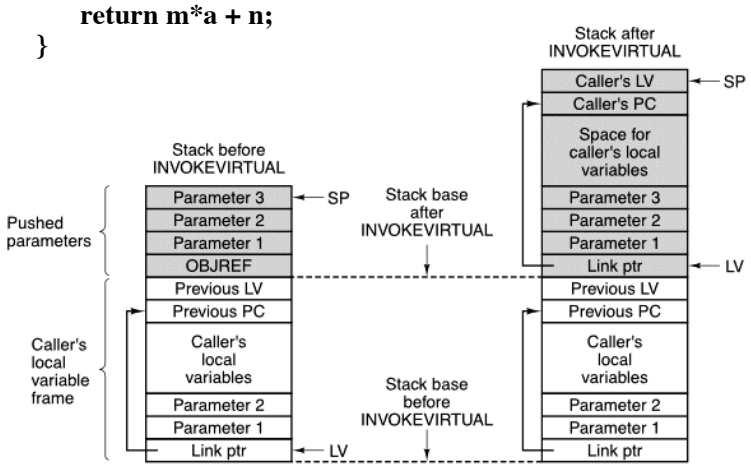
Note also BIPUSH - constant is loaded directly from instruction, rather than constant area.

Why - faster - why load address of data when you can load data directly.

Constant area used for longer things like character strings.

IJVM Procedure Invocation

```
Public int funnyFunc(int m, int n) {
  int a;
  if (n == 0)
     return 1;
  else
     a = funnyFunc(m, n-1);
     return m*a + n;
}
```




```
0x0002 // parms
0X0001 // locals
BIPUSH 0
ILOAD N
IF_ICMPEQ  L2
BIPUSH 1
RETURN
L2: BIPUSH 0
ILOAD M
ILOAD N
BIPUSH 1
ISUB
INVOKEVIRTUAL
ILOAD M
IMULT
PUSH N
IADD
ISTORE 0
RETURN
```

Simplified calling mechanism - essentially C or Pascal.

At right above is possible IJVM ISA for our funnyFunc

Note a function needs to say how many parameters and how many locals it has, so machine can set up stack appropriately.

Below is how IJVM expects stack

Note this is different from what we talked about earlier. A little more complicated - why?

1 need to store previous LV & PC so we know how to restore things when we return.

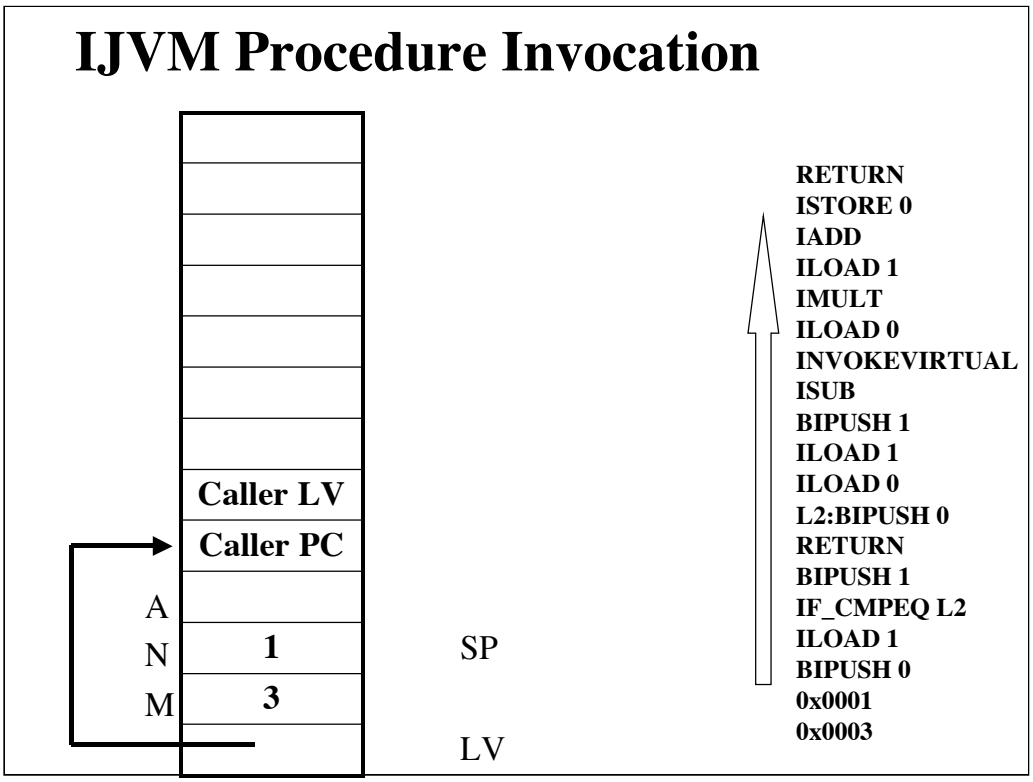
2 “ObjRef - don’t really need that. However, we will use it to point to caller PC and caller LV so we can restore these on return. In this case we set it to Zero at L2

3. Now put parameters on stack. Note we have N, but need to put N-1 on stack.

Now call INVOKEVIRTUAL

4 INVOKEVIRATUAL - 2 parms tells INVOKEVIRTUAL how far back to go to find OBJREF.

4. 1 local tells INVOKEVIRTUAL how far to skip before storing Caller’s PC



On entry, LV points to"Link Ptr",in the stack area of main memory, SP points to A (the current top of the stack). M and N were set by the caller, we'll see when we get there.

If (N==0) goto L2

BIPUSH 0 pushes 0 on the stack (update SP)

ILOAD 1 pushes the value of N on the stack - 1 is the offset of N from LV - ILOAD always refers to offsets from LV

IP_CMPEQ compares the two top stack entries AND pops them off the stack!

We branch first time

BIPUSH 0 pushes 0 on the stack.

ILOAD 0 pushes m on the stack

ILOAD 1 pushes n on th estack

BIPUSH 1 pushes 1 on the stack

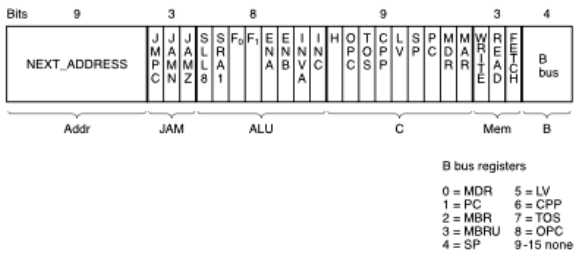
ISUB subtracts the top stack entry form the next one down (pops both), and pushes the result back on the stack

Hey - cool - we are now set for the INVOKEVIRTUAL! What does it do?

1. Look at code, and push # of locals on stack
2. Save PC and LV on stack
3. SET LV to current SP - 2 - #parms - #locals
4. Set PC to method offset + 4.

MAL

- **SP = SP+1; rd**
  - **B=0100 (4)**
  - **Mem = 010 (rd)**
  - **C=000001000 (SP)**
  - **ALU = 00110101 (F0+F1+ENB+INC)**
  - **JAM = 000 (0)**
  - **ADDR = ?**



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	$\bar{A}$
1	0	1	1	0	0	$\bar{B}$
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	1	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
0	1	0	0	0	1	1
0	1	0	0	1	0	–1

We are ready to write/examine microcode. But, don't want to write binary. So, let's invent a notation that is more mnemonic.

Suppose we want to increment the value of the SP register, initiate a read from main memory, and have a next instr at loc 122.

SP = SP+1; rd

Huh? Is that all? How could that work, what does that have to do with the 36 bit microinstruction?

How indeed. Good question.

B = 4

JAM = 0

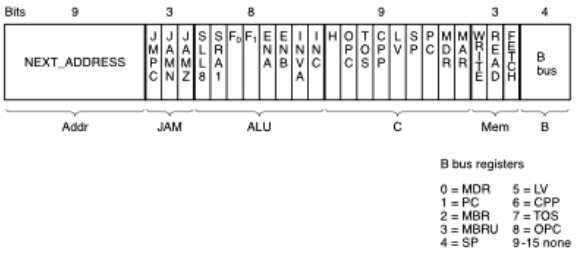
ALU = F0+F1+ENB+INC (What do I mean by this? See 4-2)

What about addr? We'll let assembler decide that. It can actually put the next instruction anywhere, as long as it sets the addr field right.

Why would it want to put it somewhere strange? Because of the way JMPC, JAMZ, JAMN work. For example, POP and DUP are only 2 apart (0x57, 0x59)

MAL

- **MDR = SP**
  - **B=0100 (SP)**
  - **Mem = 000 (no operation)**
  - **C=000000010 (MDR)**
  - **ALU = 00110100 (F0+F1+ENB+INC)**
  - **JAM = 000 (don't jump)**
  - **ADDR = ?**

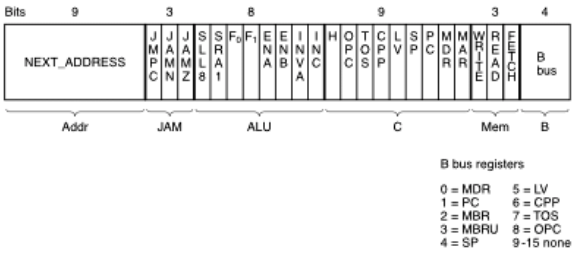


F_0	F_1	ENA	ENB	INVA	INC	Function
0	1	1	0	0	0	A
0	1	0	1	0	0	B
0	1	1	0	1	0	$\bar{A}$
1	0	1	1	0	0	$\bar{B}$
1	1	1	1	0	0	A + B
1	1	1	1	0	1	A + B + 1
1	1	1	0	0	1	A + 1
1	1	0	1	0	1	B + 1
1	1	1	1	1	1	B – A
1	1	0	1	1	1	B – 1
1	1	1	0	1	1	–A
0	0	1	1	0	0	A AND B
0	1	1	1	0	0	A OR B
0	1	0	0	0	0	0
0	1	0	0	0	1	1
0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

Another simple MAL statement and its binary.

MAL

- **MDR = H+SP**
  - **B=0100 (SP)**
  - **Mem = 000 (no operation)**
  - **C=000000010 (MDR)**
  - **ALU = 00111100 (F0+F1+ENA+ENB)**
  - **JAM = 000 (don't jump)**
  - **ADDR = ?**

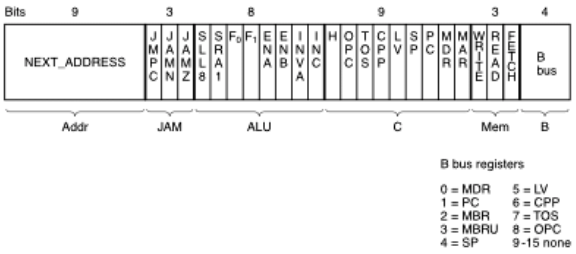


F_0	F_1	ENA	ENB	INVA	INC	Function
0	1	1	0	0	0	A
0	1	0	1	0	0	B
0	1	1	0	1	0	$\overline{A}$
1	0	1	1	0	0	$\overline{B}$
1	1	1	1	0	0	A + B
1	1	1	1	0	1	A + B + 1
1	1	1	0	0	1	A + 1
1	1	0	1	0	1	B + 1
1	1	1	1	1	1	B – A
1	1	0	1	1	1	B – 1
1	1	1	0	1	1	–A
0	0	1	1	0	0	A AND B
0	1	1	1	0	0	A OR B
0	1	0	0	0	0	0
0	1	0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

Note that H is ALWAYS the A input to the ALU. It can be disabled or inverted, but no other register can be A input. Why not? (Cost, of course).

MAL

- **MDR = MDR+SP?**
  - **B=0100 (SP)**
  - **Mem = 000 (no operation)**
  - **C=000000010 (MDR)**
  - **ALU = 00111100 (F0+F1+ENA+ENB) ???**
  - **JAM = 000 (don't jump)**
  - **ADDR = ?**



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	$\bar{A}$
1	0	1	1	0	0	$\bar{B}$
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	1	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
0	1	0	0	0	1	1
0	1	0	0	1	0	–1

This seems reasonable as MAL symbolic code, but we can't generate a microinstruction for it. Why NOT? Because one input to ALU is NOT selectable, it is always H.

We could use TWO microinstructions: one to move MDR (or SP) to H, the second to add and store.

H = H-MDR is similarly illegal. Can only do H as the subtrahend (note table in lower right - all we have is B-A, not A-B, and H is A input to ALU).

Legal arithmetic ops

- **Source, dest, can be:**
  - **MAR**
  - **MDR**
  - **PC**
  - **MBR**
  - **SP**
  - **LV**
  - **TOS**
  - **OPC**
- **Dest can also be H**

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

Branches

- **If (Z) goto L1; else goto L2**
  - **Sets JAMZ bit**
- **If (N)…**
  - **Sets JAMN bit**
- **goto (MBR or *value*)**
  - **Sets JMPC bit**
- **Note L1 and L2 must be 256 bytes apart**
  - **Assembler problem**
- **goto (MBR); fetch**

Note final goto will not be affected by rd, it takes several (3) microcycles for data to show up.

Mic-1 Microcode - Main Loop

```
Quiz:
instr = program[PC];
PC = PC+1;
execute(instr);

How about:
PC = PC+1;
nextInstr = program[PC];
execute(instr);
```

- **Main Loop Microcode**
  - **B=0001 (PC)**
  - **Mem = 001 (fetch)**
  - **C=000000101 (PC+MAR)**
  - **ALU = 00110101 (F0+F1+ENB+INC)**
  - **JAM = 100 (JMPC)**
  - **ADDR = 00000000**

What does main loop have to do?

Increment PC

Fetch instruction

Decode.

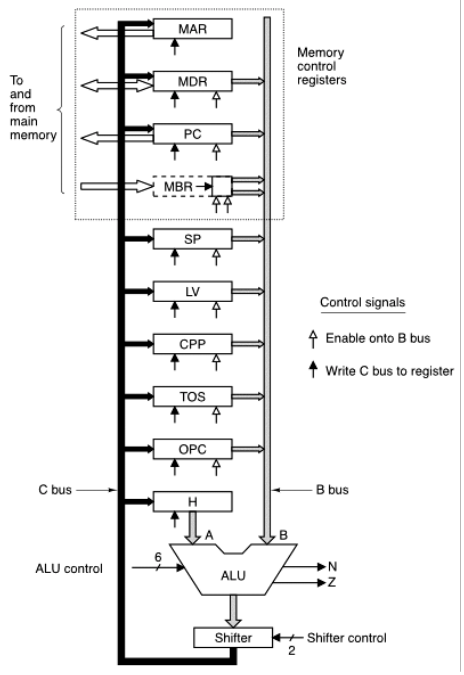
Let's assume we already have current instruction. Remember it takes a while to get an instruction.

So how about: increment PC to point to next instruction; start fetch of that instruction, and decode

All in ONE microinstruction!

ILOAD 0

H = LV
MAR = MBRU+H; rd
MAR = SP = SP + 1
PC = PC + 1; fetch; wr
TOS = MDR; goto Main1



First instruction moves LV into H

Next sets MAR to LV + MBRU and initiates a read.

MBRU?

1. Remember we started a load of the next byte after the opcode in main, so MBR now has the offset we need (e.g., ILOAD 0).

2. MBRU means expand MBR to 32 bits with high order zeros.

3 Updates MAR and SP (why? Because we are adding a new entry on the stack)

4 Update PC to point past the offset, and start a fetch (when we go back to main loop, need to have opcode in MBR!!!) Also start a write - this writes contents of MBR to MAR addr. This is confusing, track carefully!)

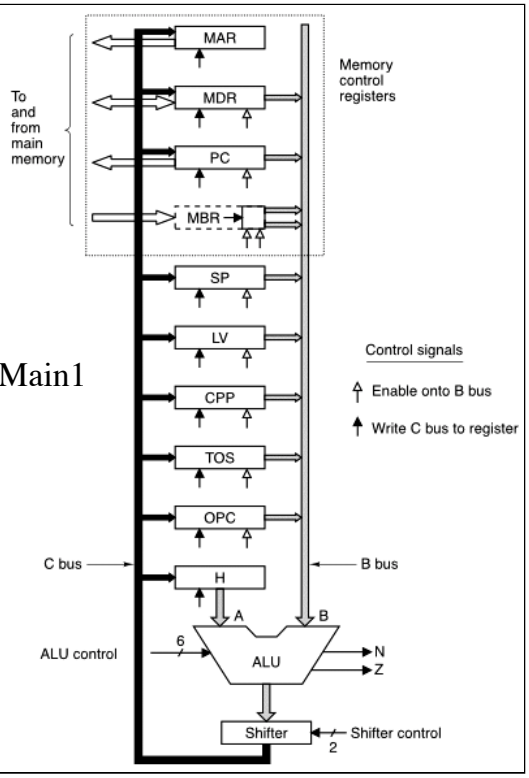
5. Move MDR (value we just loaded) to TOS register, which always holds the top entry in the stack, to save having to load it when needed, and go back to main loop

Good quiz question - trace operation of an instruction like this, or implement a new instruction.

ISUB

MAR = SP = SP-1; rd
H = TOS
MDR = TOS = MDR - H; wr; goto Main1

1
4



Pretty simple, huh? Now we see reason for TOS - this would require two reads, which would require an additional two microcyles, without TOS